1. Source code and the note

```
case(branch)
    1'b1:
    begin
      pcwrite <= 1'b1;
      ifid_write <= 1'b1;
      ifid_flush <= 1'b1;
      idex_flush <= 1'b1;
      exmem_flush <= 1'b1;
    end
```

如果 branch instruction 出現，flush 沒用的 pipelined register

```
if(memread == 1'b1 && (instr_i[9:5] == idex_regt || (instr_i[4:0] == idex_regt && instr_i[15:10] != 6'b001000)))
begin
      pcwrite <= 1'b0;
      ifid_write <= 1'b0;
      ifid_flush <= 1'b0;
      idex_flush <= 1'b1;
      exmem_flush <= 1'b0;
end
```

當 ID/EX.MemRead and((ID/EX.RegisterRt = IF/ID.RegisterRs) or(ID/EX.RegisterRt = IF/ID.RegisterRt)), 阻止 pc 的更新,把 idex pipelined register 設成 0，ifid_write 設成 0

```
MUX_4to1 #(.size(1)) Branch_Type(
        .data0_i(ALU_zero_mem),
        .data1_i(~ALU_zero_mem),
        .data2_i(~alu_result_mem[31]),
        .data3_i(~(ALU_zero_mem | alu_result_mem[31])),
        .select_i(BranchType_mem),
        .data_o(branchtype_result)
);
```

決定 Branch 的種類，beq 和 bne 就看 subtraction 出來的結果是否是 1，bge 看 result 的最後 1 bit，如果是 0 表示是大減小，bgt 要額外通過 alu_zero 檢查 A&B 是否一樣

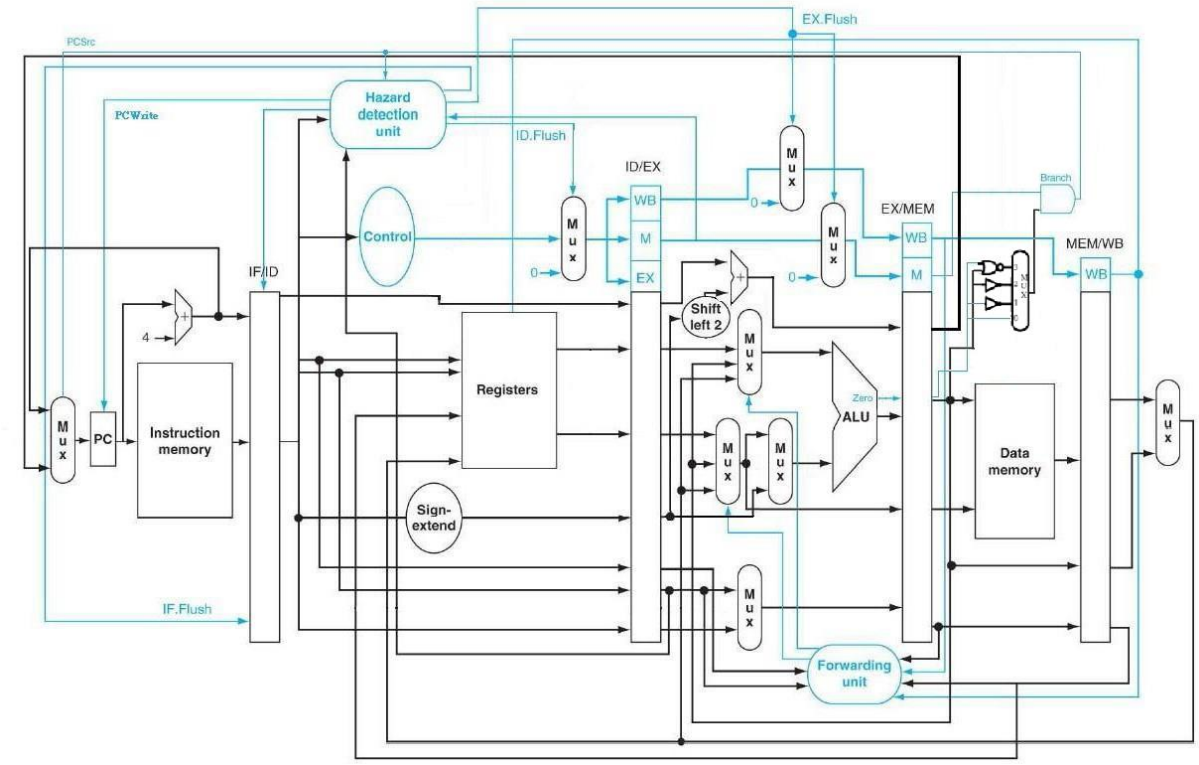
```
    if(regwrite_mem == 1'b1 && idex_regs == exmem_regd && exmem_regd != 5'd0)
        forwarda <= 2'b01;
    else if(regwrite_wb == 1'b1 && idex_regs == memwb_regd && memwb_regd != 5'd0)
        forwarda <= 2'b10;
    else
        forwarda <= 2'b00;
```

