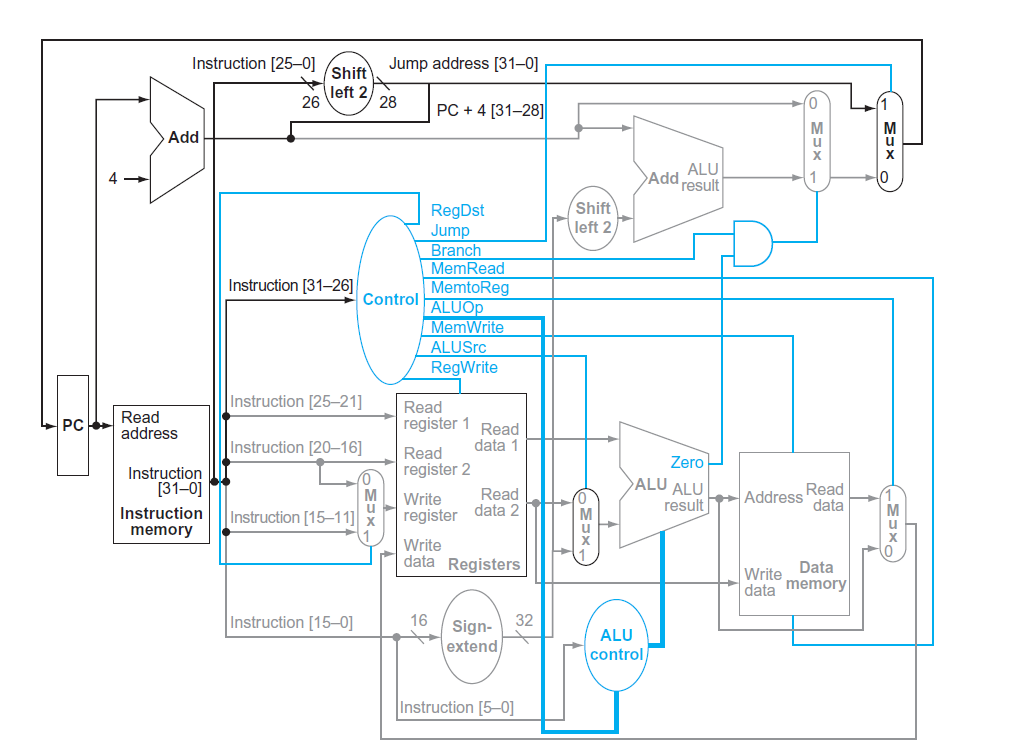
**0810740 張又仁**

**Computer Organization**

**Architecture diagrams:**

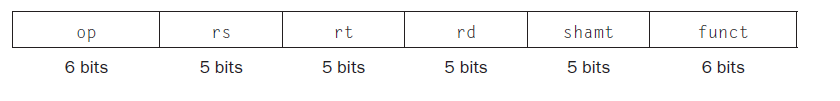
我主要是參考了課本的這個架構:



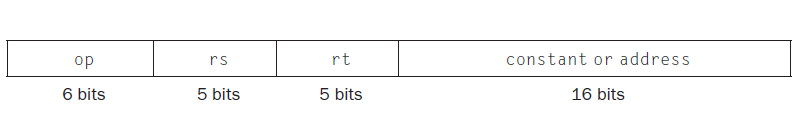
以上的架構是考慮jump,load,store,jr指令的架構，但是jal的時候，卻無法正確把資料存在register31，因為jal的指令使用的是I-type並沒有包含存到register31的bits，因此設計架構上，就會需要額外的Mux來放進去write register，因此最後就會像下圖手畫圖一樣，instruction[20-16]跟instruction[15-11]先根據是不是R-type來選出來，接著這個MUX的結果再與regsister31的地址一起放到MUX的0和1，假如Jal=1，就會選擇register31的地址當作write reg!其餘的部分都只用上圖架構就可以完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

