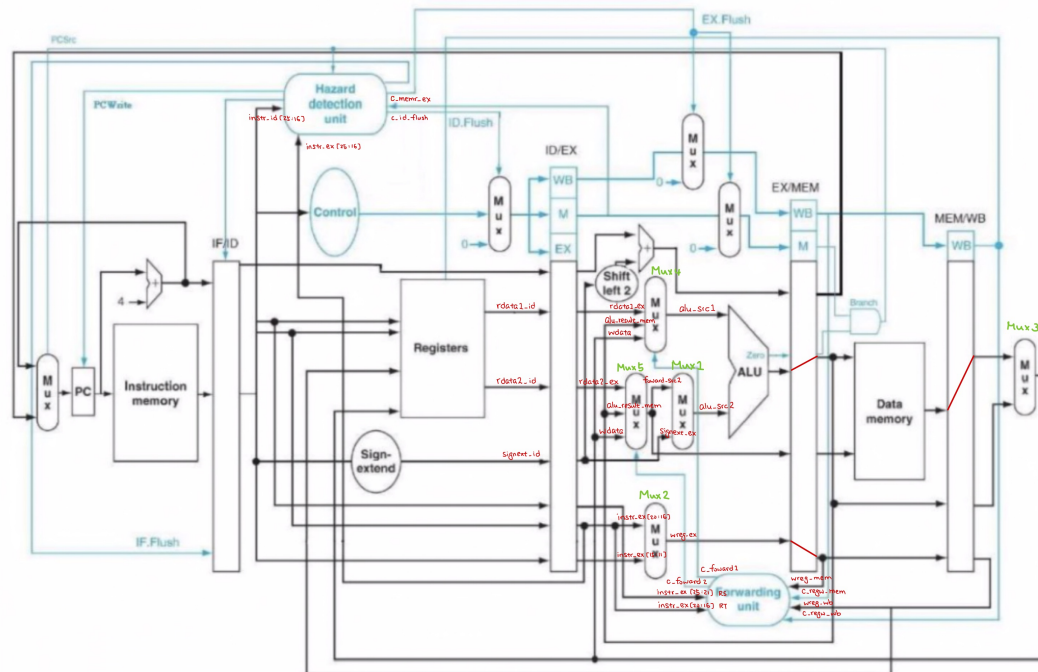


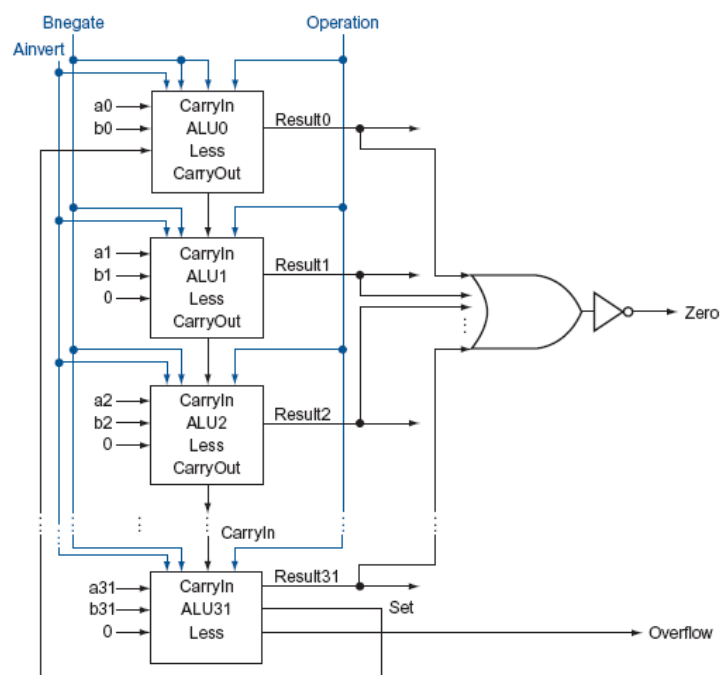
# Computer Organization

## Architecture diagrams:

### Pipe\_CPU\_1.v



### ALU.v



## Hardware module analysis:

### 硬體分析

- **module Pipe\_CPU\_1 (clk\_i, rst\_i)**
  - **module Forwarding (wreg\_mem\_i, wreg\_wb\_i, rs\_i, rt\_i, regw\_mem\_i, regw\_wb\_i, forwardA\_o, forwardB\_o)**  
偵測是否發生 EX Hazard 或 MEM Hazard，若發生的話，送出控制訊號到選擇 ALU source 的 MUX 使其選擇相對應的訊號來源，Forwarding 可以避免 CPU 需要 Stall 而花費更多時間。
  - **module Hazard (pcsrc\_i, memread\_i, instr\_id\_i, instr\_ex\_i, pcwrite\_o, id\_flush\_o, if\_id\_write\_o)**  
檢查是否發生 Load-use hazard 或 Branch hazard，若發生的話則需要送出控制訊號把 CPU Stall 下來（Load-use hazard）或 Flush 掉後面的指令（Branch hazard）。
  - **module Pipe\_Reg (clk\_i, rst\_i, data\_i, data\_o)**  
區隔不同階段（IF、ID、EX、MEM、WB）之間的訊號，在每一個 clock cycle 把前一個階段的訊號儲存起來，並在下一個 clock cycle 來臨時送往下一個階段。這次實驗的重點則是在發生 Hazard 的時候，要判斷是哪個類型的 Hazard，並在需要的時候把暫存器的內容 Stall 下來或甚至 Flush 掉。
  - **module ProgramCounter (clk\_i, rst\_i, pc\_in\_i, pc\_out\_o)**
