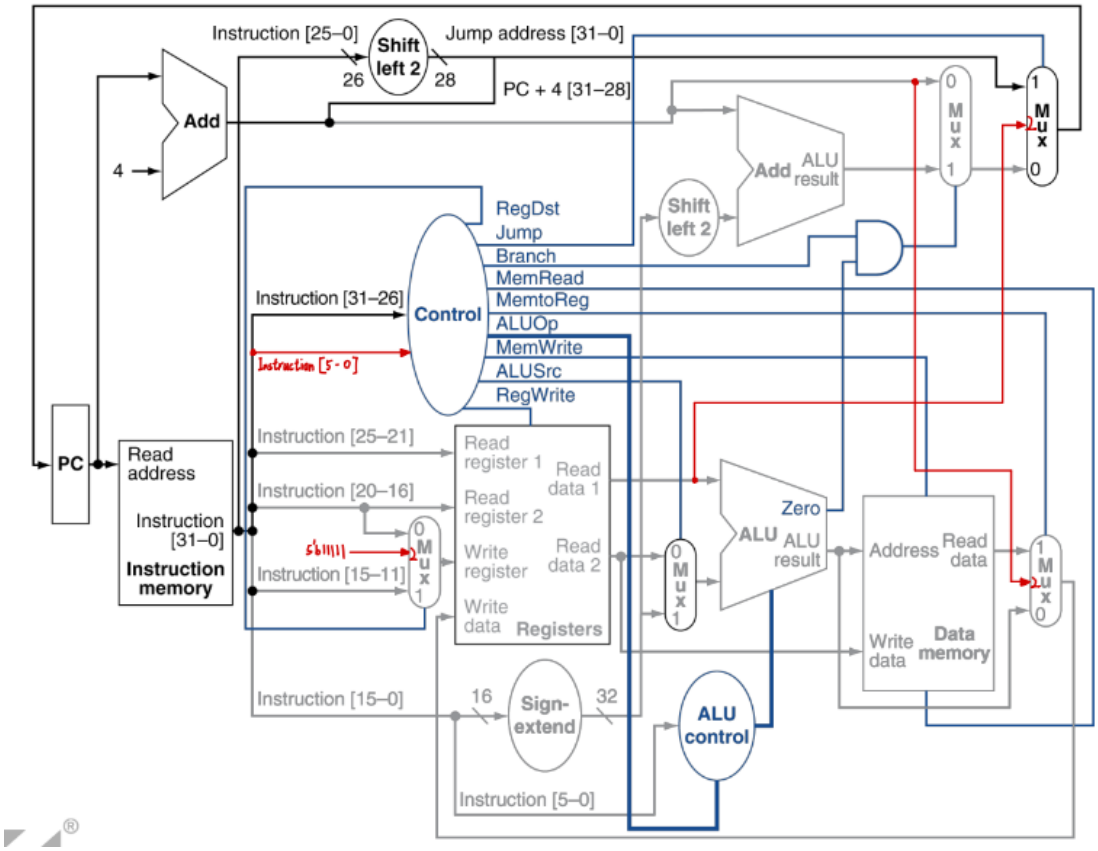
Computer Organization Lab3

Name:張博凱

ID:110550063

Architecture diagrams:

底圖是課程上使用的包含 jump 功能的圖，紅色的線是為了實作 jal 和 jr 而畫的線。



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
lw		lw	xxxxxx	add	0010
sw		sw	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

```
30  //Select exact operation
31  always @(funct_i, ALUOp_i)
32  begin
33      case (ALUOp_i)
34          0: ALUCtrl_o <= 4'b0010;  //addi lw sw
35          1: ALUCtrl_o <= 4'b0111;  //slti
36          2: ALUCtrl_o <= 4'b0110;  //beq
37          3:  begin
38                  case (funct_i)
39                      32: ALUCtrl_o <= 4'b0010;  //add
40                      34: ALUCtrl_o <= 4'b0110;  //sub
41                      36: ALUCtrl_o <= 4'b0000;  //and
42                      37: ALUCtrl_o <= 4'b0001;  //or
43                      42: ALUCtrl_o <= 4'b0111;  //slt
44                  endcase
45              end
46          default: ALUCtrl_o <= 4'b0000;
47      endcase
48  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，並且在等於 0(R-format)時，判斷 function field 的值是否為 8(jr)，最後輸出各種 control signal，下表是對應的結果。

	R-format	addi	lw	sw	slti	Beq	jump	jal	jr
Op5	0	0	1	1	0	0	0	0	0
Op4	0	0	0	0	0	0	0	0	0
Op3	0	1	0	1	1	0	0	0	0
Op2	0	0	0	0	0	1	0	0	0
Op1	0	0	1	1	1	0	1	1	0
Op0	0	0	1	1	0	0	0	1	0
RegDst	1	0	0	0	0	0	0	2	0
ALUSrc	0	1	1	1	1	0	0	0	0
MemtoReg	0	0	1	0	0	0	0	2	0
RegWrite	1	1	1	0	1	0	0	1	0
MemRead	0	0	1	0	0	0	0	0	0
MemWrite	0	0	0	1	0	0	0	0	0
Branch	0	0	0	0	0	1	0	0	0
ALUOp1	1	0	0	0	0	1	0	0	0
ALUOp0	1	0	0	0	1	0	0	0	0
Jump	0	0	0	0	0	0	1	1	2

```
54  //Main function
55  always @(instr_op_i, instr_funct_i)
56  begin
57      case (instr_op_i)
58          0:  begin  //R-format
59                  if(instr_funct_i == 6'b001000) begin //jr
60                      RegDst_o <= 2'b00;
61                      ALUSrc_o <= 0;
62                      MemtoReg_o <= 2'b00;
63                      RegWrite_o <= 0;
64                      MemRead_o <= 0;
65                      MemWrite_o <= 0;
66                      Branch_o <= 0;
67                      ALU_op_o <= 2'b00;
68                      Jump_o <= 2'b10;
69                  end
70                  else begin
70              else begin
71                  RegDst_o <= 2'b01;
72                  ALUSrc_o <= 0;
73                  MemtoReg_o <= 2'b00;
74                  RegWrite_o <= 1;
75                  MemRead_o <= 0;
76                  MemWrite_o <= 0;
77                  Branch_o <= 0;
78                  ALU_op_o <= 2'b11;
79                  Jump_o <= 2'b00;
80              end
81          end
93      3:  begin  //jal
94              RegDst_o <= 2'b10;
95              ALUSrc_o <= 0;
96              MemtoReg_o <= 2'b10;
97              RegWrite_o <= 1;
98              MemRead_o <= 0;
99              MemWrite_o <= 0;
100             Branch_o <= 0;
101             ALU_op_o <= 2'b00;
102             Jump_o <= 2'b01;
103         end
82      2:  begin  //jump
83              RegDst_o <= 2'b00;
84              ALUSrc_o <= 0;
85              MemtoReg_o <= 2'b00;
86              RegWrite_o <= 0;
87              MemRead_o <= 0;
88              MemWrite_o <= 0;
89              Branch_o <= 0;
90              ALU_op_o <= 2'b00;
91              Jump_o <= 2'b01;
92          end
104     4:  begin  //beq
105             RegDst_o <= 2'b00;
106             ALUSrc_o <= 0;
107             MemtoReg_o <= 2'b00;
108             RegWrite_o <= 0;
109             MemRead_o <= 0;
110             MemWrite_o <= 0;
111             Branch_o <= 1;
112             ALU_op_o <= 2'b10;
113             Jump_o <= 2'b00;
114         end
```

