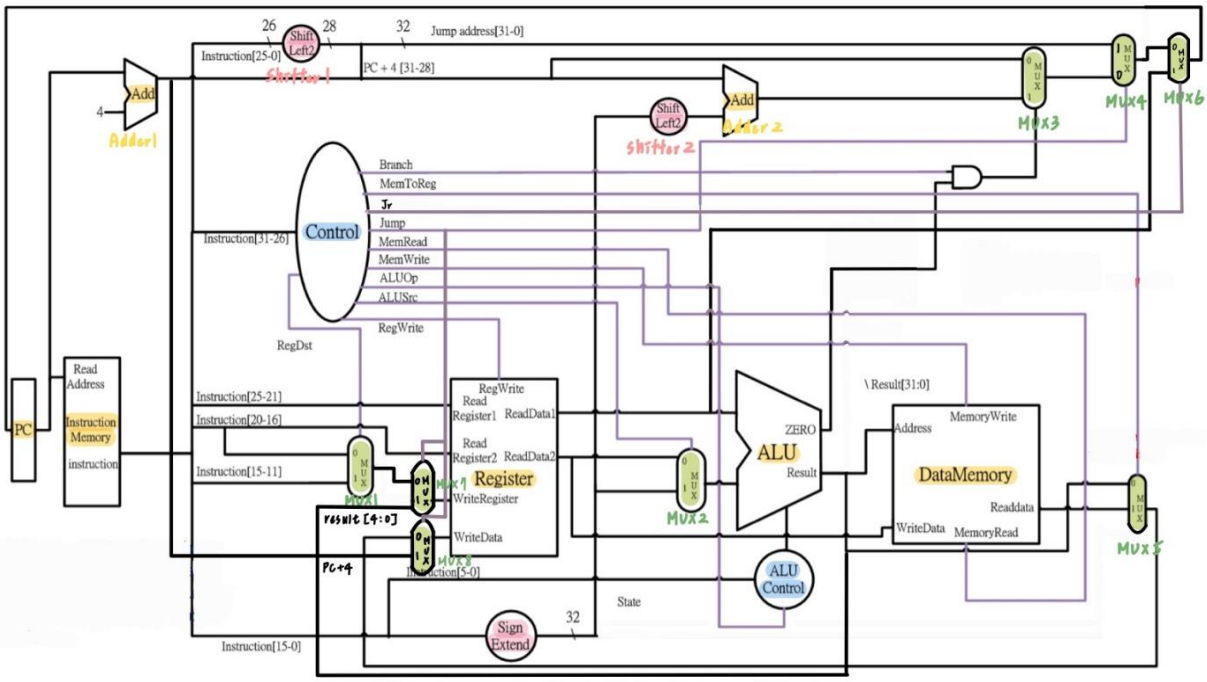
Computer Organization Lab3

Name: 龔祐萱

ID: 0713216

Architecture diagrams:

依照作業中所給的 CPU 電路圖，為使其符合各項指令要求，進行了以下幾個主要的修改，並整理如下圖:



1. jr 指令

為了使 PC 可以指到 Reg[31]所存的指令，增加了一個 MUX6 以及增加一個控制訊號 Jr 來控制 MUX6 的執行。如果 Jr=1，表示執行 jr 指令，因此輸出 Reg[31]所儲存的指令到 PC；如果 Jr=0，表示執行 jr 以外的指令，則依該指令的規定輸出下一個指令的位置至 PC。

2. jal 指令

在不更改 Register File 的前提之下，為了使下一個指令可以儲存至 Reg[31]中，增加了 MUX7 和 MUX8，兩者的控制訊號皆為 Jump。如果執行 jal 指令時，控制訊號 ALUOp=1、ALU_control_output=3，使 ALU 會輸出 5 bit 的 1 至 MUX7。同時，控制訊號 Jump=1，使得 MUX7 會將 ALU 的 output 輸至 WriteRegister 中表示 Reg[31]，MUX8 會將 PC+4 輸至 WriteData，將下一個指令存入 Reg[31]中。

3. jump 指令

為了使控制訊號的輸出較直覺，因此修改了 MUX4 的輸出結果。當 Jump=1 時，MUX4 會輸出 jump address；當 Jump=0 時，MUX4 會輸出 PC+4 或 target address。

Hardware module analysis:

1. **<u>R-type instruction (add, sub, and, or, slt) 執行過程</u>**

   (1) PC 會指到 instruction memory 中的一個位置，並將該位置的指令輸出

   (2) 指令的 31-26 bit 會輸入 control (decoder)進行解碼，並輸出 control signal

   指令的 25-21 bit 會輸入 Register 作為 Read Register1，並輸出該 register 的資料

   指令的 20-16 bit 會輸入 Register 作為 Read Register2，並輸出該 register 的資料

   指令的 15-11 bit 會輸入 Register 作為 WriteRegister

   指令的 5-0 bit 會輸入 ALU control，判斷此指令的 function field 是對應到哪種運算方式，並輸入至 ALU 中

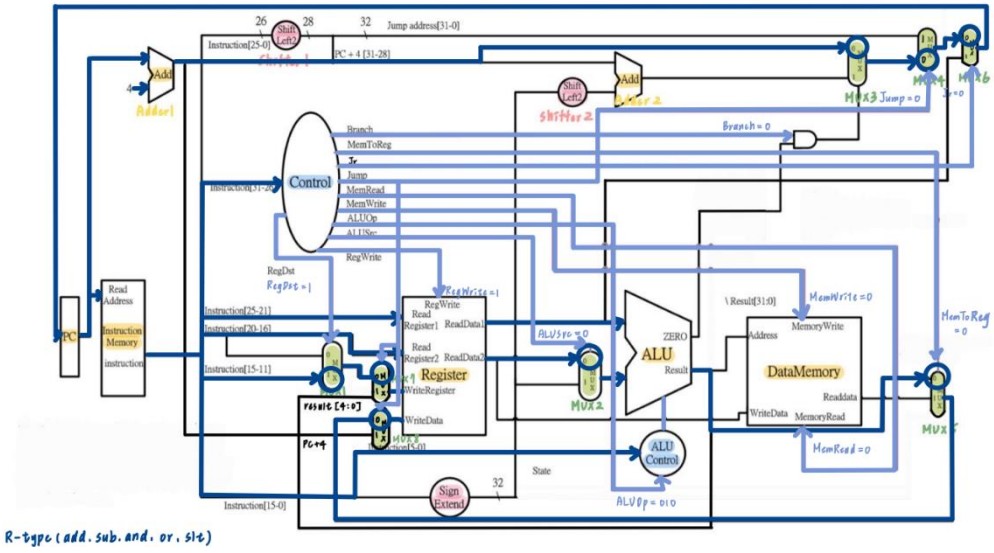
   (3) 從 register 輸出的 data 會輸入 ALU 進行運算，而運算方式是由 ALU _control_ output 判斷

