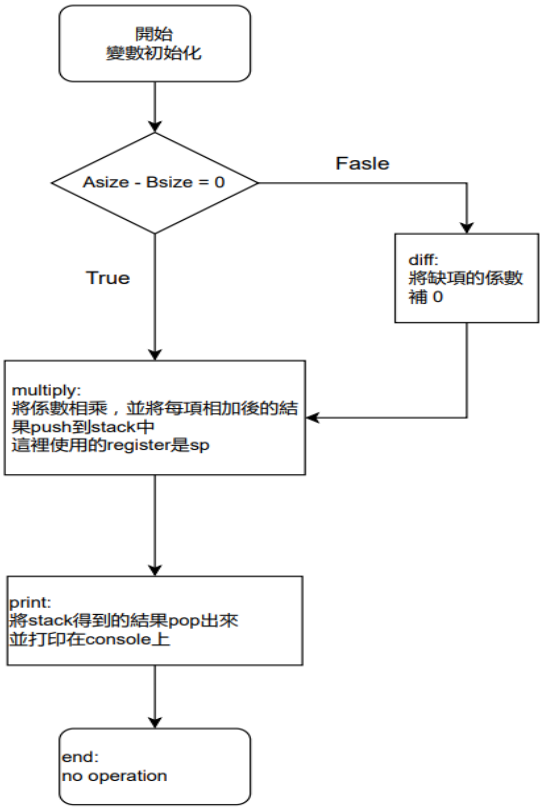
Computer Architecture HW5

110061596 程朝略

1. Assembly Coding :

   a. Flow Chart :



想法 ：

i. 首先，先將參數初始化，將 PolynomialA, PolynomialB 的 address load 到 register t0, t1 中。也將 Asize, Bsize load 到 register a0, a1 中

ii. 將 a0 與 a1 相減，判斷兩個 size 是否為相等，不相等則將缺項的係數補 0

iii. 將 PolynomialA 及 PolynomiaB 的係數 load 到 registers s1-s8 中，並根據多項式相乘，將各次方相同的係數相加，存回到 register sp 的 stack 當中。(這裡有更聰明簡潔的辦法，但因為想不到所以用暴力一點的方法)

iv. 將存回到 sp register 中的結果 pop 出來，並利用迴圈去 ecall 出各次方係數。過程中，需要將係數或字串先 load 到 a0 中，(由最低次方到最高次方)

v. 程式結束

b. Correct relust :

for **Test pattern 1** and **Test pattern 2**

Console

```
Product polynomial is 1 + 3x^1 + 9x^2 + 15x^3 + 29x^4 + 43x^5 + 20x^6

Product polynomial is 5 + 10x^1 + 30x^2 + 26x^3 + 52x^4 + 24x^5
```

c. Source code :

Initialize and the the main of program:

```
.data
.align 4
# test pattern 1
# polynomialA: .word 5, 0, 10, 6
# polynomialB: .word 1, 2, 4
# ASize: .word 4
# BSize: .word 3
# test pattern 2
polynomialA: .word 1, 3, 7, 4
polynomialB: .word 1, 0, 2, 5
ASize: .word 4
BSize: .word 4
str1: .string "Product polynomial is "
str2: .string "x^"
str3: .string " + "
```

```
.text
.global _start
# Start coding in the section below, don't change the values above #
_start:
    la t0, polynomialA  # loading address
    la t1, polynomialB
    lw a0, ASize        # loading size
    lw a1, BSize
    add a2, a0, a1

    mv s11, a2      # move the degree to s11

    addi a3, a3, 4

    sub a4, a0, a1  # len diff

    la a0, str1
    li a7, 4
    ecall

    bne a4, zero, diff   # for checking the lenght of two polynomials

    jal multiply
    jal print

    j end
```

