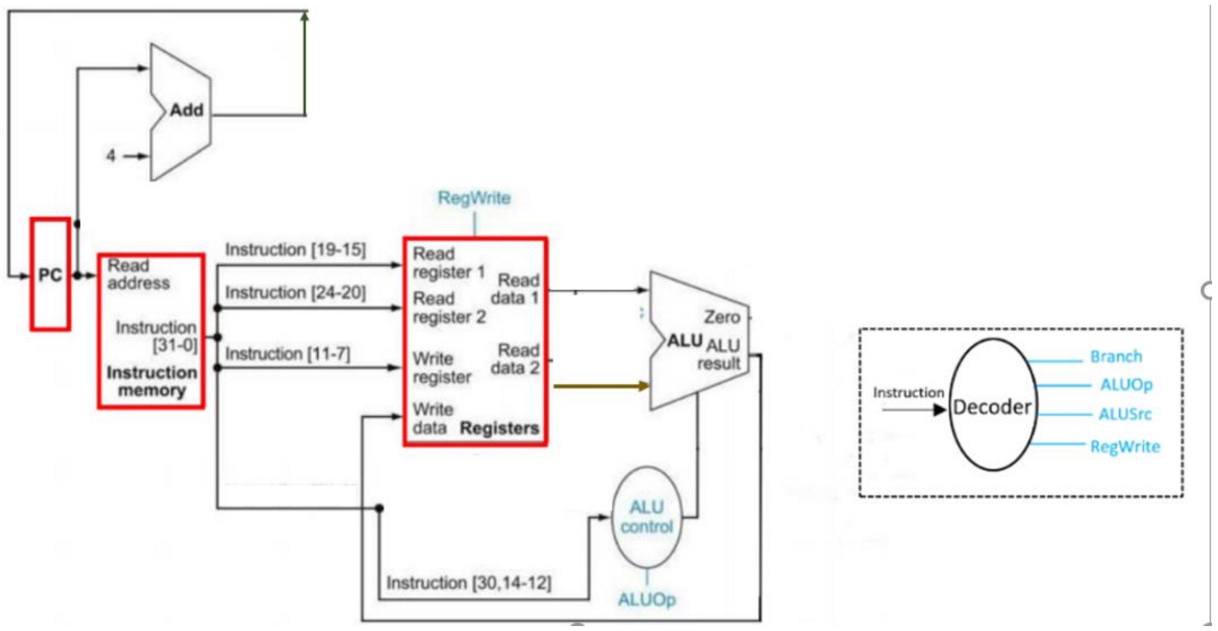
Computer Organization

0711099 林佑檍

0810749 張君實

Architecture diagram:



Detailed description of the implementation:

1.adder 的實作:

```
module Adder(
    input  [32-1:0] src1_i,
        input  [32-1:0] src2_i,
        output [32-1:0] sum_o
        );

/* Write your code HERE */

reg[31:0] result;
assign sum_o = result;

always @(*) begin
        result = src1_i + src2_i;
end

endmodule
```