   (4) 將 ALU 的運算結果送回 register file，並寫入 WriteRegister 中

   (5) PC 會指到下一個要執行的指令(PC+4)

   (6) 以下是 R-type instruction 的 control signal 整理表格以及執行流程圖

RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite
1	0	0	1	0	0
Branch	ALUOp (3 bits)	Jump	Jr	ALU_control_output (4 bits)	
0	010	0	0	add : 0010 sub : 0110 and : 0000 or : 0001 slt : 0111	



2. R-type instruction (jr)執行過程

(1) PC 會指到 instruction memory 中的一個位置，並將該位置的指令輸出

(2) 指令的 31-26 bit 會輸入 control (decoder)進行解碼，並輸出 control signal

指令的 25-21 bit 會輸入 Register 作為 Read Register1，並輸出該 register 的資料

指令的 20-16 bit 會輸入 Register 作為 Read Register2

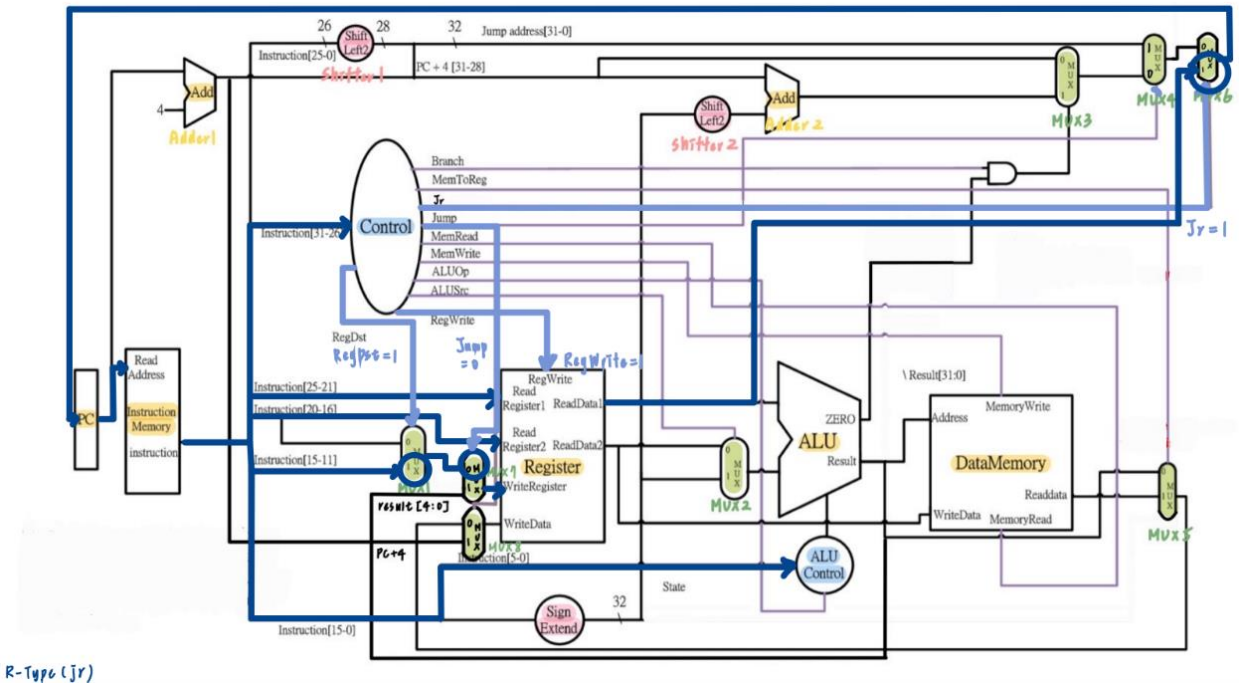
指令的 15-11 bit 會輸入 Register 作為 WriteRegister

指令的 5-0 bit 會輸入 ALU control，判斷此指令的 function field 是對應到哪種運算方式，並輸入至 ALU 中，但因 jr 的 ALUOp 無法對應到 ALU 中所設定的運算，因此 ALU default

(3) Register 所輸出的 RS_data 會輸至 MUX6 中並輸出，表示 PC 會指到下一個要執行的指令

(4) 以下是 R-type instruction 的 control signal 整理表格以及執行流程圖

RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite
1	0	0	1	0	0
Branch	ALUOp (3 bits)	Jump	Jr	ALU_control_output (4 bits)	
0	010	0	0	1111	



3. addi / slti instruction 執行過程

(1) PC 會指到 instruction memory 中的一個位置，並將該位置的指令輸出

(2) 指令的 31-26 bit 會輸入 control (decoder)進行解碼，並輸出 control signal

指令的 25-21 bit 會輸入 Register 作為 Read Register1，並輸出該 register 的資料

指令的 20-16 bit 會輸入 Register 作為 WriteRegister

指令的 15-0 bit 會輸入 sign extend 中，將 16 bits 的 constant 變成 32 bits 的 constant，並輸入至 ALU 執行運算

