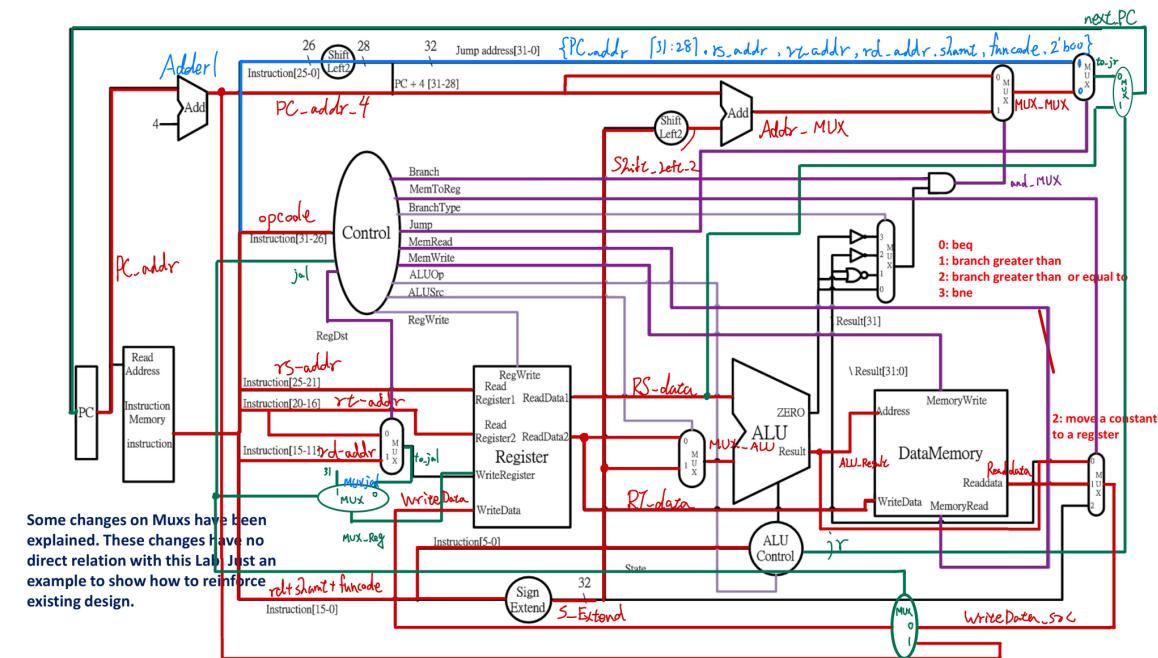


Computer Organization

Architecture diagrams:



上圖是此次作業的架構，主要是參照上圖做接線，紅線綠色線為多加jr, jal的MUX與Control的output。

Hardware module analysis:

i. lw (load word)

Reg[Rt] \leftarrow Mem[Rs+Imm]

6'b100011	Rs[25:21]	Rt[20:16]	Imm[15:0]
-----------	-----------	-----------	-----------

右圖為load的控制訊號，依照教授給的講義（上圖）可得出右圖的控制訊號。

```
'b100011: begin // lw
  RegWrite_o <= 1;
  ALU_op_o <= 3'b111;
  ALUSrc_o <= 1;
  RegDst_o <= 0;
  Branch_o <= 0;
  Jump_o <= 0;
  MemRead_o <= 1;
  MemWrite_o <= 0;
  MemtoReg_o <= 1;

  jal_o <= 0;

end
```

ii. sw (store word)

Mem[Rs+Imm] \leftarrow Reg[Rt]

6'b101011	Rs[25:21]	Rt[20:16]	Imm[15:0]
-----------	-----------	-----------	-----------

如同load，藉由上圖可得save的控制訊號。

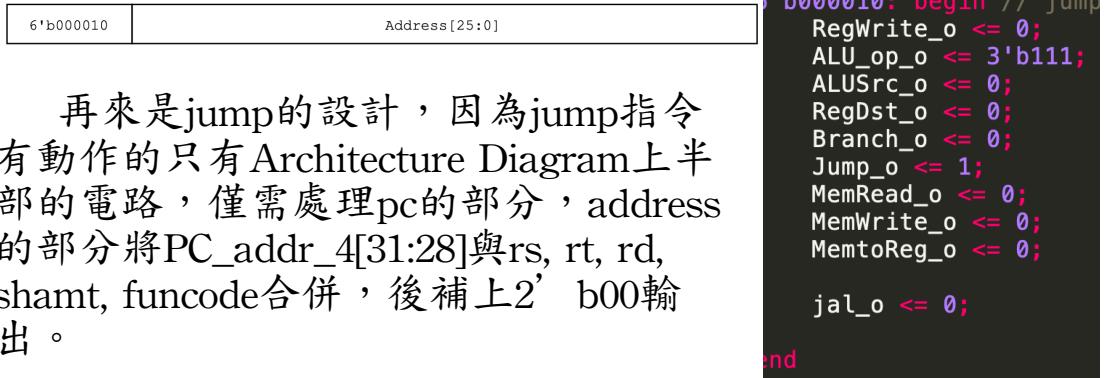
```
'b101011: begin // sw
  RegWrite_o <= 0;
  ALU_op_o <= 3'b111;
  ALUSrc_o <= 1;
  RegDst_o <= 0;
  Branch_o <= 0;
  Jump_o <= 0;
  MemRead_o <= 0;
  MemWrite_o <= 1;
  MemtoReg_o <= 0;

  jal_o <= 0;

end
```

