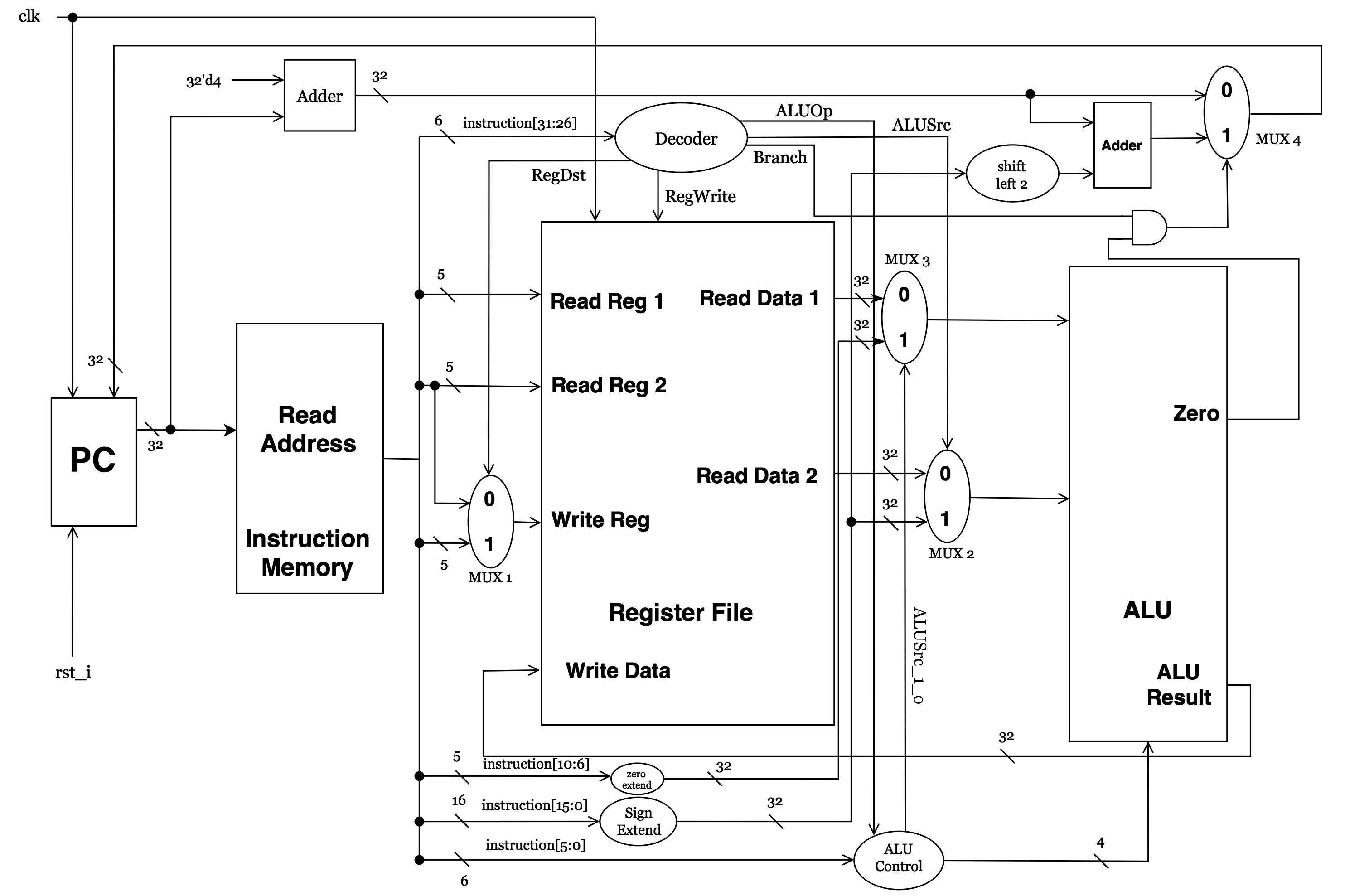
0416315 王定偉、0416005 張彧豪

**Computer Organization**

**Architecture diagram:**

****

**Detailed description of the implementation:**

1. 以Simple\_Single\_CPU為主要的模組，會去呼叫這個simple CPU所需要的所有模組，包含PC、IM、RF、Decoder、ALU\_Ctrl、Sign\_Extend、MUX，ALU等模組。此CPU為一個clock cycle執行一個指令。
2. Decoder模組：
3. 我是根據傳入的 Op filed去map那些輸出的值，以下是我採用的對應表：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| instr\_op\_i | ALU\_op\_o | Type | RegDst\_o | RegWrite\_o | ALUSrc\_o | Branch\_o |
| 6’d0 | 3’b000 | R | 1’b1 | 1’b1 | 1’b0 | 1’b0 |
| 6’d4 | 3’b001 | BEQ | 1’b0 | 1’b0 | 1’b0 | 1’b1 |
| 6’d5 | 3’b010 | BNE | 1’b0 | 1’b0 | 1’b0 | 1’b1 |
| 6’d8 | 3’b011 | I | 1’b0 | 1’b1 | 1’b1 | 1’b0 |
| 6’d13 | 3’b100 | ORI | 1’b0 | 1’b1 | 1’b1 | 1’b0 |
| 6’d15 | 3’b101 | LUI | 1’b0 | 1’b1 | 1’b1 | 1’b0 |

