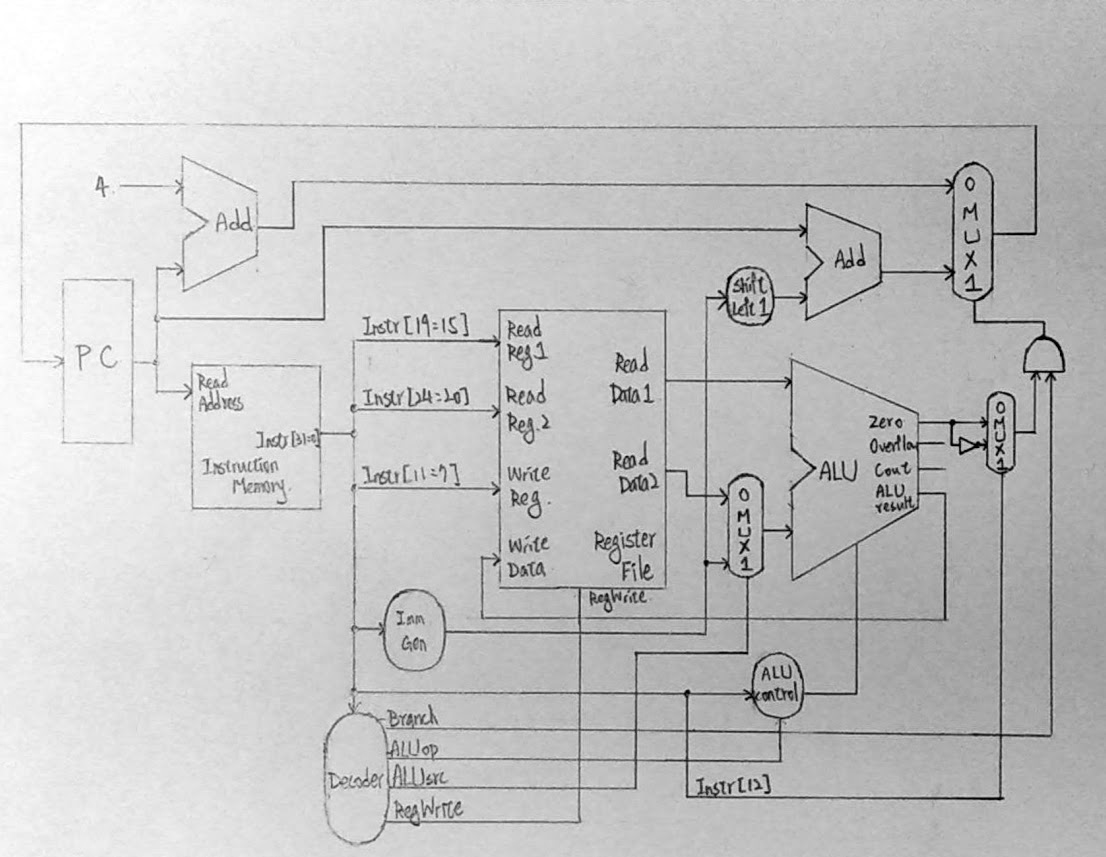
**Computer Organization**

**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。

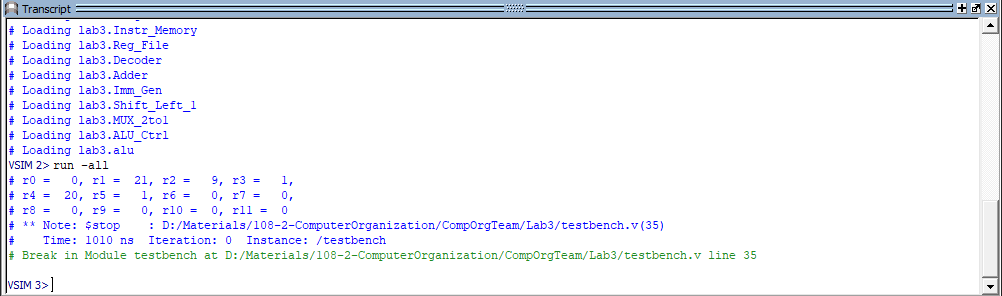
注意PC\_src在計算時要判斷branch指令為beq還是bne（Branch為true的話，檢查funct3[0]是否為1，若為1則須在AND之前將ZERO先NOT）

* Decoder.v (更動)

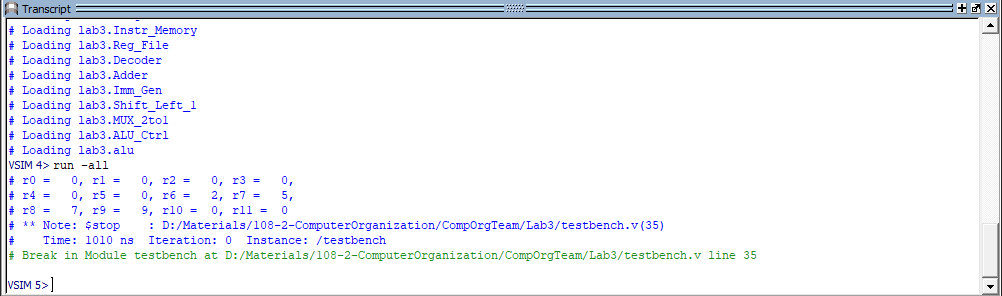
因為原本的寫法會造成ALUCtrl無法判別addi和sub，所以我們把I-type的ALUop改成了2’b11

**Implementation results:**

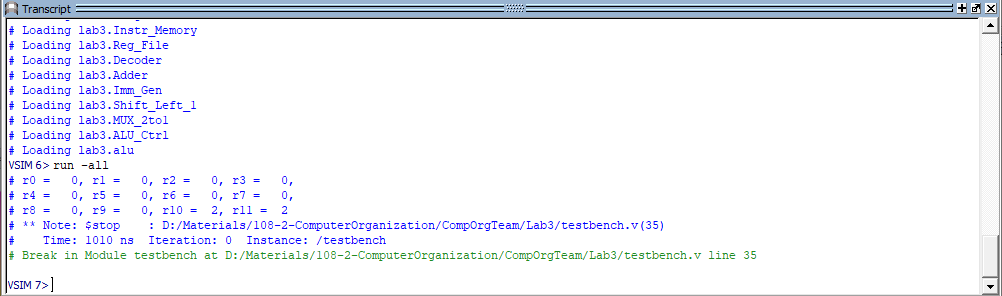
**CO\_test\_data1:**

****

**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。

1. 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。

1. 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。

1. 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。

1. 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。

1. 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。

1. 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。