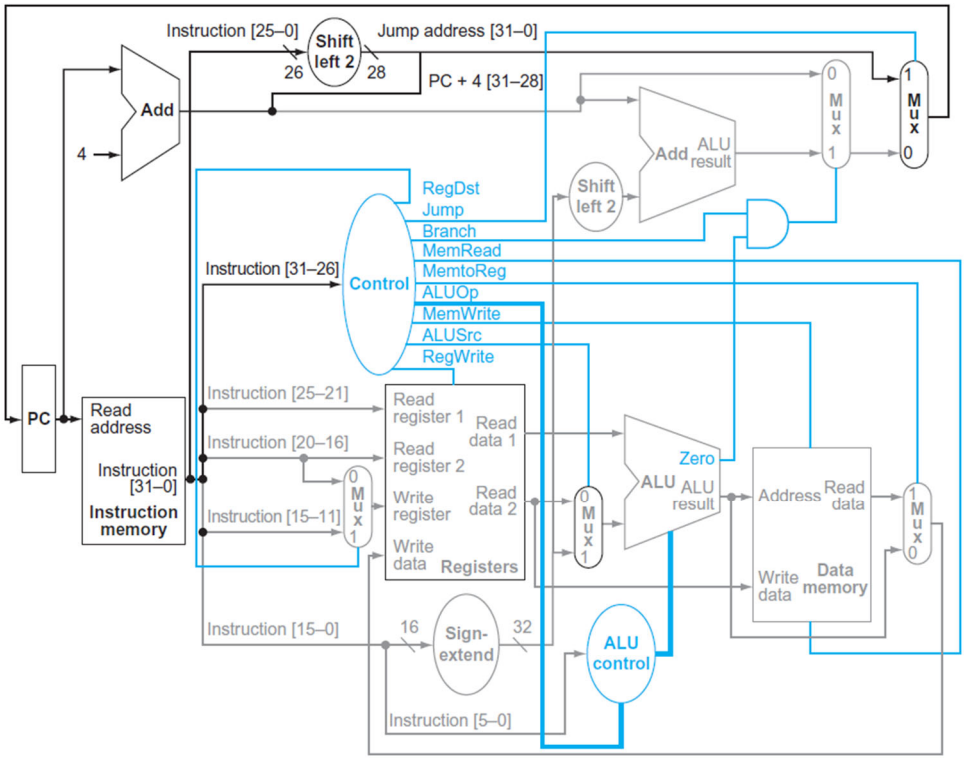
0810740 張又仁

Computer Organization

Architecture diagrams:

我主要是參考了課本的這個架構:

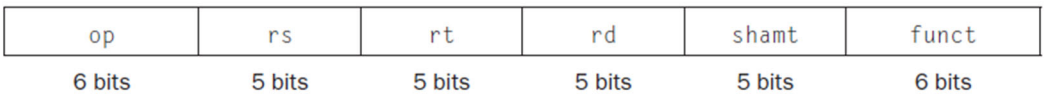


以上的架構是考慮 jump, load, store, jr 指令的架構，但是 jal 的時候，卻無法正確把資料存在 register31，因為 jal 的指令使用的是 I-type 並沒有包含存到 register31 的 bits，因此設計架構上，就會需要額外的 Mux 來放進去 write register，因此最後就會像下圖手畫圖一樣，instruction[20-16]跟 instruction[15-11]先根據是不是 R-type 來選出來，接著這個 MUX 的結果再與 regsister31 的地址一起放到 MUX 的 0 和 1，假如 Jal=1，就會選擇 register31 的地址當作 write reg!其餘的部分都只用上圖架構就可以完成這次作業要求的 Hw2 指令+jal+jr+sw+lw+jump 的要求!

我們可以知道這一次我們實踐的包括 5 個 Stage，分別是 IF，ID，EX，MEM，WB，與上次不一樣的是這次要多考慮 memory，主要是因為 lw，sw 會用到 memory。

跟上次結報一樣，考慮整個跑的流程是，PC 把要被 Decoder 的地址傳出，傳出的值我設為 pc_out_o，接著這個值會跑去兩個地方，一個是去 decoder 被解析，一個是去 PC_source 去得到下一個指令的 PC 位置，decoder 的話，一開始會先

從地址抓出 32bits 的指令，接著把指令根據課本與目前作業用到的指令來看，可以粗略區分成 R-type:



