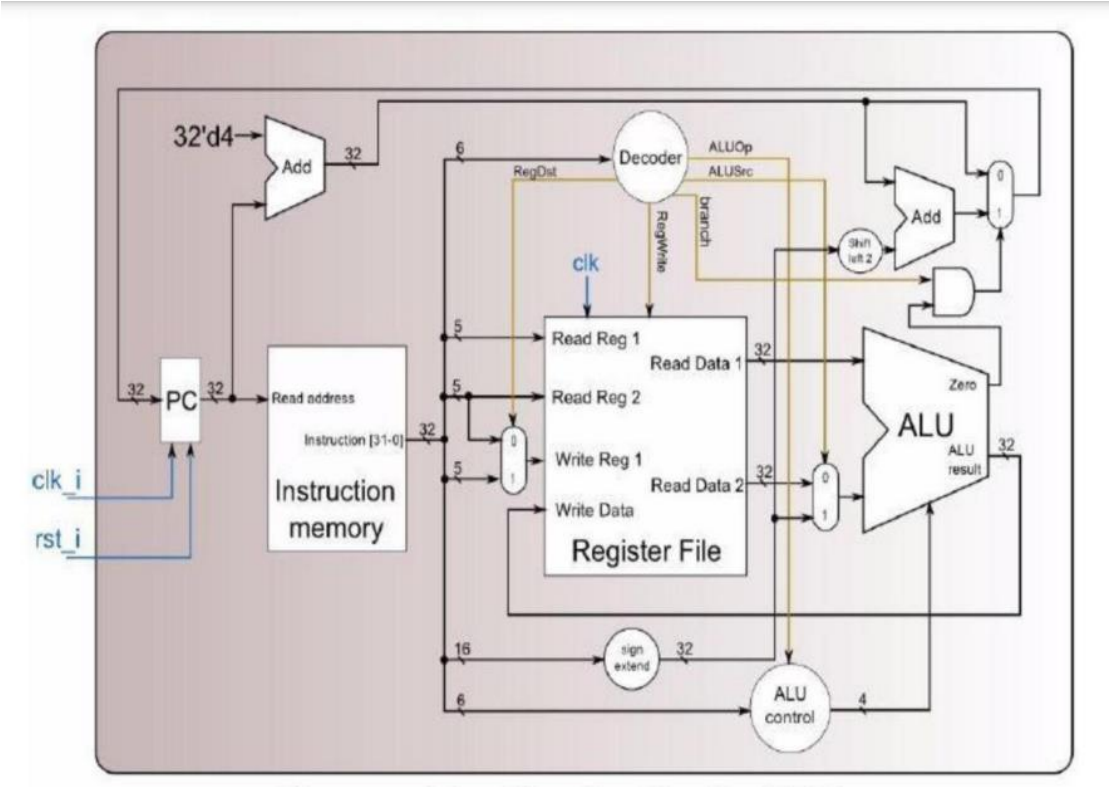
Computer Organization Lab2

Name:林秉

ID:109550112

Architecture diagrams:



Hardware module analysis:

Adder:

```
//Main function
assign sum_o = src1_i + src2_i;
endmodule
```

We simply add result and return to sum.

ALU_ctrl:

```
wire [5:0] add, sub, And, Or, slt, slti;
assign add = 6'b100000;
assign sub = 6'b100010;
assign And = 6'b100100;
assign Or = 6'b100101;
assign slt = 6'b100110;
assign slti = 6'b000000;
```

We followed the function field in pdf to give 6 instructions.

```
always@(*)
    begin
      if(ALUOp_i[2] == 1)
        begin
            ALUCtrl_o[0] = (funct_i == slt) || (funct_i == Or);
            ALUCtrl_o[1] = (funct_i == add) || (funct_i == sub) || (funct_i == slt);
            ALUCtrl_o[2] = (funct_i == slt) || (funct_i == sub);
        end
        else if(ALUOp_i[1:0] == 2'b11)
        begin
            ALUCtrl_o[2:0] = 3'b110;
        end

        else if(ALUOp_i[1] == 1)
        begin
            ALUCtrl_o[2:0] = 3'b010;
        end

        else if(ALUOp_i[0] == 1)
        begin
            ALUCtrl_o[2:0] = 3'b111;
```

Then we checked the table of ALU_OP to assign the ALU_ctrl output by the following graph. When ALU_op[2] = 0, means it's R_format(Which you'll see in decoder), thus we assign value for R_format instructions here. For rest I_format instructions, we assign values one-by-one. The ALUCtrl_o[3] would remain 0 since there's no need in this lab.

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

ALU:

```
assign zero_o = result_o == 0;

always@(*)
    begin
    case(ctrl_i)
        0: result_o = src1_i & src2_i;
        1: result_o = src1_i | src2_i;
        2: result_o = src1_i + src2_i;
        6: result_o = src1_i - src2_i;
        7: if(src1_i<src2_i)
             begin
                result_o = 1;
             end
           else
             begin
                result_o = 0;
             end
       12: result_o = ~(src1_i | src2_i);
    endcase
    end
endmodule
```

Zero = 1 iff result is 0. Then again, we follow the ALUctrl instructions with given values by the above graph.