在 Adder.v 中，Input src1_i 和 src2_i 如果有改變，就對他們相加並 output 至 sum_o。

2.alu 的實作:

```
module alu(
	input                   rst_n,         // negative reset            (input)
	input signed [32-1:0]   src1,          // 32 bits source 1          (input)
	input signed [32-1:0]   src2,          // 32 bits source 2          (input)
	input        [ 4-1:0]   ALU_control,   // 4 bits ALU control input  (input)
	output reg   [32-1:0]   result,        // 32 bits result            (output)
	output reg              zero,          // 1 bit when the output is 0, zero must be set (output)
	output reg              cout,          // 1 bit carry out           (output)
	output reg              overflow       // 1 bit overflow            (output)
	);

/* Write your code HERE */

always @(*) begin
	if(~rst_n) begin        //reset
		result   <= 0;
		zero     <= 0;
		cout     <= 0;
		overflow <= 0;
	end
	else begin
		case (ALU_control)
			4'b0000: begin  //AND
				result = src1&src2;
			end
			4'b0001: begin  //OR
				result = src1|src2;
			end
			4'b0010: begin  //add
				result = src1+src2;
				if(src1[31]==src2[31] && src1[31]!=result[31]) begin
					overflow = 1;
				end
			end
			4'b0110: begin  //sub (beq)
				result = src1-src2;
				if(src1[31]==src2[31] && src1[31]!=result[31]) begin
					overflow = 1;
				end
			end
			4'b0111: begin  //xor
				result = src1^src2;
			end
			4'b1000: begin  //slt
				result = (src1<src2 ? 1:0);
			end
			4'b1010: begin  //shift right
				result = src1 >>> src2;
			end
			4'b1100: begin  //shift left
				result = src1 << src2;
			end
			default: begin
				result = 0;
			end
		endcase
		zero = !result;
	end
end
```

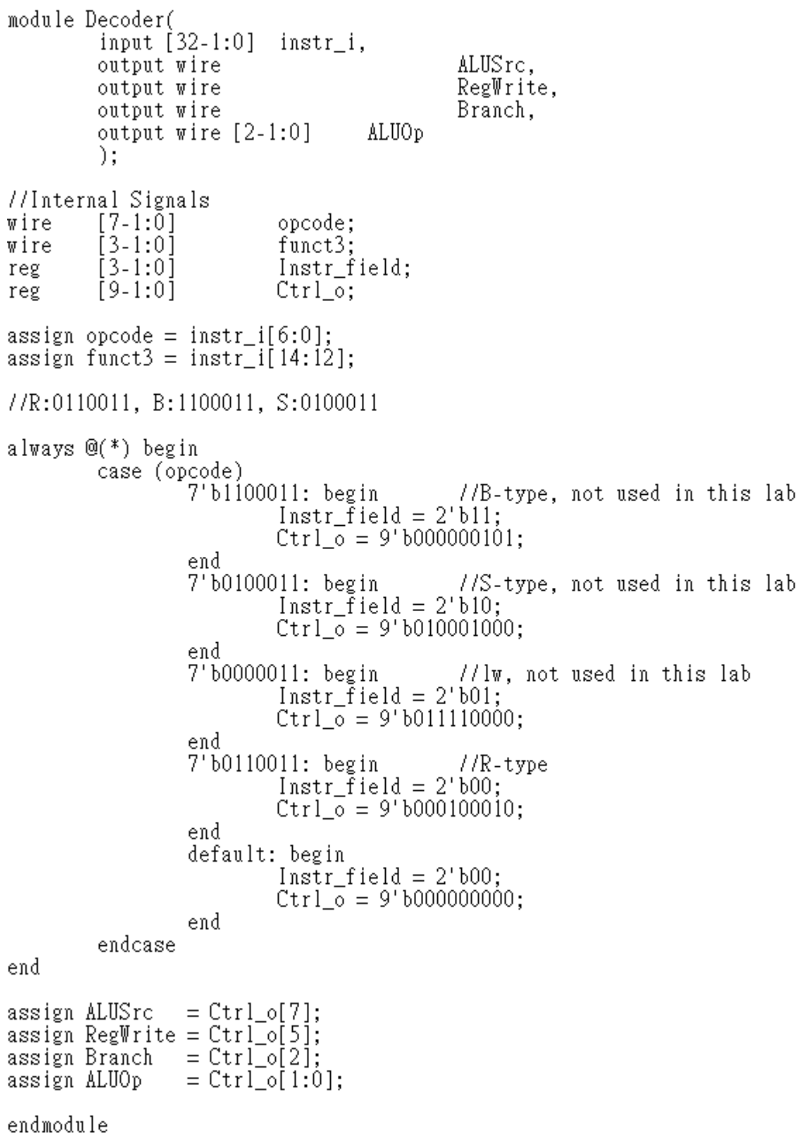
在 alu.v 中，假設 input 有變動，首先先看 rst_n 是不是 1，不是的話將所有 output 歸零，否則接著看 ALU_control，對於不同的 ALU_control 做不同的運算。

