ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA KĨ THUẬT MÁY TÍNH

BÀI BÁO CÁO ASSIGNMENT LAB 05 & 06
THIẾT KẾ HỆ THỐNG SỐ VỚI HDL

Sinh viên: Trương Thiên Quý
MSSV: 23521321
Lớp: CE213.P21
Giảng viên hướng dẫn: Hồ Ngọc Diễm

THIẾT KẾ BỘ PROCESSOR ĐƠN GIẢN

1. Giới thiệu đề tài

Trong lĩnh vực kiến trúc máy tính, bộ xử lý (Processor) là thành phần trung tâm đóng vai trò thực thi các tập lệnh (Instruction Set Architecture - ISA) đã được định nghĩa. Quá trình thiết kế một bộ xử lý, từ cơ bản đến phức tạp, là bước nền tảng giúp hiểu rõ hoạt động của hệ thống máy tính ở mức phần cứng.

Đề tài này tập trung vào việc xây dựng một **bộ xử lý đơn giản dạng Single-Cycle**, bao gồm các thành phần chính:

- Instruction Fetch,
- Control Unit,
- Datapath,
- PC Counter.

  Mỗi lệnh được thực thi trọn vẹn trong **một chu kỳ xung nhịp**, giúp đơn giản hóa luồng dữ liệu và luồng điều khiển.

  Hệ thống hỗ trợ các nhóm lệnh cơ bản như:
- Các phép toán số học - logic,
- Tải/ghi dữ liệu từ bộ nhớ (load/store),
- Lệnh nhảy (jump) và rẽ nhánh (branch).

Việc thiết kế và mô phỏng này giúp hiểu sâu cách vận hành nội bộ của một CPU, là bước đệm trước khi tiến tới các kiến trúc phức tạp như pipelined hoặc superscalar processors.

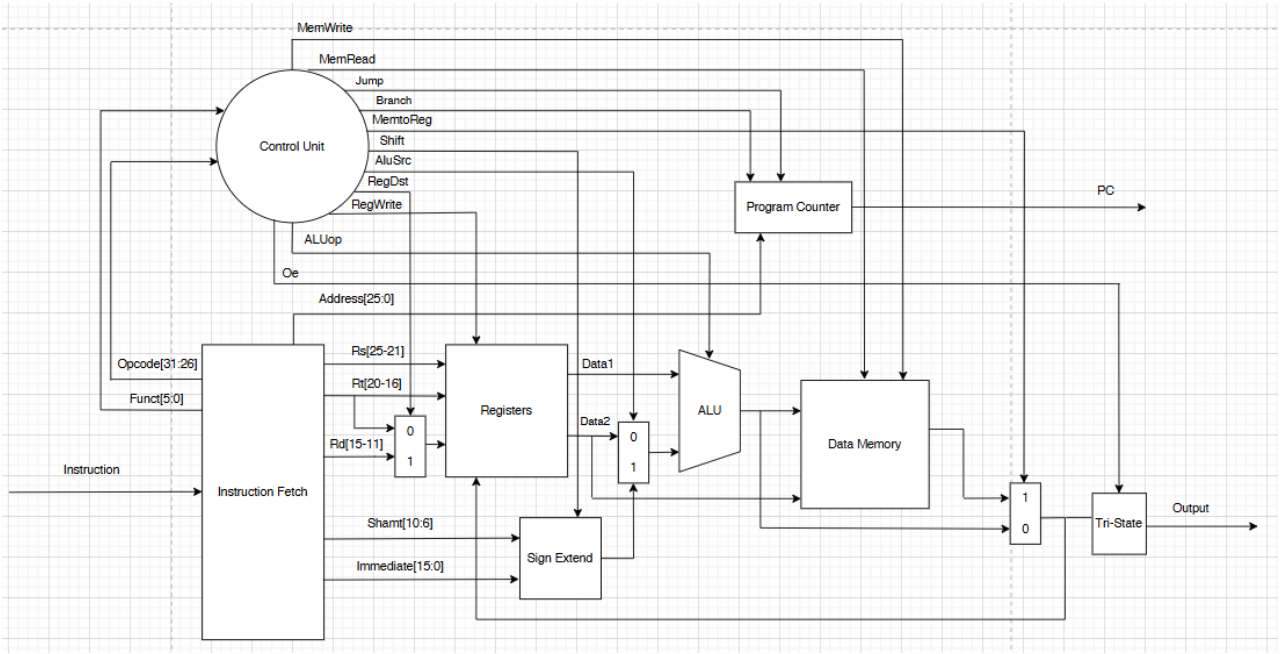
2. Kiến trúc tổng quát

Hệ thống hoạt động theo các bước tuần tự sau:

1. **Instruction Fetch**: Nạp lệnh từ bộ nhớ dựa trên địa chỉ hiện tại của PC.
2. **Instruction Decode + Control Unit**: Giải mã lệnh và sinh các tín hiệu điều khiển thích hợp.
3. **Datapath Execution**: Thực hiện các phép toán (ALU), đọc/ghi bộ nhớ dữ liệu, ghi kết quả vào thanh ghi.
4. **PC Update**: Cập nhật địa chỉ PC cho lệnh tiếp theo (PC + 4, branch hoặc jump nếu cần).

Mỗi chu kỳ xung nhịp (clk), hệ thống sẽ thực hiện đầy đủ các bước trên cho một lệnh.

3. Sơ đồ khối của hệ thống:



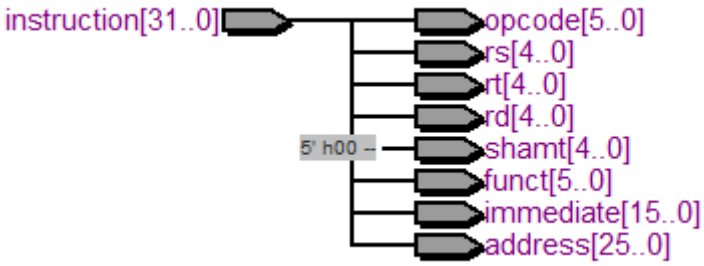
4. Mô tả chi tiết các module

4.1. Instruction Fetch

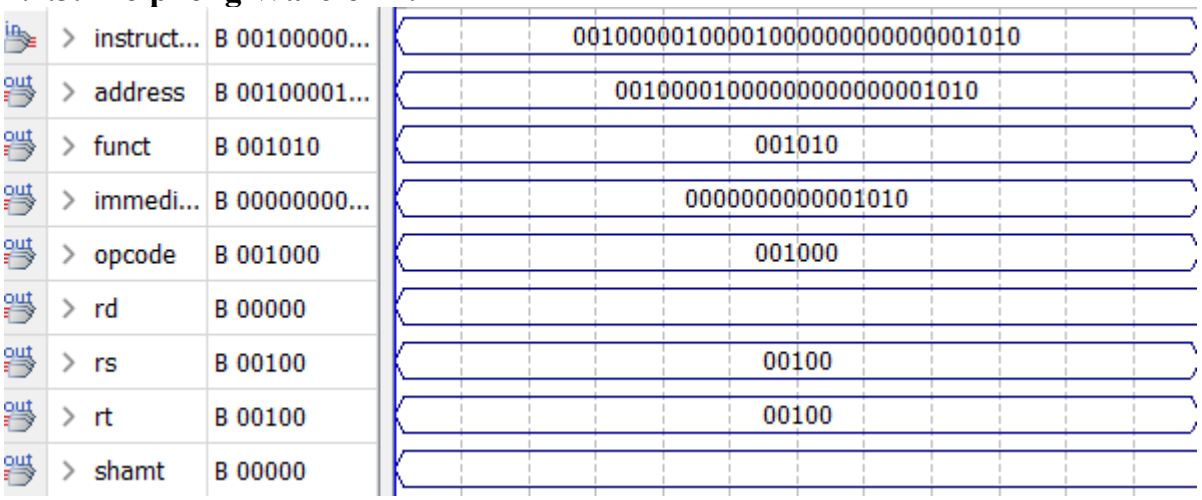
4.1.1. Code thực thi:

```
module InstructionFetch(
    input [31:0] instruction,
    output [5:0] opcode,
    output [4:0] rs,
    output [4:0] rt,
    output [4:0] rd,
    output [4:0] shamt,
    output [5:0] funct,
    output [15:0] immediate,
    output [25:0] address
);
assign opcode = instruction[31:26];
assign rs = instruction[25:21];
assign rt = instruction[20:16];
assign rd = instruction[15:11];
assign shampt = instruction[10:6];
assign funct = instruction[5:0];
assign immediate = instruction[15:0];
assign address = instruction[25:0];
endmodule
```

4.1.2. RTL Viewer:



4.1.3. Mô phỏng Wareform:



4.1.4. Giải thích:

Module InstructionFetch này dùng để **tách** (decode sơ bộ) một **lệnh 32-bit** thành **các thành phần nhỏ** hơn dựa theo **định dạng lệnh của MIPS**.
Một lệnh MIPS chuẩn 32-bit sẽ có thể thuộc **3 loại**:

Loại lệnh	Ý nghĩa
R-type	Lệnh tính toán giữa các thanh ghi (add, sub, and, or, slt,...)
I-type	Lệnh với hằng số hoặc địa chỉ bộ nhớ (lw, sw, beq, addi,...)
J-type	Lệnh nhảy (jump)

Mỗi loại lệnh sẽ dùng các vùng bit khác nhau trong instruction, nên ta cần **tách ra sẵn** như vậy để các module khác dễ xử lý.

