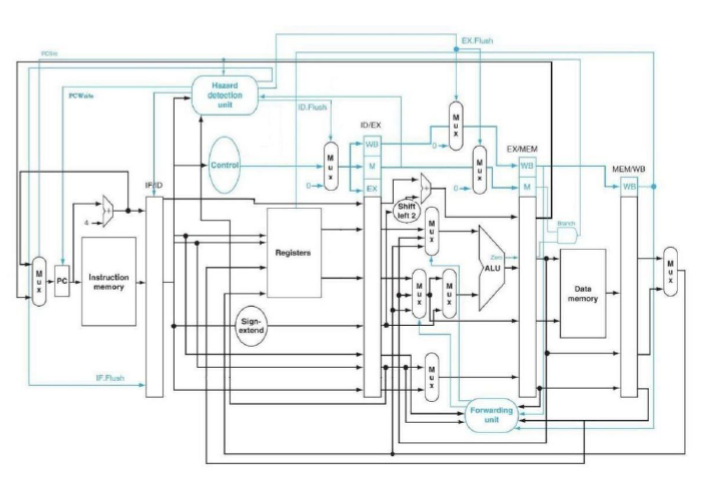
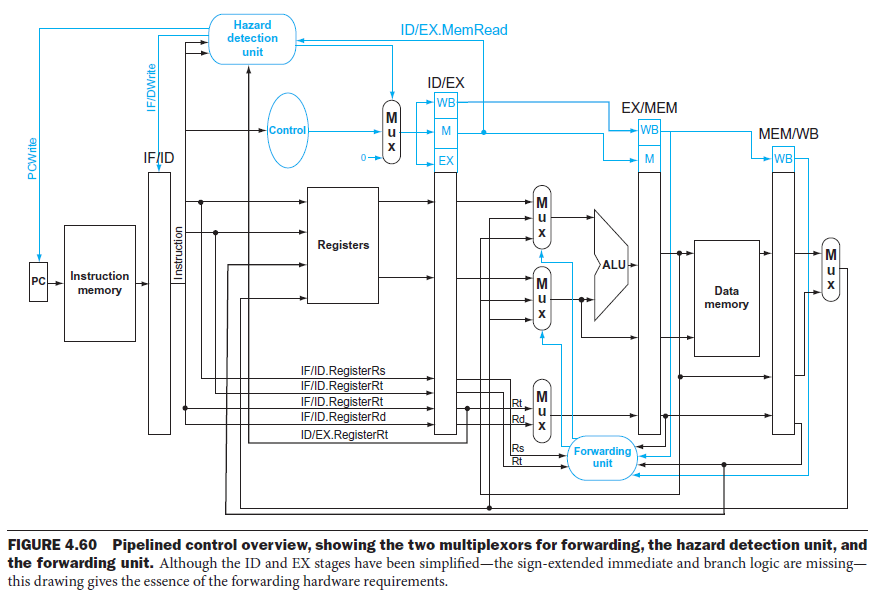
**0810740 張又仁**

**Computer Organization**

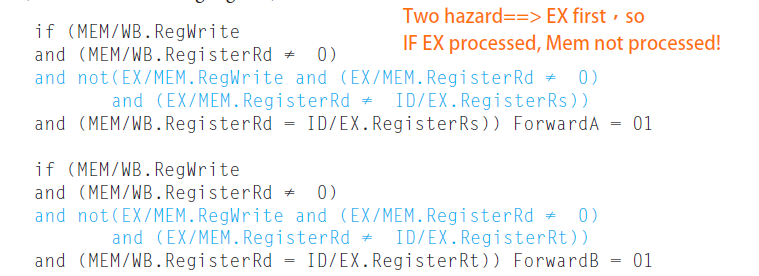
**Architecture diagrams:**

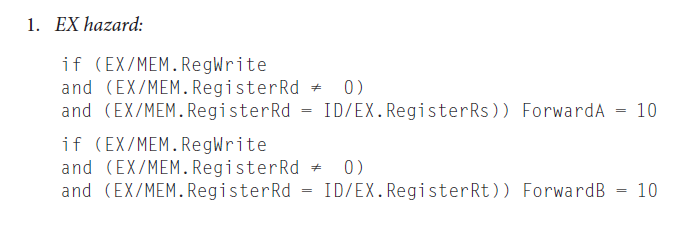
我主要是參考了spec跟課本的架構:(課本是簡易版，spec比較詳細)





