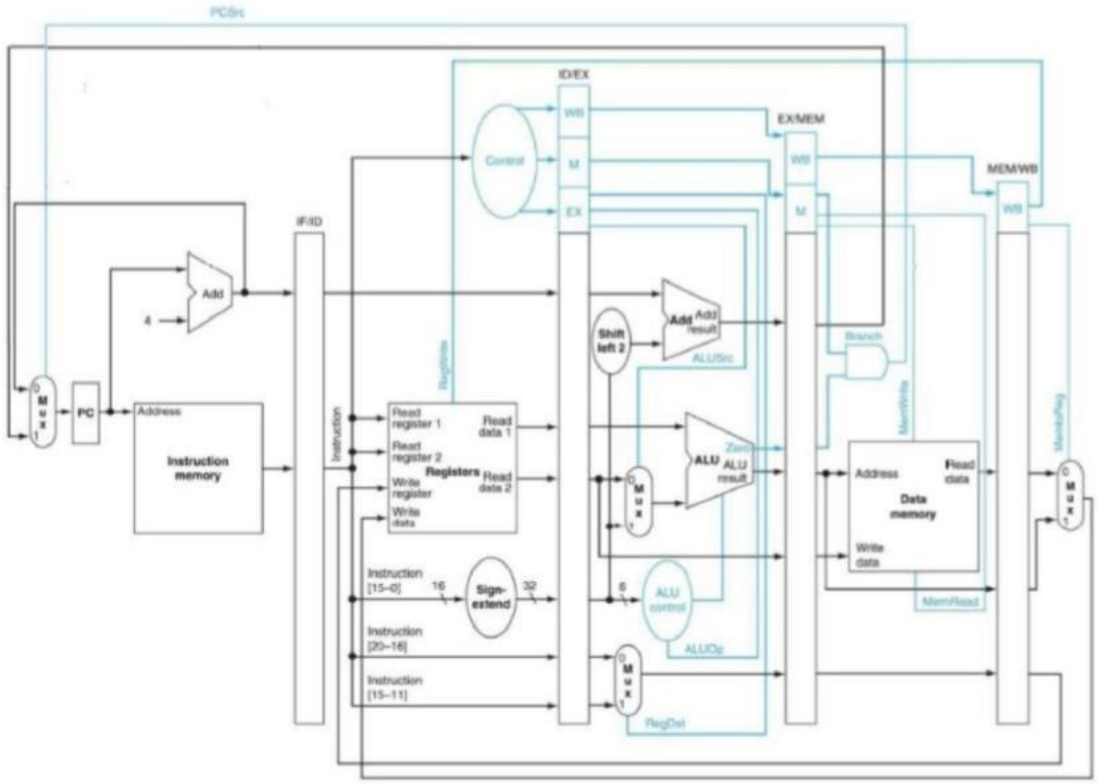
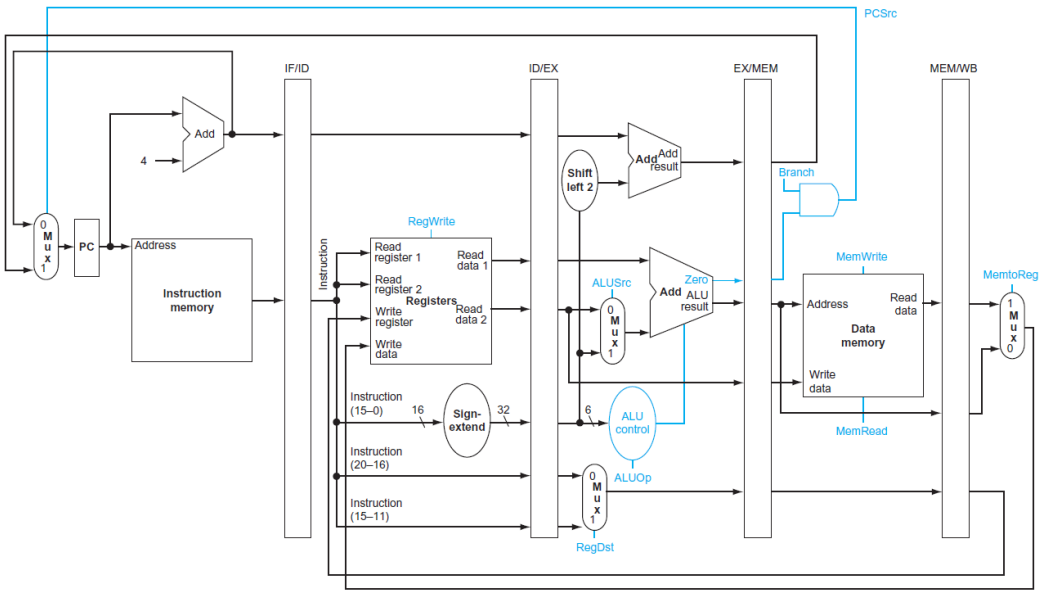
0810740 張又仁

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)





