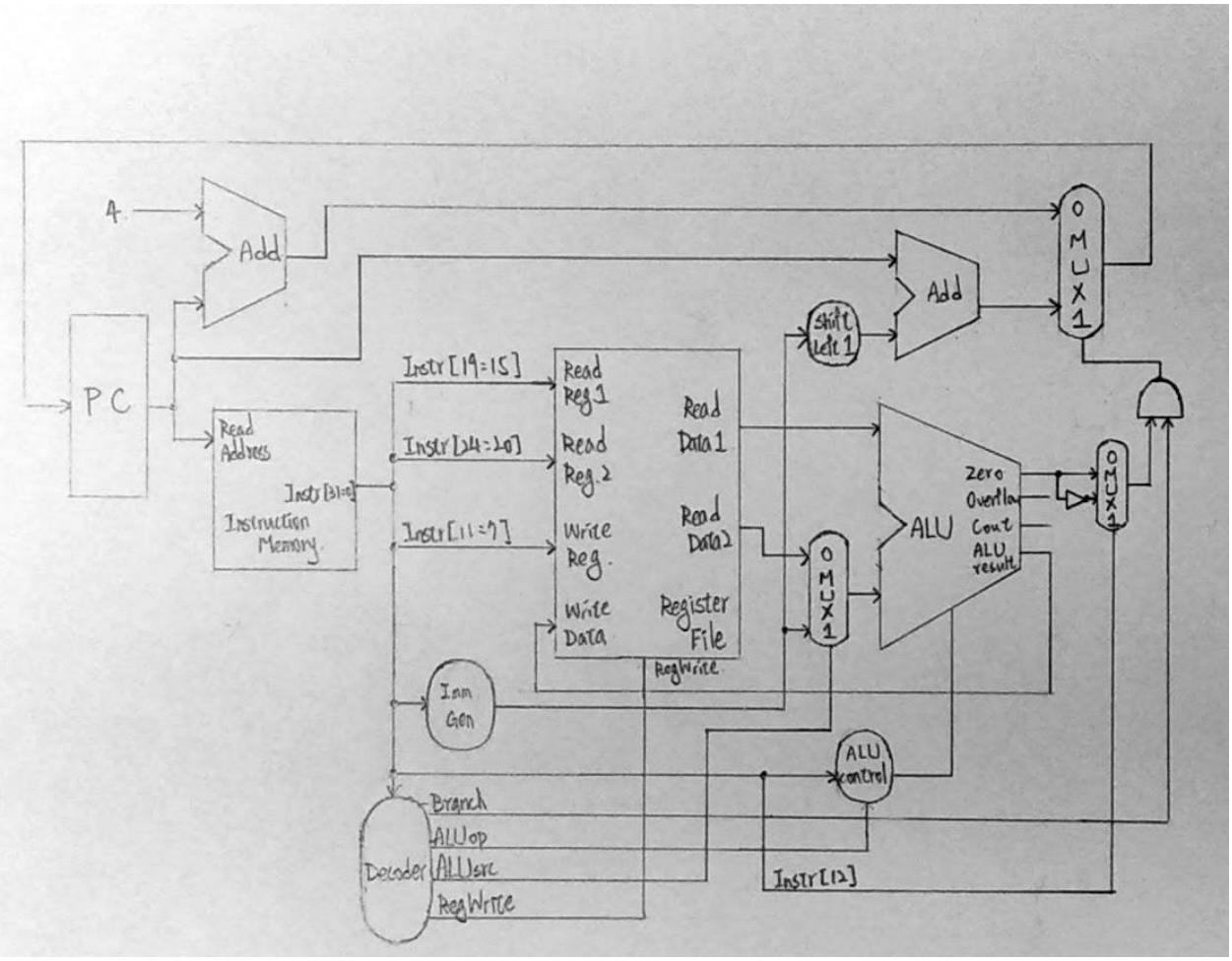
Computer Organization

0716214 江岳勳
0716222 黃偉傑

Architecture diagram:



Detailed description of the implementation:

- Adder.v

  直接將 sum_o assign 成 src1_i + src2_i

- ALU_Ctrl.v

  這個 module 的 input 有 instr[30, 14-12]（即為{funct7[5], funct3}）跟 ALUOp（來自 Decoder）；output 為連接到 ALU 的 wire：ALU_Ctrl_o。

  在 coding 上，我們使用三元運算子來 assign 輸出值。

  在邏輯上按照兩步驟：

  1. ALUOp 能用來區分 ld 和 sd(00), B-type(01), R-type(10), 除 ld 外之 I-type(11)，於是我們先判斷 instruction type。

2. funct3 和 funct7 能在我們確定 instruction field 後，進一步確認實際的 instruction，而前面已透過 ALUOp 確認過 Instruction Field，便可以在列出真值表後直接查表將對應的指令給與其 ALU_ctrl。

在寫出每種 instruction 的 case 之後，就照 ALU Design 時所定義來賦值。

- alu.v

  這個 module 的 input 有兩個 src（分別來自 Reg_File 和 MUX_2to1）、ALU_control（來自 ALU_Ctrl）；任務是根據 ALU_control 對兩個 src 進行不同操作。

  以下是我們 Design 時定義的 ALU_control 與相應操作：

ALU_Ctrl	Operation	描述
4'b0000	AND	根據 slides
4'b0001	OR	根據 slides
4'b0010	ADD	根據 slides
4'b0011	XOR	自定義
4'b0100	SLL	自定義
4'b0101	SRA	自定義
4'b0110	SUB	根據 slides
4'b0111	SLT	根據 slides
4'b1100	NOR	根據 slides
4'b1101	NAND	根據 slides
4'b1111	SRLI	自定義
4'b1000,1001,1010,1011,1110	UNUSED	

