TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

TRƯỜNG ĐIỆN- ĐIỆN TỬ

----o0o----

BÁO CÁO BÀI TẬP MÔN

KIẾN TRÚC MÁY TÍNH

Đề tài: Mô phỏng kiến trúc RISC-V bằng Verilog

Giảng viên hướng dẫn: TS. Tạ Thị Kim Huệ

Sinh viên thực hiện:

Họ và tên	MSSV
Nguyễn Đức Bình	20182380

Hà Nội 2023

MỤC LỤC

PHẦN 1
Tìm hiểu về kiến trúc bộ xử lý RISC-V

1.1 Kiến trúc RISC-V

Kiến trúc RISC-V được công bố công khai vào năm 2014, được phát triển tại Đại học California ở Berkeley bởi Yunsup Lee, Krste Asanović, David A. Patterson và Andrew Waterman. Nỗ lực này tiếp nối bốn dự án thiết kế kiến trúc RISC lớn trước đây tại UC Berkeley, dẫn đến cái tên RISC-V, trong đó *V* đại diện cho số năm La Mã.

RISC-V bắt đầu với các mục tiêu chính sau:

Thiết kế kiến trúc tập lệnh RISC (ISA) phù hợp để sử dụng trong nhiều ứng dụng, trải rộng từ các thiết bị nhúng công suất vi mô đến các bộ đa xử lý máy chủ đám mây hiệu suất cao.

Cung cấp ISA mà bất kỳ ai cũng có thể sử dụng miễn phí cho bất kỳ ứng dụng nào. Điều này trái ngược với ISA của hầu hết tất cả các bộ vi xử lý có sẵn trên thị trường khác, vốn là tài sản trí tuệ được bảo vệ cẩn thận của công ty thiết kế chúng.

Kết hợp các bài học kinh nghiệm từ nhiều thập kỷ trước về thiết kế bộ xử lý, tránh những sai lầm và các tính năng không tối ưu mà các kiến trúc khác phải giữ lại trong các thế hệ mới hơn để duy trì khả năng tương thích với các thế hệ trước, đôi khi là cổ xưa.

Cung cấp ISA cơ sở nhỏ nhưng đầy đủ phù hợp để sử dụng trong các thiết bị nhúng. ISA cơ sở là tập hợp các khả năng tối thiểu mà bất kỳ bộ xử lý RISC-V nào phải thực hiện. RISC-V cơ sở là một kiến trúc bộ xử lý 32-bit với 31 thanh ghi có mục đích chung. Tất cả các hướng dẫn đều dài 32 bit. ISA cơ sở hỗ trợ phép cộng và trừ số nguyên, nhưng không bao gồm phép nhân và phép chia số nguyên. Điều này là để tránh buộc các triển khai bộ xử lý tối thiểu phải bao gồm phần cứng nhân và chia khá đắt tiền cho các ứng dụng không yêu cầu các hoạt động đó.

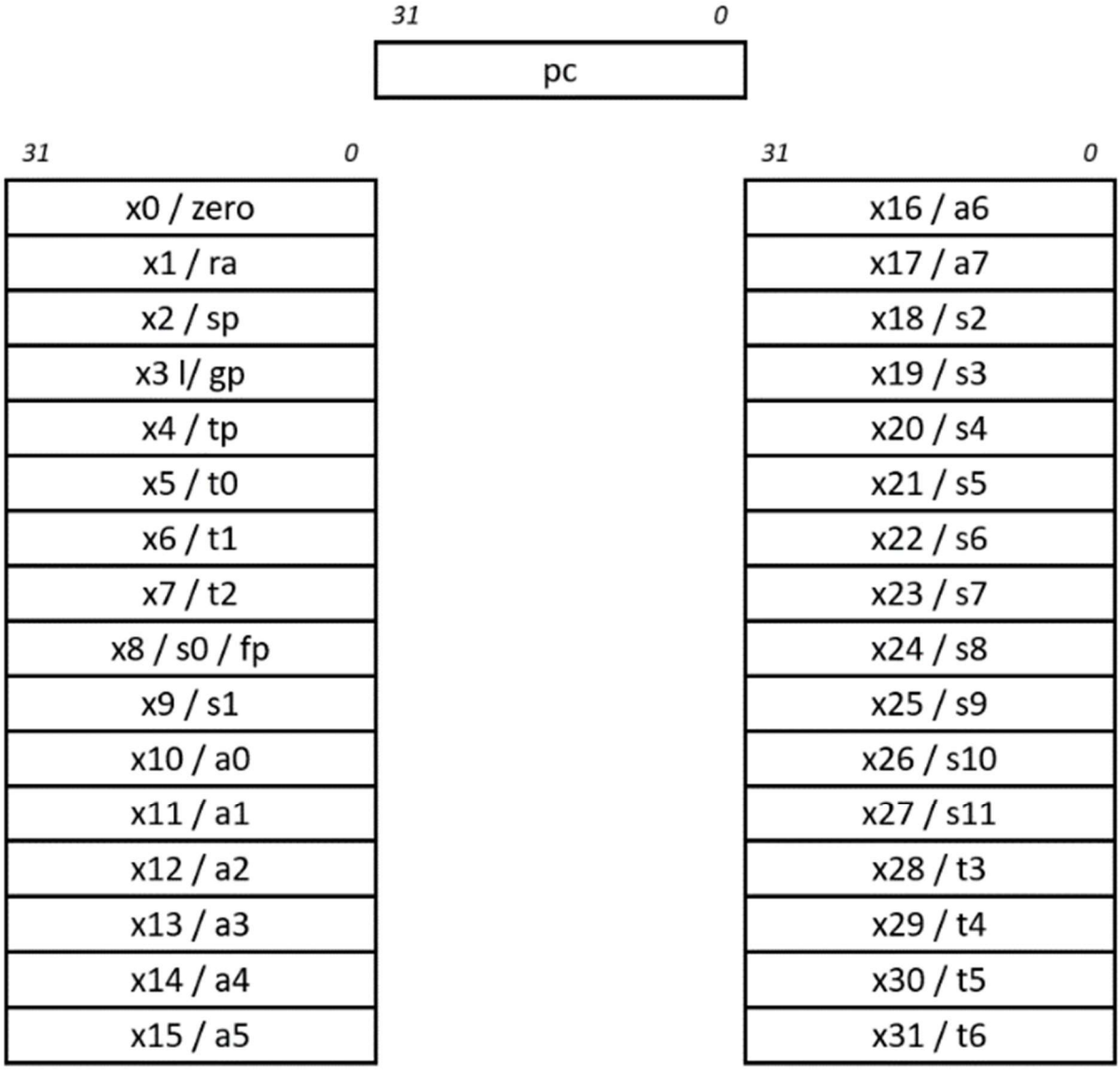
Cung cấp các phần mở rộng ISA tùy chọn để hỗ trợ toán học dấu phẩy động, các phép toán trong bộ nhớ nguyên tử cũng như phép nhân và chia.

Cung cấp các phần mở rộng ISA bổ sung để hỗ trợ các chế độ thực thi đặc quyền, tương tự như các kiến trúc đặc quyền x86, x64 và ARM.

Hỗ trợ tập lệnh nén, triển khai các phiên bản 16-bit của nhiều lệnh 32-bit. Trong các bộ xử lý triển khai phần mở rộng này, các lệnh 16 bit có thể được xen kẽ tự do với các lệnh 32 bit.

Cung cấp các phần mở rộng ISA tùy chọn để hỗ trợ kích thước từ bộ xử lý 64 bit và thậm chí 128 bit bằng cách sử dụng bộ nhớ ảo được phân trang trên bộ xử lý đơn và đa lõi cũng như trong các cấu hình đa xử lý.

Bộ thanh ghi ISA cơ sở RISC-V này được trình bày trong hình 1.1.



Hình 1. 1 Bộ thanh ghi ISA cơ sở RISC-V

1.2 Tập lệnh RISC-V

Tập lệnh RISC-V cơ bản chỉ bao gồm 47 lệnh. Tám là hướng dẫn hệ thống thực hiện các cuộc gọi hệ thống và truy cập bộ đếm hiệu suất. 39 lệnh còn

lại thuộc các loại hướng dẫn tính toán, hướng dẫn luồng điều khiển và hướng dẫn truy cập bộ nhớ. Sau đây là một số lệnh thường dùng.

1.2.1 Hướng dẫn tính toán

Tất cả các hướng dẫn tính toán ngoại trừ **`lui`** và **`auipc`** sử dụng dạng toán hạng ba. Toán hạng đầu tiên là thanh ghi đích, toán hạng thứ hai là thanh ghi nguồn, và toán hạng thứ ba là thanh ghi nguồn thứ hai hoặc một giá trị tức thì. Hướng dẫn ghi nhớ sử dụng một giá trị tức thì (ngoại trừ **`auipc`**) kết thúc bằng chữ cái `i`.