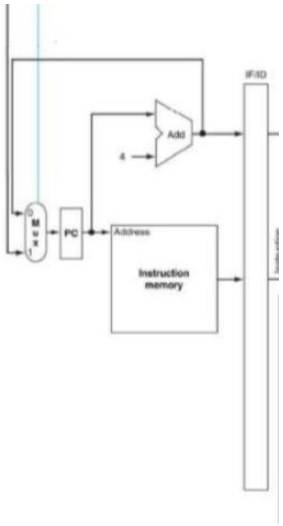
以上的架構是考慮 beq, load, store, d 指令的架構，但是 beq 成功觸發的時候，實際上還是有線會連到 ID/EX 及 EX/MEM 的 register，促使他們把資料清空，畢竟 beq 在確認觸發前也先把下面的指令抓進來了，所以這是上面的圖沒有畫出來的部分(但因為只是多兩條訊號線過去這兩個 registe，我就不再額外畫了)。

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。

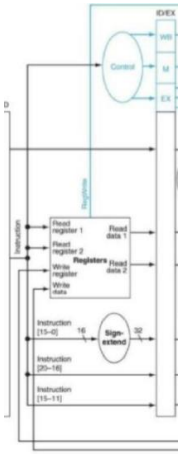
Coding:

```
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:

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

與 I-type:

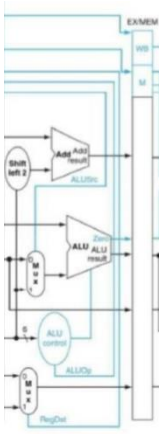
op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

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(ze
ro1[0]),.over(overflow1[0]));
or_32
an2(.src1(src1),.src2(src2),.result(result2),.cout(cout1[1]),.zero(ze
ro1[1]),.over(overflow1[1]));
add_32
an3(.src1(src1),.src2(src2),.result(result3),.cout(cout1[2]),.zero(ze
ro1[2]),.over(overflow1[2])
//,.a(as)
);
sub_32
an4(.src1(src1),.src2(src2),.result(result4),.cout(cout1[3]),.zero(ze
ro1[3]),.over(overflow1[3]));
nor_32
an5(.src1(src1),.src2(src2),.result(result5),.cout(cout1[4]),.zero(ze
```

```
ro1[4]),.over(overflow1[4]));
slt_32
an6(.src1(src1),.src2(src2),.result(result6),.cout(cout1[5]),.zero(ze
ro1[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
```

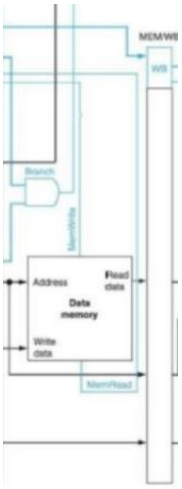
endmodule

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。
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。

----------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。

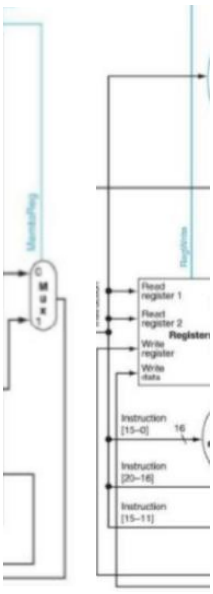
And gate 的部分會確認 ALU 結果是 0，而且也有 branch 的指令，接著就會把結果傳回 IF stage，讓下個 PC 會是 branch 指向的地址。

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:

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

r0=         0, r1=         3, r2=         4, r3=         1, r4=         6, r5=         2, r6=         7, r7=         1

r8=         1, r9=         0, r10=        3, 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=         3, 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
$stop called at time : 210 ns : File "C:/Users/T480/Desktop/Lab4/lab4_code/TestBench.v" Line 58
```

這個 test 會做的事情如以下:

```
begin:
addi    $1,$0,3;        // a = 3
addi    $2,$0,4;        // b = 4
addi    $3,$0,1;        // c = 1
sw      $1,4($0);       // A[1] = 3
add     $4,$1,$1;       // $4 = 2*a
or      $6,$1,$2;       // e = a |     b
and     $7,$1,$3;       // f = a &     c
sub     $5,$4,$2;       // d = 2*a     - b
slt     $8,$1,$2;       // g = a <     b
beq     $1,$2,begin
lw      $10,4($0);      // i = A[1]
```

可以發現與我跑出來的最後結果一致。

第二個實驗:

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

r0=     0, r1=     16, r2=     20, r3=     8, r4=     16, r5=     8, r6=     24, r7=     26

r8=     8, r9=     100, 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
```

這個 test 會執行以下的程式:

```
I1:     addi    $1,$0,16
I2:     addi    $2,$1,4
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:    addi    $9,$0,100
```

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:

```
00100000000000010000000000010000//I1
00100000000000110000000000001000//I3
00100000000010010000000001100100//I10
10101100000000010000000000000100//I4
00100000001001110000000000001010//I8
10001100000001000000000000000100//I5
00000000011000010011000000100000//I7
00000000111000110100000000100100//I9
00000000100000110010100000100010//I6
00100000001000100000000000000100//I2
```

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!)

2. 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!