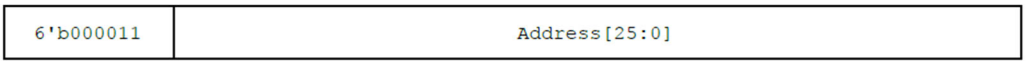
與 I-type:

這兩個 Type 最大的區別就是，R-type 處理的指令，會用到三個 Register，並且有 op 跟 funct 同時與 ALU_control 做溝通，相反的，I-type 處理的指令只有兩個 Register 以及一個 16bits 的常數，ALU 的溝通只能依靠 op 的部分。抓好指令後，Decoder 要做得就是分析這個指令並且輸出給 RegDst，RegWrite，branch，ALUop，ALUsrc，Jump，MemRead，MemtoReg，MemWrite，Jal 這幾個 mux 需要的控制信號以及 Rtype 訊號，用來告訴 ALU_Ctrl 現在吃到的是不是屬於 R-type，個別來分析，因為上次結報已經分析了 HW2 擁有的指令，因此這次僅僅針對新增的 sw，lw，jump，jal，jr 這五個指令來分析，Regdst 主要是看是不是 R-type，這五個指令來看的話，只有 jr 是 R-type。RegWrite 則是要看這個指令需不需要最後 WB 的 stage 來把值存回某個 register，lw 跟 jal 都需要，但是要特別注意的是 jal 跟 jump 指令是像以下的形式(J-type):

i. jal (jump and link)

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

Reg[31] ← PC+4
PC ← { PC[31:28], address<<2 }



後面地址是存跑到程式哪裡的位置，因此並沒有辦法直接透過指令得到儲存的位置，故這時候就會需要我上面手畫圖多出來的 Jal Mux 來讓 Decoder 發訊息告訴 Jal Mux 看到 jal 指令時就要把 write register 設為 5’b11111，也就是 resister31，同時把 write back 的 data 設成 PC+4(因為 PC 指令已執行完)而 Jal 只有在這時候會=1!。而 ALUOP 則是 Decoder 用來告訴 ALU_CRL 這是什麼指令，並且委由 ALU_CRL 傳送實際 ALU 要做的指令，詳細的部分會在下面配合 coding 來解釋，最後 ALUsrc 則是告訴 ALU 除了吃 Data1 以外，Data2 的來源如果是來自最後 16 位則輸出 1，否則如果是來自 Reg2 就輸出 0，sw，lw 屬於 I-type，因此都會用到最後 16 位。MemRead 如果是 1 表示要讀取 Memory，也就是指令 lw。MemtoReg，如果是 1，表示 wb 寫入的資料來源是 Memory，一樣是 lw。MemWrite

如果是 1，表示 Regdata2 要寫入 memory，也就是指令 sw。另外，因為我們現在的實驗已經有 3 種 type 了，而 ALUOP 雖然負責處理不是 R-type 但會用到 ALU 的指令，但是 Jump，jal 等也都不會用到 ALUOP，因此為了區別這樣的指令與 jr 的不同(jr 必須讀出 reg 資料放回 PC_sorce 的電路裡)，我設計 Decoder 同時也要輸出 Rtype_o 訊號，好讓 ALU_Ctrl 能夠分辨得出 jr。

Decoder 分析好後，接著 RF 就會根據讀到的 Register 地址來輸出這個地址的值，以及在最後處理寫回 Register 的部分。SE 則是把最後 16 位的數值變成 32 位，要注意的是，如果第 16 位 sign bit 是 0，表示這個數是正數，因此左半邊就加 16 個 0 即可，然而如果第 16 位 sign bit 是 1，表示這個數是負數，負數的話往左邊擴展的 bits 就要全部填入 1。

ALUCrtl 得到最後 6 位的 funct 及 ALUOP 以後就要告訴 ALU 要做什麼事，以這個實驗來看的話，因為 R-type 都會有 funct 對應的值，所以 ALUCrtl 處理 R-type 可以直接看 funct 的值來做判斷，相反的，I-type 因為最後十六位要看成一整個數值，因此 I-type 的 opcode 就會經由 Decoder 變成對應的 ALUOP 值，J-type 同理，因為 R-type 指令有對應的 ALU 指令，所以我們只看這次新增的 sw，lw，jump，jal，sw 跟 lw 因為有用到地址的加減，因此 ALU 要做的是 add➔表示 ALU_up 輸出的結果會讓 ALU 執行 add，而 jump 跟 jal 因為不會用到 Reg 及 Mem 的 ALU，因此輸出是 0。指令 Jr 比較特別的是，因為是 R-type 且又長成底下的形式:

ii. jr (jump register)

