

TRƯỜNG ĐẠI HỌC BÁCH KHOA TP. HỒ CHÍ MINH

Khoa Điện – Điện tử

Bộ môn Điện tử



BÁO CÁO BÀI TẬP LỚN

CẤU TRÚC MÁY TÍNH

THIẾT KẾ CPU DỰA TRÊN CẤU TRÚC RV32

GVHD: Trần Hoàng Linh

Sinh viên thực hiện:

Đinh Thế Bảo

MSSV: 1510152

Tp. Hồ Chí Minh, tháng 12 năm 201

I. Mục tiêu

Thực hiện thiết kế 1 CPU chạy đơn chu kỳ và pipeline dựa trên cấu trúc RISC-V

II. Lý thuyết

2.1. Cấu trúc lệnh của CPU RV32

CPU được xây dựng dựa trên cấu trúc RV32 có tổng cộng 32 lệnh hợp ngữ, với mỗi lệnh có độ dài 32 bit và có 7 bit opcode [6:0] để xác định được loại lệnh.

Các lệnh hợp ngữ cơ bản của RV32 có cấu trúc như sau:

RV32I Base Instruction Set							
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

Hình 2.1.1: Cấu trúc của các lệnh cơ bản của RV32

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 bit và có 32 thanh ghi từ $x_0 \div x_{31}$. Như vậy, ta cần có 5 bit 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:

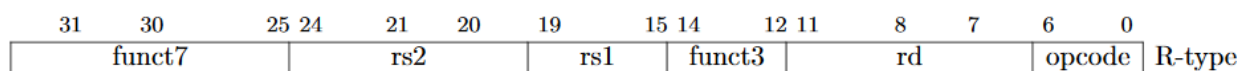
Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	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

Hình 2.1.2: Chức năng của các thanh ghi.

Với giá trị của thanh ghi x0 luôn là 0.

2.2. Nhóm lệnh R (R-format)

Nhóm lệnh R có cấu trúc tổng quát:



Hình 2.2.1: Cấu trúc nhóm lệnh loại R.

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

Hình 2.2.2: Các lệnh thuộc nhóm lệnh R.

Với opcode là 7'b0110011.

Nhóm lệnh loại R có nhiệm vụ 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.

2.3. Nhóm lệnh I

Nhóm lệnh I có cấu trúc tổng quát:

imm[11:0]	rs1	funct3	rd	opcode	I-type
-----------	-----	--------	----	--------	--------

Hình 2.3.1: Cấu trúc nhóm lệnh loại I.

Nhóm lệnh I thực hiện hai chức năng là tính toán và lấy dữ liệu.

➤ Chức năng 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

Hình 2.3.2: Các lệnh nhóm I thực hiện chức năng tính toán.

Với nhóm lệnh I thực hiện chức năng tính toán, ta có opcode là 7'b0010011.

Nhóm lệnh I thực hiện chức năng tính toán (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.

➤ Lấy dữ liệu

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

Hình 2.3.3: Các lệnh nhóm I thực hiện chức năng lấy dữ liệu.

Với nhóm lệnh I thực hiện chức năng lấy dữ liệu, ta có opcode là 7'b0000011.

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.

2.4. Nhóm lệnh S

Nhóm lệnh loại S có cấu trúc tổng quát:

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
-----------	-----	-----	--------	----------	--------	--------

Hình 2.4.1: Cấu trúc tổng quát các lệnh nhóm S.

Nhóm lệnh S có opcode là 7'b0100011.

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

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

Hình 2.4.3: Các lệnh nhóm S

2.5. Nhóm lệnh B

Nhóm lệnh loại B có cấu trúc tổng quát:

imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
---------	-----------	-----	-----	--------	----------	---------	--------	--------

Hình 2.5.1: Cấu trúc tổng quát nhóm lệnh loại B

Nhóm lệnh B có opcode là 7'b1100011.

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, ...).

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

Hình 2.5.2: Các lệnh nhóm B

2.6. Nhóm lệnh U

Nhóm lệnh U có cấu trúc tổng quát:

imm[31:12]	rd	opcode	U-type
------------	----	--------	--------

Hình 2.6.1: Cấu trúc tổng quát các lệnh nhóm U.

Nhóm lệnh U gồm 2 lệnh LUI và AUIPC.

➤ Với lệnh LUI

imm[31:12]	rd	0110111	LUI
------------	----	---------	-----

Hình 2.6.2: Cấu trúc lệnh LUI.

Nhóm lệnh B có opcode là 7'b0110111.

Lệnh này load giá trị imm [31:12] vào thanh ghi rd.

