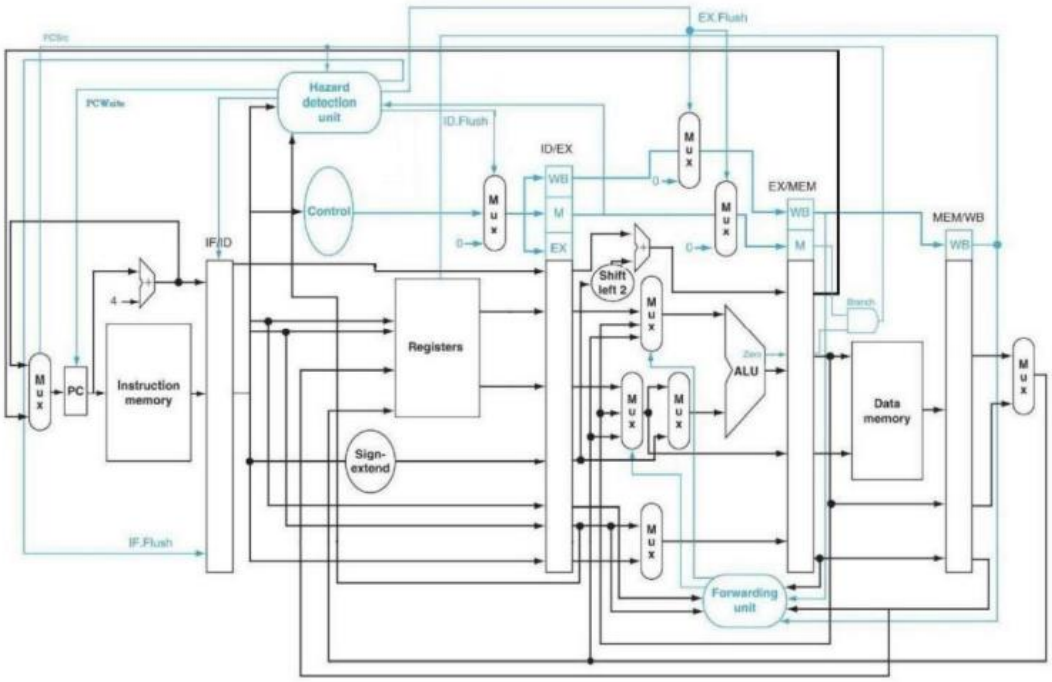
Computer Organization Lab5

Name:張博凱

ID:110550063

Architecture diagrams:

這次的和 Lab4 只差 hazard 的處理，我的實作和圖片一樣，只是圖片有些線路沒畫出來，以及 alu control，但這些部分都和上次的一樣，我唯一的不同是把 alu 的 zero 改成 jump，詳細放在下面說明。



Hardware module analysis:

Adder.v 把兩個 source 相加。

```
//Main function
assign sum_o = src1_i + src2_i;
```

ALU_Ctrl.v 是利用 opcode 和 function code 得出控制 ALU 的 control signal，依照以下表格得出 ALUCtrl_o，和 Lab4 不一樣的是，這次少了 xor，但多了 bne、bge、bgt 要處理。