`add`, **`addi`**, **`sub`**: Thực hiện cộng và trừ. Giá trị tức thời trong `addi`lệnh là giá trị có dấu 12 bit. Lệnh `sub` trừ toán hạng nguồn thứ hai với toán hạng đầu tiên. Không có `subi`hướng dẫn vì `addi` có thể thêm giá trị âm ngay lập tức.

`sll`, **`slli`**, **`srl`**, **`srli`**, **`sra`**, **`srai`**: Thực hiện logic trái và thay đổi đúng (**`sll`** và **`srl`**), và thay đổi số học đúng (**`sra`**). Dịch chuyển logic chèn các bit 0 vào các vị trí trống. Dịch chuyển bên phải số học sao chép bit dấu hiệu vào các vị trí trống. Số vị trí bit cần dịch chuyển được lấy từ 5 bit thấp nhất của thanh ghi nguồn thứ hai hoặc từ giá trị tức thời 5 bit.

`and`, **`andi`**, **`or`**, **`ori`**, **`xor`**, **`xori`**: Thực hiện các hoạt động Bitwise lượng trên hai toán hạng nguồn. Toán hạng ngay lập tức là 12 bit.

`lui`: Nạp ngay phía trên. Lệnh này tải các bit 12-31 của thanh ghi đích với giá trị tức thời 20 bit. Đặt một thanh ghi thành giá trị tức thời 32 bit tùy ý yêu cầu hai hướng dẫn: Đầu tiên, **`lui`** đặt các bit 12-31 thành 20 bit trên của giá trị. Sau đó, **`addi`** thêm vào 12 bit thấp hơn để tạo thành kết quả 32 bit hoàn chỉnh. `lui` có hai toán hạng: thanh ghi đích và giá trị tức thời.

`auipc`: Thêm ngay phía trên vào PC. Lệnh này thêm giá trị tức thời 20 bit vào 20 bit phía trên của bộ đếm chương trình. Hướng dẫn này cho phép định địa chỉ tương đối PC trong RISC-V. Để tạo một địa chỉ tương đối PC 32-bit hoàn chỉnh, hãy **`auipc`** tạo một kết quả từng phần, sau đó một **`addi`** lệnh sẽ thêm vào 12 bit thấp hơn.

1.2.2 Hướng dẫn luồng kiểm soát

Các lệnh rẽ nhánh có điều kiện thực hiện so sánh giữa hai thanh ghi và dựa trên kết quả, có thể chuyển quyền điều khiển trong phạm vi của độ lệch địa chỉ 12 bit đã ký từ PC hiện tại. Có sẵn hai lệnh nhảy vô điều kiện, một trong số đó (`jalr`) cung cấp quyền truy cập vào toàn bộ dải địa chỉ 32 bit.

`beq`, **`bne`**, **`blt`**, **`bltu`**, **`bge`**, **`bgeu`**: Chi nhánh nếu bằng nhau (**`beq`**), không bằng (**`bne`**), ít hơn (**`blt`**), ít hơn unsigned (**`bltu`**), lớn hơn hoặc bằng (**`bge`**), hoặc cao

hơn hoặc tương đương, unsigned (**bgeu**). Các lệnh này thực hiện phép so sánh được chỉ định giữa hai thanh ghi và nếu điều kiện được thỏa mãn, chuyển quyền điều khiển đến bù địa chỉ được cung cấp trong giá trị tức thời có dấu 12 bit.

jal: Nhảy và liên kết. Chuyển quyền điều khiển đến địa chỉ tương đối của PC được cung cấp ở giá trị tức thời có chữ ký 20-bit và lưu trữ địa chỉ của lệnh tiếp theo (địa chỉ trả về) trong thanh ghi đích.

jalr: Nhảy và liên kết, đăng ký. Tính địa chỉ đích là tổng của thanh ghi nguồn và giá trị tức thời 12 bit có dấu, sau đó nhảy đến địa chỉ đó và lưu trữ địa chỉ của lệnh tiếp theo trong thanh ghi đích. Khi được đặt trước `auipc`lệnh, `jalr`lệnh có thể thực hiện một bước nhảy tương đối PC ở bất kỳ đâu trong không gian địa chỉ 32 bit.

1.2.3 Hướng dẫn truy cập bộ nhớ

Các hướng dẫn truy cập bộ nhớ truyền dữ liệu giữa một thanh ghi và một vị trí bộ nhớ. Toán hạng đầu tiên là thanh ghi được tải hoặc lưu trữ. Thứ hai là một thanh ghi chứa địa chỉ bộ nhớ. Giá trị tức thời 12 bit đã ký được thêm vào địa chỉ trong thanh ghi để tạo ra địa chỉ cuối cùng cho tải hoặc lưu trữ.

Hướng dẫn tải thực hiện phần mở rộng dấu cho các giá trị có dấu hoặc phần mở rộng bằng 0 cho các giá trị chưa được ký. Phép toán mở rộng dấu hoặc số không điền vào tất cả 32 bit trong thanh ghi đích khi một giá trị dữ liệu nhỏ hơn (một byte hoặc nửa từ) được tải. Tải không có dấu được chỉ định bằng một *chữ u* ở cuối trong ghi nhớ.

lb, **lbu**, **lh**, **lhu**, **lw**: Load một byte 8-bit (**lb**), một nữa vòng trái đất 16-bit (**lh**) hoặc 32-bit word (**lw**) vào thanh ghi đích. Đối với tải byte và nửa từ khóa, lệnh sẽ ký mở rộng (**lb** và **lh**) hoặc mở rộng bằng không (**lbu** và **lhu**) để điền vào thanh ghi đích 32 bit. Ví dụ, lệnh **lw x1, 16(x2)** tải từ tại địa chỉ (**x2**+ **16**) vào thanh ghi **x1**.

sb, **sh**, **sw**: Lưu trữ một byte (**sb**), **half-word** (**sh**) hoặc **word** (**sw**) đến một vị trí bộ nhớ phù hợp với kích thước của giá trị dữ liệu.

1.2.4 Hướng dẫn giả

Kiến trúc RISC-V có một tập lệnh thực sự bị cắt giảm, thiếu một số loại lệnh có trong các tập lệnh của các kiến trúc bộ xử lý khác. Các chức năng của nhiều hướng dẫn quen thuộc hơn đó có thể được thực hiện với các lệnh RISC-V, mặc dù có lẽ không phải theo cách trực quan ngay lập tức.

Trình hợp dịch RISC-V hỗ trợ một số lệnh giả, mỗi lệnh này chuyển thành một hoặc nhiều lệnh RISC-V cung cấp một loại chức năng mà người ta có thể

mong đợi trong tập lệnh bộ xử lý có mục đích chung. Bảng sau đây trình bày một số hướng dẫn giả RISC-V hữu ích nhất:

Bảng 1. 1 Một số hướng dẫn giả RISC-V

Chỉ dẫn giả	(Các) lệnh RISC-V	Chức năng
`nop`	`addi x0, x0, 0`	Không hoạt động
`mv rd,rs`	`addi rd, rs, 0`	Sao chép `rs` vào`rd`
`not rd, rs`	`ori rd, rs, -1`	`rd = NOT rs`
`neg rd, rs`	`sub rd, x0, rs`	`rd = -rs`
`j offset`	`jal x0, offset`	Nhảy vô điều kiện
`jal offset`	`jal x1, offset`	Lệnh gọi hàm gần (độ lệch 20 bit)
`call offset`	`auipc x1, offset[31:12] + offset[11]` `jalr x1, offset[11:0](x1)`	Lệnh gọi hàm xa (độ lệch 32 bit)
`ret`	`jalr x0, 0(x1)`	Trở lại từ chức năng
`beqz rs, offset`	`beq rs, x0, offset`	Nhánh nếu bằng 0
`bgez rs, offset`	`bge rx, x0, offset`	Nhánh nếu lớn hơn hoặc bằng 0
`bltz rs, offset`	`blt rs, x0, offset`	Nhánh nếu nhỏ hơn 0
`bgt rs, rt, offset`	`blt rt, rs, offset`	Nhánh nếu lớn hơn

Chỉ dẫn giả	(Các) lệnh RISC-V	Chức năng
`ble rs, rt, offset`	`bge rt, rs, offset`	Nhánh nếu nhỏ hơn hoặc bằng

Trong các danh sách lệnh này, `rd` là thanh ghi đích, `rs` là thanh ghi nguồn, `csr` là thanh ghi điều khiển và trạng thái, `symbol` là địa chỉ dữ liệu tuyệt đối, và `offset` là địa chỉ lệnh tương đối với PC.

