# 計算機結構 LAB3 Report

生機四 詹育晟 b10605023

## module explanation

### Adder.v

將兩個 32 位元輸入相加，結果輸出到 res_o 在取指（IF）階段。用它計算 PC + 4，確保指令按順序執行在解碼（ID）階段，用它計算分支目標地址：將符號擴展並左移後的立即數加到當前 PC

### ALU_Control.v

ALU_Control 負責看兩位元的 ALUOp_i 給你下令做什麼運算：

- 00：通通減法（SUB），專門給分支判斷用
- 01：通通加法（ADD），用來算 load/store 的記憶體位址
- 10：碰到 R 型指令，乾脆把 funct7+funct3 拼成 10 位元，一次比對出 AND、XOR、SLL、ADD、SUB、MUL…等指令
- 11：I 型立即數運算，只要看 funct3，000 做 ADDI，101 做 SRAI
  如果碰到怪怪的輸入，就維持 ALUCtr_o = 3'bxxx 給你個錯誤提示。全部邏輯都丟到同一個 always @(*) 裡面，任何輸入一變就馬上更新，超直接。

### ALU.v

這顆 ALU 就像一個 32 位元的萬用運算器，靠著三位元的 ALUCtr_i 來決定要做哪種操作：收到 AND、XOR、SLL 就跑位元運算，收到 ADD、SUB、MUL 就跑加減乘法，收到 ADDI、SRAI 就跑立即數加和算術右移。它在同一個組合邏輯區塊裡一次搞定所有運算，還能透過 Zero_o 旗標告訴你兩個輸入是不是一樣。遇到輸入不符任何指令，就走 default，直接輸出 0，保證

不會出包，超級好跟五級流水線整合。

## Branch_Flusher.v

Branch_Flusher 就是負責把分支預測搞錯的指令「沖掉」的模組。它會先看 ID 階段到底有沒有預測跳轉（ID_predict_i）又是不是分支指令（ID_Branch_i），決定先不先把那兩筆指令標記成要刷掉。接著到了 EX 階段，用實際執行的結果（EX_zero_i）和之前的預測（EX_predict_i）比一比，如果不一樣，就同時把 IF/ID 和 ID/EX 的快取清空，並把 next_pc_select_o 設成正確的 PC 來源（是該走哪個分支或接著走順序 PC）。這樣一來，就能在預測失誤時，立刻清空錯誤指令、跳回正確流程，保持流水線的穩定性。

## Branch_Predictor.v

Branch_Predictor 就是一個超簡版的分支預測小腦袋，拿全局共享的二位元計數器來上下晃：一開始 reset 就被設成「強烈跳轉」(STRONGLY_TAKEN)，代表預設覺得分支一定會跳；每次到 EX 階段遇到分支，就看實際是不是要跳（EX_gtTaken_i），如果預測對了就往更強烈的方向走，預測錯了就往「弱」那邊慢慢飽和倒；一旦累積到最弱就會開始反向預測。最後它只會把這個狀態的最高位元丟出來（predict_o），告訴 ID 階段要不要跳。這種機制能在常碰到、行為穩定的分支上快速修正，偶爾錯也不會一次大轉彎，讓我們的流水線誤判沖刷次數降到最低，跑得更順。

## Control.v

Control 就像流水線的交通號誌，先把所有訊號都關掉（ALUOp=00、ALUSrc=0、RegWrite=0、MemtoReg=0、MemRead=0、MemWrite=0、Branch=0），等到真的有指令要執行（NoOp_i 為 0）才打開對應的號誌：

- R 型指令：打開 ALU 運算和寫回暫存器
- I 型算術指令：除了 ALU，還要把第二個輸入換成立即數
- 載入指令：開 ALU 和記憶體讀，再把讀到的資料寫回暫存器
- 儲存指令：開 ALU 和記憶體寫
- 分支指令：開減法比大小（Branch），打開分支標誌

