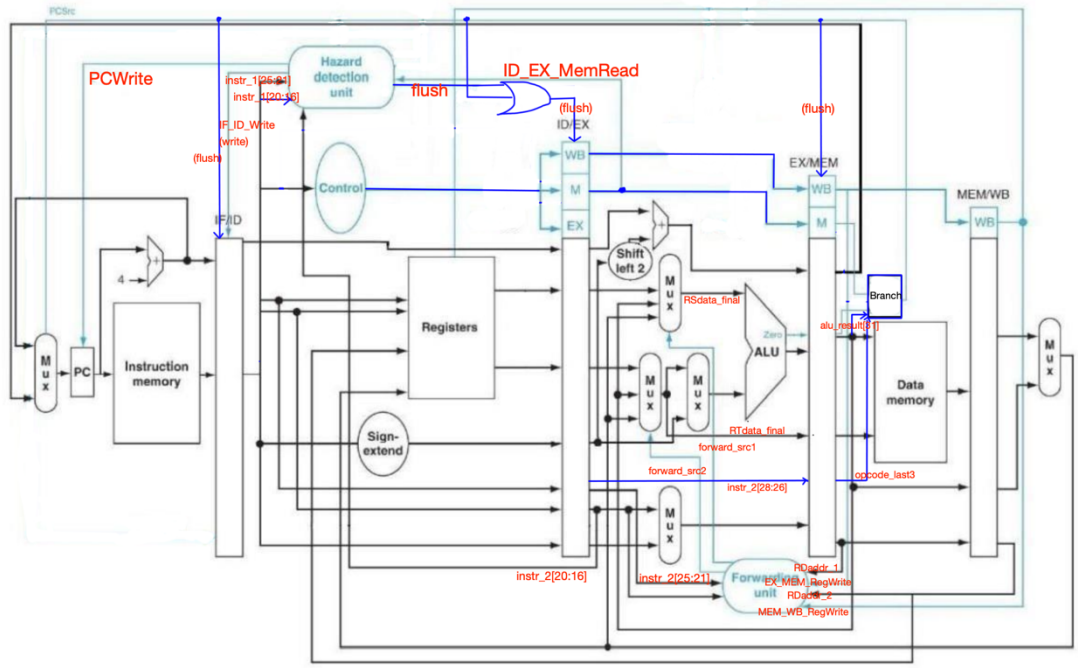
Computer Organization Lab5

ID: 110550029 Name: 陳芷萱

//以 iverilog 編譯

Architecture diagrams



Hardware module analysis

本次作業的設計以 Lab4 為基礎，在以下模組做更動：

- Decode.v

  加入 BNE、BGE、BGT 三種指令，輸出訊號與 BEQ 相同，皆使用 ALU 做減法，後續再根據結果做不同處理。

```
end
6'b000100: begin //beq
        ALU_op_o<=3'b110; ALUSrc_o<=0; RegDst_o<=0; RegWrite_o<=0; Branch_o<=1;
        MemToReg_o<=0; MemRead_o<=0; MemWrite_o<=0;
end
6'b000101: begin //bne
        ALU_op_o<=3'b110; ALUSrc_o<=0; RegDst_o<=0; RegWrite_o<=0; Branch_o<=1;
        MemToReg_o<=0; MemRead_o<=0; MemWrite_o<=0;
end
6'b000001: begin //bge
        ALU_op_o<=3'b110; ALUSrc_o<=0; RegDst_o<=0; RegWrite_o<=0; Branch_o<=1;
        MemToReg_o<=0; MemRead_o<=0; MemWrite_o<=0;
end
6'b000111: begin //bgt
        ALU_op_o<=3'b110; ALUSrc_o<=0; RegDst_o<=0; RegWrite_o<=0; Branch_o<=1;
        MemToReg_o<=0; MemRead_o<=0; MemWrite_o<=0;
```

- Branch.v

  為處理新增的 BNE、BGE、BGT 三種指令，新增此模組替代原本的

AND gate 判斷是否 branch。此模組接收 decoder 輸出之 branch、ALU 運算結果之最高位以及 zero 訊號與指令 opcode 部分末三位（4 種 branch 指令的 opcode 前三位相同）。

