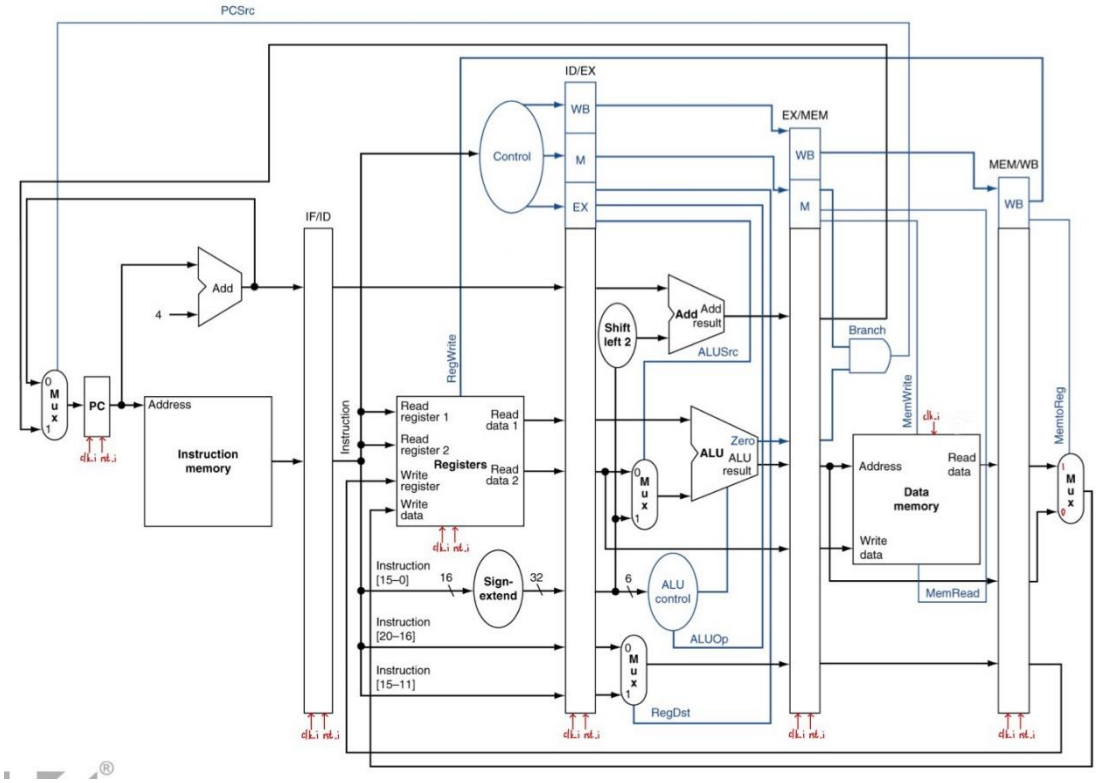
Computer Organization Lab4

ID:110550063　Name:張博凱

Architecture diagrams:

我將連接到 MemtoReg 的 mux 的 0 和 1 對調，比較符合之前的習慣，並且加上 clock 和 reset。



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，和 Lab3 不一樣的是，這次多了 xor 和 mult 要處理。

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
		xor	100110	xor	1111
		slt	101010	slt	0111
		mult	011000	mult	0011

```
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                     24: ALUCtrl_o <= 4'b0011;  //mult
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                     38: ALUCtrl_o <= 4'b1111;  //xor
45                     42: ALUCtrl_o <= 4'b0111;  //slt
46                 endcase
47             end
48         default: ALUCtrl_o <= 4'b0000;
49     endcase
50 end
```

ALU.v 是 ALU，有 and、or、add、subtract、set on less than、nor 這些功能，除此之外，再增加 xor 和 mult，這兩個是 Lab3 沒有的。

```
34  //Main function
35  assign zero_o = (result_o == 0);
36  always @(ctrl_i, src1_i, src2_i)
37  begin
38      case (ctrl_i)
39          0: result_o <= src1_i & src2_i;
40          1: result_o <= src1_i | src2_i;
41          2: result_o <= src1_i + src2_i;
42          3: 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          15: result_o <= src1_i ^ src2_i;
47          default: result_o <= 0;
48      endcase
49  end
```

Decoder.v 是將輸入的 op field 的值對應到相對應的功能，最後輸出各種 control signal，下表是對應的結果，和 Lab3 相比，少了 jump、jal、jr。