```
        8:  begin  //addi
                RegDst_o <= 2'b00;
                ALUSrc_o <= 1;
                MemtoReg_o <= 2'b00;
                RegWrite_o <= 1;
                MemRead_o <= 0;
                MemWrite_o <= 0;
                Branch_o <= 0;
                ALU_op_o <= 2'b00;
                Jump_o <= 2'b00;
            end
        35: begin  //lw
                RegDst_o <= 2'b00;
                ALUSrc_o <= 1;
                MemtoReg_o <= 2'b01;
                RegWrite_o <= 1;
                MemRead_o <= 1;
                MemWrite_o <= 0;
                Branch_o <= 0;
                ALU_op_o <= 2'b00;
                Jump_o <= 2'b00;
            end
        10: begin  //slti
                RegDst_o <= 2'b00;
                ALUSrc_o <= 1;
                MemtoReg_o <= 2'b00;
                RegWrite_o <= 1;
                MemRead_o <= 0;
                MemWrite_o <= 0;
                Branch_o <= 0;
                ALU_op_o <= 2'b01;
                Jump_o <= 2'b00;
            end
        43: begin  //sw
                RegDst_o <= 2'b00;
                ALUSrc_o <= 1;
                MemtoReg_o <= 2'b00;
                RegWrite_o <= 0;
                MemRead_o <= 0;
                MemWrite_o <= 1;
                Branch_o <= 0;
                ALU_op_o <= 2'b00;
                Jump_o <= 2'b00;
            end
         default:  begin
                        RegDst_o <= 2'b01;
                        ALUSrc_o <= 0;
                        MemtoReg_o <= 2'b00;
                        RegWrite_o <= 1;
                        MemRead_o <= 0;
                        MemWrite_o <= 0;
                        Branch_o <= 0;
                        ALU_op_o <= 2'b11;
                        Jump_o <= 2'b00;
                    end
    endcase
end
```

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
```

另外還有 MUX_3to1.v，當 select 等於 0，輸出 source0，當 select 等於 1，輸出 source1，當 select 等於 2，輸出 source2。

```
32  //Main function
33  always @(data0_i, data1_i, data2_i, select_i)
34  begin
35      case (select_i)
36          0: data_o <= data0_i;
37          1: data_o <= data1_i;
38          2: data_o <= data2_i;
39          default: data_o <= data0_i;
40      endcase
41  end
```

Shift_Left_Two.v 是將 input data 往左 shift 2 bits。

```
17  //shift left 2
18  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。

```
51  //Greate componentes
52  ProgramCounter PC(      //PC
53          .clk_i(clk_i),
54          .rst_i (rst_i),
55          .pc_in_i(pc_in_i),
56          .pc_out_o(pc_out_o)
57          );
```

```
59  Adder Adder1(          //PC+4
60          .src1_i(32'd4),
61          .src2_i(pc_out_o),
62          .sum_o(pc_plus_four)
63          );
```

```
65  Instr_Memory IM(       //Instruction_memory
66          .pc_addr_i(pc_out_o),
67          .instr_o(instruction)
68          );
```

```
70  MUX_3to1 #(.size(5)) Mux_Write_Reg(
71          .data0_i(instruction[20:16]),
72          .data1_i(instruction[15:11]),
73          .data2_i(5'b11111),
74          .select_i(regdst),
75          .data_o(write_register)
76          );
```

```
78  Reg_File Registers(            //Register_File
79          .clk_i(clk_i),
80          .rst_i(rst_i),
81          .RSaddr_i(instruction[25:21]),
82          .RTaddr_i(instruction[20:16]),
83          .RDaddr_i(write_register),
84          .RDdata_i(writedata),
85          .RegWrite_i(regwrite),
86          .RSdata_o(read_data1),
87          .RTdata_o(read_data2)
88          );
```

```
90  Decoder Decoder(       //Decoder
91          .instr_op_i(instruction[31:26]),
92          .instr_funct_i(instruction[5:0]),
93          .RegWrite_o(regwrite),
94          .ALU_op_o(alu_op),
95          .ALUSrc_o(alusrc),
96          .RegDst_o(regdst),
97          .Branch_o(branch),
98          .Jump_o(jump),
99          .MemRead_o(memread),
100         .MemWrite_o(memwrite),
101         .MemtoReg_o(memtoreg)
102         );
```

```
104 ALU_Ctrl AC(           //ALU_control
105         .funct_i(instruction[5:0]),
106         .ALUOp_i(alu_op),
107         .ALUCtrl_o(alu_control)
108         );
```

```
110 Sign_Extend SE(         //Sign_extend
111         .data_i(instruction[15:0]),
112         .data_o(sign_ext_o)
113         );
```