opcode	ALUOp	operation	funct	ALU funct	ALU ctrl
addi	000	addi	xxxxxx	add	0010
lw		lw	xxxxxx	add	0010
sw		sw	xxxxxx	add	0010
slti	001	slti	xxxxxx	slt	0111
beq	100	beq	xxxxxx	beq	1000
bne	101	bne	xxxxxx	bne	1001
bge	110	bge	xxxxxx	bge	1010
bgt	111	bgt	xxxxxx	bgt	1011
R-format	011	add	100000	add	0010
		sub	100010	sub	0110
		and	100100	and	0000
		or	100101	or	0001
		slt	101010	slt	0111
		mult	011000	mult	0011

```
case (ALUOp_i)
    0: ALUCtrl_o <= 4'b0010;  //addi lw sw
    1: ALUCtrl_o <= 4'b0111;  //slti
    3:  begin
            case (funct_i)
                24: ALUCtrl_o <= 4'b0011;  //mult
                32: ALUCtrl_o <= 4'b0010;  //add
                34: ALUCtrl_o <= 4'b0110;  //sub
                36: ALUCtrl_o <= 4'b0000;  //and
                37: ALUCtrl_o <= 4'b0001;  //or
                42: ALUCtrl_o <= 4'b0111;  //slt
            endcase
        end
    4:  ALUCtrl_o <= 4'b1000; //beq
    5:  ALUCtrl_o <= 4'b1001; //bne
    6:  ALUCtrl_o <= 4'b1010; //bge
    7:  ALUCtrl_o <= 4'b1011; //bgt
    default: ALUCtrl_o <= 4'b0000;
```

ALU.v 是 ALU，這次的做法和前幾次不一樣，首先是將輸出改成 result_o 和 jump_o，在遇到 branch 的時候，判斷是否需要 branch，並且將 result 設為 0，而且和之前一樣，會再和 branch 的 control 做 and。

```
6:  begin
        result_o <= src1_i - src2_i;
        jump_o <= 0;
    end
7:  begin
        result_o <= src1_i < src2_i ? 1 : 0;
        jump_o <= 0;
    end
8:  begin
        result_o <= 0;
        jump_o <= src1_i == src2_i ? 1 : 0;
    end
9:  begin
        result_o <= 0;
        jump_o <= src1_i != src2_i ? 1 : 0;
    end
```

Decoder.v 是將輸入的 op field 的值對應到相對應的功能，最後輸出各種 control signal，下表是對應的結果，和 Lab4 相比，多了 bne、bge、bgt，並且將 aluop 變成 3bit 去處理。

	R-format	addi	lw	sw	slti	Beq	bne	bge	bgt
Op5	0	0	1	1	0	0	0	0	0
Op4	0	0	0	0	0	0	0	0	0
Op3	0	1	0	1	1	0	0	0	0
Op2	0	0	0	0	0	1	1	0	1
Op1	0	0	1	1	1	0	0	0	1
Op0	0	0	1	1	0	0	1	1	1
RegDst	1	0	0	0	0	0	0	0	0
ALUSrc	0	1	1	1	1	0	0	0	0
MemtoReg	0	0	1	0	0	0	0	0	0
RegWrite	1	1	1	0	1	0	0	0	0
MemRead	0	0	1	0	0	0	0	0	0
MemWrite	0	0	0	1	0	0	0	0	0
Branch	0	0	0	0	0	1	1	1	1

ALUOp2	0	0	0	0	0	1	1	1	1
ALUOp1	1	0	0	0	0	0	0	1	1
ALUOp0	1	0	0	0	1	0	1	0	1

```
5:  begin  //bne
        RegDst_o <= 0;
        ALUSrc_o <= 0;
        MemtoReg_o <= 0;
        RegWrite_o <= 0;
        MemRead_o <= 0;
        MemWrite_o <= 0;
        Branch_o <= 1;
        ALU_op_o <= 3'b101;
    end
```

MUX_2to1.v 是基礎的 multiplexer，當 select 等於 0，輸出 source0，當 select 等於 1，輸出 source1。

```
31  //Main function
32  always @(data0_i, data1_i, select_i)
33  begin
34      case (select_i)
35          0: data_o <= data0_i;
36          1: data_o <= data1_i;
37          default: data_o <= data0_i;
38      endcase
39  end
```

MUX_3to1.v 是 3 選 1 的 multiplexer，當 select 等於 0，輸出 source0，當 select 等於 1，輸出 source1，當 select 等於 2，輸出 source2。

```
always @(data0_i, data1_i, data2_i, select_i)
begin
    case (select_i)
        0: data_o <= data0_i;
        1: data_o <= data1_i;
        2: data_o <= data2_i;
        default: data_o <= data0_i;
    endcase
end
```

