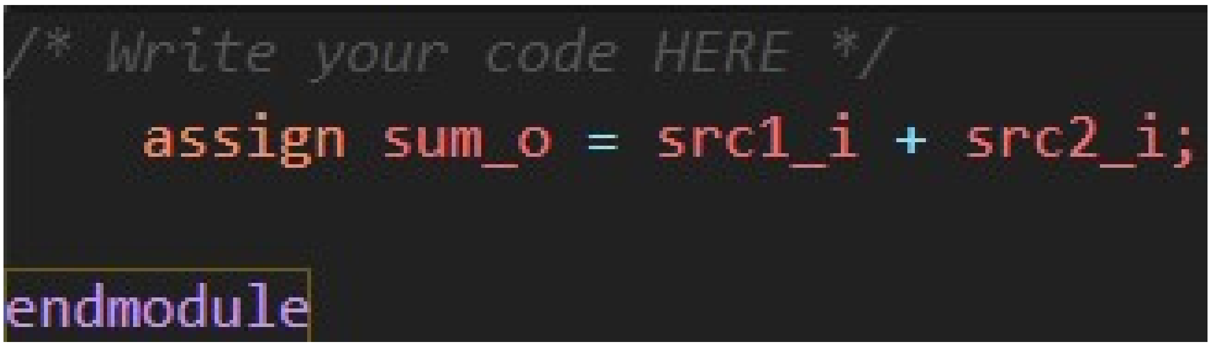
LAB 3 : Single-Cycle CPU

109550164 徐聖哲

1.Code Explain

1.Adder

Because src_i and src2_i are both wire, thus I use assign to implement pc + 4 to sum_o

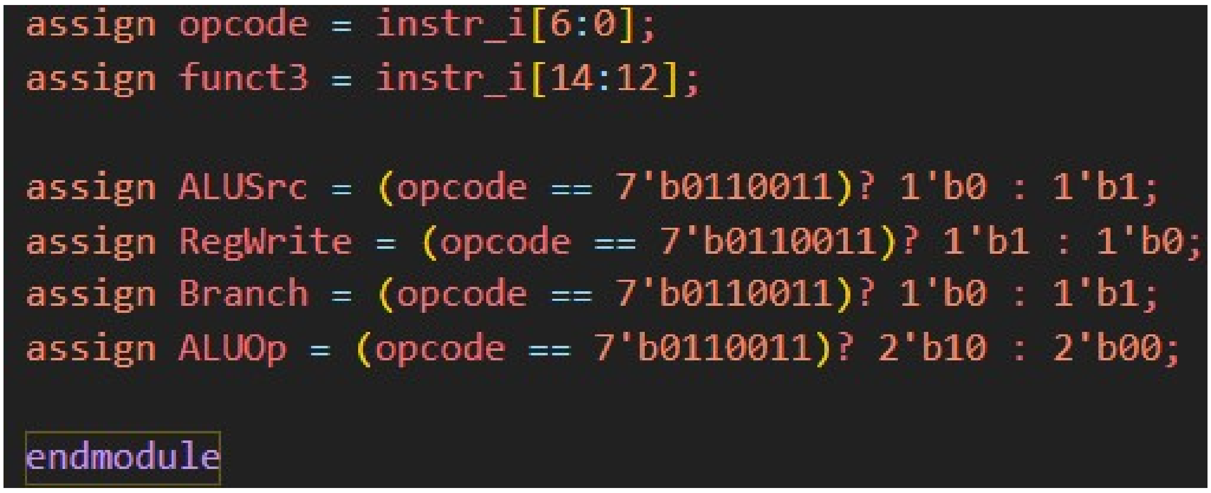


```
/* Write your code HERE */
    assign sum_o = src1_i + src2_i;

endmodule
```

2.Decoder

opcode is instruction [6:0] and functc lies in instruction[14:12]

From chap4-part1, I observe that the opcode of R-format is 0110011, thus I assign ALUSrc, RegWrite, Branch and ALUop based on the table.