若 branch_i（decoder 輸出之 branch 訊號）為 1，表示指令為 BEQ、BNE、BGE、BGT，再根據 opcode 末三位辨認是何種指令。若指令為 BEQ 或 BNQ，可透過 zero 判別 src1 與 src2 是否相等（若相等則相減等於 0），並決定輸出結果；若指令為 BGE，可透過 ALU 運算結果之最高位判別 src1 是否大於等於 src2（若 src1≥src2 則 src1-src2≥0，sign bit 為 0）；若指令為 BGT，同樣可透過 ALU 運算結果之最高位判別 src1 是否大於等於 src2，再根據 zero 排除 src1 與 src2 相等的狀況，最後決定輸出結果。

```
input           branch_i, zero_i, alu_msb_i;
input  [3-1:0]  opcode_last3_i;
output          branch_o;

//Internal signals
reg     branch_o;

//Parameter

//Main function
always @(branch_i, zero_i, alu_msb_i,opcode_last3_i) begin
        if(branch_i) begin
                case(opcode_last3_i)
                        3'b100: branch_o <= zero_i; //beq
                        3'b101: branch_o <= ~zero_i; //bne
                        3'b001: branch_o <= ~alu_msb_i; //bge
                        3'b111: branch_o <= (~alu_msb_i)&(~zero_i); //bgt
                        default: branch_o <= 0;
                endcase
        end
        else branch_o <= 0;
end
```

- MUX_4to1.v

以 MUX_2to1.v 為基礎，將輸入訊號由 2 個增為 4 個，並根據 select 訊號選擇對應的輸入作為輸出。

```
input   [size-1:0] data0_i, data1_i, data2_i, data3_i;
input   [2-1:0]    select_i;
output  [size-1:0] data_o;

//Internal Signals
reg     [size-1:0] data_o;

//Main function
always @(data0_i, data1_i, data2_i, data3_i, select_i) begin
        case(select_i)
                2'b00: data_o<=data0_i;
                2'b01: data_o<=data1_i;
                2'b10: data_o<=data2_i;
                2'b11: data_o<=data3_i;
                default: data_o<=data0_i;
        endcase
end
```

- Forwarding.v

此模組用以判斷是否發生 hazard 並 forwarding。輸入訊號為 EX 與 MEM 階段的 rd address 和 RegWrite 訊號，以及 ID 階段的 rs、rt address，並輸出 forward_src1_o 與 forward_src2_o 兩訊號，分別判斷 ALU 的兩個 source 是否需要 forwarding 以及 forwarding data 來源。

無 hazard 情況時不需 forwarding，預設將 forward_src1_o 與 forward_src2_o 皆設為 00。若 EX 階段的 RegWrite 訊號為 1 表示 EX 階段的指令會將資料寫入暫存器（rd address 不為 0 確保不對$0 寫入），此時若寫

入位置與 ID/EX 階段的 rs 或 rt 相同，會發生 data hazard 而需要 forwarding，因此將輸出設為 01（在 CPU 內會對應到 EX 階段的 ALU 運算結果）。MEM/WB 階段亦做相同處理，然判定時需確認 EX 階段是否已有發生 hazard（輸出訊號是否仍為 0），若符合 hazard 情況且 EX 階段未發生 hazard 則輸出設為 10（在 CPU 內會對應到 MEM 階段的 ALU 運算結果）。

```
input  [5-1:0]  EX_MEM_rd_i, MEM_WB_rd_i, ID_EX_rs_i, ID_EX_rt_i;
input   EX_MEM_RegWrite_i, MEM_WB_RegWrite_i;
output [2-1:0]  forward_src1_o, forward_src2_o;

//Internal signals
reg     forward_src1_o, forward_src2_o;

//Parameter

//Main function
always @(EX_MEM_rd_i, MEM_WB_rd_i, ID_EX_rs_i, ID_EX_rt_i, EX_MEM_RegWrite_i, MEM_WB_RegWrite_i) begin
        //initialize
        forward_src1_o = 2'b00; forward_src2_o = 2'b00;
        //EX hazard
        if(EX_MEM_RegWrite_i&&EX_MEM_rd_i!=0) begin
                if(EX_MEM_rd_i==ID_EX_rs_i) forward_src1_o=2'b01;
                if(EX_MEM_rd_i==ID_EX_rt_i) forward_src2_o=2'b01;
        end
        if(MEM_WB_RegWrite_i&&MEM_WB_rd_i!=0) begin
                if(MEM_WB_rd_i==ID_EX_rs_i && forward_src1_o==2'b00) forward_src1_o=2'b10;
                if(MEM_WB_rd_i==ID_EX_rt_i && forward_src2_o==2'b00) forward_src2_o=2'b10;
        end
end
```