➤ Với lệnh AUIPC

imm[31:12]	rd	0010111	AUIPC
------------	----	---------	-------

Hình 2.6.3: Cấu trúc lệnh AUIPC.

Nhóm lệnh B có opcode là 7'b0010111

Lệnh này load giá trị ở PC vào thanh ghi rd

2.7. Nhóm lệnh J

Nhóm lệnh J gồm 2 lệnh JAL và JALR.

➤ Với lệnh JAL

imm[20 10:1 11 19:12]	rd	1101111	JAL
-----------------------	----	---------	-----

Hình 2.7.1: Cấu trúc lệnh JAL.

Lệnh JAL có opcode là 7'b1101111.

➤ Với lệnh JALR

imm[11:0]	rs1	000	rd	1100111	JALR
-----------	-----	-----	----	---------	------

Hình 2.7.2: Cấu trúc lệnh JALR

Lệnh JALR có opcode là 7'b1100111.

2.8. Biến đổi Immediates

Các giá trị bit Immediates trước khi đưa vào bộ ALU để thực hiện tính toán, sẽ được mở rộng dấu ở khối ImmGen theo phương thức:

31	30	25	24	21	20	19	15	14	12	11	8	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]		imm[10:5]			rs2			rs1		funct3		imm[4:1]		imm[11]		opcode	B-type
imm[31:12]										rd			opcode		U-type		
imm[20]		imm[10:1]			imm[11]		imm[19:12]			rd			opcode		J-type		

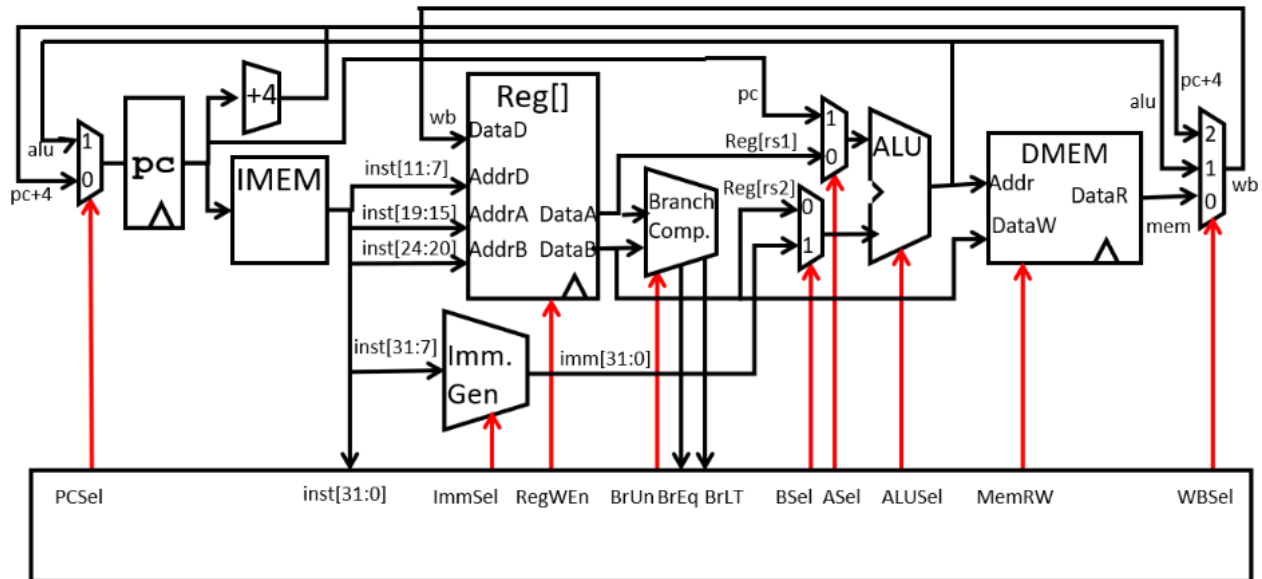
Hình 2.8.1: Cấu trúc các nhóm lệnh.

31	30	20	19	12	11	10	5	4	1	0	
— inst[31] —						inst[30:25]	inst[24:21]	inst[20]	I-immediate		
— inst[31] —						inst[30:25]	inst[11:8]	inst[7]	S-immediate		
— inst[31] —					inst[7]	inst[30:25]	inst[11:8]	0	B-immediate		
inst[31]	inst[30:20]			inst[19:12]		— 0 —					U-immediate
— inst[31] —				inst[19:12]		inst[20]	inst[30:25]	inst[24:21]	0	J-immediate	

Hình 2.8.2: Giá trị cách thức mở rộng Immediate.

