ĐẠI HỌC QUỐC GIA TP.HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA ĐIỆN – ĐIỆN TỬ
BỘ MÔN ĐIỆN TỬ
----------o0o----------

BÁO CÁO BÀI TẬP LỚN CẤU TRÚC MÁY TÍNH

THIẾT KẾ CPU RISC-V PIPELINE

GVHD: **TS. Trần Hoàng Linh**

SVTH: **MSSV:**

Nguyễn Hoàng Bách **1710566**

TP. HỒ CHÍ MINH, THÁNG 7 NĂM 2020

MỤC LỤC

I. GIỚI THIỆU VỀ CPU RV32 3
1. Tổng quát 3
2. Các tập lệnh 4
a. Nhóm lệnh R-format 4
b. Nhóm lệnh I (Load Data) 5
c. Nhóm lệnh I (Tính toán) 5
d. Nhóm lệnh S (Store data) 5
e. Nhóm lệnh B (rẽ nhánh) 6
f. Nhóm lệnh U 6
g. Nhóm lệnh J (nhảy không điều kiện) 7
II. CẤU TRÚC RISC-V PIPELINE 7
1. Các trạng thái xử lý câu lệnh trong cấu trúc CPU Pipeline 7
2. Hazard trong pipeline 8
a. Structural Hazard 8
b. Data Hazard 9
c. Control hazard / Branch hazard 10
3. Tín hiệu điều khiển trong pipeline 11
III. THIẾT KẾ CPU RISC-V PIPELINE 12
1. Instruction Fetch 12
a. PC 12
b. PC_add4: 13
c. IMEM 13
d. Prediction 13
e. Mux_predict 15
f. Thanh ghi IF_to_ID 15
2. Instruction Decode 17
a. Regs 17

b. Imm_gen 18
c. Branch_comp 18
d. Controller 18
e. Mux_hazard.....2
f. Hazard_detection.....2
g. Thanh ghi ID_to_EX.....3
3. Execute.....5
a. Mux_A 5
b. Mux_B.....6
c. Addsum 6
d. Alu.....7
e. MuxBB.....7
f. Forwarding_unit.....7
g. Thanh ghi EX_to_MEM: 8
4. Memory Access.....10
a. Khối DMEM: 10
b. Thanh ghi MEM_to_WB 11
5. Write Back 12
Khối MuxW: 12
IV. PHỤ LỤC.....13
1. Tập thanh ghi Regs.....13
2. Khối Branch_comp 17
3. Khối Controller 21
4. Alu.....28

I. GIỚI THIỆU VỀ CPU RV32

1. Tổng quát

- CPU RV32 có tổng cộng 32 lệnh hợp ngữ, trong đó mỗi lệnh có độ dài 32bit và có 7 bit [6:0] (opcode) để xác định loại lệnh.

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

- Tập lệnh của RV32 còn được gọi là tập lệnh kiểu load-store, điều đó có nghĩa là data trong bộ nhớ muốn được thực thi thì trước hết phải được lấy ra bỏ vào băng thanh ghi rồi mới được tính toán. Sau khi tính toán, data sẽ được lưu lại vào memory.

- Các thanh ghi trong băng thanh ghi (Register Bank) có độ dài 32 bits và có 32 thanh
  ghi (từ $x_0 - x31 \Longrightarrow$ Cần có 5 bits để xác định địa chỉ của các thanh ghi trong băng thanh ghi. Trong đó, chức năng của 32 thanh ghi được cho như ở bảng dưới.

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

- Dữ liệu trong cả bộ nhớ dữ liệu (DMEM) và bộ nhớ chương trình (IMEM) đều có độ dài 32bit và được sắp xếp theo kiểu ***little edian.***
  DMEM được định địa chỉ theo từng byte (= 8 bits) chứ không theo word (= 32 bits). Nếu định địa chỉ theo word thì lấy địa chỉ của byte có trọng số thấp nhất

2. Các tập lệnh

a. Nhóm lệnh R-format

0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

- Nhóm lệnh này bao gồm các lệnh có cấu trúc như ở hình sau
- Nhóm lệnh này có opcode là [6:0] = 0110011

- Nhóm lệnh này thực hiện lấy hai giá trị lưu ở thanh ghi rs1 và rs2 thực hiện đưa vào khối ALU để tính toán, sau đó lưu kết quả vào thanh ghi rd.

b. Nhóm lệnh I (Load Data)

imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	010	rd	0000011	lh
imm[11:0]	rs1	011	rd	0000011	lw
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	110	rd	0000011	lhu

- Nhóm lệnh này có opcode là [6:0] = 0000011
- Nhóm lệnh này thực hiện lấy hai giá trị lưu ở thanh ghi ***rs1*** và giá trị lưu ở ***imm[11:0]*** (được mở rộng dấu) để tính tổng ***rs1 + ext(imm[11:0])***. Sau đó lấy giá trị lưu trong DMEM tại địa chỉ ***rs1 + ext(imm[11:0])***, lưu vào thanh ghi ***rd***.

c. Nhóm lệnh I (Tính toán)

imm[11:0]		rs1	000	rd	0010011	addi
imm[11:0]		rs1	010	rd	0010011	slti
imm[11:0]		rs1	011	rd	0010011	sltiu
imm[11:0]		rs1	100	rd	0010011	xori
imm[11:0]		rs1	110	rd	0010011	ori
imm[11:0]		rs1	111	rd	0010011	andi
0000000	shamt	rs1	001	rd	0010011	slli
0000000	shamt	rs1	101	rd	0010011	srli
0100000	shamt	rs1	101	rd	0010011	srai

- Nhóm lệnh này có opcode là [6:0] = 0010011
- Nhóm lệnh này (trừ 3 lệnh SRAI, SRLI, SLLI) thực hiện lấy giá trị lưu ở thanh ghi ***rs1*** và giá trị lưu ở ***imm[11:0]*** (được mở rộng dấu), thực hiện đưa vào khối ALU để tính toán. Kết quả được lưu vào thanh ghi ***rd***.

d. Nhóm lệnh S (Store data)

Imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	sb
Imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	sh
Imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	sw

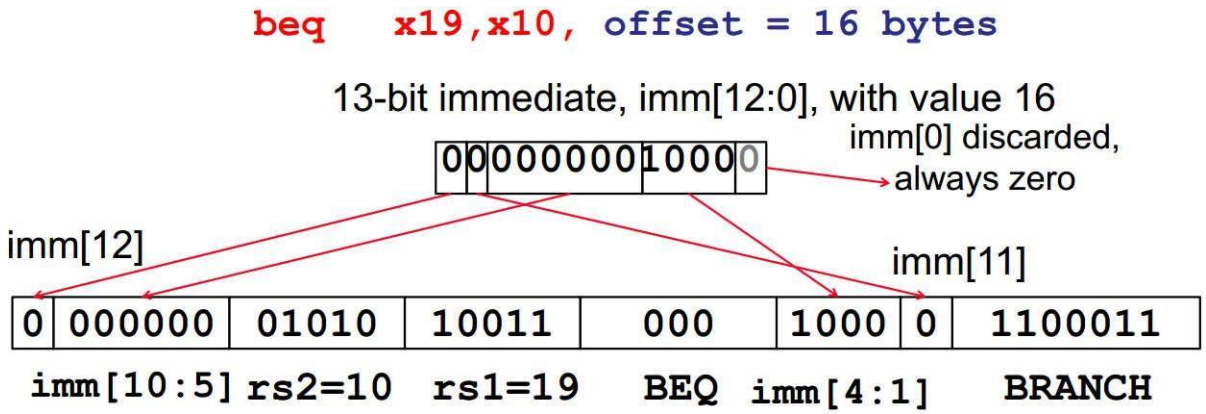
- Nhóm lệnh này có opcode là [6:0] = 0100011
- Nhóm lệnh này thực hiện lấy hai giá trị lưu ở thanh ghi ***rs1*** và giá trị lưu ở ***imm[11:5]*** và ***imm[4:0]*** (ghép lại và mở rộng dấu) để tính tổng ***rs1 +***

ext(imm[11:5]imm[4:0]). Sau đó lấy giá trị lưu trong thanh ghi ***rs2*** lưu vào DMEM tại địa chỉ ***rs1 + ext(imm[11:5]imm[4:0])***.

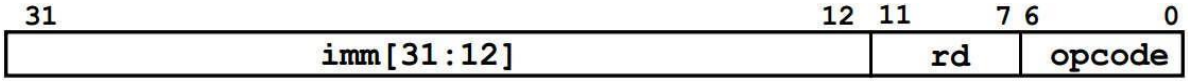
e. Nhóm lệnh B (rẽ nhánh)

imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

- Nhóm lệnh này có opcode là [6:0] = 1100011
- Nhóm lệnh này sẽ thực hiện chuyển giá trị của thanh ghi PC thành giá trị được lưu trong các phần ***imm*** giá trị lưu trong ***rs1*** và ***rs2*** thỏa điều kiện câu lệnh (bằng, không bằng, lớn hơn hoặc bằng, …).
- Khi lấy giá trị lưu ở phần ***imm*** ta phải ghép lại cho đúng thứ tự và mở rộng dấu, bit LSB luôn luôn bằng 0.
  Lấy ví dụ như ở hình dưới:



f. Nhóm lệnh U



- Với lệnh LUI

  Opcode = 0110111

  Lệnh này load giá trị ***imm[31:12]00000000000*** vào thanh ghi ***rd***.
- Với lệnh AUIPC
  Opcode = 0010111
  Lệnh này load giá trị ở ***PC*** vào thanh ghi ***rd***.

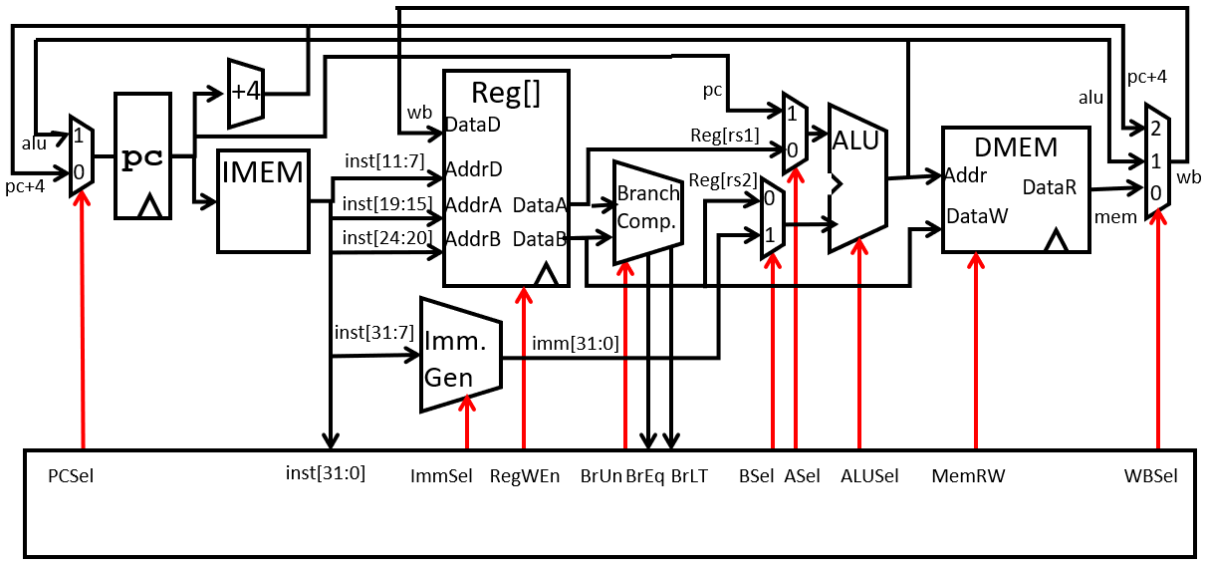
g. Nhóm lệnh J (nhảy không điều kiện)

imm[20|10:1|11|19:12]			rd	1101111	JAL
imm[11:0]	rs1	000	rd	1100111	JALR

II. CẤU TRÚC RISC-V PIPELINE

1. Các trạng thái xử lý câu lệnh trong cấu trúc CPU Pipeline

Cấu trúc CPU RISC-V pipeline được phát triển từ CPU RISC-V đơn chù kỳ có sơ đồ khối dưới đây.