multiply :

```
multiply:
    sub sp, sp, a2   # pop 7 positions to store a0 - a7 and return address
    sw ra, 0(sp)     # return address

    lw s1, 0(t0)     # for polynomial 1
    lw s2, 4(t0)
    lw s3, 8(t0)
    lw s4, 12(t0)

    lw s5, 0(t1)     # for polynomial 2
    lw s6, 4(t1)
    lw s7, 8(t1)
    lw s8, 12(t1)

    mul a0, s1, s5   # deg 0
    sw  a0, 4(sp)
    mul a1, s1, s6   # deg 1
    mul a2, s2, s5
    add a1, a1, a2
    sw  a1, 8(sp)
    mul a2, s1, s7   # deg 2
    mul a3, s2, s6
    add a2, a2, a3
    mul a3, s3, s5
    add a2, a2, a3
    sw  a2, 12(sp)
    mul a3, s1, s8   # deg 3
    mul a4, s2, s7
    add a3, a3, a4
    mul a4, s3, s6
    mul a5, s4, s5
    add a4, a4, a5
```

```
    add a3, a3, a4
    sw  a3, 16(sp)
    mul a4, s2, s8   # deg 4
    mul a5, s3, s7
    add a4, a4, a5
    mul a5, s4, s6
    add a4, a4, a5
    sw  a4, 20(sp)
    mul a5, s3, s8   # deg 5
    mul a6, s4, s7
    add a5, a5, a6
    sw  a5, 24(sp)
    mul a6, s4, s8   # deg 6
    sw  a6, 28(sp)
    jr x1
```

diff :

```
diff:                # for checking the lenght of two polynomials
    mul a5, a3, a1   # a3 = 12
    add a3, a5, t1
    sw zero, 0(a3)
    jal multiply
```

print:

```
print:
    addi sp, sp, 4    # where to start pop stack
    addi a1, zero, 1  # index
    addi s11, s11, -1   # termination index
    li a7, 1
    ecall

    loop:
        addi sp, sp, 4  # next element to pop

        la a0, str3  # "+"
        li a7, 4
        ecall

        lw a0, 0(sp) # "coefficient"
        li a7, 1
        ecall

        la a0, str2  # "x"
        li a7, 4
        ecall

        li a7, 1     # "power of deg"
        mv a0, a1
        ecall

        addi a1, a1, 1  # increment the element by 1
        bne a1, s11, loop
```

end:

```
end: nop
```

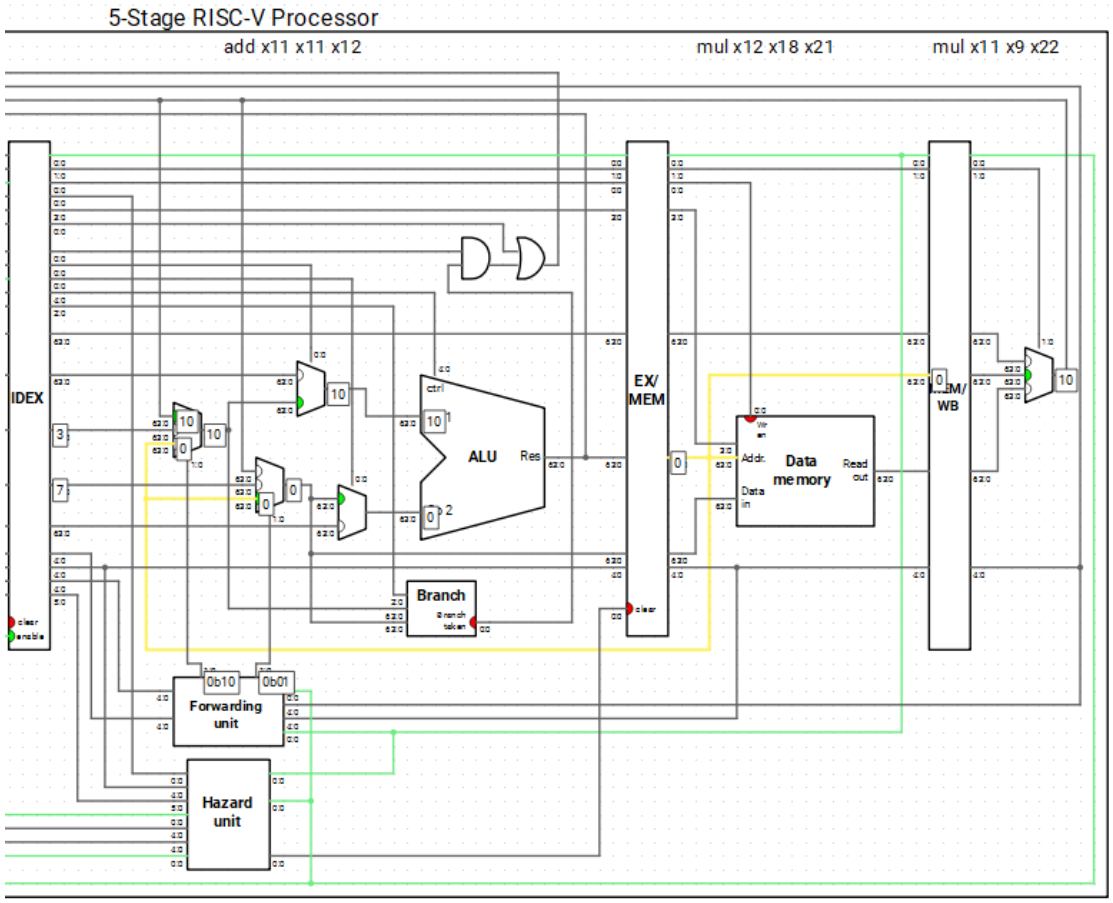
(以上為 Assembly code 部分)