In MIPS, you can use

jr r31

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

PC ← Reg[Rs]

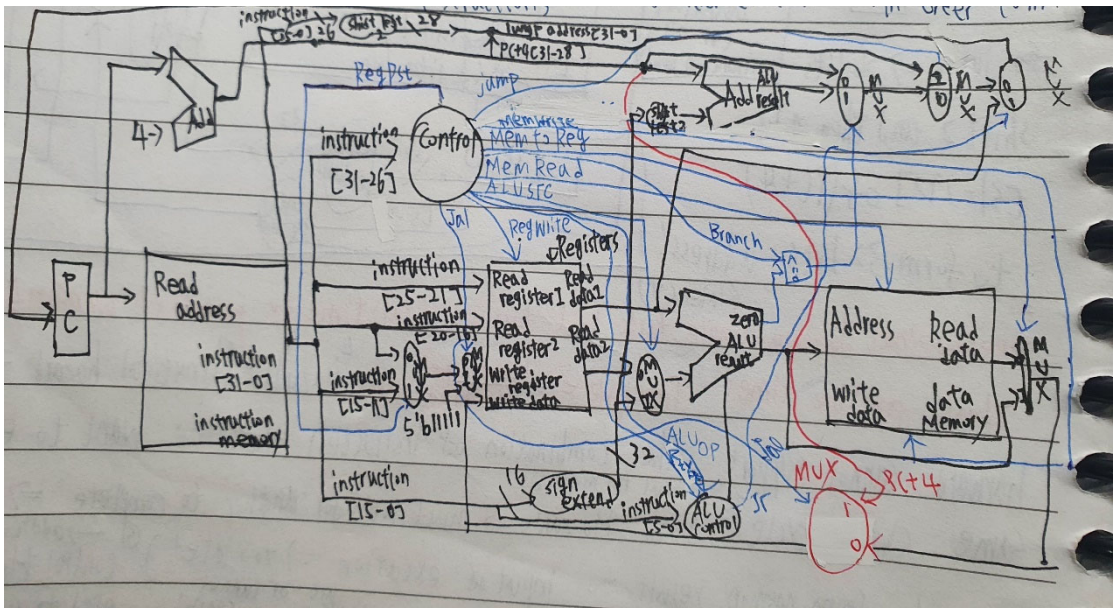
6'b000000	Rs[25:21]	0	0	0	6'b001000

前面的 Rs 是指跳到所儲存的地址的 register，因此會用到 Reg 讀取資料的部分，但是 ALU 輸出的結果卻會被寫入 Register0，因此為了簡化設計，遇到指令 Jr 時，ALU_ctrl 會輸出 and 的訊號給 ALU，這樣的會輸出結果就會是 0，而不會導致 register0 被改寫的問題!同時遇到 Jr 時，會產生出 Jr_o 給 PC_sorce 的最後一個 MUX，讓 PC 的來源選擇 Rs 讀取出來的資料!

最後 ALU 輸出的結果再跑去 Mem，如果有用到的話就讀或寫資料進去，接著出來的資料會再經過兩個 MUX，一個是針對資料來源是 ALU 還是 MEM，一個是針對資料來源是前面 MUX 的結果還是 PC+4，最後才回去 RF，如果有要寫入站存器就寫入，沒有的話，這一整個 single cycle 就結束 Reg 及 MEM 的運算了。

接著再回去討論 PC 輸出往上跑去 adder 的地方，第一個碰到的 adder 是輸出 PC+4 的加法器，就是當沒有用到 jump 指令的時候，下一個輸入指令的地址就是 PC+4，而這個 PC+4 會在往下碰到一個 adder，把最後 16 位經過擴展成 32 位的 offset

相加，如果有確定要使用 beq 而且兩值經過 sub 結果為 0，就會選擇 PC+4+offset，反之以任何一項不滿足，就只會使用 PC+4 當作下一個指令的地址。同時 PC+4 的結果也會遇到 jump 後 26 位擴展成 28 位的資料，把 PC+4 取最前面[31-28]跟這 28 位資料合併成 jump 要跳的地址，並且把這個地址跟 beqMUx 的結果做一個 MUX，如果是 1 的話就輸出 jump 地址，接著會遇到最後一個地址 MUX，一個是 Rs 讀出來的資料，一個是剛剛 jumpMux 的結果，如果指令是 jr 就會吃 rs 的資料，如果不是就會把 jumpMux 的結果輸出給下個 PC。
以上大致就是一個指令跑在這一整個 CPU 的過程。