Chi tiết các thành phần:

Tên tín hiệu	Vị trí trong instruction	Ý nghĩa
opcode	[31:26] (6 bit cao nhất)	Mã loại lệnh
rs	[25:21]	Thanh ghi nguồn 1
rt	[20:16]	Thanh ghi nguồn 2 (hoặc đích với I-type)
rd	[15:11]	Thanh ghi đích (R-type)
shamt	[10:6]	Độ dịch bit (cho lệnh shift)
funct	[5:0]	Mã hàm con (R-type xác định phép toán cụ thể)
immediate	[15:0]	Giá trị hằng hoặc offset (I-type)
address	[25:0]	Địa chỉ nhảy (J-type)

4.2 Datapath

1. Tổng quan

Module Datapath này là trái tim của bộ xử lý:

- Nó **nhận dữ liệu từ instruction** (rs, rt, rd, immediate, shamt...)
- Thực hiện **các phép tính**, **đọc/ghi bộ nhớ**, và **ghi kết quả lại**.
- Dùng nhiều **khối nhỏ** phối hợp với nhau qua **mux**, **ALU**, **RegisterFile**, **DataMemory**, **SignExtend**..

2. Cấu trúc chi tiết

2.1. Mux2to1_5bit — (Chọn địa chỉ ghi thanh ghi)

```
1  module Mux2to1_5bit(
2      input [4:0] a,
3      input [4:0] b,
4      output [4:0] out,
5      input sel
6  );
7      assign out = (sel) ? (b) : (a);
8  endmodule
```

Chức năng:

- Chọn **địa chỉ ghi**:
  - Nếu lệnh **R-type** → ghi vào **rd** (RegDst = 1).
  - Nếu lệnh **I-type** (ví dụ lw) → ghi vào **rt** (RegDst = 0).

Ví dụ:

- Lệnh add $t0, $t1, $t2: ghi kết quả vào $t0 (rd = 8).
- Lệnh lw $t0, 0($t1): ghi dữ liệu load vào $t0 (rt = 8).

2.2. SignExtend — (Mở rộng số 16 bit thành 32 bit)

```
module SignExtend(
    input wire [4:0] in1,
    input wire [15:0] in2,
    input sel,
    output wire [31:0] out
);
   wire[31:0] temp1, temp2;
   assign temp1 = {{27{in1[4]}}, in1};
   assign temp2 = {{16{in2[15]}}, in2};
   assign out = sel ? temp1 : temp2;
endmodule
```

Chức năng:

- Nếu là lệnh **shift** (ví dụ sll) → dùng shamt (độ dịch bit).
- Nếu là lệnh khác (I-type) → mở rộng immediate thành 32-bit.

Ví dụ:

- Lệnh sll $t0, $t1, 4: shamt = 4.
- Lệnh addi $t0, $t1, 100: immediate = 100.

2.3. RegisterFile — (Tập thanh ghi sử dụng ở Lab04):

```
module RegisterFile(
    input clk,
    input ReadWriteEn,
    input [4:0] ReadAddress1,
    input [4:0] ReadAddress2,
    input [4:0] WriteAddress,
    input [31:0] WriteData,
    output [31:0] ReadData1,
    output [31:0] ReadData2
);
    reg [31:0] registers [0:31];

    // Write operation
    always @(posedge clk) begin
        if (ReadWriteEn && (WriteAddress != 5'd0)) begin
            registers[WriteAddress] <= WriteData;
        end
    end

    // Read operation
    assign ReadData1 = (ReadAddress1 == 5'd0) ? 32'b0 : registers[ReadAddress1];
    assign ReadData2 = (ReadAddress2 == 5'd0) ? 32'b0 : registers[ReadAddress2];
endmodule
```

Chức năng:

- **Đọc** 2 thanh ghi rs, rt.
- **Ghi** kết quả vào địa chỉ đã chọn (mx1) nếu RegWrite = 1.

Ví dụ:

- Lệnh add $t0, $t1, $t2: đọc $t1 và $t2, ghi kết quả vào $t0.

2.4. Mux2to1_32bit — (Chọn nguồn cho ALU)

```
module Mux2to1_32bit(
    input [31:0] a,
    input [31:0] b,
    output [31:0] out,
    input sel
);
    assign out = (sel) ? (b) : (a);
endmodule
```

Chức năng:

- Chọn **toán hạng thứ 2** cho ALU:
  - Nếu lệnh tính toán giữa 2 thanh ghi (add, sub) → chọn **data2**.
  - Nếu lệnh có immediate (addi, lw, sw) → chọn **extend**.

2.5. ALU32bit — (Bộ tính toán ALU của Lab04)

```
module ALU32bit(
    input clk,
    input [31:0] a,
    input [31:0] b,
    input sign1,
    input sign2,
    input [2:0] S,
    output reg [31:0] ALUresult,
    output reg is0
);
    // Biến a và b sẽ được tính toán lại dựa trên sign1, sign2
    wire [31:0] a_modified = (sign1 == 1'b1) ? (a[31] ? -a : a) : a;
    wire [31:0] b_modified = (sign2 == 1'b1) ? (b[31] ? -b : b) : b;

    always @(*) begin
        case(S)
            3'b000: ALUresult <= a_modified + b_modified;          // Cộng
            3'b001: ALUresult <= a_modified - b_modified;          // Trừ
            3'b010: ALUresult <= a_modified & b_modified;          // AND
            3'b011: ALUresult <= a_modified | b_modified;          // OR
            3'b100: ALUresult <= ~(a_modified | b_modified);       // NOR
            3'b101: ALUresult <= a_modified << b_modified[4:0];    // Dịch trái
            3'b110: ALUresult <= a_modified >> b_modified[4:0];    // Dịch phải
            3'b111: ALUresult <= (a_modified < b_modified) ? 32'd1 : 32'd0;  // Kiểm tra nhỏ hơn
            default: ALUresult <= 32'd0;                           // Mặc định là 0
        endcase

        is0 <= (a == b);  // Kiểm tra nếu a == b
    end
endmodule
```

Chức năng:

- Thực hiện phép toán giữa data1 và mx2 dựa trên ALUcontrol:
  - Các phép như cộng, trừ, and, or, so sánh...

Kết quả:

- result: kết quả phép toán.
- beq: flag bằng 0 (phục vụ lệnh beq).

2.6. DataMemory — (Bộ nhớ dữ liệu)

```
1  module DataMemory(
2      input clk,
3      input [31:0] addr,
4      input [31:0] WriteData,
5      output reg [31:0] ReadData,
6      input WriteEn,
7      input ReadEn
8  );
9      reg [31:0] mem [0:1023];
10 always @(posedge clk) begin
11     if(ReadEn) begin
12         ReadData <= mem[addr];
13     end
14     if(WriteEn) begin
15         mem[addr] <= WriteData;
16     end
17 end
18 endmodule
```

Chức năng:

- Nếu MemRead = 1: đọc dữ liệu từ địa chỉ result.
- Nếu MemWrite = 1: ghi dữ liệu data2 vào địa chỉ result.

Ví dụ:

- lw: đọc dữ liệu từ bộ nhớ.
- sw: ghi dữ liệu vào bộ nhớ.