III. Thực hiện thiết kế CPU

Dựa vào sơ đồ khối của CPU:



Hình 3.0.1: Sơ đồ khối CPU.

Dựa vào phương pháp chia để trị, ta sẽ thực hiện thiết kế từng khối riêng lẻ và ghép chúng lại.

3.1. Khối pc

Khối pc sẽ tạo ra một thanh ghi 32 bit, khi giá trị bit rst = 0, thanh ghi pc sẽ có giá trị 32'h00000000 và chương trình sẽ bắt đầu chạy.

Code:

```
module PC (pc_in, clk, rst, pc_out);
input [31:0] pc_in;
input clk, rst;
output reg [31:0] pc_out;
reg [31:0] temp;
//assign pc_out = temp;
always @ (posedge clk)
begin
    if(rst)
    begin
        pc_out = 32'b0;
        temp =32'b0;
    end
    else
    begin
        temp = pc_in;
        pc_out = temp;
    end
end
```

```
end  
endmodule
```

3.2. Khối pc+4

Khối này sẽ thực hiện đem giá trị pc_out cộng thêm 4, sau đó đưa vào khối mux2 để thực hiện lựa chọn.

Code:

```
module cong_4(in, out);  
  //input clk, rst;  
  input [31:0] in;  
  output [31:0] out;  
  //reg [31:0] temp;  
  //always @ (posedge clk)  
  assign out = in+4;  
endmodule
```

3.3. Khối mux2

Khối mux2 sẽ dựa vào tín hiệu điều khiển pc_sel để xác định ngõ vào pc_in là kết quả của pc+4 hay alu.

Code:

```
module mux2(out, in0, in1, sel);  
  parameter WIDTH_DATA_LENGTH = 32;  
  output [WIDTH_DATA_LENGTH-1:0] out;  
  input [WIDTH_DATA_LENGTH-1:0] in0, in1;  
  input sel;  
  assign out = (sel)? in1:in0;  
endmodule
```

3.4. Khối IMEM

Khối IMEM sẽ thực hiện lấy giá trị hex tương ứng với câu lệnh để thực thi theo giá trị pc nhận được.

Code:

```
/*  
  ***@Author: NGUYEN TRUNG HIEU  
  ***@Date:   Nov 1st, 2018  
*/  
  
/***** Instruction Memory Block *****/  
module IMEM (inst, PC);  
  parameter INST_WIDTH_LENGTH = 32;
```

```

parameter    PC_WIDTH_LENGTH = 32;
parameter    MEM_WIDTH_LENGTH = 32;
parameter    MEM_DEPTH = 1<<18;
output reg    [INST_WIDTH_LENGTH-1:0]inst;
input         [PC_WIDTH_LENGTH-1:0]PC;

/***** Instruction Memmory *****/
reg          [MEM_WIDTH_LENGTH-1:0]IMEM[0:MEM_DEPTH-1];

wire         [17:0]pWord;
wire         [1:0]pByte;

assign       pWord = PC [19:2];
assign       pByte = PC [1:0];

initial begin
$readmemh ("D:/CauTruc/RISCV_project/giaithua.txt", IMEM);
end

always@(PC)
begin
    if (pByte == 2'b00)
        inst <= IMEM[pWord];
    else
        inst <= 'hz;
end

endmodule

```

3.5. Khối ImmGen

Khối ImmGen có nhiệm vụ thực hiện mở rộng giá trị Immediate nhận được thành 32 bit.

Code:

```

module ImmGenV3 (clk, rst, imm_in, imm_out, imm_sel);

    input clk, rst;
    input [31:0] imm_in;
    input [2:0] imm_sel;
    output [31:0] imm_out;

    reg [31:0] temp_out;

    parameter i_format = 3'b000;
    parameter s_format = 3'b001;
    parameter b_format = 3'b010;

```

```

parameter u_format = 3'b011;
parameter j_format = 3'b100;
parameter unsigned_format = 3'b101;

always @(imm_sel or imm_in)
/*if (rst) begin
    temp_out = 32'h0;
end
else*/
begin

    case (imm_sel)
        i_format: begin
            temp_out[31:11] = 21'h1FFFFFF * imm_in[31];
            temp_out[10:5] = imm_in[30:25];
            temp_out[4:1] = imm_in[24:21];
            temp_out[0] = imm_in[20];
        end

        s_format: begin
            temp_out[31:11] = 21'h1FFFFFF * imm_in[31];
            temp_out[10:5] = imm_in[30:25];
            temp_out[4:1] = imm_in[11:8];
            temp_out[0] = imm_in[7];
        end

        b_format: begin
            temp_out[31:12] = 20'hFFFFFF * imm_in[31];
            temp_out[11] = imm_in[7];
            temp_out[10:5] = imm_in[30:25];
            temp_out[4:1] = imm_in[11:8];
            temp_out[0] = 1'b0;
        end

        u_format: begin
            temp_out[31] = imm_in[31];
            temp_out[30:20] = imm_in[30:20];
            temp_out[19:12] = imm_in[19:12];
            temp_out[11:0] = 12'h000;
        end

        j_format: begin
            temp_out[31:21] = 12'hFFF * imm_in[31];