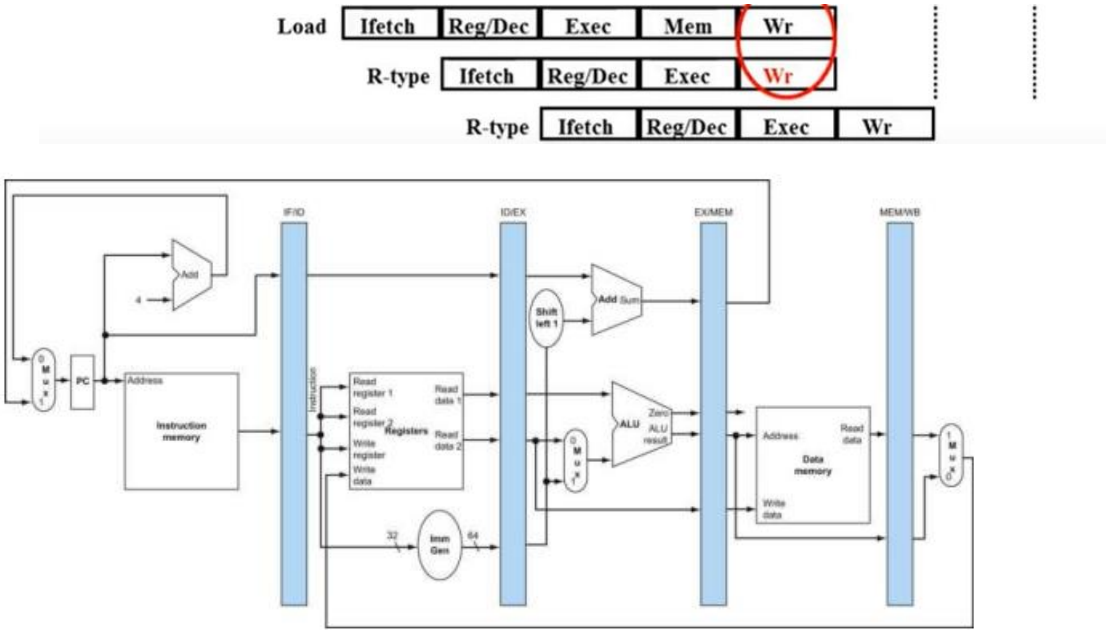


Dựa trên thiết kế của singel-cycle ta nhận thấy rằng, để thực hiện 1 câu lệnh cần nhiều nhất là 5 trạng thái. Các trạng thái được nêu chi tiết ở sau đây

IF	ID	EX	MEM	WB
Instruction Fetch	Instruction decode/register file read	Execute/address calculation	Memory access	Write back
Nạp lệnh	Giải mã lệnh	Tính toán	Truy cập MEM (nếu có)	Ghi kết quả vào bộ nhớ (nếu có)

Cấu trúc pipeline cải thiện hiệu suất bằng cách tăng số câu lệnh được xử lý trong cùng một thời điểm tuy nhiên giảm thời gian xử lý cho từng câu lệnh riêng biệt. Trong cấu trúc CPU pipeline, các giá trị của tầng pipeline phải được lưu trữ để được xử lý ở tầng pipeline

tiếp theo. Do đó cấu trúc CPU đơn chu kì sẽ được chia làm 5 ứng với 5 trạng thái xử lý của một câu lệnh tiêu biểu và thêm vào đó các thanh ghi để lưu trữ.



IF/ID register: Đại diện cho phân chia của Pineline Register giữa tìm nạp lệnh và giải mã lệnh
ID/EX register: Đại diện cho phân chia của Pineline Register giữa giải mã lệnh và tính toán
EX/MEM resgister: Đại diện cho phân chia của Pineline Register giữa tính toán và truy cập bộ nhớ

MEM/WB register: Đại diện cho phân chia của Pineline Register giữa truy cập bộ nhớ và ghi kết quả.

2. Hazard trong pipeline

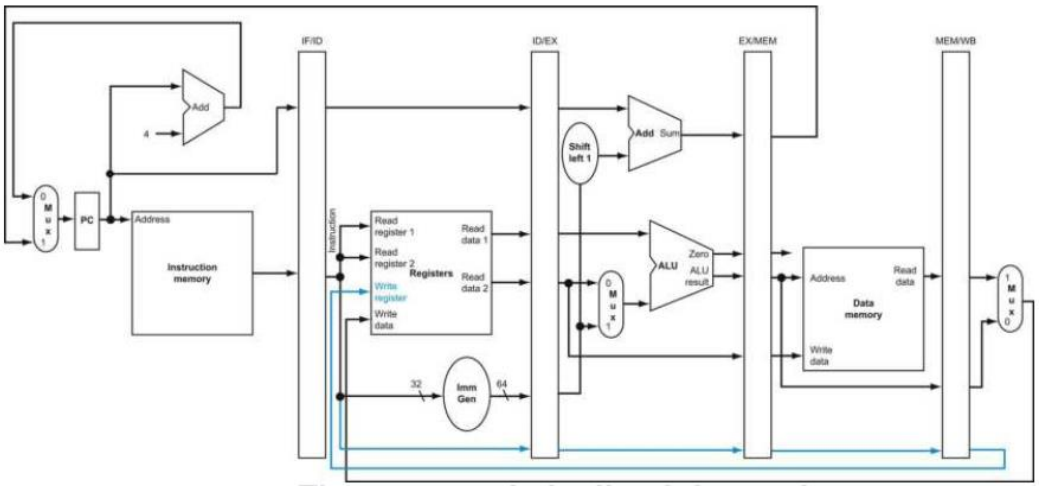
Trong các trường hợp đặc biệt, câu lệnh kế tiếp sẽ không được thực thi trong chu kỳ kế tiếp. Các sự kiện này được gọi là hazard. Có 3 loại hazard cần quan tâm trong việc thiết kế CPU pipeline:

a. Structural Hazard

Structual hazard xảy ra khi có hai câu lệnh cần truy cập vào phần cứng trong cùng một chu kỳ nhưng phần cứng chỉ có thể thực thi một câu lệnh trong một chu kỳ.

Strutual hazard trong cấu trúc pipeline xảy ra với lệnh Load, khi câu lệnh cần ghi dữ liệu vào bộ nhớ trong lúc một lệnh khác đang truy cập bộ nhớ.

Để giải quyết Structual hazard trong trường hợp này, dữ liệu của câu lệnh R-format sẽ được lưu vào các thanh ghi để được xử lý sau khi xử lý lệnh Load



b. Data Hazard

Data hazard xảy ra khi một câu lệnh không thể được xử lý trong chu kỳ đã định trước vì dữ liệu cần được xử lý chưa xuất hiện. Data hazard có thể được chia làm 2 loại:

- R-format instruction

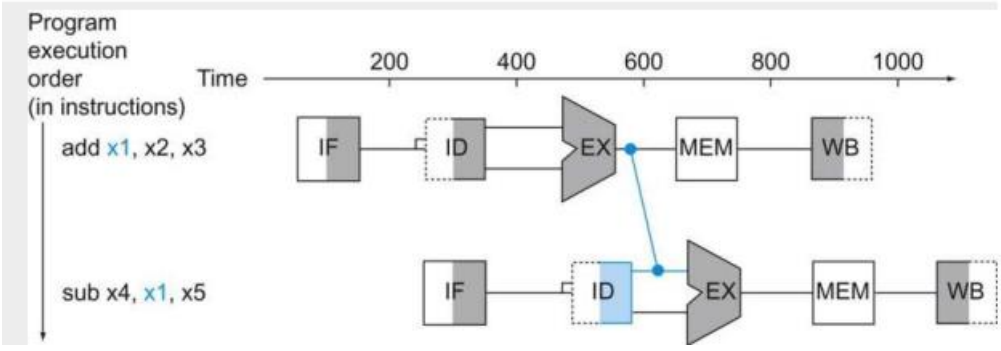
Data hazard có thể được khắc phục bằng cách sử dụng “stall” – không làm gì cả cho đến khi dữ liệu để tính toán xuất hiện.

Xét ví dụ sau:

add x1, x2,x3

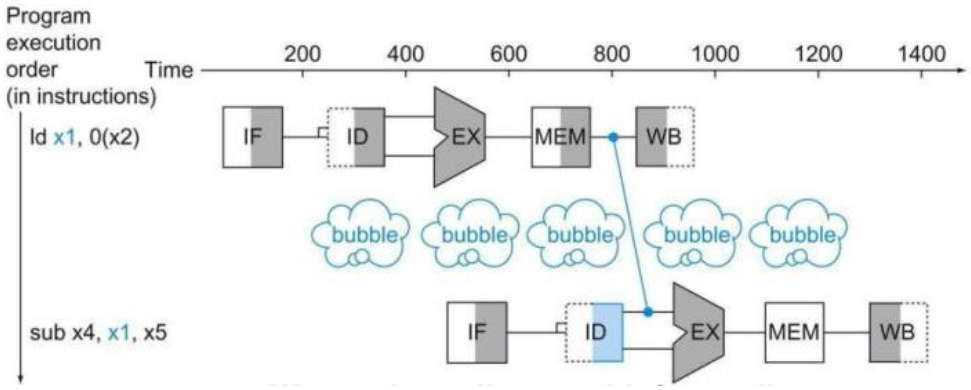
sub x4, x1,x5

Nếu sử dụng stall ta mất 3 chu kì trống. Tuy nhiên dữ liệu cho x đã có sau khi ALU tính toán tại Excecute stage mà không cần đợi đến Write back để lấy dữ liệu. Việc gửi dữ liệu trực tiếp ngay từ ALU gọi là “forwarding”.



- Load Instruction

Load data hazard xuất hiện khi dữ liệu từ lệnh Load chưa được xuất hiện nhưng dữ liệu đó đã được sử dụng cho một câu lệnh khác. Khi xuất hiện câu lệnh sử dụng kết quả từ lệnh Load, phần cứng sẽ được stall một chu kì.

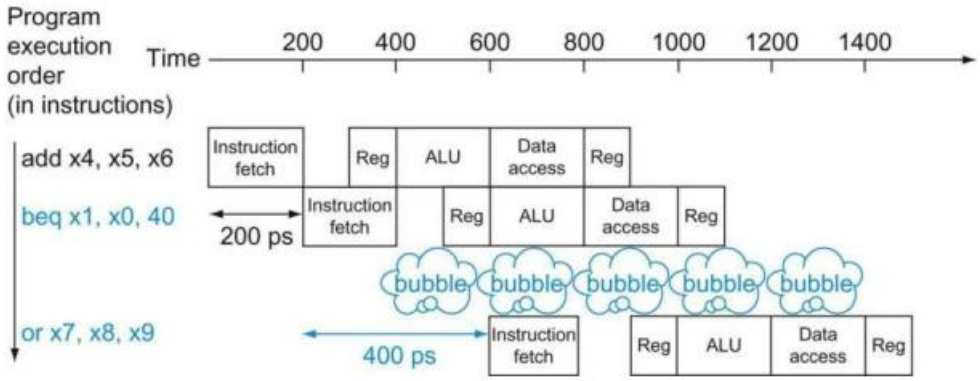


c. Control hazard / Branch hazard

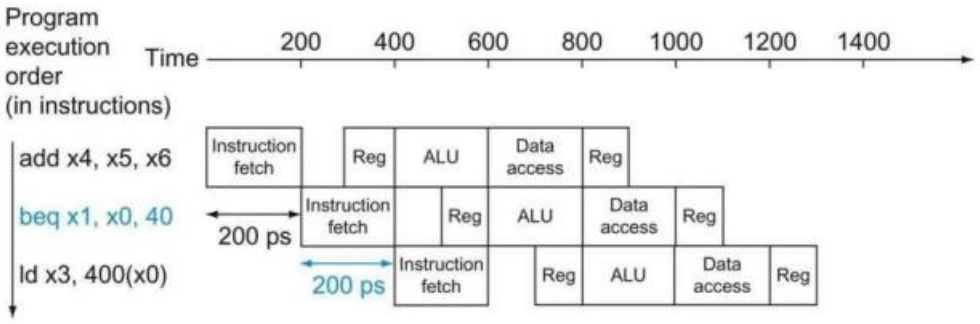
Control hazard xảy ra khi một câu lệnh không được xử lý trong chu kỳ đã được định vì câu lệnh được giải mã ở chu kì trước không phải câu lệnh cần được thực thi.