- Hazard_Detect.v

  此模組用以偵測 load-use 並 stall。輸入訊號為 ID 階段的 MemRead 訊號、rt address 與 IF 階段的 rs、rt address，輸出 PCWrite_o（PC 的值是否更新）、IF_ID_Write_o（IF/ID 暫存器的值是否更新）、flush_o（ID/EX 暫存器的控制訊號是否清空）。

  若 ID/EX 階段的 MemRead 訊號為，表示 ID 階段的指令會從記憶體讀取資料，此時若儲存的暫存器與後一個指令的 source 之一相同（IF 階段的 rs 或 rt address），會發生 load-use 的 hazard，需要 stall 一個 clock cycle，將 PCWrite_o 與 IF_ID_Write_o 設為 0（留住 PC 與 IF/ID 暫存器的值）並將 flush_o 設為 1（清除 ID 階段的指令）；未發生 load-use hazard 時，則將 PCWrite_o 與 IF_ID_Write_o 設為 1，flush_o 設為 0。

```
input           ID_EX_MemRead_i;
input  [5-1:0]  ID_EX_rt_i, IF_ID_rs_i, IF_ID_rt_i;
output          PCWrite_o, IF_ID_Write_o, flush_o;

//Internal signals
reg     PCWrite_o, IF_ID_Write_o, flush_o;

//Parameter

//Main function
always @(ID_EX_MemRead_i, ID_EX_rt_i, IF_ID_rs_i, IF_ID_rt_i) begin
        if(ID_EX_MemRead_i && ( ID_EX_rt_i==IF_ID_rs_i || ID_EX_rt_i==IF_ID_rt_i)) begin
                PCWrite_o=0;
                IF_ID_Write_o=0;
                flush_o=1;
        end
        else begin
                PCWrite_o=1;
                IF_ID_Write_o=1;
                flush_o=0;
        end
end
```

- Pipe_CPU_1.v

  將 Hazard Detection Unit、Forwarding Unit、forwarding 所需之 4to1 mutiplexer 與 Branch 模組按照 Architecture Diagram 接入，並且 EX/MEM 暫存器需要再多存 opcode 末三位的資訊。

為解決 branch hazard，當確定需要進行 branch 時，需要 flush 所有後方的指令。由於 Branch 模組位於 MEM 階段，branch 與否會於此階段確定，因此將 Branch 模組的輸出接入前方的 pipeline register，即 IF/ID、ID/EX、EX/MEM 的 flush 輸入。pipeline register 中若有不需考慮 flush 者，flush 輸入 0，若有不需考慮 write 者，write 輸入 1。

```
Pipe_Reg #(.size(64)) IF_ID( // Modify N, which is the total length of input/output
        .clk_i(clk_i),
        .rst_i(rst_i),
        .flush(PCSrc),
        .write(IF_ID_Write),
        .data_i({sum_0_0, instr_0}),
        .data_o({sum_0_1, instr_1})
);
```

```
Pipe_Reg #(.size(170)) ID_EX( // Modify N, which is the total length of input/output
       .clk_i(clk_i),
       .rst_i(rst_i),
       .flush(flush|PCSrc),
       .write(1'b1),
       .data_i({WB_0, M_0, EX, sum_0_1, RSdata_0, RTdata_0, se_0, instr_1}),
       .data_o({WB_1, M_1, RegDst, ALUOp, ALUSrc, sum_0_2, RSdata_1, RTdata_1, se_1, instr_2})
);
```

```
MUX_4to1 #(.size(32)) Mux3( // Modify N, which is the total length of input/output
        .data0_i(RSdata_1),
        .data1_i(ALUresult_1),
        .data2_i(WriteData),
        .data3_i(0),
        .select_i(forward_src1),
        .data_o(RSdata_final)
);

MUX_4to1 #(.size(32)) Mux4( // Modify N, which is the total length of input/output
        .data0_i(RTdata_1),
        .data1_i(ALUresult_1),
        .data2_i(WriteData),
        .data3_i(0),
        .select_i(forward_src2),
        .data_o(RTdata_final)
);
```

```
Pipe_Reg #(.size(110)) EX_MEM( // Modify N, which is the total length of input/output
       .clk_i(clk_i),
       .rst_i(rst_i),
       .flush(PCSrc),
       .write(1'b1),
       .data_i({WB_1, M_1, sum_1_0, Zero_0, ALUresult_0, RTdata_final, instr_2[28:26], RDaddr_0}),
       .data_o({WB_2, branch, MemRead, MemWrite, sum_1_1, Zero_1, ALUresult_1, RTdata_2,
opcode_last3, RDaddr_1})
);
```

```
Pipe_Reg #(.size(71)) MEM_WB( // Modify N, which is the total length of input/output
        .clk_i(clk_i),
        .rst_i(rst_i),
        .flush(1'b0),
        .write(1'b1),
        .data_i({WB_2, ReadData_0, ALUresult_1, RDaddr_1}),
        .data_o({RegWrite, MemtoReg, ReadData_1, ALUresult_2, RDaddr_2})
);

Branch Branch(
        .branch_i(branch),
        .zero_i(Zero_1),
        .alu_msb_i(ALUresult_1[31]),
        .opcode_last3_i(opcode_last3),
        .branch_o(PCSrc)
        );
```