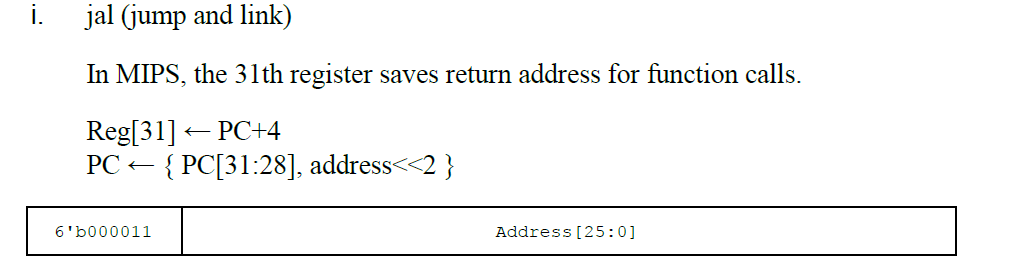
跟上次結報一樣，考慮整個跑的流程是，PC把要被Decoder的地址傳出，傳出的值我設為pc\_out\_o，接著這個值會跑去兩個地方，一個是去decoder被解析，一個是去PC\_source去得到下一個指令的PC位置，decoder的話，一開始會先從地址抓出32bits的指令，接著把指令根據課本與目前作業用到的指令來看，可以粗略區分成R-type:



與I-type:



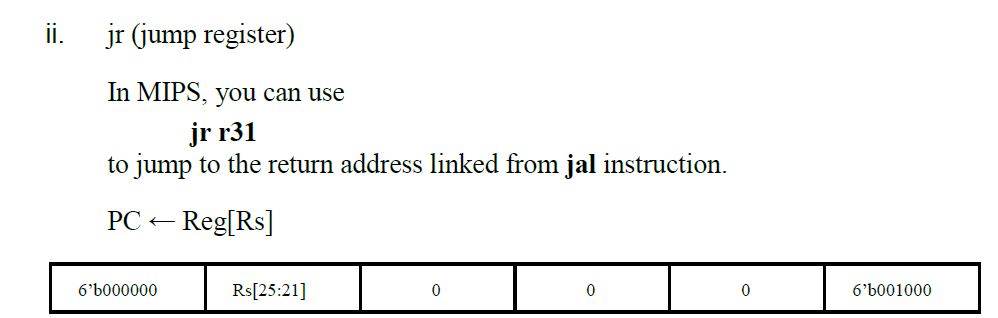
這兩個Type最大的區別就是，R-type處理的指令，會用到三個Register，並且有op跟funct同時與ALU\_control做溝通，相反的，I-type處理的指令只有兩個Register以及一個16bits的常數，ALU的溝通只能依靠op的部分。抓好指令後，Decoder要做得就是分析這個指令並且輸出給RegDst，RegWrite，branch，ALUop，ALUsrc，Jump，MemRead，MemtoReg，MemWrite，Jal這幾個mux需要的控制信號以及Rtype訊號，用來告訴ALU\_Ctrl現在吃到的是不是屬於R-type，個別來分析，因為上次結報已經分析了HW2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):



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:

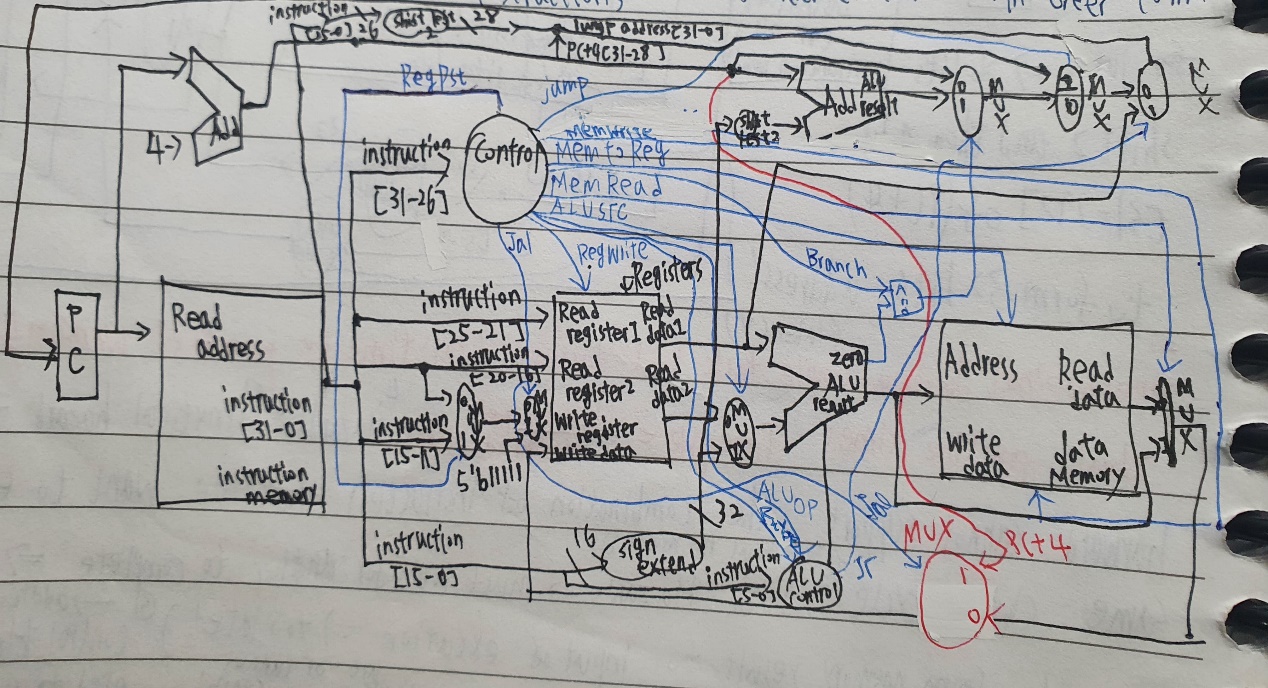


前面的Rs是指跳到所儲存的地址的register，因此會用到Reg讀取資料的部分，但是ALU輸出的結果卻會被寫入Register0，因此為了簡化設計，遇到指令Jr時，ALU\_ctrl會輸出and的訊號給ALU，這樣的會輸出結果就會是0，而不會導致register0被改寫的問題!同時遇到Jr時，會產生出Jr\_o給PC\_sorce的最後一個MUX，讓PC的來源選擇Rs讀取出來的資料!

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

1. RF 與上一次實驗所用的相同
2. SE 與上一次實驗所用的相同
3. 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!

1. ALU 與上一次實驗所用的相同
2. MUX與上一次實驗所用的相同
3. Adder與上一次實驗所用的相同
4. Shifter與上一次實驗所用的相同
5. 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。

1. 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)

);

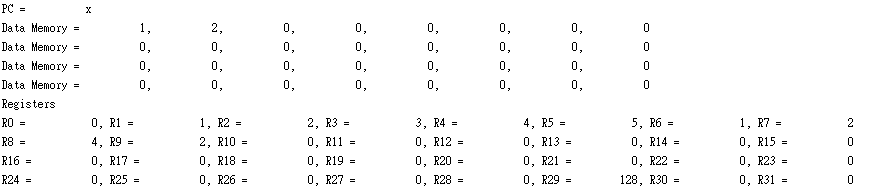
endmodule

Features:

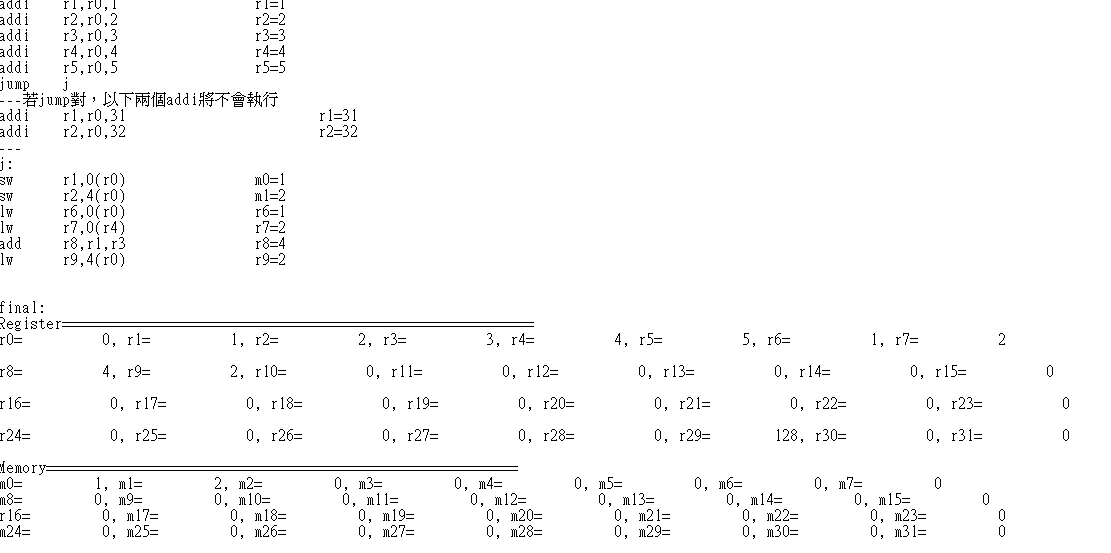
可以發現除了Memory是多的元件以外，其他的原件都是很早就定義好的，頂多就是decoder跟訊號逮屬的元件稍做一點修改而已，整體CPU架構來看與上一次實驗差異不大!