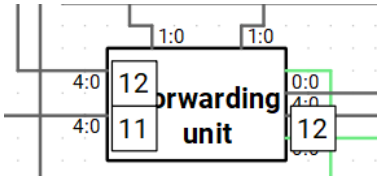
2. Hazard in your code :

(1) R-type RAW at following 1st instruction :

```
68      mul a2, s2, s5
69      add a1, a1, a2
```

The register a2 is used to store the result of multiplying s2 and s5 in the first instruction (line 68). However, a2 is also used in the following instruction add as rs2 (line 69).





(EX stage register number : rs1 = 11, rs2 = 12)

(MEM stage register number : rd = 12)

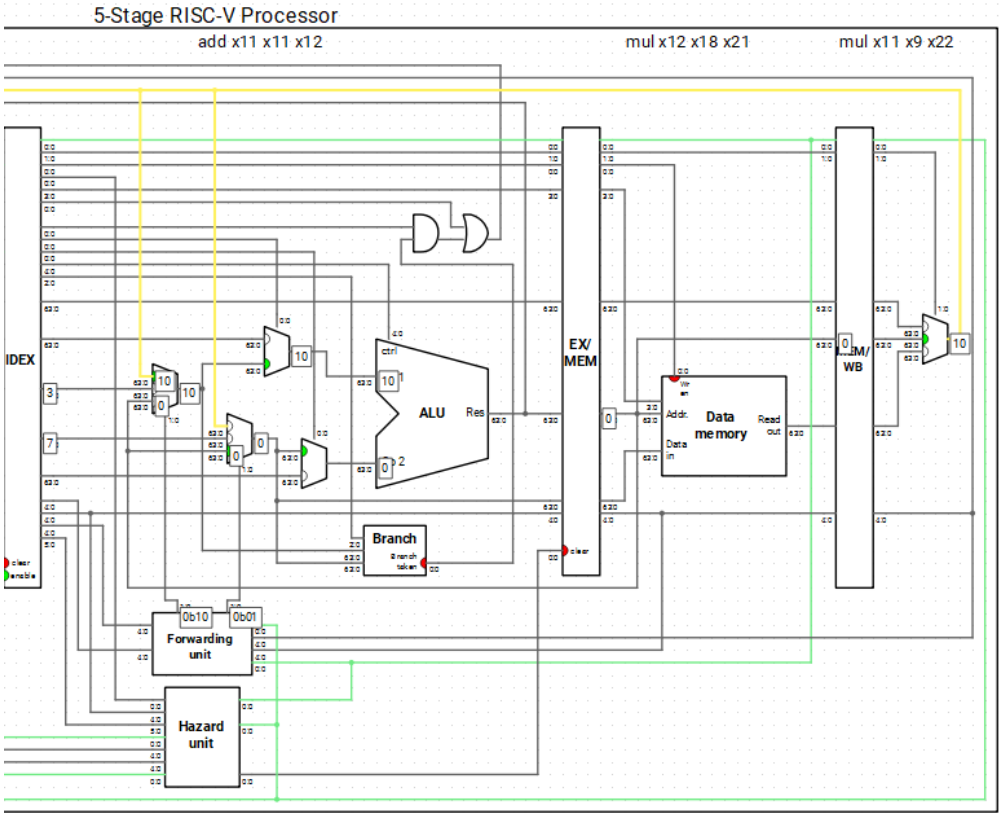
Explain :