```

```

        temp_out[20] = imm_in[31];
        temp_out[19:12] = imm_in[19:12];
        temp_out[11] = imm_in[20];
        temp_out[10:1] = imm_in[30:21];
        temp_out[0] = 0;
    end

    unsigned_format: begin
        temp_out [31:12] = 20'h0;
        temp_out [11:0] = imm_in[31:20];
    end

    default temp_out = 32'hFFFFFFF;

endcase

end

assign imm_out = temp_out;

endmodule

```

3.6. Khối Reg[]

Khối Register có nhiệm vụ ghi (đọc) dữ liệu từ thanh ghi có địa chỉ cho trước.

Code:

```

module register_v2(clk,rst,wre,addressA,addressB,addressD,DataD,DataA,DataB);
input clk,rst,wre;
input [4:0] addressA,addressB,addressD;
input [31:0] DataD;
output reg [31:0] DataA,DataB;
wire clk_n;
reg [31:0] register_address [31:0];
assign clk_n=~clk;
always @(clk)
    begin
        if(rst)
            begin
                DataA=0;
                DataB=0;
                register_address[5'd0] <= 32'b0;
                register_address[5'd1] <= 32'b0;
                register_address[5'd2] <= 32'b0;
                register_address[5'd3] <= 32'b0;
            end
        end
    end

```

```

        register_address[5'd4] <= 32'b0;
        register_address[5'd5] <= 32'b0;
        register_address[5'd6] <= 32'b0;
        register_address[5'd7] <= 32'b0;
        register_address[5'd8] <= 32'b0;
        register_address[5'd9] <= 32'b0;
        register_address[5'd10] <= 32'b0;
        register_address[5'd11] <= 32'b0;
        register_address[5'd12] <= 32'b0;
        register_address[5'd13] <= 32'b0;
        register_address[5'd14] <= 32'b0;
        register_address[5'd15] <= 32'b0;
        register_address[5'd16] <= 32'b0;
        register_address[5'd17] <= 32'b0;
        register_address[5'd18] <= 32'b0;
        register_address[5'd19] <= 32'b0;
        register_address[5'd20] <= 32'b0;
        register_address[5'd21] <= 32'b0;
        register_address[5'd22] <= 32'b0;
        register_address[5'd23] <= 32'b0;
        register_address[5'd24] <= 32'b0;
        register_address[5'd25] <= 32'b0;
        register_address[5'd26] <= 32'b0;
        register_address[5'd27] <= 32'b0;
        register_address[5'd28] <= 32'b0;
        register_address[5'd29] <= 32'b0;
        register_address[5'd30] <= 32'b0;
        register_address[5'd31] <= 32'b0;
    end
    else
        DataA <= register_address[addressA];
        DataB <= register_address[addressB];
    end
always @(posedge clk_n)
    begin
        if(wre)
            begin
                register_address[addressD]<=DataD;
            end
        end
    end
endmodule

```

3.7. Khối Brach Comp

Khối Brach Comp có nhiệm vụ thực hiện so sánh hai giá trị, được lấy ra từ thanh ghi, bao gồm 2 trường hợp là có dấu và không dấu.