## Forwarder.v

Forwarder 模組就像資料的捷運線，專門抓 EX 階段要用的兩個暫存器號

（EX_Rs1_i、EX_Rs2_i），看後面的 MEM 或 WB 階段有沒有要把結果寫回同樣的暫存器。如果 MEM 階段要寫的暫存器剛好是 EX 要讀的，就直接從 MEM 拿資料給 ALU，用編碼 10；如果 MEM 不符合但 WB 符合，就從 WB 拿，用編碼 01；最後都不符合就回到自己讀到的值，用 00。

## Hazard_Detection.v

Hazard_Detection 就是來擋住那個「load-use」問題的安全衛士：如果 EX 階段上一週期剛好在做記憶體讀取（EX_MemRead_i），而它要寫回的暫存器（EX_Rd_i）正好是 ID 階段現在要讀的來源暫存器（ID_Rs1_i 或 ID_Rs2_i），那就會一路截停下來。它會把 NoOp_o 拉高，插入一個氣泡；同時把 PCWrite_o 關掉，凍結程式計數器；再把 Stall_o 拉高，讓 IF/ID 那邊暫存器不更新。等到資料真的從記憶體回寫完畢之後，就把這些訊號恢復正常，讓流水線繼續跑。其他時候它就乖乖地把所有訊號都維持在「正常通行」的狀態。

## Imm_Gen.v

Imm_Gen 專門把那 32 位指令裡散落各處的位元拉出來拼成我們需要的立即數。它先看最低 7 位的操作碼，判斷這條指令是算術型、載入型、存儲型還是分支型：

- 如果是 I 型（算術或載入），就直接把 instr_i[31:20] 擷取下來；
- 如果是 S 型（存儲），把 instr_i[31:25] 和 instr_i[11:7] 拼在一起；
- 如果是 SB 型（分支），則按 RISC-V 規範把符號位、instr_i[7]、instr_i[30:25] 跟 instr_i[11:8] 拚成 12 位；
  其他格式就直接給 0。如此一來，無論後面 ALU 怎麼跑位移或加法，都有正確的偏移值可用。

## MUX32_Double.v

MUX32_Double 是一個四選一的 32 位元多工器，根據兩位元的 select_i 控制訊號動態選擇四組輸入中的其中之一輸出到 res_o。在本設計中，它既用於分支預測時從不同階段（IF、ID、EX）的 PC 路徑中選擇正確的下一個 PC，也用於執行階段的資料轉發，將來自 MEM 或 WB 階段的結果「跳過」回 ALU。

## MUX32.v

MUX32 是最簡化的二選一 32 位元多工器，根據單一控制位 select_i 在兩組輸入之間切換：當 select_i=0 時，輸出 src0_i；當 select_i=1 時，輸出 src1_i。在整個五級流水線中，這個模組主要用於執行階段的 ALU 資料來源選擇（決定是採用立即數還是暫存器讀出的資料），以及在寫回階段根據 MemtoReg 信號從 ALU 結果或記憶體讀回資料中選擇最終要寫入暫存器檔的資料。

## Pipeline_EX_MEM.v

在 EX→MEM 階段，我們得把執行階段算好的結果和控制訊號都「拍」到下一級，所以寫了這個 Pipeline_EX_MEM 寄存器。每個時脈上升沿，它就把 ALU 的輸出 (ALUout_i)、要寫進記憶體的資料 (WriteData_i)、目的暫存器編號 (Rd_i)，以及各種寫回、讀寫記憶體的控制旗標，一次性鎖存在自己的「快取」輸出裡。只要 reset（rst_i）變低，它就全部清零，確保一開始或重置後不會殘留髒訊號。