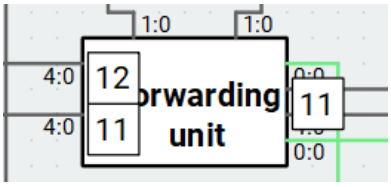
mul 指令(MEM stage)的 rd 與下一行 add 指令(EX stage)的 rs2 兩者 register number 一樣，都為 x12。此時，processor 偵測到兩指令間有 data dependency，因此將 mul 指令計算完 x12 的值= 0，利用 forwarding unit 產生控制訊號 0b01，將 mul 指令計算完 x12 的值設定為 add 指令 rs2 的值。所以，add 指令中 rs2 的值為 mul 指令計算完成後的值 ＝0，而非本身在 x12 中的值 ＝7

(2) R-type RAW at following 2nd instruction :

```
67      mul a1, s1, s6
68      mul a2, s2, s5
69      add a1, a1, a2
```

The register a1 is used to store the result of multiplying s1 and s6 in the first instruction (line 67). However, a1 is also used in the third instruction add as rs1 (line 69).





(EX stage register number : rs1 = 11, rs2 = 12)

(WB stage register number : rd = 11)

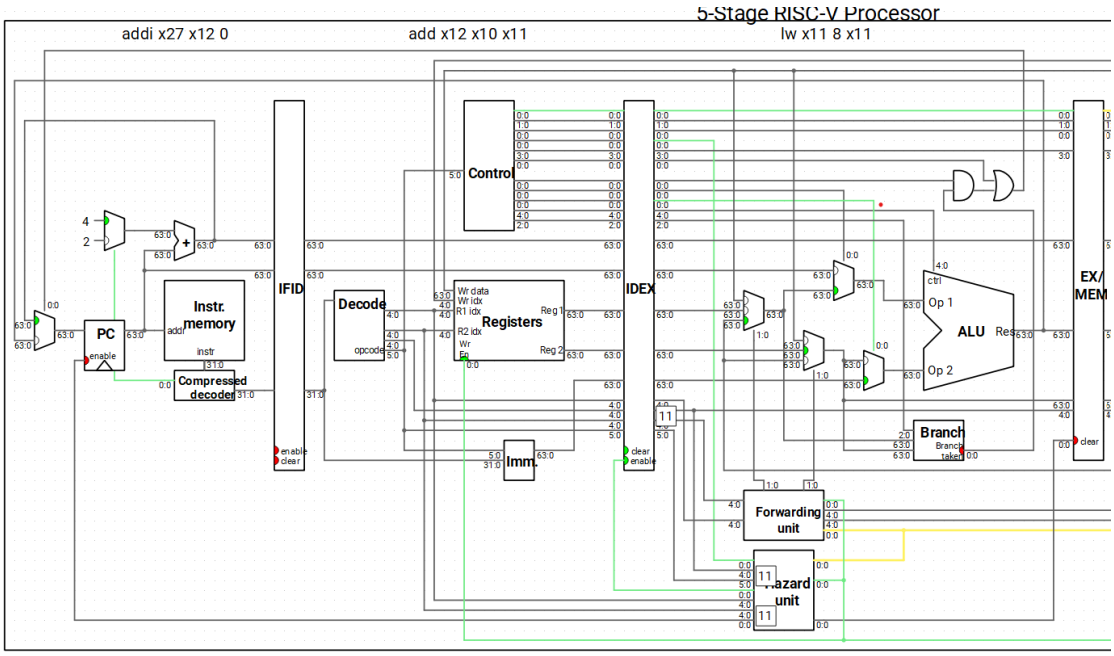
Explain :