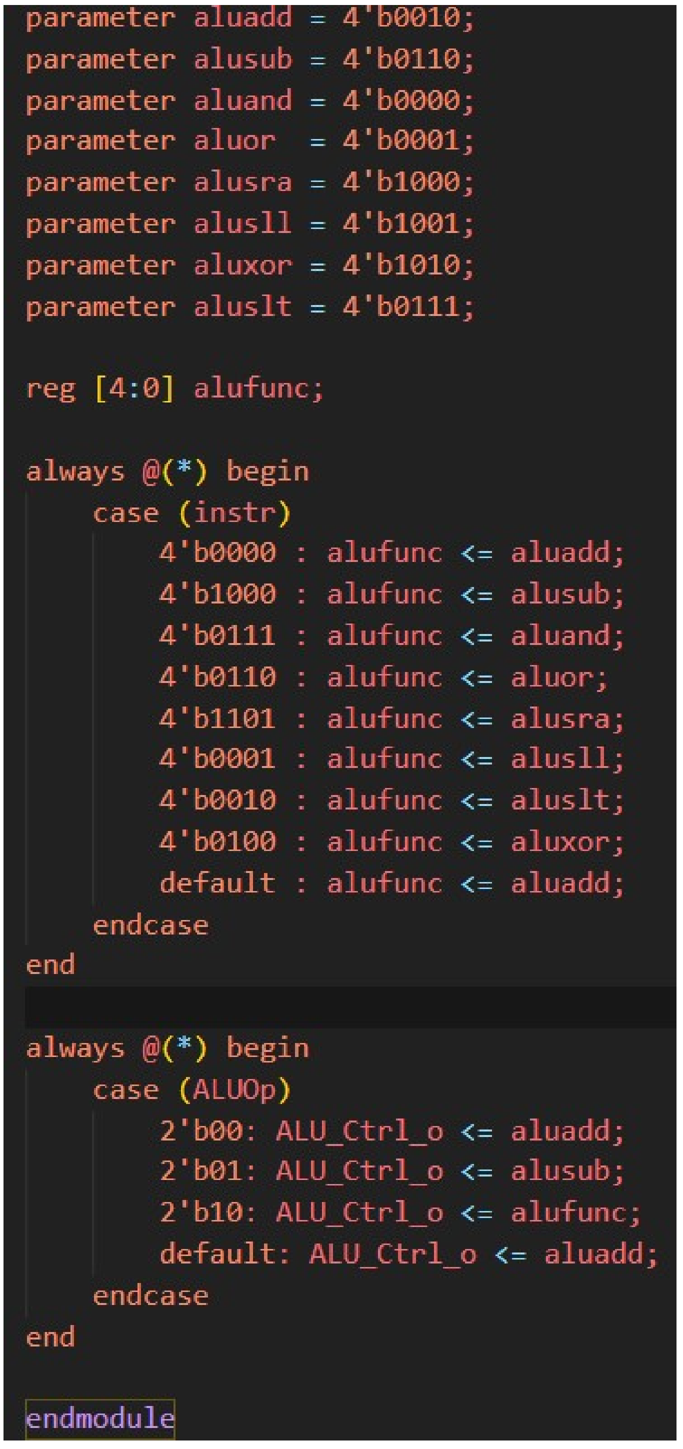
```
assign opcode = instr_i[6:0];
assign funct3 = instr_i[14:12];

assign ALUSrc = (opcode == 7'b0110011)? 1'b0 : 1'b1;
assign RegWrite = (opcode == 7'b0110011)? 1'b1 : 1'b0;
assign Branch = (opcode == 7'b0110011)? 1'b0 : 1'b1;
assign ALUOp = (opcode == 7'b0110011)? 2'b10 : 2'b00;

endmodule
```

3.ALU_Ctrl

By the instruction[30] and funct3 form instr which determione which reg-reg instruction to implement

As for alucontrol signal, add, sub, and, or and slt is defined in lab2, so I only define sra→1000, sll→1001 and xor→1010 this time

I use case in this .v to make code clean and choose right alufunc to implement



```
parameter aluadd = 4'b0010;
parameter alusub = 4'b0110;
parameter aluand = 4'b0000;
parameter aluor  = 4'b0001;
parameter alusra = 4'b1000;
parameter alusll = 4'b1001;
parameter aluxor = 4'b1010;
parameter aluslt = 4'b0111;

reg [4:0] alufunc;

always @(*) begin
    case (instr)
        4'b0000 : alufunc <= aluadd;
        4'b1000 : alufunc <= alusub;
        4'b0111 : alufunc <= aluand;
        4'b0110 : alufunc <= aluor;
        4'b1101 : alufunc <= alusra;
        4'b0001 : alufunc <= alusll;
        4'b0010 : alufunc <= aluslt;
        4'b0100 : alufunc <= aluxor;
        default : alufunc <= aluadd;
    endcase
end

always @(*) begin
    case (ALUOp)
        2'b00: ALU_Ctrl_o <= aluadd;
        2'b01: ALU_Ctrl_o <= alusub;
        2'b10: ALU_Ctrl_o <= alufunc;
        default: ALU_Ctrl_o <= aluadd;
    endcase
end

endmodule
```