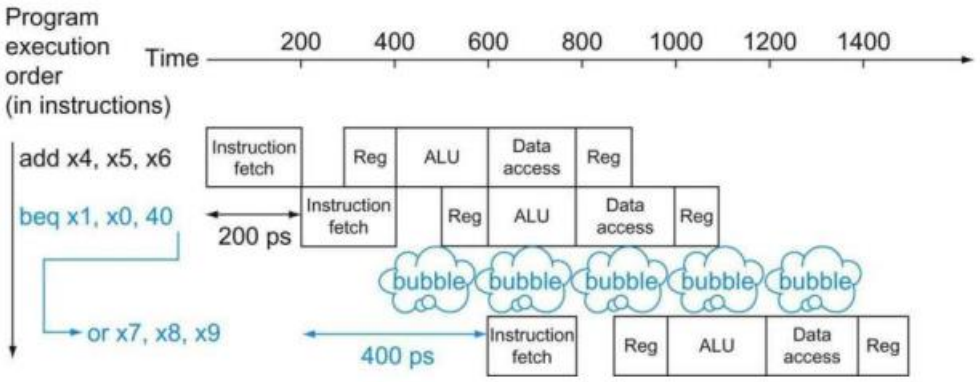
Có thể xử lý control hazard theo hai cách:

- Stall : khi có lệnh nhảy có điều kiện, chương trình sẽ được stall cho đến khi thực thi xong lệnh nhảy.



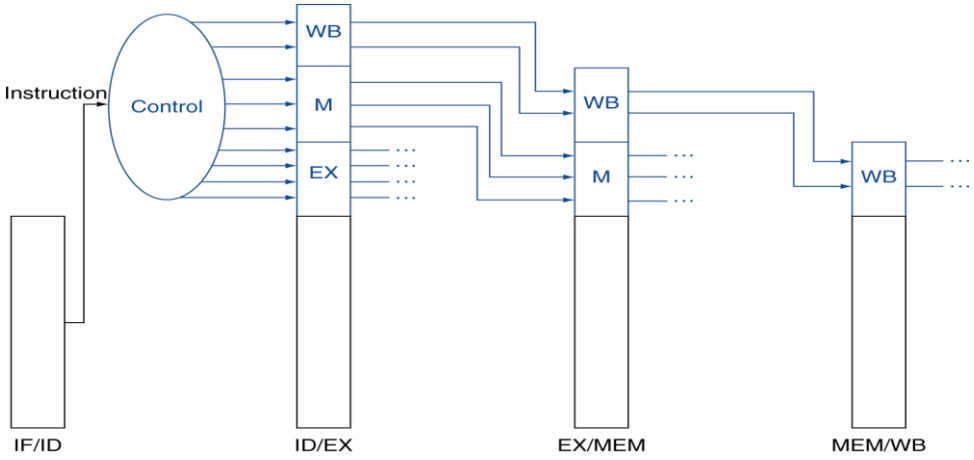
- Predict: khi có lệnh nhảy có điều kiện, chương trình thực hiện dự đoán địa chỉ câu lệnh kế tiếp và thực thi. Tuy nhiên nếu kết quả dự đoán sai sẽ phải thực hiện lại từ đầu.



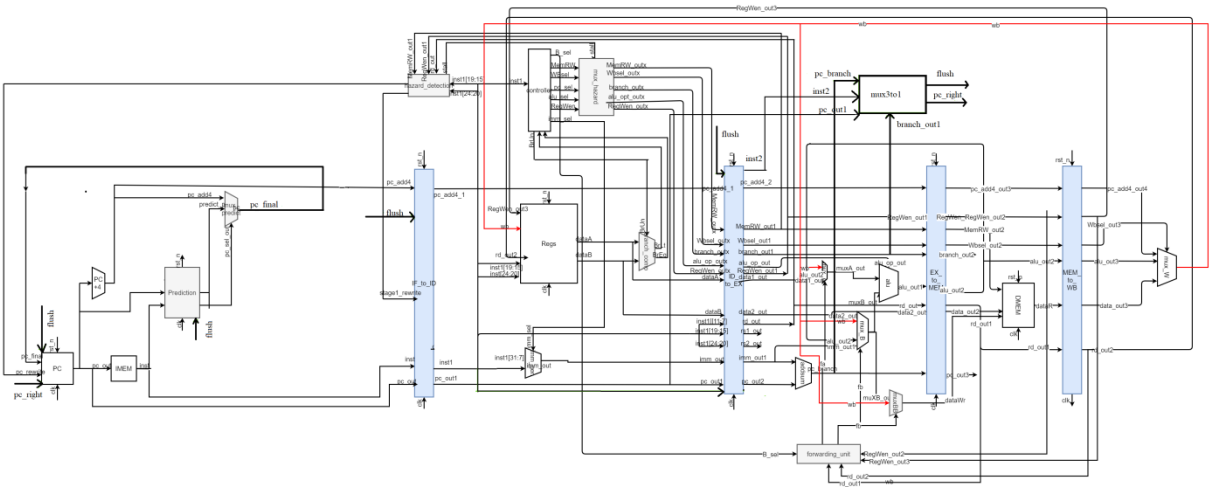


3. Tín hiệu điều khiển trong pipeline

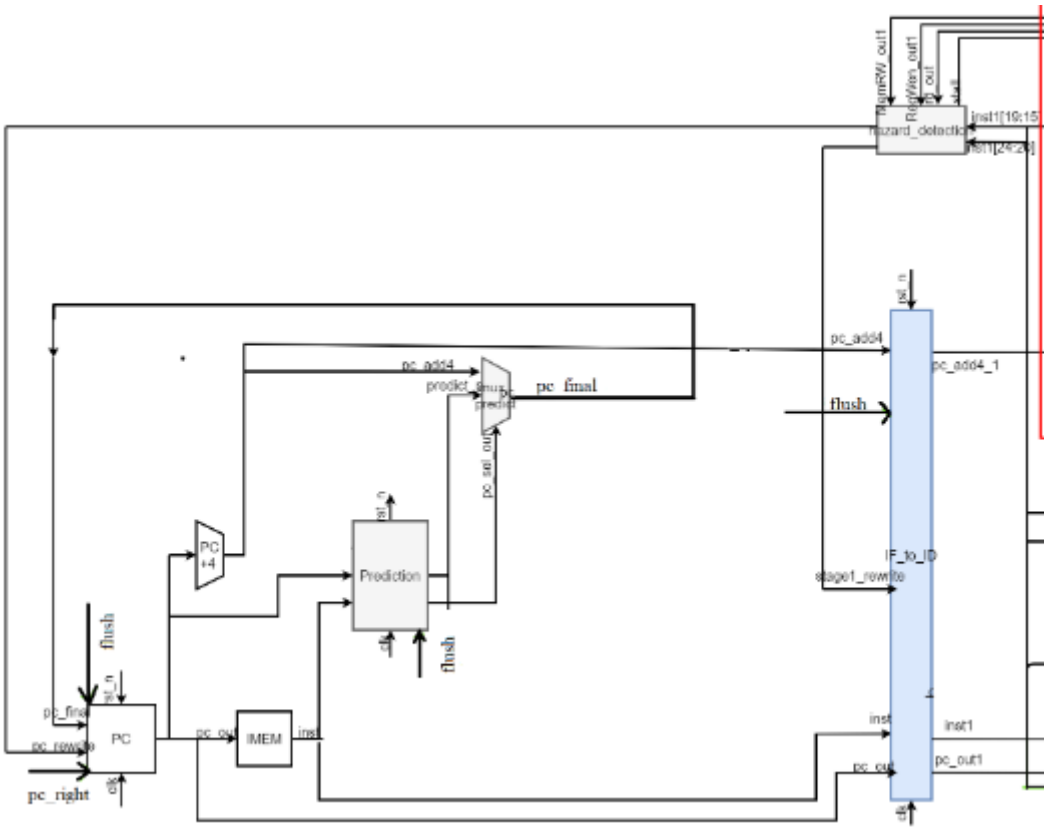
Tín hiệu điều khiển trong cấu trúc CPU pipeline cơ bản được giữ nguyên so với đơn chu kì. Tuy nhiên không phải tất cả tín hiệu đều được sử dụng trong cùng một chu kì cho nên các tín hiệu điều khiển cũng được lưu trữ qua các thanh ghi pipeline cho đến khi được sử dụng



III. THIẾT KẾ CPU RISC-V PIPELINE



1. Instruction Fetch



a. PC

- Bộ đếm chương trình, chứa địa chỉ câu lệnh đang được nạp vào bộ nhớ.
- Hoạt động:

  + PC bị reset về địa chỉ ban đầu khi có tín hiệu ***rst_n*** tích cực thấp.

+ PC được cập nhật địa chỉ câu lệnh tiếp theo khi có tín hiệu ***clk*** tích cực cao. Khi đó PC lấy dữ liệu ***pc_final*** từ khối **prediction.**

+ Khi có tín hiệu **flush** thì lấy địa chỉ từ **pc_right**.

+ Khi có ***pc_rewrite***, PC giữ nguyên giá trị hiện có.

- Đoạn code Verilog cho PC:

```
module PC (input clk, rst_n, pc_rewrite,flush,
           input [31:0] pc_in,pc_right,
           output reg [31:0] pc_out);
always @(posedge clk)
begin
  if (!rst_n)
    pc_out <= 32'h00400000;
  else if (pc_rewrite)
    pc_out <= pc_out;
  else if (flush)
    pc_out <= pc_right;
  else
    pc_out <= pc_in;
end
endmodule
```

b. PC_add4:

- Thực hiện tính địa chỉ của câu lệnh kế tiếp.
- Ngõ vào: ***pc_out*** ; ngõ ra ***pc_add_4***
- Đoạn code Verilog cho khối PC_add4

```
module pc_add4 (input [31:0] pc_in,
                output reg [31:0] pc_out);
always @(pc_in)
begin
 pc_out= pc_in +4;
end
endmodule
```

c. IMEM

- Bộ nhớ chương trình, chứa câu lệnh có địa chỉ lưu trong **PC**.

d. Prediction

- Dự đoán địa chỉ của câu lệnh tiếp theo, gồm hai khối nhỏ là **lookup** và **bht.**

  + lookup: từ giá trị giá trị ***pc_out*** và ***inst*** hiện tại để dự đoán địa chỉ tiếp theo ***predict_pc.*** Đoạn code Verilog cho khối **lookup:**

```
module lookup (input [31:0] pc, inst,
               output [31:0] predict_pc);
  reg [11:0] predict;
always @(pc, inst) begin
  if ((inst[6:2]==5'b11000) || (inst[6:2]==5'b11011) || (inst[6:2]==5'b11001)) begin
  case(pc[11:0])
    12'h00c: predict= 12'h034;
    12'h01c: predict= 12'h038;
    12'h034: predict= 12'h058;
    12'h044: predict= 12'h03c;
    12'h024: predict= 12'h010;
    12'h054: predict= 12'h020;
endcase
end
end

assign predict_pc= {20'h00400, predict};
endmodule
```

+ bht: Khi có tín hiệu ***clk*** tích cực cao, khối bht sử dụng ***inst*** và ***pc_sel*** để quyết định nhảy hoặc không nhảy với các biến trạng thái current, next có các trạng thái: SN,WN,WT,ST. Kết quả lưu ***ở pc_sel_out***. Khi **có rst_n,** biến trạng thái trở current về giá trị SN.

```
module bht (input clk, rst_n,
            input [31:0] pc,
            input x,
            output reg y);
parameter [1:0] SN=2'b00, WN=2'b01, WT=2'b10, ST=2'b11;
reg [1:0] current, next;

always @(current, x, pc) begin
  if (pc[6:2]==5'b11000) begin
  case (current)
    ST: if (x)
          next = ST;
        else
          next = WT;
    WT: if (x)
          next = ST;
        else
          next = WN;
    WN: if (x)
          next = WT;
        else
          next = SN;
    SN: if (x)
          next = WN;
        else
          next = SN;
  endcase
  //y= ((current== ST) || (current==WT));
end
end

always @(current) begin
  case (current)
    2'b00: y=1'b0;
    2'b01: y=1'b0;
    2'b10: y=1'b1;
    2'b11: y=1'b1;
  endcase
end

always @(posedge clk) begin
  if (!rst_n)
    current <= SN;
  else
    current <= next;
end
endmodule
```

e. Mux_predict

- Sử dụng tín hiệu ***pc_sel_out*** từ khối **Prediction** để lựa chọn giữa ***pc_add4*** và ***pc_predict***, *cho kết quả ở* ***pc_mux_out.***

```
1  module mux_predict (input [31:0] pc, predict,
2                      input prediction,
3                      output [31:0] pc_mux_out);
4   assign pc_mux_out= (prediction==1'b1)? predict:pc;
5   endmodule
```

f. Thanh ghi IF_to_ID

- Lưu các giá tri từ tầng Instruction Fetch bao gồm: ***pc_add4, flush, inst, pc_out.*** Cập nhật sau mỗi chu kì clock ***clk*** và bị xóa khi có ***rst_n.***