（簡而言之，比上面參考的架構多了 Rtype，Jal，Jr 的訊號，以及 writebackData 多一個 MUX for jal，PC_sorce 多一個 MUX for jr，writeReg 多一個 MUX for Jal）

Hardware module analysis:

1. PC 與上一次實驗所用的相同
2. IM 與上一次實驗所用的相同
3. Decoder:

與上次的差異在於多了 Jump，MemRead，MemtoReg，MemWrite，Jal，Rtype 的訊號，個別信號的意義在上面分析裡也都提及了。
Code:

```
module Decoder(
    instr_op_i,
    RegWrite_o,
    ALU_op_o,
    ALUSrc_o,
    RegDst_o,
```

```
    Branch_o,
    Jump_o,
     MemRead_o,
        MemWrite_o,
        MemtoReg_o,
        Jal_o,
        Rtype_o
    );

//I/O ports
input  [6-1:0] instr_op_i;

output         RegWrite_o;
output [3-1:0] ALU_op_o;
output         ALUSrc_o;
output         RegDst_o;
output         Branch_o;
output         Jump_o;
output         MemRead_o;
output         MemWrite_o;
output         MemtoReg_o;
output         Jal_o;
output         Rtype_o;
//Internal Signals


//Parameter

wire           RegWrite_o;
wire   [3-1:0] ALU_op_o;
wire           ALUSrc_o;
wire           RegDst_o;
wire           Branch_o;

wire rtype;
wire beq;
wire addi;
wire slti;
```

```
wire jump;    // 000010
wire lw;      // 100011
wire sw;      // 101011
wire jal;     // 000011


//Main function

//process ALUop field
assign rtype = (instr_op_i==0);
assign beq   = (instr_op_i==4);
assign addi  = (instr_op_i==8);
assign slti =  (instr_op_i==10);
assign jump  = (instr_op_i==2);
assign lw    = (instr_op_i==35);
assign sw    = (instr_op_i==43);
assign jal   = (instr_op_i==3);
//process output signal
assign RegWrite_o = (((rtype | addi )| (slti|lw))|jal);
assign ALUSrc_o   = ((addi|lw) | (slti|sw) );//1 to use original 16bits
assign RegDst_o   = rtype;//1 for rd
assign Branch_o   = beq;
assign Jump_o     = (jump | jal);
assign MemRead_o  = lw;
assign MemWrite_o = sw;
assign MemtoReg_o = lw;
assign Jal_o      = jal;
assign Rtype_o    = rtype;

//only use when there is another command than R-type
//by book==> rtype-->100  beq-->001  and addi subi doesn't influence the
add or sub operation but we still only can use ALU_op_o as the command
to alu
//so addi-->ALU need add-->0010-->010
//so slti-->ALU need slt-->0111-->111
//so beq-->ALU need sub-->0110-->110
//so lw/sw-->ALU need add-->0010-->010
assign ALU_op_o[2] = (beq | slti );
```

```
assign ALU_op_o[1] = (((beq | addi )| slti)|(lw|sw));
assign ALU_op_o[0] = slti;
endmodule
```

Features:

與上一次實驗一樣，先把指令分成是 R-type 跟不是 R-type，接著處理輸出信號，而 ALU 與上次一樣用的是 3 個 bits 版本，也就是把 ALU 要吃的 4 個 bits 擷取最後三個出來當 ALUOP 的結果。Lw 跟 sw 會用到 add，因此 0010 的後三位得到 010。而 jump，jal 因為不需要使用 ALU 也不會有寫入 MEM 跟 REG 的問題，因此不需要管！