  - **module Adder (src1\_i, src2\_i, sum\_o)**  
有兩個，一個是 PC 的 Adder ( $PC \leftarrow PC + 4$ )，另一個是算要 Branch 到哪裡（位址）的 Adder
  - **module Instr\_Memory (pc\_addr\_i, instr\_o)**
  - **module MUX\_2to1 (data0\_i, data1\_i, select\_i, data\_o)**  
這次共新增了五個，其中兩個判斷是否需要 Forwarding 作為 ALU 的 Source，另外三個則是在發生 Hazard 需要 Stall 或 Flush 的時候負責把控制訊號設為 0。  
其餘的四個是之前的成品，第一個是選擇 PC 的來源，是要按順序執行下去還是要跳到 branch 的位置，第二個是選擇 ALU 的 Source（從 Register 取出來的值還是從 Instruction[15:0]並 extend 後的值），第三個是選擇指令的哪個部分（Instruction[20:16]或 Instruction[15:11]）作為 Write Register，最後一個是選擇要寫回暫存器的值是 ALU 運算的結果還是從 Data memory 取出來的值
  - **module Reg\_File RF (clk\_i, rst\_i, RSaddr\_i, RTaddr\_i, RDaddr\_i, RDdata\_i, RegWrite\_i, RSdata\_o, RTdata\_o)**
  - **module Decoder (instr\_op\_i, RegWrite\_o, ALU\_op\_o, ALUSrc\_o,**

**RegDst\_o, Branch\_o, Jump\_o, MemRead\_o, MemWrite\_o, MemtoReg\_o)**

根據指令的 OP Field (Instruction 的高 6 bit) 決定要送往各個單元的控制訊號

■ **module ALU\_Ctrl (funct\_i, ALUOp\_i, ALUCtrl\_o)**

根據 Decoder 送來的控制訊號 (aluop) 以及 Instruction 的 Function Field (如有需要的話), 送出 aluctrl 訊號來決定 ALU 實際要執行的動作