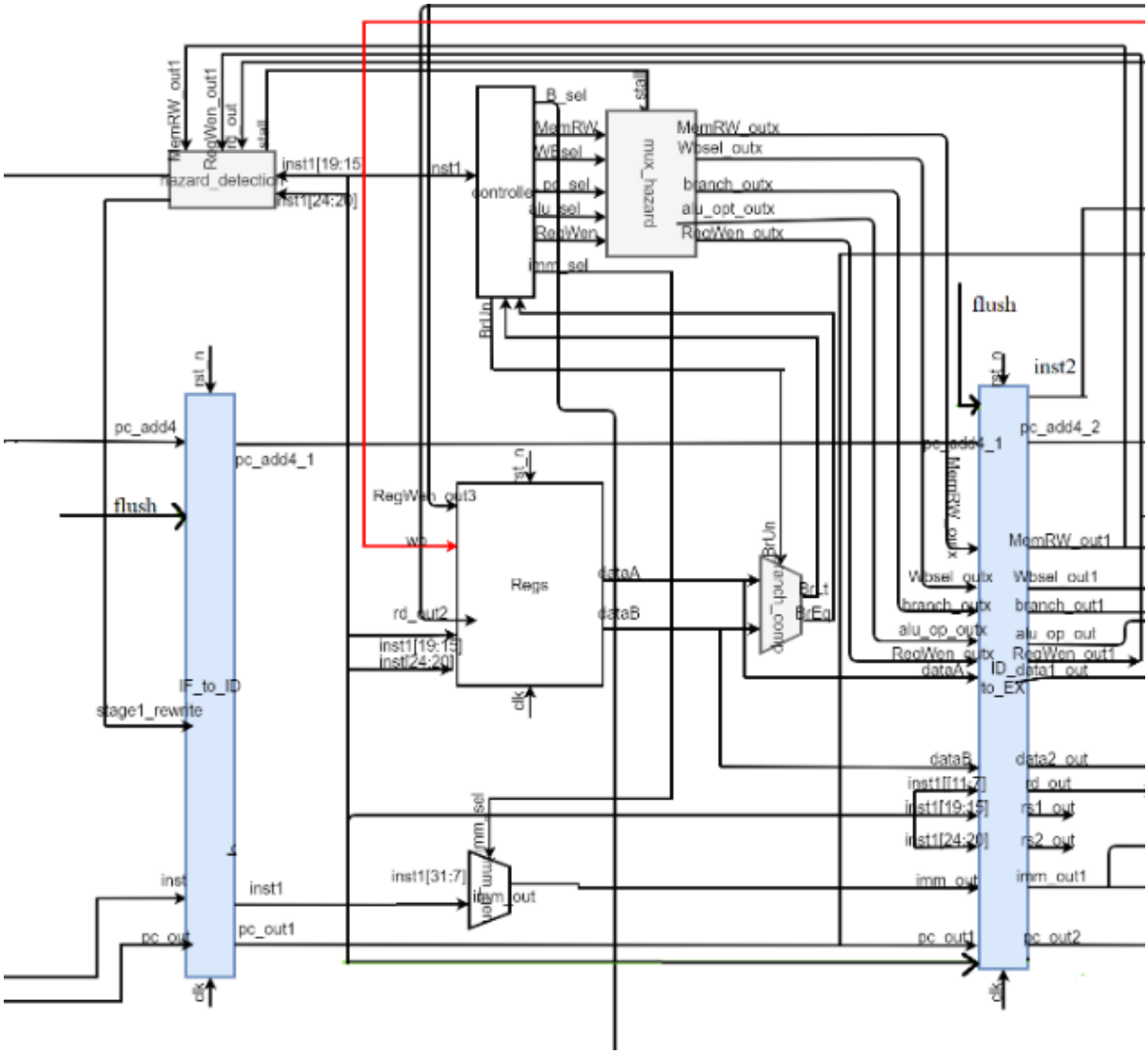
- Khi có tín hiệu ***stage1_rewrite***, các giá trị lưu trong thanh ghi được giữ nguyên.
- Khi có tín hiệu **flush** thì xoá các giá trị trong thanh ghi.
- Đoạn code Verilog cho thanh ghi:

```
module IF_to_ID ( input clk, rst_n, stage1_rewrite, flush,//clock, reset, flush data
                  input [31:0] pc_in, pc_add4,//pc+4 for next clock cycle, pc for branch instruction calculation
                  input [31:0] inst_in, //instruction fetch
                  output reg [31:0] pc_out, pc_add4_out, //pc_out
                  output reg [31:0] inst_out); //instruction out



always @(posedge clk) begin
  if (!rst_n) begin
    pc_out <= 32'h00400000;
    pc_add4_out <= 32'h00000000;
    inst_out <= 32'hllllllll;
  end
  //end
  else if (stage1_rewrite) begin
    pc_out <= pc_out;
    pc_add4_out <= pc_add4_out;
    inst_out <= inst_out;
  end
  else if (flush) begin
    pc_out <= 32'h00400000;
    pc_add4_out <= 32'h00000000;
    inst_out <= 32'hllllllll;
    end
  else begin
    pc_out <= pc_in;
    pc_add4_out <= pc_add4;
    inst_out <= inst_in;
  end
end
endmodule
```

2. Instruction Decode



a. Regs

- Tập các thanh ghi cho phép truy xuất dữ liệu, gồm 32 thanh ghi mỗi thanh ghi 32 bit.
- Hoạt động:

  + Khi có tín hiệu clk tích cực thấp, regs sử dụng giá trị inst1[19:15] để xác định miền rs1 và xuất giá trị lưu trong rs1 đến ngõ ra dataA.

  + Khi có tín hiệu clk tích cực cao, regs sử dụng giá trị ins1[24:29] để xác định miền rs2 và xuất giá trị lưu trong rs1 đến ngõ ra dataB.

  + Khi có tín hiệu clk tích cực cao, regs sử dụng giá trị rd_out2 để xác định định miền rd.

+ Khi có tín hiệu cho phép ghi vào thanh ghi RegWen_out3, giá trị của wb sẽ được lưu vào Regs

- Để đảm bảo trong mỗi chu kì chỉ có một câu lệnh được xử lý, các giá trị rd_out2 và RegWen_out3 được lấy từ tầng Write back.
- Đoạn code Verilog cho tập các thanh ghi : Phụ lục.

b. Imm_gen

- Mở rộng dấu giá trị Immediate tùy thuộc vào câu lệnh được thực thi.
- Ngõ vào inst[31:7] , ngõ ra imm_out, dung imm_sel từ controller để quyết định cách mở rộng dấu.
- Đoạn code Verilog cho imm_gen

```
module imm_gen (input [31:7] imm_in,
                input [2:0] imm_sel,
                output reg [31:0] imm_out);
 always @(imm_sel, imm_in)
begin
  case (imm_sel)
    3'b000: //I-format
            imm_out= (imm_in[31]==1'b1) ? {21'b111111111111111111111, imm_in[30:20]}:{21'b000000000000000000000, imm_in[30:20]};
    3'b001: //S-format
            imm_out= (imm_in[31]==1'b1) ? {21'b111111111111111111111, imm_in[30:25], imm_in[11:7]}: {21'b000000000000000000000, imm_in[30:25], imm_in[11:7]};
    3'b010: //B-format
            imm_out= (imm_in[31]==1'b1) ? {20'b11111111111111111111, imm_in[7], imm_in[30:25], imm_in[11:8], 1'b0}: {20'b00000000000000000000, imm_in[7], imm_in[30:25], imm_in[11:8], 1'b0};
    3'b011: //U-format
            imm_out= {imm_in[31:12], 12'b000000000000};
    3'b100: //J-format
            imm_out= (imm_in[31]==1'b1) ? {11'b11111111111, imm_in[31], imm_in[19:12], imm_in[20], imm_in[30:21], 1'b0}: {11'b00000000000, imm_in[31], imm_in[19:12], imm_in[20], imm_in[30:21], 1'b0};
    3'b101: //R-format
            imm_out= 32'h00000000;
    default:;
  endcase
end
endmodule
```

c. Branch_comp

- So sánh giá hai giá trị data và dataB, cho kết quả nhỏ hơn BrLt hoặc bằng BrEq.
- Ngõ vào BrUn quyết định phép so sánh là có dấu hay không dấu.
- Đoạn code Verilog: Phụ lục.

d. Controller

- Điều khiển việc xử lý các câu lệnh
- Ngõ vào:

  + inst1: dựa vào câu lệnh đang được thực thi để quyết định các tín hiệu ngõ ra.

  + BrEq, BrLT: sử dụng kết quả so sánh hai thanh ghi từ khối branch_comp để quyết định các giá trị ngõ ra.
- Ngõ ra:

  + pc_sel: quyết định PC tiếp theo từ địa chỉ nhảy hoặc địa chỉ câu lệnh kế tiếp.

+ RegWen: tín hiệu cho phép ghi vào tập thanh ghi.

+ BrUn: quyết định phép so sánh có dấu hay không dấu.

+ B_sel: quyết định chọn giá trị tại mux_B trong đơn chu kỳ, tuy nhiên trong cấu trúc Pipeline B_sel được xử dụng để xác định hazard trong khối forwarding.

+ MemRW: quyết định cho phép truy xuất vào bộ nhớ dữ liệu DMEM.

+ alu_sel: quyết định loại phép tính được thực thi bởi ALU.

+ WBsel: quyết định giá trị được ghi lại vào tập thanh ghi.

- Đoạn code Verilog: Phụ lục.

e. Mux_hazard

- Xóa các giá trị được lưu khi có tín hiệu stall.
- Đoạn code Verilog:

```
module mux_hazard (input stall,
                   input RegW, branch, MemRW,
                   input [1:0] MemReg,
                   input [3:0] alu_op,
                   output reg RegW_out, branch_out, MemRW_out,
                   output reg [1:0] MemReg_out,
                   output reg [3:0] alu_op_out  );
always @(stall, RegW, branch, MemRW, MemReg, alu_op) begin
  if (stall) begin
    RegW_out= 1'b0;
    branch_out= 1'b0;
    MemRW_out= 1'b0;
    MemReg_out= 2'b00;
    alu_op_out= 4'b0000;
  end
  else begin
    RegW_out= RegW;
    branch_out= branch;
    MemRW_out= MemRW;
    MemReg_out= MemReg;
    alu_op_out= alu_op;
  end
end
endmodule
```

f. Hazard_detection

- Phát hiện data hazard khi một lệnh cần sử dụng giá trị chưa được xử lý xong từ lệnh Load. Khi có hazard các tín hiệu điều khiển bị xóa, giá trị được lưu tại PC và thanh ghi ID_to_EX được giữ nguyên không cập nhật.
- Đoạn code Verilog:

```
module hazard_detection (input [4:0] rs1, rs2, rd,
                         input MemRead, RegWen,
                         output reg pc_rewrite, stage1_rewrite, stall);

always @(rs1, rs2, rd, MemRead) begin
  if ((MemRead==1'b0) && (RegWen==1'b1)) begin
    if (rd==rs1 || rd==rs2) begin
      pc_rewrite= 1'b1;
      stage1_rewrite =1'b1;
      stall= 1'b1;
    end
  end
  else begin
    pc_rewrite= 1'b0;
    stage1_rewrite =1'b0;
    stall= 1'b0;
  end
end
endmodule
```

g. Thanh ghi ID_to_EX

- Lưu các giá trị tại tầng Instruction Decode được sử dụng tại các tầng sau đó bao gồm:

  + Các tín hiệu điều khiển: MemRW_outx, Wbsel_outx, branch_out, alu_op_outx,RegWen_outx.