Code:

```
module branchComp (clk_1, rst, rs1_out, rs2_out, br_un, br_eq, br_lt);

    input clk_1, rst;
    input br_un;
    input [31:0] rs1_out, rs2_out;
    output wire br_eq;
    output wire br_lt;

    //Temporary register
    reg temp_eq;
    reg temp_lt;
    wire rs1_signed, rs2_signed;
    wire [30:0] temp_rs1_out;
    wire [30:0] temp_rs2_out;
    assign br_eq = temp_eq;
    assign br_lt = temp_lt;

    //always @(temp_rs2_out or temp_rs1_out or rs2_out or rs1_out or rs1_signed or rs2_signed) begin
    always @(temp_rs2_out or temp_rs1_out or rs2_out or rs1_out or rs1_signed or rs2_signed) begin
        assign rs1_signed = rs1_out[31];
        assign rs2_signed = rs2_out[31];
        assign temp_rs1_out = rs1_out[30:0];
        assign temp_rs2_out = rs2_out[30:0];
    //end

    always @ (posedge clk_1)

    //Unsigned
    if (br_un == 1) begin

        if (rs1_out == rs2_out) begin
            temp_eq = 1'b1;
            temp_lt = 1'b0;
        end

        else begin
            temp_eq = 1'b0;

            if (rs1_out < rs2_out) begin
```

```

        temp_lt = 1'b1;
    end

    else begin
        temp_lt = 1'b0;
    end

end

end

//Signed
else if (br_un == 0) begin
    //rs1_out, rs2_out > 0 || rs1_out > 0, rs2_out < 0 || rs1_out < 0, rs2_out
    t > 0 || rs1_out, rs2_out < 0

    //rs1_out > 0
    if (rs1_signed == 0) begin

        //rs2_out > 0
        if (rs2_signed == 0) begin

            if (temp_rs1_out == temp_rs2_out) begin
                temp_eq = 1'b1;
                temp_lt = 1'b0;
            end

            else begin
                temp_eq = 1'b0;

                if (temp_rs1_out < temp_rs2_out) begin
                    temp_lt = 1'b1;
                end

                else begin
                    temp_lt = 1'b0;
                end

            end

        end

    end

    //rs2_out < 0
    else begin
        temp_eq = 1'b0;
    end
end

```



```

        temp_lt = 1'b0;
    end

end

//rs1_out < 0
else if (rs1_signed == 1) begin

    //rs2_out > 0
    if (rs2_signed == 0) begin
        temp_eq = 1'b0;
        temp_lt = 1'b1;
    end

    //rs2_out < 0
    else begin

        if (temp_rs1_out == temp_rs2_out) begin
            temp_eq = 1'b1;
            temp_lt = 1'b0;
        end

        else begin
            temp_eq = 1'b0;

            if (temp_rs1_out < temp_rs2_out) begin
                temp_lt = 1'b0;
            end

            else begin
                temp_lt = 1'b1;
            end
        end
    end
end

end

end

endmodule

```

3.8. Khối Br_un_set_unit

Khối này là một phần nằm trong khối control, thực hiện nhiệm vụ xác định 2 giá trị khối brach comp cần so sánh là giá trị có dấu hay không dấu để đưa ra tín hiệu điều khiển thích hợp.

Code:

```
module br_un_set_unit (rst, inst, br_un);
    input rst;
    input [32:0] inst;
    output br_un;

    wire [7:0] control;
    reg temp_br_un;

    parameter inst_beq = 8'b11000000;
    parameter inst_bne = 8'b11000001;
    parameter inst_blt = 8'b11000100;
    parameter inst_bge = 8'b11000101;
    parameter inst_bltu = 8'b11000110;
    parameter inst_bgeu = 8'b11000111;

    assign control[7:3] = inst[6:2];
    assign control[2:0] = inst[14:12];
    assign br_un = temp_br_un;

    always @(control)
        begin
            case (control)
                inst_beq: temp_br_un = 1'b0;
                inst_bne: temp_br_un = 1'b0;
                inst_blt: temp_br_un = 1'b0;
                inst_bge: temp_br_un = 1'b0;
                inst_bltu: temp_br_un = 1'b1;
                inst_bgeu: temp_br_un = 1'b1;
                default: temp_br_un = 1'bx;
            endcase
        end
endmodule
```

3.9. Khối control

Khối control thực hiện nhiệm vụ kiểm soát chức năng, ngõ ra của các khối mux, ImmGen, BrachComp, ALU, DMEM để CPU có thể hoạt động trơn tru.

