我根據 matrix template 裡的 geeksforgeeks 網址中第三個方法—Tabulation 去實作 assembly code，將最後算出來的答案存進 x31 裡。

1. 實作想法與細節:

我一開始是用網址中的第一個方法—直覺、無記憶化的遞迴版本下去實作，但我在處理 assembly 的遞迴上遇到了相當大的困難，雖然 HW2 的 7.也是要將遞迴 C code 轉成 riscv，但我覺得那題跟這次的難度果然還是有一定的差距，所以我後來就選擇第三個方法去實作了。

然而一開始選擇第三個方法去寫 assembly 的時候，我不知道該怎麼 allocate 二維陣列，這時突然想到 m[n][n]可以把它想成 m[n*n]的一維陣列(如下圖的紅框部分)，所以存取 m[i][j]就是去 m[i*n + j]的地方取值就好(後來發現 55 行其實是多餘的)。

藍框的部分是一開始不知道該怎麼個別存取 Matrix_array 中的個別元素，後來 google 發現可以藉由 la 將 Matrix_array 的位址存進 x10，再透過 lw,offset 去存取個別元素就好。

```
35  _start:
36      la x10, Matrix_array        #load Matrix_array address to x10
37      lw x11, Number_of_matrix    #load Number_of_matrix to size
38
39      jal x1, MatrixChainOrder    #call the function
40
41  MatrixChainOrder:
42      mul x20, x11, x11   #need to allocate x20 = n*n elements
43      slli x20, x20, 2    #each element is 4 bytes
44      addi x21, x0, -1    #x21 = -1
45      mul x20, x20, x21   #x20 *= -1
46
47      add x12, x2, x0     #record the base address of m[][]
48      add x2, x2, x20     #allocate m[n][n]
49
```

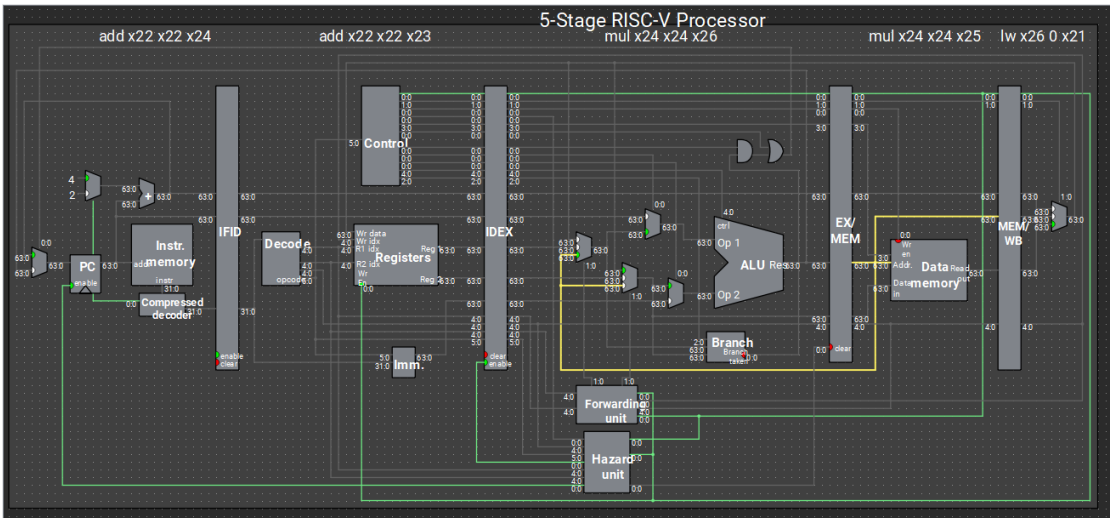
```
54      # m[i][i] = m[i*n + i]
55      add x21, x0, x0     #set x21 = 0 before use
56      mul x21, x5, x11    #x21 = i*n
57      add x21, x21, x5    #x21 = i*n + i
58      slli x21, x21, 2    #each element is 4 bytes
59      add x21, x12, x21   #x21 = address of m[i][i]
```

2. Hazards:

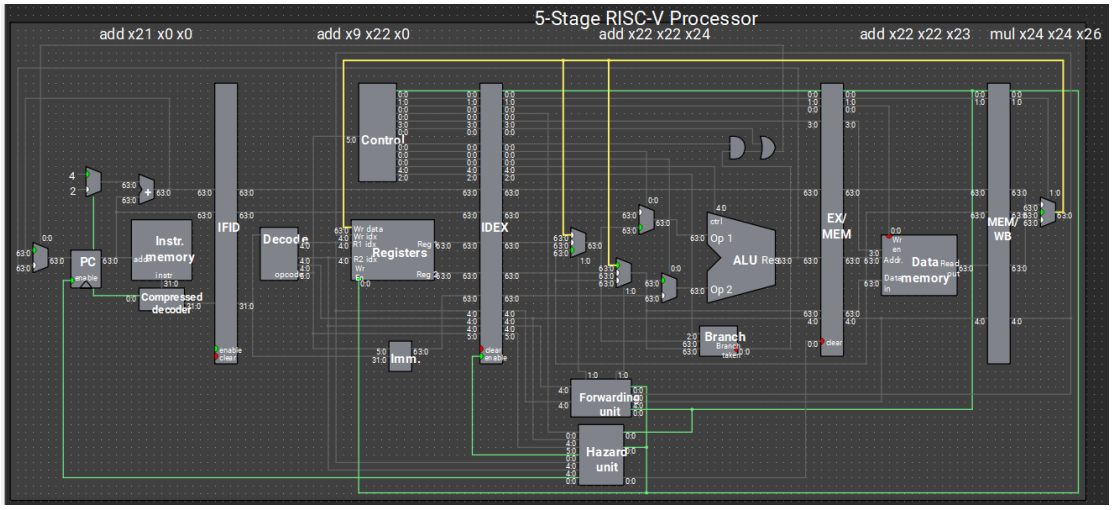
Type (1)(2):

```
131     mul x24,x24,x25
132     mul x24,x24,x26
133
134     #m[i][k] + m[k + 1
135     add x22, x22, x23
136     add x22, x22, x24
```