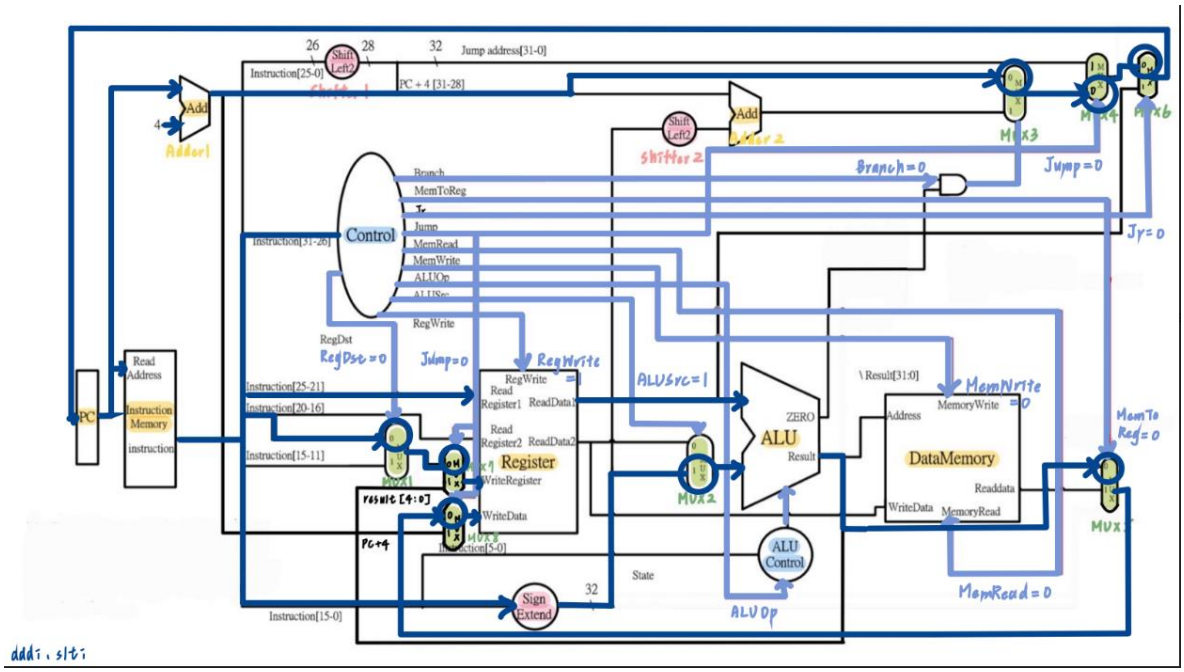
(3) 從 register 的 data 以及執行過 sign extension 的 32 bits constant 會輸入 ALU 進行運算，而運算方式是由 ALU _control_ output 判斷

(4) 將 ALU 的運算結果送回 Register，並寫入 WriteRegister 中

(5) PC 會指到下一個要執行的指令(PC+4)

(6) 以下是 addi / slti instruction 的 control signal 整理表格以及執行流程圖

RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite
0	1	0	1	0	0
Branch	ALUOp (3 bits)	Jump	Jr	ALU_control_output (4 bits)	
0	addi : 000 slti : 011	0	0	addi : 1000 slti : 0101	



4. <u>beg instruction 執行過程</u>

(1) PC 會指到 instruction memory 中的一個位置，並將該位置的指令輸出

(2) 指令的 31-26 bit 會輸入 control (decoder)進行解碼，並輸出 control signal

指令的 25-21 bit 會輸入 Register 作為 Read Register1，並輸出該 register 的資料

指令的 20-16 bit 會輸入 Register 作為 Read Register2，並輸出該 register 的資料

指令的 15-0 bit 會輸入 sign extend 中，將 16 bits 的 address 變成 32 bits 的 address，並輸入至 shifter 向左移 2 bits 並加上 PC+4[31:28]變成 target address

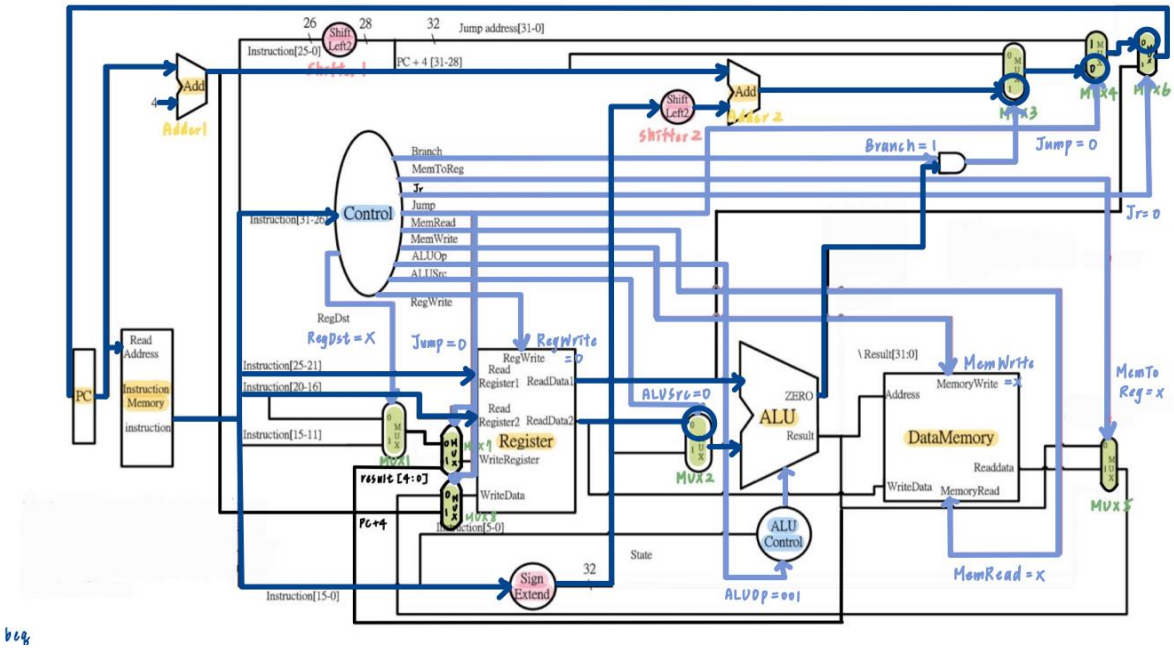
(3) 從 Register 輸出的 data 會輸入 ALU 進行運算，而運算方式是由 ALU _control_ output 判斷

(4) ALU 會輸出 Zero 判斷兩個輸入 ALU 的值是否相等，Zero 和 Branch 會決定 PC 下一個指到的指令位置

(5) PC 會指到下一個要執行的指令(PC+4 or target address)

(6) 以下是 beq instruction 的 control signal 整理表格以及執行流程圖

RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite
0	0	0	0	0	0
Branch	ALUOp (3 bits)	Jump	Jr	ALU_control_output (4 bits)	
1	001	0	0	1010	



