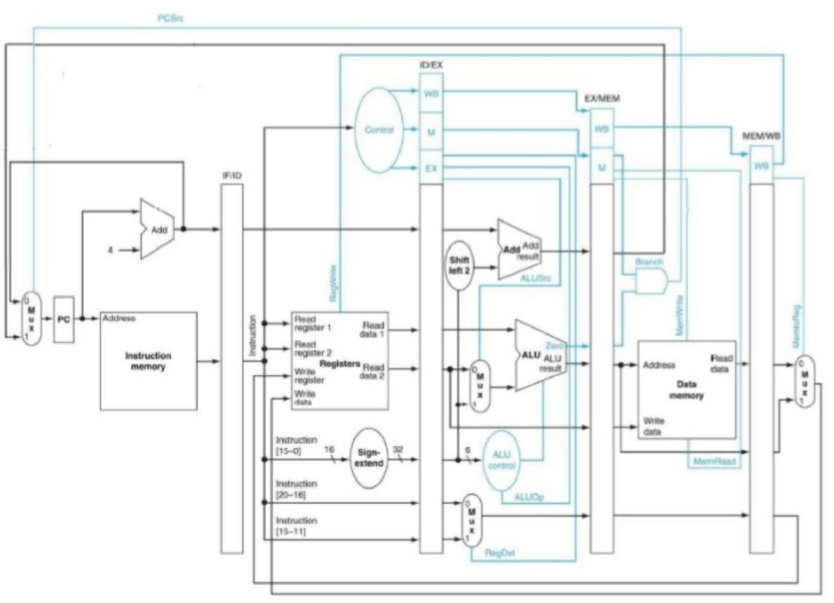
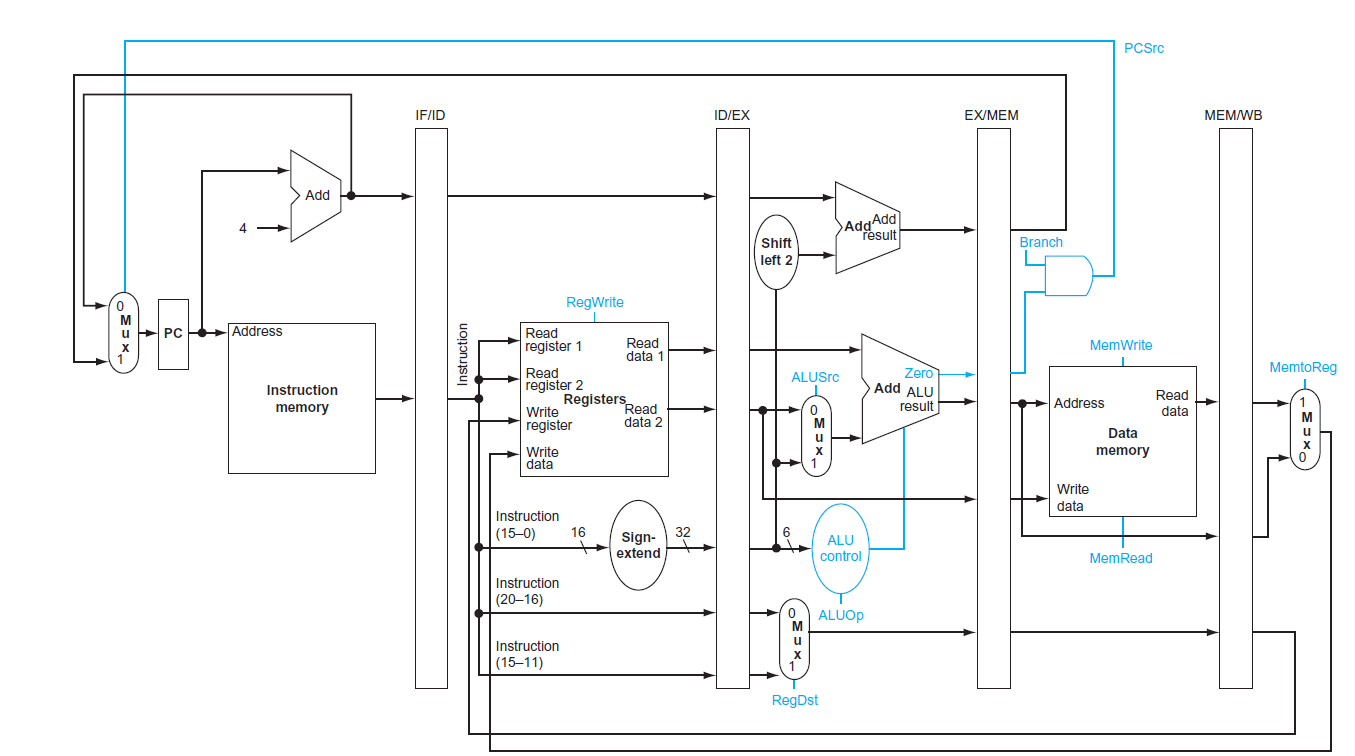
**0810740 張又仁**

**Computer Organization**

**Architecture diagrams:**

我主要是參考了spec跟課本的架構:(課本是簡易版，spec比較詳細)





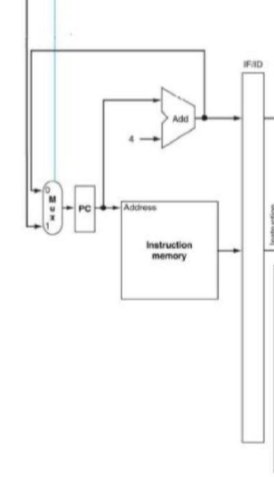
以上的架構是考慮beq,load,store,d指令的架構，但是beq成功觸發的時候，實際上還是有線會連到ID/EX及EX/MEM的register，促使他們把資料清空，畢竟beq在確認觸發前也先把下面的指令抓進來了，所以這是上面的圖沒有畫出來的部分(但因為只是多兩條訊號線過去這兩個registe，我就不再額外畫了)。