按照講義 if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)and (EX/MEM.RegisterRd = ID/EX.RegisterRs))，表示發生 EX hazard，就從 ex/mem pipelined register 把 Rs 值 forward 回去，if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)and (MEM/WB.RegisterRd = ID/EX.RegisterRs))，表示發生 MEM hazard，就從 mem/wb pipelined register 把 Rs 值 forward 回去

```
    if(regwrite_mem == 1'b1 && idex_regt == exmem_regd && exmem_regd != 5'd0)
        forwardb <= 2'b01;
    else if(regwrite_wb == 1'b1 && idex_regt == memwb_regd && memwb_regd != 5'd0)
        forwardb <= 2'b10;
    else
        forwardb <= 2'b00;
```

Rt 值的 forward 原理同 Rs 值

2. Your architecture



3. Hardware module analysis

用上 Lab4 的 module，另新增 Hazard_Detection，Forwarding_Unit, MUX_3to1, MUX_4to1

Adder: 使用 2 個 Adder

(i) 負責計算 PC+4

(ii) 負責計算 imm

ALU: 負責邏輯及加減運算

ALU_Ctrl: 負責 ALU 的 control signal

Data_Memory: 負責存放 data 的 memory

Decoder: 針對不同的 operation, 給予對應的 control signal，個人認為最重要的核心，要傳到不同階段的 Pipe_Reg

Instr_Memory: 負責解讀 instruction

MUX_2to1: 使用 4 個 2 x 1 MUX

(I) 判斷 Write Register 的值

(II) 檢查 R –format 還是 I -format

(III) 判斷 Write Data 的值

(IV) 檢查是否執行 branch

ProgramCounter: 了解目前的 PC

Reg_File: 拆解 operation 里要用到的 register，并存到對應 register

Shift_Left_Two_32: 向左移 2 個 bit

Sign_Extend: 延伸 leftmost bit

Pipe_CPU_1: 不同階段的 data 存到 Pipe_Reg 并連起來

Pipe_Reg: 使用 4 個 Pipe_Reg

(I) 存 IF/ID stage

(II) 存 ID/EX stage

(III) 存 EX/MEM stage

(IV) 存 MEM/WB stage

MUX_3to1: 使用 2 個 3 x 1 MUX

(I) 檢查是否要 Forward A

(II) 檢查是否要 Forward B

MUX_4to1: 使用 1 個 4x 1 MUX

(I) 檢查 branch 的種類

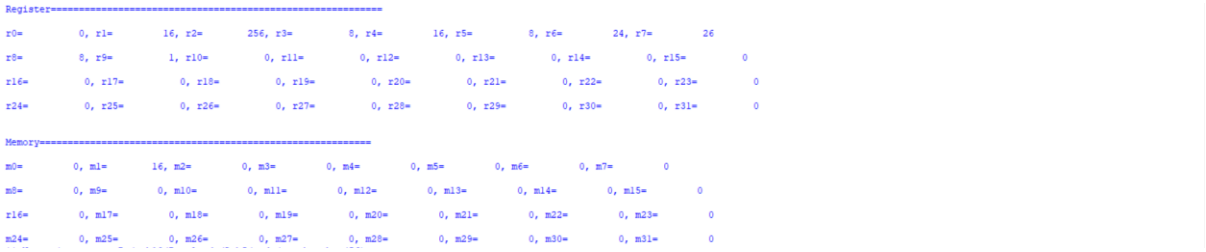
Hazard_Detection: 檢查是否有 load-use hazard 的發生

Forwarding_Unit: 如果發生 data hazard，需要把結果 forward 給下一個指令

4. Finished part

Basic 及 Advanced instruction set

- **CO_P5_test_1**



5. Problems you met and solutions

寫 Forwarding Unit 時 MUX 的值沒有對準，導致 bug，後來看仔細了才發現

6. Summary

了解了蠻多 CPU 的運算原理，還了解了 CPU 遇到 hazard 時所使用的解決方法來提高效能，覺得能想出這邏輯構造的人真的腦子特別好