PHẦN 2:

Thiết kế và tổng hợp các khối xử lý trong RISC V bằng ngôn ngữ Verilog

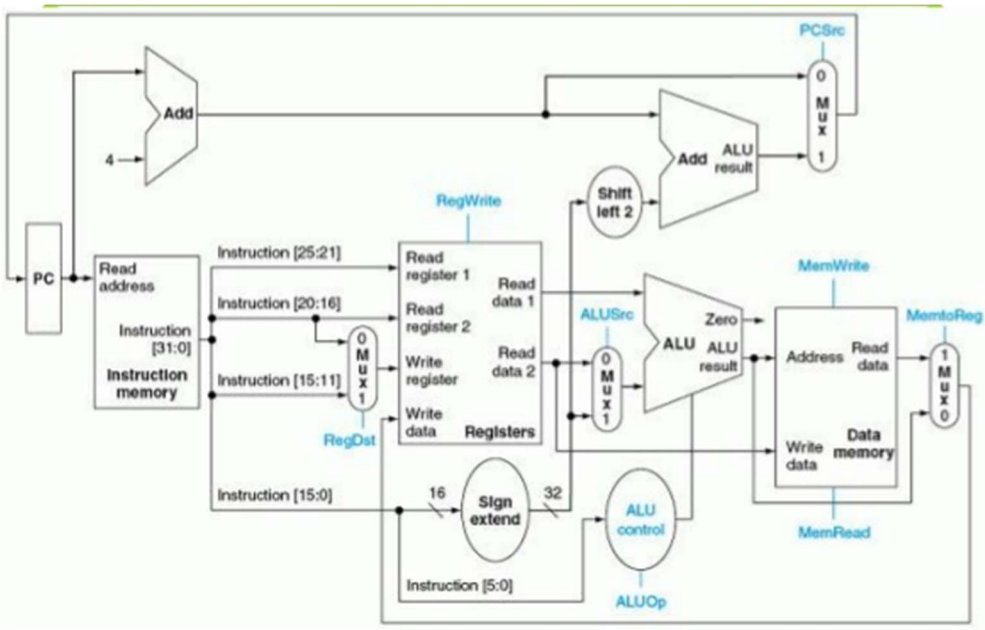
2.1 Datapath và Control logic

Datapath (Đường dẫn dữ liệu) là một tập hợp các khối lệnh như ALU, multipliers, … để xử lý dữ liệu trên thanh ghi hoặc buses.

Control logic là một phần quan trọng của chương trình dùng để điều khiển các lệnh phù hợp với phản hồi của người dùng và cũng như tự điều khiển hoạt động đã được cấu trúc sẵn trong chương trình.

Cấu trúc của Datapath và Control logic

Mô tả cấu trúc của Datapath và Control logic được dưới hình 2.1.



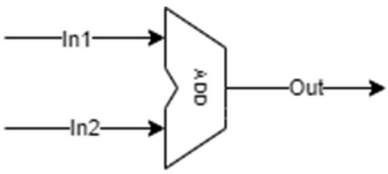
Hình 2. 1 Cấu trúc của Datapath và Control logic

Cấu trúc Datapath và Control logic gồm có các khối: Add, PC, IMEM, MUX, REG, ALU, DMEM, … Mỗi khối thực hiện một chức năng riêng.

2.2 Cấu trúc từng khối và thực hiện bằng Verilog

2.2.1 Khối Add

Hình 2.2.1 mô tả cấu trúc của khối Add.



Hình 2.2.1 Cấu trúc của khối Add

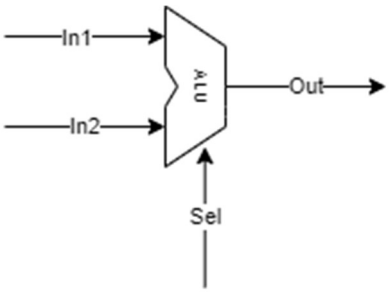
Cấu trúc: gồm có 2 đầu vào và 1 đầu ra.

Chức năng: Thực hiện phép cộng.

```
module add4 (
    add4i, add4o
);
input [31:0] add4i;
output [31:0] add4o;

assign add4o = add4i +4;

endmodule
```

2.2.2 Khối ALU

Hình 2.2.3 mô tả cấu trúc khối ALU.



Hình 2.2.3 Cấu trúc khối ALU

Cấu trúc: gồm có 2 đầu vào, 1 đầu ra và 1 chân select.

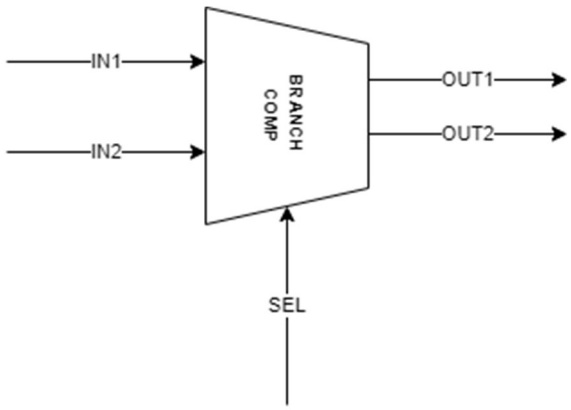
Chức năng: Khối ALU thực hiên các phép toán học logic như cộng, trừ, and, or

```
module alu (
    aluSel, aluin1, aluin2, aluout
);

input [2:0] aluSel;
input [31:0] aluin1, aluin2;
output [31:0] aluout;
reg [31:0] r;

always @(*) begin
    case (aluSel)
        1: r = aluin1 - aluin2; //sub
```

```
        4: r = aluin2; // for lui
        default: r = aluin1 + aluin2;//add
    endcase
end

assign aluout = r;

endmodule
```

2.2.3 Khối Branch Compare

Hình 2.2.4 mô tả cấu trúc khối Branch Compare.



Hình 2.2.4 Cấu trúc khối Branch Compare

Cấu trúc: Gồm có 2 đầu vào, 2 đầu ra và 1 chân select.

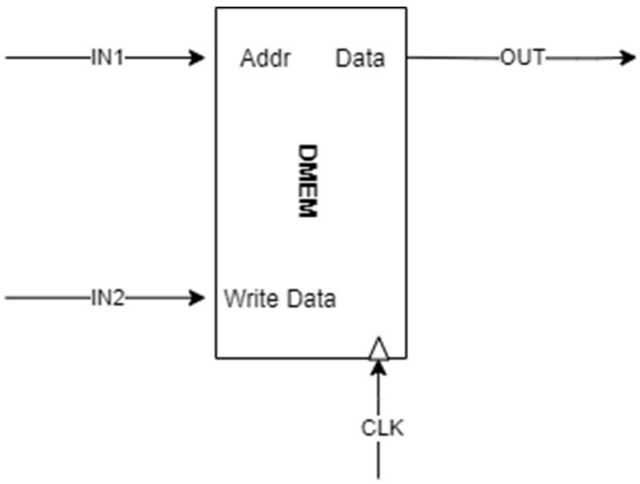
Chức năng: Thực hiện so sánh 2 dữ liệu đầu vào.

```
module branchcomp (
    input1, input2, BrUn, BrEq, BrLt
);
input [31:0] input1;
input [31:0] input2;
input BrUn;
output BrEq, BrLt;

reg BrEqR, BrLtR;
reg [31:0] r1, r2;
reg [30:0] ru1, ru2;
reg r1_31, r2_31;

always @(*) begin
    r1 = input1;
    r2 = input2;
    ru1 = input1 [30:0];
    ru2 = input2 [30:0];
```

```
    r1_31 = input1[31];
    r2_31 = input2[31];

    // trường hợp so sánh hai số không dấu
    if(BrUn) begin
        if(r1 != r2) begin
            BrEqR = 0;
            BrLtR = (r1 < r2) ? 1 : 0;
        end
        else BrEqR = 1;
    end
    else begin // trường hợp so sánh hai số có dấu, bit 31 là bit dấu
        if(r1_31 != r2_31) begin
            if(ru1 == 0 && ru2 == 0) BrEqR = 1;
            else begin
                BrEqR = 0;
                if(r1_31 == 1) BrLtR = 1;
                else  BrLtR = 0;
            end
        end
        else begin
            if(ru1 != ru2) begin
                BrEqR = 0;
                if(r1_31 == 1) BrLtR = (ru1 > ru2) ? 1 : 0;
                else           BrLtR = (ru1 < ru2) ? 1 : 0;
            end
            else BrEqR = 1;
        end
    end
end

assign BrEq = BrEqR;
assign BrLt = BrLtR;

endmodule
```

2.2.5 Khối DMEM

Hình 2.2.5 mô tả cấu trúc khối DMEM.