4.ALU

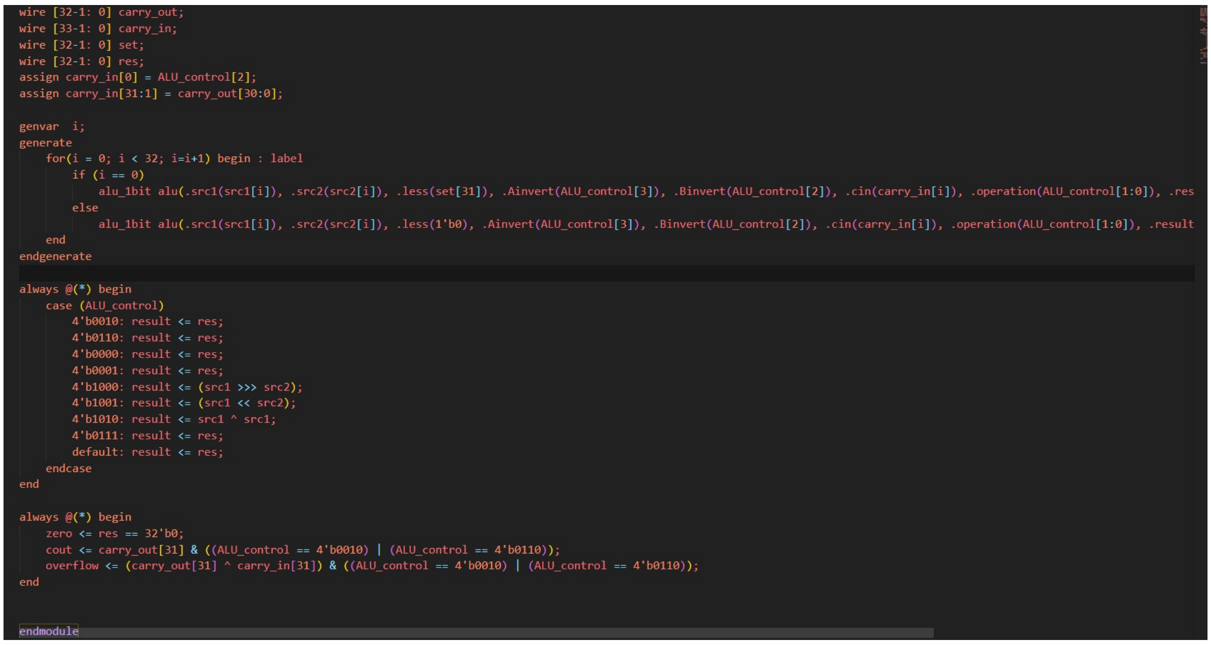
I use the alu.v in lab2, so only the first always block is new

In the first always block, I use case block to determine result, for those writed in lab2, I stay the sam with “result <= res;”

sra→1000, I use >>> to implement shift right arithemtic

sll→1001, I use << to implement shift left logically

x0r→1010, I use the simbol ^ to implement xor



```
wire [32-1: 0] carry_out;
wire [33-1: 0] carry_in;
wire [32-1: 0] set;
wire [32-1: 0] res;
assign carry_in[0] = ALU_control[2];
assign carry_in[31:1] = carry_out[30:0];

genvar  i;
generate
    for(i = 0; i < 32; i=i+1) begin : label
        if (i == 0)
            alu_1bit alu(.src1(src1[i]), .src2(src2[i]), .less(set[31]), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(carry_in[i]), .operation(ALU_control[1:0]), .res
        else
            alu_1bit alu(.src1(src1[i]), .src2(src2[i]), .less(1'b0), .Ainvert(ALU_control[3]), .Binvert(ALU_control[2]), .cin(carry_in[i]), .operation(ALU_control[1:0]), .result
    end
endgenerate

always @(*) begin
    case (ALU_control)
        4'b0010: result <= res;
        4'b0110: result <= res;
        4'b0000: result <= res;
        4'b0001: result <= res;
        4'b1000: result <= (src1 >>> src2);
        4'b1001: result <= (src1 << src2);
        4'b1010: result <= src1 ^ src1;
        4'b0111: result <= res;
        default: result <= res;
    endcase
end

always @(*) begin
    zero <= res == 32'b0;
    cout <= carry_out[31] & ((ALU_control == 4'b0010) | (ALU_control == 4'b0110));
    overflow <= (carry_out[31] ^ carry_in[31]) & ((ALU_control == 4'b0010) | (ALU_control == 4'b0110));
end


endmodule
```

