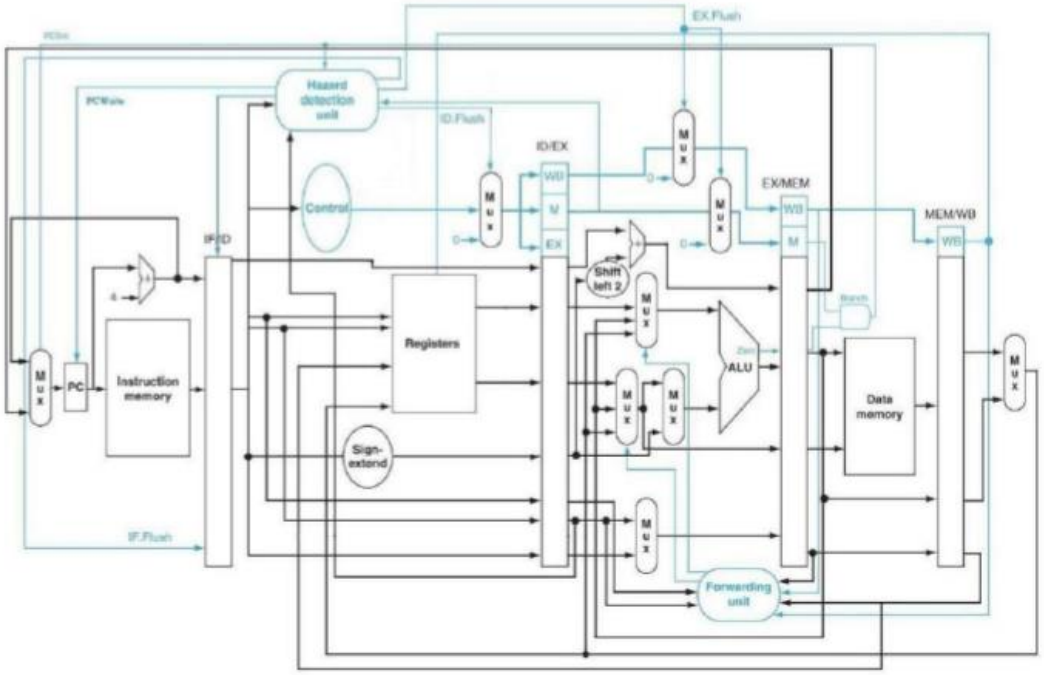
0810740 張又仁

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)



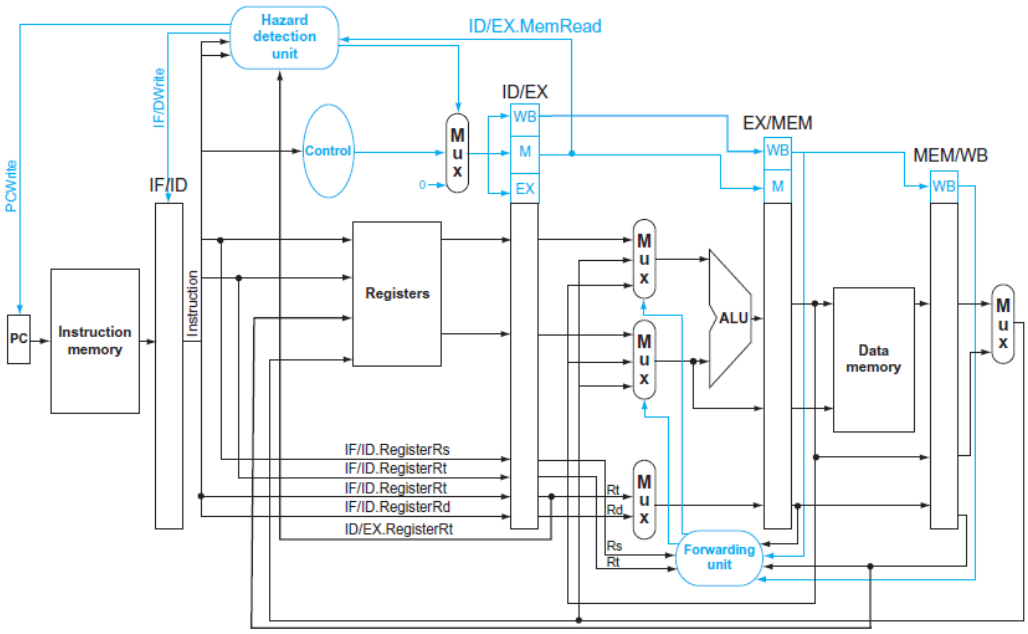


FIGURE 4.60 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements.

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:

Two hazard==> EX first，so
IF EX processed, Mem not processed!

