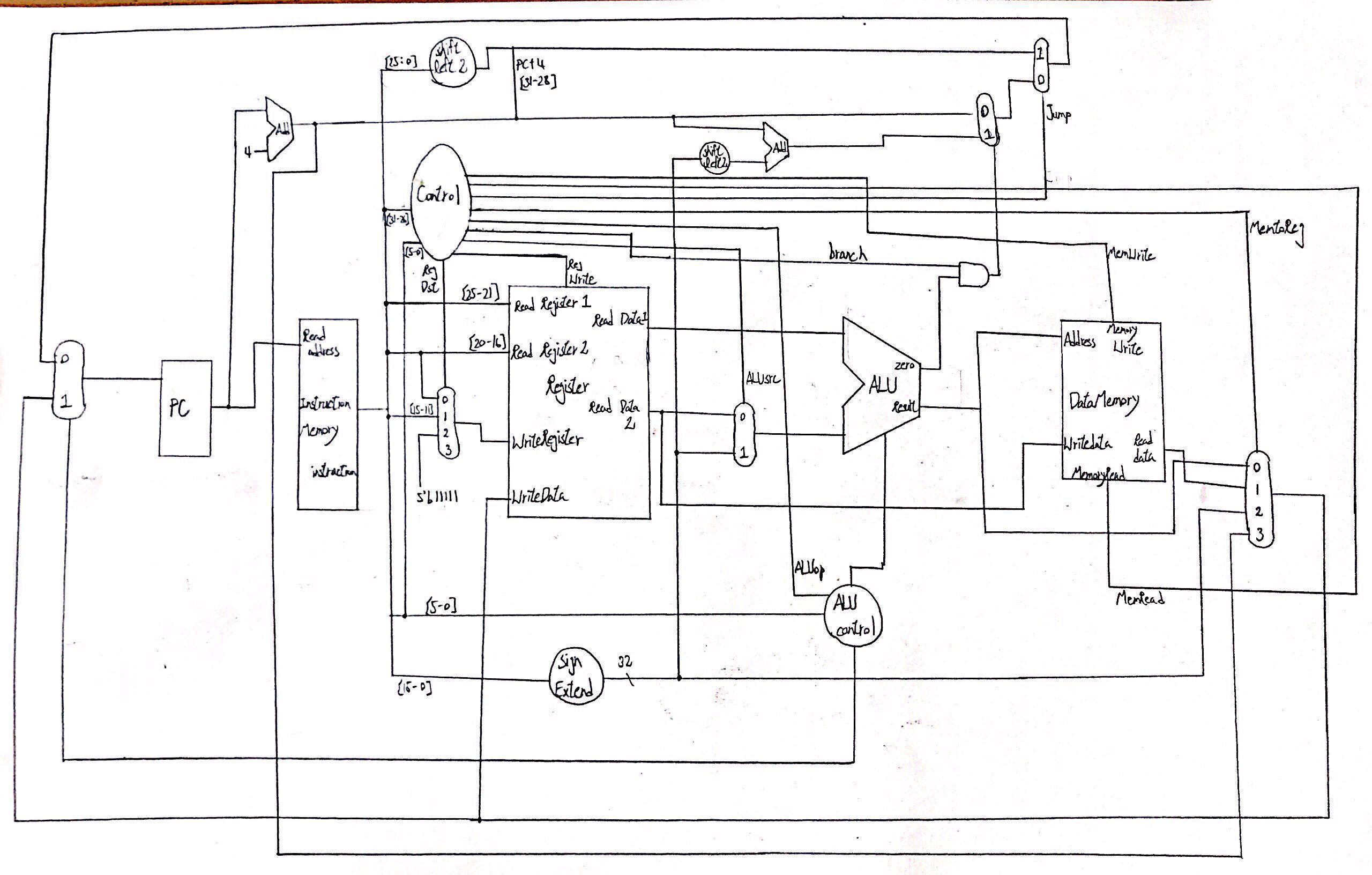
**Computer Organization**

**Architecture diagram:**

****

**Detailed description of the implementation:**

Lw : 從memory中取rs+imm位置的值存到register(rt)中, 需要ALU計算並read memory , 再write back回去register中。  
Sw : 把register(rt)的值存到memory中rs+imm的位置, 需要control signal去write memory ( stack pointer存在reg[29] )。

J : 讀opcode跟address, 讓PC = 該address即可 ( address須先做shift left 2, 並跟當前PC的前四位組合 )。

mul : 輸出兩數相乘, 用ALU做。

NOP : 什麼事都不做, decoder會視為R-type發出訊號, 但因為都是0, 所以不影響結果。

Jal : Jump到address ( 同上jump), 但需要把reg[31]存PC+4, 也就是下一條指令的位置, 讓之後可以把它存到memory, 以便之後可以跳回來。

Jr : R-type, 我們用ALU\_Ctrl去判斷這條指令, 並發出一個訊號讓PC = Reg[rs] ( 那個訊號在其他情況都是0, 只有jr才會是1, 因為他需要memory的結果卻沒有辦法讀到 )。

Ble : 用ALU判斷有沒有 <= , 有的話讓result = 0, zero就會等於1, 跟branch and 後, PC就會跳過幾行指令了。

Bnez : 跟bne一樣, 只是會讀到0, 若不等於0就讓result = 0,zero = 1, branch=1, 就會branch到後幾行指令了, 基本上不用更改。  
Bltz : 判斷rs的首bit是不是1, 是的話就是小於0, 就會讓result = 0,zero = 1, branch=1, 就會branch到後幾行指令了。

Li : decoder要判斷這種指令, 發出去的訊號都跟做addi一樣的即可(做的事也一樣, 就是+0的意思)。

**Problems encountered and solutions:**

一開始並不懂為什麼jal要存reg[31], 也不懂jal存那個到底要做什麼, 做完後實際在debug 一行一行看知道原理。  
 還有就是一開始在想ble, bltz的時候, 都想不到要怎麼讓PC branch, 後來只好用發訊號的方式決定, 但我們總覺得這方法好像有點糟糕, 不曉得有沒有更好的辦法。

最後就是mips轉machine code真的好累…, 希望之後都有machine code ~。

**Lesson learnt (if any):**

了解reg 跟 mem的交互作用, 更熟悉指令的運作原理, 也更熟悉verilog。