■ **module Sign\_Extend (data\_i, data\_o)**

把十六位元的數 extend 成三十二位元, 並保持正負號不變

■ **module ALU (src1\_i, src2\_i, ctrl\_i, result\_o, zero\_o)**

用 Behavioral 的方式實現, 由 aluctrl 控制訊號來決定要執行什麼動作

■ **module Shift\_Left\_Two\_32 (data\_i, data\_o)**

Shift Left 兩次等同於乘以 4, 因為一個 word 是四個 byte

### 訊號分析

● **Pipe\_CPU\_1.v : Pipelined CPU 的主要架構**

■ clk\_i : clock 訊號

■ rst\_i : reset 訊號

● **Forwarding.v : 判斷是否需要 Forwarding, 判斷的條件參照講義 78 與 80 頁, 以下附上與講義上名稱的對照**

■ wreg\_mem\_i : EX/MEM.RegisterRd

■ wreg\_wb\_i : MEM/WB.RegisterRd

■ rs\_i : ID/EX.RegisterRs

■ rt\_i : ID/EX.RegisterRt

■ regw\_mem\_i : EX/MEM.RegWrite

■ regw\_wb\_i : MEM/WB.RegWrite

■ forwardA\_o : ForwardA

■ forwardB\_o : ForwardB

● **Hazard.v : 偵測是否發生 Load-use hazard 或 Branch hazard**

■ pcsrc\_i : 判斷 Branch 是否發生

■ memread\_i : ID/EX.MemRead

■ instr\_id\_i : 停留在 ID 階段的指令, 進來後可以找出指令中對應的 Rs 與 Rt 暫存器

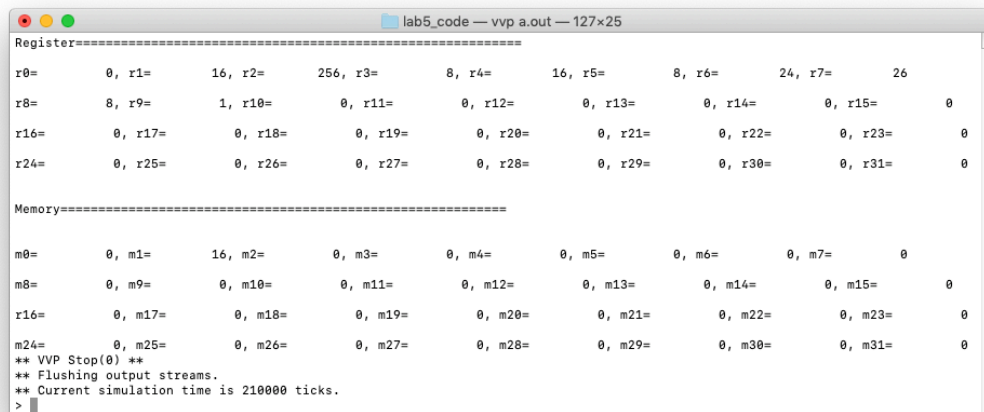
■ instr\_ex\_i : 停留在 EX 階段的指令, 進來後可以找出指令中對應的 Rt 暫存器

■ pcwrite\_o : 若需要 Stall 住 CPU, 則要送出控制訊號使 Program Counter 停止送出下個指令的位址

- id\_flush\_o：若發生 Load-use hazard 或 Branch hazard，需要把 ID 階段的控制訊號設為 0
- if\_id\_write\_o：若發生 Load-use hazard，需要讓 IF/ID 暫存器停止把控制訊號與資料傳下去
- **Pipe\_Reg.v**：區隔不同階段 ( IF、ID、EX、MEM、WB ) 之間的訊號，每隔一個 clock cycle 更新一次
  - clk\_i：clock 訊號
  - rst\_i：reset 訊號
  - data\_i：在當前的 clock cycle 輸入(下個階段需要的資料以及控制訊號)並儲存在暫存器
  - data\_o：下個 clock cycle 輸出 ( 資料以及控制訊號 ) 到下一階段
- **ProgramCounter.v**：每過一個 clock cycle 更新一次 PC
  - clk\_i：clock 訊號
  - rst\_i：reset 訊號
  - pc\_in\_i：新計算出來的 PC
  - pc\_out\_o：每過一個 clock 給 Instruction Memory 一個新的 PC
- **Adder.v**：把兩個 source 訊號相加後輸出
  - src1\_i：32-bit 輸入
  - src2\_i：32-bit 輸入
  - sum\_o：相加後的 32-bit 輸出
- **Instr\_Memory.v**：由 PC 指向的位址去 Fetch Instruction
  - pc\_addr\_i：輸入 PC 指向的位址
  - instr\_o：輸出 32-bit Instruction
- **MUX\_2to1.v**：由控制訊號決定要輸出哪一個輸入訊號
  - data0\_i：1-bit 輸入
  - data1\_i：1-bit 輸入
  - select\_i：控制訊號
  - data\_o：1-bit 輸出
- **Reg\_File.v**：暫存器
  - clk\_i：clock 訊號
  - rst\_i：reset 訊號
  - RSaddr\_i：選擇 RS 暫存器
  - RTaddr\_i：選擇 RT 暫存器
  - RDaddr\_i：選擇 RD 暫存器
  - RDdata\_i：要寫入 RD 暫存器的資料
  - RegWrite\_i：控制是否要寫回暫存器
  - RSdata\_o：讀取到的 RS 暫存器的值
  - RTdata\_o：讀取到的 RT 暫存器的值