我們的ALU\_op\_o與instr\_op\_i之間並沒有什麼特殊的關係，純粹只是循序的指定。

1. ALU\_Control模組：
2. 因為有shamt要送進ALU的緣故，我們在這個模組有多加一個輸出序號ALUSrc\_1\_o，此訊號會傳到Read\_Data 1後方的mux當作選擇線。
3. 只有R-type的指令需要再次利用function filed判斷要給ALU什麼值。
4. 以下為我們採用的對應表：

R-type 以外：

|  |  |  |  |
| --- | --- | --- | --- |
| ALUOp\_i | Instruction | ALUSrc\_1\_o | ALUCtrl\_o |
| 3’b001 | BEQ | 1’b0 | 4’b0110 |
| 3’b010 | BNE | 1’b0 | 4’b1110 |
| 3’b011 | Addi | 1’b0 | 4’b0010 |
| 3’b100 | LUI | 1’b0 | 4’b0100 |
| 3’b101 | ORI | 1’b0 | 4’b0001 |

這五個指令除了BEQ、Addi的ALUCtrl\_o是拿lab1中減法、加法做對應，其它是拿尚未被佔用的指令去指派的。

R-type：

|  |  |  |  |
| --- | --- | --- | --- |
| ALUOp\_i | Instruction | ALUSrc\_1\_o | ALUCtrl\_o |
| 3’b000 | ADD | 1’b0 | 4’b0010 |
| SUB | 1’b0 | 4’b0110 |
| AND | 1’b0 | 4’b0000 |
| OR | 1’b0 | 4’b0001 |
| SLT | 1’b0 | 4’b0111 |
| SLTU | 1’b0 | 4’b1111 |
| SLL | 1’b1 | 4’b0011 |
| SLLV | 1’b0 | 4’b0011 |

SLL指令因為會用到shamt，所以它對應的ALUSrc\_1\_o會是1。

R-type中前五個指令是照lab1去指定的，而後面三個指令是拿尚未被佔用的值去指定的。

1. MUX 3：

MUX 3的出現主要是要處理SLL及SLLV運算，因為在這兩個運算中，要被shift的運算元皆由Read data2來，而shift amount則是從Read data1或是shamt來，為了不要多接一條線進去ALU單獨處理shamt，我們選擇在Read data1後方加一個2-to-1 mux，輸入為Read data1及shamt，利用ALU\_Control送出的ALUSrc\_1\_o作為選擇線，ALUSrc\_1\_o為0選Read data1作為輸出，反之則選zero-signed shamt作為輸出。

1. ALU模組：
2. 各種運算與對應的ctrl\_i表：

|  |  |
| --- | --- |
| Operation | ctrl\_i |
| AND | 4’b0000 |
| OR | 4’b0001 |
| ADD | 4’b0010 |
| SLL, SLLV | 4’b0011 |
| LUI | 4’b0100 |
| SUB, BEQ | 4’b0110 |
| SLT | 4’b0111 |
| BNE | 4’b1110 |
| SLTU | 4’b1111 |

1. 基本上跟上一次lab的ALU模組很像，不一樣的地方在於內部我們實作的是33-bit ALU，且判斷A\_invert的方法有稍微改變。
2. 為了實作SLTU，我們改實作33-bit的ALU，第33bit是根據ctrl\_i的值去指派，當ctrl\_i代表的SLTU時我們將第33bit皆設為0，其他狀況第33bit跟第32bit為同值，以此來實作SLTU。
3. 判斷A\_invert的方法我們改為若ctrl\_i最高位為1則A\_invert的訊號為1，反之A\_invert的訊號值跟ctrl\_i最高位相同，這樣改是因為要配合1-bit ALU的operation訊號(原本只用ctrl\_i最高位會導致SLTU跟BNE運算結果錯誤)。
4. 在Decoder把BEQ、BNE分開是因為ALU只有一個zero輸出，為了讓BEQ跟BNE能順利執行，在BEQ指令下，result為0則zero輸出為1，反之為0；在BNE指令下，result為0則zero輸出為0，反之為1，藉由這種方法讓Branch可以順利執行。

**Problems encountered and solutions:**

1. SLTU的實作方法，原本不太知道這個方法該怎麼利用lab1的程式碼去實作，想了一段時間後看到這次lab要用到的sign-extension，想到可以使用sign-extension的方法去把原本32-bit的運算元改成33bit且第33bit是根據不同的運算而決定的，不是SLTU的運算第33bit接跟第32bit一樣，而在SLTU運算下第33bit皆為0。
2. 原本想說在實作BEQ跟BNE的時候是不是要從ALU多接一條線出來，因為如果不接的話BNE好像無法實作，最後想到可以讓zero不是只在result為0的時候為1，在BNE的運算下，zero在result不為0時要為1，BEQ時則維持原狀，利用ctrl訊號去控制，這樣就可不必從ALU多接線出來另外做BNE的branch判斷。
3. SLTU及BNE原本的ctrl\_i是跟SLT及BEQ一樣，但這樣做的結果會導致結果錯誤，原因在第33bit的值及zero的輸出，故我們將SLTU及BNE單獨指派ctrl\_i，以利設定1-bit ALU的輸入訊號及zero輸出。
4. 我們將ALU改成像lab1一樣，只在reset為1時才運作，會這樣改是因為我們發現當第一條指令的運算與寫入的暫存器是一樣時會造成多做一次該指令運算的缺陷，觀察波形圖後發現只要讓ALU在reset為1的時運作即可避免此錯誤，故我們將rst\_i傳入ALU，並指定ALU只在rsi\_i是1時運作。

**Lesson learnt (if any):**

1. 如何適當的配對現有的運算與尚未被使用到的opcode。
2. SLTU的實作方法。