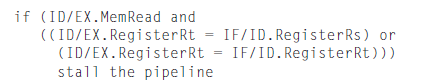
以上的架構是考慮beq,bne,bgt,bge,load,store,R type指令的架構，這一次的實驗主要是要處理Load-use以及forwarding還有branch發生時，要把原本預期not taken做的事情給清掉，在Forwarding的部分，會需要使用到新的原件，也就是Forwarding unit，我們可以根據課本:





也就是需要判斷有沒有用到WB的寫入，還有需要MEM/WB的地址拿去跟當前的Rs及Rt地址比較，也需要看有沒有用到Reg的寫入，以及把EX/MEM的地址拿去跟當前的Rs及Rt比較，以上面的圖來說，Forwarding的架構都有完成。

接著，Load-use的判斷根據課本:



則是需要看有沒有使用到Mem-read，然後把當前的Rs跟Rt以及ALU的Rt看地址有沒有一樣，因此以上面的圖來說，Load-use的架構也都有畫出來，但是關於把指令重抓，我有用到暫存器去存當前的PC以及IF/ID並且用MUX去判斷要用哪個暫存的值，這部分我認為畫出來會破壞general架構，就不畫了。

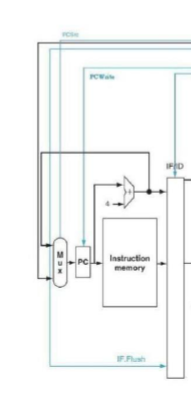
最後檢驗一下branch的部分，branch的部分需要做的就是如果要branch\_taken就把沒有用到的，ID/EX及EX/MEM pipe line 暫存器給flush掉，要注意的是，雖然設計圖上有IF/ID flush，但是實際寫coding的時候，taken發生與PC進來到IF/ID是同時發生的🡺因此只要把ID/EX及EX/MEM的值清掉即可，這是我在實作時與上面設計圖不太一樣的地方(由於本次實驗要多考慮bne及bge及bgt，我實作的方式會需要在branch前面多幾個and gate及or gate，但因為我認為上面的圖很general的處理好了，因此根據多的指令讓設計有點變化的部分我就不畫出來了，另外，實作的時候也有用到ALU\_control，但因為牽涉的訊號源在這邊被簡化了，因此就不畫出來探討)。

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hazard問題，上次的作業其實已經有處理到Branch的部分，因此這次實作重點就是把Forwarding及把Hazard detection檢驗出來，還有多把指令bne及bgt還有bge考慮進來就可以了!

**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5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，一個是當Load-use hazard發生時，我們要多一個stall而必須重新抓指令的Redo元件，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與上一次實驗所用的相同
6. Redo

Redo就是當hazard發生時，我們就會把下一個PC設為現在的PC，並且讓IF/ID保留當前的值，但是，實際上下一個PC進來時仍然會有運作，因此我們只能再用暫存器。

Coding:

module Redo(

redo,

in,

out

);

parameter size = 0;

input redo;

input [size-1:0] in;

output [size-1:0] out;

reg [size-1:0] last;

assign out = (redo)? last : in;

always @(\*) begin

if(redo)

last <= last; //remain the last value

else

last <= in;

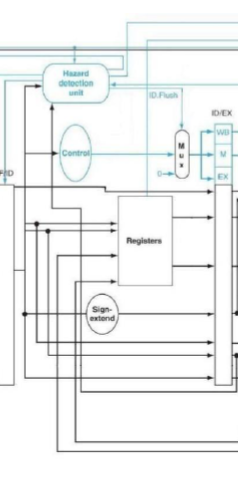
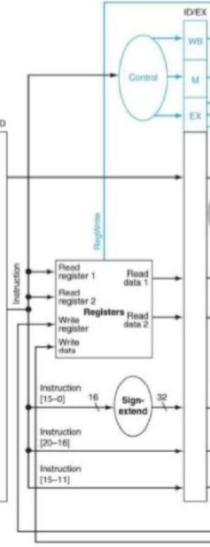
end

endmodule

Features:

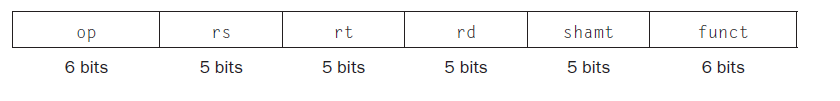
Redo的架構跟Register還有MUX很像，原則上就是兩者的混合版。

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

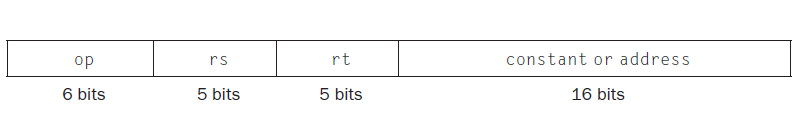


ID stage 與上一次不一樣的地方在多了Hazard detection以及需要把rs的地址往下傳(處理Forwarding的部分)，故總共有4個主要的component，一個是負責把訊號出書的decoder，一個是把最後16bits擴展成32bits的sign extend，一個是負責檢測有沒有Load-use Hazard的Hazard-detection unit，最後則是處理register寫入讀取的RF。

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，bne，bgt，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，訊號方面相比以往多了bne及bgt，其實就跟beq的實現方式一樣，先把相關的值根據對應的指令設為1，接著當滿足條件時(例如相減為Zero或相減大於0等)就會達成branch\_taken的條件，至於bge的部分，因為bge本身就是包含了bgt+beq，因此可以僅僅利用前面舊有的bgt及branch訊號來及架構來處理就可以了，但這次要注意的是我們從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或是Hazard的發生。

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\_bne\_i\_tmp, ID\_EX\_bgt\_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\_taken及Stall做&&確認沒有觸發flush，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，再加上判斷forwarding的rs 5bits，總和156bits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. Lw\_Hazard\_Detection

檢測有沒有發生Load-use Hazard誠如上面所寫的，就是把Rs及Rt還有MemRead及下個stage的Rt拿來做比較就可以知道了，出來的值會影響到後面ID/EX的flush(因為我們在當前ID stage才判斷出來)以及前面IF stage的Redo。

Coding:

module Lw\_Hazard\_Detection(

branch,

IF\_ID\_RS,

IF\_ID\_RT,

ID\_EX\_RT,

ID\_EX\_memr,

stall

);

input branch;

input [4:0] IF\_ID\_RS;

input [4:0] IF\_ID\_RT;

input [4:0] ID\_EX\_RT;

input ID\_EX\_memr;

output stall;

assign stall = !branch && ID\_EX\_memr

&& ( (IF\_ID\_RS==ID\_EX\_RT && IF\_ID\_RS!=0) ||

(IF\_ID\_RT==ID\_EX\_RT && IF\_ID\_RT!=0) );

endmodule

Features:

由於branch taken的時候，我統一會做一個nops並且跳到對應的地址，因此這時候Hazard並不需要也不應該被偵測🡺因為stall會導致指令再多跑一次，但是已經跳到其他地址，因此不需要思考stall!剩下的部分就是把課本給的條件拿來實作即可。

1. Decoder

比上一次的部分多考慮bne，bgt以及bge這三個指令。

Coding:

module Decoder(

instr\_op\_i,

RegWrite\_o,

ALU\_op\_o,

ALUSrc\_o,

RegDst\_o,

Branch\_o,

Bne\_o,

Bgt\_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 Bne\_o;

output Bgt\_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

wire bne;

wire bge;

wire bgt;

//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);

assign bne = (instr\_op\_i==5);

assign bge = (instr\_op\_i==1);

assign bgt = (instr\_op\_i==7);

//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|bge);

assign Bne\_o = (bne);

assign Bgt\_o = (bgt|bge);

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 bne bgt bge-->ALU need sub-->0110-->110

//so lw/sw-->ALU need add-->0010-->010

assign ALU\_op\_o[2] = (((beq|bne)|( bgt|bge))| slti );

assign ALU\_op\_o[1] = (((((beq|bne)|( bgt|bge)) | addi )| slti)|(lw|sw));

assign ALU\_op\_o[0] = slti;

endmodule

Features:

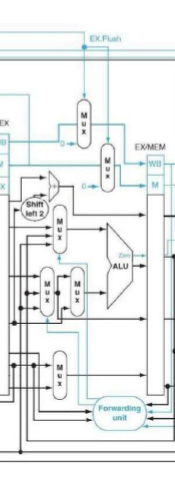
因為bne及bgt及bge還有beq都是同性質的指令，只有在最後判斷taken的地方需要額外判斷不同條件，因此在ALU\_OP的地方都是要用成相減的指令，而同時把bge想成是beq或bgt，就可以很好完成這部分的coding了