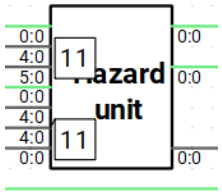
mul 指令(WB stage)的 rd 與下兩行 add 指令(EX stage)的 rs1 兩者 register number 一樣，都為 x11。此時，processor 偵測到兩指令間有 data dependency，因此將 mul 指令計算完 x11 的值= 10，利用 forwarding unit 產生控制訊號 0b10，將 mul 指令計算完 x11 的值設定為 add 指令 rs1 的值。所以，add 指令中 rs1 的值為 mul 指令計算完成後的值=10，而非本身在 x11 中的值 = 3

(3) Load RAW at the 1st instruction :

```
31      lw a1, BSize
32      add a2, a0, a1
```

The register a1 is used as rd to store the result of load instruction (line 31). However, a1 is also the rs2 of the add instruction (line 32). They have data dependency.





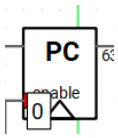
(ID stage register number : rs1 = 10, rs2 = 11)

(EX stage register number : rd = 11)

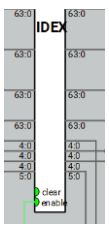
(ID/EX.MemRead control signal is enable)

Explain :

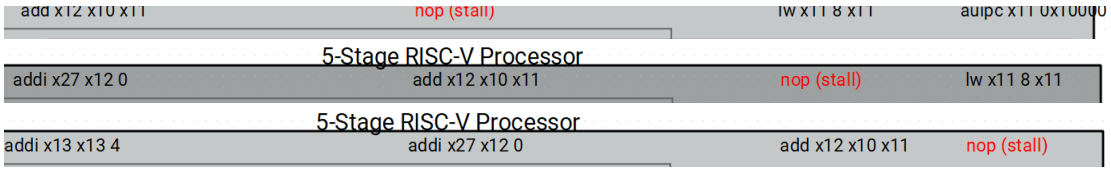
lw 指令(MEM stage)的 rd 與下一行 add 指令的 rs2 兩者 register number 一樣，都為 x11, ID/EX.MemRead enable。此時，processor 偵測到兩指令間有 data dependency。此時會將 ID/EX 的 clear enable 訊號設定為 -1 (signed value)，即類似於在 EX stage insert bubble ; 將 PC enable 的控制訊號設定為 0 (signed value)，即不更新 PC 值。造成下一個 cycle 的 EX stage 為 nop，下下個 cycle 的 MEM stage 為 nop，下下個 cycle 的 WB stage 為 nop。



不更新 PC 值



將 ID/EX clear enable，使下個 cycle 產生 nop

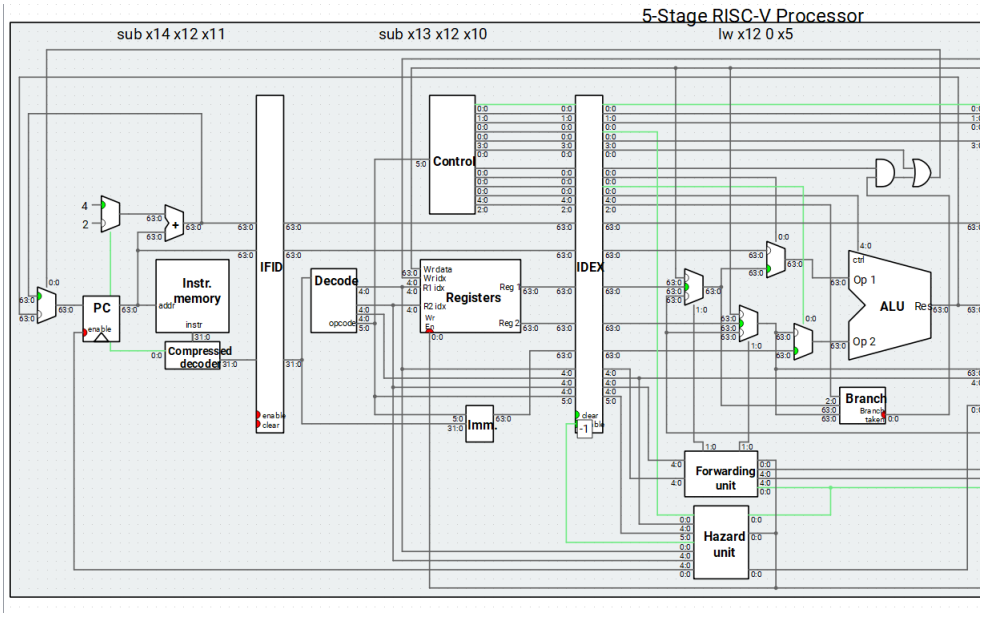


(發生 Type1 load use data hazard 後，下一個 cycle 使 Ex stage 產生 nop，讓原本在 ID stage 的指令停在原本的地方，將產生一個 cycle 的 stall)