R-type，Jump，Jal，Jr:000

Beq:110

Addi:010

Slti:111

Lw:010

Sw:010

4. RF 與上一次實驗所用的相同

5. SE 與上一次實驗所用的相同

6. ALU_ctrl:

與上一次的差異在於多輸出了 JR 訊息給 PC_sorce 電路的最後一個 MUX，以及多輸入了 Rtype_o 來辨別 jr 與其他 jump 的不同！

Code:

```
module ALU_Ctrl(
          Rtype_i,
          funct_i,
          ALUOp_i,
          ALUCtrl_o,
          Jr_o
          );

//I/O ports
input      Rtype_i;
input      [6-1:0] funct_i;
input      [3-1:0] ALUOp_i;

output     [4-1:0] ALUCtrl_o;
output             Jr_o;
//Internal Signals
reg        [4-1:0] ALUCtrl_o;
```

```
//Parameter
assign     Jr_o=(Rtype_i==1&&funct_i==8);



//Select exact operation
always @(funct_i,ALUOp_i) begin
        if(Rtype_i==1)
          case(funct_i)
              32: ALUCtrl_o <= 4'b0010;//add
              34: ALUCtrl_o <= 4'b0110;//sub
            36,8: ALUCtrl_o <= 4'b0000;//and / jr
              37: ALUCtrl_o <= 4'b0001;//or
              42: ALUCtrl_o <= 4'b0111;//slt
          endcase
        else
        begin
            ALUCtrl_o[3]   <= 0;
            ALUCtrl_o[2:0] <= ALUOp_i[2:0];//for lw/sw/slti/beq/addi
        end
end
endmodule
```

Features:

可以看到現在只有在 Rtype_i 是 1 的時候才去分析 funct_i，同時我把 jr 使用的 ALU 行為設成 and，主要是因為 jr 指令被當作 R-type，因此可以被寫入，但 jr 指令我們並不希望他被寫入，而剛好 jr 固定的格式就是寫入的 register 是 0(rd，rt 的欄位)，因此可以利用這個特性讓 jr 讀出的資料在 ALU 裡面跟 0 去做 and，這樣得到的結果還是一樣是 0!就可以解決 jr 這個不需要存 register 的特別 R-type 了。另外在這個元件也會同時發出 Jr_o 的訊號，主要是因為 Jr 指令是 R-type，而 Decoder 沒有吃 funct_i 的 bits，因此我們無法在 Decoder 就找到是不是 Jr,而這個訊號會迫使 rs 的讀出資料變成 PC_sorce 的最後結果，也就達成 jr 指令的進行了!

7. ALU 與上一次實驗所用的相同
8. MUX 與上一次實驗所用的相同
9. Adder 與上一次實驗所用的相同
10. Shifter 與上一次實驗所用的相同

11.Memory

Memory 主要負責長期儲存資料用的。

Code:

```
module Data_Memory
(
    clk_i,
    addr_i,
    data_i,
    MemRead_i,
    MemWrite_i,
    data_o
);

// Interface
input               clk_i;
input   [31:0]      addr_i;
input   [31:0]      data_i;
input               MemRead_i;
input               MemWrite_i;
output  [31:0]      data_o;

// Signals
reg     [31:0]      data_o;

// Memory
reg     [7:0]       Mem             [0:127];    // address: 0x00~0x80
integer             i;

// For Testbench to debug
wire    [31:0]      memory          [0:31];
assign  memory[0] = {Mem[3], Mem[2], Mem[1], Mem[0]};
assign  memory[1] = {Mem[7], Mem[6], Mem[5], Mem[4]};
assign  memory[2] = {Mem[11], Mem[10], Mem[9], Mem[8]};
assign  memory[3] = {Mem[15], Mem[14], Mem[13], Mem[12]};
assign  memory[4] = {Mem[19], Mem[18], Mem[17], Mem[16]};
assign  memory[5] = {Mem[23], Mem[22], Mem[21], Mem[20]};
assign  memory[6] = {Mem[27], Mem[26], Mem[25], Mem[24]};
assign  memory[7] = {Mem[31], Mem[30], Mem[29], Mem[28]};
```

```
assign  memory[8] = {Mem[35], Mem[34], Mem[33], Mem[32]};
assign  memory[9] = {Mem[39], Mem[38], Mem[37], Mem[36]};
assign  memory[10] = {Mem[43], Mem[42], Mem[41], Mem[40]};
assign  memory[11] = {Mem[47], Mem[46], Mem[45], Mem[44]};
assign  memory[12] = {Mem[51], Mem[50], Mem[49], Mem[48]};
assign  memory[13] = {Mem[55], Mem[54], Mem[53], Mem[52]};
assign  memory[14] = {Mem[59], Mem[58], Mem[57], Mem[56]};
assign  memory[15] = {Mem[63], Mem[62], Mem[61], Mem[60]};
assign  memory[16] = {Mem[67], Mem[66], Mem[65], Mem[64]};
assign  memory[17] = {Mem[71], Mem[70], Mem[69], Mem[68]};
assign  memory[18] = {Mem[75], Mem[74], Mem[73], Mem[72]};
assign  memory[19] = {Mem[79], Mem[78], Mem[77], Mem[76]};
assign  memory[20] = {Mem[83], Mem[82], Mem[81], Mem[80]};
assign  memory[21] = {Mem[87], Mem[86], Mem[85], Mem[84]};
assign  memory[22] = {Mem[91], Mem[90], Mem[89], Mem[88]};
assign  memory[23] = {Mem[95], Mem[94], Mem[93], Mem[92]};
assign  memory[24] = {Mem[99], Mem[98], Mem[97], Mem[96]};
assign  memory[25] = {Mem[103], Mem[102], Mem[101], Mem[100]};
assign  memory[26] = {Mem[107], Mem[106], Mem[105], Mem[104]};
assign  memory[27] = {Mem[111], Mem[110], Mem[109], Mem[108]};
assign  memory[28] = {Mem[115], Mem[114], Mem[113], Mem[112]};
assign  memory[29] = {Mem[119], Mem[118], Mem[117], Mem[116]};
assign  memory[30] = {Mem[123], Mem[122], Mem[121], Mem[120]};
assign  memory[31] = {Mem[127], Mem[126], Mem[125], Mem[124]};

initial begin
    for(i=0; i<128; i=i+1)
        Mem[i] = 8'b0;

end

always@(posedge clk_i) begin
    if(MemWrite_i) begin
        Mem[addr_i+3] <= data_i[31:24];
        Mem[addr_i+2] <= data_i[23:16];
        Mem[addr_i+1] <= data_i[15:8];
        Mem[addr_i]   <= data_i[7:0];
    end
```

```
end

always@(addr_i or MemRead_i) begin
    if(MemRead_i)
        data_o = {Mem[addr_i+3], Mem[addr_i+2], Mem[addr_i+1],
Mem[addr_i]};
end

endmodule
```

