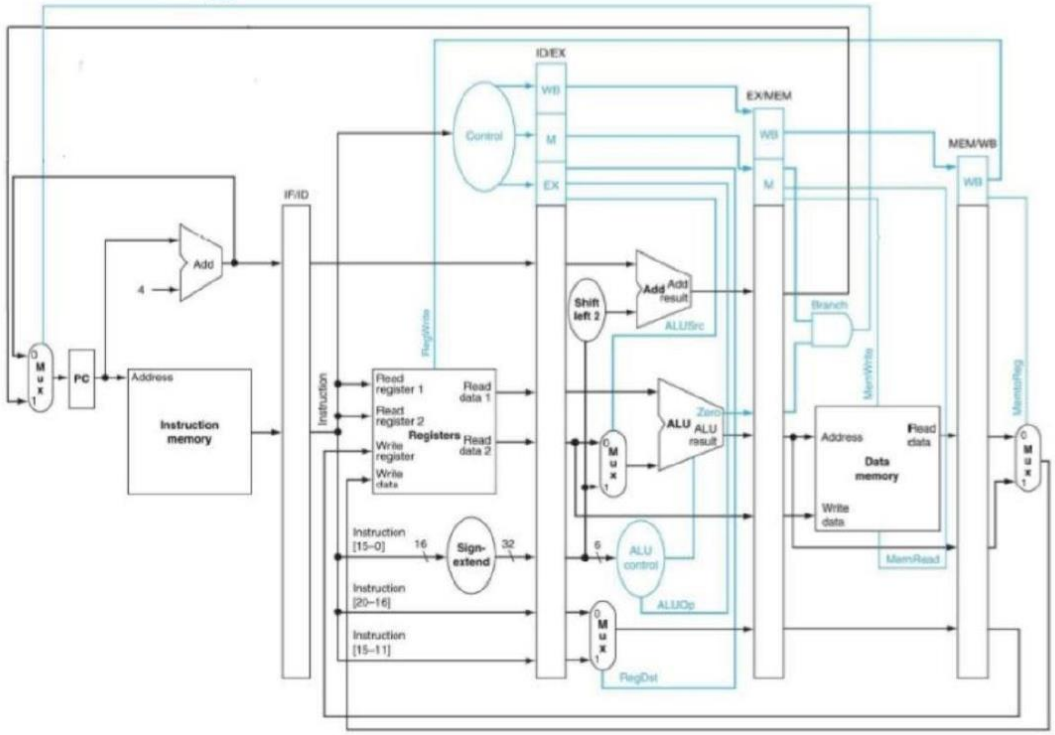
Computer Organization Lab4

Name:單宇晟

ID:109550087

Architecture diagrams:



Hardware module analysis:

和 lab3 的結構相似，差別在多了 Pipe_Reg 和 Pipe_CPU，以及 control 指令的傳輸方式。與之前的 lab 都不同的是，這次是要做 pipelined CPU，所以 Decoder(Control) 把指令傳出去的方式變得完全不一樣，我們需要先判斷各個指令會在哪個階段被用到，並一層層傳到某兩個階段間(IF, ID, EX, MEM, WB)的 Pipe_Reg，我認為這也是這次 lab 主要需要重新設計、理解的地方。而相比於前幾次的 lab，這次 lab 的優點在於效率更好。

Finished part:

Register==========

r0= 0, r1= 3, r2= 4, r3= 1, r4= 6, r5= 2, r6= 7, r7= 1

r8= 1, r9= 0, r10= 3, 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= 0, r31= 0