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pipe line的register，主要是因為之前實作的時候，一個指令相當於就是占用一個大clock，但是當我們把clock切細以後，可以發現我們可以讓這五個stage同時運作，而不是每次都只有一個stage在運作，這就是pipe line的概念，這樣總體處理速度就會提高，因此這次的作業就是在模擬如果這次clock相當於是被切細的clock，那我們要如何讓這五個stage能夠盡量同時一起運作呢?我們都知道電路如果沒有用Register，那麼訊號都會是趨近即時性的，用了register就可以保留上個狀態的值，把這個概念結合起來，我們就可以知道如果要讓五個stage一起運作給不同的指令，那就是放入4個register把後面四個stage的狀態保留，並且讓新的指令可以進到第一個stage，這就是這次作業模擬pipe line的方式。

**Hardware module analysis**:

跟上次結報一樣，我們一樣考慮整個跑的流程。

-----------------------------------IF stage--------------------------



一開始的IF stage，總共有四個主要的component，一個是PC把現在要處理的指令地址丟進來，一個是PC前面的MUX去選擇下個PC的地址，一個是把PC+4計算出來的adder，最後一個則是IM，把指令抓出來的地方。

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IM抓出指令，一個是去PC\_source去得到PC+4的地址，做完這個stage以後，會先進入第一個IF/ID的暫存器，IF/ID的size因為需要乘載現在PC抓出的指令與PC+4地址的兩個值，因此大小為64bits，接著再繼續往前。

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. MUX與上一次實驗所用的相同
4. Adder與上一次實驗所用的相同
5. Pipe\_reg:

這個register與RF的架構很像，唯一不同的是這個register僅僅是作為訊號滯留的功用，因此除了輸入與輸出，這個register並沒有特別的功用。

Coding:

module Pipe\_Reg(

clk\_i,

rst\_i,

data\_i,

data\_o

);

parameter size = 0;

input clk\_i;

input rst\_i;

input [size-1:0] data\_i;

output reg [size-1:0] data\_o;

always@(posedge clk\_i) begin

if(~rst\_i)

data\_o <= 0;

else

data\_o <= data\_i;

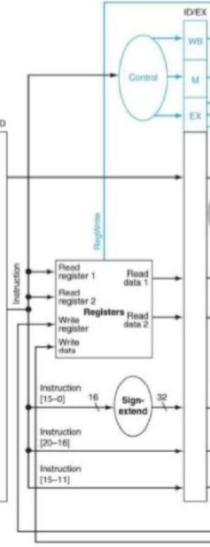
end

endmodule

Features:

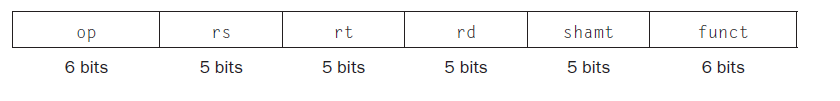
當clk處於posedge且rst沒有啟動時，就會把data\_i的值丟入data\_o，這也就達成了我們對於piple line滯留訊號的功能，因為要一個小clock從Pipe register的輸入丟到輸出，接著還要再一個小clock把Pipe register的輸出拿來做使用。

-----------------------------------ID stage--------------------------

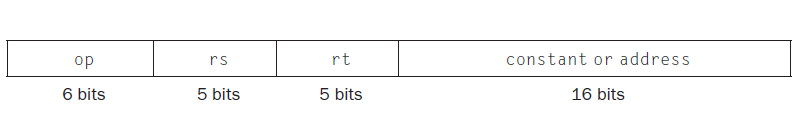


ID stage 總共有3個主要的component，一個是負責把訊號出書的decoder，一個是把最後16bits擴展成32bits的sign extend，最後則是處理register寫入讀取的RF。

decoder的話，一開始會把從地址抓出的32bits指令的[31:26]輸入，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，訊號方面前面都有分析過了，因此這次強調的是我們從decoder分析出的這幾個訊號只是一個暫時訊號，真正存到暫存器再給MUX的訊號還必須確認有沒有branch。

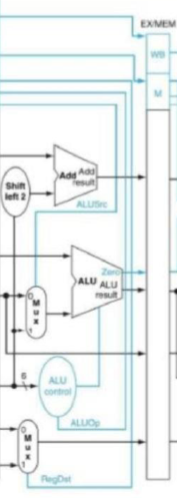
RF會根據讀到的Register地址來輸出這個地址的值，以及在最後WB satge處理寫回Register的部分。

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。

ID stage處理完以後，就會把Decoder產生的ID\_EX\_reg\_write\_i\_tmp,ID\_EX\_alu\_op\_i\_tmp,ID\_EX\_alu\_src\_i\_tmp,ID\_EX\_reg\_dst\_i\_tmp,ID\_EX\_branch\_i\_tmp,ID\_EX\_mem\_read\_i\_tmp,ID\_EX\_mem\_write\_i\_tmp,ID\_EX\_mem\_to\_reg\_i\_tmp,ID\_EX\_Rtype\_i這共11bits的資料拿去跟branch確認沒有觸發的訊號做&&，以及PC+4還有RF讀出的資料一跟資料2的data以及SE延伸出的32bits資料共128bits跟要再ALU決定資料二來源的rt,rd的address共10bits，總和149bits的資料丟入ID/EX暫存器，接著在下一個clock才繼續往後面stage執行。

1. Decoder 與上一次實驗所用的相同(雖然這次沒有用到Jal跟Jr或是jump，但並不影響設計!)
2. RF 與上一次實驗所用的相同
3. SE 與上一次實驗所用的相同
4. Pipe\_reg與上面一樣

-----------------------------------EX stage--------------------------



EX stage這邊會用到6個主要的component，一個是產生ALU指令的ALUCtrl，一個是負責計算數值結果的ALU，一個是把延伸成32bits的地址數值左移兩位Shifter，一個是把左移結果跟PC+4的結果作相加的adder(主要是為了branch)，以及兩個MUX，一個處理ALU第二個資料的來源，一個處理REG判斷是寫回rt還是rd register。

