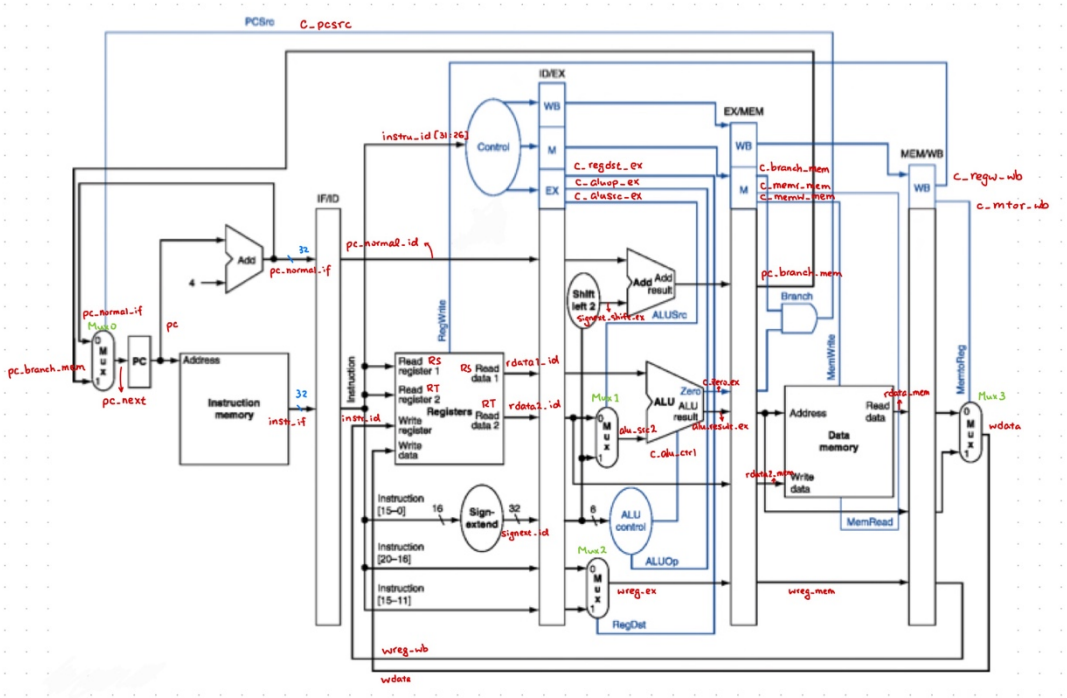
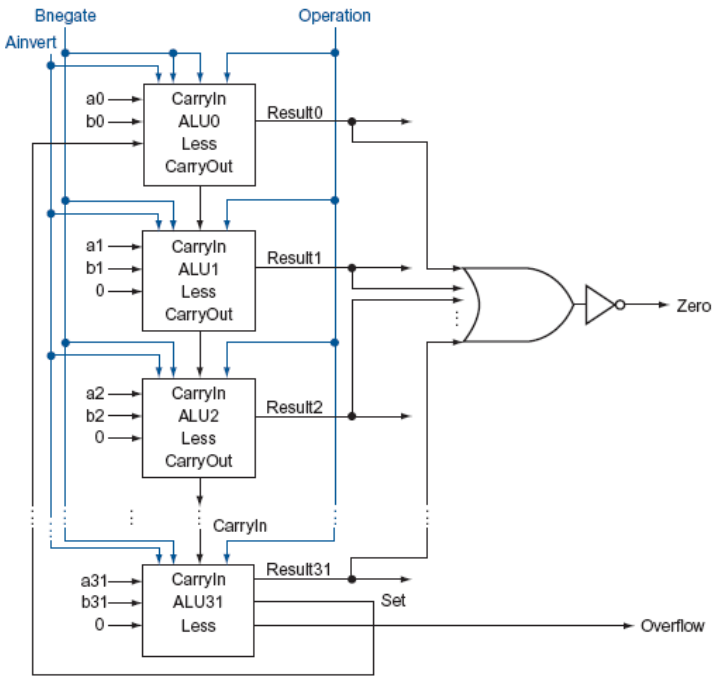
Computer Organization

Architecture diagrams:

Pipe_CPU_1.v



ALU.v



Hardware module analysis:

硬體分析

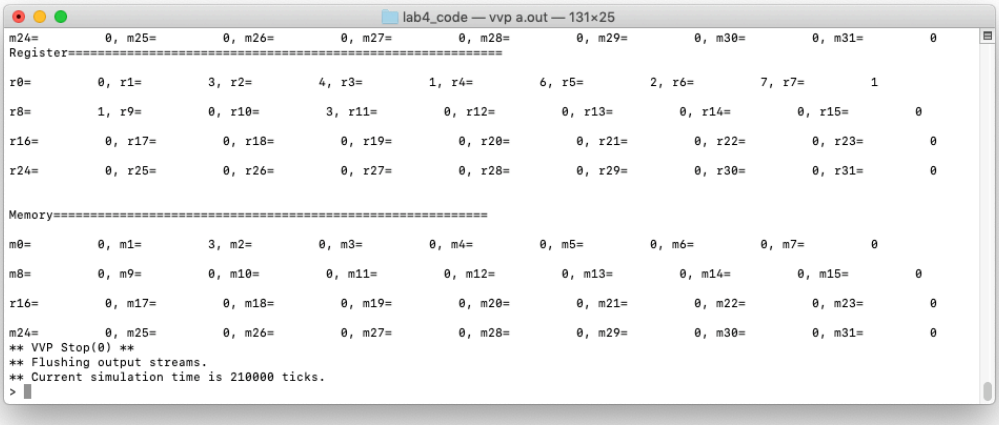
- **module Pipe_CPU_1 (clk_i, rst_i)**
  - **module Pipe_Reg (clk_i, rst_i, data_i, data_o)**
    本次實驗的重點，功能是區隔不同階段（IF、ID、EX、MEM、WB）之間的訊號，在每一個 clock cycle 把前一個階段的訊號儲存起來，並在下一個 clock cycle 來臨時送往下一個階段
  - **module ProgramCounter (clk_i, rst_i, pc_in_i, pc_out_o)**
  - **module Adder (src1_i, src2_i, sum_o)**
    有兩個，一個是 PC 的 Adder（PC <- 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)**
    有四個，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