**Experiment result:**

**第一個實驗:**

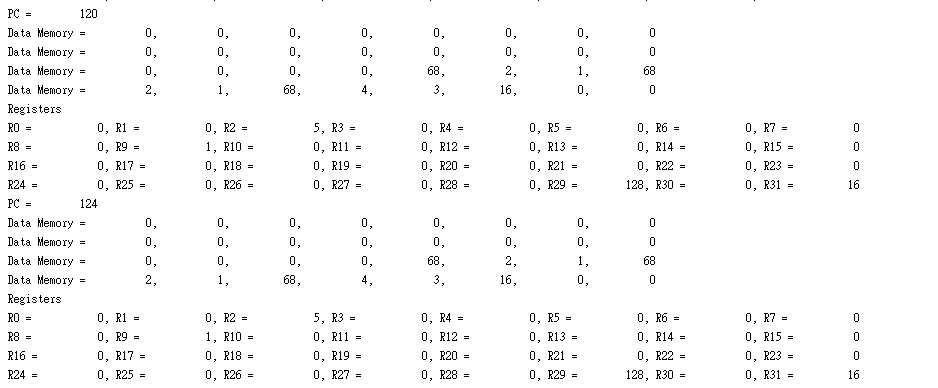
****

這個test會做的事情如以下:

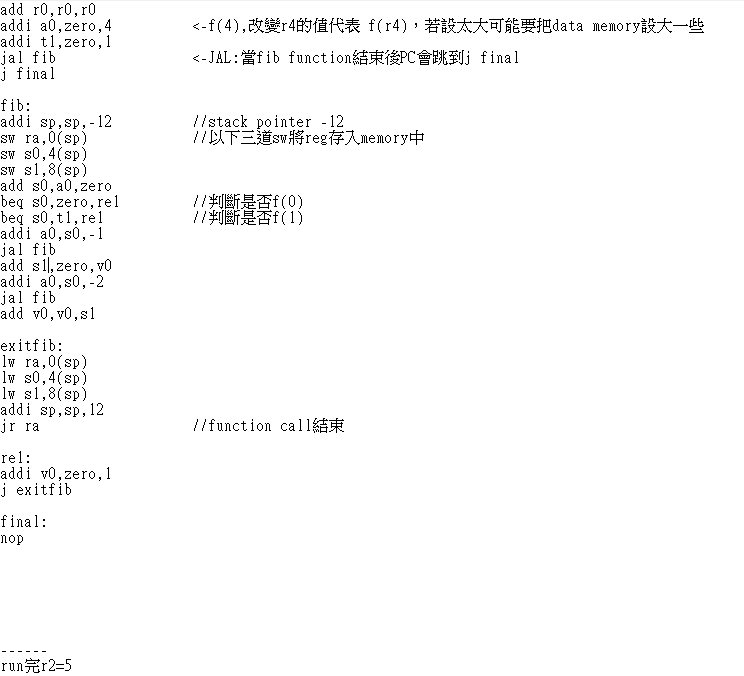
****

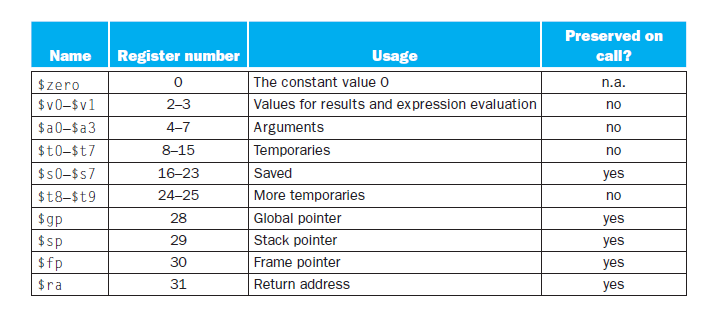
可以發現與我跑出來的最後結果一致。

**第二個實驗:**

****

這個test會執行以下的程式:



根據課本: 

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。