  + Các tín hiệu từ tập thanh ghi: data, dataB

  + Câu lệnh đang được thực thi: inst

  + Giá trị tức thời: imm_out

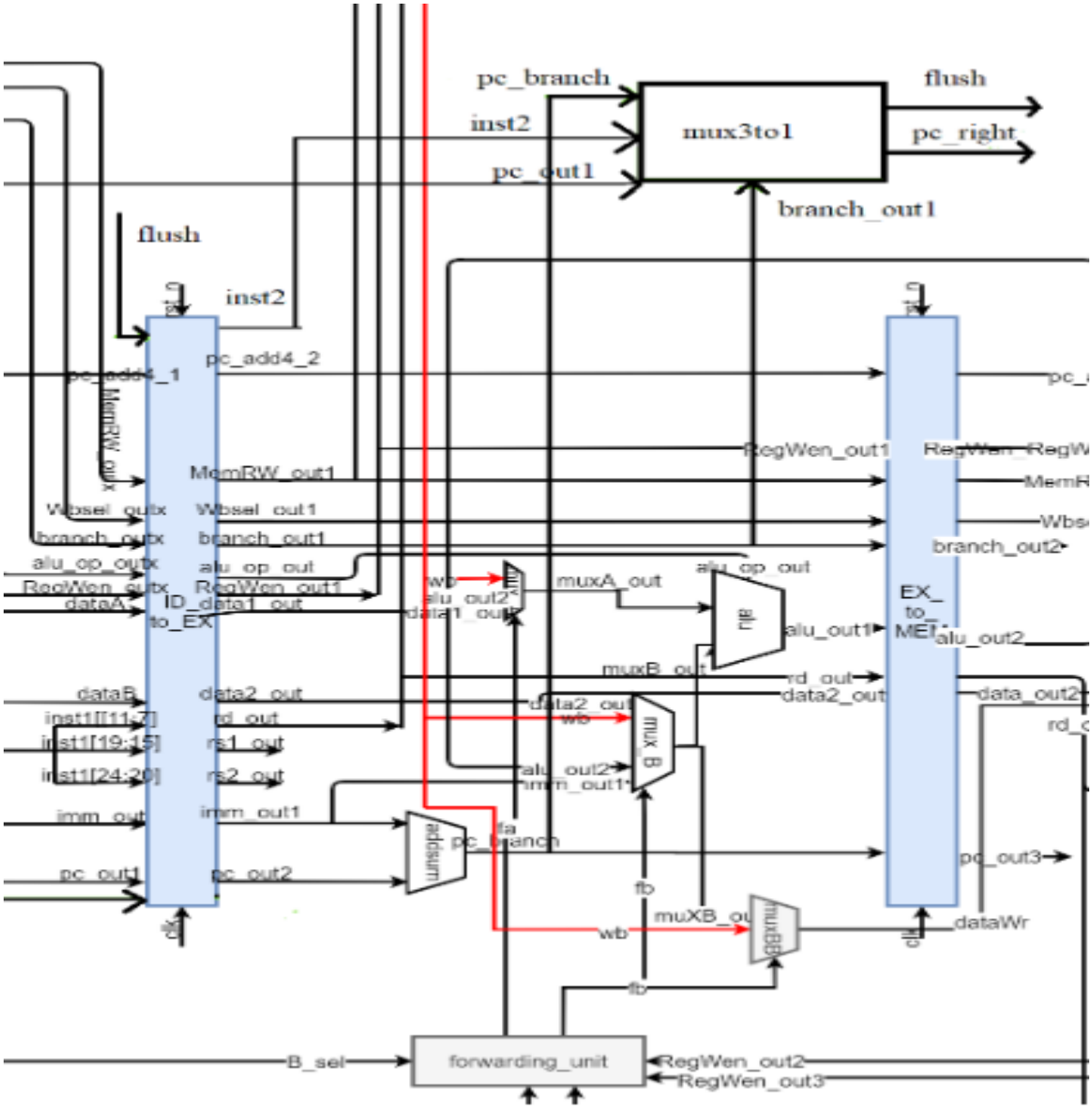
  + Giá trị PC được lưu từ tầng Instruction Fetch: pc_out1.

  + Khi có rst_n hoặc flush thì xoá hết các giá trị.

```
1  module ID_to_EX (input clk, rst_n,flush, //clock, reset
2                   input [31:0] pc_in, pc_add4, //pc for branch instruction calculation
3                   input [31:0] data1_in, data2_in, //value of register1, register2
4                   input [31:0] imm_in,inst, //value of immediate generator
5                   input [4:0] rd, rs1, rs2,
6                   input [3:0] alu_op,
7                   input branch, memRW,
8                   input RegW,
9                   input [1:0] MemReg,
10                  input B_sel_in,
11
12                  output reg B_sel_out,
13                  output reg [3:0] alu_op_out,
14                  output reg branch_out, memRW_out,
15                  output reg RegW_out,
16                  output reg [1:0] MemReg_out,
17                  output reg [31:0] pc_out, pc_add4_out, //pc_out
18                  output reg [4:0] rd_out, rs1_out, rs2_out,
19                  output reg [31:0] data1_out, data2_out, //value of rs1, rs2
20                  output reg [31:0] imm_out,inst_out); //value of immediate generator out
```

```
22  always @(posedge clk) begin
23    if ((!rst_n) || flush) begin
24      B_sel_out  <= 1'b0;
25      alu_op_out <= 1'b0;
26      branch_out <= 1'b0;
27      memRW_out <= 1'b0;
28      RegW_out <= 1'b0;
29      MemReg_out <= 2'b00;
30      rd_out <= 5'b11111;
31      rs1_out <= 5'b00000;
32      rs2_out <= 5'b00000;
33      pc_out <= 32'h00000000;
34      pc_add4_out <= 32'h00000000;
35      data1_out <= 32'h00000000;
36      data2_out <= 32'h00000000;
37      imm_out <= 32'h00000000;
38      inst_out <= 32'h11111111;
39    end
40    else begin
41      B_sel_out <= B_sel_in;
42      alu_op_out <= alu_op;
43      branch_out <= branch;
44      memRW_out <= memRW;
45      RegW_out <= RegW;
46      MemReg_out <= MemReg;
47      rd_out <= rd;
48      rs1_out <= rs1;
49      rs2_out <= rs2;
50      pc_out <= pc_in;
51      pc_add4_out <= pc_add4;
52      data1_out <= data1_in;
53      data2_out <= data2_in;
54      imm_out <= imm_in;
55      inst_out <= inst;
56    end
57  end
58  endmodule
```

3. Execute



a. Mux_A

- Lựa chọn giá trị toán hạng muxA_out đưa vào khối Alu từ wb, data1_out và alu_out2. Tín hiệu điều khiển fa từ forwarding_unit.
- Đoạn code Verilog:

```
module mux_A (input [31:0] rs1, wb, alu_out,
              input [1:0] A_sel,
              output reg [31:0] muxA_out);
always @(A_sel, wb, alu_out, rs1) begin
  case (A_sel)
    2'b00: muxA_out= rs1;
    2'b01: muxA_out= wb;
    2'b10: muxA_out= alu_out;
    default:;
  endcase
end
endmodule
```

b. Mux_B

- Lựa chọn giá trị toán hạng muxB_out đưa vào khối Alu từ data2_out, alu_out2,wb và imm_out1. Tín hiệu điều khiển fb từ forwarding unit.
- Đoạn code Verilog:

```
module mux_B (input [31:0] rs2, wb, alu_out, imm_in,
              input [1:0] B_sel,
              output reg [31:0] muxB_out);
always @(B_sel, wb, alu_out, rs2) begin
  case (B_sel)
    2'b00: muxB_out= rs2;
    2'b01: muxB_out= wb;
    2'b10: muxB_out= alu_out;
    2'b11: muxB_out= imm_in;
    default:;
  endcase
end
endmodule
```

c. Mux2to1

+ So sánh 2 địa chỉ pc_branch và địa chỉ pc_out1 hiện thời để xuất ra tín hiệu flush và địa chỉ pc_right.

```
1  module mux2to1 (input [31:0] pc_branch, pc_out1, inst,
2                  input branch,
3                  output reg flush,
4                  output reg [31:0] pc_right);
5
6  always @(pc_branch,pc_out1) begin
7    if (((inst[6:2]==5'b11011) || (inst[6:2]==5'b11001) || (inst[6:2] ==5'b11000))
8          && (pc_branch != pc_out1) && branch)
9          begin
10             flush <= 1'b1;
11             pc_right <= pc_branch;
12         end
13     else begin
14         flush <= 1'b0;
15         pc_right <= 32'hzzzzzzzz;
16     end
17 end
18
19 endmodule
```

d. Addsum

- Tính toán địa chỉ nhảy pc_branch từ pc_out2 và imm_out1.
- Đoạn code Verilog:

```
module addsum (input [31:0] pc, imm,
               output [31:0] pc_branch);
assign pc_branch= pc + imm;
endmodule
```

e. Alu

- Thực thi các lệnh ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND.

  + Ngõ vào: muxA_out, muxB_out, alu_op_out

  + Ngõ ra: alu_out1.
- Đoạn code Verilog: Phụ lục

f. MuxBB

- Chọn tín hiệu dataWr từ khối DMEM từ hai ngõ vào muxB_out và wb, tín hiệu chọn fb.
- Đoạn code Verilog:

```
module muxBB (input [1:0] fb,
              input [31:0] dataB, muxB_out,
              output [31:0] dataW);
assign dataW= (fb != 2'b00) ? dataB: muxB_out;
endmodule
```

g. Forwarding_unit

- Kiểm soát Hazard, kiểm tra thanh đích Rd trong stage Memory Access/Write back có giống với thanh ghi mong muốn hay không.

  + Ngõ vào: rd_out1, rd_out2, RegWen_out2, RegWen_out3, rs1_out, rs2_out, B_sel

  + Ngõ ra: fa, fb
- Hoạt động:

  + Kiểm tra thanh ghi đích rd_out_1 trùng với rs1: fa chọn giá trị alu_out_2 vào khối ALU.

  + Kiểm tra thanh ghi đích rd_out_2 và rd_out_3 trùng với rs1: fa chọn giá trị wb vào khối ALU

  + Không có Hazard: fa chọn data1_out

  + Không có tín hiệu B_sel: fb chọn data2_out cho ALU

  + Khi có tín hiệu B_sel: fb chọn imm_out1 cho ALU.

+ Kiểm tra thanh ghi đích rd_out_1 trùng với rs2: fb chọn giá trị alu_out_2 vào khối ALU.