Features:

可以從上面的 code 發現我們這個實驗的 Memory 總共有 32 個，一個 Memory 有 4 個 bytes，然後 addr_i 主要是 1 個 byte 的單位。

12.CPU

與上一次不一樣的是，這次的架構與我手繪圖畫的一樣，也就是我們多考慮了 Memory 以及根據多考慮的 sw，lw，jump，jal，jr 這五個指令而多了其他的 MUX 及內部訊號。

Code:

```
module Simple_Single_CPU(
        clk_i,
        rst_i
        );

//I/O port
input       clk_i;
input       rst_i;



//Internal Signles

// wire for PC
wire [31:0] pc_in_i;
wire [31:0] pc_out_o;//for adder1 and IM

// wire for adder 1
wire [31:0] sum_o_add1;

// wire for IM
```

```
wire [31:0] instr_o;

// wire for MUX_write_reg
wire [4:0]  data_o_write_reg;

// wire for RF
wire [31:0] RSdata_o;
wire [31:0] RTdata_o;

// wire for ALU
wire [31:0] result_o;
wire        zero_o;

// wire for SE
wire [31:0] data_o_SE;

// wire for Mux__alu_src
wire [31:0] Src_data_o;

// wire for Decoder
wire        RegWrite_o;
wire [2:0]  ALU_op_o;
wire        ALUSrc_o;
wire        RegDst_o;
wire        Branch_o;
wire        Jump_o;
wire        MemRead_o;
wire        MemWrite_o;
wire        MemtoReg_o;
wire        Jal_o;
wire        Rtype_o;
// wire for ALUCtrl
wire [3:0]  ALUCtrl_o;
wire        Jr_o;
//Internal Signles
wire        and_out;
and AND(and_out,Branch_o,zero_o);
```

```
// wire for shift-left
wire [31:0] data_o_shift;

// wire for adder2
wire [31:0] sum_o_add2;



// wire for branch result
wire [31:0] mux_branch_result;

// wire for new added data memory
wire [31:0] result_from_mem;

// wire for write back data
wire [31:0] write_back_data;

// wire for jump
wire [31:0] instr_shl2;
//Greate componentes
ProgramCounter PC(
        .clk_i(clk_i),
        .rst_i (rst_i),
        .pc_in_i(pc_in_i) ,
        .pc_out_o(pc_out_o)
);

Adder Adder1(//Pc+4
        .src1_i(32'd4),
        .src2_i(pc_out_o),
        .sum_o(sum_o_add1)
);

Instr_Memory IM(
        .pc_addr_i(pc_out_o),
        .instr_o(instr_o)
);
```

```
MUX_2to1 #(.size(5)) Mux_Write_Reg(
        .data0_i(instr_o[20:16]),
        .data1_i(instr_o[15:11]),
        .select_i(RegDst_o),
        .data_o(data_o_write_reg)
);

wire [4:0]  final_write_reg;
wire [31:0] final_write_data;
MUX_2to1 #(.size(5)) Jal_Write_Reg(
        // Jal
        .data0_i(data_o_write_reg),
        .data1_i(5'b11111),//reg 31==>store PC+4 to reg 31
        .select_i(Jal_o),
        .data_o(final_write_reg)
);
MUX_2to1 #(.size(32)) Jal_Write_Data(
        // Jal
        .data0_i(write_back_data),//after memory processing
        .data1_i(sum_o_add1),//Pc+4 ==>to store in reg31
        .select_i(Jal_o),
        .data_o(final_write_data)
);

Reg_File Registers(
        .clk_i(clk_i),
        .rst_i(rst_i),
        .RSaddr_i(instr_o[25:21]) ,
