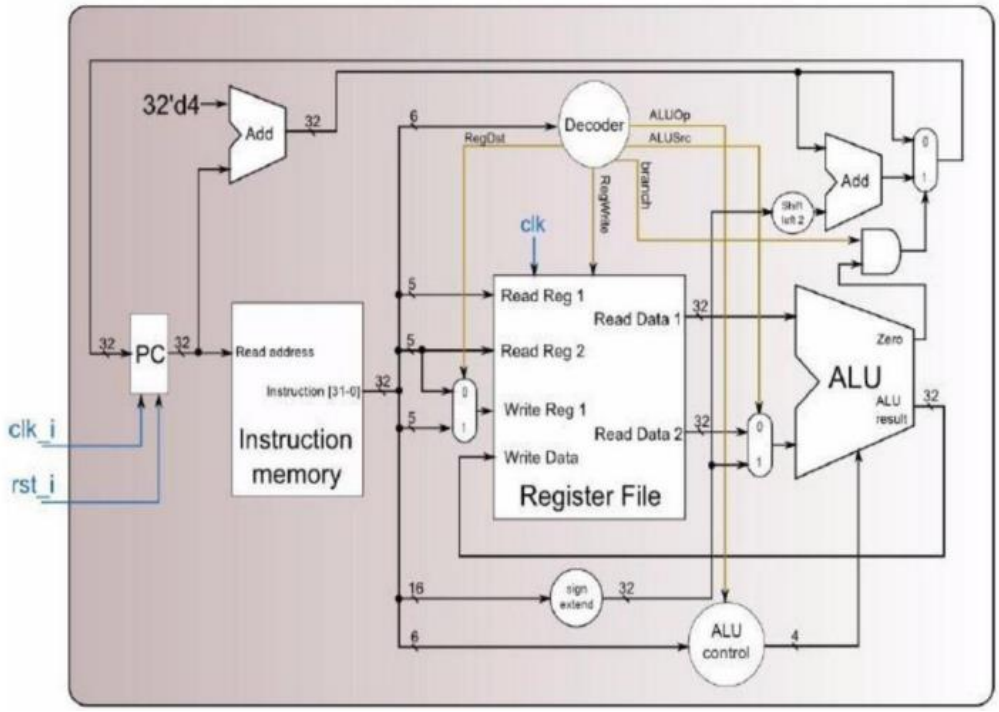
Computer Organization

Architecture diagrams:



Hardware module analysis:

Adder.v 把兩個 source 相加。

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

ALU_Ctrl.v 是利用 opcode 和 function code 得出控制 ALU 的 control signal，依照以下表格得出 ALUCtrl_o。

opcode	ALUOp	operation	funct	ALU funct	ALU ctrl
addi	00	addi	xxxxxx	add	0010
slti	01	slti	xxxxxx	slt	0111
beq	10	beq	xxxxxx	sub	0110
R-format	11	add	100000	add	0010
		sub	100010	sub	0110
		and	100100	and	0000
		or	100101	or	0001
		slt	101010	slt	0111

```
31  //Select exact operation
32  always @(funct_i, ALUOp_i)
33  begin
34      case (ALUOp_i)
35          0: ALUCtrl_o <= 4'b0010;  //addi
36          1: ALUCtrl_o <= 4'b0111;  //slti
37          2: ALUCtrl_o <= 4'b0110;  //beq
38          3:  begin
39                  case (funct_i)
40                      32: ALUCtrl_o <= 4'b0010;  //add
41                      34: ALUCtrl_o <= 4'b0110;  //sub
42                      36: ALUCtrl_o <= 4'b0000;  //and
43                      37: ALUCtrl_o <= 4'b0001;  //or
44                      42: ALUCtrl_o <= 4'b0111;  //slt
45                  endcase
46              end
47          default: ALUCtrl_o <= 4'b0000;
48      endcase
49  end
```

ALU.v 是使用課本的 ALU，有 and、or、add、subtract、set on less than、nor 這些功能。

```
35  //Main function
36  assign zero_o = (result_o == 0);
37  always @(ctrl_i, src1_i, src2_i)
38  begin
39      case (ctrl_i)
40          0: result_o <= src1_i & src2_i;
41          1: result_o <= src1_i | src2_i;
42          2: result_o <= src1_i + src2_i;
43          6: result_o <= src1_i - src2_i;
44          7: result_o <= src1_i < src2_i ? 1 : 0;
45          12: result_o <= ~(src1_i | src2_i);
46          default: result_o <= 0;
47      endcase
48  end
```