ALUCrtl藉由sign\_extend的[5:0]得到原本指令最後6位的funct及ALUOP還有Rtype訊號以後就要告訴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mult，mult是標準的Rtype指令，因此ALU對應的也會有更動是要給mult的計算，在這裡我設丟出的ALU指令是0101。

ALU原則上與之前都一樣，只是多一個要處理mult的部分，因為這邊的mult是暫時考慮成mul來使用，因此可以確保我們只需要考慮到32bits，而乘法器本身的實現其實會包含複雜的shifter跟累加還有一些判斷的MUX，為了簡化，我這邊的實現就直接使用了Verilog的’\*’來代替。

Shifter主要是因為我們處理地址的時候會為了能夠容納更多的地址而把最後兩個0省略，但在運算時還是要考慮進去，因此就要把它變回原形。

ADDer主要是為了下個stage的branch觸發後要輸出的地址而做的運算。

比較要注意的是，決定寫回哪個reg的MUX在前幾次的作業裡我都是直接放在RF的旁邊，但是當我們真的要用pipe line實現五個stage的分類時，就要特別注意因為Decoder是跟RF同一個stage，但是REG資料寫回是在WB的stage才能決定好，所以決定寫回哪個reg必須跟著最後決定什麼data寫入一起從WB跑到RF處理，另外，為了區別這兩種的差異，因此即便我們可以在ID的階段，就拿rs,rt還有剛產生的reg\_dst訊號能來決定是哪個reg要被寫回，再把這個結果跟著pipe\_reg傳到最後，我仍然認為還是按著spec的設計比較能讓我們看出這兩個差異，就沒有做額外的修改了。

EX stage完成後，一樣是把EX\_MEM\_branch\_i,EX\_MEM\_mem\_write\_i,EX\_MEM\_mem\_read\_i,EX\_MEM\_mem\_to\_reg\_i,EX\_MEM\_reg\_write\_i這幾個訊號拿去跟branch確認沒有觸發做&&，共五個bits，接著再把Alu計算結果還有是否zero以及branch地址跟當使用sw指令就會在MEM使用到的data2讀取的資料以及會在最後WB一起傳給RF的reg dst地址，共102bits，總和107bits的暫存器拿來當作這個stage在一個clock結束的結果。

1. ALU\_ctrl:

與上一次的差異在於多一個mult的判斷。

Coding:

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)

24: ALUCtrl\_o <=4'b0101;//mult

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:

因為具有連貫性，因此可以直接把上次作業的拿來多加一個給mult，其餘不變。

2. ALU:

一樣多一個mult的部分，其餘不變。

Coding:

module ALU(

clk, // system clock (input)

rst\_n, // negative reset (input)

src1, // 32 bits source 1 (input)

src2, // 32 bits source 2 (input)

ALU\_control, // 4 bits ALU control input (input)

result, // 32 bits result (output)

zero // 1 bit when the output is 0, zero must be set (output)

//cout, // 1 bit carry out (output)

// overflow // 1 bit overflow (output)

//,as

//,result1,result2,result3,result4,result5,result6

);

/\*

//I/O ports

input signed [32-1:0] src1;

input signed [32-1:0] src2;

input [4-1:0] ctrl\_i;

output [32-1:0] result\_o;

output zero\_o;

//Internal signals

reg [32-1:0] result\_o;

wire zero\_o;

\*/

input clk;

input rst\_n;

input [32-1:0] src1;

input [32-1:0] src2;

input [4-1:0] ALU\_control;

output [32-1:0] result;

output reg zero;

//output

reg cout;

//output

reg overflow;

//Parameter

reg [32-1:0] result;

//output

wire [32-1:0] result1;

//output

wire [32-1:0] result2;

//output

wire [32-1:0] result3;

//output

wire [32-1:0] result4;

//output

wire [32-1:0] result5;

//output

wire [32-1:0] result6;

wire [6-1:0] zero1;

wire [6-1:0] cout1;

wire [6-1:0] overflow1;

and\_32 an1(.src1(src1),.src2(src2),.result(result1),.cout(cout1[0]),.zero(zero1[0]),.over(overflow1[0]));

or\_32 an2(.src1(src1),.src2(src2),.result(result2),.cout(cout1[1]),.zero(zero1[1]),.over(overflow1[1]));

add\_32 an3(.src1(src1),.src2(src2),.result(result3),.cout(cout1[2]),.zero(zero1[2]),.over(overflow1[2])

//,.a(as)

);

sub\_32 an4(.src1(src1),.src2(src2),.result(result4),.cout(cout1[3]),.zero(zero1[3]),.over(overflow1[3]));

nor\_32 an5(.src1(src1),.src2(src2),.result(result5),.cout(cout1[4]),.zero(zero1[4]),.over(overflow1[4]));

slt\_32 an6(.src1(src1),.src2(src2),.result(result6),.cout(cout1[5]),.zero(zero1[5]),.over(overflow1[5]));

always@( clk or posedge rst\_n)

begin

if(!rst\_n)

begin

result=0;

zero=0;

cout=0;

overflow=0;

end

else

begin

if(ALU\_control==4'b0000)//and

begin

result=result1;

cout=cout1[0];

zero=zero1[0];

overflow=overflow1[0];

end

else if(ALU\_control==4'b0001)//or

begin

result=result2;

cout=cout1[1];

zero=zero1[1];

overflow=overflow1[1];

end

else if(ALU\_control==4'b0010)//add

begin

result=result3;

cout=cout1[2];

zero=zero1[2];

overflow=overflow1[2];

end

else if(ALU\_control==4'b0110)//sub

begin

result=result4;

cout=cout1[3];

zero=zero1[3];

overflow=overflow1[3];

end

else if(ALU\_control==4'b1100)//nor=~p&~Q

begin

result=result5;

cout=cout1[4];

zero=zero1[4];

overflow=overflow1[4];

end

else if(ALU\_control==4'b0111)//slt

begin

result=result6;

cout=cout1[5];

zero=zero1[5];

overflow=overflow1[5];

end

else if(ALU\_control==4'b0101)//mult==>doesn't consider overflow

begin

result=src1 \* src2;

cout=0;

overflow=0;

end

if(result==0)

zero=1;

else

zero=0;

end

end

//Main function

endmodule

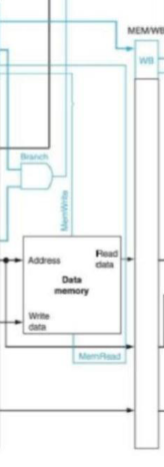
Features:

Mult的實現在這邊是當作mul，然而乘法器在這次的實驗是簡化過後的，因此考量到實現的花費心力與結果的使用性能比不高，就直接用\*來實現。

3. Shifter與上一次實驗所用的相同。

1. Adder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. Pipe\_Reg與上面所用的相同。

----------------------------------MEM stage--------------------------



MEM stage主要包含了兩個component，一個是判斷branch有沒有觸發的 and gate，一個是處理MEM讀取跟寫入memory的DM。

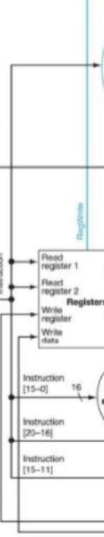
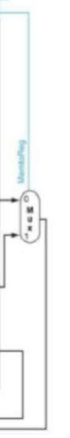
And gate的部分會確認ALU結果是0，而且也有branch的指令，接著就會把結果傳回IF stage，讓下個PC會是branch指向的地址。

DM的部分則是會看有沒有MEMtoreg的訊號去決定要不要讀資料以及有沒有要寫入Memory的訊號去決定要不要使用address跟writedata來決定改誰哪個記憶體跟改寫什麼值。

MEM stage結束後，會把最後WB需要的reg add，5個bits，要不要寫入REG跟要不要MEM\_to\_Reg的訊號以及ALU 運算結果還有MEM讀取的資料，共66bits，總和71bits的暫存器作為MEM stage的結果。

1. And gate一樣是在CPU.v中使用and gate實現
2. DM與上一次實驗所用的相同

-----------------------------------WB stage--------------------------



WB stage，大致上只使用到兩個component，一個專門決定寫回的資料要用ALU的結果還是剛剛MEM讀出結果的MUX，一個是RF的下半部，也就是寫回register的部分。

MUX的部分會根據MEM\_to\_reg來決定輸出的資料。

RF會根據REG\_write來決定要不要寫入。

WB stage完成後，一個指令在pipe\_line的過程也就結束了。

1. MUX與上一次實驗所用的相同
2. RF與上一次實驗所用的相同

----------------------------------ALL stage--------------------------

照著上面spec的設計圖把各個stage要做得以及擁有什麼component都決定好以後，最後就是要來決定我們一整個大循環的.v檔究竟要怎麼安排了。原則上會跟上面按照各個stage解析的過程有一致性。

Coding:

module Pipe\_CPU\_1(

clk\_i,

rst\_i

);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

I/O ports

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

input clk\_i;

input rst\_i;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Internal signal

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\* IF stage \*\*\*\*/

wire [31:0] pc\_in\_i;

wire [31:0] pc\_out\_o;

//piple reg

wire [31:0] IF\_ID\_pc\_4\_i;

wire [31:0] IF\_ID\_in\_i;

wire [31:0] IF\_ID\_pc\_4\_o;

wire [31:0] IF\_ID\_in\_o;

/\*\*\*\* ID stage \*\*\*\*/

//control signal

wire ID\_EX\_mem\_to\_reg\_i\_tmp;

wire ID\_EX\_reg\_write\_i\_tmp;

wire ID\_EX\_mem\_to\_reg\_i;

wire ID\_EX\_reg\_write\_i;

wire ID\_EX\_mem\_to\_reg\_o;

wire ID\_EX\_reg\_write\_o;

// WB stage

wire ID\_EX\_mem\_read\_i\_tmp;

wire ID\_EX\_mem\_write\_i\_tmp;

wire ID\_EX\_branch\_i\_tmp;

wire ID\_EX\_mem\_read\_i;

wire ID\_EX\_mem\_write\_i;

wire ID\_EX\_branch\_i;

wire ID\_EX\_Rtype\_i;

wire ID\_EX\_mem\_read\_o;

wire ID\_EX\_mem\_write\_o;

wire ID\_EX\_branch\_o;

wire ID\_EX\_Rtype\_o;

// MEM stage

wire [3-1:0] ID\_EX\_alu\_op\_i\_tmp;

wire ID\_EX\_alu\_src\_i\_tmp;

wire ID\_EX\_reg\_dst\_i\_tmp;

wire [3-1:0] ID\_EX\_alu\_op\_i;

wire ID\_EX\_alu\_src\_i;

wire ID\_EX\_reg\_dst\_i;

wire [3-1:0] ID\_EX\_alu\_op\_o;

wire ID\_EX\_alu\_src\_o;

wire ID\_EX\_reg\_dst\_o;

// EX stage

wire [31:0] ID\_EX\_pc\_4\_i = IF\_ID\_pc\_4\_o;

wire [31:0] ID\_EX\_read\_data\_1\_i;

wire [31:0] ID\_EX\_read\_data\_2\_i;

wire [31:0] ID\_EX\_sign\_extend\_i;

wire [4:0] ID\_EX\_ins\_rs\_i;

wire [4:0] ID\_EX\_ins\_rt\_i;

wire [4:0] ID\_EX\_ins\_rd\_i;

wire [5:0] ID\_EX\_ins\_op\_i;

wire [31:0] ID\_EX\_pc\_4\_o;

wire [31:0] ID\_EX\_read\_data\_1\_o;

wire [31:0] ID\_EX\_read\_data\_2\_o;

wire [31:0] ID\_EX\_sign\_extend\_o;

wire [4:0] ID\_EX\_ins\_rs\_o;

wire [4:0] ID\_EX\_ins\_rt\_o;

wire [4:0] ID\_EX\_ins\_rd\_o;

wire [5:0] ID\_EX\_ins\_op\_o;

wire ID\_lw\_stall;