+ Kiểm tra thanh ghi đích rd_out_2 và rd_out_3 trùng với rs2: fb chọn giá trị wb vào khối ALU

- Đoạn code Verilog:

```
module forwarding_unit (input [4:0] EX_MEM_rd, MEM_WB_rd,
                        _2     input [4:0] rs1, rs2,
                               input RegW_in1, RegW_in2,
                               input B_sel,
                               output reg [1:0] fa, fb);

always @(EX_MEM_rd, MEM_WB_rd, RegW_in1, RegW_in2, rs1, rs2, B_sel) begin

 fa=2'b00;
if (RegW_in1 && EX_MEM_rd) begin
    if (EX_MEM_rd == rs1)
      fa= 2'b10;
end
if (RegW_in2 && MEM_WB_rd) begin
    if (!(RegW_in1 && EX_MEM_rd && (EX_MEM_rd == rs1)) && (MEM_WB_rd==rs1))
      fa= 2'b01;
end
/////////
if (B_sel)
  fb=2'b11;
else begin
  fb= 2'b00;
   if (RegW_in1 && EX_MEM_rd) begin
    if (EX_MEM_rd == rs2)
      fb= 2'b10;
   end
   if (RegW_in2 && MEM_WB_rd) begin
    if (!(RegW_in1 && EX_MEM_rd && (EX_MEM_rd == rs2)) && (MEM_WB_rd==rs2))
      fb= 2'b01;
   end
end
end
endmodule
```

h. Thanh ghi EX_to_MEM:

- Lưu trữ các giá trị từ tầng Execute được sử dụng cho các tầng kế tiếp, bao gồm:

+ Tín hiệu điều khiển: RegWen_out1, MemRW_out1,Wbsel_out1,brach_out1

+ Tín hiệu ngõ ra ALU: alu_out1

+ Toán hạng đích rd được lưu từ tầng Decode: rd_out1

+ Giá trị cần được ghi vào DMEM lưu từ tầng Decode: data2_out1

+ Địa chỉ PC lưu từ tầng Fetch: pc_add4_out2

+ Địa chỉ nhảy: pc_branch

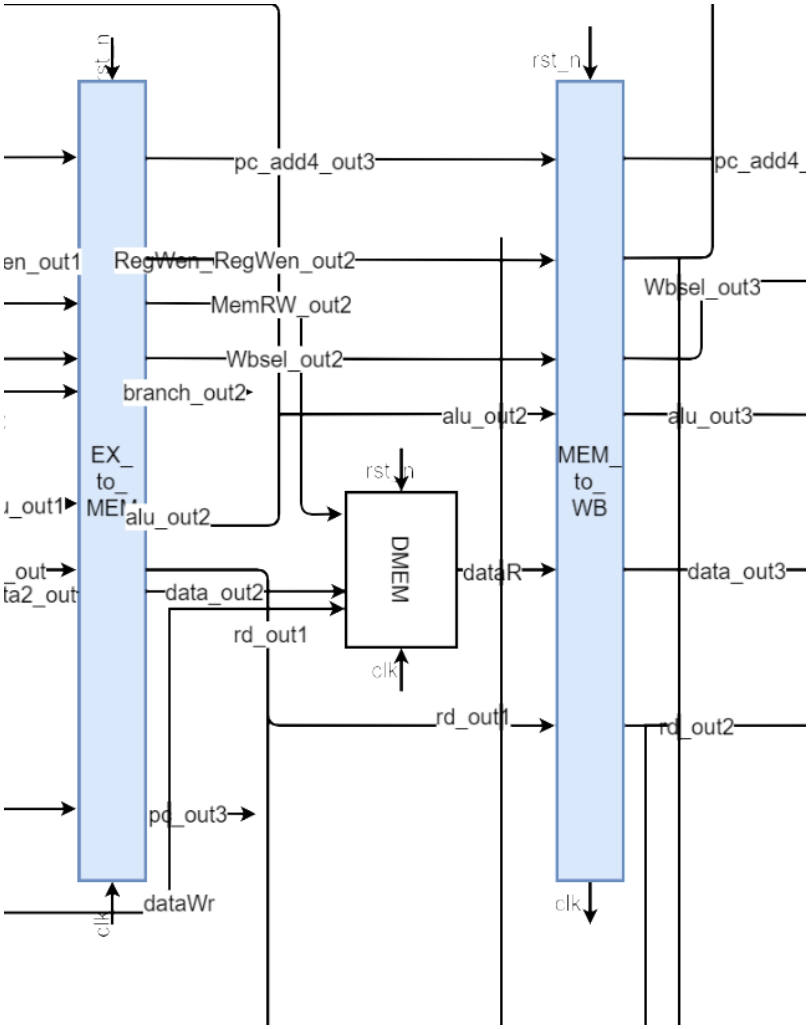
- Đoạn code Verilog

```
module EX_to_MEM (input clk, rst_n,
                  input [31:0] pc_branch, pc_add4, //calculated pc for branch instruction
                  input [31:0] data_in, // data for store_word instruction
                  input [31:0] alu_in, //result from alu
                  input [4:0] rd,
                  input branch, memRW,
                  input RegW,
                  input [1:0] MemReg,

                  output reg branch_out, memRW_out,
                  output reg RegW_out,
                  output reg [1:0] MemReg_out,
                  output reg [4:0] rd_out,
                  output reg [31:0] pc_branch_out, pc_add4_out, //pc of label
                  output reg [31:0] alu_out, //addr for DMEM
                  output reg [31:0] data_out); //data store in DMEM

always @(posedge clk) begin
  if (!rst_n) begin
    branch_out <= 1'b0;
    memRW_out <= 1'b0;
    RegW_out <= 1'b0;
    MemReg_out <= MemReg;
    rd_out <= 5'b00000;
    pc_branch_out <= 32'h00000000;
    pc_add4_out <= 32'h00000000;
    alu_out <= 32'h00000000;
    data_out <= 32'h00000000;
  end
  else begin
    branch_out <= branch;
    memRW_out <= memRW;
    RegW_out <= RegW;
    MemReg_out <= MemReg;
    rd_out <= rd;
    pc_branch_out <= pc_branch;
    pc_add4_out <= pc_add4;
    alu_out <= alu_in;
    data_out <= data_in;
  end
end
endmodule
```

4. Memory Access



a. Khối DMEM:

- Bộ nhớ dữ liệu, sử dụng cho các lệnh Load và Store.
- Hoạt động: Tại mỗi cạnh lên clk, DMEM kiểm tra nếu có tín hiệu cho phép truy cập bộ nhớ MemRW_out2

  + Đọc: từ giá trị dataWr để gửi giá trị cần đọc từ DMEM đến dataR

  + Ghi: ghi giá trị data_out2 vào thanh ghi có địa chỉ lấy từ rd_out1.
- Đoạn code Verilog:

```
module DMEM (input clk, rst_n,
             input [31:0] dataW, Addr,
             input MemRW,              //0: read, 1: write
             output [31:0] dataR);
 reg [7:0] DMEMi [0:256];

/*always @(Addr) begin
   if (MemRW==1'b1)
     DMEMi[Addr] = dataW;
   else
     DMEMi[Addr] =  DMEMi[Addr];
 end

 always @(posedge clk) begin
   if (!rst_n)
     dataR <= 32'h00000000;
   else
     dataR <= (MemRW==1'b0)? DMEMi[Addr]: 32'h00000000;
 end
 */
 assign dataR=(MemRW==1'b0)? {DMEMi[Addr+3], DMEMi[Addr+2], DMEMi[Addr+1], DMEMi[Addr]}: 32'hxxxxxxxx;
always @(posedge clk) begin
  if (!rst_n) begin
    //dataR <= 32'hxxxxxxxx;
    {DMEMi[Addr+3], DMEMi[Addr+2], DMEMi[Addr+1], DMEMi[Addr]} <= 32'hxxxxxxxx;
  end
  else begin
    if (MemRW==1'b1) begin
      {DMEMi[Addr+3], DMEMi[Addr+2], DMEMi[Addr+1], DMEMi[Addr]} <= dataW;
      //dataR <= 32'bxxxxxxxx;
    end
    else begin
      {DMEMi[Addr+3], DMEMi[Addr+2], DMEMi[Addr+1], DMEMi[Addr]} <=  {DMEMi[Addr+3], DMEMi[Addr+2], DMEMi[Addr+1], DMEMi[Addr]};
      //dataR <= DMEMi[Addr];
    end
  end
end
endmodule
```

b. Thanh ghi MEM_to_WB

- Lưu các giá trị từ tầng Memory Access cần sử dụng ở tầng Write Back, bao gồm:

  + Tín hiệu điều khiển: Wbsel_out2, RegWen_out2

  + Thanh ghi đích: rd_out1.

  + Giá trị từ ALU: alu_out2

  + Giá trị PC: pc_add4_out3

- Đoạn code Verilog:

```
module MEM_to_WB (input clk, rst_n,
                  input [31:0] alu_in, //for normal instruction except load instruction
                  input [31:0] pc_add4,
                  input [31:0] dataR_in, //data from load instruction
                  input [4:0] rd,
                  input RegW,
                  input [1:0] MemReg,

                  output reg RegW_out,
                  output reg [1:0] MemReg_out,
                  output reg [4:0] rd_out,
                  output reg [31:0] alu_out,
                  output reg [31:0] pc_add4_out,
                  output reg [31:0] dataR_out);
 always @(posedge clk) begin
   if (!rst_n) begin
     RegW_out <= 1'b0;
     MemReg_out <= MemReg;
     rd_out <= 5'b00000;
     alu_out <= 32'h00000000;
     pc_add4_out <= 32'h00000000;
     dataR_out <= 32'h00000000;
   end
   else begin
     RegW_out <= RegW;
     MemReg_out <= MemReg;
     rd_out <= rd;
     alu_out <= alu_in;
     pc_add4_out <= pc_add4;
     dataR_out <= dataR_in;
   end
 end
 endmodule
```

5. Write Back

Khối MuxW:

- Chọn giá trị wb được ghi trở lại vào DMEM từ pc_add4_out4, alu_out3 và data_out3.
- Đoạn code Verilog:

```
module mux_W (input [31:0] mem, alu, pc_add4,
              input [1:0] WB_sel,
              output reg [31:0] wb);
always @(WB_sel, mem, alu, pc_add4) begin
  case (WB_sel)
    2'b00: wb= mem;
    2'b01: wb= alu;
    2'b10: wb= pc_add4;
    default: wb= 32'hxxxxxxxx;
  endcase
end
endmodule
```

IV. PHỤ LỤC

1. Tập thanh ghi Regs

```
module regs (input clk, rst_n,
             input [4:0] rs1, rs2, rd,
             input [31:0] wb,
             input RegWen,
             output reg [31:0] dataA, dataB);
wire [31:0] reg_0;
reg [31:0] reg_1, reg_2, reg_3, reg_4, reg_5, reg_6, reg_7,
          reg_8, reg_9, reg_10, reg_11, reg_12, reg_13, reg_14, reg_15,
          reg_16, reg_17, reg_18, reg_19, reg_20, reg_21, reg_22, reg_23,
          reg_24, reg_25, reg_26, reg_27, reg_28, reg_29, reg_30, reg_31;
wire [31:0] dataD;
reg [4:0] rs11, rs22;
//always @(rs1, rs2, rd)
always @(negedge clk)
begin
 // rs11 <= rs1;
  case (rs1)
    5'b00000: dataA<= 32'h00000000;
    5'b00001: dataA<= reg_1;
    5'b00010: dataA<= reg_2;
    5'b00011: dataA<= reg_3;
    5'b00100: dataA<= reg_4;
    5'b00101: dataA<= reg_5;
    5'b00110: dataA<= reg_6;
    5'b00111: dataA<= reg_7;
    5'b01000: dataA<= reg_8;
    5'b01001: dataA<= reg_9;
    5'b01010: dataA<= reg_10;
    5'b01011: dataA<= reg_11;
```

```
    5'b01100: dataA<= reg_12;
    5'b01101: dataA<= reg_13;
    5'b01110: dataA<= reg_14;
    5'b01111: dataA<= reg_15;
    5'b10000: dataA<= reg_16;
    5'b10001: dataA<= reg_17;
    5'b10010: dataA<= reg_18;
    5'b10011: dataA<= reg_19;
    5'b10100: dataA<= reg_20;
    5'b10101: dataA<= reg_21;
    5'b10110: dataA<= reg_22;
    5'b10111: dataA<= reg_23;
    5'b11000: dataA<= reg_24;
    5'b11001: dataA<= reg_25;
    5'b11010: dataA<= reg_26;
    5'b11011: dataA<= reg_27;
    5'b11100: dataA<= reg_28;
    5'b11101: dataA<= reg_29;
    5'b11110: dataA<= reg_30;
    5'b11111: dataA<= reg_31;
  endcase
end

//always @(rs2, rs1, rd)
always @(negedge clk)
begin
  //rs22 <= rs2;
  case (rs2)
    5'b00000: dataB<= 32'h00000000;
    5'b00001: dataB<= reg_1;
    5'b00010: dataB<= reg_2;
    5'b00011: dataB<= reg_3;
    5'b00100: dataB<= reg_4;
```

```
    5'b00101: dataB<= reg_5;
    5'b00110: dataB<= reg_6;
    5'b00111: dataB<= reg_7;
    5'b01000: dataB<= reg_8;
    5'b01001: dataB<= reg_9;
    5'b01010: dataB<= reg_10;
    5'b01011: dataB<= reg_11;
    5'b01100: dataB<= reg_12;
    5'b01101: dataB<= reg_13;
    5'b01110: dataB<= reg_14;
    5'b01111: dataB<= reg_15;
    5'b10000: dataB<= reg_16;
    5'b10001: dataB<= reg_17;
    5'b10010: dataB<= reg_18;
    5'b10011: dataB<= reg_19;
    5'b10100: dataB<= reg_20;
    5'b10101: dataB<= reg_21;
    5'b10110: dataB<= reg_22;
    5'b10111: dataB<= reg_23;
    5'b11000: dataB<= reg_24;
    5'b11001: dataB<= reg_25;
    5'b11010: dataB<= reg_26;
    5'b11011: dataB<= reg_27;
    5'b11100: dataB<= reg_28;
    5'b11101: dataB<= reg_29;
    5'b11110: dataB<= reg_30;
    5'b11111: dataB<= reg_31;
  endcase
end
//reg flag;
assign reg_0= 32'h00000000;
//always @(dataD)
always @(posedge clk)
```

```
begin
 // flag<=0;
  case (rd)
    //5'b00000: reg_0= 32'h00000000;
    5'b00001: reg_1= dataD;
    5'b00010: reg_2= dataD;
    5'b00011: reg_3= dataD;
    5'b00100: reg_4= dataD;
    5'b00101: reg_5= dataD;
    5'b00110: reg_6= dataD;
    5'b00111: reg_7= dataD;
    5'b01000: reg_8= dataD;
    5'b01001: reg_9= dataD;
    5'b01010: reg_10= dataD;
    5'b01011: reg_11= dataD;
    5'b01100: reg_12= dataD;
    5'b01101: reg_13= dataD;
    5'b01110: reg_14= dataD;
    5'b01111: reg_15= dataD;
    5'b10000: reg_16= dataD;
    5'b10001: reg_17= dataD;
    5'b10010: reg_18= dataD;
    5'b10011: reg_19= dataD;
    5'b10100: reg_20= dataD;
    5'b10101: reg_21= dataD;
    5'b10110: reg_22= dataD;
    5'b10111: reg_23= dataD;
    5'b11000: reg_24= dataD;
    5'b11001: reg_25= dataD;
    5'b11010: reg_26= dataD;
    5'b11011: reg_27= dataD;
    5'b11100: reg_28= dataD;
    5'b11101: reg_29= dataD;
```

```
    5'b11110: reg_30= dataD;
    5'b11111: reg_31= dataD;
  endcase
end
assign dataD= (RegWen)? wb: dataD;
endmodule
/*
always @(posedge clk)
begin
  if (!rst_n)
    dataD <<= 32'h00000000;
  else if (RegWen) begin
    //flag<=1;
    dataD <<= wb;
  end
  else //begin
    //flag<=0;
    dataD <<= dataD;
end
*/
```