Decoder.v 是將輸入的 op field 的值對應到相對應的功能，並且輸出各種 control signal，下表是對應的結果。

	R-format	addi	slti	Beq
Op5	0	0	0	0
Op4	0	0	0	0
Op3	0	1	1	0
Op2	0	0	0	1
Op1	0	0	1	0
Op0	0	0	0	0
RegDst	1	0	0	0
ALUSrc	0	1	1	0
RegWrite	1	1	1	0
Branch	0	0	0	1
ALUOp1	1	0	0	1
ALUOp0	1	0	1	0

```
41  //Main function
42  always @(instr_op_i)
43  begin
44      case (instr_op_i)
45          0:  begin  //R-format
46                  RegDst_o <= 1;
47                  ALUSrc_o <= 0;
48                  RegWrite_o <= 1;
49                  Branch_o <= 0;
50                  ALU_op_o <= 2'b11;
51              end
52          4:  begin  //beq
53                  RegDst_o <= 0;
54                  ALUSrc_o <= 0;
55                  RegWrite_o <= 0;
56                  Branch_o <= 1;
57                  ALU_op_o <= 2'b10;
58              end
```

```
59          8:  begin  //addi
60                  RegDst_o <= 0;
61                  ALUSrc_o <= 1;
62                  RegWrite_o <= 1;
63                  Branch_o <= 0;
64                  ALU_op_o <= 2'b00;
65              end
66          10: begin  //slti
67                  RegDst_o <= 0;
68                  ALUSrc_o <= 1;
69                  RegWrite_o <= 1;
70                  Branch_o <= 0;
71                  ALU_op_o <= 2'b01;
72              end
73          default:  begin
74                      RegDst_o <= 1;
75                      ALUSrc_o <= 0;
76                      RegWrite_o <= 1;
77                      Branch_o <= 0;
78                      ALU_op_o <= 2'b11;
```

MUX_2to1.v 是基礎的 multiplexer，當 select 等於 0，輸出 source0，當 select 等於 1，輸出 source1。

```
31  //Main function
32  always @(data0_i, data1_i, select_i)
33  begin
34      case (select_i)
35          0: data_o <= data0_i;
36          1: data_o <= data1_i;
37          default: data_o <= data0_i;
38      endcase
39  end
```

Shift_Left_Two.v 是將 input data 往左 shift 2 bits。

```
18  //shift left 2
19  assign data_o = data_i << 2;
```

Sign_Extend.v 是將 input data 的 MSB 往左延伸 16 bits。

```
22  //Sign extended
23  assign data_o = {{16{data_i[15]}}, data_i[15:0]};
```