## Pipeline_ID_EX.v

在 ID→EX 階段，我們需要把從暫存器檔讀出的操作數、立即數、當前指令和控制訊號一次性封裝好送到執行階段，用一個同步的暫存器組來完成。這個 Pipeline_ID_EX 模組正是做這件事：每當時脈上升沿，如果沒有重置或分支沖刷（flush_i），它就把所有的資料（包括 RS1data_i、RS2data_i、immed_i、pc_i、instr_i）以及一長串的控制旗標（Rd_i、ALUOp_i、ALUSrc_i、RegWrite_i、MemtoReg_i、MemRead_i、MemWrite_i、Branch_i、Predict_i）和分支位址（pc_branch_i、pc_default_i）都鎖存到對應的輸出端。若遇到重置或分支沖刷，它會把這些暫存器全部清零，避免把錯誤指令帶進下一級。

## Pipeline_IF_ID.v

在取指（IF）與解碼（ID）階段之間，我們必須把取到的指令和對應的程式計數器（PC）值穩定地傳給下一級，這就是 Pipeline_IF_ID 模組的工作。它相當於一組同步暫存器：每次時脈上升沿，如果系統沒有被重置（rst_i 仍為高），也沒有收到分支沖刷信號（flush_i 為低），且沒有被危害單元要求停流水（Stall_i 為低），它就把最新的 instr_i、pc_i 和預設的下一個 PC (pc_default_i) 同步送到輸出 instr_o、pc_o 與 pc_default_o。當遇到 reset 或分支失誤需要沖

刷時，這些暫存器會被清零，避免舊指令或錯誤的位址流入解碼階段；若正在停流水，則會保持前一次的值不變，以此實現「插氣泡」的效果。

## Pipeline_MEM_WB.v

在 MEM→WB 階段，Pipeline_MEM_WB 就像一座橋樑，負責把記憶體讀回的資料、ALU 計算結果，以及寫回暫存器的控制信號，統一暫存到下一個時脈週期。具體來說，它在每次時脈上升沿，先檢查是否處於復位狀態（rst_i 低），若是就把所有輸出清成零，避免殘留髒訊號；否則便把來自 MEM 階段的 ALUout_i、MemoryData_i、目的暫存器 Rd_i、寫回使能 RegWrite_i、以及選擇來源的 MemtoReg_i 全部一口氣鎖存到對應的輸出端。

## Sign_Extend.v

Sign_Extend 模組負責將 12 位元的立即數轉換成 32 位元的帶符號值，簡單來說，就是把最高位（符號位）複製 20 次塞滿到高位，低 12 位則原樣保留。這種做法能確保在後續 ALU 或位址運算中，立即數的正負號能夠正確傳遞，而且只要一條連續的線路就能完成整個擴展，硬體實現既高效又直觀。

## CPU.v

把整個模組串聯起來：

- **取指 (IF) 階段** 用一個 MUX32_Double 選擇下一個 PC（順序、ID 預測分支、EX 更正分支...），然後透過 PC 寄存器鎖存，接著從指令記憶體抓指令並計算 PC+4，最後把結果推到 Pipeline_IF_ID。
- **解碼 (ID) 階段** 透過暫存器檔讀取兩個來源值，再經過 Imm_Gen/Sign_Extend 萃取立即數、Adder 算分支目標、Control 產生所有控制信號，然後打包送進 Pipeline_ID_EX。
- **執行 (EX) 階段** 先透過前級的 Forwarder 把 MEM/WB 階段的結果轉發回來，接著用 MUX32 切立即數或暫存器值，經 ALU_Control 決定運算，再進 ALU 完成算術邏輯運算，最後寄存到 Pipeline_EX_MEM。
- **存取記憶體 (MEM) 階段** 用 Pipeline_EX_MEM 提供的 ALU 結果和寫資料到記憶體，並把讀出的值和控制信號鎖存到 Pipeline_MEM_WB。
- **寫回 (WB) 階段** 再透過一個簡單的 MUX32，根據 MemtoReg 選 ALU 或記憶體的輸出，寫回暫存器檔。
  同時全域還有：

- **Hazard_Detection** 針對 load-use 危害插入氣泡並凍結 PC/IF_ID。
- **Branch_Predictor** 用二位元飽和計數器預測分支方向。
- **Branch_Flusher** 檢查 EX 結果，若預測錯誤就 flush IF_ID/ID_EX 並重新選 PC。