/\*\*\*\* EX stage \*\*\*\*/

wire [ 4:0] EX\_MEM\_reg\_dst\_i;

wire [31:0] EX\_MEM\_write\_data\_i;

wire [31:0] EX\_MEM\_alu\_result\_i;

wire EX\_MEM\_zero\_i;

wire [31:0] EX\_MEM\_add\_result\_i;

wire EX\_MEM\_branch\_i;

wire EX\_MEM\_mem\_write\_i;

wire EX\_MEM\_mem\_read\_i;

wire EX\_MEM\_mem\_to\_reg\_i;

wire EX\_MEM\_reg\_write\_i;

wire [31:0] EX\_shift\_left\_2\_o;

wire [31:0] EX\_alu\_src\_2;

// MEM stage

//control signal

wire [3:0] ALUCtrl\_o;

/\*\*\*\* MEM stage \*\*\*\*/

//control signal

wire [ 4:0] EX\_MEM\_reg\_dst\_o;

wire [31:0] EX\_MEM\_write\_data\_o;

wire [31:0] EX\_MEM\_alu\_result\_o;

wire EX\_MEM\_zero\_o;

wire [31:0] EX\_MEM\_add\_result\_o;

wire EX\_MEM\_branch\_o;

wire EX\_MEM\_mem\_write\_o;

wire EX\_MEM\_mem\_read\_o;

wire EX\_MEM\_mem\_to\_reg\_o;

wire EX\_MEM\_reg\_write\_o;

wire [ 4:0] MEM\_WB\_reg\_dst\_i;

wire [31:0] MEM\_WB\_alu\_result\_i;

wire [31:0] MEM\_WB\_read\_data\_i;

wire MEM\_WB\_mem\_to\_reg\_i;

wire MEM\_WB\_reg\_write\_i;

wire MEM\_branch\_take;

/\*\*\*\* WB stage \*\*\*\*/

//control signal

wire [ 4:0] MEM\_WB\_reg\_dst\_o;

wire [31:0] MEM\_WB\_alu\_result\_o;

wire [31:0] MEM\_WB\_read\_data\_o;

wire MEM\_WB\_mem\_to\_reg\_o;

wire MEM\_WB\_reg\_write\_o;

wire [31:0] MEM\_write\_data\_o;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Instantiate modules

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//Instantiate the components in IF stage

MUX\_2to1 #(.size(32)) Mux0(

.data0\_i(IF\_ID\_pc\_4\_i),

.data1\_i(EX\_MEM\_add\_result\_o),

.select\_i(MEM\_branch\_take),

.data\_o(pc\_in\_i)

);

ProgramCounter PC(

.clk\_i(clk\_i),

.rst\_i (rst\_i),

.pc\_in\_i(pc\_in\_i) ,

.pc\_out\_o(pc\_out\_o)

);

Instruction\_Memory IM(

.addr\_i(pc\_out\_o),

.instr\_o(IF\_ID\_in\_i)

);

Adder Add\_pc(

.src1\_i(32'd4),

.src2\_i(pc\_out\_o),

.sum\_o(IF\_ID\_pc\_4\_i)

);

Pipe\_Reg #(.size(64)) IF\_ID( //N is the total length of input/output

.clk\_i(clk\_i),

.rst\_i(rst\_i),

.data\_i({

(MEM\_branch\_take)? 32'b0 : IF\_ID\_pc\_4\_i,

(MEM\_branch\_take)? 32'b0 : IF\_ID\_in\_i

}),

.data\_o({

IF\_ID\_pc\_4\_o,//for PC+4

IF\_ID\_in\_o//for instruction

})

);

Reg\_File RF(

.clk\_i(clk\_i),

.rst\_i(rst\_i),

.RSaddr\_i(IF\_ID\_in\_o[25:21]) ,

.RTaddr\_i(IF\_ID\_in\_o[20:16]) ,

.RDaddr\_i(MEM\_WB\_reg\_dst\_o), // WBstage with address and data and controls

.RDdata\_i(MEM\_write\_data\_o), //WBstage

.RegWrite\_i (MEM\_WB\_reg\_write\_o), // WBstage

.RSdata\_o(ID\_EX\_read\_data\_1\_i),

.RTdata\_o(ID\_EX\_read\_data\_2\_i)

);

Decoder Control(

.instr\_op\_i(IF\_ID\_in\_o[31:26]),

.RegWrite\_o(ID\_EX\_reg\_write\_i\_tmp),

.ALU\_op\_o(ID\_EX\_alu\_op\_i\_tmp),

.ALUSrc\_o(ID\_EX\_alu\_src\_i\_tmp),

.RegDst\_o(ID\_EX\_reg\_dst\_i\_tmp),

.Branch\_o(ID\_EX\_branch\_i\_tmp),

.MemRead\_o(ID\_EX\_mem\_read\_i\_tmp),

.MemWrite\_o(ID\_EX\_mem\_write\_i\_tmp),

.MemtoReg\_o(ID\_EX\_mem\_to\_reg\_i\_tmp),

.Rtype\_o(ID\_EX\_Rtype\_i)

);

Sign\_Extend Sign\_Extend(

.data\_i(IF\_ID\_in\_o[15:0]),

.data\_o(ID\_EX\_sign\_extend\_i)

);

