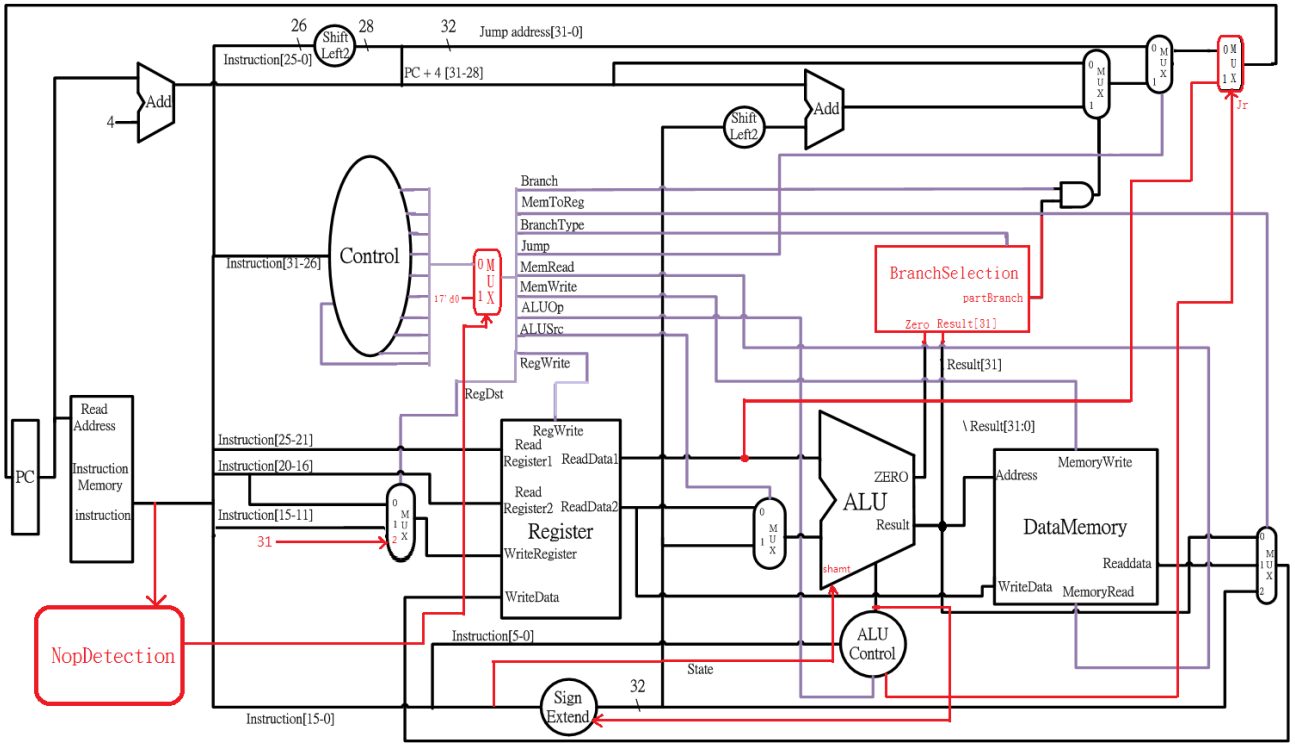
Computer Organization

HDL simulator:

Vivado

Architecture diagram:



Detailed description of the implementation:

這次的 Lab 是基於 Lab2 的簡單版 single cycle cpu 再加上 momeory unit 來實作一個較完整的 single cycle cpu。

藍字部分主要為 Lab2 修改或增加的功能。

Simple_Single_CPU

主要是由外部的 clk_i 與 rst_i 來控制內部的運作，可以藉由目前的 Program Counter 來取得 Instruction Memory，並經由 Decoder 發送控制訊號，經過一連串的選取後可以使得暫存器、Program Counter 以及 data memory 內部的值改變。

內部主要由以下模組構成:

Program Counter

由 clk、rst 與 pc_in 作為輸入決定 pc_o，每當 clk 正緣觸發時會將 pc_in 傳給 pc_o。pc_o 存放的是 Instruction Memory 的地址。

Instruction_Memory

輸入為目前要執行的指令地址，輸出則是 32-bit 的指令，而指令是根據地址從測資的 txt 檔所取得，一次會取得 32-bit。

Reg_File

內部存放 32 個暫存器，有四個輸入，兩個輸出，輸入有目前讀取的暫存器的地址*2，與寫入的暫存器地址與資料，輸出則是讀取暫存器的資料*2。

NopDetection

若指令中全為 0 的話就是 Nop，此時輸出訊號 Nop 會等於 1。

Decoder(Control)

根據指令中的 Op code(instruction[31:26])進行解碼，並依據不同類型的指令輸出控制訊號給其他模組。

- ✓ RegWrite：依據指令決定是否寫入 Register_File。
- ✓ ALU_op: 可依據不同指令初步判定要讓 ALU 執行加法或是減法等等的運算，輸出 3-bit 的 control 給 ALU_Crtl。
- ✓ ALUsrc: 用來決定要執行運算的是 Register 或是 Immediate。
- ✓ RegDst: 用來決定是要寫入的暫存器地址是 rd(指令為 R-type)還是 rt(指令為 I-type)。
- ✓ Branch: 若指令為 beq 或 bne 則輸出 1，其餘則輸出 0。
- ✓ Jump: 指令為 j 或 jal 時輸出 1，其餘為 0。
- ✓ MemRead: 指令為 lw 時為 1，其餘為 0。
- ✓ MemWrite: 指令為 sw 時為 1，其餘為 0。
- ✓ MemToReg: 指令為 lw 時為 1，jal 時為 2，其餘為 0。
- ✓ BranchType: 指令為 beq 時為 1，bne、bnez 時為 2，ble 時為 3，bltz 時為 4，其餘為 0。