2. Khối Branch_comp

```
module branch_comp (input [31:0] rs1, rs2,
                    input BrUn,
                    output reg BrEq, BrLt);
wire [31:0] rs1_n, rs2_n;
assign rs1_n= (~rs1) +1;
assign rs2_n= (~rs2) +1;


always @(BrUn, rs1, rs2) begin
  if (BrUn==1'b1) begin
    if (rs1 < rs2) begin
```

```
    BrLt = 1'b1;
    BrEq = 1'b0;
  end
  else if (rs1 == rs2) begin
    BrEq= 1'b1;
    BrLt= 1'bx;
  end
  else begin
    BrEq= 1'b0;
    BrLt= 1'b0;
  end
end
else begin
  case ({rs1[31], rs2[31]})
    2'b10: begin              //rs1<0, rs2>0
            BrLt= 1'b1;
            BrEq= 1'b0;
           end
    2'b01: begin              //rs1 >0, rs2 <0
            BrLt= 1'b0;
            BrEq= 1'b0;
           end
    2'b00:   begin            // rs1, rs2 >0
            if (rs1 < rs2) begin
              BrLt = 1'b1;
              BrEq = 1'b0;
            end
            else if (rs1 == rs2) begin
              BrEq= 1'b1;
              BrLt= 1'bx;
            end
            else begin
              BrEq= 1'b0;
```

```
            BrLt= 1'b0;
           end
           end
    2'b11: begin
           if (rs1_n < rs2_n) begin
            BrLt = 1'b0;
            BrEq = 1'b0;
           end
           else if (rs1_n == rs2_n) begin
            BrEq= 1'b1;
            BrLt= 1'bx;
           end
           else begin
            BrEq= 1'b0;
            BrLt= 1'b1;
           end
          end
   endcase
  end
end
endmodule
    module branch_comp (input [31:0] rs1, rs2,
                input BrUn,
                output reg BrEq, BrLt);
wire [31:0] rs1_n, rs2_n;
assign rs1_n= (~rs1) +1;
assign rs2_n= (~rs2) +1;


always @(BrUn, rs1, rs2) begin
  if (BrUn==1'b1) begin
   if (rs1 < rs2) begin
    BrLt = 1'b1;
```

```
    BrEq = 1'b0;
  end
  else if (rs1 == rs2) begin
    BrEq= 1'b1;
    BrLt= 1'bx;
  end
  else begin
    BrEq= 1'b0;
    BrLt= 1'b0;
  end
end
else begin
  case ({rs1[31], rs2[31]})
    2'b10: begin              //rs1<0, rs2>0
            BrLt= 1'b1;
            BrEq= 1'b0;
           end
    2'b01: begin              //rs1 >0, rs2 <0
            BrLt= 1'b0;
            BrEq= 1'b0;
           end
    2'b00:   begin            // rs1, rs2 >0
            if (rs1 < rs2) begin
              BrLt = 1'b1;
              BrEq = 1'b0;
            end
            else if (rs1 == rs2) begin
              BrEq= 1'b1;
              BrLt= 1'bx;
            end
            else begin
              BrEq= 1'b0;
              BrLt= 1'b0;
```

```
            end
            end
        2'b11: begin
            if (rs1_n < rs2_n) begin
              BrLt = 1'b0;
              BrEq = 1'b0;
            end
            else if (rs1_n == rs2_n) begin
              BrEq= 1'b1;
              BrLt= 1'bx;
            end
            else begin
              BrEq= 1'b0;
              BrLt= 1'b1;
            end
          end
    endcase
  end
end
endmodule
```

3. Khối Controller

```
module controller (input [31:0] inst,
                   input BrEq, BrLt,
                   output reg PCsel, RegWen, BrUn, Bsel, Asel, MemRW,
                   output reg [2:0] imm_sel,
                   output reg [3:0] Alu_sel,
                   output reg [1:0] WBsel);


always @(inst, BrEq, BrLt) begin
  case (inst[6:2])
```

```
//R-instruction
5'b01100:  begin
            PCsel = 1'b0;
            imm_sel = 3'b101;
            RegWen = 1'b1;
            //BrUn = 1'b0; //=x
            Asel= 1'b0;
            Bsel= 1'b0;
            MemRW= 1'bx;
            WBsel= 2'b01;
             case ({inst[30], inst[14:12]})
               4'b0000:  Alu_sel= 4'b0000; //ADD
               4'b1000:  Alu_sel= 4'b0001; //SUB
               4'b0001:  Alu_sel= 4'b0010; //SLL
               4'b0010:  Alu_sel= 4'b0011; //SLT
               4'b0011:  Alu_sel= 4'b0100; //SLTU
               4'b0100:  Alu_sel= 4'b0101; //XOR
               4'b0101:  Alu_sel= 4'b0110; //SRL
               4'b1101:  Alu_sel= 4'b0111; //SRA
               4'b0110:  Alu_sel= 4'b1000; //OR
               4'b0111:  Alu_sel= 4'b1001; //AND
               default:  Alu_sel= 4'bxxxx;
             endcase
           BrUn= (Alu_sel==4'b0100);
          end

 //I-instruction (arithmetic)
 5'b00100: begin
            PCsel = 1'b0;
            imm_sel = 3'b000;
            RegWen = 1'b1;
            //BrUn = 1'b0;//=x
            Asel= 1'b0;
```

```
            Bsel= 1'b1;
            MemRW= 1'bx;
            WBsel= 2'b01;
            casex ({inst[30], inst[14:12]})
              4'bx000: Alu_sel= 4'b0000;
              4'bx010: Alu_sel= 4'b0011;
              4'bx011: Alu_sel= 4'b0100;
              4'bx100: Alu_sel= 4'b0101;
              4'bx110: Alu_sel= 4'b1000;
              4'bx111: Alu_sel= 4'b1001;
              4'b0001: Alu_sel= 4'b0010;
              4'b0101: Alu_sel= 4'b0110;
              4'b1101: Alu_sel= 4'b0111;
            endcase
            BrUn= (Alu_sel==4'b0100);
          end

//I-instruction (load-LW)
5'b00000: begin
            PCsel = 1'b0;
            imm_sel = 3'b000;
            RegWen = 1'b1;
            BrUn = 1'b0; //=x
            Alu_sel= 4'b0000;
            Asel= 1'b0;
            Bsel= 1'b1;
            MemRW= 1'b0;
            WBsel= 2'b00;
          end

//S-instruction (store-SW)
5'b01000: begin
            PCsel = 1'b0;
```

```
            imm_sel = 3'b001;
            RegWen = 1'b0;
            BrUn = 1'b0;//=x
            Alu_sel= 4'b0000;
            Asel= 1'b0;
            Bsel= 1'b1;
            MemRW= 1'b1;
            WBsel= 2'b00;//=xx
          end

//B-instruction
5'b11000: begin
            case (inst[14:12])
              3'b000: begin //BEQ
                        imm_sel=3'b010;
                        RegWen= 1'b0;
                        BrUn= 1'b0;//=x
                        Asel= 1'b1;
                        Bsel= 1'b1;
                        Alu_sel=4'b0000;
                        MemRW= 1'bx;
                        WBsel= 2'b00;//=xx
                        if (BrEq)
                          PCsel=1'b1;
                        else
                          PCsel= 1'b0;
                      end
              3'b001: begin //BNE
                        imm_sel=3'b010;
                        RegWen= 1'b0;
                        BrUn= 1'b0;//=x
                        Asel= 1'b1;
                        Bsel= 1'b1;
```

```
            Alu_sel=4'b0000;
            MemRW= 1'bx;
            WBsel= 2'b00;//=xx
            if (BrEq)
              PCsel=1'b0;
            else
              PCsel= 1'b1;
          end

3'b100: begin //BLT
            imm_sel=3'b010;
            RegWen= 1'b0;
            BrUn= 1'b0;
            Asel= 1'b1;
            Bsel= 1'b1;
            Alu_sel=4'b0000;
            MemRW= 1'bx;
            WBsel= 2'b00;//=xx
            if (BrLt)
              PCsel= 1'b1;
            else
              PCsel= 1'b0;
          end
3'b110: begin //BLTU
            imm_sel=3'b010;
            RegWen= 1'b0;
            BrUn= 1'b1;
            Asel= 1'b1;
            Bsel= 1'b1;
            Alu_sel=4'b0000;
            MemRW= 1'bx;
            WBsel= 2'b00;//=xx
            if (BrLt)
```

```
            PCsel= 1'b1;
          else
            PCsel= 1'b0;
        end
    3'b101: begin //BGE
          imm_sel=3'b010;
          RegWen= 1'b0;
          BrUn= 1'b0;
          Asel= 1'b1;
          Bsel= 1'b1;
          Alu_sel=4'b0000;
          MemRW= 1'bx;
          WBsel= 2'b00;//=xx
          if (BrLt)
            PCsel= 1'b0;
          else
            PCsel= 1'b1;
        end
    3'b111: begin //BGEU
          imm_sel=3'b010;
          RegWen= 1'b0;
          BrUn= 1'b1;
          Asel= 1'b1;
          Bsel= 1'b1;
          Alu_sel=4'b0000;
          MemRW= 1'bx;
          WBsel= 2'b00;//=xx
          if (BrLt)
            PCsel= 1'b0;
          else
            PCsel= 1'b1;
        end
  endcase
```

```
        end
    //U-instruction (LUI)
    5'b01101:  begin
                PCsel = 1'b0;
                imm_sel = 3'b011;
                RegWen = 1'b1;
                BrUn = 1'b0;//=x
                Alu_sel= 4'b1010;
                Asel= 1'b0; //=x
                Bsel= 1'b1;
                MemRW= 1'bx;
                WBsel= 2'b01;
              end
    //U-instruction (AUIPC)
    5'b00101:  begin
                PCsel = 1'b0;
                imm_sel = 3'b011;
                RegWen = 1'b1;
                BrUn = 1'b0;//=x
                Alu_sel= 4'b0000;
                Asel= 1'b1;
                Bsel= 1'b1;
                MemRW= 1'bx;
                WBsel= 2'b01;
              end
     //J-instruction (JAL)
     5'b11011: begin
                PCsel = 1'b1;
                imm_sel = 3'b100;
                RegWen = 1'b1;
                BrUn = 1'b0;//=x
                Alu_sel= 4'b0000;
                Asel= 1'b1;
```

```
                Bsel= 1'b1;
                MemRW= 1'bx;
                WBsel= 2'b10;
              end

        //J-instruction (JARL)
        5'b11001: begin
                if (inst[14:12]==3'b000) begin
                  PCsel = 1'b1;
                  imm_sel = 3'b000;
                  RegWen = 1'b1;
                  BrUn = 1'b0;//=x
                  Alu_sel= 4'b0000;
                  Asel= 1'b0;
                  Bsel= 1'b1;
                  MemRW= 1'bx;
                  WBsel= 2'b10;
                 end
              end
   endcase
 end
 endmodule
```