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

```
52  //Main function
53  always @(instr_op_i, instr_funct_i)
54  begin
55      case (instr_op_i)
56          0:  begin  //R-format
57                  RegDst_o <= 1;
58                  ALUSrc_o <= 0;
59                  MemtoReg_o <= 0;
60                  RegWrite_o <= 1;
61                  MemRead_o <= 0;
62                  MemWrite_o <= 0;
63                  Branch_o <= 0;
64                  ALU_op_o <= 2'b11;
65              end

66          4:  begin  //beq
67                  RegDst_o <= 0;
68                  ALUSrc_o <= 0;
69                  MemtoReg_o <= 0;
70                  RegWrite_o <= 0;
71                  MemRead_o <= 0;
72                  MemWrite_o <= 0;
73                  Branch_o <= 1;
74                  ALU_op_o <= 2'b10;
75              end

76          8:  begin  //addi
77                  RegDst_o <= 0;
78                  ALUSrc_o <= 1;
79                  MemtoReg_o <= 0;
80                  RegWrite_o <= 1;
81                  MemRead_o <= 0;
82                  MemWrite_o <= 0;
83                  Branch_o <= 0;
84                  ALU_op_o <= 2'b00;
85              end

86          10: begin  //slti
87                  RegDst_o <= 0;
88                  ALUSrc_o <= 1;
89                  MemtoReg_o <= 0;
90                  RegWrite_o <= 1;
91                  MemRead_o <= 0;
92                  MemWrite_o <= 0;
93                  Branch_o <= 0;
94                  ALU_op_o <= 2'b01;
95              end

96          35: begin  //lw
97                  RegDst_o <= 0;
98                  ALUSrc_o <= 1;
99                  MemtoReg_o <= 1;
100                 RegWrite_o <= 1;
101                 MemRead_o <= 1;
102                 MemWrite_o <= 0;
103                 Branch_o <= 0;
104                 ALU_op_o <= 2'b00;
105             end

106         43: begin  //sw
107                 RegDst_o <= 0;
108                 ALUSrc_o <= 1;
109                 MemtoReg_o <= 0;
110                 RegWrite_o <= 0;
111                 MemRead_o <= 0;
112                 MemWrite_o <= 1;
113                 Branch_o <= 0;
114                 ALU_op_o <= 2'b00;
115             end
```

```
116         default:  begin
117                       RegDst_o <= 1;
118                       ALUSrc_o <= 0;
119                       MemtoReg_o <= 0;
120                       RegWrite_o <= 1;
121                       MemRead_o <= 0;
122                       MemWrite_o <= 0;
123                       Branch_o <= 0;
124                       ALU_op_o <= 2'b11;
125                   end
126     endcase
127 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
```

Shift_Left_Two.v 是將 input data 往左 shift 2 bits。

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

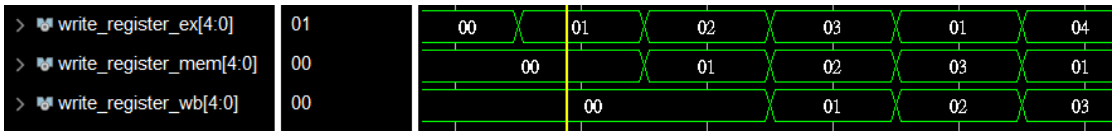
Sign_Extend.v 是將 input data 的 MSB 往左延伸 16 bits。

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

Pipelined_CPU.v 是要把所有的 module 都接線，因此我設定了許多 wire，wire 的名稱基本上都是它的功能再加上他在哪個 state，並且設定好 module 的 input 和 output，這次因為是 pipeline 的結構，多了 pipeline register，因此和之前的 Lab 會相差許多。

Simulation results:

以 write register 為例，從圖可以看到，有 pipeline 的效果，名稱不同表示它在不同的 state 的值，其餘的線路也有同樣的效果。



Test1:

```
- Register File -
 r0 =     0  r1 =     3  r2 =     4  r3 =     1
 r4 =     6  r5 =     2  r6 =     7  r7 =     1
 r8 =     1  r9 =     0 r10 =     3 r11 =     0
r12 =     0 r13 =     0 r14 =     0 r15 =     0
r16 =     0 r17 =     0 r18 =     0 r19 =     0
r20 =     0 r21 =     0 r22 =     0 r23 =     0
r24 =     0 r25 =     0 r26 =     0 r27 =     0
r28 =     0 r29 =     0 r30 =     0 r31 =     0

- Memory Data -
 m0 =     0  m1 =     3  m2 =     0  m3 =     0
 m4 =     0  m5 =     0  m6 =     0  m7 =     0
 m8 =     0  m9 =     0 m10 =     0 m11 =     0
m12 =     0 m13 =     0 m14 =     0 m15 =     0
m16 =     0 m17 =     0 m18 =     0 m19 =     0
m20 =     0 m21 =     0 m22 =     0 m23 =     0
m24 =     0 m25 =     0 m26 =     0 m27 =     0
m28 =     0 m29 =     0 m30 =     0 m31 =     0
```