4.2.1. Code thực thi:

```
module Datapath(
    input clk,
    input [4:0] rs,
    input [4:0] rt,
    input [4:0] rd,
    input [4:0] shamt,
    input [15:0] immediate,
    input sign1,
    input sign2,
    input RegDst,
    input RegWrite,
    input ALUsrc,
    input [2:0] ALUcontrol,
    input MemWrite,
    input MemRead,
    input MemtoReg,
    input oe,
    input shift,
    output [31:0] out,
    output beq,
    output [31:0] extend
);

    wire [4:0] mx1;
    wire [31:0] mx2, mx3;
    wire [31:0] data1, data2;
    wire [31:0] result, read;

    // Mux để chọn địa chỉ ghi
    Mux2to1_5bit Mux1(
        .a(rt),
        .b(rd),
        .out(mx1),
        .sel(RegDst)
    );

    // Bộ mở rộng 16-bit (sign/zero extend)
    SignExtend SignExt(
        .in1(shampt),
        .in2(immediate),
        .sel(shift),
        .out(extend)
    );

    // Tập thanh ghi
    RegisterFile Regs(
        .clk(clk),
        .ReadWriteEn(RegWrite),
        .ReadAddress1(rs),
        .ReadAddress2(rt),
        .WriteAddress(mx1),
        .WriteData(mx3),
        .ReadData1(data1),
        .ReadData2(data2)
    );

    // Mux chọn nguồn cho ALU
    Mux2to1_32bit Mux2(
        .a(data2),
        .b(extend),
        .out(mx2),
        .sel(ALUsrc)
    );

    // ALU thực hiện phép tính
    ALU32bit ALU(
```

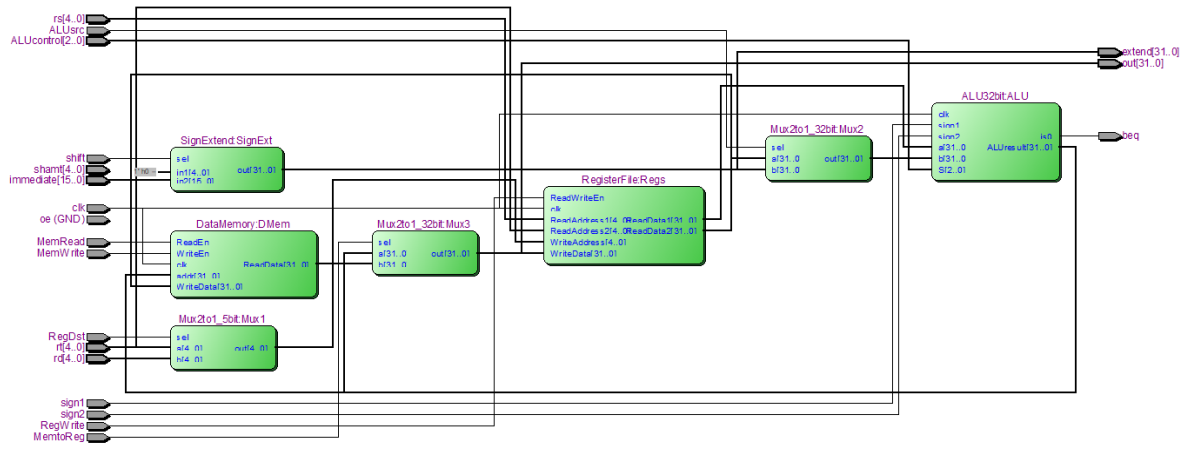
```
        .clk(clk),
        .a(data1),
        .b(mx2),
        .sign1(sign1),
        .sign2(sign2),
        .S(ALUcontrol),
        .ALUresult(result),
        .is0(beq)
    );

    // Bộ nhớ dữ liệu
    DataMemory DMem(
        .clk(clk),
        .addr(result),
        .WriteData(data2),
        .ReadData(read),
        .WriteEn(MemWrite),
        .ReadEn(MemRead)
    );

    // Mux chọn dữ liệu ghi lại vào thanh ghi
    Mux2to1_32bit Mux3(
        .a(result),
        .b(read),
        .out(mx3),
        .sel(MemtoReg)
    );

    // Xuất kết quả nếu được phép
   //assign out = oe ? mx3 : 32'bz;
   assign out = mx3;
endmodule
```

Testbench:

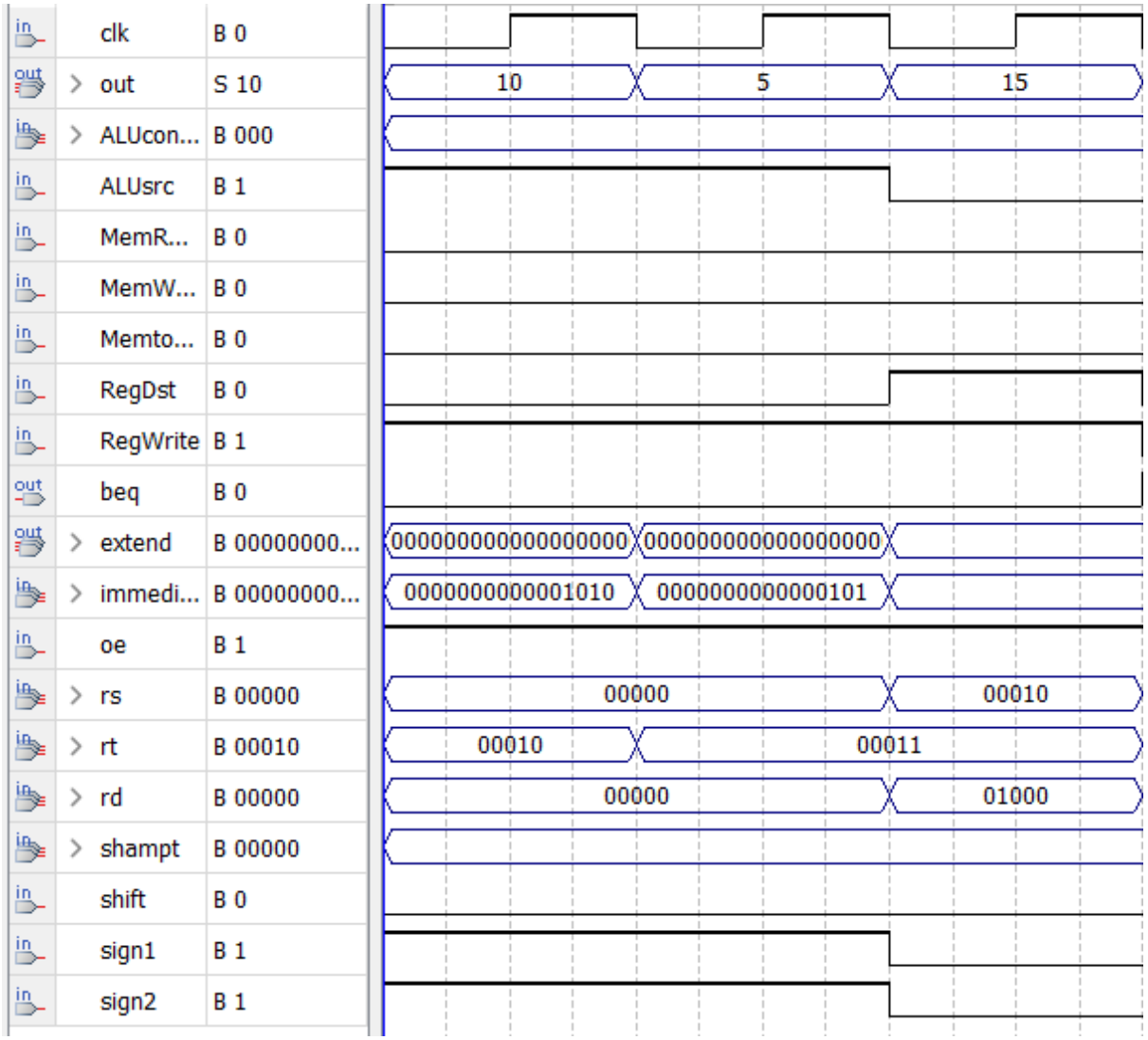
```
`timescale 1ns / 1ps

module Datapath_tb;

    // Inputs
    reg clk = 0;
    reg [4:0] rs, rt, rd, shamt;
    reg [15:0] immediate;
    reg sign1, sign2, RegDst, RegWrite, ALUsrc;
    reg [2:0] ALUcontrol;
    reg MemWrite, MemRead, MemtoReg, oe, shift;

    // Outputs
    wire [31:0] out;
    wire beq;
    wire [31:0] extend;

    // Instantiate the Unit Under Test (UUT)
    Datapath uut (
        .clk(clk), .rs(rs), .rt(rt), .rd(rd), .shamt(shamt),
        .immediate(immediate), .sign1(sign1), .sign2(sign2),
        .RegDst(RegDst), .RegWrite(RegWrite), .ALUsrc(ALUsrc),
        .ALUcontrol(ALUcontrol), .MemWrite(MemWrite), .MemRead(MemRead),
        .MemtoReg(MemtoReg), .oe(oe), .shift(shift),
        .out(out), .beq(beq), .extend(extend)
    );

    // Clock generation
    always #5 clk = ~clk;

    initial begin
        // Initial values
        rs = 5'd0; rt = 5'd2; rd = 5'd0; shamt = 5'd0;
```

```
34          immediate = 16'd10;
35          sign1 = 1; sign2 = 1;
36          RegDst = 0; RegWrite = 1; ALUsrc = 1;
37          ALUcontrol = 3'b000; // Add
38          MemWrite = 0; MemRead = 0; MemtoReg = 0;
39          oe = 1; shift = 0;
40
41          #10;
42          // Change immediate to 5
43           rs = 5'd0; rt = 5'd3; rd = 5'd0; shamt = 5'd0;
44          immediate = 16'd5;
45          sign1 = 1; sign2 = 1;
46          RegDst = 0; RegWrite = 1; ALUsrc = 1;
47          ALUcontrol = 3'b000; // Add
48          MemWrite = 0; MemRead = 0; MemtoReg = 0;
49          oe = 1; shift = 0;
50
51          #10;
52          // Change rd and rt
53          rs = 5'd2;
54          rt = 5'd3;
55          rd = 5'd4;
56          ALUsrc = 0;
57          RegDst = 1;
58           sign1 = 0; sign2 = 0;
59          #10;
60          // Kết thúc mô phỏng
61          $stop;
62      end
63
64  endmodule
```