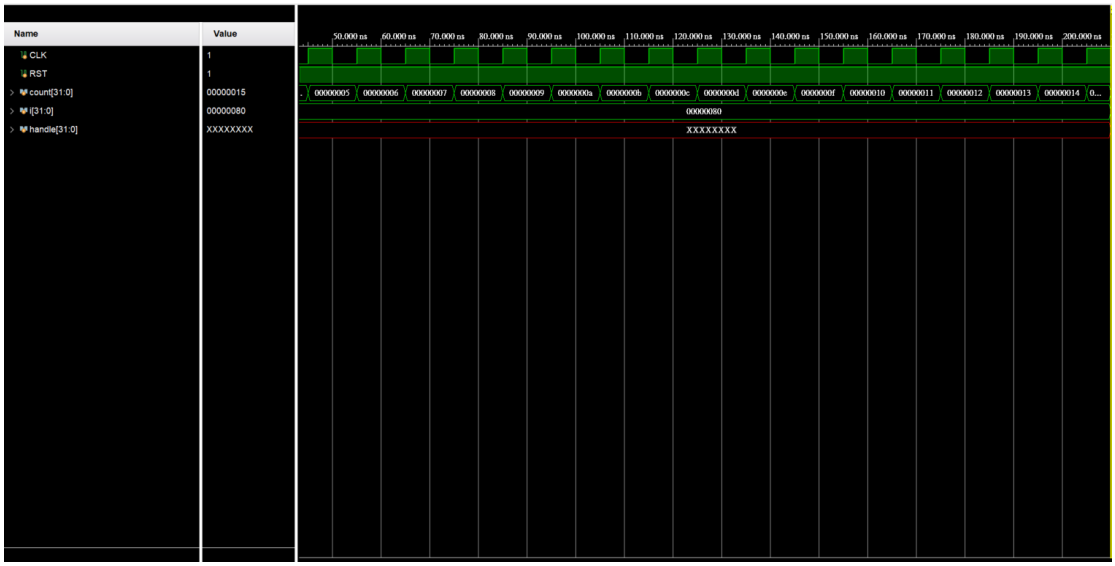
Memory==========

m0= 0, m1= 3, m2= 0, m3= 0, m4= 0, m5= 0, m6= 0, m7= 0

m8= 0, m9= 0, m10= 0, m11= 0, m12= 0, m13= 0, m14= 0, m15= 0

r16= 0, m17= 0, m18= 0, m19= 0, m20= 0, m21= 0, m22= 0, m23= 0

m24= 0, m25= 0, m26= 0, m27= 0, m28= 0, m29= 0, m30= 0, m31= 0



```
begin:
addi    $1,$0,3;        //  a = 3
addi    $2,$0,4;        //  b = 4
addi    $3,$0,1;        //  c = 1
sw      $1,4($0);       //  A[1] = 3
add     $4,$1,$1;       //  $4 = 2*a
or      $6,$1,$2;       //  e = a |    b
and     $7,$1,$3;       //  f = a &    c
sub     $5,$4,$2;       //  d = 2*a   - b
slt     $8,$1,$2;       //  g = a <    b
beq     $1,$2,begin
lw      $10,4($0);      //  i = A[1]
```

如圖所示，最後列出的結果與程式邏輯相符，有用到的 register 都有收到正確的值，而 memory 則也有達成 sw 指令所給出的命令(儲存 m1)。

Problems you met and solutions:

這次 lab 主要的問題是在於理解 pipe_reg 的功用，程式有很大一部份都可以沿用 lab3 的 code，甚至我認為最麻煩的 pipe_CPU 也有很多 module 的使用方式都和上次一樣。另外確實有發生一個小問題，就是助教給的 testbench 裡，載入 testcase 的路徑是"testbench/CO_P4_test_1.txt"，而我一開始並沒有注意到，導致我輸出的所有值都是 0，幸好最後有發現，重新改掉就解決了。

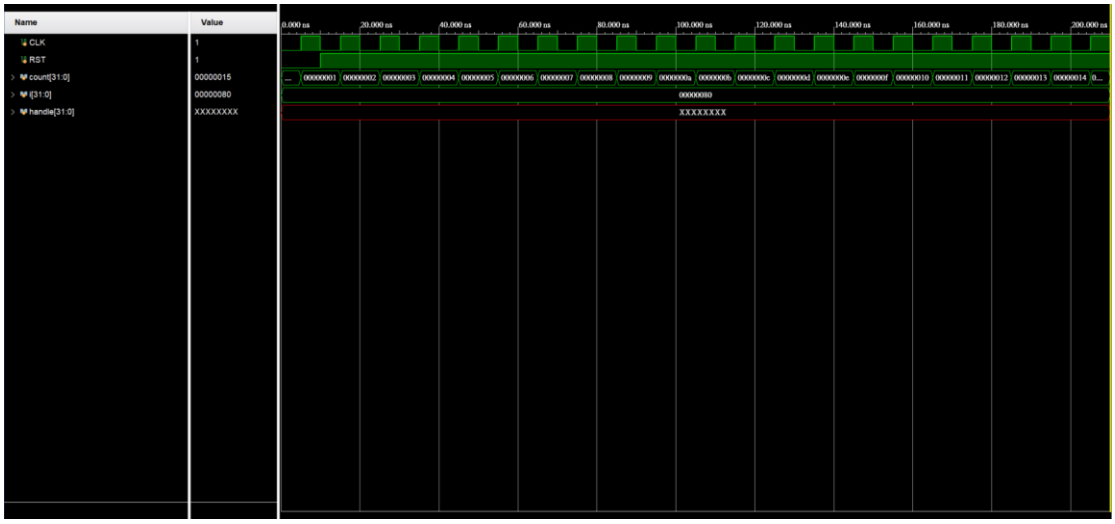
Bonus (optional):

```
Register==========================================================

r0=     0, r1=     16, r2=     20, r3=     8, r4=     16, r5=     8, r6=     24, r7=     26
r8=     8, r9=     100, 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=     0, r31=     0

Memory============================================================

m0=     0, m1=     16, m2=     0, m3=     0, m4=     0, m5=     0, m6=     0, m7=     0
m8=     0, m9=     0, m10=     0, m11=     0, m12=     0, m13=     0, m14=     0, m15=     0
r16=     0, m17=     0, m18=     0, m19=     0, m20=     0, m21=     0, m22=     0, m23=     0
m24=     0, m25=     0, m26=     0, m27=     0, m28=     0, m29=     0, m30=     0, m31=     0
```



Machine code:

00100000000000010000000000010000 // I1
00000000000000000000000000000000 // nop
00100000000000110000000000001000 // I3
00100000001000100000000000000100 // I2
10101100000000001000000000000100 // I4
10001100000001000000000000000100 // I5
00100000001001110000000000001010 // I8
00000000011000010011000000100000 // I7
00000000100000110010100000100010 // I6
00000000111000110100000000100100 // I9
00100000000010010000000001100100 // I10

為了解決 data hazard of I1/I2, I5/I6, and I8/I9 data dependency，我把 I2 和 I3 做交換，並在 I1 和 I3 中間加入一個 nop(bubble)；至於 I5/I6、I8/I9，我交換了 I6 和

I8 的執行順序，這樣 I6 和 I5 之間就至少可以有一個 cycle 的時間，I8 和 I9 也是。

Summary:

我認為這次的 lab 其實不難，主要注意各個階段的指令是要從 pipe_reg 去讀取的，而不是直接用 decoder 傳過去，另外，這次的 CPU 也只要照著 architecture diagrams 去牽線就好，比較麻煩的就只是要知道每個階段間的 pipe_reg 分別需要甚麼指令，才會有多餘的傳輸，也才能數對 pipe_reg 的 input signal 位數。