//Instantiate the components in ID stage

assign ID\_EX\_ins\_rt\_i = IF\_ID\_in\_o[20:16];

assign ID\_EX\_ins\_rd\_i = IF\_ID\_in\_o[15:11];

assign ID\_EX\_mem\_to\_reg\_i = ID\_EX\_mem\_to\_reg\_i\_tmp && !MEM\_branch\_take;

assign ID\_EX\_reg\_write\_i = ID\_EX\_reg\_write\_i\_tmp && !MEM\_branch\_take;

assign ID\_EX\_mem\_read\_i = ID\_EX\_mem\_read\_i\_tmp && !MEM\_branch\_take;

assign ID\_EX\_mem\_write\_i = ID\_EX\_mem\_write\_i\_tmp && !MEM\_branch\_take;

assign ID\_EX\_branch\_i = ID\_EX\_branch\_i\_tmp && !MEM\_branch\_take;

assign ID\_EX\_alu\_op\_i[0] = ID\_EX\_alu\_op\_i\_tmp[0] && !MEM\_branch\_take;

assign ID\_EX\_alu\_op\_i[1] = ID\_EX\_alu\_op\_i\_tmp[1] && !MEM\_branch\_take;

assign ID\_EX\_alu\_op\_i[2] = ID\_EX\_alu\_op\_i\_tmp[2] && !MEM\_branch\_take;

assign ID\_EX\_alu\_src\_i = ID\_EX\_alu\_src\_i\_tmp && !MEM\_branch\_take;

assign ID\_EX\_reg\_dst\_i = ID\_EX\_reg\_dst\_i\_tmp && !MEM\_branch\_take;

Pipe\_Reg #(.size(149)) ID\_EX(

.clk\_i(clk\_i),

.rst\_i(rst\_i),

.data\_i({

ID\_EX\_mem\_to\_reg\_i,

ID\_EX\_reg\_write\_i,

ID\_EX\_mem\_read\_i,

ID\_EX\_mem\_write\_i,

ID\_EX\_branch\_i,

ID\_EX\_alu\_op\_i,//3bits

ID\_EX\_alu\_src\_i,

ID\_EX\_reg\_dst\_i,

ID\_EX\_Rtype\_i,

//32bits

ID\_EX\_pc\_4\_i,

ID\_EX\_read\_data\_1\_i,

ID\_EX\_read\_data\_2\_i,

ID\_EX\_sign\_extend\_i,

//5bits

ID\_EX\_ins\_rt\_i,

ID\_EX\_ins\_rd\_i

}),

.data\_o({

ID\_EX\_mem\_to\_reg\_o,

ID\_EX\_reg\_write\_o,

ID\_EX\_mem\_read\_o,

ID\_EX\_mem\_write\_o,

ID\_EX\_branch\_o,

ID\_EX\_alu\_op\_o,

ID\_EX\_alu\_src\_o,

ID\_EX\_reg\_dst\_o,

ID\_EX\_Rtype\_o,

ID\_EX\_pc\_4\_o,

ID\_EX\_read\_data\_1\_o,

ID\_EX\_read\_data\_2\_o,

ID\_EX\_sign\_extend\_o,

ID\_EX\_ins\_rt\_o,

ID\_EX\_ins\_rd\_o

})

);

Shift\_Left\_Two\_32 Shifter(

.data\_i(ID\_EX\_sign\_extend\_o),

.data\_o(EX\_shift\_left\_2\_o)

);

ALU\_Ctrl ALU\_Control(

.Rtype\_i(ID\_EX\_Rtype\_o),

.funct\_i(ID\_EX\_sign\_extend\_o[5:0]),

.ALUOp\_i(ID\_EX\_alu\_op\_o),

.ALUCtrl\_o(ALUCtrl\_o)

);

MUX\_2to1 #(.size(32)) Mux\_ALUSRC(

.data0\_i(ID\_EX\_read\_data\_2\_o),

.data1\_i(ID\_EX\_sign\_extend\_o),

.select\_i(ID\_EX\_alu\_src\_o),

.data\_o(EX\_alu\_src\_2)

);

ALU ALU(

.clk(clk\_i),

.rst\_n(rst\_i),

.src1(ID\_EX\_read\_data\_1\_o),

.src2(EX\_alu\_src\_2),

.ALU\_control(ALUCtrl\_o),

.result(EX\_MEM\_alu\_result\_i),

.zero(EX\_MEM\_zero\_i)

);

MUX\_2to1 #(.size(5)) Mux\_WriReg(

.data0\_i(ID\_EX\_ins\_rt\_o),

.data1\_i(ID\_EX\_ins\_rd\_o),

.select\_i(ID\_EX\_reg\_dst\_o),

.data\_o(EX\_MEM\_reg\_dst\_i)

);

Adder Branch\_pc(

.src1\_i(ID\_EX\_pc\_4\_o),

.src2\_i(EX\_shift\_left\_2\_o),

.sum\_o(EX\_MEM\_add\_result\_i)

);

//Instantiate the components in EX stage

assign EX\_MEM\_write\_data\_i = ID\_EX\_read\_data\_2\_o;

assign EX\_MEM\_branch\_i = ID\_EX\_branch\_o && !MEM\_branch\_take;

assign EX\_MEM\_mem\_write\_i = ID\_EX\_mem\_write\_o && !MEM\_branch\_take;

assign EX\_MEM\_mem\_read\_i = ID\_EX\_mem\_read\_o && !MEM\_branch\_take;

assign EX\_MEM\_mem\_to\_reg\_i = ID\_EX\_mem\_to\_reg\_o && !MEM\_branch\_take;

assign EX\_MEM\_reg\_write\_i = ID\_EX\_reg\_write\_o && !MEM\_branch\_take;

