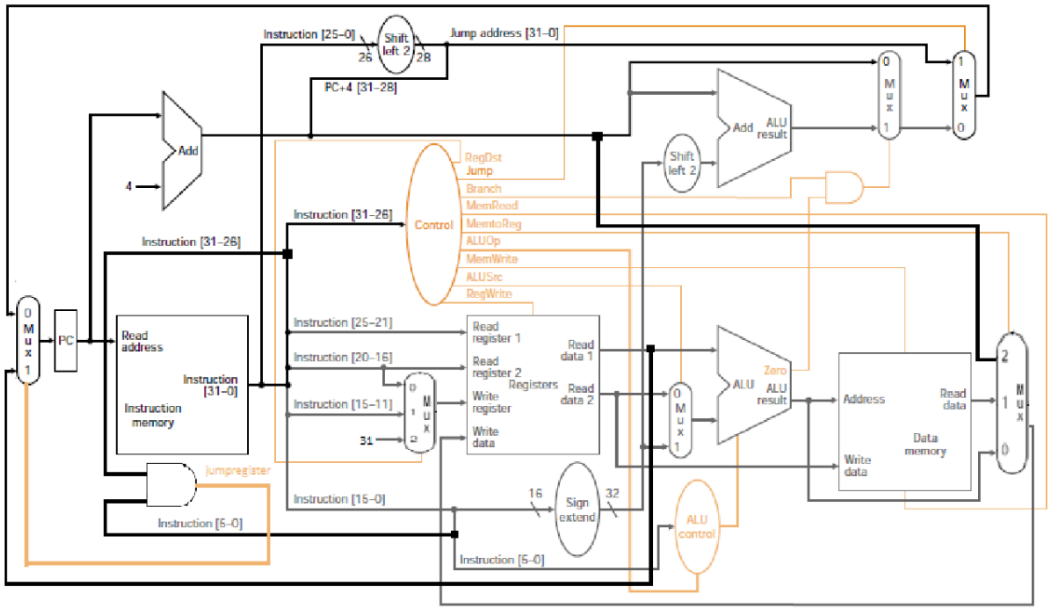
Computer Organization Lab3

Name:單宇晟

ID:109550087

Architecture diagrams:



Hardware module analysis:

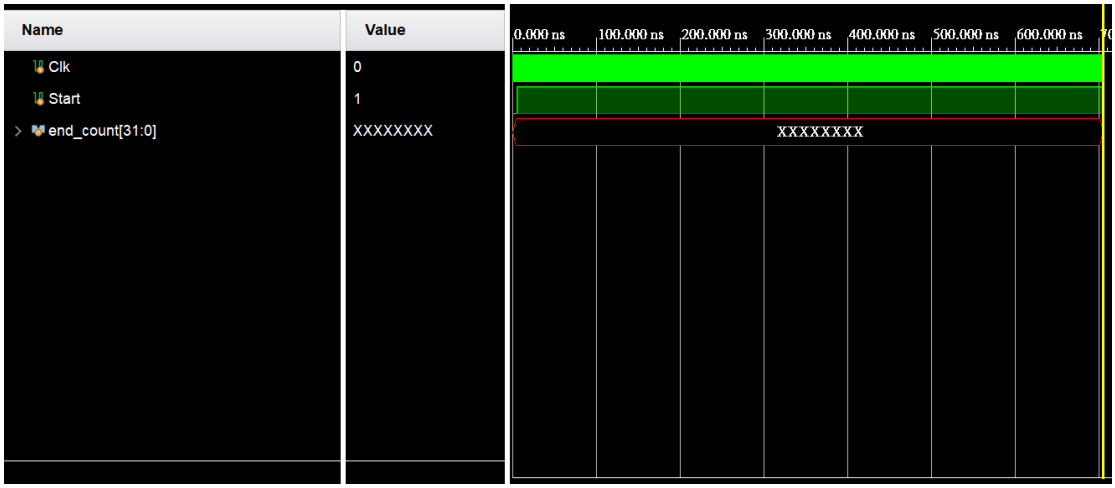
這次 diagram 的實作方法，主要可以參考 simple_single_CPU.v。首先，會用 PC 來提供指令位址，之後會計算出下個指令的位址(PC Add 4)，同時 instruction memory 會給出指令，並且將各位元分別傳送到指定的地方做運算，並且 decoder 也會給出控制訊號，這次我用到了 4 個 MUX2to1，分別判斷 jr, R/I-format, jump, branch 的情況，要注意的是因為這次並沒有實作其他的 branchtype(PDF 的圖上有)，所以右上方的 MUX 判斷的時候和上次一樣用 branch & zero 就好。