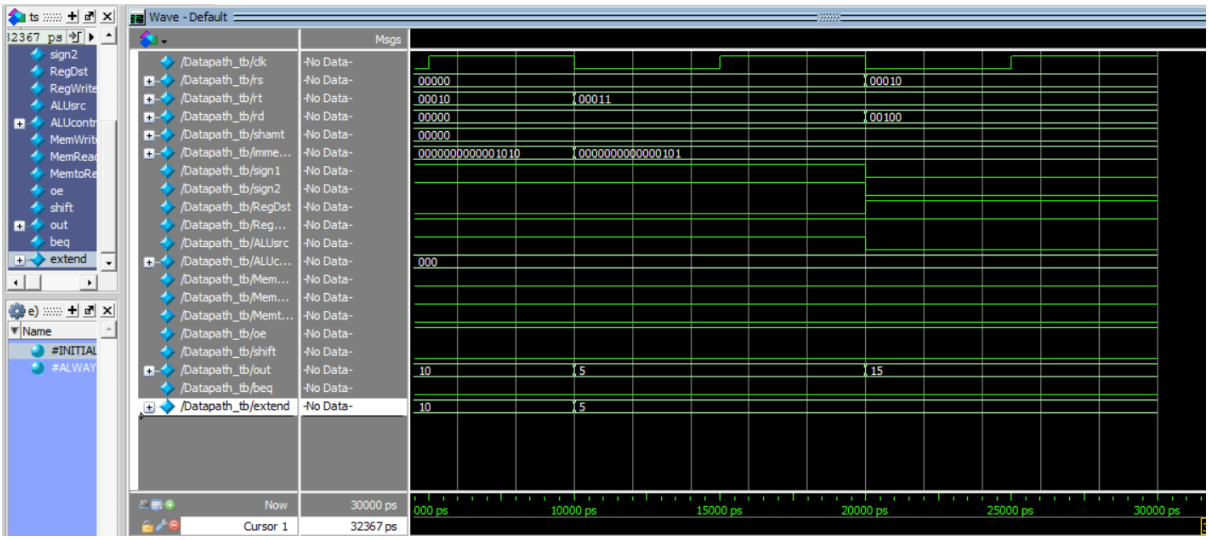
4.2.2. RTL Viewer:



4.2.3. Mô phỏng Wareform:



Testbench:



4.2.4. Giải thích:

Datapath là phần của bộ vi xử lý chịu trách nhiệm **xử lý dữ liệu** và **thực hiện tính toán** theo yêu cầu của chương trình.

Trong thiết kế này, Datapath gồm các khối chính sau:

1. Register File (Tập thanh ghi)

- Chứa nhiều thanh ghi (như $t0, $t1, v.v.).
- Đọc dữ liệu từ 2 thanh ghi (rs, rt) để đưa vào xử lý.
- Sau khi tính toán xong sẽ ghi kết quả vào thanh ghi đích (rd hoặc rt tùy loại lệnh).

2. Sign Extend (Mở rộng bit)

- Nếu cần mở rộng số 16-bit (immediate) thành 32-bit để làm phép toán chính xác.
- Hoặc dùng dịch bit (shamt) nếu là lệnh shift.

3. ALU (Bộ tính toán số học và logic)

- Là trái tim của datapath.
- Nhận 2 đầu vào và thực hiện phép toán:
  - Cộng, trừ, and, or, so sánh nhỏ hơn, dịch bit, v.v.
- Xuất ra kết quả tính toán (ALUresult).

4. Data Memory (Bộ nhớ dữ liệu)

- Chỉ sử dụng cho các lệnh load (lw) và store (sw).
- Đọc hoặc ghi dữ liệu vào bộ nhớ tại địa chỉ do ALU tính ra.

5. Mux (Bộ chọn)

- Dùng để quyết định:
  - Ghi dữ liệu vào đâu (rd hay rt).
  - Lấy dữ liệu nào để tính toán (từ thanh ghi hay immediate).
  - Lấy kết quả nào để ghi lại (ALU hay DataMemory).

Quá trình chung:

1. Instruction chỉ định cần đọc dữ liệu từ thanh ghi nào (rs, rt).
2. Nếu cần, mở rộng số immediate.

3. MUX quyết định đầu vào ALU: thanh ghi thứ 2 hoặc immediate.
4. ALU thực hiện phép toán.
5. Nếu cần, truy cập bộ nhớ đọc/ghi.
6. MUX quyết định ghi dữ liệu nào lại thanh ghi.
7. Kết quả cuối cùng được lưu về thanh ghi.

4.3 Control Unit

4.3.1 Code thực thi:

```
module ControlUnit(
    input clk,
    input [5:0] opcode,
    input [5:0] funct,
    output reg sign1,
    output reg sign2,
    output reg RegDst,
    output reg Jump,
    output reg Branch,
    output reg RegWrite,
    output reg ALUsrc,
    output reg [2:0] ALUop,
    output reg MemWrite,
    output reg MemRead,
    output reg MemtoReg,
    output reg oe,
    output reg shift
);

    always @(posedge clk) begin
        // Reset all control signals
        sign1 <= 0;
        sign2 <= 0;
        RegDst <= 0;
        Jump <= 0;
        Branch <= 0;
        RegWrite <= 0;
        ALUsrc <= 0;
        ALUop <= 3'b000;
        MemWrite <= 0;
        MemRead <= 0;
        MemtoReg <= 0;
        oe <= 0;
        shift <= 0;

        case(opcode)
            6'b100011: begin // lw
                RegDst <= 0;
                RegWrite <= 1;
                ALUsrc <= 1;
                MemRead <= 1;
                MemtoReg <= 1;
                oe <= 1;
            end
            6'b101011: begin // sw
                ALUsrc <= 1;
                MemWrite <= 1;
            end
            6'b000000: begin // R-type
                RegDst <= 1;
                RegWrite <= 1;
                ALUsrc <= 0;
                MemtoReg <= 0;
                oe <= 1;
                case(funct)
                    6'b100000: ALUop <= 3'b000; // add
                    6'b100001: begin           // addu
                        ALUop <= 3'b000;
                        sign1 <= 1;
                        sign2 <= 1;
                    end
                    6'b100010: ALUop <= 3'b001; // sub
                    6'b100011: begin           // subu
                        ALUop <= 3'b001;
                        sign1 <= 1;
                        sign2 <= 1;
```

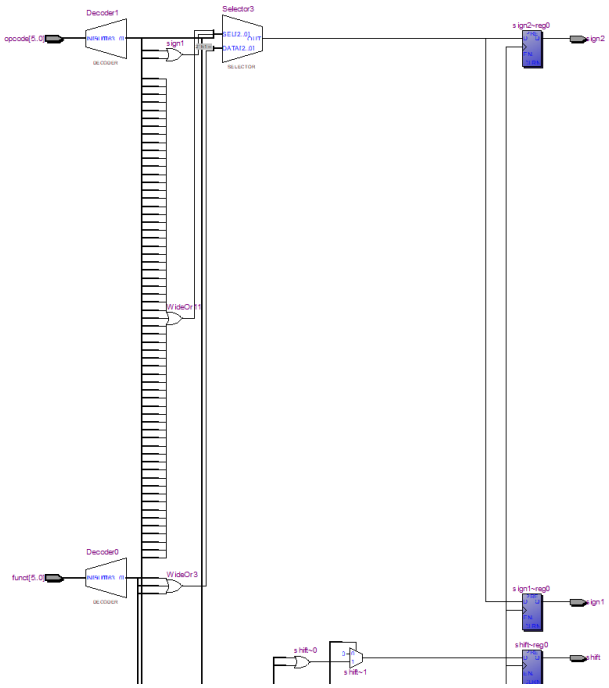
```
                end
                6'b100100: ALUop <= 3'b010; // and
                6'b100101: ALUop <= 3'b011; // or
                6'b100111: ALUop <= 3'b100; // nor
                6'b101010: ALUop <= 3'b111; // slt
                6'b101011: begin           // sltu
                    ALUop <= 3'b111;
                    sign1 <= 1;
                    sign2 <= 1;
                end
                6'b000000: begin           // sll
                    ALUop <= 3'b101;
                    shift <= 1;
                end
                6'b000010: begin           // srl
                    ALUop <= 3'b110;
                    shift <= 1;
                end
            endcase
        end
        6'b000100: begin // beq
            Branch <= 1;
            ALUsrc <= 0;
        end
        6'b000010: begin // j
            Jump <= 1;
        end
        6'b001000: begin // addi
            RegDst <= 0;
            RegWrite <= 1;
            ALUsrc <= 1;
            ALUop <= 3'b000;
            MemtoReg <= 0;
                    oe <= 1;

            end
            6'b001001: begin // addiu
                RegDst <= 0;
                RegWrite <= 1;
                ALUsrc <= 1;
                ALUop <= 3'b000;
                MemtoReg <= 0;
                oe <= 1;
                sign1 <= 1;
                sign2 <= 1;
            end
            6'b001100: begin // andi
                RegDst <= 0;
                RegWrite <= 1;
                ALUsrc <= 1;
                ALUop <= 3'b010;
                MemtoReg <= 0;
                oe <= 1;
            end
            6'b001101: begin // ori
                RegDst <= 0;
                RegWrite <= 1;
                ALUsrc <= 1;
                ALUop <= 3'b011;
                MemtoReg <= 0;
                oe <= 1;
            end
            6'b001010: begin // slti
                RegDst <= 0;
                RegWrite <= 1;
                ALUsrc <= 1;
               ALUop <= 3'b111;
               MemtoReg <= 0;
               oe <= 1;
           end
            6'b001011: begin // sltiu
                RegDst <= 0;
                RegWrite <= 1;
                ALUsrc <= 1;
                ALUop <= 3'b111;
                MemtoReg <= 0;
                oe <= 1;
                sign1 <= 1;
                sign2 <= 1;
            end
        endcase
    end

endmodule
```