Pipe\_Reg #(.size(107)) EX\_MEM(

.clk\_i(clk\_i),

.rst\_i(rst\_i),

.data\_i({

EX\_MEM\_reg\_dst\_i,//5bits

EX\_MEM\_write\_data\_i,//32bits

EX\_MEM\_alu\_result\_i,//32bits

EX\_MEM\_zero\_i,

EX\_MEM\_add\_result\_i,//32bits

EX\_MEM\_branch\_i,

EX\_MEM\_mem\_write\_i,

EX\_MEM\_mem\_read\_i,

EX\_MEM\_mem\_to\_reg\_i,

EX\_MEM\_reg\_write\_i

}),

.data\_o({

EX\_MEM\_reg\_dst\_o,

EX\_MEM\_write\_data\_o,

EX\_MEM\_alu\_result\_o,

EX\_MEM\_zero\_o,

EX\_MEM\_add\_result\_o,

EX\_MEM\_branch\_o,

EX\_MEM\_mem\_write\_o,

EX\_MEM\_mem\_read\_o,

EX\_MEM\_mem\_to\_reg\_o,

EX\_MEM\_reg\_write\_o

})

);

//Instantiate the components in MEM stage

assign MEM\_branch\_take = EX\_MEM\_zero\_o && EX\_MEM\_branch\_o;

assign MEM\_WB\_reg\_dst\_i = EX\_MEM\_reg\_dst\_o;

assign MEM\_WB\_alu\_result\_i = EX\_MEM\_alu\_result\_o;

assign MEM\_WB\_mem\_to\_reg\_i = EX\_MEM\_mem\_to\_reg\_o;

assign MEM\_WB\_reg\_write\_i = EX\_MEM\_reg\_write\_o;

Data\_Memory DM(

.clk\_i(clk\_i),

.addr\_i(EX\_MEM\_alu\_result\_o),

.data\_i(EX\_MEM\_write\_data\_o),

.MemRead\_i(EX\_MEM\_mem\_read\_o),

.MemWrite\_i(EX\_MEM\_mem\_write\_o),

.data\_o(MEM\_WB\_read\_data\_i)

);

Pipe\_Reg #(.size(71)) MEM\_WB(

.clk\_i(clk\_i),

.rst\_i(rst\_i),

.data\_i({

MEM\_WB\_reg\_dst\_i,//5bits

MEM\_WB\_alu\_result\_i,//32bits

MEM\_WB\_read\_data\_i,//32bits

MEM\_WB\_mem\_to\_reg\_i,

MEM\_WB\_reg\_write\_i

}),

.data\_o({

MEM\_WB\_reg\_dst\_o,

MEM\_WB\_alu\_result\_o,

MEM\_WB\_read\_data\_o,

MEM\_WB\_mem\_to\_reg\_o,

MEM\_WB\_reg\_write\_o

})

);

//Instantiate the components in WB stage

MUX\_2to1 #(.size(32)) Mux\_WB(

.data0\_i(MEM\_WB\_alu\_result\_o),

.data1\_i(MEM\_WB\_read\_data\_o),

.select\_i(MEM\_WB\_mem\_to\_reg\_o),

.data\_o(MEM\_write\_data\_o)

);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

signal assignment

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

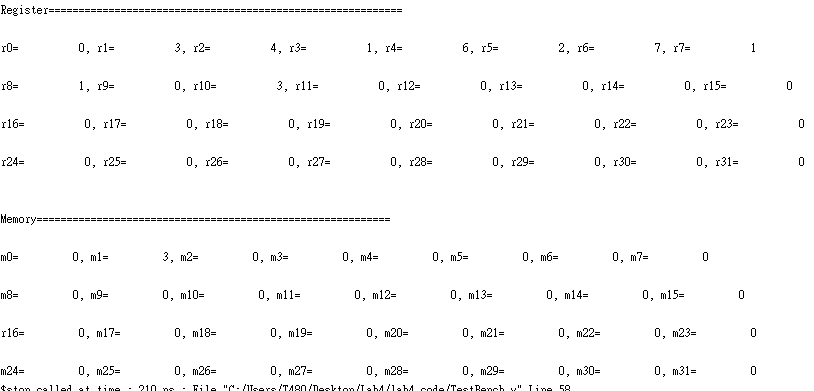
endmodule

Features:

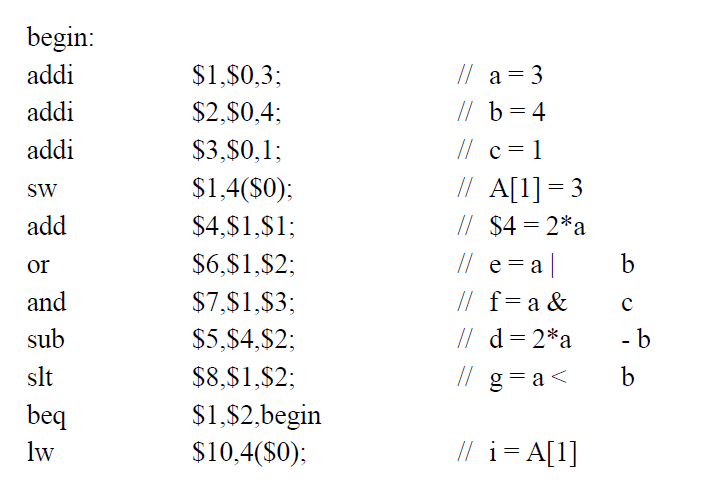
可以發現如果我們照著spec個別分析每個stage需要什麼component，然後再分析每個Pipe\_Reg需要吃什麼資料以後，我們就可以很迅速的把整個大的循環給建立起來。

**Experiment result:**

**第一個實驗:**

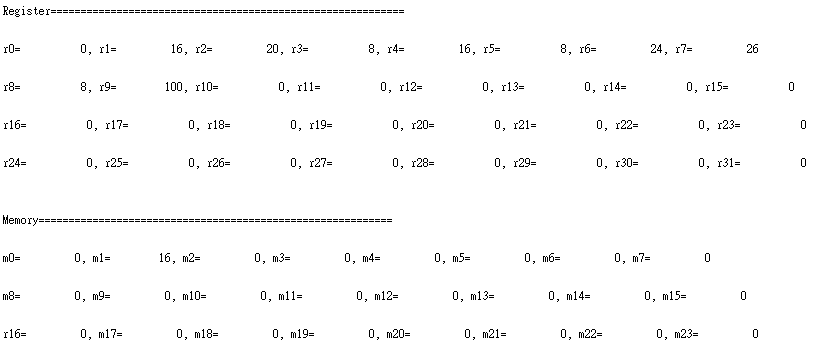


這個test會做的事情如以下:

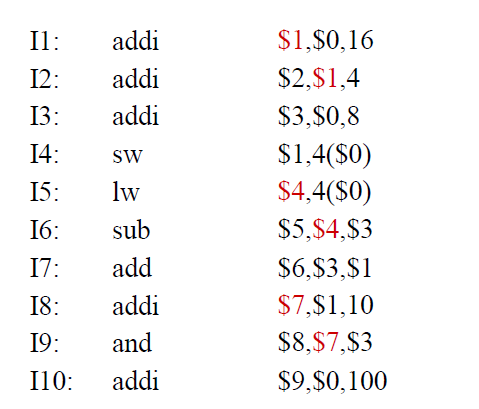
****

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:



可以發現這個順序的話，指令會出現data hazard，這時候要怎麼辦呢?

我自己的話，我會先知道這次實踐的指令中，每個指令都是在ID stage就需要抓取資料，然後除了sw在MEM階段就結束了，其他指令都是在WB階段才結束循環。因此可以判斷的是，ID stage跟WB stage隔了兩個stage，在不使用nops或是forwarding的狀況下，如果是一般的指令最少需要兩行指令來讓hazard被解決掉，因此秉持著這個精神，我先分析出I3跟I10的指令是保證不需要用其他的register作為輸入的指令，而最常被拿來讀取的是$1跟$3，故在編排上我會先把I1跟I3放最前面，好讓後面越來越少Hazard發生，所以可以先得到I1

I3

I10

接著，我們可以發現I2只跟I1有關，I4只跟I1有關，I5跟I4，I6跟I5還有I3有關，I7跟I1還有I3有關，I8跟 I1有關，I9跟I8還有I3有關，所以第四行開始我們可以放心填入的只有I2,I4,I8,但究竟要填入哪個呢?可以發現I4會跟I5有關，連動跟I6有關，所以我們會希望可以填入的指令中，相關最多的先解決，因此I4先填入，接著I3有關的就可以填入。

I1

I3

I10

I4

考量到lw也是要在ID就要把要抓的資料拿出來，因此sw跟lw至少也要隔一個指令才不會有hazard,所以接著可以填入的是I2,I7,I8,但也要注意I6需要I5，而I9需要I8，所以最好先讓I8跟I5可以被完成，故先放入I8，再放入I5。

I1

I3

I10

I4

I8

I5

接著可以放入得有I2,I7，因為剩下的I6跟I9一個需要再兩行，一個只需要再一行就可以使用，故I2跟I7可以隨便選一個，再放入I9，剩下的兩個就可以隨便放了。

一個可能的擺法:

I1

I3

I10

I4

I8

I5

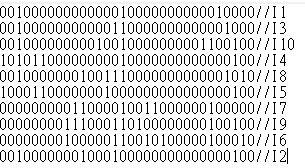
I7

I9

I6

I2

所以我最後修改後的order就是:



**Problems you met and solutions:**

1. 為何IF/ID不需要跟branch確認沒有觸發做&&?

A:當branch確認有沒有觸發的時候是在MEM stage，同時，他也會把訊號傳給PC去決定這個時候的PC到底是要PC+4還是branch的值，因此這個時候IF/ID存的已經是branch確定好以後的PC地址所做出的結果了，(也就是說只有兩個stage被先行吃進來，跟我在上面分析說一般的指令要處理data hazard在不使用forwarding的條件下會需要隔兩行，仔細看一下個別存入跟讀取的stage在比較一下，stage 2跟stage 5,stage 1 跟stage 4，就會發現是同個道理!)

1. 這次實驗少考慮的地方有那些呢?

A:這次的實驗我覺得比較可惜的地方是我branch的處理不夠完善，例如在決定要不要清零時，也可以在把branch的指令跟PC+4的比比看，看多執行的那兩行是不是可以不用全部清掉，這樣子評估的話，估計會在多兩個MUX去決定一個Reg要不要清掉，不過仔細看一下課本，好像沒有提到多出來的兩行再加以利用而不清零的討論，因此我判定可能是實務上執行的話，會因為多了MUX而影響速度跟電路設計，故最後認為還是在branch觸發時直接把那兩行清零比較符合效益比吧!另外，Mult的設計我也覺得處理的不夠好，畢竟課本第三章就是在教怎麼實現乘法器，但坦白說如果ALU裡面的乘法器真的要把那一大坨的電路設計放進去或是要拆分成一個上32bits一個下32bits的方式做拆解MIPS指令，那電路設計上來說似乎就要再多考慮其他的原件，像是要怎麼實現拆成上下呢?有一種方式是也讓ALU多吃一個訊號跟多一個register，然後拆成連續的兩次把值傳出去，但仔細想想這樣的設計就會蠻複雜的。還有，這次也沒有實現data hazard detection跟load stall，以及branch prediction的優化。

**Summary:**

這次的實驗，我認為相比之前的電路複雜超級多，像是我發現我不大可能像之前一樣先把整個大方向結構先生出來，而是必須從spec個別stage分析，然後在最後才能把它整個組合起來!而且很多細部的處理仔細想想才發現在pipe line的狀況下可以多出很多種的處理方式跟問題(如mult)，只能說計算機組織真的是一個很可怕的課程，會在越後面教越多的同時也讓我們知道我們一個小小的CPU到底是經歷了多少前人費盡腦汁的血汗努力，才能誕生出來，這應該算是我目前最respect的一個科目了。也是經由這幾次的作業，我才能大體上明白身為一個工程師原來需要考量的真的有很多很多，而且課本上提供的也只是”問題的方向”，但是”問題的解決”實際上還會根據需求而產生很多變數!計算機組織，真的是有夠靈活的一個學問呢!