3.RF 與上一次實驗所用的相同。

4.SE 與上一次實驗所用的相同。

5.Pipe\_reg與上面一樣。

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



EX stage這邊會用到10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4個ALU相關MUX(flush的MUX不計)，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，兩個三選一的選擇器處理有沒有要forwarding而改變rs,rt的來源，以及一個Fowarding的探測，檢驗需要不要forwarding，以及最後一個EX/MEM的pipe line 暫存器。

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。

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。

Forwarding主要就是根據課本，判斷當前Rs或Rt地址下的數值有沒有需要從EX/MEM或MEM/WB丟過來。

EX stage完成後，一樣是把EX\_MEM\_branch\_i, EX\_MEM\_bne\_i, EX\_MEM\_bgt\_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\_taken做&&確認沒有觸發flush，共8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109bits的暫存器拿來當作這個stage在一個clock結束的結果。

1. ALU\_ctrl與上一次實驗所用的相同。

2. ALU與上一次實驗所用的相同。

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。
4. Mux\_3to1

僅僅就是把Mux\_2to1的概念變成三個，因此在select的輸入端會需要多一個1bit來處理到3個選擇。

Coding:

module MUX\_3to1(

data0\_i,

data1\_i,

data2\_i,

select\_i,

data\_o

);

parameter size = 0;

//I/O ports

input [size-1:0] data0\_i;

input [size-1:0] data1\_i;

input [size-1:0] data2\_i;

input [ 1:0] select\_i;

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

//Internal Signals

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

//Main function

always @(\*) begin

if (select\_i==0)

data\_o <= data0\_i;

else if(select\_i==1)

data\_o <= data1\_i;

else

data\_o <= data2\_i;

end

endmodule

Features:

與Mux\_2to1的架構一致。

1. Forwarding:

主要用來判斷有沒有需要把前面算出的結果，還沒更新到ALU的數據直接拿過來使用，因此輸出的結果會給Rs及Rt最前面的MUX來先做判斷。

Coding:

module Forwarding(

RS\_addr,

RT\_addr,

EX\_MEM\_write,

EX\_MEM\_addr,

MEM\_WB\_write,

MEM\_WB\_addr,

RS\_pick,

RT\_pick

);

input [4:0] RS\_addr;

input [4:0] RT\_addr;

input EX\_MEM\_write;

input [4:0] EX\_MEM\_addr;

input MEM\_WB\_write;

input [4:0] MEM\_WB\_addr;

// 0 from ID/EX

// 1 from EX/MEM

// 2 from MEM/WB

output reg [1:0] RS\_pick;

output reg [1:0] RT\_pick;

always @(\*) begin

// deal with rs

if(MEM\_WB\_write && MEM\_WB\_addr!=0 && RS\_addr==MEM\_WB\_addr)

RS\_pick <= 2;

else if(EX\_MEM\_write && EX\_MEM\_addr!=0 && RS\_addr==EX\_MEM\_addr)

RS\_pick <= 1;

else

RS\_pick <= 0;

// deal with rt

if(MEM\_WB\_write && MEM\_WB\_addr!=0 && RT\_addr==MEM\_WB\_addr)

RT\_pick <= 2;

else if(EX\_MEM\_write && EX\_MEM\_addr!=0 && RT\_addr==EX\_MEM\_addr)

RT\_pick <= 1;

else

RT\_pick <= 0;

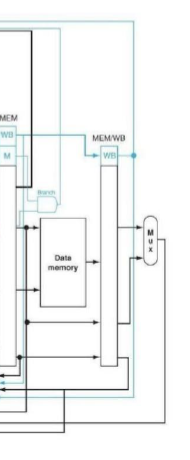
end

endmodule

Features:

與上面提到的一樣，就是把課本提供的給實踐，在這裡因為我是用non-blocking的寫法，因此必須if跟else if才能避免輸出結果錯誤，但如果是用blocking，其實就可以都用if就可以了。(不過考慮到coding的易讀性，我覺得這裡寫成non-blocking比較能對上。

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



MEM stage主要包含了3個component，一個是判斷branch有沒有觸發的 branch gate，一個是處理MEM讀取跟寫入memory的DM以及最後要再做處理的MEM/WB暫存器。