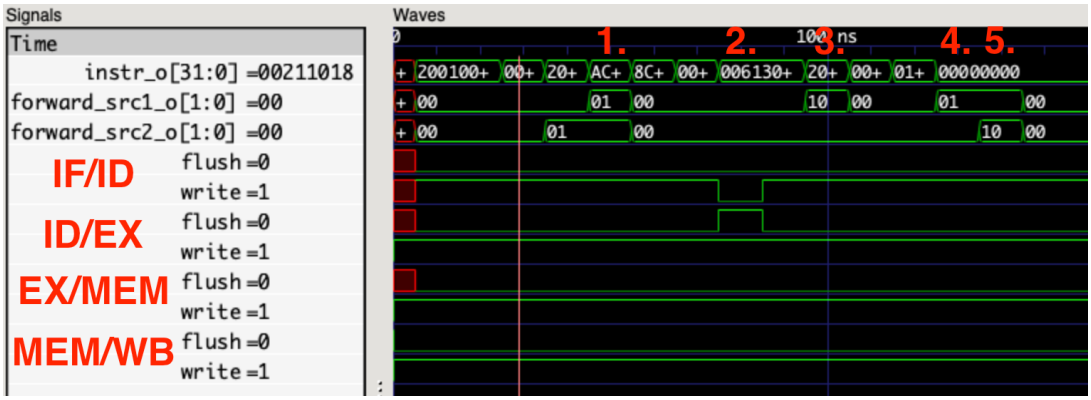
```
Forwarding Forwarding_Unit(
        .EX_MEM_rd_i(RDaddr_1),
        .EX_MEM_RegWrite_i(WB_2[1]),
        .MEM_WB_rd_i(RDaddr_2),
        .MEM_WB_RegWrite_i(RegWrite),
        .ID_EX_rs_i(instr_2[25:21]),
        .ID_EX_rt_i(instr_2[20:16]),
        .forward_src1_o(forward_src1),
        .forward_src2_o(forward_src2)
        );

Hazard_Detect Hazard_Detection_Unit(
        .ID_EX_MemRead_i(M_1[1]),
        .ID_EX_rt_i(instr_2[20:16]),
        .IF_ID_rs_i(instr_1[25:21]),
        .IF_ID_rt_i(instr_1[20:16]),
        .PCWrite_o(PCWrite),
        .IF_ID_Write_o(IF_ID_Write),
        .flush_o(flush)
        );
```

Finished part

- Testcase1

  1. MEM 階段執行到 I1，EX 階段執行到 I2 時，偵測到 EX hazard，對 src1 以及 src2 進行 forwarding，直接取 I1 計算的結果。
  2. EX 階段執行到 I5，ID 階段執行到 I6 時，偵測到 load-use hazard，stall 一個 clock cycle，不更新 IF/ID 暫存器並 flush ID/EX 暫存器、清除 I6。
  3. WB 階段執行到 I5，EX 階段執行到 I6 時，偵測到 MEM hazard，對 src1 進行 forwarding，直接取 I5 讀取的結果。
  4. MEM 階段執行到 I8，EX 階段執行到 I9 時，偵測到 EX hazard，對 src1 以及 src2 進行 forwarding，直接取 I8 計算的結果。
  5. WB 階段執行到 I8， MEM 階段執行到 I9，EX 階段執行到 I10 時，src1 偵測到 EX hazard，進行 forwarding 直接取 I9 計算的結果；src2 偵測到 MEM hazard，進行 forwarding 直接取 I8 計算的結果。