Testbench:

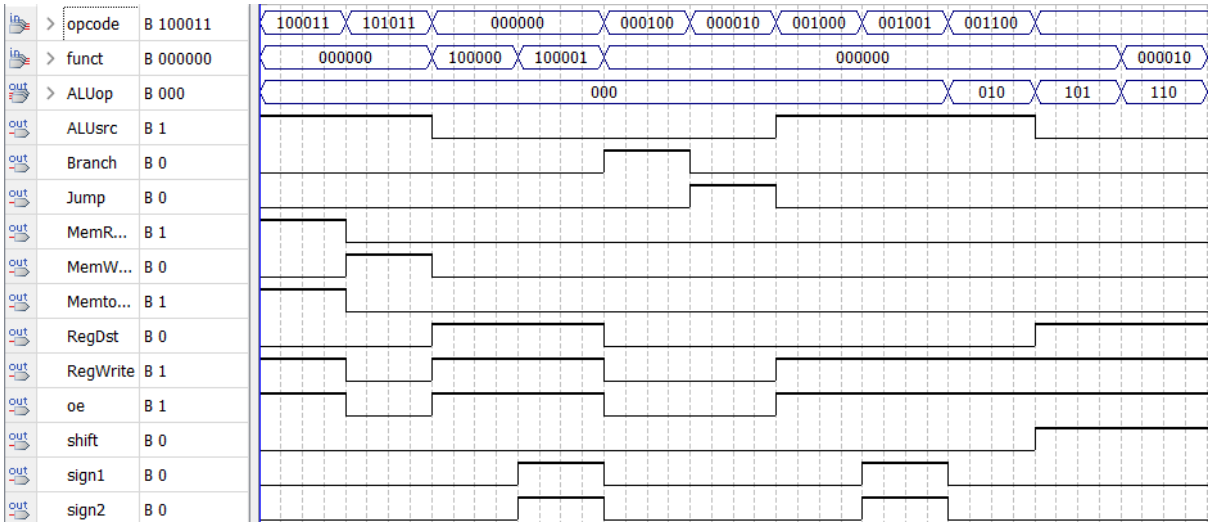
```
1  `timescale 1ns / 1ps
2
3  module ControlUnit_tb;
4
5      // Inputs
6      reg [5:0] opcode;
7      reg [5:0] funct;
8
9      // Outputs
10     wire sign1, sign2, RegDst, Jump, Branch, RegWrite, ALUsrc;
11     wire [2:0] ALUop;
12     wire MemWrite, MemRead, MemtoReg, oe, shift;
13
14     // Instantiate the Unit Under Test (UUT)
15     ControlUnit uut (
16         .opcode(opcode), .funct(funct),
17         .sign1(sign1), .sign2(sign2), .RegDst(RegDst),
18         .Jump(Jump), .Branch(Branch), .RegWrite(RegWrite),
19         .ALUsrc(ALUsrc), .ALUop(ALUop), .MemWrite(MemWrite),
20         .MemRead(MemRead), .MemtoReg(MemtoReg), .oe(oe), .shift(shift)
21     );
22
23     initial begin
24         // Test lw
25         opcode = 6'b100011; funct = 6'bxxxxxx;
26         #10;
27
28         // Test sw
29         opcode = 6'b101011;
30         #10;
31
32         // Test R-type: add
33         opcode = 6'b000000; funct = 6'b100000;
```

```
34          #10;
35
36          // Test R-type: sub
37          opcode = 6'b000000; funct = 6'b100010;
38          #10;
39
40          // Test R-type: sll
41          opcode = 6'b000000; funct = 6'b000000;
42          #10;
43
44          // Test beq
45          opcode = 6'b000100;
46          #10;
47
48          // Test j
49          opcode = 6'b000010;
50          #10;
51
52          // Test addi
53          opcode = 6'b001000;
54          #10;
55
56          // Test sltiu
57          opcode = 6'b001011;
58          #10;
59
60          // Dừng mô phỏng
61          $stop;
62      end
63
64  endmodule
```

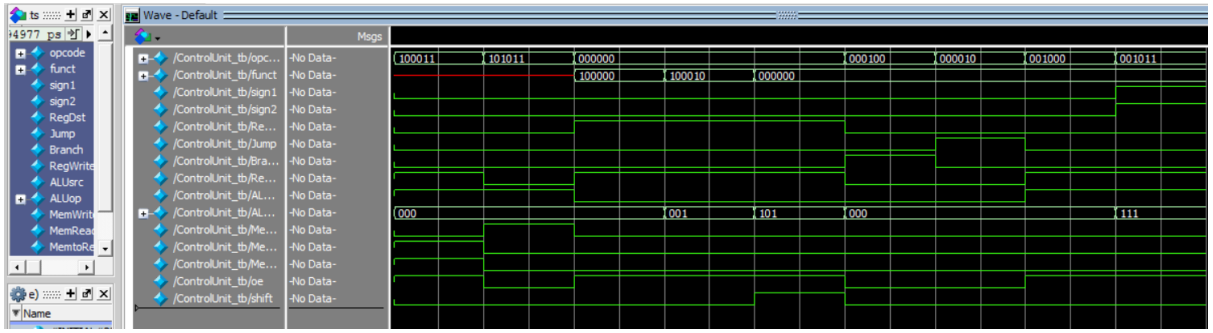
4.3.2 RTL Viewer:



4.3.3 Mô phỏng Wareform:



TestBench:



4.3.4 Giải thích:

Chức năng chung

- **ControlUnit** là bộ phận "điều hành" của bộ xử lý.
- Nó **nhận vào** opcode (và funct đối với R-type) của lệnh máy.
- **Xuất ra** các tín hiệu điều khiển để hướng dẫn Datapath hoạt động đúng cho từng loại lệnh.
- Giống như "bộ não" quyết định datapath nên đi theo luồng nào.

Nguyên lý hoạt động:

1. **Khi có cạnh lên của xung clock (posedge clk)**:
   - Các tín hiệu điều khiển sẽ **reset** về mặc định (gần như 0 hết).
2. **Dựa vào opcode**, Control Unit **kích hoạt** các tín hiệu điều khiển cần thiết.
3. Nếu là lệnh R-type (opcode == 6'b000000), cần đọc thêm funct để biết phép tính nào trong R-type (add, sub, sll, srl, ...).

Cách điều khiển

- **Lệnh lw (load word)**:
  - Cần đọc từ bộ nhớ => MemRead = 1
  - Cần ghi vào thanh ghi => RegWrite = 1
  - Địa chỉ lấy từ ALU (rs + immediate).
  - Dữ liệu lấy từ bộ nhớ, ghi về thanh ghi (nên MemtoReg = 1).
- **Lệnh sw (store word)**:
  - Đọc dữ liệu từ thanh ghi và ghi xuống bộ nhớ => MemWrite = 1
  - Không ghi gì về thanh ghi.