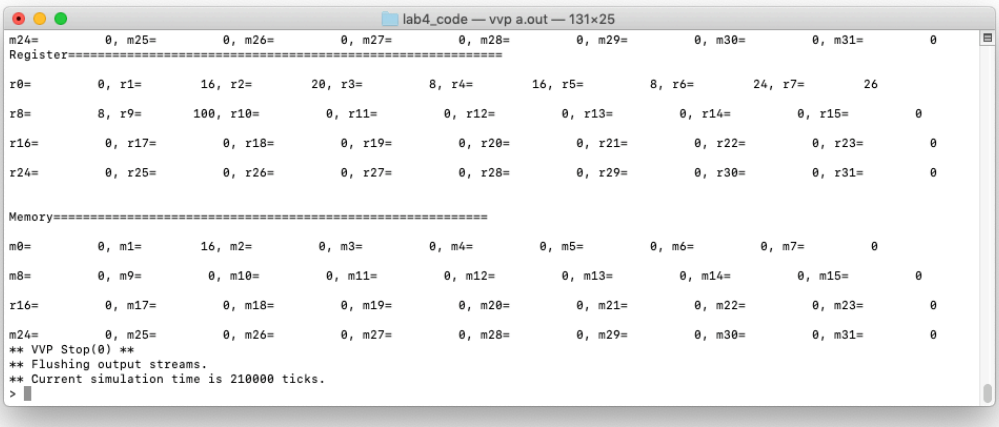
- **Pipe_CPU_1.v：Pipelined CPU 的主要架構**
  - clk_i：clock 訊號
  - rst_i：reset 訊號
- **Pipe_Reg.v：區隔不同階段（IF、ID、EX、MEM、WB）之間的訊號，每隔一個 clock cycle 更新一次**
  - clk_i：clock 訊號
  - rst_i：reset 訊號
  - data_i：在當前的 clock cycle 輸入（下個階段需要的資料以及控制訊號）並儲存在暫存器
  - data_o：下個 clock cycle 輸出（資料以及控制訊號）到下一階段
- **ProgramCounter.v：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:



CO_P4_test_data1.txt 的實驗結果



CO_P4_test_data2.txt 的實驗結果（解決 data hazard 後的結果）

兩題做出來的結果都跟助教提供的解答相同。

Problems you met and solutions:

這次的實驗相較於之前其實算是相對輕鬆的，因為這次助教提供的架構圖跟實際要完成的東西是一模一樣的，只要仔細觀察並瞭解老師上課講義上的內容就可以完成。因為這次是 Pipelined CPU 的架構，同時有最多五個指令在執行（IF、ID、EX、MEM、WB 五個階段），但也因此會發生 data hazard 的問題，例如第二個測資若沒有考慮到 data hazard 的問題便會產生錯誤的結果（r2=4、r5=-8、r8=0，也就是 I2、I6、I9 要寫回的暫存器），我解決的方法是透過在 I1、I2 之間，I5、I6 之間，I8、I9 之間個加入兩個 NOP 指令（修改後的

machine code 放在最下面），其實這是最偷懶的方式，如果不想拖慢 CPU 的執行速度，可以試著調換指令的順序（在不影響結果的前提下），或是可以加入 forwarding 的硬體來達成。

Summary:

這次實驗的重點是加入了 Pipe Register 的硬體架構，讓 CPU 可以同時執行數個指令（之前的 Single Cycle CPU 同一時間只能執行一個指令），加快 CPU 的執行速度，讓我驚豔的是硬體的架構並不複雜，只是單純的在不同的 clock cycle 輸入輸出資料而已，簡單又有效地提高了電腦運行的速度。這次實驗最主要的工作就是把線接好，因為要區隔不同階段的訊號，之前原本會跨過多個階段的線路就要加以分別，因為數量不算少，算是比較需要注意的地方，否則一不小心手殘接錯要花很多時間除錯呢。

Machine Code for Test Data 2:

00100000000000010000000000010000 // I1: addi **$1**, $0, 16
00000000000000000000000000000000 // NOP
00000000000000000000000000000000 // NOP
00100000001000100000000000000100 // I2: addi $2, **$1**, 4
00100000000000110000000000001000 // I3: addi $3, $0, 8
10101100000000010000000000000100 // I4: sw $1, 4($0)
10001100000001000000000000000100 // I5: lw **$4**, 4($0)
00000000000000000000000000000000 // NOP
00000000000000000000000000000000 // NOP
00000000100000110010100000100010 // I6: sub $5, **$4**, $3
00000000011000010011000000100000 // I7: add $6, $3, $1
00100000001001110000000000001010 // I8: addi **$7**, $1, 10
00000000000000000000000000000000 // NOP
00000000000000000000000000000000 // NOP
00000000111000110100000000100100 // I9: and $8, **$7**, $3
00100000000010010000000001100100 // I10: addi $9, $0, 100