Hình 2.2.5 Cấu trúc khối DMEM

Cấu trúc: Gồm có 3 đầu vào, 1 đầu ra và 1 chân select.

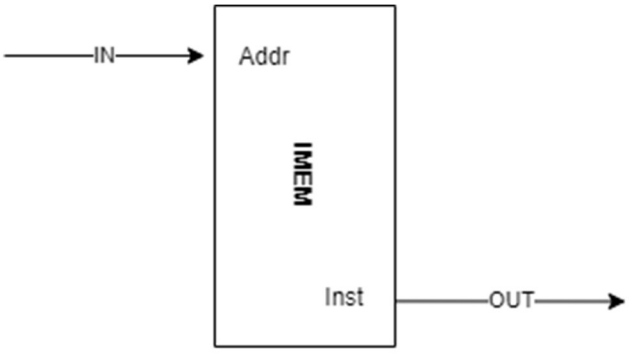
Chức năng: Khối DMEM có chức năng là nơi ghi dữ liệu.

```
module dmem (
    clk, address, WriteData, ReadData, MemRW
);

input clk, MemRW;
input [31:0] address, WriteData;
output [31:0] ReadData;
reg [31:0] r;

reg [7:0] array [31:0];

always @(posedge clk) begin
    if (MemRW) begin // chia dữ liệu 32bit thành các mảnh 8bit rồi ghi
        array[address] = WriteData[7:0];
        array[address+1] = WriteData[15:8];
        array[address+2] = WriteData[23:16];
        array[address+3] = WriteData[31:24];
    end
end

always @(*) begin // đọc dữ liệu 32bit từ địa chỉ cơ sở
    r = {array[address+3], array[address+2], array[address+1], array[address]}
;
end

assign ReadData = r;

initial begin
        array[0] = 8'd10;
        array[1] = 8'd20;
        array[2] = 8'd30;
```

```
        array[3] = 8'd40;
        array[4] = 8'd50;
        array[5] = 8'd60;
        array[6] = 8'd70;
        array[7] = 8'd80;
        array[8] = 8'd90;
        array[9] = 8'd100;
        array[10] = 8'd101;
        array[11] = 8'd102;
        array[12] = 8'd103;
        array[13] = 8'd104;
        array[14] = 8'd105;
        array[15] = 8'd106;
        array[16] = 8'd107;
        array[17] = 8'd108;
        array[18] = 8'd109;
        array[19] = 8'd200;
        array[20] = 8'd201;
        array[21] = 8'd202;
        array[22] = 8'd203;
        array[23] = 8'd204;
        array[24] = 8'd205;
        array[25] = 8'd206;
        array[26] = 8'd207;
        array[27] = 8'd208;
        array[28] = 8'd209;
        array[29] = 8'd300;
        array[30] = 8'd301;
        array[31] = 8'd302;
end

endmodule
```

2.2.6 Khối IMEM

Hình 2.2.6 mô tả cấu trúc khối IMEM.



Hình 2.2.6 Cấu trúc khối IMEM

Cấu trúc: Gồm có 1 đầu vào và 1 đầu ra.

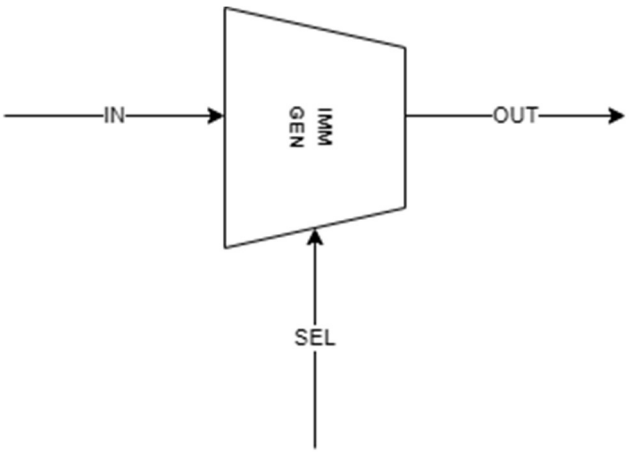
Chức năng: Khối IMEM là nơi lưu trữ các Instruction cho hệ thống.

```
module IMEM (
    address, inst
);

input [31:0] address;
output [31:0] inst;

reg [7:0] array[1024:0];

//hướng dẫn 32bit được tổng hợp từ các giá trị trong 4 địa chỉ liên tiếp
assign inst = {array[address+3], array[address+2], array[address+1], array[add
ress]};

initial begin
// add x18,x19,x10
array[0] = 8'b00110011;
array[1] = 8'b10001001;
array[2] = 8'b10101001;
array[3] = 8'b00000000;
// addi x15,x1,-50
array[4] = 8'b10010011;
array[5] = 8'b10000111;
array[6] = 8'b11100000;
array[7] = 8'b11111100;
// sw x14, 8(x2)
array[8] = 8'b00100011;
array[9] = 8'b00100100;
array[10] = 8'b11100001;
array[11] = 8'b00000000;
// lw x14, 8(x2)
array[12]  = 8'b00000011;
array[13]  = 8'b00100111;
array[14] = 8'b10000001;
array[15] = 8'b00000000;

// // bne x19,x10,16
// array[16] = 8'b01100011;
// array[17] = 8'b10011000;
// array[18] = 8'b10101001;
// array[19] = 8'b00000000;

//jalr x15,x1,-16
// array[16] = 8'b01100111;
// array[17] = 8'b11000000;
// array[18] = 8'b00001011;
// array[19] = 8'b00000001;
```

```
// //jal x5, 16
// array[16] = 8'b11101111;
// array[17] = 8'b00000010;
// array[18] = 8'b00000000;
// array[19] = 8'b00000001;

// lui x10, 0xEDAB3
array[16] = 8'b00110111;
array[17] = 8'b00110101;
array[18] = 8'b10101011;
array[19] = 8'b11101101;

end

endmodule
```

2.2.7 Khối IMM GEN

Hình 2.2.7 mô tả cấu trúc của khối IMM GEN.



Hình 2.2.7 Cấu trúc của khối IMM GEN

Cấu trúc: gồm có 1 đầu vào, 1 đầu ra và 1 chân select.

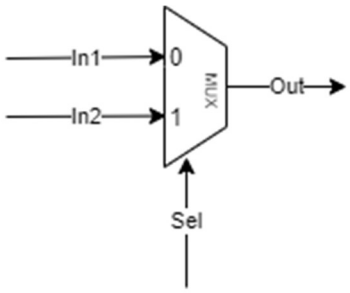
Chức năng: Tạo ra các Immediate từ Instruction theo Format đã quy định sẵn.

```
module immgen (
    inst, immSel, immgen
);

input [31:0] inst;
input [2:0] immSel;
output [31:0] immgen;

reg [31:0] immgenR;

always @(*) begin
```

```
   case (immSel)
    //I-type
    0:  immgenR = {{20{inst[31]}}, inst[31:20]};
    //S-type
    1:  immgenR = {{20{inst[31]}}, inst[31:25], inst[11:7]};
    //B-type
    2:  immgenR = {{20{inst[31]}}, inst[7], inst[30:25], inst[11:8], 1'b0} ;
    //J-type
    3:  immgenR = {{12{inst[31]}}, inst[19:12], inst[20], inst[30:25], inst[24
:21], 1'b0};
    //U-type
    4:  immgenR = {inst[31:12], {12{1'b0}}};
    default: immgenR = 0;
    endcase
end

assign immgen = immgenR;

endmodule
```

2.2.8 Khối MUX

Hình 2.2.8 mô tả cấu trúc khối MUX.



Hình 2.2.8 Cấu trúc khối MUX

Cấu trúc: Gồm 2 (MUX-2) or 4 (MUX-4) đầu vào, 1 đầu ra và 1 chân select.

Chức năng: Đầu ra là một trong số các đầu vào phụ thuộc vào giá trị của chân select.