5.Simple Single CPU

See lab3 slide to fill in all the parameter which create line for cpu

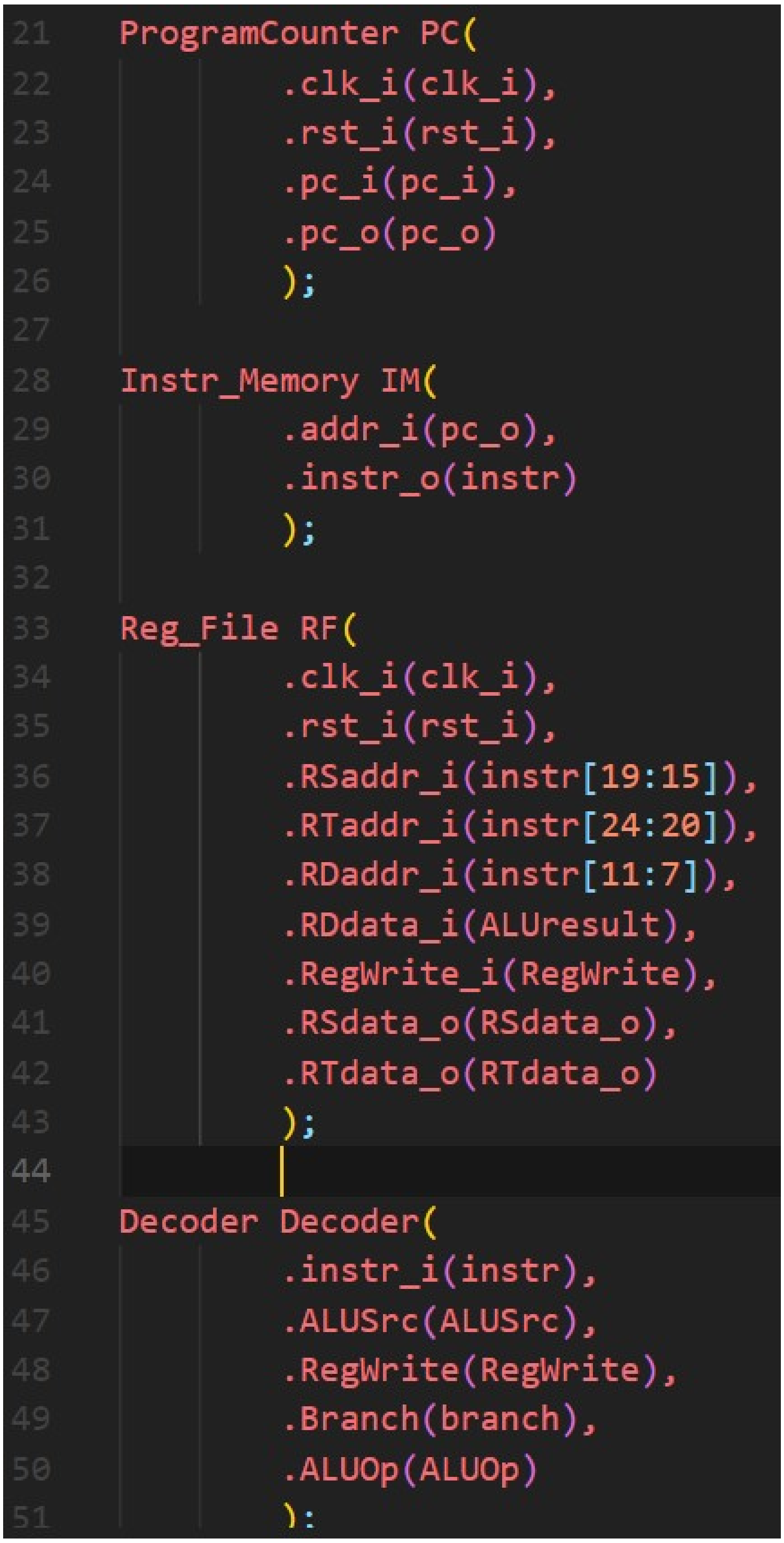
For programCounter, pc_o is the current instruction adder