5. lw instruction 執行過程

(1) PC 會指到 instruction memory 中的一個位置，並將該位置的指令輸出

(2) 指令的 31-26 bit 會輸入 control (decoder)進行解碼，並輸出 control signal
指令的 25-21 bit 會輸入 Register 作為 Read Register1，並輸出該 register 的資料
指令的 20-16 bit 會輸入 Register 作為 WriteRegister
指令的 15-0 bit 會輸入 sign extend 中，將 16 bits 的 constant 變成 32 bits 的 constant，並輸入至 ALU 執行運算

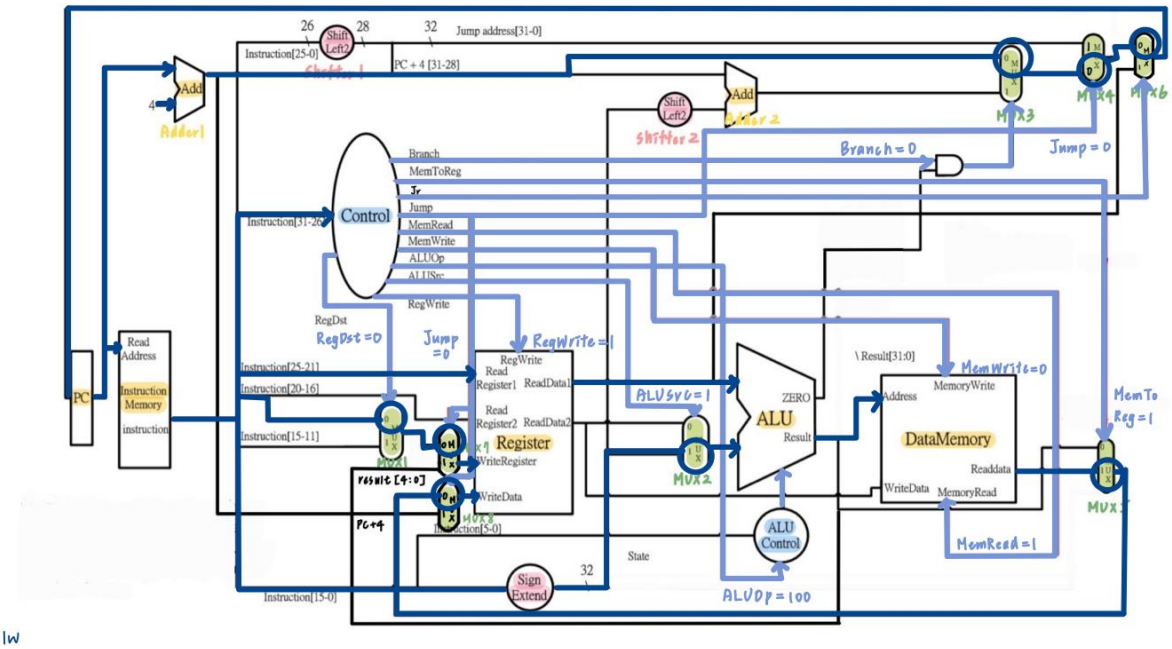
(3) 從 Register 輸出的 data 以及執行過 sign extension 的 32 bits constant 會輸入 ALU 進行運算，而運算方式是由 ALU _control_ output 判斷，算出要讀取資料的 DataMemory 位置

(4) 將 DataMemory 所讀取的資料送回 Register，並將 WriteData 寫入 WriteRegister 中

(5) PC 會指到下一個要執行的指令(PC+4)

(6) 以下是 lw instruction 的 control signal 整理表格以及執行流程圖

RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite
0	1	0	1	1	0
Branch	ALUOp (3 bits)	Jump	Jr	ALU_control_output (4 bits)	
0	100	0	0	0010	



6. sw instruction 執行過程

(1) PC 會指到 instruction memory 中的一個位置，並將該位置的指令輸出

(2) 指令的 31-26 bit 會輸入 control (decoder)進行解碼，並輸出 control signal
指令的 25-21 bit 會輸入 Register 作為 Read Register1，並輸出該 register 的資料
指令的 20-16 bit 會輸入 Register 作為 Read Register2
指令的 15-0 bit 會輸入 sign extend 中，將 16 bits 的 constant 變成 32 bits 的 constant，並輸入至 ALU 執行運算

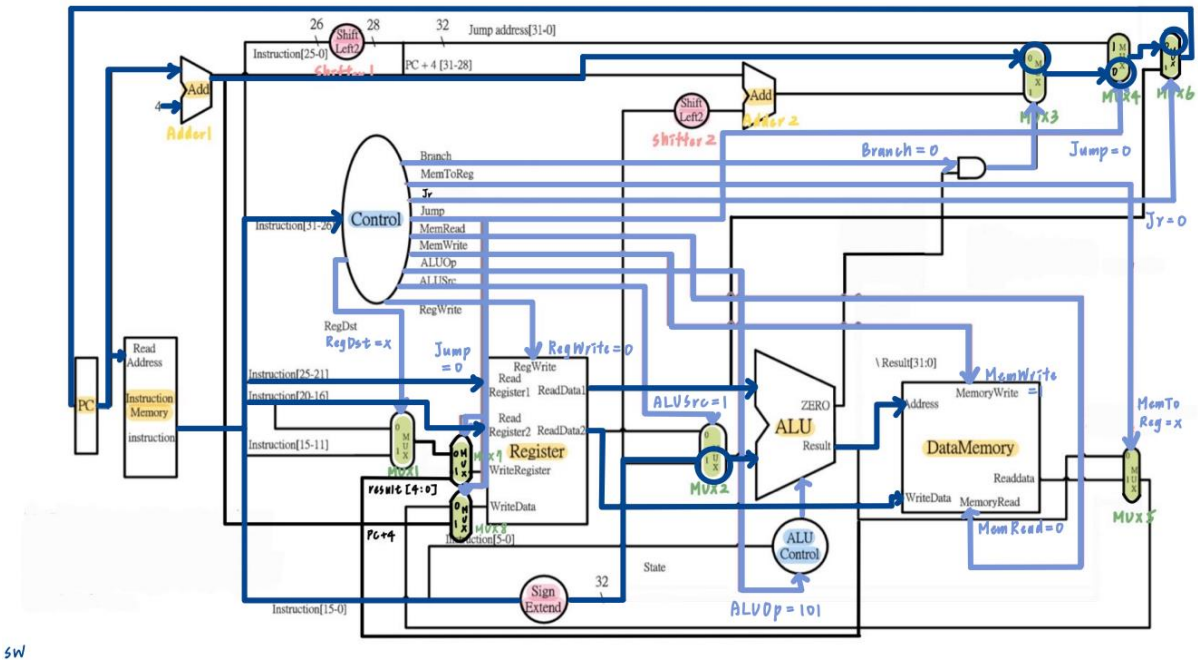
(3) 從 Register 輸出的 data 以及執行過 sign extension 的 32 bits constant 會輸入 ALU 進行運算，而運算方式是由 ALU _control_ output 判斷，即可得到要存取的 DataMemory 位置

