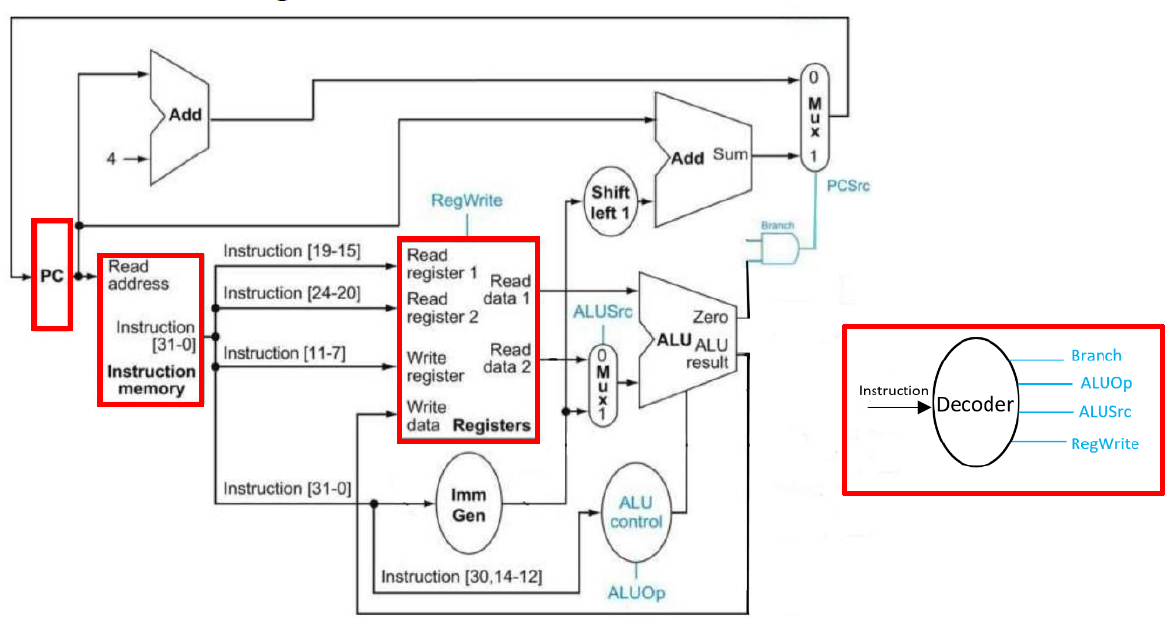
**Computer Organization**

**Architecture diagram:**

****

**Detailed description of the implementation:**

1. ALU

利用input的4-bit ALU\_control訊號，完成不同控制。

除了Lab2的7個基本指令，這次ALU還新增了4種新指令如下；

4’b0011: result = a – b // branch not equal 會進行減法

4’b1110: result = a ^ b // exclusive or

4’b1111: result = a << b // a shift left b-bit

4’b1001: result = a>>>b // a shift right arithmetic b-bit

Zero detection輸出訊號則較麻煩，當result == 0時，bne要把Zero設為0，其他都設為1；當result != 0時，bne的Zero要設為1，其餘設為0。我們這組利用ALU\_Ctrl == 4’b0011來判斷是否為bne。

補充說明，我們把input的src1和src2轉成signed number，這樣才能利用behavioral的方式進行shift left logical和shift right arithmetic的指令。

1. ALU\_Control

所有指令可以利用ALUop訊號，分成3種：load/store、Branch、ALU運算

1. load/store：**I-type(load) & S-type**

input：ALUop = 00，output：ALU\_Ctrl = 0010，因為load和store都在ALU裡面進行加法。

1. Branch：**beq & bne**

input：ALUop = 01，output：ALU\_Ctrl = 0010，因為所有的Branch指令在ALU裡面都進行減法。

1. 算術運算：**R-tpye & I-type**

input：ALUop = 10 && Funct3，利用ALUop和Funct3 ( instr[2:0] )，依據不同指令對ALU\_Ctrl給予不同訊號。

補充：instr[3] = instruction[30]；instr[2:0] = instruction[14:12] (=Funct3)

1. Imm\_Gen

面對的指令可分為I-type和B-type，我們在元件內部建立一個內部訊號Type，當指令是I-type時，Type設為1，其他設為0。

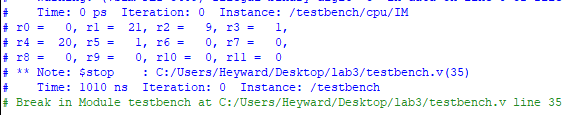
I-type指令只要把輸入的12-bit依序放到輸出Lsb的12-bit即可；

B-type因為指令的12-bit是拆散的，需要依照指令格式對應回來比較麻煩。

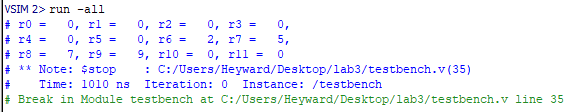
最後再把immd[31:12]依正負號判斷(instr[31])用sign-bit補滿即可。

**Implementation results:**

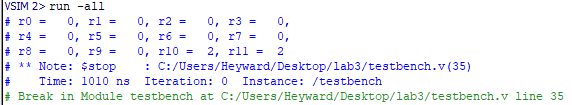
**Test\_data\_1**

****

**Test\_data\_2**



**Test\_data\_3**

****

**Problems encountered and solutions:**

一開始照著講義寫，忘記把xor, sll, sra等新的指令寫進ALU\_Ctrl裡面，測試半小時才發現忘記寫進去；Immediate Generator有兩種定址模式，我們也想了很久，才想到把進來的種類分成I-type和B-type，再依據不同情況把Immediate產生，至於正負號我們也爭論了一下到底要不要考慮，因為兩種type正負號的Sign-bit在instruction的不同位置，最後才想到先處理LSB的12-bit，再一口氣處理Sign-bit的問題。

最後測試時發現Decoder會把slti的ALUop設成00，幸好助教即時發布更新的Decoder讓我們免於Debug的地獄中。

**Comment:**

這次作業其實比上次簡單，因為助教幫我們把最難的Decoder寫完了(灑花)，我們只要把ALU新增一些指令，再把線接一接就大致完成了，而且在實作的過程中把Datapath和Controlpath弄得更清楚了，也對ALUop、ALU\_ctrl和ALU之間的關係更加明瞭。