```
115 MUX_2to1 #(.size(32)) Mux_ALUSrc(
116         .data0_i(read_data2),
117         .data1_i(sign_ext_o),
118         .select_i(alusrc),
119         .data_o(alu_input2)
120         );
```

```
122 ALU ALU(                 //ALU
123         .src1_i(read_data1),
124         .src2_i(alu_input2),
125         .ctrl_i(alu_control),
126         .result_o(alu_result),
127         .zero_o(zero_o)
128         );
```

```
Data_Memory Data_Memory(  //Data_memory
    .clk_i(clk_i),
    .addr_i(alu_result),
    .data_i(read_data2),
    .MemRead_i(memread),
    .MemWrite_i(memwrite),
    .data_o(mem_data_o)
    );
```

```
Adder Adder2(           //branch_adder
        .src1_i(pc_plus_four),
        .src2_i(shift_left_o),
        .sum_o(branch_adder_o)
        );
```

```
Shift_Left_Two_32 Shifter( //branch_shift_left_2
       .data_i(sign_ext_o),
       .data_o(shift_left_o)
       );
```

```
Shift_Left_Two_32 Shifter2( //jump_shift_left_2
        .data_i({6'b000000, instruction[25:0]}),
        .data_o(jump_addr)
        );
```

```
MUX_2to1 #(.size(32)) Mux_Branch_Source(
        .data0_i(pc_plus_four),
        .data1_i(branch_adder_o),
        .select_i(branch_control),
        .data_o(branch_addr_mux_o)
        );
```

```
MUX_3to1 #(.size(32)) Mux_PC_Src(
      .data0_i(branch_addr_mux_o),
      .data1_i({pc_plus_four[31:28], jump_addr[27:0]}),
      .data2_i(read_data1),
      .select_i(jump),
      .data_o(pc_in_i)
      );
```

```
MUX_3to1 #(.size(32)) Mux_WriteReg_Src(
         .data0_i(alu_result),
         .data1_i(mem_data_o),
         .data2_i(pc_plus_four),
         .select_i(memtoreg),
         .data_o(writedata)
         );

assign branch_control = branch & zero_o;
```

Finished part:

Test1:

可以看到在 PC = 20 的時候，也就是 jump j 的指令，jump 有成功運作，並且最後結果的值也都是正確的，代表 lw 和 sw 有成功運作。

```
1  addi    r1,r0,1          r1=1
2  addi    r2,r0,2          r2=2
3  addi    r3,r0,3          r3=3
4  addi    r4,r0,4          r4=4
5  addi    r5,r0,5          r5=5
6  jump    j
7  ---若jump對，以下兩個addi將不會執行
8  addi    r1,r0,31             r1=31
9  addi    r2,r0,32             r2=32
10 ---
11 j:
12 sw  r1,0(r0)        m0=1
13 sw  r2,4(r0)        m1=2
14 lw  r6,0(r0)        r6=1
15 lw  r7,0(r4)        r7=2
16 add r8,r1,r3        r8=4
17 lw  r9,4(r0)        r9=2
```

```
1.  PC =        0
2.  PC =        4
3.  PC =        8
4.  PC =       12
5.  PC =       16
6.  PC =       20
7.  PC =       32
8.  PC =       36
9.  PC =       40
10. PC =       44
11. PC =       48
```