4. Alu

```
module alu (input [31:0] in1, in2,
          input [3:0] alu_sel,
          output reg [31:0] alu_out);

//Find max{in1, in2} for SLT, SLTU
wire lt1, lt2, eq1, eq2;
branch_comp c1 (.rs1(in1), .rs2(in2), .BrUn(1'b0), .BrEq(eq1), .BrLt(lt1));
```

```
branch_comp c2 (.rs1(in1), .rs2(in2), .BrUn(1'b1), .BrEq(eq2), .BrLt(lt2));

/*LUI, adding wwith signed 12-bit number
wire [31:0] num_s;
assign num_s= 32'h00001000;
wire special;
assign special= (alu_sel==4'b1010);*/


always @(alu_sel, in1, in2, eq1, eq2, lt1, lt2) begin
  case (alu_sel)
    4'b0000: alu_out= in1 + in2;
            /*begin
              if (special==1'b0)
                alu_out= in1 + in2;
              else
                alu_out= in1 - num_s + in2;
            end*/

    4'b0001: alu_out= in1 - in2;

    4'b0010: alu_out= in1 << in2;

    4'b0011: begin
              if (eq1)
                alu_out= 32'h00000000;
              else if (lt1)
                alu_out= 32'hffffffff;
              else
                alu_out= 32'h00000000;
            end

    4'b0100: begin
```

```
            if (eq2)
              alu_out= 32'h00000000;
            else if (lt2)
              alu_out= 32'hffffffff;
            else
              alu_out= 32'h00000000;
          end

      4'b0101: alu_out = in1 ^ in2;

      4'b0110: alu_out= in1 >> in2;

      4'b0111: alu_out= in1 >>> in2;

      4'b1000: alu_out= in1 | in2;

      4'b1001: alu_out= in1 & in2;

      4'b1010: alu_out= in2; // for LUI

      default: alu_out= 32'hxxxxxxxx;
    endcase
  end
  endmodule
```

5. Module Tổng của thiết kế

```
module cpu_top (input clk, rst_n);

  //I/O PCmux
  wire [31:0] pc_add4, pc_final, pc_right;
  wire [31:0] inst,inst2;
  wire  pc_sel, pc_src,flush;
  wire [31:0] pc_branch; //addsum
  wire [31:0] pc_out,pc_out3;
```

```
  wire pc_sel_out;
  wire [31:0] predict_pc;
    prediction prediction (.clk(clk), .rst_n(rst_n), .pc(pc_out), .inst(inst), .pc_sel(flush),
.pc_sel_out(pc_sel_out), .predict_pc(predict_pc));

    mux_predict mux_predict (.pc(pc_add4), .predict(predict_pc),
.prediction(pc_sel_out), .pc_mux_out(pc_final));

  //I/O PC

  wire pc_rewrite;
    PC PC (.clk(clk), .rst_n(rst_n), .pc_in(pc_final), .pc_out(pc_out),
.pc_rewrite(pc_rewrite), .flush(flush),.pc_right(pc_right));

    pc_add4 pc_add (.pc_in(pc_out), .pc_out(pc_add4));

  //I/O IMEM

    IMEM IMEM (.PC(pc_out), .inst(inst));

  /////*IF_to_ID*////
  wire [31:0] pc_out1, pc_add4_1, pc_add4_2, pc_add4_3, pc_add4_4;
  wire [31:0] inst1;

  wire stage1_rewrite;
  IF_to_ID stage1 (.clk(clk), .rst_n(rst_n), .pc_in(pc_out), .inst_in(inst),
.pc_out(pc_out1), .inst_out(inst1), .pc_add4(pc_add4), .pc_add4_out(pc_add4_1),
                .stage1_rewrite(stage1_rewrite), .flush(flush));

  ////*ID_to_EX*////
   wire [31:0] wb, dataA, dataB; //I/O regs
   wire [31:0] imm_out; //I/O imm gen
   wire B_sel, B_sel_out; // mux B
```

```
  wire A_sel;
  wire MemRW; //DMEM
  wire [1:0] WBsel;
  //wire RegWen;
  wire [3:0] alu_sel; //alu
wire branch;
wire branch_out1, alu_src_out, MemRW_out1, RegWen_out1;
wire [3:0] alu_op_out;
wire [1:0] Wbsel_out1;
wire [4:0] rd_out, rs1_out, rs2_out;
wire [31:0] pc_out2, data1_out, data2_out, imm_out1;
//assign branch= inst1[6]&inst[5]&(~inst[4])&(~inst[3])&(~inst[2]);
wire stall;
hazard_detection hazard_detection (.rd(rd_out), .rs1(inst1[19:15]), .rs2(inst1[24:20]),
.MemRead(MemRW_out1), .RegWen(RegWen_out1),
                        .pc_rewrite(pc_rewrite), .stage1_rewrite(stage1_rewrite),
.stall(stall));


wire [3:0] alu_op_outx;
wire [1:0]  Wbsel_outx;
wire branch_outx, MemRW_outx, RegWen_outx;
 mux_hazard mux_hazard (.stall(stall), .alu_op(alu_sel), .branch(pc_sel),
.MemRW(MemRW), .RegW(RegWen), .MemReg(WBsel),
                .alu_op_out(alu_op_outx), .branch_out(branch_outx),
.MemRW_out(MemRW_outx), .RegW_out(RegWen_outx),
.MemReg_out(Wbsel_outx));


 ID_to_EX stage2 (.clk(clk), .rst_n(rst_n), .pc_in(pc_out1), .data1_in(dataA),
.data2_in(dataB), .imm_in(imm_out), .rd(inst1[11:7]), .rs1(inst1[19:15]),
.rs2(inst1[24:20]),
```

```
        .pc_add4(pc_add4_1), .pc_add4_out(pc_add4_2), .B_sel_in(B_sel),
.flush(flush), .inst(inst1),
        .alu_op(alu_op_outx), .branch(branch_outx), .memRW(MemRW_outx),
.RegW(RegWen_outx), .MemReg(Wbsel_outx),
        .alu_op_out(alu_op_out), .branch_out(branch_out1),
.B_sel_out(B_sel_out),
        .memRW_out(MemRW_out1), .RegW_out(RegWen_out1),
.MemReg_out(Wbsel_out1), .rd_out(rd_out), .rs1_out(rs1_out), .rs2_out(rs2_out),
        .pc_out(pc_out2), .data1_out(data1_out), .data2_out(data2_out),
.imm_out(imm_out1),.inst_out(inst2));

addsum addsum (.pc(pc_out2), .imm(imm_out1), .pc_branch(pc_branch));

mux2to1 mux (.pc_branch(pc_branch),. pc_out1(pc_out1),.inst(inst2),
.flush(flush),.pc_right(pc_right),.branch(branch_out1));
/////*EX_to_MEM*////
wire branch_out2, MemRW_out2, RegWen_out2;
wire [1:0] Wbsel_out2;
wire [4:0] rd_out1;
wire [31:0] data_out2, alu_out1, alu_out2;
EX_to_MEM stage3  (.clk(clk), .rst_n(rst_n), .pc_branch(pc_branch),
.data_in(data2_out), .alu_in(alu_out1), .rd(rd_out), .pc_add4(pc_add4_2),
.pc_add4_out(pc_add4_3),
        .branch(branch_out1), .memRW(MemRW_out1), .RegW(RegWen_out1),
.MemReg(Wbsel_out1),.flush(flush),
        .branch_out(pc_src), .memRW_out(MemRW_out2),
        .RegW_out(RegWen_out2), .MemReg_out(Wbsel_out2),
        .pc_branch_out(pc_out3), .data_out(data_out2), .alu_out(alu_out2),
.rd_out(rd_out1));

////*MEM_to_WB*////
wire RegWen_out3;
wire [1:0] Wbsel_out3;
```

```
wire [4:0] rd_out2;
wire [31:0] data_out3, alu_out3;
wire [31:0] dataR; //DMEM output
MEM_to_WB stage4 ( .clk(clk), .rst_n(rst_n), .alu_in(alu_out2), .dataR_in(dataR),
.rd(rd_out1), .pc_add4(pc_add4_3), .pc_add4_out(pc_add4_4),
                .RegW(RegWen_out2), .MemReg(Wbsel_out2),
                .RegW_out(RegWen_out3), .MemReg_out(Wbsel_out3),
                .dataR_out(data_out3), .alu_out(alu_out3), .rd_out(rd_out2));

///*FORWARDING_UNIT*///
wire [1:0] fa, fb;
forwarding_unit fw (.EX_MEM_rd(rd_out1), .MEM_WB_rd(rd_out2),
.RegW_in1(RegWen_out2), .RegW_in2(RegWen_out3), .rs1(rs1_out), .rs2(rs2_out),
.fa(fa), .fb(fb), .B_sel(B_sel_out));

//I/O mux_A
  wire [31:0] muxA_out;
    mux_A mux_A (.A_sel(fa), .wb(wb), .alu_out(alu_out2), .rs1(data1_out),
.muxA_out(muxA_out));


  //I/O mux_B

  wire [31:0] muxB_out;
    mux_B mux_B (.B_sel(fb), .wb(wb), .alu_out(alu_out2), .rs2(data2_out),
.imm_in(imm_out1), .muxB_out(muxB_out));
  //I/O regs


    regs regs (.clk(clk), .rst_n(rst_n), .rs1(inst1[19:15]), .rs2(inst1[24:20]), .rd(rd_out2),
.RegWen(RegWen_out3), .wb(wb), .dataA(dataA), .dataB(dataB));

  //I/O imm_gen
```

```
  wire [2:0] imm_sel;

    imm_gen imm_gen (.imm_in(inst1[31:7]), .imm_sel(imm_sel),
.imm_out(imm_out));

  //I/O branch_comp
  wire BrUn, BrLt, BrEq;

    branch_comp branch_comp (.rs1(dataA), .rs2(dataB), .BrUn(BrUn), .BrEq(BrEq),
.BrLt(BrLt));



  //I/O alu

    alu alu (.in1(muxA_out), .in2(muxB_out), .alu_sel(alu_op_out),
.alu_out(alu_out1));



  //I/O DMEM
  wire [31:0] dataWr;
  muxBB muxBB (.fb(fb), .dataB(data_out2), .muxB_out(muxB_out),
.dataW(dataWr));

  DMEM DMEM (.clk(clk), .rst_n(rst_n), .dataW(dataWr), .Addr(alu_out2),
.MemRW(MemRW_out2), .dataR(dataR));
  //DMEM_2 DMEM (.clk(clk), .rst_n(rst_n), .dataW(dataB), .Addr(alu_out[4:0]),
.MemRW(MemRW), .dataR(dataR));

  //I/O mux_W

    mux_W mux_W (.mem(data_out3), .alu(alu_out3), .pc_add4(pc_add4_4),
.WB_sel(Wbsel_out3), .wb(wb));
```

```
controller controller (.inst(inst1), .BrEq(BrEq), .BrLt(BrLt), .PCsel(pc_sel),
                                                    .RegWen(RegWen), .BrUn(BrUn),
                    .Bsel(B_sel), .Asel(A_sel), .MemRW(MemRW),
                                                    .imm_sel(imm_sel), .Alu_sel(alu_sel),
.WBsel(WBsel));
endmodule
```