Code:

```
module control(clk, inst, BrUn, Br_Eq, Br_LT, PCSel, ImmSel, RegWEn, ASel, BSel,
ALUSel, MemRW, WBSel);
input clk;
input Br_Eq, Br_LT;
input [31:0] inst;
output PCSel;
output wire [2:0] ImmSel;
output RegWEn, BrUn, ASel, BSel;
output [3:0] ALUSel;
output [2:0] MemRW;
output [1:0] WBSel;
wire [10:0] address;
reg [16:0] data;
assign address [10] = inst [30];
assign address [9:7] = inst [14:12];
assign address [6:2] = inst [6:2];
assign address [1] = Br_Eq;
assign address [0] = Br_LT;
assign PCSel = data [16];
assign ImmSel = data [15:13];
assign RegWEn = data [12];
assign BrUn = data [11];
assign BSel = data [10];
assign ASel = data [9];
assign ALUSel = data [8:5];
assign MemRW = data [4:2];
assign WBSel = data [1:0];
always @(address)
begin
    casez(address)
    //R type
    //ADD
    11'b00000110zz : data = 17'b0zzz10000000zzz01;
    //SUB
    11'b10000110zz : data = 17'b0zzz10000001zzz01;
    //SLL
    11'b00010110zz : data = 17'b0zzz10000010zzz01;
    //SLT
    11'b00100110zz : data = 17'b0zzz10000011zzz01;
```

```

//SLTU
11'b001101100zz : data = 17'b0zzz10000100zzz01;
//XOR
11'b010001100zz : data = 17'b0zzz10000101zzz01;
//SRL
11'b010101100zz : data = 17'b0zzz10000110zzz01;
//SRA
11'b110101100zz : data = 17'b0zzz10000111zzz01;
//OR
11'b011001100zz : data = 17'b0zzz10001000zzz01;
//AND
11'b011101100zz : data = 17'b0zzz10001001zzz01;
//I type
//ADDI
11'bz00000100zz : data = 17'b000010100000zzz01;
//SLTI
11'bz01000100zz : data = 17'b000010100011zzz01;
//SLTIU
11'bz01100100zz : data = 17'b000010100100zzz01;
//XORI
11'bz10000100zz : data = 17'b000010100101zzz01;
//ORI
11'bz11000100zz : data = 17'b000010101000zzz01;
//ANDI
11'bz11100100zz : data = 17'b000010101001zzz01;
//SLLI
11'b000100100zz : data = 17'b000010100010zzz01;
//SRLI
11'b010100100zz : data = 17'b000010100110zzz01;
//SRAI
11'b110100100zz : data = 17'b000010100111zzz01;
//LB
11'bz00000000zz : data = 17'b0000101000000000;
//LH
11'bz00100000zz : data = 17'b00001010000000100;
//LW
11'bz01000000zz : data = 17'b00001010000001000;
//LBU
11'bz10000000zz : data = 17'b01011010000000000;
//LHU
11'bz10100000zz : data = 17'b01011010000000100;
// S type
//SB
11'bz00001000zz : data = 17'b000100100000011zz;
//SH

```

```

        11'bz00101000zz : data = 17'b000100100000100zz;
        //SW
        11'bz01001000zz : data = 17'b000100100000101zz;
//B type
    //BEQ
        11'bz000110001z : data = 17'b101000110000zzzzz;
        11'bz000110000z : data = 17'b001000110000zzzzz;
    //BNE
        11'bz001110000z : data = 17'b101000110000zzzzz;
        11'bz001110001z : data = 17'b001000110000zzzzz;
    //BLT
        11'bz100110000z : data = 17'b001000110000zzzzz;
        11'bz100110001z : data = 17'b101000110000zzzzz;
    //BGE
        11'bz10111000z0z : data = 17'b101000110000zzzzz;
        11'bz10111000z1z : data = 17'b001000110000zzzzz;
    //BLTU
        11'bz110110000z : data = 17'b001000110000zzzzz;
        11'bz110110001z : data = 17'b101000110000zzzzz;
    //BGEU
        11'bz111110000z : data = 17'b101000110000zzzzz;
        11'bz111110001z : data = 17'b001000110000zzzzz;
//U type
    //LUI
        11'bzzzz01101zz : data = 17'b001110100000zzz01;
//J type
    //JAL
        11'bzzzz11011zz : data = 17'b110010110000zzz10;
    //JALR
        11'bz00011001zz : data = 17'b100010100000zzz10;

        endcase
    end
endmodule

```

3.10. Khối ALU

Khối ALU nhận tín hiệu từ khối control từ đó thực hiện phương pháp tính toán tương ứng.

Code:

```

module alu(A,B,ALU_Out,ALU_Sel,CarryOut);
    input [31:0] A,B; // ALU 32-bit Inputs
    input [3:0] ALU_Sel; // ALU Selection
    output [31:0] ALU_Out; // ALU 32-bit Output
    output CarryOut; // Carry Out Flag