iii. jump

$PC \leftarrow \{ PC[31:28], address << 2 \}$



i. jal (jump and link)

In MIPS, the 31th register saves return address for function calls.

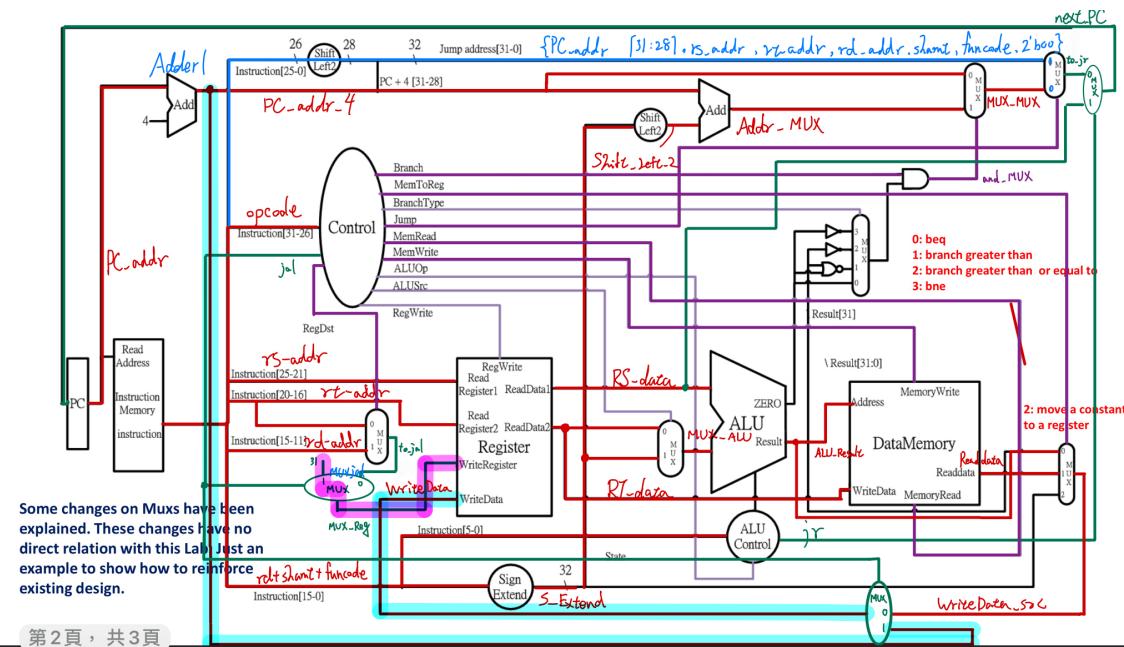
$Reg[31] \leftarrow PC+4$
 $PC \leftarrow \{ PC[31:28], address << 2 \}$

6'b000011	Address [25:0]
-----------	----------------

而jump and link的設計，我在decoder裡增加了jal這個control，運作方式如下圖：

```
'b00011: begin // jal
    RegWrite_o <= 1;
    ALU_op_o <= 3'b100;
    ALUSrc_o <= 0;
    RegDst_o <= 0;
    Branch_o <= 0;
    Jump_o <= 1;
    MemRead_o <= 0;
    MemWrite_o <= 0;
    MemtoReg_o <= 0;

    jal_o <= 1;
end
```



先將\$ra沿紫色螢光線寫入WriteRegister (5' b11111)，將PC_addr_4由藍色螢光線送至WriteData，兩者中間都以jal控制訊號作為selector，藉以達成將PC_addr_4存入\$ra的操作。

ii. jr (jump register)

In MIPS, you can use

jr r31

to jump to the return address linked from **jal** instruction.