至於 lw 和 sw，他們運作的方式其實有點像，只是 lw 是對記憶體做讀取、sw 是儲存，此外，sw 也不會把記憶體的值寫回暫存器。

雖然 singlecycle 的結構較為簡單，但缺點是相比 multicycle，singlecycle 的 clk 週期必須是相同長度，而此長度會由電路裡最長的路徑來決定，因此效率比較不好，且我們無法在每個指令中使用不同數目的 clk。

Finished part:

```
PC =        128
Data Memory =        1,        2,        0,        0,        0,        0,        0,        0
Data Memory =        0,        0,        0,        0,        0,        0,        0,        0
Data Memory =        0,        0,        0,        0,        0,        0,        0,        0
Data Memory =        0,        0,        0,        0,        0,        0,        0,        0
Registers
R0 =        0, R1 =        1, R2 =        2, R3 =        3, R4 =        4, R5 =        5, R6 =        1, R7 =        2
R8 =        4, R9 =        2, R10 =        0, R11 =        0, R12 =        0, R13 =        0, R14 =        0, R15 =        0
R16 =        0, R17 =        0, R18 =        0, R19 =        0, R20 =        0, R21 =        0, R22 =        0, R23 =        0
R24 =        0, R25 =        0, R26 =        0, R27 =        0, R28 =        0, R29 =        0, R30 =      128, R31 =        0
```



CO_P3_test_data1:

addi r1,r0,1 r1=1

addi r2,r0,2 r2=2

addi r3,r0,3 r3=3

addi r4,r0,4 r4=4

addi r5,r0,5 r5=5

這部分就是直觀的 addi，因此 r1, r2, r3, r4, r5 的值會分別是 1, 2, 3, 4, 5

jump j

addi r1,r0,31 r1=31

addi r2,r0,32 r2=32

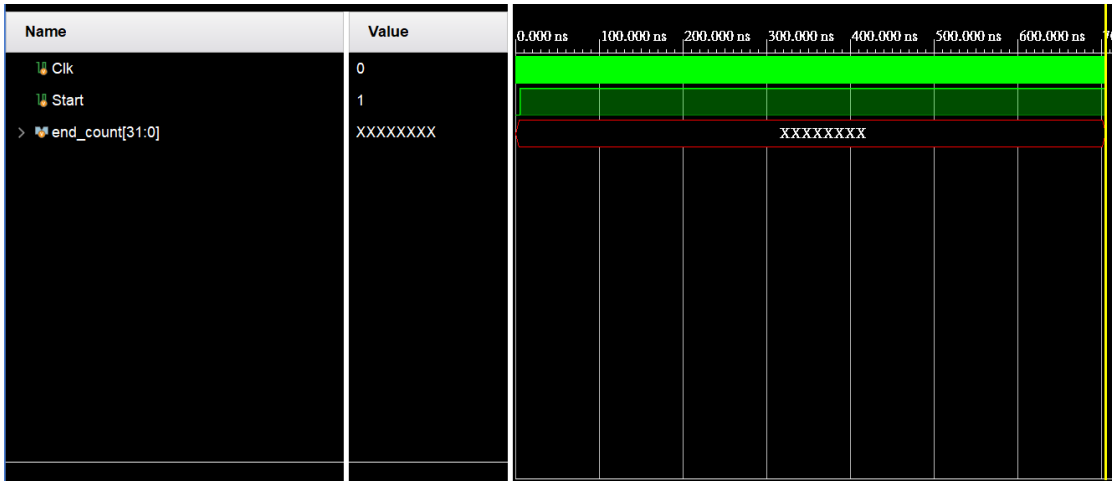
這裡因為 jump 執行了，所以 jump 下面的兩行會被跳過，直接去執行 j 的部分

j: …

這裡主要測試 lw, sw 的功能

CO_P3_test_data2:

```
PC =        128
Data Memory =        0,        0,        0,        0,        0,        0,        0,        0
Data Memory =        0,        0,        0,        0,        0,        0,        0,        0
Data Memory =        0,        0,        0,        0,       68,        2,        1,       68
Data Memory =        2,        1,       68,        4,        3,       16,        0,        0
Registers
R0 =        0, R1 =        0, R2 =        5, R3 =        0, R4 =        0, R5 =        0, R6 =        0, R7 =        0
R8 =        0, R9 =        1, R10 =        0, R11 =        0, R12 =        0, R13 =        0, R14 =        0, R15 =        0
R16 =        0, R17 =        0, R18 =        0, R19 =        0, R20 =        0, R21 =        0, R22 =        0, R23 =        0
R24 =        0, R25 =        0, R26 =        0, R27 =        0, R28 =        0, R29 =        0, R30 =      128, R31 =       16
```