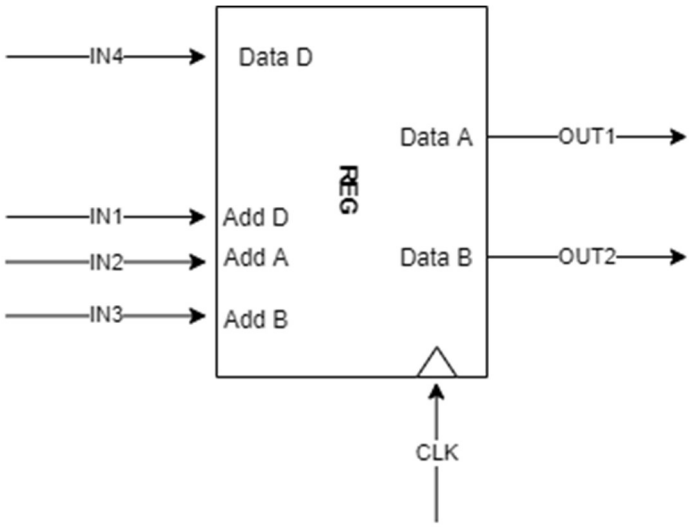
```
module mux2 (
    muxi0, muxi1, muxo, muxsel
);

input [31:0] muxi0, muxi1;
output [31:0] muxo;
input muxsel;

assign muxo = muxsel ? muxi1 : muxi0;
```

```
endmodule
```

```
module mux4 (
    muxi0, muxi1, muxi2, muxi3, muxo, muxsel
);

input [31:0] muxi0, muxi1, muxi2, muxi3;
output [31:0] muxo;
input [1:0] muxsel;

assign muxo = muxsel[1] ? (muxsel[0] ? muxi3 : muxi2) : (muxsel[0] ? muxi1 : m
uxi0);

endmodule
```

2.2.9 Khối REG

Hình 2.2.9 mô tả cấu trúc khối REG.



Hình 2.2.9 Cấu trúc khối REG

Cấu trúc: gồm có 5 đầu vào, 2 đầu ra và 1 chân select.

Chức năng: Thực hiện đọc or ghi lệnh.

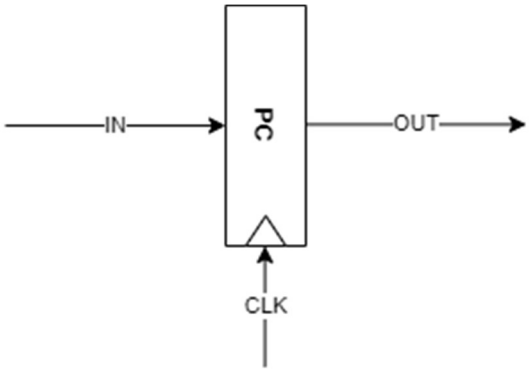
```
module registor (
    clk, rst, RegWEn, Writedata, rs1, rs2, rd, ReadData1, ReadData2
);

input clk, rst, RegWEn;
input [31:0] Writedata;
input [4:0] rs1, rs2, rd;
output [31:0] ReadData1, ReadData2;

reg [31:0] array [31:0];
```

```
// thanh ghi được làm trống sau mỗi lần reset hệ thống

always @(rs1 or rs2 or rst) begin
    if(rst) begin
        array[0] = 32'd0;
        array[1] = 32'd0;
        array[2] = 32'd0;
        array[3] = 32'd0;
        array[4] = 32'd0;
        array[5] = 32'd0;
        array[6] = 32'd0;
        array[7] = 32'd0;
        array[8] = 32'd0;
        array[9] = 32'd0;
        array[10] = 32'd0;
        array[11] = 32'd0;
        array[12] = 32'd0;
        array[13] = 32'd0;
        array[14] = 32'd0;
        array[15] = 32'd0;
        array[16] = 32'd0;
        array[17] = 32'd0;
        array[18] = 32'd0;
        array[19] = 32'd0;
        array[20] = 32'd0;
        array[21] = 32'd0;
        array[22] = 32'd0;
        array[23] = 32'd0;
        array[24] = 32'd0;
        array[25] = 32'd0;
        array[26] = 32'd0;
        array[27] = 32'd0;
        array[28] = 32'd0;
        array[29] = 32'd0;
        array[30] = 32'd0;
        array[31] = 32'd0;
    end
end

assign ReadData1 = array[rs1];
assign ReadData2 = array[rs2];

always @(posedge clk) begin
    if(RegWEn)
        array[rd] = Writedata;
end

initial begin
        array[0] = 32'd1;
```

```
        array[1] = 32'd52;
        array[2] = 32'd3;
        array[3] = 32'd4;
        array[4] = 32'd5;
        array[5] = 32'd6;
        array[6] = 32'd7;
        array[7] = 32'd8;
        array[8] = 32'd9;
        array[9] = 32'd10;
        array[10] = 32'd11;
        array[11] = 32'd12;
        array[12] = 32'd13;
        array[13] = 32'd14;
        array[14] = 32'd15;
        array[15] = 32'd16;
        array[16] = 32'd17;
        array[17] = 32'd18;
        array[18] = 32'd9;
        array[19] = 32'd20;
        array[20] = 32'd21;
        array[21] = 32'd22;
        array[22] = 32'd23;
        array[23] = 32'd24;
        array[24] = 32'd25;
        array[25] = 32'd26;
        array[26] = 32'd27;
        array[27] = 32'd28;
        array[28] = 32'd29;
        array[29] = 32'd30;
        array[30] = 32'd31;
        array[31] = 32'd32;
end

endmodule
```

2.2.10 Khối PC

Hình 2.2.10 mô tả cấu trúc khối PC.



Hình 2.2.10 Cấu trúc khối PC

Cấu trúc: Gồm 2 đầu vào, 1 đầu ra.

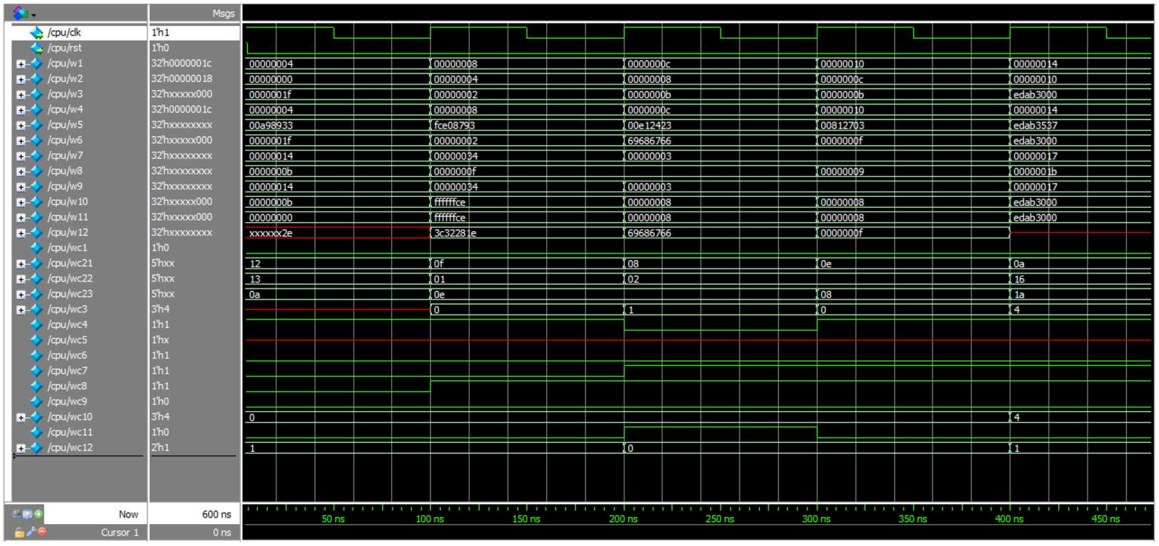
Chức năng: Đếm số tiến trình của chương trình.

```
module pc (
    clk, rst, pcin, pcout
);
input  clk;
input [31:0] pcin;
output [31:0] pcout;

input rst;
reg [31:0] r;

// đầu ra tiếp theo bằng đầu vào hiện tại khi gặp sườn lên của xung clk
always @(posedge clk) begin
    if(rst == 0)
        r <= pcin;
    else
        r <= 0;
end

assign pcout = r;

endmodule
```

PHẦN 3

Mô phỏng quá trình thực hiện lệnh ở từng khối

Kết quả mô phỏng các lệnh trong IMEM được mô tả trong hình 3.1 dưới đây.



