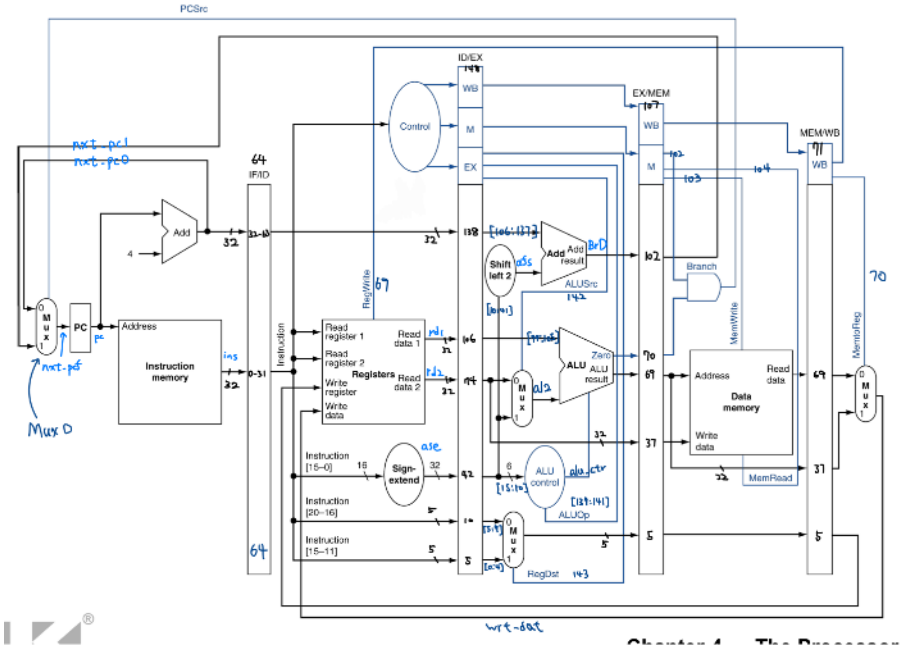
Computer Organization - Lab4 Pipelined CPU

Architecture diagrams



這次的 Pipelined CPU 我採用與課本相同的電路圖（如左圖），每個 Stage 中間的黑色數字為資料往下傳需要的 bits 數量，深藍色為它所在那個 Stage 的哪一個或那一區的 bits，而控制元件 Control 跟 ALU control 採用與前一次大致相同，把不需要用的指令拿掉簡化而已。

另外，在 MemToReg 進 MUX 前我有將訊號 inverse ，較符合直覺。

此外，在即將進入 ID/EX 的控制訊號以及 EX/MEM 的控制訊號，還有要進入 IF/ID 的指令，若現在在 MEM 階段的 Branch 要執行 (PCSrc = 1)，那要把這些控制訊號設為 0 ，指令設為全 0 (NOP)，這些指令才不會影響到運算的結果，

Hardware Module Analysis

- **Decoder**

```
always @(*) begin
    RegDst_o <= rfmt;

    RegWrite_o <= rfmt | addi | slti | lowd;
    Branch_o <= bieq;
    ALUSrc_o <= addi | slti | lowd | stwd;

    MemToReg_o <= lowd;
    MemRead_o <= lowd;
    MemWrite_o <= stwd;
end
```

這次的 **Decoder** 我採用與之前相同的的方法，不同的 **Instruction AND** 起來做區別，再將不同的控制訊號 **OR** 起來，達到正確的 **control** 訊號。

- **ALU control**

這次多的 **mult** 一樣屬於 **R–format** ，唯一不同的是後面要給 ALU 的控制訊號要是乘法，所以我在 **ALU control** 把乘法的 **function format** 獨立出來，並傳 ALU 所沒有使用到的編碼當作乘法，這邊我是用 3(0011)。

```
if (funct_i == 5'b11000) begin
    ALUCtrl_o <= 4'b0011;
```

Problem Met & Solution

這次有要實作 mult 乘法指令，但是沒有 testcase 可以試，所以我把 testcase1 裡面的 $4 = $1 + $1 改成：

addi $9 $0 2
mult $4 $1 $2

–> 經過實測如 **Result** 裡面 Testcase1–2 除了 $9，與答案符合，符合預期。

這次有要實作 beq branch 指令，一開使我沒有想到要怎麼把中間的三個指令清為 nop ，後來才想到如果把所有中間的 control 訊號設為 0 ，不管他會怎麼運算，他都不會影響到我的記憶體結果，所以我將所有訊號 & (~Branch) 。

Result

Testcase 1

```
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
m16= 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
```

Testcase 1-2

```
Use mult(r9 = 2)
Register================================================================
 r0= 0,  r1= 3,  r2= 4,  r3= 1,  r4= 6,  r5= 2,  r6= 7,  r7= 1
 r8= 1,  r9= 2, 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
m16= 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
```

Testcase 2

```
I1:   addi  $1,$0,16   )data hazard → 2NOP
I2:   addi  $2,$1,4    )
I3:   addi  $3,$0,8
I4:   sw    $1,4($0)
I5:   lw    $4,4($0)   ) data hazard → 2NOP
I6:   sub   $5,$4,$3   )
I7:   add   $6,$3,$1
I8:   addi  $7,$1,10   )data hazard → 2NOP
I9:   and   $8,$7,$3   )
I10:  addi  $9,$0,100
```

```
1  00100000000000010000000000010000
2  00000000000000000000000000000000
3  00000000000000000000000000000000
4  00100000010001000000000000000100
5  00100000000000110000000000001000
6  10101100000000010000000000000100
7  10001100000001000000000000000100
8  00000000000000000000000000000000
9  00000000000000000000000000000000
10 00000001000001100101000000100010
11 00000000011000010011000000100000
12 00100000001001110000000000001010
13 00000000000000000000000000000000
14 00000000000000000000000000000000
15 00000000111000110100000000100100
16 00100000000010010000000001100100
```

在每個 Data Hazard 中，插入兩個 nop operation ，即可以讓前個指令有足夠的 stage 可以在下個指令前執行完，即能達到正確的計算結果。

```
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
```

Summary

在寫這份作業之前，我已經複習過 Pipeline CPU 的上課部分，但實際寫才發現其實還有一些細節自己在看書的時候沒有想清楚，所以藉由這次作業，不僅加深對於 Pipeline 的運作模式，也更清楚了之前沒注意到的小細節（比如說 Branch)。