Shift_Left_Two.v 是將 input data 往左 shift 2 bits。

```
17  //shift left 2
18  assign data_o = data_i << 2;
```

Sign_Extend.v 是將 input data 的 MSB 往左延伸 16 bits。

```
22  //Sign extended
23  assign data_o = {{16{data_i[15]}}, data_i[15:0]};
```

Forwarding.v 是在做 data forwarding，判斷標準和上課時的一樣，先判斷 ex_mem 再判斷 mem_wb。

```
always @(*)
begin
    if(ex_mem_regwrite_i & (ex_mem_rd_i!=0) & (ex_mem_rd_i == id_ex_rs_i)) forwardA_o <= 2'b01;
    else if(mem_wb_regwrite_i & (mem_wb_rd_i!=0) & (mem_wb_rd_i == id_ex_rs_i)) forwardA_o <= 2'b10;
    else  forwardA_o <= 2'b00;

    if(ex_mem_regwrite_i & (ex_mem_rd_i!=0) & (ex_mem_rd_i == id_ex_rt_i)) forwardB_o <= 2'b01;
    else if(mem_wb_regwrite_i & (mem_wb_rd_i!=0) & (mem_wb_rd_i == id_ex_rt_i)) forwardB_o <= 2'b10;
    else  forwardB_o <= 2'b00;
end
```

Hazard_Detection.v 是 hazard detection unit，先判斷是否有需要 branch，再去判斷有沒有 load use hazard，進行 stall 和 flush 的操作。

```
if(branch_i) begin
    pc_write_o <= 1;
    if_id_write_o <= 0;
    if_flush_o <= 1;
    id_flush_o <= 1;
    ex_flush_o <= 1;
end
```

```
else begin
    if(memread_ex_i & ((id_ex_rt_i == if_id_rs_i)|(id_ex_rt_i == if_id_rt_i))) begin
        pc_write_o <= 0;
        if_id_write_o <= 0;
        if_flush_o <= 0;
        id_flush_o <= 1;
        ex_flush_o <= 0;
    end
    else begin
        pc_write_o <= 1;
        if_id_write_o <= 1;
        if_flush_o <= 0;
        id_flush_o <= 0;
        ex_flush_o <= 0;
    end
end
```

Pipe_CPU_1.v 是要把所有的 module 都接線，和上次比增加了 forwarding 和 hazard detection unit，除此之外再 alu 那邊，將上次的 zero 換成 jump，代表是否符合條件要進行 branch，還有 pipe register 多了 write 和 flush，這次 id 和 ex 為了符合 diagram，因此使用了 mux，並不是直接將 register 做 flush，其餘的部分都是照著 diagram 接線。

優點:
可以不需要考慮 instruction reorder，直接做 forwarding，還能節省 cycle，少掉多餘的 stall。
缺點:
Module 負擔增加，會降低整體的性能。

Finished part:

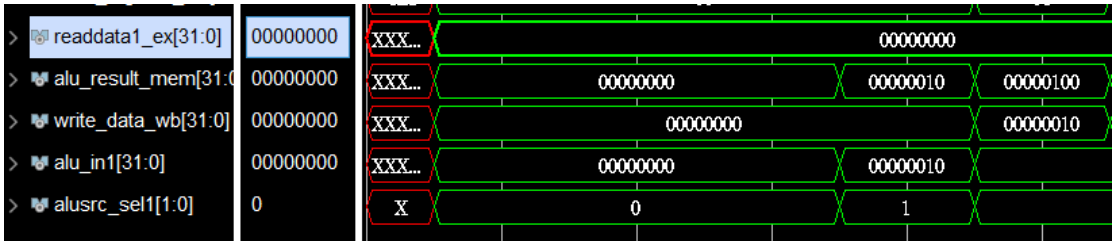
Test1:

```
====================================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

m16=    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
```