Test2 的程式比較難按照步驟解釋，相比於 test1 他多測試了 jal 跟 jr 的功能是否正常運行，而依照程式邏輯，最後出來的 R2 就會是 5。

Problems you met and solutions:

這次 lab 相比於上次 lab，又多了一些 function，所以在 decoder 裡面又變得更複雜，也更多不同 case 要處理，我原本的寫法是用 if else 去寫，來 assign Jump_o, MemRead_o, MemWrite_o, MemtoReg_o 這些值，像是這樣:

```
ALUSrc_o = (instr_op_i == 6'b001000 || instr_op_i == 6'b001010 || instr_op_i == 6'b100011 || instr_op_i == 6'b101011)? 1'b1:1'b0;
RegWrite_o = ((instr_op_i == 6'b000000 && instr_func_i == 6'b001000) || instr_op_i == 6'b001000 || instr_op_i == 6'b001010
|| instr_op_i == 6'b000100 || instr_op_i == 6'b101011 || instr_op_i == 6'b000010)? 1'b0:1'b1; // jr beq addi slti sw jump
RegDst_o = (instr_op_i == 6'b000000 || instr_op_i == 6'b000011)? 1'b1:1'b0; // beq
Branch_o = (instr_op_i == 6'b000100)? 1'b1:1'b0; // beq

Jump_o = (instr_op_i == 6'b000011)? 2'b00: (instr_op_i == 6'b000000 && instr_func_i == 6'b001000)? 2'b10:2'b01; // jal jr
MemRead_o = (instr_op_i == 6'b100011)? 1'b1:1'b0; // lw
MemWrite_o = (instr_op_i == 6'b101011)? 1'b1:1'b0; // sw
MemtoReg_o = (instr_op_i == 6'b100011 || instr_op_i == 6'b101011)? 2'b01: (instr_op_i == 6'b000011)? 2'b10:2'b00; // lw sw
```

不過後來發現這樣如果不小心漏掉某個 case 或是打錯就會很難找到 bug，所以就改成了現在這種寫法:

```
6'b100011: begin // lw
    ALU_op_o <= 3'b000;
    ALUSrc_o <= 1;
    RegWrite_o <= 1;
    RegDst_o <= 0;
    Branch_o <= 0;
    Jump_o <= 0;
    MemRead_o <= 1;
    MemWrite_o <= 0;
    MemtoReg_o <= 1;
end
6'b101011: begin // sw
    ALU_op_o <= 3'b000;
    ALUSrc_o <= 1;
    RegWrite_o <= 0;
    // RegDst_o <= x;
    Branch_o <= 0;
    Jump_o <= 0;
    MemRead_o <= 0;
    MemWrite_o <= 1;
    // MemtoReg_o <= x;
end
```

把每個情況都分開寫，雖然這樣會讓我的 code 變很冗長，但卻比較方便

debug。另外一個問題倒是困擾我很久，就是 pdf 上給的 diagram 只是參考，並不像是上次 lab 那樣照著牽線就好，加上助教給的模板也沒有改過，一些要額外用到 module、或是要用到 mux3to1 的地方都沒有直接標明出來，要我們自己判斷，因此在我花在 simple_single_cpu 上的時間非常多，幸好最後我並沒有卡住，還是有順利的把每條線、每個 module 弄好，按時的完成了這次的 lab。

Summary:

這次 lab 主要是在上次的架構下加入一些新功能(lw, sw, jump, jal, jr)，所以大致上的架構其實都沒有變，實際上要改的檔案也比上次少很多。不過實際上還是花了我不少時間，正如上面提到的，這次的實作並不能完全照 PDF 上的圖做，要自己額外確認每條線的去向，同時還要知道那些額外的參數(Jump_o, MemRead_o, MemWrite_o, MemtoReg_o)在幹嘛以及他們的重要性，只要一個地方 assign 錯就可能全部都跑不出來，而這次更多了 DataMemory 這個 module。經過了這兩次 lab，我也終於完成了一個功能「稍微」完善的 single cycle CPU，之後應該就要做 pipeline CPU 了，希望我也能夠順利地做出來。