Test2:

```
- Register File -
 r0 =     0  r1 =     0  r2 =     4  r3 =     5
 r4 =    49  r5 =     0  r6 =     3  r7 =     5
 r8 =     1  r9 =     0 r10 =     7 r11 =     7
r12 =     0 r13 =     0 r14 =     0 r15 =     0
r16 =     0 r17 =     0 r18 =     0 r19 =     0
r20 =     0 r21 =     0 r22 =     0 r23 =     0
r24 =     0 r25 =     0 r26 =     0 r27 =     0
r28 =     0 r29 =     0 r30 =     0 r31 =     0
```

```
- Memory Data -
 m0 =     0  m1 =     7  m2 =     0  m3 =     0
 m4 =     0  m5 =     0  m6 =     0  m7 =     0
 m8 =     0  m9 =     0 m10 =     0 m11 =     0
m12 =     0 m13 =     0 m14 =     0 m15 =     0
m16 =     0 m17 =     0 m18 =     0 m19 =     0
m20 =     0 m21 =     0 m22 =     0 m23 =     0
m24 =     0 m25 =     0 m26 =     0 m27 =     0
m28 =     0 m29 =     0 m30 =     0 m31 =     0
```

Test3:

data hazard 發生時，如果不用 forwarding 解決的話，中間需要有兩個 instructions，才能在 id state 取用 register file 時，wb state 已經將結果寫入 register file。因此我選擇交換 instruction 的位置，讓會發生 data hazard 的 instruction 中間都有兩個 instructions，以下為順序:

1 -> 10 -> 3 -> 2 -> 4 -> 5 -> 8 -> 7 -> 6 -> 9

1 和 2 中間放兩個，5 和 6 中間放兩個，8 和 9 中間放兩個。

```
1  00100000000000010000000000010000
2  00100000000010010000000001100100
3  00100000000000110000000000001000
4  00100000001000100000000000000100
5  10101100000000010000000000000100
6  10001100000001000000000000000100
7  00100000001001110000000000001010
8  00000000011000010011000000100000
9  00000000100000110010100000100010
10 00000000111000110100000000100100
```

```
- Register File -
 r0 =     0  r1 =    16  r2 =    20  r3 =     8
 r4 =    16  r5 =     8  r6 =    24  r7 =    26
 r8 =     8  r9 =   100 r10 =     0 r11 =     0
r12 =     0 r13 =     0 r14 =     0 r15 =     0
r16 =     0 r17 =     0 r18 =     0 r19 =     0
r20 =     0 r21 =     0 r22 =     0 r23 =     0
r24 =     0 r25 =     0 r26 =     0 r27 =     0
r28 =     0 r29 =     0 r30 =     0 r31 =     0

- Memory Data -
 m0 =     0  m1 =    16  m2 =     0  m3 =     0
 m4 =     0  m5 =     0  m6 =     0  m7 =     0
 m8 =     0  m9 =     0 m10 =     0 m11 =     0
m12 =     0 m13 =     0 m14 =     0 m15 =     0
m16 =     0 m17 =     0 m18 =     0 m19 =     0
m20 =     0 m21 =     0 m22 =     0 m23 =     0
m24 =     0 m25 =     0 m26 =     0 m27 =     0
m28 =     0 m29 =     0 m30 =     0 m31 =     0
```

Problems you met and solutions:

這次的 lab 是寫 pipeline cpu，相較於前面幾次的 lab 又更亂更雜，在想每個 wire 的名字時想不到好的方式去命名，原本是想要沿用上次的命名方式，可是這次的 wire 太多，所以作罷。不過之後想到可以利用所在的 state 去分類，並且把每個功能相同 wire 都命名成一樣的名字，只修改最後的 state 名，這樣下來可讀性也大幅提升。

Summary:

這次做得 pipeline cpu，原本想說會很複雜且不好做，但仔細看就能發現，其實這次要實作的部分，絕大部分都是在接線，除此之外，只需要在 alu、alu_ctrl、和 decoder 做出些微的修改，讓整個 cpu 可以實作出 xor 和 mult 的功能即可，有了之前的 lab 的經驗，整體的難度不高，只不過在打的時候要格外小心，以免錯在一些小地方而 debug 不出來。