        .RTaddr_i(instr_o[20:16]) ,
        .RDaddr_i(final_write_reg),
        .RDdata_i(final_write_data),
        .RegWrite_i (RegWrite_o),
        .RSdata_o(RSdata_o),
        .RTdata_o(RTdata_o)
);

Decoder Decoder(
        .instr_op_i(instr_o[31:26]),
```

```
        .RegWrite_o(RegWrite_o),
        .ALU_op_o(ALU_op_o),
        .ALUSrc_o(ALUSrc_o),
        .RegDst_o(RegDst_o),
        .Branch_o(Branch_o),
        .Jump_o(Jump_o),
        .MemRead_o(MemRead_o),
        .MemWrite_o(MemWrite_o),
        .MemtoReg_o(MemtoReg_o),
        .Jal_o(Jal_o),
        .Rtype_o(Rtype_o)
);

ALU_Ctrl AC(
        .Rtype_i(Rtype_o),
        .funct_i(instr_o[5:0]),
        .ALUOp_i(ALU_op_o),
        .ALUCtrl_o(ALUCtrl_o),
        .Jr_o(Jr_o)
);

Sign_Extend SE(
        .data_i(instr_o[15:0]),
        .data_o(data_o_SE)
);

MUX_2to1 #(.size(32)) Mux_ALUSrc(
        .data0_i(RTdata_o),
        .data1_i(data_o_SE),//address for
        .select_i(ALUSrc_o),
        .data_o(Src_data_o)
);

ALU ALU(

.clk(clk_i),
.rst_n(rst_i),
        .src1(RSdata_o),
```

```
        .src2(Src_data_o),
        .ALU_control(ALUCtrl_o),
        .result(result_o),
        .zero(zero_o)
);

Adder Adder2(
        .src1_i(sum_o_add1),
        .src2_i(data_o_shift),
        .sum_o(sum_o_add2)  //result of address for using data from
instru original 16 bits
);

Shift_Left_Two_32 Shifter(
        .data_i(data_o_SE),
        .data_o(data_o_shift)//address*4
);

MUX_2to1 #(.size(32)) Mux_PC_Source(
        .data0_i(sum_o_add1),
        .data1_i(sum_o_add2),
        .select_i(and_out),
        .data_o(mux_branch_result)
);

// MEM
Data_Memory Data_Memory(
        .clk_i(clk_i),
        .addr_i(result_o),
        .data_i(RTdata_o),
        .MemRead_i(MemRead_o),
        .MemWrite_i(MemWrite_o),
        .data_o(result_from_mem)//memory data
);

MUX_2to1 #(.size(32)) Mux_Write_Back_Data(
        .data0_i(result_o),
        .data1_i(result_from_mem),
```

```
        .select_i(MemtoReg_o),
        .data_o(write_back_data)
);

// Jump
Shift_Left_Two_32 Shifter_jump(
        .data_i(instr_o),
        .data_o(instr_shl2)
);

wire [31:0] jump_pc;
MUX_2to1 #(.size(32)) Mux_PC_Jump(
        .data0_i(mux_branch_result),
        .data1_i({sum_o_add1[31:28],instr_shl2[27:0]}),
        .select_i(Jump_o),
        .data_o(jump_pc)
);

// Jal
MUX_2to1 #(.size(32)) Mux_PC_Jal(
        .data0_i(jump_pc),
        .data1_i(RSdata_o),
        .select_i(Jr_o),//jr
        .data_o(pc_in_i)
);
endmodule
```

Features:

可以發現除了 Memory 是多的元件以外，其他的原件都是很早就定義好的，頂多就是 decoder 跟訊號逮屬的元件稍做一點修改而已，整體 CPU 架構來看與上一次實驗差異不大!