最終結果與解答相符：

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

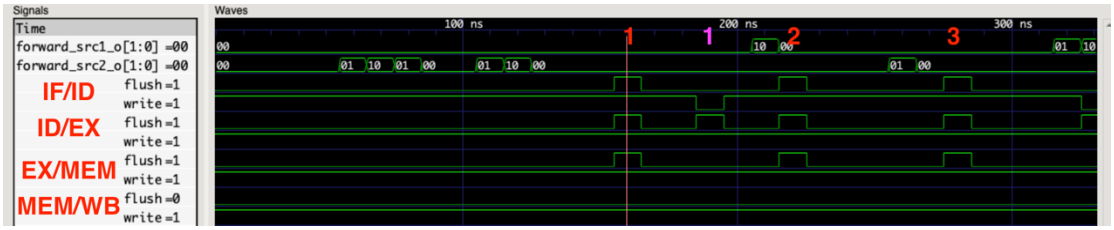
- Testcase2

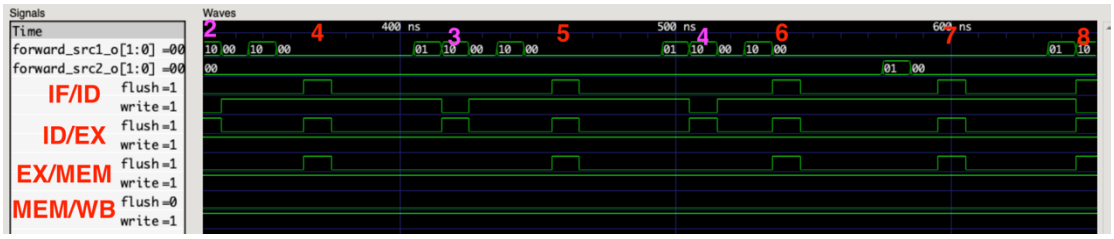
  執行流程：

  I1 (r2=3) -> I2 (m0=3) -> I3 (r2=1) -> I4 (m1=1) -> **I5 (m2=0)** -> I6 (r2=5) -> I7 (m3=5) -> I8 (r2=0) -> **I9 (r5=16)** -> **I10 (r8=2)** -> I11 (to I14) -> I14 (r3=m0=3) -> I15 (to I17) -> I17 (r3=4) -> **I18 (m0=4)** -> I19 (to I12)
  -> I12 (r2=4) -> I13 (x) -> **I14 (r3=m1=1)** -> I15 (x) -> I16 (to I12)

-> I12 (r2=8) -> I13 (x) -> I14 (r3=m2=0) -> I15 (x) -> I16 (to I12)
-> I12 (r2=12) -> I13 (x) -> I14 (r3=m3=5) -> I15 (to I17) -> **I17 (r3=6)** -> **I18 (m3=6)** -> I19 (to I12)
-> **I12 (r2=16)** -> I13 (to I20)

- I11、I13、I15、I16、I19 為 branch 指令，符合條件並 branch 共 8 次，與波形圖相符。
- I14 與 I15 間會發生 load-use hazard，需保持 IF/ID 的狀態並 flush ID/EX，共做 4 次，與波形圖符合。





最終結果與解答相符：

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

Problems you met and solutions

加上 hazard detection 與 forwarding 的模組後，此次的電路又較 Lab4 複雜許多，因此 debug 花了非常久的時間，必須逐一檢查波形圖中的相關訊號與指令結果是否符合、是否有做對應的 forwarding 與 stall。

此次作業除了需要一定程度修改提供的 Architecture Diagram、另做設計外，編譯後才發覺提供的 pipeline reg 與 PC 之輸入訊號與先前不同，因此又重新調整了設計。此外，測試 testcase_2 時結果原先一直不正確，後來才發現是因為 branch 時沒有考慮 flush 後方指令所導致。

Summary

透過此次作業的實作，我對於複雜的 pipeline、hazard 與 forwarding 運作機制有了更加詳細的了解，也更加熟悉指令與資料在 CPU 內的流動與順序。此次作業中並未實作到 prediction、exception handling 等更為複雜的功能，希望未來有機會能夠挑戰。