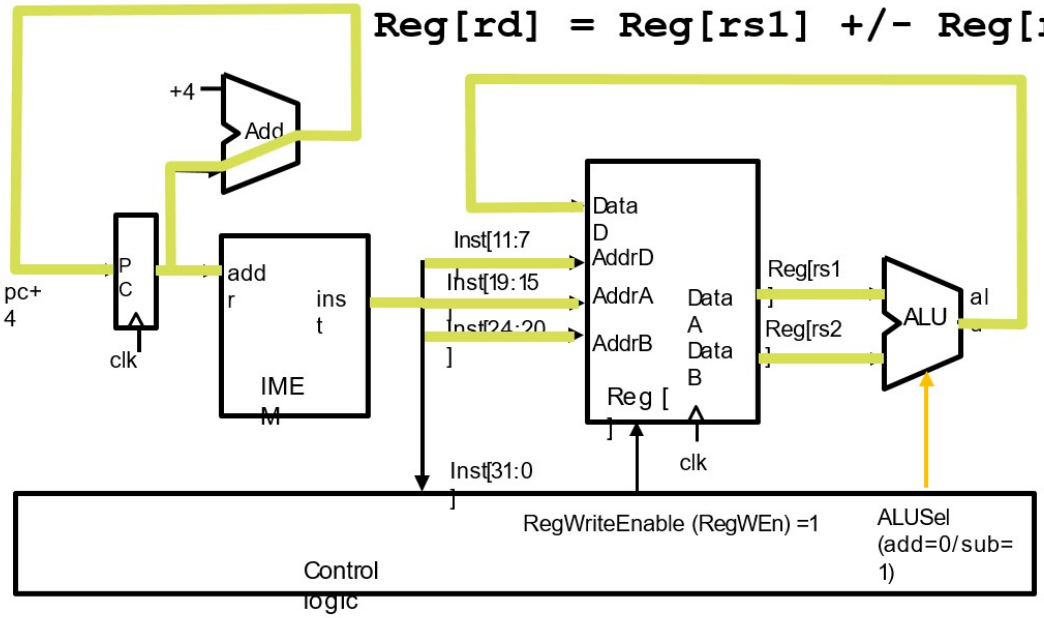
Hình 3.1 Kết quả mô phỏng các lệnh trong IMEM

Dựa theo sơ đồ khối Hình3.1 thể hiện giá trị của các dây cũng như đầu vào, đầu ra của từng khối. Với xung clock là 100ns, trong chu kỳ đầu tiên, máy tính thực hiện lệnh **`add x18,x19,x10`** với giá trị trong thanh ghi **`x19 = 20`** và **`x10 = 11`**, tổng sẽ được đưa vào thanh ghi **`x18`** qua dây **`w3`** là **`0x1f (=31)`**. Chu kỳ thứ hai, máy tính thực hiện lệnh **`addi x15,x1,-50`** tương tự. Chu kỳ tiếp theo máy tính thực hiện lệnh **`sw x14, 8(x2)`**, dữ liệu được lưu vào DMEM là **`0x0f`** được thể hiện trên dây **`w8`**. Để kiểm tra xem dữ liệu đã được lưu hay chưa, máy tính thực hiện lệnh tiếp theo **`lw x14, 8(x2),`** dữ liệu được đọc ra thể hiện trên dây **`w12`** là **`0x0f`** chính là dữ liệu vừa được lưu trong DMEM. Lệnh tiếp theo là **`lui x10, 0xEDAB3`** máy tính thực hiện nhập số **`0xEDAB3`** vào thanh ghi x10 của registor, kết quả có thể thấy ở dây **`w6`**.

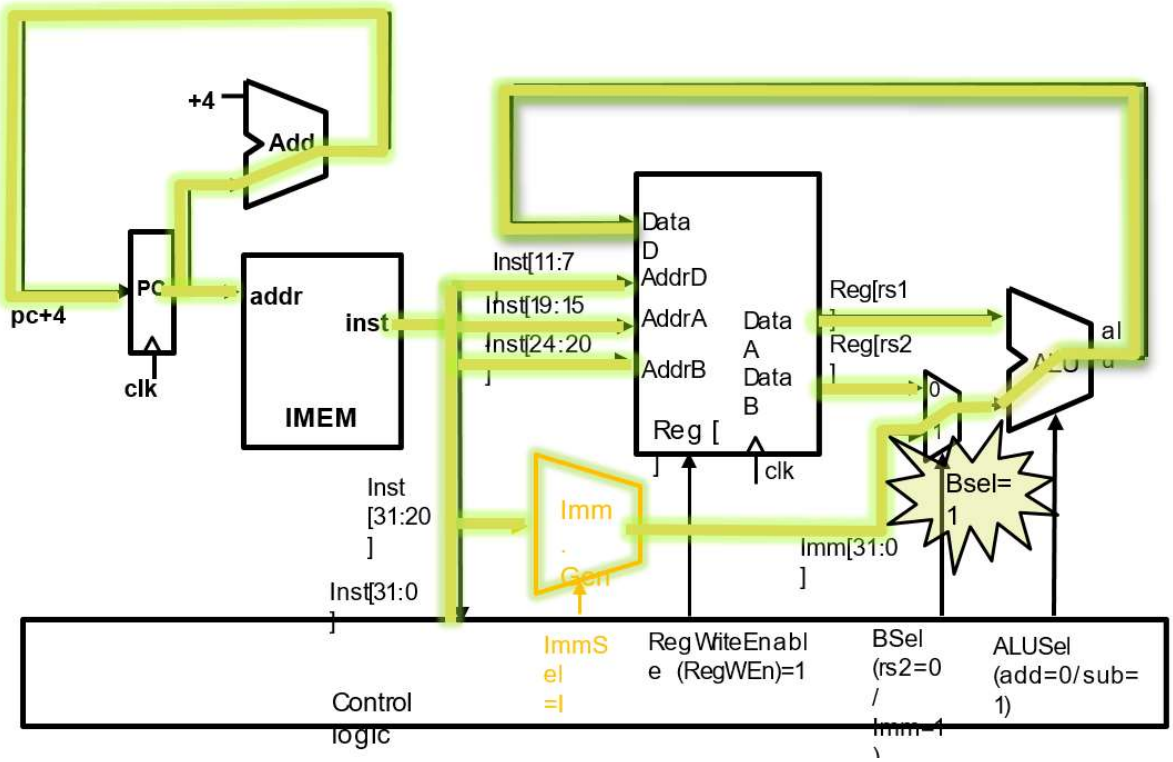
Sơ đồ đường dữ liệu các lệnh đơn khác:

Datapath for add, sub

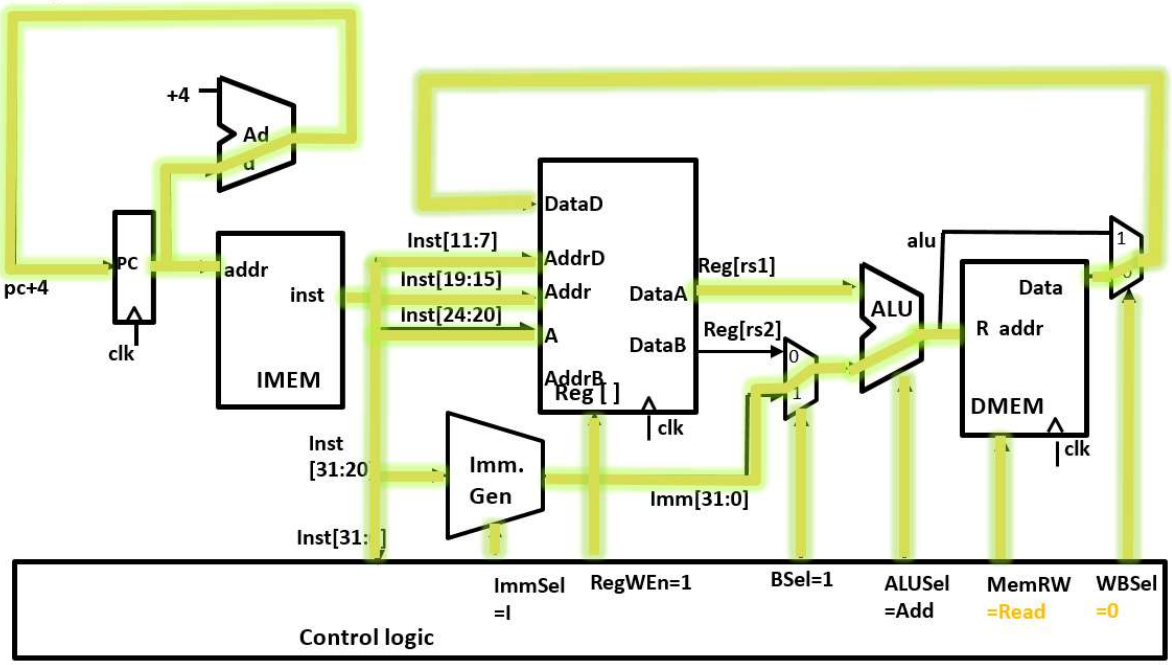
Reg[rd] = Reg[rs1] +/- Reg[rs2]