```
- Register File -                                                - Memory Data -
 r0 =   0  r1 =   1  r2 =   2  r3 =   3   m0 =   1  m1 =   2  m2 =   0  m3 =   0
 r4 =   4  r5 =   5  r6 =   1  r7 =   2   m4 =   0  m5 =   0  m6 =   0  m7 =   0
 r8 =   4  r9 =   2 r10 =   0 r11 =   0   m8 =   0  m9 =   0 m10 =   0 m11 =   0
r12 =   0 r13 =   0 r14 =   0 r15 =   0  m12 =   0 m13 =   0 m14 =   0 m15 =   0
r16 =   0 r17 =   0 r18 =   0 r19 =   0  m16 =   0 m17 =   0 m18 =   0 m19 =   0
r20 =   0 r21 =   0 r22 =   0 r23 =   0  m20 =   0 m21 =   0 m22 =   0 m23 =   0
r24 =   0 r25 =   0 r26 =   0 r27 =   0  m24 =   0 m25 =   0 m26 =   0 m27 =   0
r28 =   0 r29 = 128 r30 =   0 r31 =   0  m28 =   0 m29 =   0 m30 =   0 m31 =   0
```

Test2:

前面幾次的 jump 都是 jal fib，也就是跳到 PC = 20，並且在進入 exitfib 的時候，會進行 lw ra,0(sp)，還有 jr ra，而 jr ra 會跳到 PC = 56，也就是除了第一次外的前面幾次 jal 的下一個指令位置，這代表 jal 和 jr 都有成功運作。除此之外，中間幾次的 jump 也都有成功，而且在最後有跳回 PC = 16，也就是 j final，並且結束程式，因此最後的輸出也是正確的。

```
                    43.  PC =        80
11.  PC =       44  44.  PC =        84
12.  PC =       48  45.  PC =        88
13.  PC =       52  46.  PC =        56  143.  PC =       84
14.  PC =       20  47.  PC =        60  144.  PC =       88
15.  PC =       24  48.  PC =        64  145.  PC =       16
```

```
1  add r0,r0,r0
2  addi a0,zero,4      <-f(4),改變r4的值代表 f(r4)，若設太大可能要把data memory設大一些
3  addi t1,zero,1
4  jal fib         <-JAL:當fib function結束後PC會跳到j final
5  j final
6
7  fib:
8  addi sp,sp,-12      //stack pointer -12
9  sw ra,0(sp)     //以下三道sw將reg存入memory中
10 sw s0,4(sp)
11 sw s1,8(sp)
12 add s0,a0,zero
13 beq s0,zero,re1     //判斷是否f(0)
14 beq s0,t1,re1       //判斷是否f(1)
15 addi a0,s0,-1
16 jal fib
17 add s1,zero,v0
18 addi a0,s0,-2
19 jal fib
20 add v0,v0,s1
22 exitfib:
23 lw ra,0(sp)
24 lw s0,4(sp)
25 lw s1,8(sp)
26 addi sp,sp,12
27 jr ra            //function call結束
28
29 re1:
30 addi v0,zero,1
31 j exitfib
32
33 final:
34 nop
```

- Register File -																			
r0 =	0	r1 =	0	r2 =	5	r3 =	0	m0 =	0	m1 =	0	m2 =	0	m3 =	0				
r4 =	0	r5 =	0	r6 =	0	r7 =	0	m4 =	0	m5 =	0	m6 =	0	m7 =	0				
r8 =	0	r9 =	1	r10 =	0	r11 =	0	m8 =	0	m9 =	0	m10 =	0	m11 =	0				
r12 =	0	r13 =	0	r14 =	0	r15 =	0	m12 =	0	m13 =	0	m14 =	0	m15 =	0				
r16 =	0	r17 =	0	r18 =	0	r19 =	0	m16 =	0	m17 =	0	m18 =	0	m19 =	0				
r20 =	0	r21 =	0	r22 =	0	r23 =	0	m20 =	68	m21 =	2	m22 =	1	m23 =	68				
r24 =	0	r25 =	0	r26 =	0	r27 =	0	m24 =	2	m25 =	1	m26 =	68	m27 =	4				
r28 =	0	r29 =	128	r30 =	0	r31 =	16	m28 =	3	m29 =	16	m30 =	0	m31 =	0				

Problems you met and solutions:

這次的 lab 和上次的問題都一樣，都是在於不好 debug，不過助教提供查看 waveform 的方式十分有效，可以讓我不用再一直盯著 code 卻找不到錯的地方。

Summary:

這次的 lab 是從上次做完的成品做進一步的延伸，在有了上次的經驗的情況下，這次的 lab 就顯得比較輕鬆，但還是有地方是需要多花時間才能完成的，例如 jal 和 jr 是上課沒有實做出來的部分，另外因為內容又比上次的還多，因此更需要將各個 module 都整理好，才會比較容易 debug。