(4) 將 DataMemory 的 WriteData 存入經 ALU 運算所得的 DataMemory 位置

(5) PC 會指到下一個要執行的指令(PC+4)

(6) 以下是 sw instruction 的 control signal 整理表格以及執行流程圖

RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite
0	1	0	0	0	1
Branch	ALUOp (3 bits)	Jump	Jr	ALU_control_output (4 bits)	
0	101	0	0	0010	



7. jump instruction 執行過程

(1) PC 會指到 instruction memory 中的一個位置，並將該位置的指令輸出

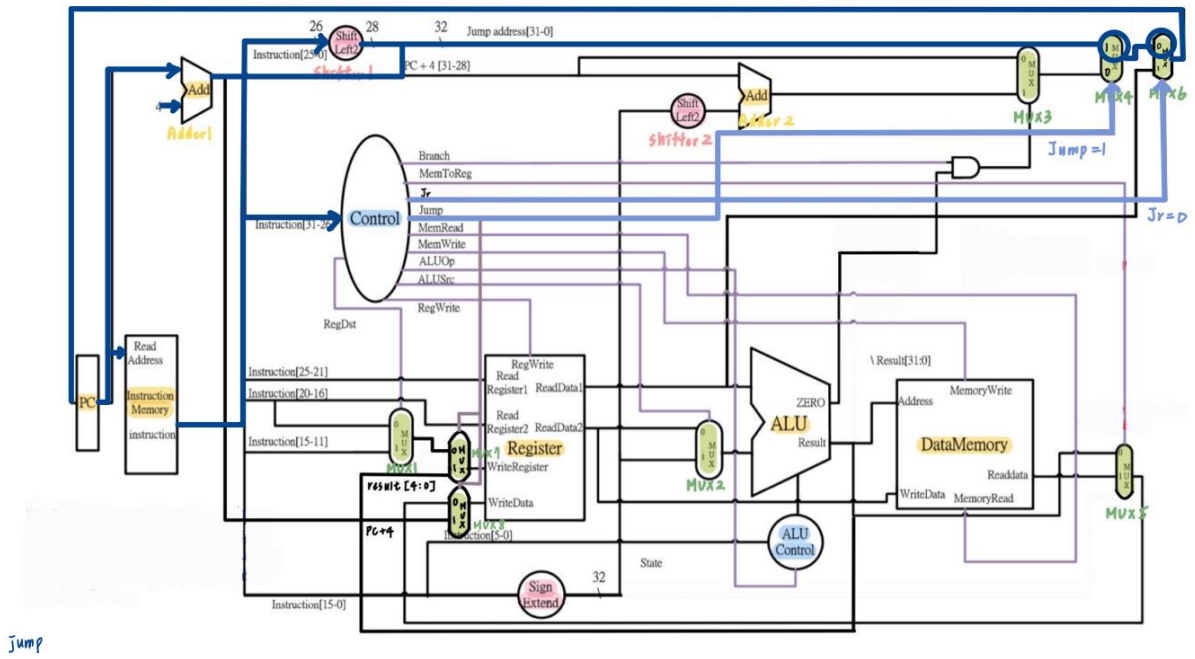
(2) 指令的 31-26 bit 會輸入 control (decoder)進行解碼，並輸出 control signal
指令的 25-0 bit 會輸入 shifter 左移 2 bits

(3) 從 shifter 輸出的 data 會和 PC+4 的 31-28 bit 結合，形成 jump address

(4) PC 會指到下一個要執行的指令(jump address)

(5) 以下是 jump instruction 的 control signal 整理表格以及執行流程圖

RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite
0	0	0	0	0	0
Branch	ALUOp (3 bits)	Jump	Jr	ALU_control_output (4 bits)	
0	111	1	0	1111	



Jump

8. jal instruction 執行過程

(1) PC 會指到 instruction memory 中的一個位置，並將該位置的指令輸出

(2) 指令的 31-26 bit 會輸入 control (decoder)進行解碼，並輸出 control signal
指令的 25-0 bit 會輸入 shifter 左移 2 bits

(3) 從 shifter 輸出的 data 會和 PC+4 的 31-28 bit 結合，形成 jump address

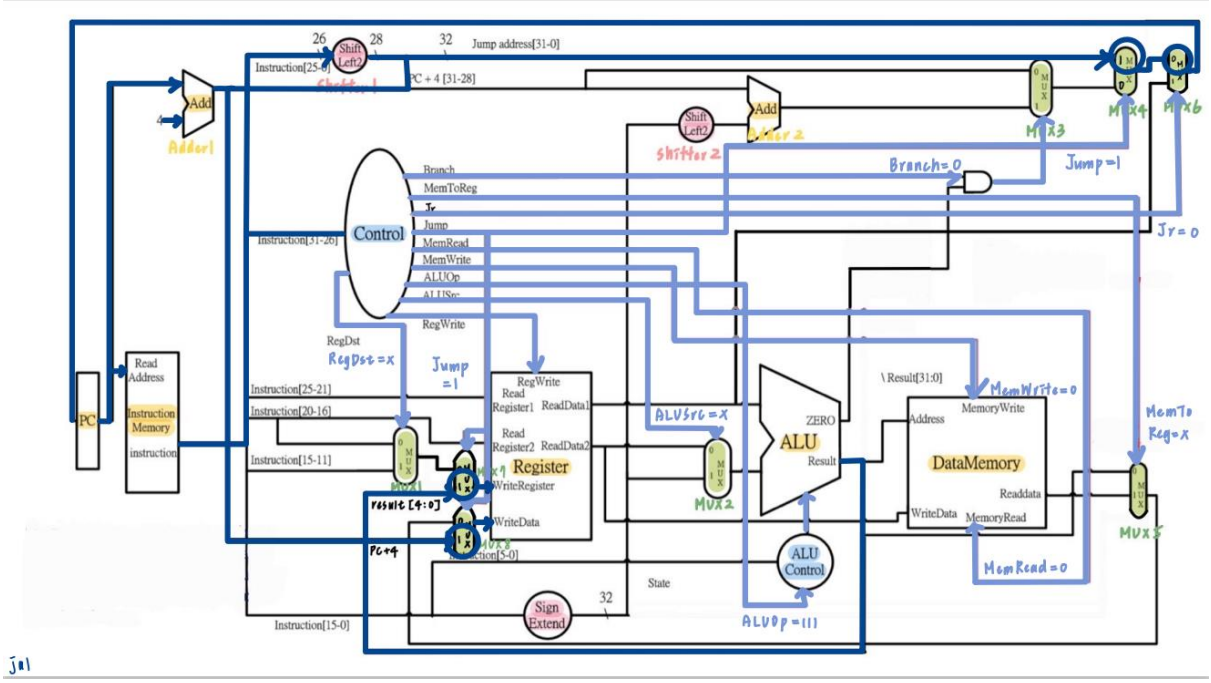
(4) 因 ALUOp 為 110，因此 ALU control 會輸出控制訊號 0011 至 ALU 中，使 ALU 強制輸出 5 bit 11111 至 Register 的 WriteRegister 中