Experiment result:

第一個實驗:

```
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
```

這個 test 會做的事情如以下:

```
addi    r1,r0,1                 r1=1
addi    r2,r0,2                 r2=2
addi    r3,r0,3                 r3=3
addi    r4,r0,4                 r4=4
addi    r5,r0,5                 r5=5
jump    j
---若jump對，以下兩個addi將不會執行
addi    r1,r0,31                        r1=31
addi    r2,r0,32                        r2=32
---
j:
sw      r1,0(r0)                m0=1
sw      r2,4(r0)                m1=2
lw      r6,0(r0)                r6=1
lw      r7,0(r4)                r7=2
add     r8,r1,r3                r8=4
lw      r9,4(r0)                r9=2
```

```
final:
Register═══════════════════════════════════════════
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

Memory═══════════════════════════════════════════
m0=         1, m1=         2, 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
r16=        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
```

可以發現與我跑出來的最後結果一致。

第二個實驗:

```
PC =        120
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,         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
PC =        124
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,         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
```

這個 test 會執行以下的程式:

```
add r0,r0,r0
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
addi t1,zero,1
jal fib                 <-JAL:當fib function結束後PC會跳到j final
j final

fib:
addi sp,sp,-12          //stack pointer -12
sw ra,0(sp)             //以下三道sw將reg存入memory中
sw s0,4(sp)
sw s1,8(sp)
add s0,a0,zero
beq s0,zero,re1         //判斷是否f(0)
beq s0,t1,re1           //判斷是否f(1)
addi a0,s0,-1
jal fib
add s1,zero,v0
addi a0,s0,-2
jal fib
add v0,v0,s1

exitfib:
lw ra,0(sp)
lw s0,4(sp)
lw s1,8(sp)
addi sp,sp,12
jr ra                   //function call結束

re1:
addi v0,zero,1
j exitfib

final:
nop



------
run完r2=5
```

根據課本:

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

雖然助教只有說看 register2 是不是 5 即可，但其他的 REG 跟 MEM 一樣可以用簡單的分析看出來，我們可以發現 t1 這個 register 最後結果一定會是 1(r9=1)，因為只有出現在最上面，而 R31 最後一定是存主程式 jal 出去的下個指令也就是 16，R29 是 stack pointer 並且指向的是最後的 MEM，故最後正確結束程式時還是要變回 128，大致這樣判斷，因此我認為我在這個 test 應該是沒有錯的。

Problems you met and solutions:

Q:在這次的實驗中，我認為遇到的問題比前面都少很多，基本上都蠻順利的，比較花時間想的就是 jr 的問題，因為 jr 不需要改寫任何 REG 跟 MEM，因此如果 ALU_Ctrl 無法分辨它們的話，只要 jal 或 jump 有用到結尾 6 個 bits 剛好是 8 的指令，就會被誤判是 jr。

A:因此我就讓 Decoder 多輸出一個 Rtype_o 的訊號，讓 ALU_Ctrl 可以辨識出這是 jr 或不是 jr，畢竟 jump，jal 的 ALUOP 也是 0，因此，這邊的判斷如果還是用 ALUOP==0 來判斷是不是 R-type 的話就會出現問題!

Summary:

這次的實驗，我認為相比之前的電路複雜許多，像是 jal 雖然不需要用到 REG 跟 MEM 之間的 ALU，但卻需要寫入 register31，而且還是要從上半段的 PC+4 把資料丟下來，因此我原本以為的分成 address 跟 REGMEM 區域的分類就立刻看不太見了，而 jr 也是同理，必須把 REG 讀出來的值往上面 address 區域丟入值。雖然變複雜了，但我覺得也變得有趣了，我對於 CPU 的理解比起上次沒有 MEM 的版本，感覺又多了更多，像是 lw 跟 sw 還有 stack pointer 怎麼進行等等，其實在進行 test 的過程裡，慢慢就能明白這些指令為何這麼安排以及能夠慢慢想像到如果今天多的是其他指令那我又需要多哪些元件呢?總之，這個實驗算是把上個實驗切成兩大半的電路大致又變成一個整體的電路了，覺得蠻酷的。