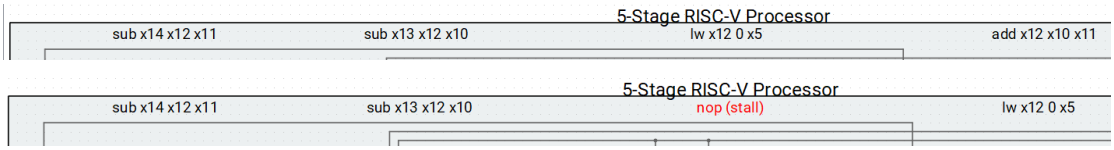
(4) Load RAW at the 2nd instruction : use simple code to generate (no such hazard in my code) :

```
34      lw a2, 0(t0) # load instruction
35      sub a3, a2, a0
36      sub a4, a2, a1
```

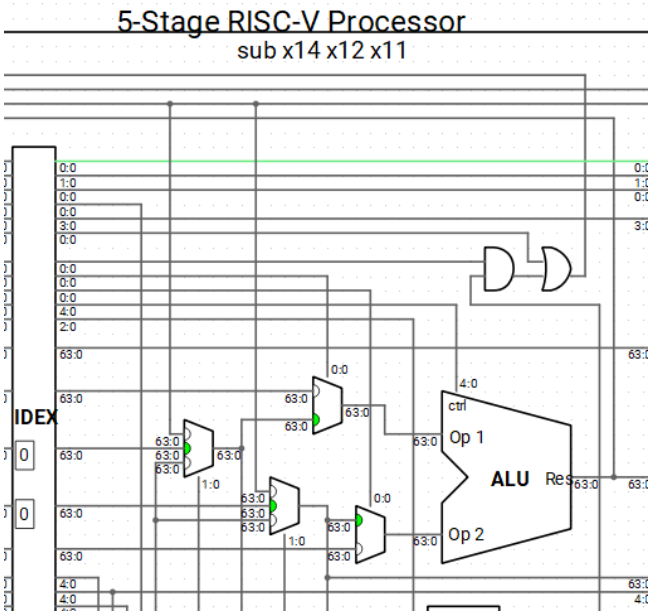
The register a2 is used as rd to store the result of load instruction (line 34). However, a2 is also the rs1 of the sub instruction in following next two instruction (line 36). They have data dependency.



Explain : 目前要分析的指令在 lw 後 2 個指令。如果 lw 的下一個指令之 rs1 或 rs2 與 lw 指令的 rd **有** data dependency，則會產生 nop，使得原本在 IF 及 ID stage 的指令留在原地。若如果 lw 的下一個指令之 rs1 或 rs2 與 lw 指令的 rd **沒有** data dependency，則不會產生 nop (ID/EX clear 不會 enable)。這個例子屬於情況一，因此原本在 IF stage 的指令停留在原地，會將 ID/EX 的 clear enable 訊號設定為 -1 (signed value)，即類似於在 EX stage insert bubble ; 將 PC enable 的控制訊號設定為 0(signed value)，即不更新 PC 值。



(sub x14, x12, x11 原本在 IF stage，因為下一個 cycle 插入一個 nop，因此在下一個 cycle 時，停留在 IF stage 不動)