ALU_crtl

根據 ALU_op 所輸出的訊號與指令中的 Function field(instruction[5:0])並輸出控制訊號給 ALU。

- ✓ 由於 Jr 是 R type，因此要看 Function filed 是否為 jr 來輸出控制訊號給 Mux_PC_Source_Third。

Extend

將 immediate (instruction[15:0])的值進行 extension，要注意的地方是只有 ori 是進行 zero-extension 其他的都是 sign-extension，所以使用 ALU_op 作為控制訊號。

ALU

將 src1、src2 以及 shamt 做輸入以 ALU_crtl 的訊號作為控制，使 ALU 可以對 src1 與 src2 執行各種運算，並將結果送回 Register File，zero output 則是 beq、bne 會使用到。

可以執行以下運算：

and, or, add, mul, jr, sub, slt, li, nor, sra, srav

MUX

2-to-1 MUX。

- ✓ Mux_DecoderSelection: 若 Nop 為 1 的話，表示不更改任何 memory 以及 register，因此會輸出讓所有的控制訊號都為 0，若為 0 則照原本 Decoder 輸出的控制訊號。
- ✓ Mux_ALU_src: 選擇作為 ALU_Src2 的是要用 rt 或是 immediate。
- ✓ Mux_PC_src: 將 PC_src 做第一次的選擇，判斷是要用 pc_next(PC+4) 或是 pc_branch 作為下一個的 Program Counter，控制線則是 Branch && part_Branch，輸出為 nextOrBranch。
- ✓ Mux_PC_Source_Second: 將 PC_src 做第二次的選擇判斷，是否要用 nextOrBranch 或是{pc_next[31:28], part_Jump[27:0]}，輸出為 JumpOrNot。
  - ➢ Jump 指令中的地址前 4-bit 是由 pc_next[31:28]而來。
- ✓ Mux_PC_Source_Third: 將 PC_src 做第三次的選擇判斷，決定最後的 PC 是要用 JumpOrNot 或是 ALU_src1(rs)，控制線為 Jr(是否為 jr)。
  - ➢ 因為 jr 指令是將 rs 內所存的地址作為下一個要執行的指令地址。

3-to-1 MUX

- ✓ Mux_Write_reg: 選擇要寫入的 Register 的地址是 rt、rd 或是$ra(reg[31])。
  - ➢ 因為在 jal 指令時需要將 PC+4 存入$ra(return address)
- ✓ Mux_MemToReg: 選擇要寫入 Register 的資料，分別是 ALU_result、MemData 或是 pc_next。
  - ➢ MemData 只有在 lw 時會被選擇。
  - ➢ pc_next 只有在指令為 jal 時會被選擇。

Adder

用到兩個 Adder。

- ✓ 計算 PC+4。
- ✓ immediate*4+PC+4。

Shift_Left_Two_32

- ✓ Shifter: 將 immediate 做 extension 的結果乘以四，beq 或 bne 的指令可能會用到其結果。
  - ➢ 因為 immediate 的部分是以 PC+4 做 offset 的部分(單位為 word)，因此要*4 變成 byte address。
- ✓ Jump_shifter: 將{6'd0, instr[25:0]}*4，j 指令會用到其結果。
  - ➢ 因為 j 與 jal 指令中[25:0]是存目前的 PC 區塊要跳到的第幾個指令(word address)，所以要*4 變成 byte address。

Branch_selection(BranchBlock)

將 ALU 執行完的 zero 與 result 作為輸入判斷 branch 是否 taken。

- ✓ beq: zero 為 1(相減為 0)

- ✓ bne、bnez: zero 為 0 (相減不為 0)
- ✓ ble: zero 為 1(相減為 0)或是 resultBit31 為 1(表示負數->小於)
- ✓ bltz: resultBit31 為 1(表示負數->小於)

Problems encountered and solutions:

有時候 simulation 時預設的時間較小，所以我們要的結果還沒跑到就停止了，會不小心以為 code 寫錯了。

在這次的 Lab3 中發現，一開始沒有發現 li 和 lui 的 opcode，以為是不同的 opcode，所以只有在後面增加 li，後來才發現 li 的 opcode 和 lui(Lab2)相同，不過我們還是保留 lui 因為不知道以後會不會用到，不過因為它寫在前面的 if (lui)，所以結果會被後面的 if (li)更改 。

在測第三個測資時，由於是要我們自己將 MIPS code 轉為 machine code，因此在翻譯的時候不小心翻錯了，有兩個是 register 的編號忘記更改，其他則是將 branch 指令的 offset 都寫成 jump 那種從上面數下來是第幾個指令，而不是以下一個指令為基準的 offset，因此 debug 花了非常久的時間，還好後來有發現並做修正。

Lesson learnt (if any):

這次的實作比 Lab2 多了一些功能，也使我們更熟悉 MIPS 的指令，也練習到如何運用 branch 以及讀寫 Data memory，對於 Jump 也更加了解，對上課所學到的知識有更多的領悟。