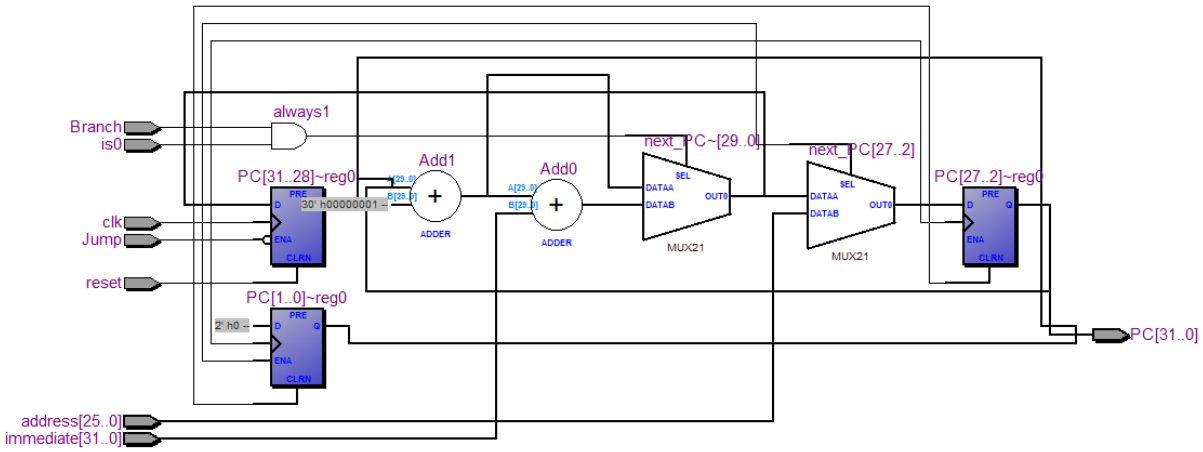
- **R-type** (add, sub, and, or, nor, sll, srl,...):
  - Thực hiện trong ALU.
  - Ghi kết quả vào thanh ghi.
  - RegDst = 1 (đích ghi từ rd).
  - Xác định phép toán bằng funct.
- **Lệnh beq (branch if equal)**:
  - Nếu 2 thanh ghi bằng nhau thì branch.
  - Cần ALU để so sánh (ALUsrc = 0, tính rs - rt).
  - Nếu kết quả ALU bằng 0 thì nhảy.
- **Lệnh j (jump)**:
  - Nhảy đến địa chỉ khác.
  - Jump = 1, không cần ALU.
- **Lệnh cộng với immediate (addi, addiu)**:
  - ALU cộng rs + immediate.
  - Ghi kết quả vào thanh ghi rt.
- **Lệnh logic với immediate (andi, ori)**:
  - ALU thực hiện phép and, or giữa rs và immediate.
- **Lệnh so sánh nhỏ hơn với immediate (slti, sltiu)**:
  - So sánh rs và immediate, ghi 1 hoặc 0 về thanh ghi.

4.4 Program Counter

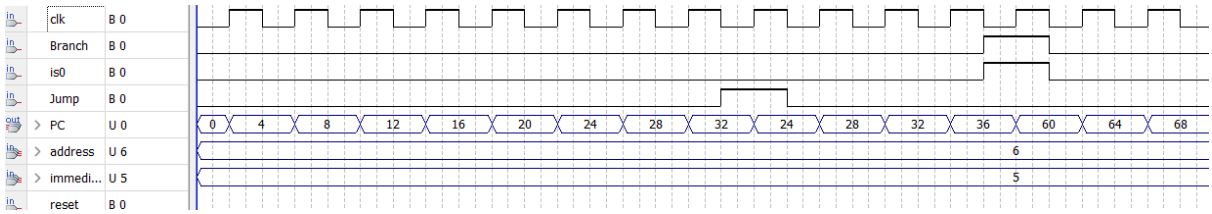
4.4.1 Code thực thi:

```
1  module PC_Counter(
2      input clk,
3      input reset,
4      input Branch,
5      input Jump,
6      input is0,
7      input [25:0] address,
8      input [31:0] immediate,
9      output reg [31:0] PC
10 );
11
12     reg [31:0] next_PC;
13
14     always @(posedge clk or posedge reset) begin
15         if (reset) begin
16             PC <= 32'd0;
17         end else begin
18             PC <= next_PC;
19         end
20     end
21
22     always @(*) begin
23         if (Jump) begin
24             next_PC = {PC[31:28], address, 2'b00};
25         end else if (Branch && is0) begin
26             next_PC = PC + 4 + (immediate << 2);
27         end else begin
28             next_PC = PC + 4;
29         end
30     end
31
32 endmodule
```

4.4.2 RTL Viewer:



4.4.3 Mô phỏng Wareform:



4.4.4 Giải thích:

Module PC_Counter chịu trách nhiệm quản lý và tính toán **Program Counter (PC)** của bộ xử lý. **PC** là thanh ghi lưu trữ địa chỉ của lệnh tiếp theo sẽ được thực thi. Mỗi

chu kỳ đồng hồ, PC được cập nhật để trỏ tới lệnh kế tiếp, hoặc có thể thay đổi nếu có nhánh (branch) hoặc nhảy (jump).

Cấu trúc và hoạt động:

1.Khối always @(posedge clk or posedge reset):

- Khi có **cạnh lên của xung đồng hồ** (clk) hoặc có tín hiệu **reset**:
  - Nếu **reset** được kích hoạt, **PC** sẽ được đặt lại về 0 (32'd0).
  - Nếu không, **PC** sẽ nhận giá trị của **next_PC**.

2.Khối always @(*):

- Xử lý tín hiệu điều khiển **Jump**, **Branch** và **is0** để tính toán giá trị mới cho **next_PC**.
- **Điều kiện Jump**:
  - Nếu **Jump** được kích hoạt, PC sẽ nhảy đến địa chỉ mới. Địa chỉ này được tính bằng cách lấy 4 bit cao nhất của **PC**, nối với 26 bit **address** từ lệnh Jump và thêm 2'b00 để căn chỉnh lại địa chỉ (do trong bộ xử lý MIPS, địa chỉ được căn chỉnh bằng từ 4 byte).
  - **Công thức**:
    next_PC = {PC[31:28], address, 2'b00};
- **Điều kiện Branch**:
  - Nếu **Branch** được kích hoạt và **is0** (kết quả so sánh ALU) là 0, tức là điều kiện nhánh đúng, thì **next_PC** sẽ được tính toán lại bằng cách cộng giá trị của **PC** hiện tại với **immediate** (đã dịch trái 2 bit để tương ứng với đơn vị từ 4 byte).
- **Công thức**:
  next_PC = PC + 4 + (immediate << 2);
- **Điều kiện mặc định** (Không phải Jump, không phải Branch):
- **PC** sẽ tự tăng lên 4 để trỏ tới lệnh kế tiếp.
- **Công thức**:

next_PC = PC + 4;

Tóm tắt hoạt động:

- **PC** được cập nhật mỗi chu kỳ đồng hồ dựa trên **next_PC**, tính toán từ các điều kiện Jump hoặc Branch.
- Nếu không có nhảy hay nhánh, **PC** chỉ đơn giản tăng lên 4 để trỏ tới lệnh tiếp theo.
- Nếu có nhảy (Jump), **PC** được cập nhật bằng địa chỉ nhảy.
- Nếu có nhánh (Branch) và điều kiện nhánh đúng, **PC** được cập nhật bằng địa chỉ nhánh.

4.5/ Khối FinalProcessor hoàn chỉnh:

Module FinalProcessor là một hệ thống hoàn chỉnh bao gồm các thành phần chính trong bộ xử lý: **Instruction Fetch**, **Control Unit**, **Datapath**, và **PC Counter**. Mục đích của module này là xử lý các lệnh (instructions) và cung cấp kết quả đầu ra cho các phép toán hoặc dữ liệu từ bộ nhớ.

Code thực thi:

```v
module FinalProcessor(
    input clk,
    input reset,
    input [31:0] instruction,
    output [31:0] result_out,  // Output để quan sát kết quả
    output [31:0] PC
);


    // ------------------- Instruction Fetch -------------------
    wire [5:0] opcode;
    wire [4:0] rs;
    wire [4:0] rt;
    wire [4:0] rd;
    wire [4:0] shamt;
    wire [5:0] funct;
    wire [15:0] imm16;
    wire [25:0] addr26;

    InstructionFetch IF (
        .instruction(instruction),
        .opcode(opcode),
        .rs(rs),
        .rt(rt),
        .rd(rd),
        .shamt(shamt),
        .funct(funct),
        .immediate(imm16),
        .address(addr26)
    );

    // ------------------- Control Unit -------------------
    wire sign1, sign2, RegDst, Jump, Branch, RegWrite, ALUsrc, MemWrite, MemRead, MemtoReg, oe, shift;
    wire [2:0] ALUop;

    ControlUnit CU (
        .clk(clk),
        .opcode(opcode),
        .funct(funct),
        .sign1(sign1),
        .sign2(sign2),
        .RegDst(RegDst),
        .Jump(Jump),
        .Branch(Branch),
        .RegWrite(RegWrite),
        .ALUsrc(ALUsrc),
        .ALUop(ALUop),
        .MemWrite(MemWrite),
        .MemRead(MemRead),
        .MemtoReg(MemtoReg),
        .oe(oe),
        .shift(shift)
    );

    // ------------------- Datapath -------------------
    wire beq;
    wire [31:0] datapath_out;
    wire [31:0] extend;

    Datapath datapath (
        .clk(clk),
        .rs(rs),
        .rt(rt),
        .rd(rd),
        .shamt(shamt),
        .immediate(imm16),
```