Simple_Single_CPU.v 是要把所有的 module 都接線，因此我設定了許多 wire，並且設定好 module 的 input 和 output。

```
45  ProgramCounter PC(      //PC
46          .clk_i(clk_i),
47          .rst_i (rst_i),
48          .pc_in_i(pc_in_i),
49          .pc_out_o(pc_out_o)
50          );
```

```
52  Adder Adder1(          //PC+4
53          .src1_i(32'd4),
54          .src2_i(pc_out_o),
55          .sum_o(pc_plus_four)
56          );
```

```
58  Instr_Memory IM(       //Instruction_memory
59          .pc_addr_i(pc_out_o),
60          .instr_o(instruction)
61          );
```

```
63  MUX_2to1 #(.size(5)) Mux_Write_Reg(
64          .data0_i(instruction[20:16]),
65          .data1_i(instruction[15:11]),
66          .select_i(regdst),
67          .data_o(write_register)
68          );
```

```
70  Reg_File RF(          //Register_File
71          .clk_i(clk_i),
72          .rst_i(rst_i),
73          .RSaddr_i(instruction[25:21]),
74          .RTaddr_i(instruction[20:16]),
75          .RDaddr_i(write_register),
76          .RDdata_i(alu_result),
77          .RegWrite_i(regwrite),
78          .RSdata_o(read_data1),
79          .RTdata_o(read_data2)
80          );

82  Decoder Decoder(      //Decoder
83          .instr_op_i(instruction[31:26]),
84          .RegWrite_o(regwrite),
85          .ALU_op_o(alu_op),
86          .ALUSrc_o(alusrc),
87          .RegDst_o(regdst),
88          .Branch_o(branch)
89          );

91  ALU_Ctrl AC(          //ALU_control
92          .funct_i(instruction[5:0]),
93          .ALUOp_i(alu_op),
94          .ALUCtrl_o(alu_control)
95          );

97  Sign_Extend SE(       //Sign_extend
98          .data_i(instruction[15:0]),
99          .data_o(sign_ext_o)
100         );

102 MUX_2to1 #(.size(32)) Mux_ALUSrc(
103         .data0_i(read_data2),
104         .data1_i(sign_ext_o),
105         .select_i(alusrc),
106         .data_o(alu_input2)
107         );

109 ALU ALU(             //ALU
110         .src1_i(read_data1),
111         .src2_i(alu_input2),
112         .ctrl_i(alu_control),
113         .result_o(alu_result),
114         .zero_o(zero_o)
115         );

117 Adder Adder2(        //branch_adder
118         .src1_i(pc_plus_four),
119         .src2_i(shift_left_o),
120         .sum_o(branch_adder_o)
121         );

123 Shift_Left_Two_32 Shifter( //shift_left_2
124         .data_i(sign_ext_o),
125         .data_o(shift_left_o)
126         );

128 MUX_2to1 #(.size(32)) Mux_PC_Source(
129         .data0_i(pc_plus_four),
130         .data1_i(branch_adder_o),
131         .select_i(pc_source_control),
132         .data_o(pc_in_i)
133         );

135 assign pc_source_control = branch & zero_o;
```

```
23  wire  [32-1:0] pc_in_i;
24  wire  [32-1:0] pc_out_o;
25  wire  [32-1:0] pc_plus_four;
26  wire  [32-1:0] instruction;
27  wire           regdst;
28  wire           regwrite;
29  wire           alusrc;
30  wire  [2-1: 0] alu_op;
31  wire           branch;
32  wire  [5-1: 0] write_register;
33  wire  [32-1:0] alu_result;
34  wire  [32-1:0] read_data1;
35  wire  [32-1:0] read_data2;
36  wire  [4-1: 0] alu_control;
37  wire  [32-1:0] alu_input2;
38  wire  [32-1:0] sign_ext_o;
39  wire           zero_o;
40  wire  [32-1:0] shift_left_o;
41  wire  [32-1:0] branch_adder_o;
42  wire           pc_source_control;
```

Finished part:

Test1:

```
addi r1, r0, 10  =>  r1 = r0 + 10                  =>  r1 = 10
addi r2, r0, 4   =>  r2 = r0 + 4                   =>  r2 = 4
slt r3, r1, r2   =>  r3 = (r1 < r2) ? 1 : 0        =>  r3 = 0
beq r3, r0, 1    =>  if(r3 == r0) pc += 4 + 1 * 4  =>  pc = pc + 8
add r4, r1, r2   =>  r4 = r1 + r2                  =>  r4 = 14
sub r5, r1, r2   =>  r5 = r1 - r2                  =>  r5 = 6
```

```
r0=          0
r1=         10
r2=          4
r3=          0
r4=          0
r5=          6
r6=          0
```

Test2:

```
addi r6, r0, 2   =>  r6 = r0 + 2
addi r7, r0, 14  =>  r7 = r0 + 14
and  r8, r6, r7  =>  r8 = r6 & r7
or   r9, r6, r7  =>  r9 = r6 | r7
addi r6, r6, -1  =>  r6 = r6 + (-1)
slti r1, r6, 1   =>  r1 = (r6 < 1) ? 1 : 0
beq  r1, r0, -5  =>  if(r1 == r0) pc += 4 + (-5) * 4
```

每個 instruction 改變的值

```
=>  r6 = 2
=>  r7 = 14
=>  r8 = 2
=>  r9 = 14
=>  r6 = 1
=>  r1 = 0
=>  pc = pc - 16
=>  r8 = 0
=>  r9 = 15
=>  r6 = 0
=>  r1 = 1
```

```
r0=    0
r1=    1
r2=    0
r3=    0
r4=    0
r5=    0
r6=    0
r7=   14
r8=    0
r9=   15
```

Problems you met and solutions:

在實作的過程，並沒有遇到什麼問題，唯一的問題是這次的 Lab 很難 DEBUG，因為可能影響的因素有很多，而且又不像其他程式語言可以很容易地將變數的值 print 出來，最後的解決辦法是先看有沒有打錯字，接著再去檢查各個 module 有沒有錯，並且一邊修改測資，觀察答案是否是預期的答案。

Summary:

這次的 Lab 是實作一個 single cycle cpu，雖然看起來很複雜，但實際上也只是將各個簡單的 module 接線，但線接到後面也是有點亂，而且也很容易因為一個小錯，就讓答案變得很奇怪，而且也不好 debug，我覺得這次的 Lab 很考驗細心的程度。