- **Decoder.v**：由 **Instruction** 來決定要發送往各個單元的控制訊號
  - **instr\_op\_i**：Instruction 的高 6 位元
  - **RegWrite\_o**：Register File 是否要寫回
  - **ALU\_op\_o**：送往 ALU Control 的控制訊號
  - **ALUSrc\_o**：控制 ALU 的第二個 Source 要採用哪一個
  - **RegDst\_o**：控制要寫回的 Register 是哪一個
  - **Branch\_o**：決定是否為 Branch 指令
  - **Jump\_o**：決定是否為 Jump 或其他跳躍指令
  - **MemRead\_o**：決定是否要寫入記憶體（通常是 sw 指令）
  - **MemWrite\_o**：決定是否要讀取記憶體（通常是 lw 指令）
  - **MemtoReg\_o**：決定是否要寫回暫存器
- **ALU\_Ctrl.v**：由 **Instruction** 及 **Decoder** 的控制訊號決定要對 ALU 進行什麼操作
  - **funct\_i**：Instruction 的低 6 位元
  - **ALUOp\_i**：Decoder 送來的 2-bit 控制訊號
  - **ALUCtrl\_o**：由上面兩個訊號決定實際要送往 ALU 的控制訊號
- **Sign\_Extend.v**：把 16-bit 的訊號 extend 成 32-bit，並維持正負號
  - **data\_i**：16-bit 輸入訊號
  - **data\_o**：32-bit 輸出訊號
- **ALU.v**：運算邏輯單元
  - **src1\_i**：32-bit 輸入訊號
  - **src2\_i**：32-bit 輸入訊號
  - **ctrl\_i**：4-bit 控制訊號
  - **result\_o**：32-bit 計算的結果
  - **zero\_o**：判斷計算出來的結果是否為 0
- **Shift\_Left\_Two\_32.v**：把輸入進來的位址乘以 4
  - **data\_i**：32-bit 輸入進來的訊號（事實上是位址）
  - **data\_o**：32-bit 輸出訊號

## Finished part:



```
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
** VVP Stop(0) **
** Flushing output streams.
** Current simulation time is 210000 ticks.
>
```

### CO\_P5\_test\_1.txt 的實驗結果 實驗結果與助教提供的解答相同

## Problems you met and solutions:

這次遇到最主要的問題是要好好理解講義上關於 Forwarding 以及 Hazard 發生的條件與對應的控制訊號和線路要怎麼安排。因為講義上大部分都是概念性的介紹與討論，實作上的細節比較沒有觸及到。所以這次花最多時間的就是把講義上的觀念，實際轉換成各個控制訊號與電路。整理好之後，其實並不困難，照著助教提供的架構圖，一切都還算順利，這次 Debug 花費最久時間只是一個拼字上的小錯誤。

## Summary:

這次的實驗主要是完成 Forwarding 以及 Hazard Detection 的功能，進一步改進了上一次實驗的效能。以上次實驗的第二筆測資為例，我們需要插入六個 nop 指令才能讓程式正確執行（等同於慢了六個 cycle），但有了這次的架構，我們只需要 Stall 一個 cycle 就能讓程式正確執行，大大提升了 CPU 的效率。關於這次實驗除錯的過程，我只有一句話想說，就是請好好注意拼字，不然會有很嚴重的後果。