3. .alu control 的實作:

```
module ALU_Ctrl(
        input   [4-1:0] instr,
        input   [2-1:0] ALUOp,
        output  reg [4-1:0] ALU_Ctrl_o
        );

/* Write your code HERE */
reg [4-1:0] ctrl;

always @(*) begin
        case (ALUOp)
                2'b00: begin    //S-type
                        ctrl = 4'b0010; //add
                end
                2'b01: begin    //B-type
                        ctrl = 4'b0110; //sub
                end
                default: begin  //R-type
                        case (instr[2:0])
                                3'b000: begin   //add, sub
                                        ctrl = 4'b0010;
                                        ctrl[2] = (instr[3]==1 ? 1:0);  //from funct7 field
                                end
                                3'b111: begin   //AND
                                        ctrl = 4'b0000;
                                end
                                3'b110: begin   //OR
                                        ctrl = 4'b0001;
                                end
                                3'b100: begin   //xor
                                        ctrl = 4'b0111;
                                end
                                3'b010: begin   //set less than
                                        ctrl = 4'b1000;
                                end
                                3'b001: begin   //shift left
                                        ctrl = 4'b1100;
                                end
                                3'b101: begin   //shift right
                                        ctrl = 4'b1010;
                                end
                                default: begin
                                        ctrl = 4'b1111;
                                end
                        endcase
                end
        endcase
        ALU_Ctrl_o = ctrl;
end

endmodule
```

在 ALU_Ctrl.v 中，我們 input 一段 instruction 和 ALUOp。假如 input 有變動，首先先看 ALUOp，如果是 00，那就是 S-type，output add alu control。如果 ALUOp 是 01，那就是 B-type，就 output sub alu control。如果 ALUOp 是 1X，那就是 R-type，接著從 instruction[2:0] 中看是哪種運算，output 對應的 alu control。

4.decoder 的實作:



```
module Decoder(
        input [32-1:0]  instr_i,
        output wire                     ALUSrc,
        output wire                     RegWrite,
        output wire                     Branch,
        output wire [2-1:0]     ALUOp
        );

//Internal Signals
wire    [7-1:0]         opcode;
wire    [3-1:0]         funct3;
reg     [3-1:0]         Instr_field;
reg     [9-1:0]         Ctrl_o;

assign opcode = instr_i[6:0];
assign funct3 = instr_i[14:12];

//R:0110011, B:1100011, S:0100011

always @(*) begin
        case (opcode)
                7'b1100011: begin       //B-type, not used in this lab
                        Instr_field = 2'b11;
                        Ctrl_o = 9'b000000101;
                end
                7'b0100011: begin       //S-type, not used in this lab
                        Instr_field = 2'b10;
                        Ctrl_o = 9'b010001000;
                end
                7'b0000011: begin       //lw, not used in this lab
                        Instr_field = 2'b01;
                        Ctrl_o = 9'b011110000;
                end
                7'b0110011: begin       //R-type
                        Instr_field = 2'b00;
                        Ctrl_o = 9'b000100010;
                end
                default: begin
                        Instr_field = 2'b00;
                        Ctrl_o = 9'b000000000;
                end
        endcase
end

assign ALUSrc   = Ctrl_o[7];
assign RegWrite = Ctrl_o[5];
assign Branch   = Ctrl_o[2];
assign ALUOp    = Ctrl_o[1:0];

endmodule
```

從 opcode 中，我們判斷 instruction 的 type。對於不同的 opcode，output 不同的 ALUSrc，RegWrite，Branch 和 ALUOp。