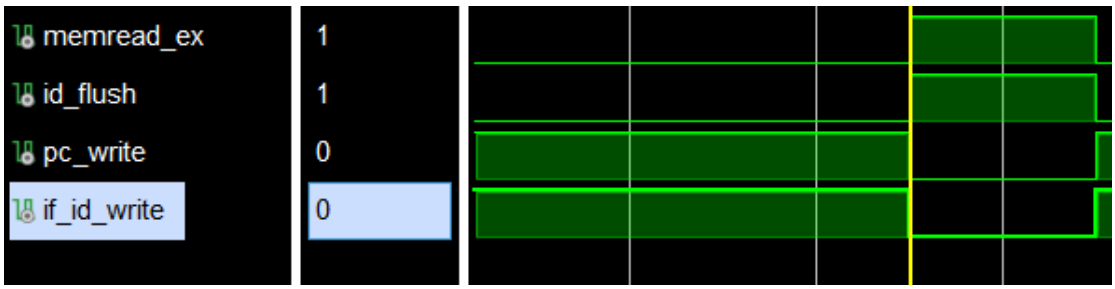
Test2:

```
=======================================Register=======================================
r0 =    0, r1 =    0, r2 =   16, r3 =    6, r4 =    0, r5 =   16, r6 =    0, r7 =    0
r8 =    2, r9 =    0, 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 =    4, m1 =    1, m2 =    0, m3 =    6, m4 =    0, m5 =    0, m6 =    0, m7 =    0
m8 =    0, m9 =    0, m10=    0, m11=    0, m12=    0, m13=    0, m14=    0, m15=    0
m16=    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
```

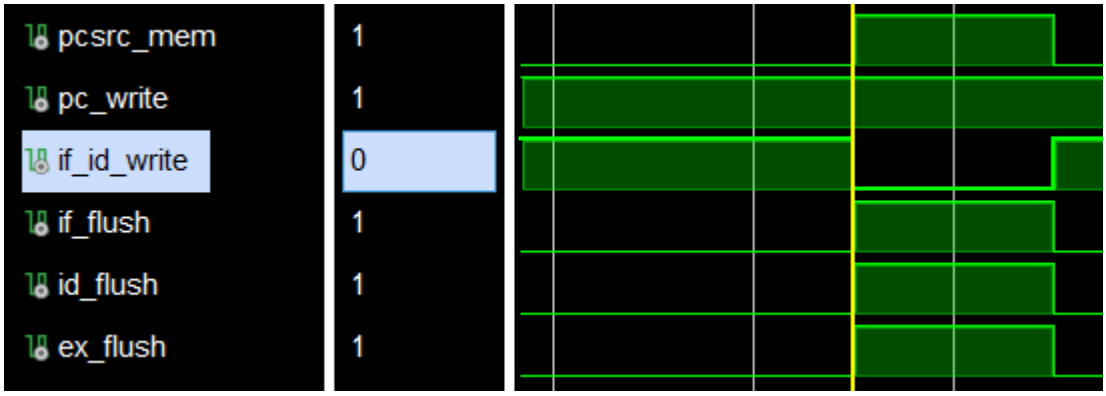
這是 test1 的 I1 到 I2，可以看到她有成功選取 mem state 的值。



這是 test1 的 I5 到 I6，可以看到 load use hazard 發生時，有進行 stall。



可以看到有做 branch 的時候，有進行 flush。



Problems you met and solutions:

在處理 flush 的時候，不是很清楚是否要直接對 register 做 flush，不過因為不會影響結果，所以最後就直接選擇照著 diagram 做。除此之外，也不知道 analysis 裡的優缺點是要寫哪部分，所以就寫這次整體的優缺點。

Summary:

這次是最後一次 Lab，可以明顯感受到整個 module 的線已經多到快看到眼花了，在做的時候都很怕打錯字，然後找不到 bug，雖然難度不高但真的需要很細心，五次的 Lab 做的都蠻快樂的，只是眼睛看得很累。