# Difficulties encountered and solutions in this lab

1. **環境設定與 Docker 容器管理**
    - 在 WSL2 內執行 make run 時，Docker-Compose 無法啟動，並持續報錯「container name 'peco' 已被佔用」。
    - 初始嘗試直接重啟或手動修改 Makefile，但每次啟動都撞到既有容器，導致開發流程反覆中斷。
2. **指令對齊與立即數生成**
    - 在實作 Imm_Gen 時，對於 SB-type 分支立即數的多段位拼接（符號位、bit7、30:25、11:8）一度出現錯誤，導致分支地址計算偏移不對。
    - 分支驗證時，流水線多級註冊的 pc_default 與 pc_branch 未同步清除，也造成暫存器裡殘留舊值，使分支沖刷邏輯效果不正確。
3. **資料相依與傳送**
    - 在第一次整合時，EX 階段 ALU 輸入錯拿舊值，經常在緊鄰的 load → use 情境下發生資料錯誤。
    - 嘗試在 Hazard_Detection 裡插入停頓，但若沒有配合 Forwarder，就會過度插 bubble，大幅降低流水線性能。
4. **分支預測與沖刷邏輯**
    - 分支預測器（Branch_Predictor）初始設計只有簡單布林判斷，錯誤率太高。
    - Branch_Flusher 的 next_pc_select 編碼與 IF/ID、ID/EX flush 控制訊號不匹配，導致沖刷後流程卡死。

# Solutions and Strategies

1. **容器清理與重新啟動**
    - 使用 docker rm peco 在每次測試前移除舊有容器，或在 docker-compose.yml 裡改名為 peco_lab3，徹底解決名稱衝突問題。
    - 之後將 make clean && make run 列為固定流程，確保每次起始都是「乾淨」環境。

2. **模組化測試與位域校驗**
    - 針對 Imm_Gen 和 Sign_Extend 撰寫小型 Testbench，分別比對 I/S/SB 型立即數位域，確保每一種格式的輸出正確。
    - 在 Pipeline_IF_ID 和 Pipeline_ID_EX 裡，加入清晰的 reset/flush 順序，並在沖刷時一律清零所有相關暫存器，避免「殘留」效應。
3. **資料轉發與危害插泡協調**
    - 完整實作 Forwarder，先從 MEM 再到 WB 階段檢查轉發條件；同時在 Hazard_Detection 裡只對 load-use 情境插一週期氣泡，其他相依都交給 Forwarder 解決。
    - 透過調整模組優先度（MEM before WB before EX 自身讀取），成功解決大多數資料相依而不插太多泡。
4. **2-bit 飽和計數器分支預測**
    - 將原本簡單布林改為二位元飽和計數器 (STRONGLY_TAKEN → WEAKLY_TAKEN ↔ WEAKLY_NON_TAKEN → STRONGLY_NON_TAKEN)，大幅降低分支誤判率。
    - 在 Branch_Flusher 中統一採用 ID_predict_i 結合 EX_zero_i 做精確判斷，並按需拉高 IF_ID_flush_o/ID_EX_flush_o 與設定 next_pc_select_o，保證沖刷後跳回正確路徑。

# Development environment

- **平台**：Windows 10 + WSL2（Ubuntu 20.04），利用 Linux 原生環境跑工具和指令。
- **容器化**：用 Docker Desktop（WSL 整合模式），搭配 docker-compose.yml，一次 make run 就能啟動包含 Icarus Verilog、Python 測試腳本的整套模擬環境。
- **編輯器**：VS Code 搭配 Remote-WSL 外掛直接在 WSL2 內編輯 Verilog，並用語法高亮與 Lint 即時檢查。
- **模擬測試**：Icarus Verilog (iverilog) 做合成與模擬，Python + pytest 自動跑單元測試和整體流水線驗證，Pass/Fail 報告一鍵生成。
- **除錯**：GTKWave 觀波形定位訊號變化，docker logs / docker exec 進容器內看錯誤、調整設定。