(5) 把 PC+4 輸入至 Register 的 WriteData 中，即可將下一個指向的指令存入 Reg[31]中

(6) PC 會指到下一個要執行的指令(jump address)

(7) 以下是 jal instruction 的 control signal 整理表格以及執行流程圖

RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite
0	0	0	0	0	0
Branch	ALUOp (3 bits)	Jump	Jr	ALU_control_output (4 bits)	
0	110	1	0	0011	



9. 優缺點

可以透過 control signal 去執行不同的指令，不需要更複雜的硬體去實作，且使用 2 level 的 control 可以先利用 opcode 區分 R-type、addi / slti、beq、lw/sw、jump 和 jal 的指令，再利用指令中的 function field 去判斷指令為 R-type 中的哪種運算，實作上會比將 opcode 和 function field 放在一起輸入 control unit 判斷更有效率。

而 single cycle CPU 會因為每個指令所需要的執行時間不同，而取執行時間最長的指令作為 clock period。如果有些較常用的指令的所需執行時間較短，則 CPU 在執行指令時會較常出現 idle time，使整體 CPU 的執行效率不高，且不符合 design principle (Making the common case fast)。

Finished part:

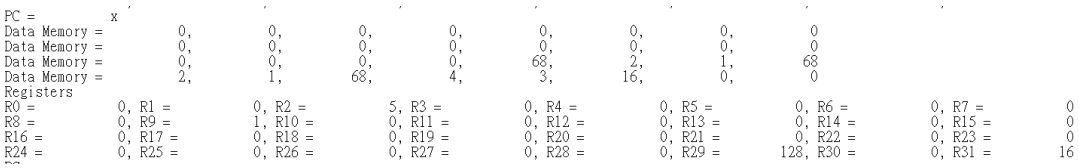
以下是測資的執行指令、指令執行過程、執行結果以及程式執行結果截圖:

1. Case 1

執行指令	指令執行過程	執行結果 (register)	執行結果 (memory)
(1) addi r1, r0, 1 (2) addi r2, r0, 2 (3) addi r3, r0, 3 (4) addi r4, r0, 4 (5) addi r5, r0, 5 (6) jump j (7) addi r1, r0, 31 (8) addi r2, r0, 32 (9) j: (10) sw r1, 0(r0) (11) sw r2, 4(r0) (12) lw r6, 0(r0) (13) lw r7, 0(r4) (14) add r8, r1, r3 (15) lw r9, 4(r0)	(1) r1 = r0+1 = 0+1 = 1 (2) r2 = r0+2 = 0+2 =2 (3) r3 = r0+3 = 0+3 =3 (4) r4 = r0+4 = 0+4 =4 (5) r5 = r0+5 = 0+5 =5 (6) Jump to j (7) Store r1=1 to m0, m0 = 1 (8) Store r2=2 to m1, m1 = 2 (9) Load m0=1 to r6, r6 = 1 (10) Load m1=2 to r7, r7 = 2 (11) r8 = r1+r3 = 1+2 = 3 (12) Load m1=2 to r9, r9 = 2	R0 = 0 R1 = 1 R2 = 2 R3 = 3 R4 = 4 R5 = 5 R6 = 1 R7 = 2 R8 = 4 R9 = 2 R10-31 = 0 R29 = 128	M1 = 1 M2 = 2 M3-M31 = 0
程式執行結果截圖			
PC = x Data Memory = 1, 2, 0, 0, 0, 0, 0, 0 Data Memory = 0, 0, 0, 0, 0, 0, 0, 0 Data Memory = 0, 0, 0, 0, 0, 0, 0, 0 Data Memory = 0, 0, 0, 0, 0, 0, 0, 0 Registers R0 = 0, R1 = 1, R2 = 2, R3 = 3, R4 = 4, R5 = 5, R6 = 1, R7 = 2 R8 = 4, R9 = 2, 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 = 128, R30 = 0, R31 = 0			

2. Case2

執行指令	指令執行過程	執行結果 (register)	執行結果 (memory)
(1) add r0, r0, r0 (2) addi a0, zero, 4 (3) addi t1, zero, 1 (4) jal fib (5) j final (6) jump j (7) fib: (8) addi sp, sp, -12 (9) sw ra, 0(sp) (10) sw s0, 4(sp) (11) sw s1, 8(sp) (12) add s0, a0, zero (13) beq s0, zero, re1 (14) beq s0, t1, re1 (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 (21) exitfib (22) lw ra, 0(sp) (23) lw s0, 4(sp) (24) lw s1, 8(sp) (25) addi sp, sp, 12 (26) jr ra (27) re1: (28) addi v0, zero, 1 (29) j exitfib (30) final: (31) nop	做完 add 和 addi 後，會將 return address 存入 Reg[31]，並跳至 fib 進行運算，完成 fib 的運算後會跳回 Reg[31]所儲存的 return address，最後跳至 final 結束整個程式的執行	R0 = 0 R1 = 0 R2 = 5 R3-R28 = 0 R9 = 1 R29 = 128 R30 = 0 R31 = 16	M1-M19 = 0 M20 = 68 M21 = 2 M22 = 1 M23 = 68 M24 = 2 M25 = 1 M26 = 68 M27 = 4 M28 = 3 M29 = 16 M30 = 0 M31 = 0
程式執行結果截圖			