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
      and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
      and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

1. *EX hazard:*

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
```

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

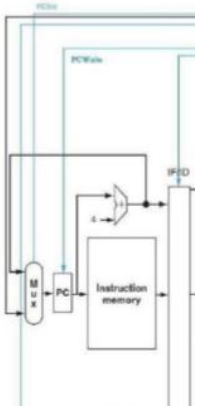
Rs 及 Rt 地址比較，也需要看有沒有用到 Reg 的寫入，以及把 EX/MEM 的地址拿去跟當前的 Rs 及 Rt 比較，以上面的圖來說，Forwarding 的架構都有完成。
接著，Load-use 的判斷根據課本:

```
if (ID/EX.MemRead and
   ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
     (ID/EX.RegisterRt = IF/ID.RegisterRt)))
     stall the pipeline
```

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:

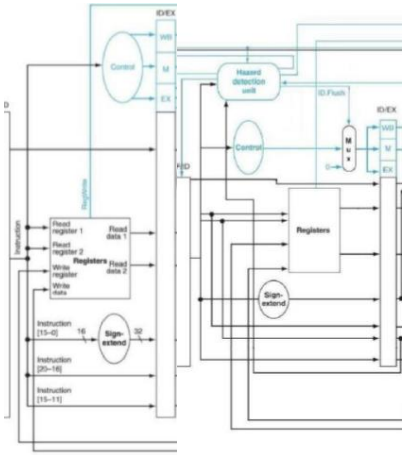
```
module Redo(
    redo,
    in,
    out
    );

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:

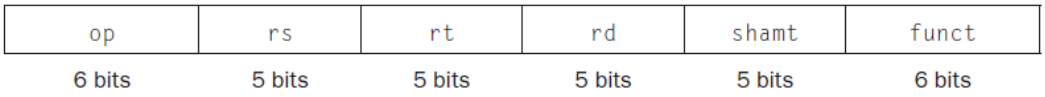
Redo 的架構跟 Register 還有 MUX 很像，原則上就是兩者的混合版。

----------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。

2. 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;
```

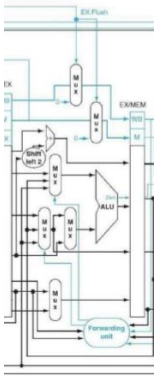
```
assign MemtoReg_o = lw;
assign Jal_o      = jal;
assign Rtype_o    = rtype;

//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。
4. Adder 與上一次實驗所用的相同。
5. MUX 與上一次實驗所用的相同。
6. Pipe_Reg 與上面所用的相同。
7. 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。

8. 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:

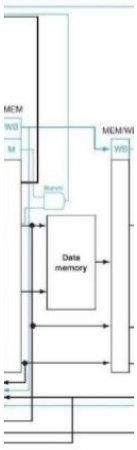
```
module Forwarding(
    RS_addr,
    RT_addr,
    EX_MEM_write,
    EX_MEM_addr,
    MEM_WB_write,
    MEM_WB_addr,
    RS_pick,
    RT_pick
    );

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----------

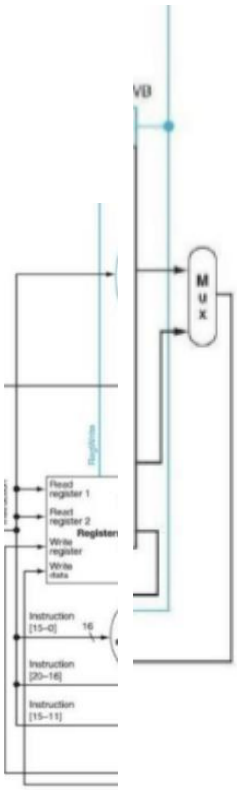


MEM stage 主要包含了 3 個 component，一個是判斷 branch 有沒有觸發的 branch gate，一個是處理 MEM 讀取跟寫入 memory 的 DM 以及最後要再做處理的 MEM/WB 暫存器。

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_resul
t_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:

```
Register============================================================

r0=          0, r1=         16, r2=        256, r3=          8, r4=         16, r5=          8, r6=         24, r7=         26

r8=          8, 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=         0, r30=         0, r31=          0


Memory==============================================================

m0=          0, m1=         16, 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
```

這個 test 會做的事情如以下:

```
I1:     addi    $1,$0,16
I2:     mult    $2,$1,$1
I3:     addi    $3,$0,8
I4:     sw      $1,4($0)
I5:     lw      $4,4($0)
I6:     sub     $5,$4,$3
I7:     add     $6,$3,$1
I8:     addi    $7,$1,10
I9:     and     $8,$7,$3
I10:    slt     $9,$8,$7

Result r1 =
16;
r2 = 256;
r3 = 8;
r4 = 16;
r5 = 8;
r6 = 24;
r7 = 26;
r8 = 8;
r9 = 1;
data_mem[1] = 16;
```

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!