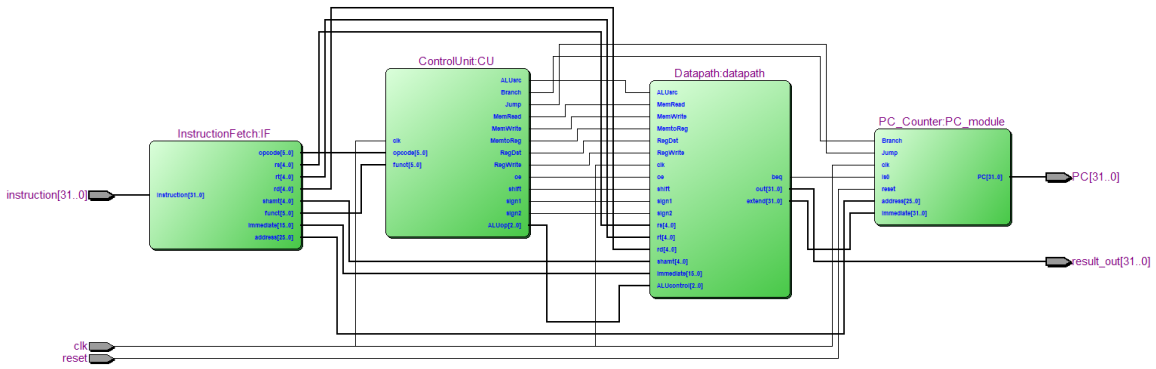
```
        .sign1(sign1),
        .sign2(sign2),
        .RegDst(RegDst),
        .RegWrite(RegWrite),
        .ALUsrc(ALUsrc),
        .ALUcontrol(ALUop),
        .MemWrite(MemWrite),
        .MemRead(MemRead),
        .MemtoReg(MemtoReg),
        .oe(oe),
        .shift(shift),
        .out(datapath_out),
        .beq(beq),
        .extend(extend)
    );

    // ------------------- PC Counter -------------------
    PC_Counter PC_module (
        .clk(clk),
        .reset(reset),
        .Branch(Branch),
        .Jump(Jump),
        .is0(beq),
        .address(addr26),
        .immediate(extend),
        .PC(PC)
    );

    // ------------------- Output -------------------
   assign result_out = datapath_out;
endmodule
```

TestBench:

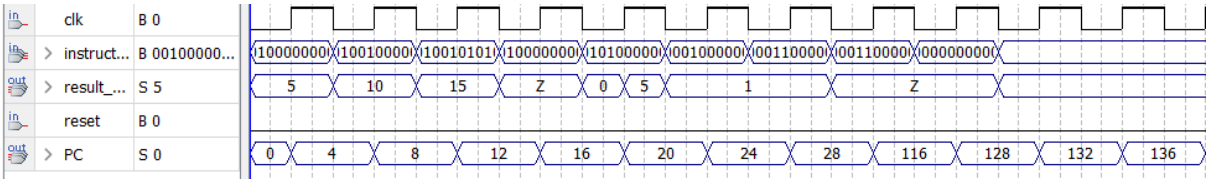
```
`timescale 1ns / 1ps

module FinalProcessor_tb;

    // ------------------ Inputs -----------------
    reg clk;
    reg reset;
    reg [31:0] instruction;

    // ------------------ Outputs ----------------
    wire [31:0] result_out;
    wire [31:0] PC;

    // ------------------ Instantiate FinalProcessor ------------------
    FinalProcessor uut (
        .clk(clk),
        .reset(reset),
        .instruction(instruction),
        .result_out(result_out),
        .PC(PC)
    );

    // ------------------ Clock Generation -----------------
    always begin
        #5 clk = ~clk;  // Toggle clock every 5ns
    end

    // ------------------ Test Stimulus ----------------
    initial begin
        // Initialize inputs
        clk = 0;
        reset = 0;
        instruction = 32'b0;
```

```
34
35          // Reset the processor
36          reset = 1;
37          #10 reset = 0;
38
39          // ------------------ Apply Test Instructions -----------------
40
41          instruction = 32'b00100000000001000000000000000101; // 001000 00000 00010 0000000000000101
42          #10;
43
44          instruction = 32'b00100000000001001000000000001010; // 001000 00000 00011 0000000000001010
45          #10;
46
47          instruction = 32'b00000001000010010101000000100000; // 000000 01000 01001 01010 00000 100000
48          #10;
49
50          instruction = 32'b10101101001010000000000000000000; // 101011 01001 01000 0000000000000000
51          #10;
52
53          instruction = 32'b10001101001010100000000000000000; // 100011 01001 01010 0000000000000000
54          #10;
55
56          instruction = 32'b00100000000000100000000000000001; // 001000 00000 00010 0000000000000001
57          #10;
58
59          instruction = 32'b00100000000000110000000000000001; // 001000 00000 00011 0000000000000001
60          #10;
61
62          instruction = 32'b00010000010000110000000000010101; // 000100 00010 00011 0000000000010101
63          #10;
64
65          instruction = 32'b00000100000000000000000000100000; // 000010 00000000000000000000000000
66          #10;
67
68          // Finish simulation after testing all instructions
69          $finish;
70      end
71
72      // ------------------ Monitor the Output -----------------
73      initial begin
74          $monitor("Time: %0d, PC: %h, Result: %h", $time, PC, result_out);
75      end
76
77  endmodule
78
```

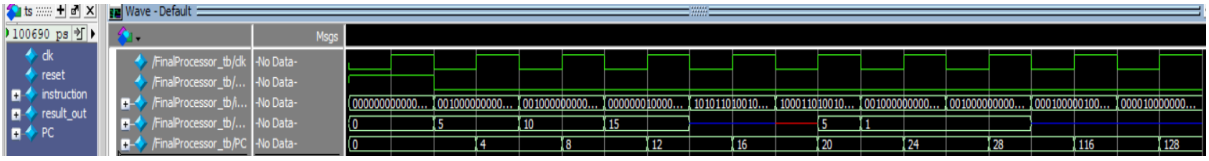
RTL Viewer:



Mô phỏng Wareform



TestBench:



Các lệnh thử:

Mã nhị phân	Lệnh MIPS	Giải thích ngắn gọn
001000 00000 01000 0000000000000101	addi $t0, $zero, 5	Gán $t0 = 5
001000 00000 01001 0000000000001010	addi $t1, $zero, 10	Gán $t1 = 10
000000 01000 01001 01010 00000 100000	add $t2, $t0, $t1	$t2 = $t0 + $t1
101011 01001 01000 0000000000000000	sw $t0, 0($t1)	Lưu giá trị $t0 vào địa chỉ [$t1 + 0]
100011 01001 01010 0000000000000000	lw $t2, 0($t1)	Tải dữ liệu từ địa chỉ [$t1 + 0] vào $t2
001000 00000 00010 0000000000000001	addi $v0, $zero, 1	Gán $v0 = 1
001000 00000 00011 0000000000000001	addi $v1, $zero, 1	Gán $v1 = 1
000100 00010 00011 0000000000010101	beq $v0, $v1, 21	Nếu $v0 == $v1 thì nhảy tới PC + 4 + (21 × 4)
000010 00000000000000000000100000	j 0x00000080	Nhảy tới địa chỉ 0x00000080

Giải thích:

Các tín hiệu đầu vào:

1. **clk**: Tín hiệu xung đồng hồ, được dùng để đồng bộ hóa các thành phần của hệ thống.
2. **reset**: Tín hiệu reset, giúp đưa hệ thống về trạng thái ban đầu.
3. **instruction**: Lệnh (32-bit) đầu vào mà bộ xử lý sẽ thực thi.

Các tín hiệu đầu ra:

1. **result_out**: Kết quả cuối cùng của phép toán hoặc giá trị đọc từ bộ nhớ sau khi xử lý.
2. **PC**: Địa chỉ của lệnh tiếp theo trong chu trình thực thi.

Các thành phần chính của module:

1. **Instruction Fetch**:
    - Module InstructionFetch lấy thông tin từ lệnh đầu vào (instruction), và chia nó thành các trường riêng biệt:
        - **opcode**: 6 bit đầu tiên của lệnh, xác định loại lệnh.
        - **rs**: Địa chỉ thanh ghi nguồn thứ nhất.
        - **rt**: Địa chỉ thanh ghi nguồn thứ hai.
        - **rd**: Địa chỉ thanh ghi đích.