3. Case3

執行指令	指令執行過程	執行結果 (register)	執行結果 (memory)
(1) addi r1, r0, 10 (2) addi r2, r0, 4 (3) slt r3, r1, r2 (4) beq r3, r0, 1 (5) add r4, r1, r2 (6) sub r5, r1, r2	(1) r1=r0+10=0+10=10 (2) r2=r0+4=0+4=4 (3) (r1=10)≮(r2=4), r3=0 (4) (r3=0)=(r0=0) goto PC+4 (5) r5=r1-r2=10-4=6	R0 = 0 R1 = 10 R2 = 4 R3 = 0 R4 = 0 R5 = 6 R6-31 = 0 R29 = 128	M1-M31 = 0

程式執行結果截圖

```
PC =          x
Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
Registers
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, 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 =       128, R30 =         0, R31 =         0
PC =          x
```

4. Case4

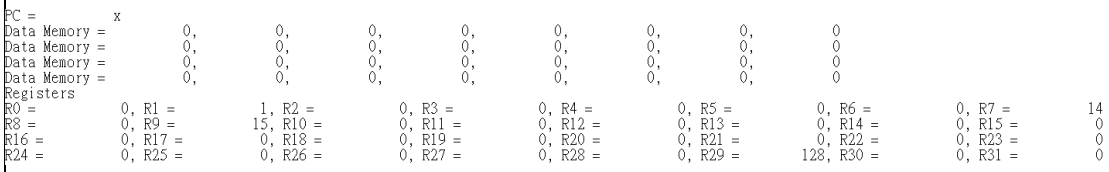
執行指令	指令執行過程	執行結果 (register)	執行結果 (memory)
(1) addi r1, r0, 2 (2) addi r2, r0, 4 (3) slt r3, r1, r2 (4) beq r3, r0, 1 (5) add r4, r1, r2 (6) sub r5, r1, r2	(1) r1=r0+10=0+2=2 (2) r2=r0+4=0+4=4 (3) (r1=2)<(r2=4), r3=1 (4) (r3=1) ≠ (r0=0), no branch (5) r4=r1+r2=2+4=6 (6) r5=r1-r2=2-4=-2	R0 = 0 R1 = 2 R2 = 4 R3 = 1 R4 = 6 R5 = -2 R6-31 = 0 R29 = 128	M1-M31 = 0

程式執行結果截圖

```
PC =          x
Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
Data Memory =          0,          0,          0,          0,          0,          0,          0,          0
Registers
R0 =          0, R1 =          2, R2 =          4, R3 =          1, R4 =          6, R5 =         -2, R6 =          0, R7 =          0
R8 =          0, R9 =          0, 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 =       128, R30 =         0, R31 =         0
PC =          x
```

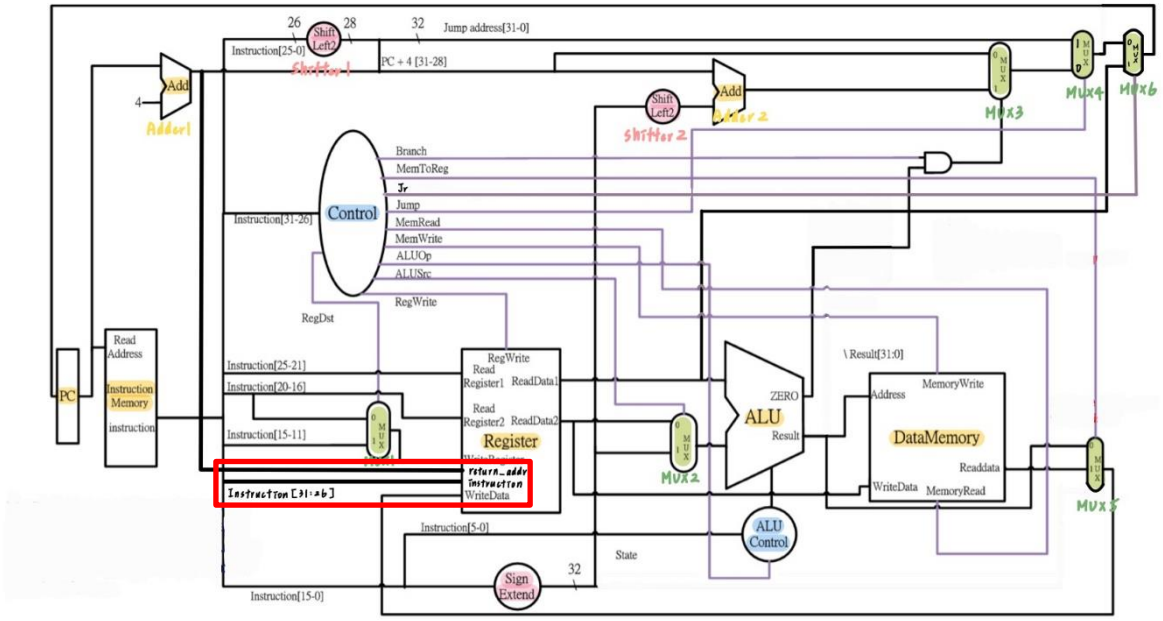
5. Case5

執行指令	指令執行過程	執行結果 (register)	執行結果 (memory)
(1) addi r6, r0, 2 (2) addi r7, r0, 14 (3) and r8, r6, r7 (4) or r9, r6, r7 (5) addi r6, r6, -1 (6) slti r1, r6, 1 (7) beq r1, r0, -5	(1) r6=r0+2=0+2=2 (2) r7=r0+14=0+14=14 (3) r8=r6 AND r7=2 (4) r9=r6 OR r7=14 (5) r6=r6-1=2-1=1 (6) (r6=1)≮1, r1=0 (7) (r1=0)=(r0=0) goto PC-20 (8) r7=r0+14=0+14=14 (9) r8=r6 AND r7 = 0 (10) r9=r6 OR r7 = 15 (11) r6=r6-1=1-1=0 (12) (r6=0)<1, r1=1 (13) (r1=1)≠(r0=0) , (14) no branch	R0 = 0 R1 = 1 R2 = 0 R3 = 0 R4 = 0 R5 = 0 R6 = 0 R7 = 14 R8 = 0 R9 = 15 R10-31 = 0 R29 = 128	M1-M31 = 0
程式執行結果截圖			