5.接線

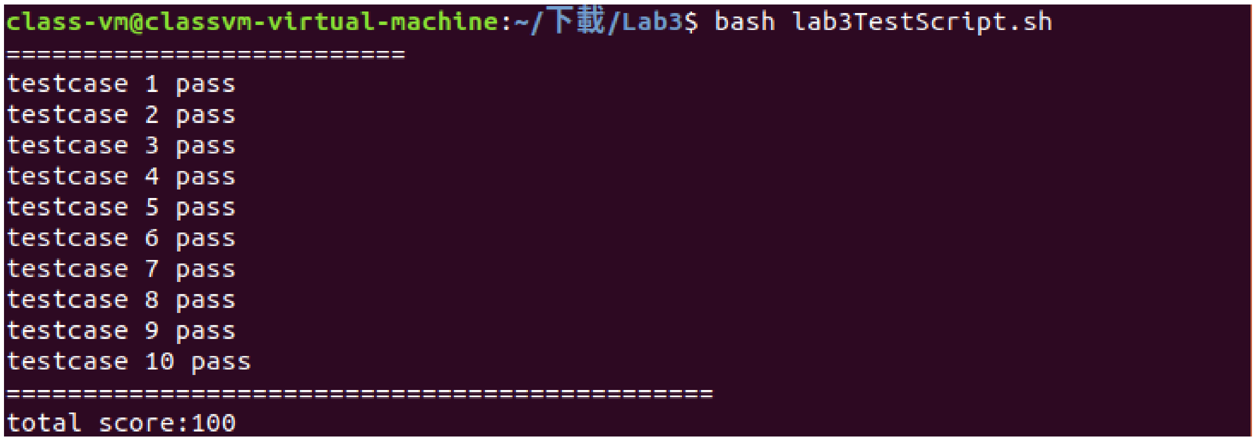
至於 instruction memory，program counter，register 都已經附了，故不贅述。

在 Simple_Single_CPU.v 中，我們將不同的元件照著 architecture diagram 接線。

下圖為 Simple_Single_CPU 的其中一部份 code。

```
Reg_File RF(
        .clk_i(clk_i),
            .rst_i(rst_i) ,
        .RSaddr_i(instr[19:15]) ,
        .RTaddr_i(instr[24:20]) ,
        .RDaddr_i(instr[11:7]) ,
        .RDdata_i(ALUresult)  ,
        .RegWrite_i (RegWrite),
        .RSdata_o(RSdata_o) ,
        .RTdata_o(RTdata_o)
        );

Decoder Decoder(
        .instr_i(instr),
        .ALUSrc(ALUSrc),
        .RegWrite(RegWrite),
        .Branch(Branch),
        .ALUOp(ALUOp)
        );

Adder PC_plus_4_Adder(
        .src1_i(pc_o),
        .src2_i(imm_4),
        .sum_o(pc_i)
            );



ALU_Ctrl ALU_Ctrl(
        .instr({instr[30],instr[14:12]}),
        .ALUOp(ALUOp),
        .ALU_Ctrl_o(ALU_control)
);

alu alu(
        .rst_n(rst_i),
        .src1(RSdata_o),
        .src2(RTdata_o),
        .ALU_control(ALU_control),
        .zero(zero),
        .result(ALUresult),
        .cout(cout),
        .overflow(overflow)
);


endmodule
```

Implementation results:



```
class-vm@classvm-virtual-machine:~/下載/Lab3$ bash lab3TestScript.sh
==========================
testcase 1 pass
testcase 2 pass
testcase 3 pass
testcase 4 pass
testcase 5 pass
testcase 6 pass
testcase 7 pass
testcase 8 pass
testcase 9 pass
testcase 10 pass
==============================================
total score:100
```

將所有寫好的檔案丟到資料夾，並執行 **lab3TestScript.sh**，就可以得到測資全過的結果。

Problems encountered and solutions:

我們在實作時最大的麻煩大概是找 **control** 的 **input output** 和接錯線吧，不過經過一段時間的檢查，最後還是找出來了。

Comment:

從這份作業中，清楚了解了 **cpu** 的運作，真的學到很多東西。期待下個作業可以學到更多東西。