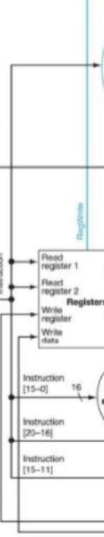
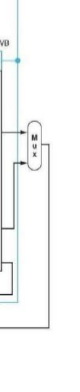
branch gate的第一個部分(設計圖上的and gate)會確認是否ALU結果是0，而且也要有branch的指令，但是要注意的是，在這個實驗尚有bne及bge還有bgt要實現，因此還要再多兩個同level的and gate及把三個and gate 接去最後的or gate輸出成真正的branch\_taken訊號，兩個and gate的部分，一個是輸出結果!=0，一個是輸出結果>0(因為我希望ALU只輸出正常相減是不是0，因此判斷bgt跟判斷bge就必須再多一個元件運算有沒有”大於”，考量到這部分的實作就跟乘法器一樣會多一個大元件，故就選擇在ALL Stage的coding直接用”>”來判斷並簡化!)，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. Branch gate一樣是在CPU.v中使用and gate及or gate實現。
2. DM與上一次實驗所用的相同。
3. Pipe\_Reg與上面所用的相同。

-----------------------------------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;

//hazard

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

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

wire [31:0] IF\_pc;

/\*\*\*\* 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\_bne\_i\_tmp;

wire ID\_EX\_bgt\_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\_bne\_o;

wire ID\_EX\_bgt\_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 [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 ID\_lw\_stall;

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

wire [31:0] EX\_rs\_data;

wire [31:0] EX\_rt\_data;

wire [ 1:0] EX\_rs\_pick;

wire [ 1:0] EX\_rt\_pick;

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\_bne\_i;

wire EX\_MEM\_bgt\_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;

// MEM stage

//control signal

wire [3:0] ALUCtrl\_o;

wire [31:0] EX\_alu\_src\_1;

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

/\*\*\*\* 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\_bne\_o;

wire EX\_MEM\_bgt\_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(IF\_pc)

);

Redo #(.size(32)) PC\_redo(

.redo(ID\_lw\_stall),

.in(IF\_pc),

.out(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\_tmp,//for PC+4

IF\_ID\_in\_o\_tmp//for instruction

})

);

Redo #(.size(64)) IF\_ID\_redo(

.redo(ID\_lw\_stall),

.in({

IF\_ID\_pc\_4\_o\_tmp,

IF\_ID\_in\_o\_tmp

}),

.out({

IF\_ID\_pc\_4\_o,

IF\_ID\_in\_o

})

);

Lw\_Hazard\_Detection lw\_hazard(

.branch(MEM\_branch\_take),

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

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

.ID\_EX\_RT(ID\_EX\_ins\_rt\_o),

.ID\_EX\_memr(ID\_EX\_mem\_read\_o),

.stall(ID\_lw\_stall)

);

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),

.Bne\_o(ID\_EX\_bne\_i\_tmp),

.Bgt\_o(ID\_EX\_bgt\_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\_rs\_i = IF\_ID\_in\_o[25:21];

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&& !ID\_lw\_stall;

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

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

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

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

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

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

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

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

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

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

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

Pipe\_Reg #(.size(156)) 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\_bne\_i,

ID\_EX\_bgt\_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\_rs\_i,

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\_bne\_o,

ID\_EX\_bgt\_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\_rs\_o,

ID\_EX\_ins\_rt\_o,

ID\_EX\_ins\_rd\_o

})

);

Forwarding forwarding(

.RS\_addr(ID\_EX\_ins\_rs\_o),

.RT\_addr(ID\_EX\_ins\_rt\_o),

.EX\_MEM\_write(EX\_MEM\_reg\_write\_o),

.EX\_MEM\_addr(EX\_MEM\_reg\_dst\_o),

.MEM\_WB\_write(MEM\_WB\_reg\_write\_o),

.MEM\_WB\_addr(MEM\_WB\_reg\_dst\_o),

.RS\_pick(EX\_rs\_pick),

.RT\_pick(EX\_rt\_pick)

);

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\_3to1 #(.size(32)) Mux\_RS(

.data0\_i(ID\_EX\_read\_data\_1\_o),

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

.data2\_i(MEM\_write\_data\_o),

.select\_i(EX\_rs\_pick),

.data\_o(EX\_rs\_data)

);

assign EX\_alu\_src\_1=EX\_rs\_data;

MUX\_3to1 #(.size(32)) Mux\_RT(

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

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

.data2\_i(MEM\_write\_data\_o),

.select\_i(EX\_rt\_pick),

.data\_o(EX\_rt\_data)

);