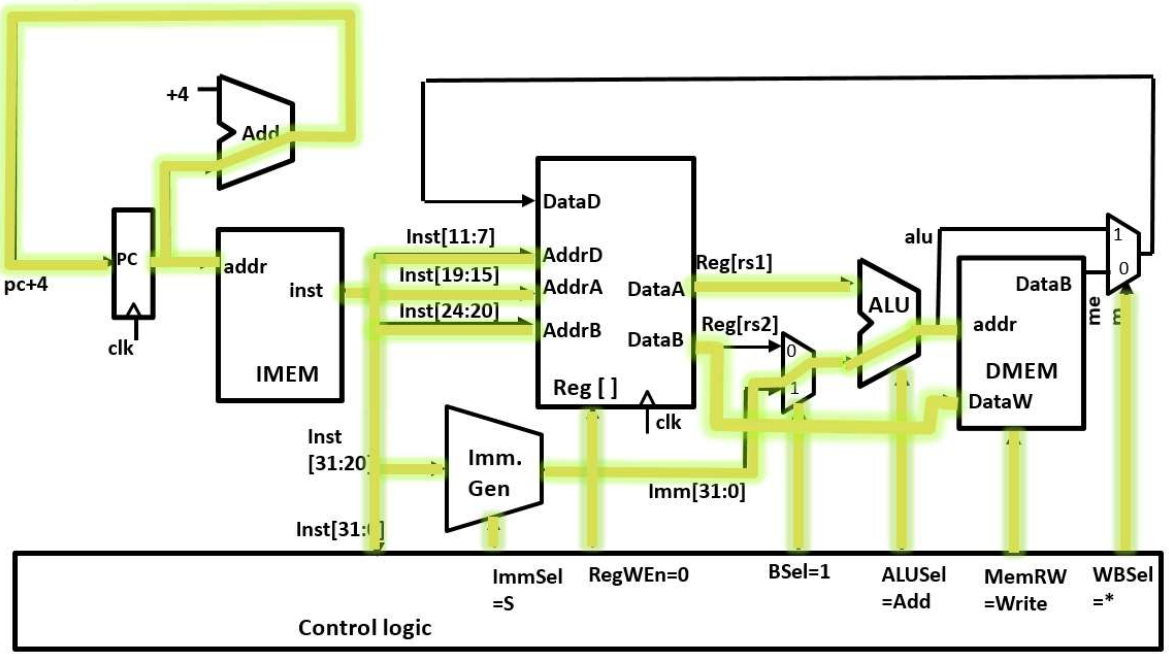
Datapath for addi



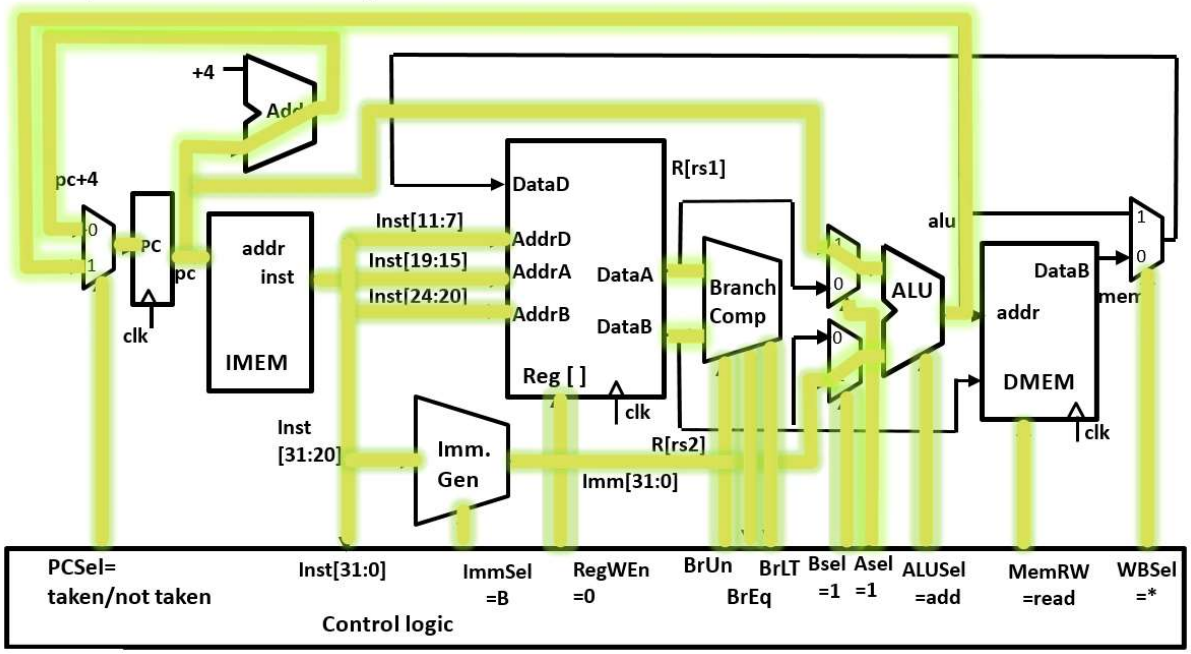
Datapath for lw



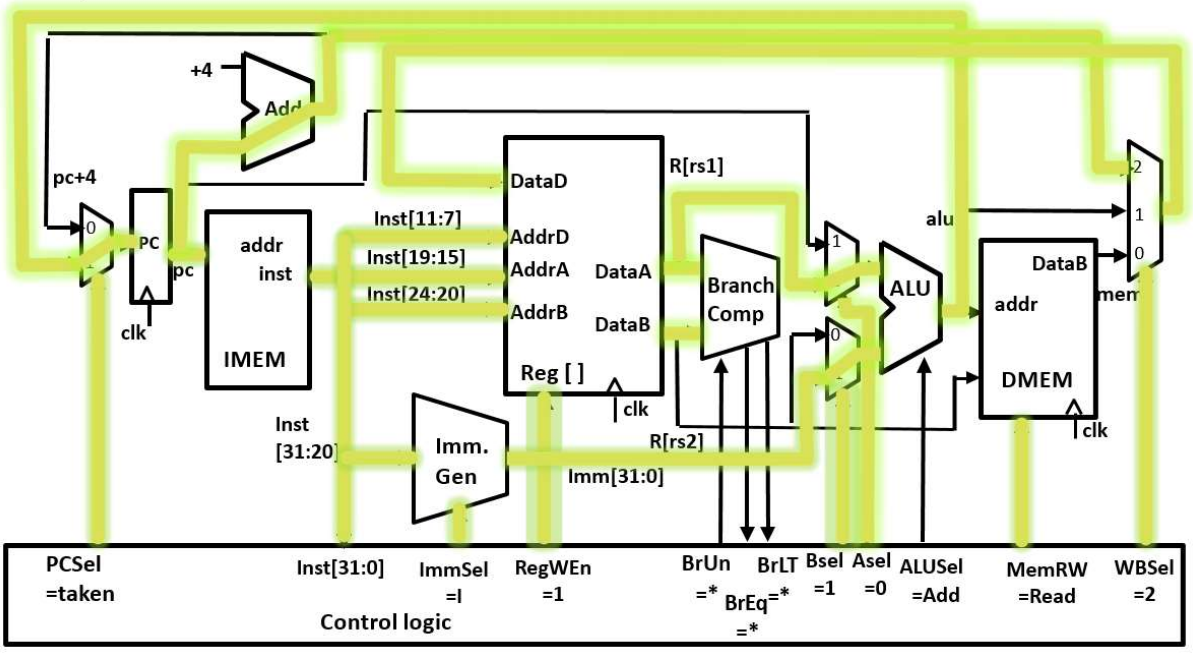
Datapath for sw



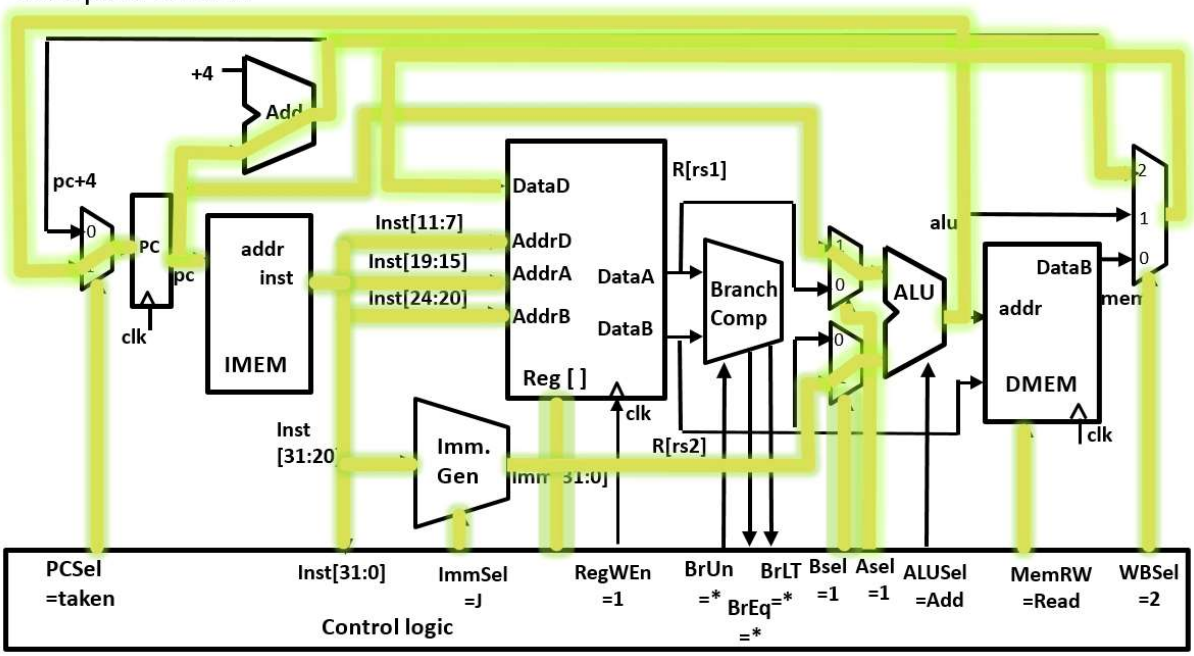
Datapath for Branch Compare



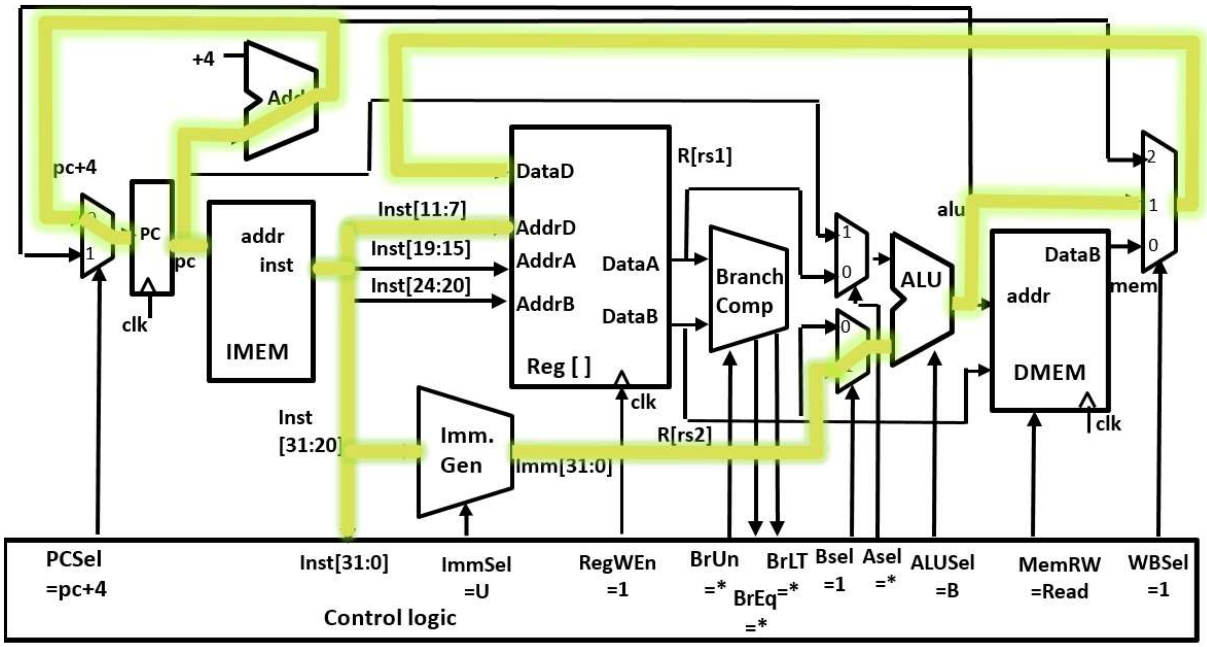
Datapath for JALR



Datapath for JAL



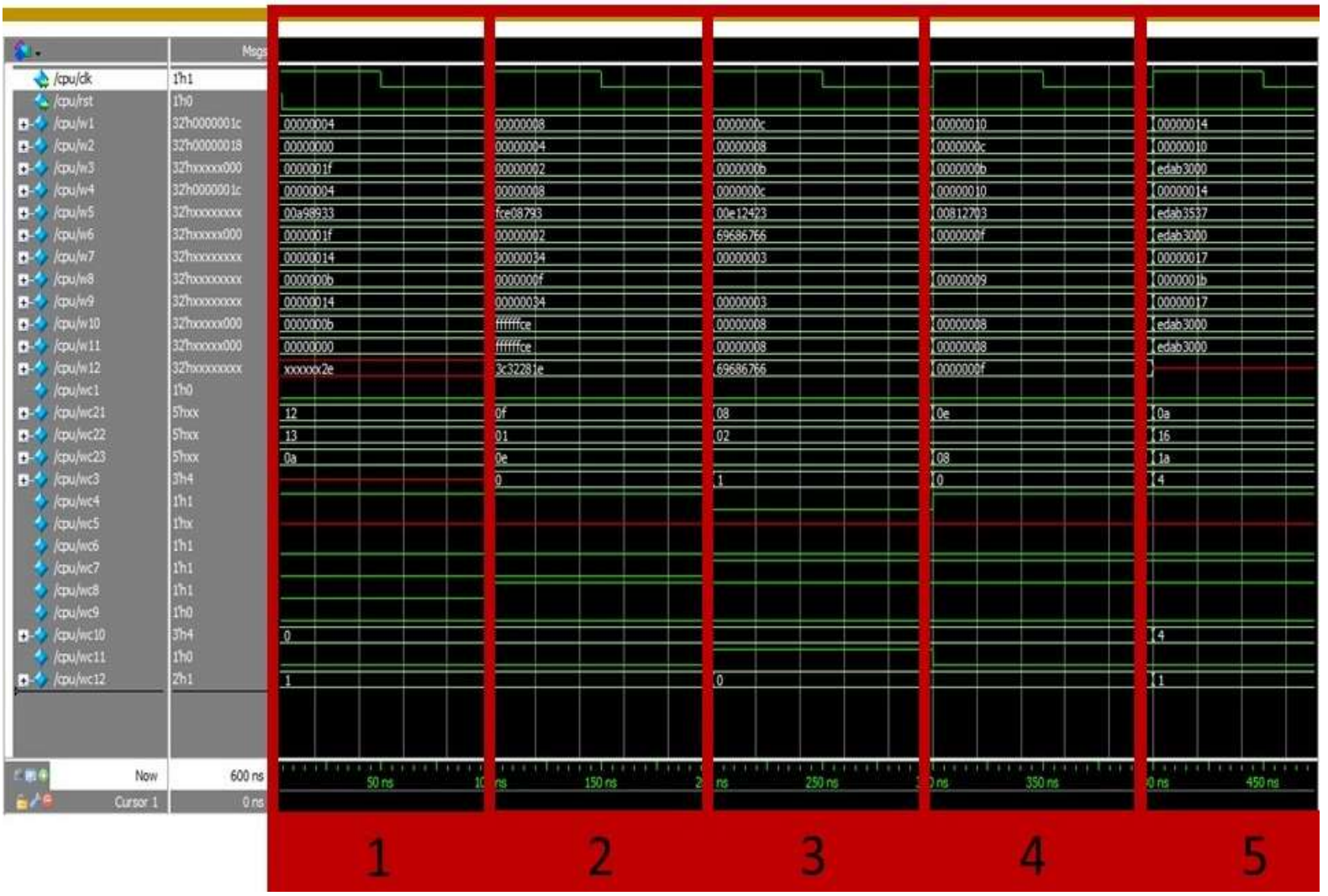
Datapath for LUI



PHẦN 4

Thực hiện mô phỏng chương trình Assembly RISC V

```
1: add x18, x19, x10                #(x19 = 20, x10 = 11)
2: addi x15, x1, -50                #(x1 = 52)
3: sw x14, 8(x2)
4: lw x14, 8(x2)
5: lui x10, 0xEDAB3
```



Tài liệu tham khảo

[1] Computer Organization and Design RISC-V Edition The Hardware Software Interface by David A. Patterson, John L. Hennessy.

[2] Great ideas in Computer Architecture (CS 61C) lecture slides, Garcia and Nikolíc, © 2020, UC Berkeley.

[3] GitHub - mircomannino/RISC-V-Simulator: A verilog simulation of the processor RISC-V

[4] Great Ideas in Computer Architecture Lecture 11 RISCV (slidetodoc.com)