For Reg_File, Write data is ALUresult , src1 is RSdata_o, drc2 is RTdata_o

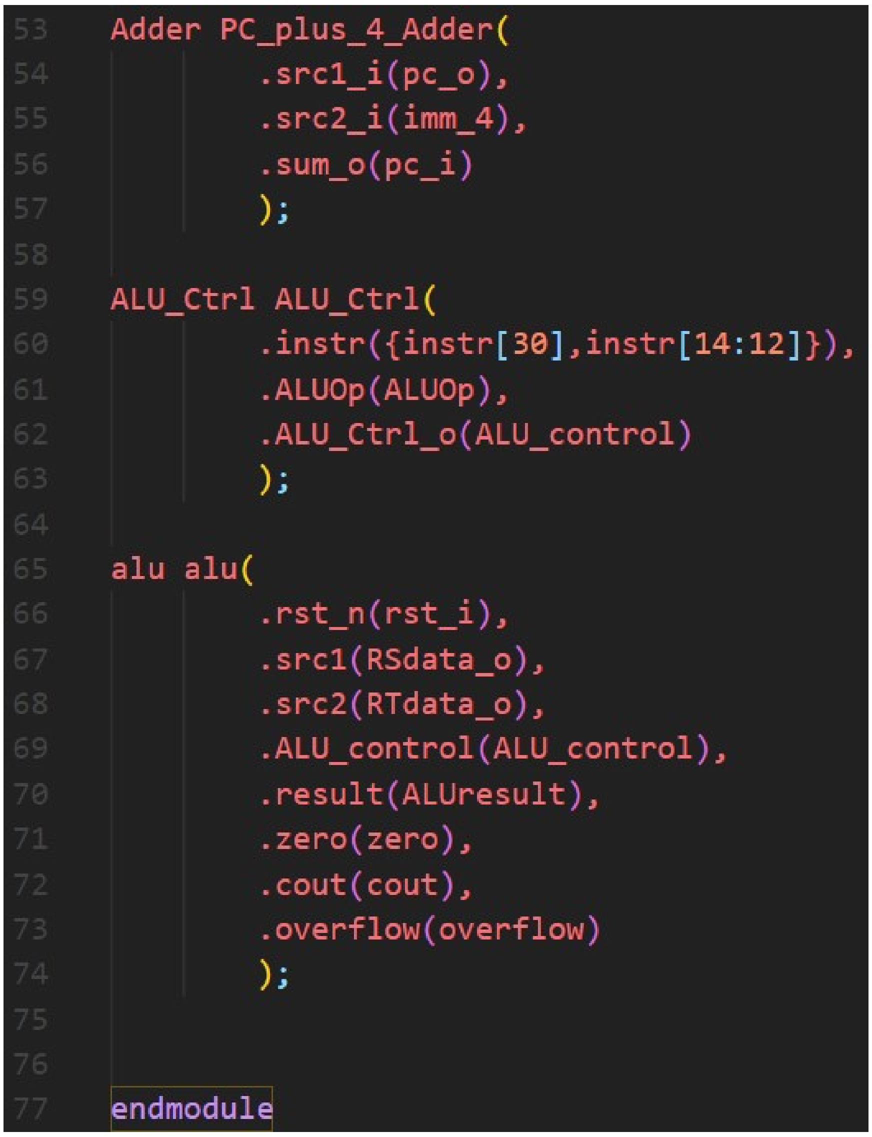
For Adder, src2 is imm_4 which means pc + 4

For ALU_Ctrl, instr is i[30] + funct3, funct3 is instruction[14:12]