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

.data0\_i(EX\_rt\_data),

.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(EX\_alu\_src\_1),

.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\_bne\_i = ID\_EX\_bne\_o && !MEM\_branch\_take;

assign EX\_MEM\_bgt\_i = ID\_EX\_bgt\_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(109)) 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\_bne\_i,

EX\_MEM\_bgt\_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\_bne\_o,

EX\_MEM\_bgt\_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))|(!EX\_MEM\_zero\_o&&EX\_MEM\_bne\_o))|((EX\_MEM\_alu\_result\_o>0&&EX\_MEM\_bgt\_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

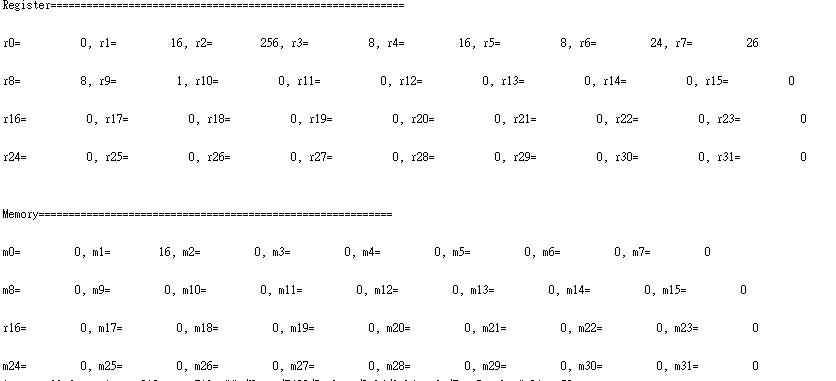
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

endmodule

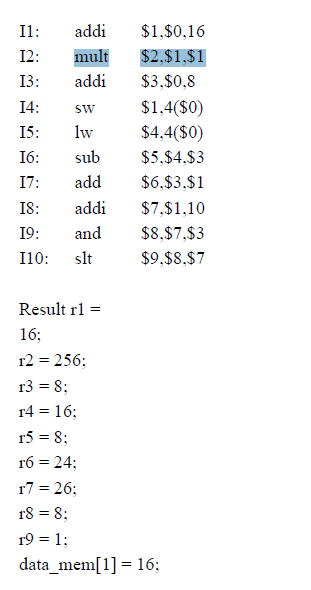
Features:

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:**

****

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。

**Problems you met and solutions:**

這個實驗跟上次實驗相比之下，僅僅是多了兩個大元件，並且元件內部要做什麼事也可以很輕易從課本找到答案，因此我沒遇到什麼值得討論的問題，倒是花了比較長的時間在思考Forwarding的實現跟ALU的關係，也就是用到三選一選擇器的部分花比較多時間，後來仔細再看一次電路設計圖就有想通，因此我認為硬體語言的實現有很大一部分跟對電路的理解有關!還有Load-use hazard的處理，我也是花了很久的時間才想到要把指令重抓一次，因為這在電路上是看不出來的!因此為了實現重抓指令，我也有嘗試把當前PC-4，但是PC-4在branch發生時一定會有問題，因此最後才想到要用Register來實現，但為了減少增加的變數，因此把MUX及register結合在一起，這樣子當Hazard發生時，可以很方便的就把上一次的結果給繼續沿用。

**Summary:**

這次的實驗做完以後，pipe line的CPU也算是基本的完成了，這幾次的實作，真的覺得對於pipe line這種比較複雜的架構，如果我們分成幾個stage去討論會比起我們看整體來要來的好上手跟理解，而這次最主要的目的就是在於Forwarding的處理以及Hazard的處理，我覺得Hazard的處理要特別注意到ID/EXE的flush還有IF stage要使用一個元件去保留上一個的值及一個選擇器讓IF可以變成上一次的值，而在Forwarding這邊，也要特別注意到三選一的選擇器擺放位置，整體而言這次的實驗偏向上次少了很多內容，但是因為課本跟spec都沒有畫出整個完整的電路(想必超級沒有general的感覺，所以課本才沒畫)，所以就要逼自己去思考到底要怎麼辦到這樣的事(也許這也是課本要改進的地方嗎?)總之，這次的實驗很有趣，也讓我理解到我們讀到的文字，例如:指令重抓一次，其實就是用register的特性實現的🡺換句話說，電路裡要使用到過去時間狀態的時候我就可以往暫存器的方向思考，增加我解題速度!