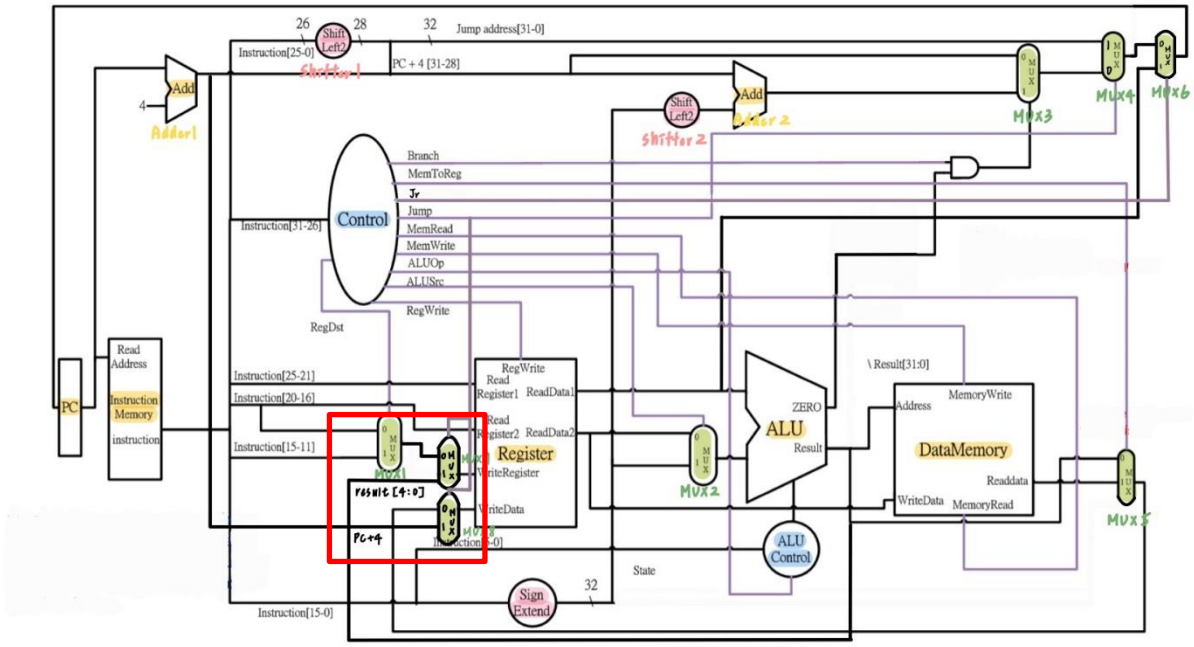
Problems you met and solutions:

在實作 jal 指令時遇到了問題，因 jal 指令的組成為 opcode 和 address，所以無法透過指令現有的資訊來指定 Reg[31]儲存下一個指令的位置，所以就進行了 Register File 的修改，在 Register File 中設定一個參數為 31，並新增一個 input，可以將 opcode 輸入 Register File 中辨識現在所執行的指令是否為 jal，如果現在所執行的指令為 jal 就將 Reg[31]存入 PC+4。以下為更改 Register File 的電路設計圖以及程式碼:




```
//Internal signals/registers
reg  signed [32-1:0] REGISTER_BANK [0:32-1];    //32 word registers
wire        [32-1:0] RSdata_o;
wire        [32-1:0] RTdata_o;

parameter return = 5'd31;

//Read the data
assign RSdata_o = REGISTER_BANK[RSaddr_i] ;
assign RTdata_o = REGISTER_BANK[RTaddr_i] ;

//Writing data when postive edge clk_i and RegWrite_i was set.
always @( posedge rst_i or posedge clk_i  ) begin
   if(rst_i == 0) begin
      REGISTER_BANK[0]  <= 0; REGISTER_BANK[1]  <= 0; REGISTER_BANK[2]  <= 0; REGISTER_BANK[3]  <= 0;
      REGISTER_BANK[4]  <= 0; REGISTER_BANK[5]  <= 0; REGISTER_BANK[6]  <= 0; REGISTER_BANK[7]  <= 0;
      REGISTER_BANK[8]  <= 0; REGISTER_BANK[9]  <= 0; REGISTER_BANK[10] <= 0; REGISTER_BANK[11] <= 0;
      REGISTER_BANK[12] <= 0; REGISTER_BANK[13] <= 0; REGISTER_BANK[14] <= 0; REGISTER_BANK[15] <= 0;
      REGISTER_BANK[16] <= 0; REGISTER_BANK[17] <= 0; REGISTER_BANK[18] <= 0; REGISTER_BANK[19] <= 0;
      REGISTER_BANK[20] <= 0; REGISTER_BANK[21] <= 0; REGISTER_BANK[22] <= 0; REGISTER_BANK[23] <= 0;
      REGISTER_BANK[24] <= 0; REGISTER_BANK[25] <= 0; REGISTER_BANK[26] <= 0; REGISTER_BANK[27] <= 0;
      REGISTER_BANK[28] <= 0; REGISTER_BANK[29] <= 128; REGISTER_BANK[30] <= 0; REGISTER_BANK[31] <= 0;
   end
   else begin
   if(instruction == 6'b000011) begin
         REGISTER_BANK[return] <= return_addr;
      end
      if(RegWrite_i)
         REGISTER_BANK[RDaddr_i] <= RDdata_i;
      else
         REGISTER_BANK[RDaddr_i] <= REGISTER_BANK[RDaddr_i];
   end
end

endmodule
```

但是，後來發現可以透過增加 MUX 和強制 ALU 輸出 5 bit 1 的方式來達到同樣的效果，且不用修改 Register File，所以就針對程式碼進行修正，增加了兩個 MUX 分別控制進入 WriteRegister 和 WriteData 的 data，並分別將 ALU_result 和 PC+4 輸至 MUX7 和 MUX8。以下為修改過後的電路圖:



Summary:

透過這次 lab 可以更加認識 single cycle CPU 各個指令的運作方式，並且以 lab2 為基礎，新增 lw/sw、jump、jal、jr 等指令，使得 CPU 的功能更加完整。雖這然這次碰到較多問題，也前前後後修改程式很多次，但也透過網路上的參考資料以及和同學討論過後找到自己問題所在，解決問題，同時也可以參考其他同學所遇到的問題以及想法，來找出更好的解決或實作方案。