- Imm_Gen.v

  從 instr 先抽出 opcode 和 funct3 後，查表根據相對應的 type 來 assign 正確的 wire 到 Imm_Gen_o：

  I-type：opcode 為 7'b0010011 或 7'b0000011 或 (7'b1100111 且 funct3 == 3'b000)

  S-type：opcode 為 7'b0100011

  B-type：opcode 為 7'b1100011

  U-type：opcode 為 7'b0110111 或 7'b0010111

  J-type：opcode 為 7'b1101111

- MUX_2to1.v

  ```
  assign data_o = (select_i == 1'b0) ? data0_i : data1_i ;
  ```

- Shift_Left_1.v

assign data_o = {data_i[30:0] , 1'b0} ;

- Simple_Single_CPU.v

  將 component 間的 wire 都照 diagram 接上便完成。

  注意 PC_src 在計算時要判斷 branch 指令為 beq 還是 bne（Branch 為 true 的話，檢查 funct3[0]是否為 1，若為 1 則須在 AND 之前將 ZERO 先 NOT）

- Decoder.v (更動)

  因為原本的寫法會造成 ALUCtrl 無法判別 addi 和 sub，所以我們把 I-type 的 ALUop 改成了 2'b11

Implementation results:

CO_test_data1:

CO_test_data2:

CO_test_data3:

Problems encountered and solutions:

1. 一開始做 ALU_Ctrl 的時候，spec 裡面只有 ld, sd, beq, and, or, add, sub 的 ALUOp，而不確定其他 instruction 的 ALUOp。
   - 在研究 Decoder 的內容後，將 case 對照 opcode 和 funct3 建出真值表，查表便能夠確定 ALUOp 的對應。
2. 原本 Decoder 輸出的 I-type 和 R-type 的 ALUOp 相同，導致在 funct3 相同的情況下會有些 Instruction 無法辨認，例如 addi 和 sub，當 addi 在 instr[30]的位置為 1 時，ALU_Ctrl 會無法辨認 addi 跟 sub，因為 sub 在同一個位置是 funct7[5]，且該 bit 為 1。
   - 在討論區問過問題後，助教給的回答是將 I-type 的 ALUop 設為 2'b11，如此便能夠在 ALU_Ctrl 中判斷現在執行的到底是 addi 還是 sub。
3. 在 ALU Design 中，若按照上課時 ALU Control 的邏輯意義，我們無法定義如 XOR 等部分 instruction。
   - 自己定義這些 instruction 的 ALU control，如上面 description of the implementation 欄位中的表格所示。
4. 我們發現在寫 case 的時候若判斷某個 bit 是否為 dont-care (x)，答案會錯，改掉就對了。
   - 在判斷 ALU control 時，在三元運算子中的條件若是 (instr[3:0]==4'bx010)的話輸出會有奇怪的 unknown 值，改成(instr[2:0]==3'b010)就正常了，我們推測應該是 Verilog 本身模擬時如果碰到這種寫法產生的問題。
5. 在寫 Imm_Gen 的時候不知道要怎麼分辨各個 type 的 instruction 。
   - 在研究過 Decoder 的寫法和在網路上找 RISC-V 的指令 opcode 後，便順利的根據 opcode 建出所有 type 的立即值了。
6. 在接 Top module (Simple Single CPU) 的時候，因為判斷 beq 和 bne 跳的根據是相反的（beq 是 ZERO 為 1 時跳，bne 則相反），這樣子不知道要怎麼判斷。
   - 我們原本的作法是給 bne 另外一個 ALU control，在 ALU 中一樣做減法，可是在最後會將 ZERO 取 NOT，這樣在 Top module 一樣只需要一個 AND gate 就可以了。可是仔細想想覺得這樣好像違背了 ALU ZERO 的設計想法，於是我們便將 bne 的判斷移到 Top module 中，在 AND gate 前先檢查是否為 bne，若是的話再將 ZERO 取 NOT。
7. 在 Debug 的時候頭很痛。
   - 我們打開 testbench 看了要怎麼輸出訊息到 log 中，模仿了之後 debug 就變輕鬆很多。

Comment:

0716214 江岳勳：

這次的作業做的時候遇到蠻多問題的，因為是第一次的協作作業，多人一起做雖然會做得更快，但也意味著 debug 會變得更加困難。還好組員超級給力，基本上兩個人就是坐在隔壁一起做這份作業，加上我們使用了 Git 來做版本控制，使得這份作業的難度直接下降了好幾個等級。

作業本身並不會很難，將真值表建出來，根據真值表的內容將線拉好，基本上就沒什麼問題了，主要是拿到作業時還有一些問題存在(Decoder、addi 跟 sub 分不出來)，因為這些問題讓我們頭痛了有點久。我們也記取了教訓，知道有些訊息只會出現在討論區，所以即使我們做的時候沒有遇到什麼大問題，我們還是仔細的看過了討論區，看到大家都發現了些什麼問題，也因此節省了我們不少時間。

0716222 黃偉傑：

這次 Lab 的 Structure 和需要實作的內容都比上次要複雜，而且有許多需要不斷查表、容易出錯的地方，幸虧這次 Lab 是 Teamwork，讓 coding 的 loading 不算太高。儘管如此，我們還是花了許多力氣才終於得到正確的 Output，我認為這歸因於 Scale 較大，而難以針對單一 module debug。我認為這次 Lab 讓我學到最珍貴的是，雖然我們在課堂上看似已經將許多細節講得鉅細靡遺，但實際的 implementation 還是要 learning by doing。