For alu, src1 is RSdata_o, drc2 is RTdata_o and result is ALUresult

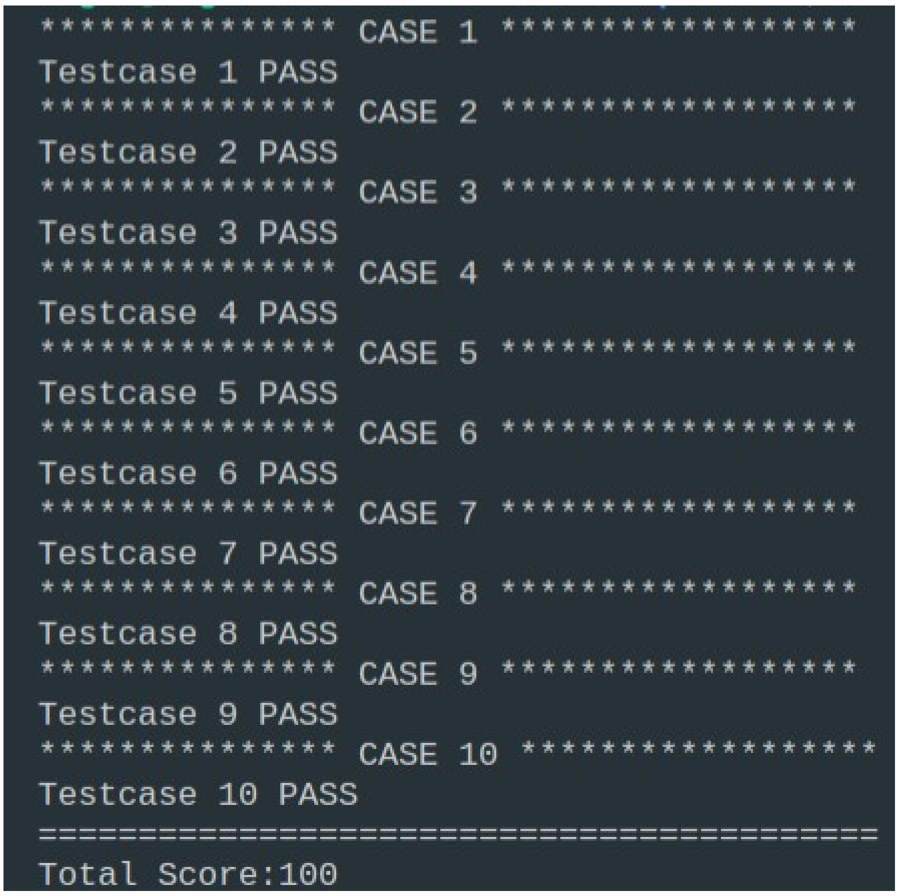


```
ProgramCounter PC(
            .clk_i(clk_i),
            .rst_i(rst_i),
            .pc_i(pc_i),
            .pc_o(pc_o)
            );

Instr_Memory IM(
            .addr_i(pc_o),
            .instr_o(instr)
            );

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),
        .result(ALUresult),
        .zero(zero),
        .cout(cout),
        .overflow(overflow)
        );


endmodule
```

2.Implementation results



```
*************** CASE 1 ******************
Testcase 1 PASS
*************** CASE 2 ******************
Testcase 2 PASS
*************** CASE 3 ******************
Testcase 3 PASS
*************** CASE 4 ******************
Testcase 4 PASS
*************** CASE 5 ******************
Testcase 5 PASS
*************** CASE 6 ******************
Testcase 6 PASS
*************** CASE 7 ******************
Testcase 7 PASS
*************** CASE 8 ******************
Testcase 8 PASS
*************** CASE 9 ******************
Testcase 9 PASS
*************** CASE 10 ******************
Testcase 10 PASS
==========================================
Total Score:100
```

3.Problems encountered and solutions

1. I first encounter problem in ALUCtrl. I was confused with the instr and aluop, but when I dicovered that aluop was always 10 in this lab, I could kepp coding.
2. Secondly, I quickly faced another problem, which is how to determine my own alucontrol parameter, because I have to use lab2code in alu.v, so I have to avoid the original parameter. Thus I spent some time to record old alu_control code and decideed new parameter for sll sra and xor.
3. In simple single cpu, I have to look at the slide in chapter 4 part 1 to finish fill in the ().
   First I didn't konw the meaning of RS and RT, until I browsed hackmad, thanks for classmate and teaching assisant. Then, Regfile is most complex, because I have to clearly understand which part of instruction is for input.