PC \leftarrow Reg[Rs]

6'b000000	Rs[25:21]	0	0	0	6'b0001000
-----------	-----------	---	---	---	------------

最後是jr的實作，我在ALUCtrl加了一個控制訊號jr，屬於R-type的指令，這裡我的想法是，把jr情況的ALUCtrl設為and，因為是R-type就會更新register，而如果是其他指令的話zero值便會出錯，而因為輸入的rd為0，故我的做法是將rs和rd做and的操作，使zero不會跑掉，而這樣一來輸出結果也是對的。

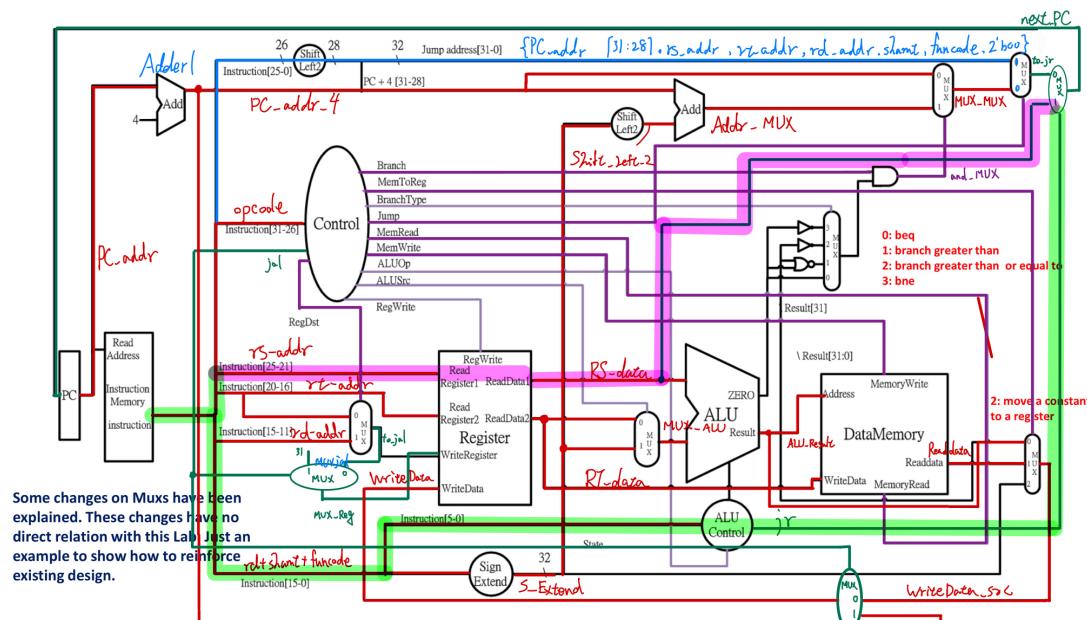
```

case(ALUOp_i)
  3'b000 : begin // R-type
    case(func_t_i)
      6'b00000 : begin // add
        ALUctrl_o <= 4'b0010;
        jr_i <= 0;
      end
      6'b100010 : begin // sub
        ALUctrl_o <= 4'b0110;
        jr_i <= 0;
      end
      6'b100100 : begin // and
        ALUctrl_o <= 4'b0000;
        jr_i <= 0;
      end
      6'b100101 : begin // or
        ALUctrl_o <= 4'b0001;
        jr_i <= 0;
      end
      6'b101010 : begin // slt
        ALUctrl_o <= 4'b0111;
        jr_i <= 0;
      end
      6'b001000 : begin // jr
        jr_i <= 1;
        ALUctrl_o <= 4'b0000;
      end
    endcase
  end
  3'b010 : ALUctrl_o <= 4'b0010; // addi
  3'b011 : ALUctrl_o <= 4'b0111; // slti
  3'b100 : ALUctrl_o <= 4'b0110; // beq
  3'b111 : ALUctrl_o <= 4'b0010; // lw, sw
  default : jr_i <= 0;
endcase

```

再來是jr的控制訊號，如下圖：

當jr指令時，沿綠色螢光線得到jr控制訊號的輸出，再沿紫色螢光線將存入rs_addr裡的\$ra的值RS_data寫入next_PC，達成jr指令的操作。



```

PC =      236
Data Memory =      0,      0,      0,      0,      0,
          0,      0,      0,      0,      0,
Data Memory =      0,      0,      0,      0,      0,
          0,      0,      0,      0,      0,
Data Memory =      0,      0,      0,      0,      0,
          68,      2,      1,      68
Data Memory =      2,      1,      68,      4,
          3,      16,      0,      0
Registers
R0 =      0, R1 =      0, R2 =      5, R3 =      0,
R4 =      0, R5 =      0, R6 =      0, R7 =      0
R8 =      0, R9 =      1, 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 =     16

```

Result:

以上是執行結果，不知道PC跟別人不一樣會不會出事（？

Summary:

這次作業難度明顯高出許多，有了更多需要自己動腦設計的指令，還有更多奇怪的bug等著我們去抵，最幹的bug應該是在看到r2的結果是5後開心地去寫了報告，實作概念全部寫完後，結果在拍執行結果時發現PC是0，這大概比出門忘記戴口罩還尷尬，然後拿一條一條線對著波形圖看後才發現問題出在ALUCtrl，當我執行jump指令時他並沒有把jr的值照著default設成0，所以就乾脆全部的case都用begin包起來給ALUCtrl和jr賦值，最後才終於有正常的數字。