        - **shamt**: Địa chỉ dịch (shift amount) cho các lệnh dịch (shift).
        - **funct**: 6 bit cuối của lệnh, xác định chi tiết hành động cần thực hiện (chỉ có trong các lệnh R-type).
        - **immediate**: Tham số immediate (dành cho các lệnh như addi, andi).
        - **address**: Địa chỉ dùng trong lệnh Jump.
2. **Control Unit (CU)**:
    - Dựa trên **opcode** và **funct** từ lệnh, **Control Unit** sẽ tạo ra các tín hiệu điều khiển:
        - **RegDst**: Điều khiển việc chọn thanh ghi đích (rd hoặc rt).
        - **Jump**: Điều khiển lệnh nhảy (jump).
        - **Branch**: Điều khiển lệnh nhánh (branch).
        - **RegWrite**: Điều khiển việc ghi vào thanh ghi.
        - **ALUsrc**: Điều khiển việc chọn toán hạng thứ hai cho ALU (có thể là dữ liệu từ thanh ghi hoặc immediate).
        - **ALUop**: Tín hiệu điều khiển ALU, xác định phép toán ALU.
        - **MemWrite**: Điều khiển việc ghi vào bộ nhớ.
        - **MemRead**: Điều khiển việc đọc từ bộ nhớ.
        - **MemtoReg**: Điều khiển việc chọn dữ liệu ghi vào thanh ghi (từ ALU hoặc bộ nhớ).
        - **oe**: Tín hiệu cho phép xuất ra dữ liệu.
        - **shift**: Điều khiển các lệnh dịch (shift).
3. **Datapath**:
    - **Datapath** thực hiện các phép toán và thao tác dữ liệu:
        - **ALU** thực hiện các phép toán (thêm, trừ, AND, OR, v.v.).
        - **Register File** lưu trữ và truy xuất dữ liệu từ các thanh ghi.
        - **Muxes** (Multiplexers) điều khiển việc lựa chọn các đầu vào cho ALU và các thanh ghi.
        - **Memory** thực hiện việc đọc/ghi từ bộ nhớ dữ liệu.
        - **ALUresult** là kết quả của phép toán ALU, có thể được sử dụng để ghi vào thanh ghi hoặc bộ nhớ.
4. **PC Counter**:
    - **PC_Counter** quản lý giá trị **Program Counter (PC)**:
        - **Jump**: Khi có lệnh nhảy (jump), PC sẽ được thay đổi theo địa chỉ Jump.
        - **Branch**: Khi có lệnh nhánh (branch), nếu điều kiện đúng (is0), PC sẽ được cập nhật với địa chỉ nhánh.
        - **PC** tăng lên 4 đơn vị (tính theo từ 4 byte) khi không có nhảy hay nhánh.
5. **Output**:
    - **result_out**: Đây là kết quả cuối cùng của hệ thống, được lấy từ datapath_out để quan sát kết quả của lệnh thực thi.
    - **PC**: Địa chỉ của lệnh tiếp theo, giúp theo dõi tiến trình thực thi chương trình.

Hoạt động tổng quan:

1. **Instruction Fetch**: Lệnh được phân tích thành các trường thông qua

InstructionFetch, bao gồm opcode, rs, rt, rd, và các thành phần khác.

2. **Control Unit**: Dựa trên opcode và funct, ControlUnit sẽ phát ra tín hiệu điều khiển cho các thành phần khác.
3. **Datapath**: Các tín hiệu điều khiển từ ControlUnit sẽ điều khiển hoạt động của các thanh ghi, ALU, bộ nhớ, và multiplexers để thực hiện các phép toán hoặc thao tác dữ liệu.
4. **PC Counter**: Dựa vào các tín hiệu như Jump, Branch, PC sẽ được cập nhật để trỏ tới lệnh tiếp theo hoặc một lệnh khác nếu có nhảy hoặc nhánh.
5. **Output**: Cuối cùng, kết quả của phép toán hoặc dữ liệu từ bộ nhớ sẽ được đưa ra ngoài thông qua result_out.

Kết luận:

Module FinalProcessor là một hệ thống đơn giản để thực hiện các lệnh trong bộ xử lý. Nó tích hợp các thành phần cơ bản của một bộ xử lý như điều khiển lệnh, xử lý dữ liệu, và cập nhật giá trị **Program Counter (PC)**. Mỗi thành phần hoạt động kết hợp với nhau để thực hiện các phép toán hoặc thao tác bộ nhớ theo yêu cầu của từng lệnh.

5. Nguyên lý hoạt động tổng thể

- Bộ xử lý khởi động bằng reset.
- PC Counter cung cấp địa chỉ lệnh.
- Instruction Fetch nạp lệnh tương ứng.
- Control Unit phân tích opcode/funct sinh tín hiệu điều khiển.
- Datapath xử lý phép toán hoặc đọc/ghi bộ nhớ.
- Kết quả được ghi vào Register File hoặc xuất ra ngoài.
- PC cập nhật để xử lý lệnh kế tiếp.

6. Kết luận

Thông qua việc thiết kế bộ xử lý này, chúng ta có thể thấy được:

- Vai trò then chốt của từng thành phần riêng biệt (Instruction Fetch, Control, Datapath, PC Counter).
- Mối quan hệ chặt chẽ giữa việc giải mã lệnh, điều khiển tín hiệu và thực hiện luồng dữ liệu.
- Sự cần thiết của việc đồng bộ hóa các hành động trong một chu kỳ xung nhịp.

Kiến trúc này đóng vai trò tiền đề để phát triển các hệ thống xử lý cao cấp hơn như:

- Pipeline Processor (chia nhỏ quá trình thành nhiều giai đoạn),
- Superscalar Processor (thực thi nhiều lệnh cùng lúc),
- Out-of-order Execution (thực thi không tuần tự).

7. Ưu điểm của thiết kế

- **Đơn giản, dễ hiểu**:
  Kiến trúc Single-Cycle giúp việc thiết kế và phân tích hoạt động của processor trở nên trực quan. Mỗi lệnh hoàn thành trong một chu kỳ, dòng dữ liệu và tín hiệu điều khiển rõ ràng.
- **Dễ dàng kiểm thử và mô phỏng**:
  Các thành phần độc lập và đơn giản (PC Counter, Instruction Fetch, Control Unit, Datapath) giúp dễ dàng test từng module riêng biệt trước khi tích hợp toàn hệ thống.
- **Nền tảng vững chắc cho các kiến trúc cao cấp**:
  Việc hiểu sâu cách thức vận hành Single-Cycle processor là bước chuẩn bị quan trọng để tiếp cận các kiến trúc phức tạp hơn như pipeline, multi-core.

- **Khả năng mở rộng cao**:
  Hệ thống có thể dễ dàng mở rộng thêm các loại lệnh khác (ví dụ: lệnh nhân, chia, lệnh dịch trái/phải nâng cao) bằng cách cập nhật Control Unit và bổ sung logic trong Datapath.

8. Hạn chế của thiết kế

- **Thời gian chu kỳ (Clock Cycle Time) dài**:
  Vì mỗi lệnh, bất kể đơn giản hay phức tạp, đều phải hoàn thành trong **một chu kỳ**, nên thời gian chu kỳ bị quyết định bởi lệnh phức tạp nhất (ví dụ: lệnh truy xuất bộ nhớ).
- **Hiệu suất sử dụng tài nguyên thấp**:
  Trong mỗi chu kỳ, nhiều thành phần phần cứng có thể bị lãng phí (ví dụ ALU không cần thiết cho lệnh load/store nhưng vẫn phải hoạt động).
- **Khó mở rộng quy mô hệ thống**:
  Khi tập lệnh lớn hơn, các tín hiệu điều khiển trở nên phức tạp hơn, dẫn tới khả năng phát sinh lỗi trong thiết kế.
- **Không thích hợp cho các hệ thống yêu cầu tốc độ cao**:
  Trong thực tế, các CPU hiện đại thường sử dụng kiến trúc **pipelining** hoặc **multi-cycle execution** để cải thiện hiệu năng. Kiến trúc single-cycle chỉ phù hợp cho mục đích giáo dục hoặc các ứng dụng đơn giản.

9. Phân tích hiệu năng

9.1 Thông số hiệu năng cơ bản

- **CPI (Cycles Per Instruction)**:
  = 1 chu kỳ / 1 lệnh (vì là Single-Cycle Processor).
- **Thời gian thực hiện 1 lệnh (Instruction Latency)**:
  = Thời gian của 1 chu kỳ xung nhịp.
- **Thông lượng (Throughput)**:
  = 1 lệnh / 1 chu kỳ.

9.2 Yếu tố ảnh hưởng hiệu năng

- **Độ dài đường dữ liệu (Critical Path)**:
  Thời gian chu kỳ bị chi phối bởi đường dữ liệu dài nhất, thường bao gồm: Thanh ghi đọc → ALU thực thi → Bộ nhớ dữ liệu → Thanh ghi ghi
- **Độ phức tạp lệnh**:
  Các lệnh đơn giản (ví dụ: cộng 2 số) phải chịu thời gian chờ tương đương các lệnh phức tạp (ví dụ: load từ bộ nhớ), gây lãng phí tài nguyên và giảm hiệu quả tổng thể.
- **Tốc độ bộ nhớ**:
  Bộ nhớ lệnh và bộ nhớ dữ liệu cần phải đủ nhanh để đáp ứng kịp chu kỳ xử lý, nếu không sẽ gây bottleneck.

9.3 Tổng kết hiệu năng

- Trong môi trường lý tưởng (không có lỗi truy cập bộ nhớ, thời gian ALU tối ưu), hiệu năng đạt mức chấp nhận được cho các ứng dụng đơn giản.
- Khi cần mở rộng lên ứng dụng thực tế yêu cầu hiệu suất cao (ví dụ xử lý đa phương tiện, game, AI), cần chuyển sang các kiến trúc tiên tiến hơn (pipelined, superscalar).