Decoder:

```
40      //Main function
41      assign r = (instr_op_i == 0);
42      assign addi = (instr_op_i == 8);
43      assign beq = (instr_op_i == 4);
44      assign slti = (instr_op_i == 10);
45
46      always@(*)
47          begin
48          RegWrite_o = addi | r | slti;
49          ALUSrc_o = addi | slti;
50          RegDst_o = r;
51          Branch_o = beq;
52          ALU_op_o[2] = r;
53          ALU_op_o[1] = addi | beq;
54          ALU_op_o[0] = beq | slti;
55          end
56      endmodule
```

Op Field
0
8
0
0
0
0
10 (0xa)
4

We assign four wires for specified instructions. When r = 1, means the instruction is R-format, else it's I-format. And with checking the instruction table, we can assign correct value for the decoder, especially for ALU_op, since it explains the specific condition segment for ALU_ctrl.

MUX:

```
always@(*)
begin
    if(select_i == 1)
        data_o = data1_i;
    else
        data_o = data0_i;
end
endmodule
```

When select_i is 1 then output data 1, likewise for select_i = 0.

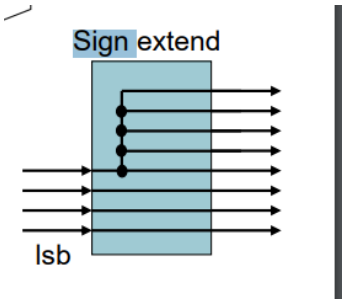
Shift_Left_Two_32:

```
assign data_o = data_i << 2;
endmodule
```

We simply shift 2 bits.

Sign_Extend:

```
always@(*)
begin
   for(i = 0;i<32;i=i+1)
   begin
      if(i>=16)
        data_o[i] = data_i[15];
      else
        data_o[i] = data_i[i];
   end
end
endmodule
```



We extend the output bits by the above diagram.

Simple_Single_CPU:

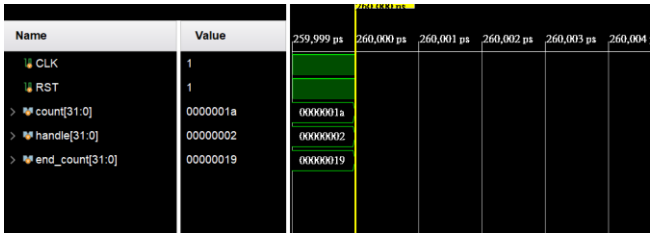
```
wire [31:0] pc_in, pc_out, add1_out, ist_out, sign_out, shift_out, add2_out, regs_out, regt_out, aluresult, muxalu_out;
wire [4:0] muxreg1_out;
wire [3:0] aluctrl;
wire [2:0] aluop;
wire regdst,branch,regwrite,alusrc,zero, adders_select;
assign adders_select = branch & zero;
```

We assign enough different wires with different bits count to allocate to different modules. And the result would be assigned by the given architecture diagram above.

Finished part:

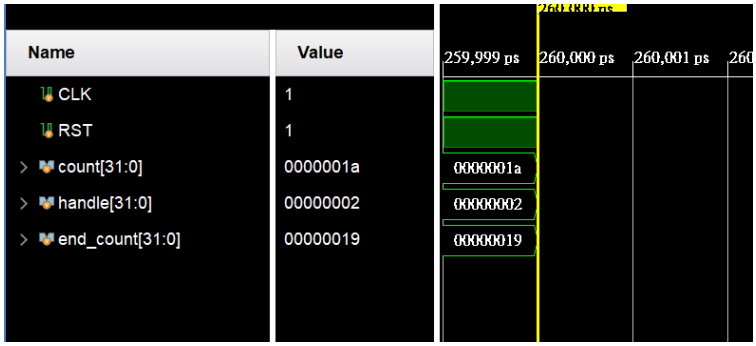
Case1:

```
# run 1000ns
          2
r0=          0
r1=         10
r2=          4
r3=          0
r4=          0
r5=          6
r6=          0
r7=          0
r8=          0
r9=          0
r10=         0
r11=         0
r12=         0
```



Case2:

```
# run 1000ns
          2
r0=          0
r1=          1
r2=          0
r3=          0
r4=          0
r5=          0
r6=          0
r7=         14
r8=          0
r9=         15
r10=         0
r11=         0
r12=         0
```



Problems you met and solutions:

Since this lab includes lots of variables calculations, when I was debugging, I find myself often using variables with typo, and the program can still run with undeclared variables. This has confused me a lot. And the ALU_operations are a little hard to understand at first. But with lots of careful double checks, I eventually solved the problems.

Summary:

This lab is more messy comparing to lab1, there are a lot of details needed to pay attention, I think I need to be careful on every tasks and focus myself for the next challenge.