```

```

integer i;
wire [31:0] A_1,B_1;
wire [31:0] A_bu2,B_bu2;
reg [31:0] ALU_Result;
assign A_1 = A;
assign B_1 = B;
assign A_bu2 = ~ A +32'b1;
assign B_bu2 = ~ B +32'b1;
assign ALU_Out = ALU_Result;
wire [31:0] y;
assign y = A;
always @(*)
begin
    case(ALU_Sel)
        4'b0000: // Addition
            begin
                ALU_Result = A + B ;
            end
        4'b0001: // Subtraction
            begin
                ALU_Result = A - B ;
            end
        4'b0010: // Logical shift left
            ALU_Result = A<<1;
        4'b0011: // SLT
            begin
                if(A[31]==1)
                    if(B[31]==1)
                        ALU_Result = (A<B)?32'd1:32'd0 ;
                    else ALU_Result = 32'b0;
                else if(B[31]==1)
                    ALU_Result = 32'b1;
                else ALU_Result = (A>B)?32'd1:32'd0 ;
            end
        4'b0100: // SLTU
            ALU_Result = (A>B)?32'd1:32'd0 ;
        4'b0101: // XOR
            ALU_Result = A ^ B;
        4'b0110: // Logical shift rights
            ALU_Result = A>>B;
        4'b0111: // SRA
            ALU_Result = A >>> B;
        4'b1000: // OR
            ALU_Result = A | B;
        4'b1001: // AND

```

```

        ALU_Result = A & B;
    default: ALU_Result = A + B ;
endcase
end
endmodule

```

3.11. Khối DMEM

```

module DMEM(clk,rst,MenRW,address,DataW,DataR);
input clk,rst;
input [2:0] MenRW;
input [31:0] address;
input [31:0] DataW;
output reg [31:0] DataR;
reg [31:0] register_address [31:0];
always @(clk)
    begin
        if(rst)
            begin
                DataR=0;
            end
        else
            begin
                case(MenRW)
                    3'b000:DataR <= register_address[address][7:0];
                    3'b001:DataR <= register_address[address][16:0];
                    3'b010:DataR <= register_address[address][31:0];
                    3'b011:register_address[address]<=DataW[7:0];
                    3'b100:register_address[address]<=DataW[15:0];
                    3'b101:register_address[address]<=DataW[31:0];
                endcase
            end
        end
    end
endmodule

```

3.12. Khối mux3

Khối mux3 thực hiện lựa chọn giá trị đưa vào thanh ghi có địa chỉ là address D là giá trị từ ngõ ra của khối DMEM, ALU, PC+4 dựa vào giá trị điều khiển của khối control.

Code:

```

module mux3(out,in0,in1,in2,sel);
parameter WIDTH_DATA_LENGTH = 32;
output reg [WIDTH_DATA_LENGTH-1:0] out;
input [WIDTH_DATA_LENGTH-1:0] in0,in1,in2;
input [1:0] sel;

```

```

//reg [31:0] temp;
//assign out=temp;
always @ (in0 or in1 or in2 or sel)
case(sel)
    2'b00: out = in0;
    2'b01: out = in1;
    2'b10: out = in2;
endcase
endmodule

```

IV. Thực hiện mô phỏng

4.1. Test bench

```

module tb_top();
reg clk,rst, clk_1;
riscv_top tb_riscv (.clk_1(clk_1),
                    .clk(clk),
                    .rst(rst)
                    );

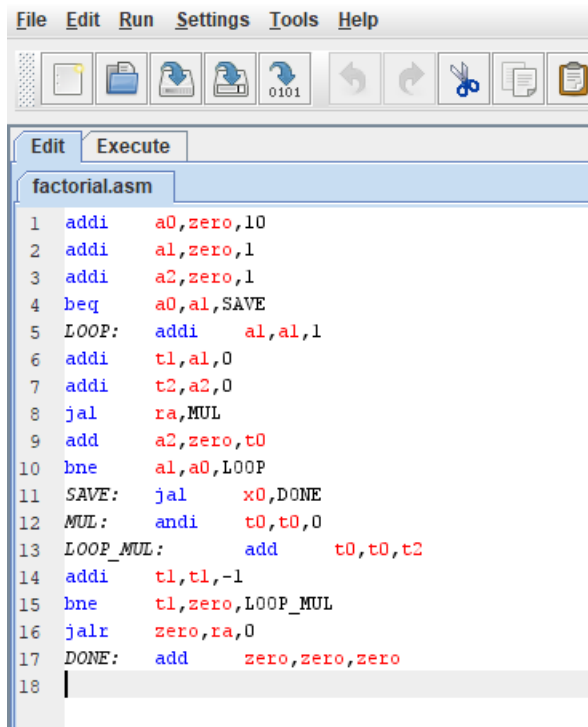
initial
begin
    clk = 0;
    rst = 0;
    clk_1 = 0;
    #6
    rst = 1;
    #24
    rst = 0;
end
always
#6 clk = !clk;
always
#1 clk_1 = !clk_1;

initial
#10000 $finish;
endmodule

```


4.2. Thực hiện mô phỏng

Code Assembly:

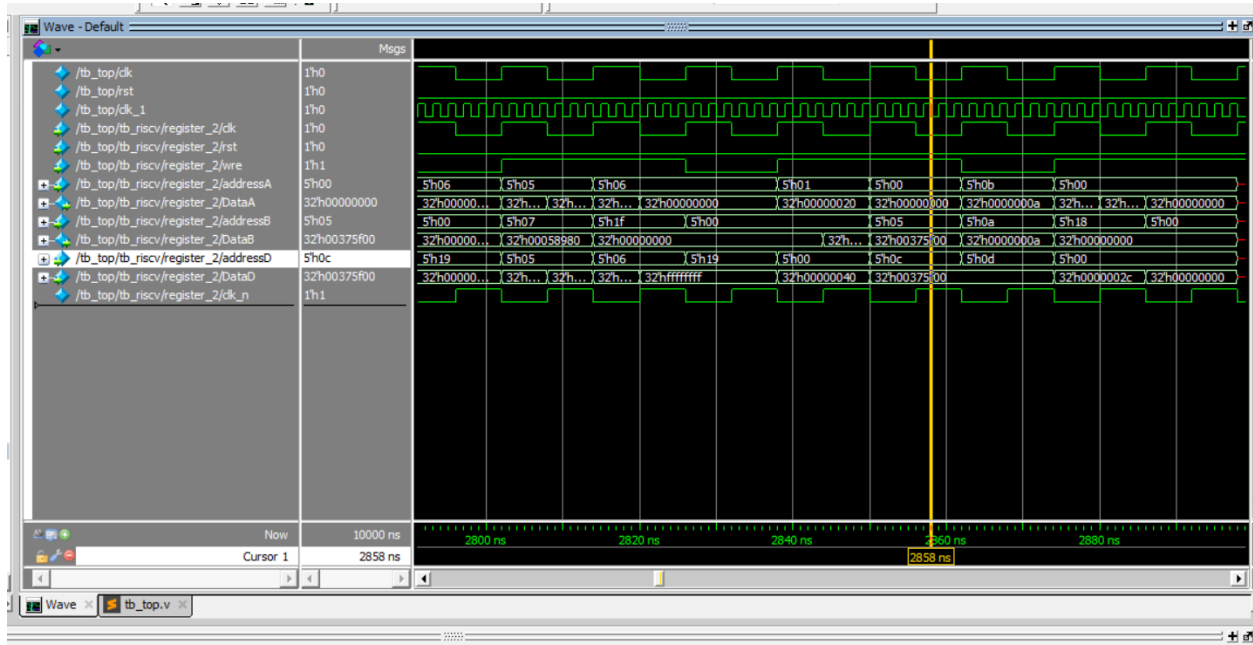


```
1  addi    a0,zero,10
2  addi    a1,zero,1
3  addi    a2,zero,1
4  beq     a0,a1,SAVE
5  LOOP:   addi    a1,a1,1
6  addi    t1,a1,0
7  addi    t2,a2,0
8  jal     ra,MUL
9  add     a2,zero,t0
10 bne     a1,a0,LOOP
11 SAVE:   jal     x0,DONE
12 MUL:    andi    t0,t0,0
13 LOOP_MUL: add    t0,t0,t2
14 addi    t1,t1,-1
15 bne     t1,zero,LOOP_MUL
16 jalr    zero,ra,0
17 DONE:   add     zero,zero,zero
18
```

Kết quả tính toán theo lý thuyết:

Registers	Floating Point	Control and Status	
Name	Number	Value	
zero	0	0x00000000	
ra	1	0x00400020	
sp	2	0x7ffffeffc	
gp	3	0x10008000	
tp	4	0x00000000	
t0	5	0x00375f00	
t1	6	0x00000000	
t2	7	0x00058980	
s0	8	0x00000000	
s1	9	0x00000000	
a0	10	0x0000000a	
a1	11	0x0000000a	
a2	12	0x00375f00	
a3	13	0x00000000	
a4	14	0x00000000	
a5	15	0x00000000	
a6	16	0x00000000	
a7	17	0x00000000	
s2	18	0x00000000	
s3	19	0x00000000	
s4	20	0x00000000	
s5	21	0x00000000	
s6	22	0x00000000	
s7	23	0x00000000	
s8	24	0x00000000	
s9	25	0x00000000	
s10	26	0x00000000	
s11	27	0x00000000	
t3	28	0x00000000	
t4	29	0x00000000	
t5	30	0x00000000	
t6	31	0x00000000	
pc	32	0x00400048	

Kết quả mô phỏng:



4.3. Nhận xét

Kết quả mô phỏng giống với kết quả lý thuyết.