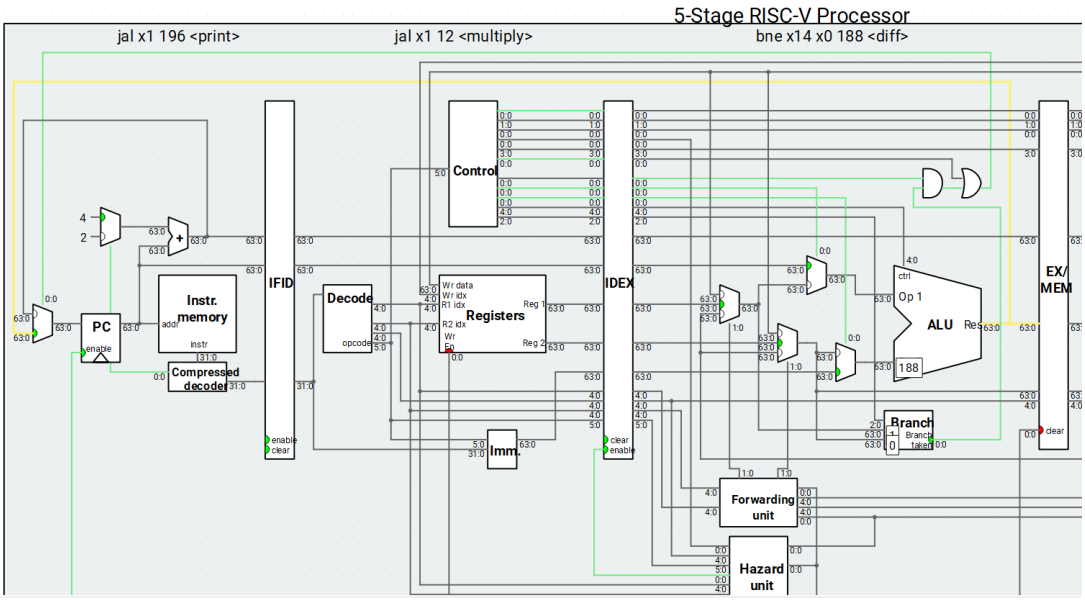


(如此一來，當 sub x14, x12, x11 指令要進入 EX stage 時，就可以將 lw 已經結束 WB stage 的值拿來使用)

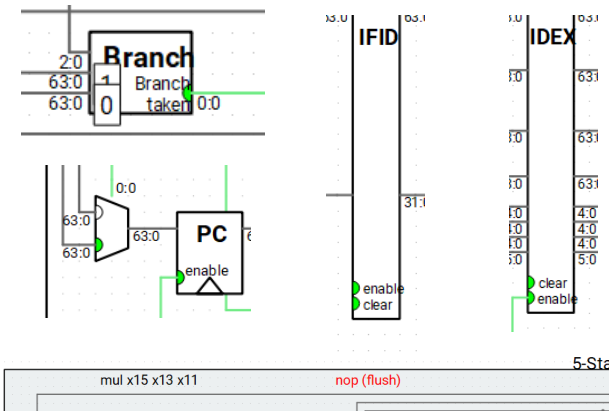
(5) **Branch instruction (control hazard)**:

```
44      bne a4, zero, diff   # for checking the lenght of two polynomials
45
46      jal multiply
47      jal print
```

The register a4 is used as rs1 and zero(x0) is used as rs2 to compare their values(in line 44). In this example, their values are not equal, and will jump to **diff** label (taken). However, the Ripes uses static prediction, which is assumed that the branch wouldn't taken. Therefore, it will implement the flush at IF/ID and ID/EX to generate nop at the following two stages. And at the next cycle, it will fetch the instruction, where the address would be PC + imm to jump to the diff function.



Explain : Ripes 的做法與課堂上老師的做法較不一樣。在此例子當中，bne 決定是否要 taken 是在 EX stage 去做判斷(課本是在 ID stage 去做判斷)。由上圖可以看到，此時 bne 指令中 rs1 的值 ＝1，而 rs2 的值 ＝0，因此 bne 條件式 taken，會先將 ID/EX stage 中的 PC 值加上 imm 後的結果回傳給 IF stage 中的 PC value。此時 PC 的控制訊號也是選擇 branch 的控制訊號(非 PC + 4 的控制訊號)。另外，因為要 flush 掉後面 2 個指令，因此 IF/ID 的 clear 為 enable，且 ID/EX 的 clear 也為 enable，使的在下一個 cycle 時，IF 抓到的指令是 diff 的第一行指令，ID 與 EX 則為 nop。



(預測 not taken，但實際上為 taken，將 flush 掉後面 2 兩個指令，並在下一個 cycle ，IF 階段進入 diff)

(下一個 cycle 的情況)