Type 1 hazard 發生在 131,132 行的 x24，如下圖黃線所示，mul x24,x24,x25 在執行完 EX 階段後會將 x24 * x25 的結果 forward 到位於 EX 階段 mul x24,x24,x26 的 rs1 = x24。



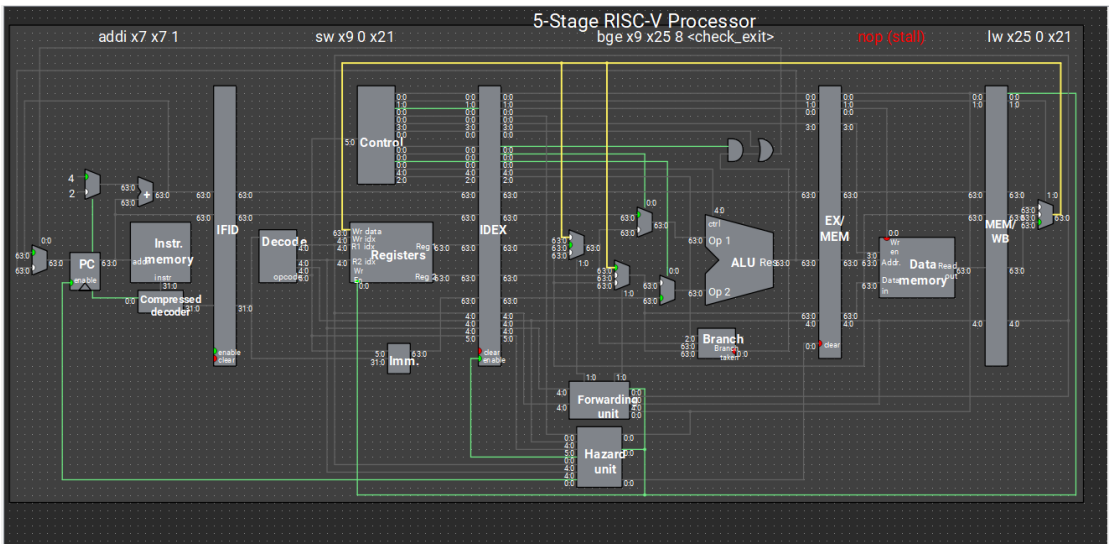
Type 2 hazard 發生在 132,136 行的 x24，如下圖黃線所示，mul 在執行完 MEM 階段後會將 x24 的結果 forward 到 EX 階段 add 的 rs2 = x24。



Type(3):

```
146     lw x25, 0(x21)      #x25
147     bge x9, x25, check_exit
```

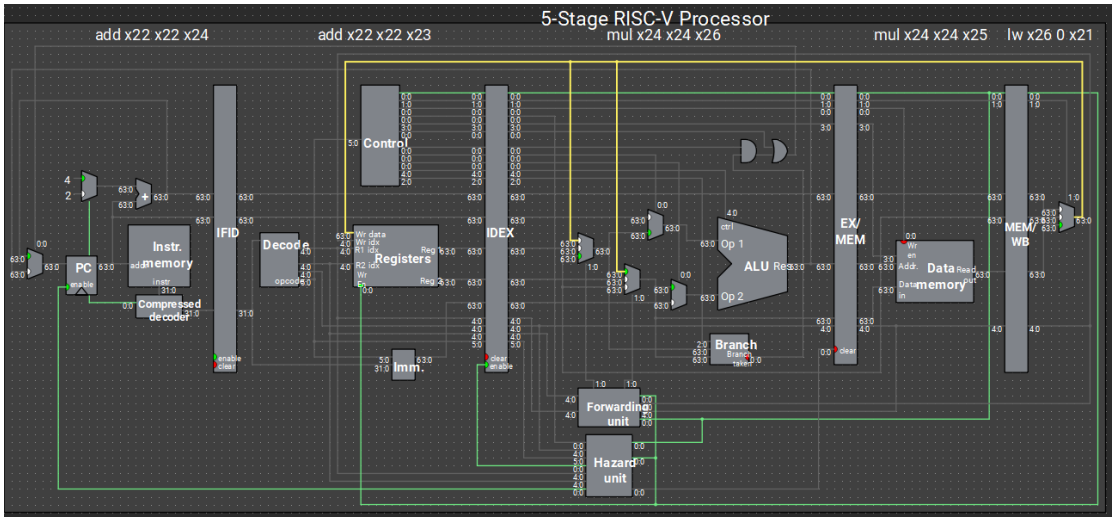
lw 的下一行就直接使用 x25，發生了 type 3 的 hazard。當 lw 做完 EX 階段後，Ripes 偵測到了 hazard 就將 bge 的指令 stall 1 cycle，這樣 lw 才能在做完 MEM 階段後將 memory 中 0(x21) = x25 的值 forward 到位於 EX 階段 bge 的 rs2 = x25 中，如下圖黃線所示。



Type(4):

```
127     lw x26, 0(x21)
128
129
130     #p[i - 1] * p[k]
131     mul x24,x24,x25
132     mul x24,x24,x26
```

Type 4 hazard 發生在 127, 132 行的 x26，因為 lw, mul 相差兩個 cycle 所以不需要 stall，lw 會在 MEM 階段做完後將 memory 中 0(x21) = x26 的值 forward 到位在 EX 階段 mul 的 rs2 = x26，如下圖黃線所示，。

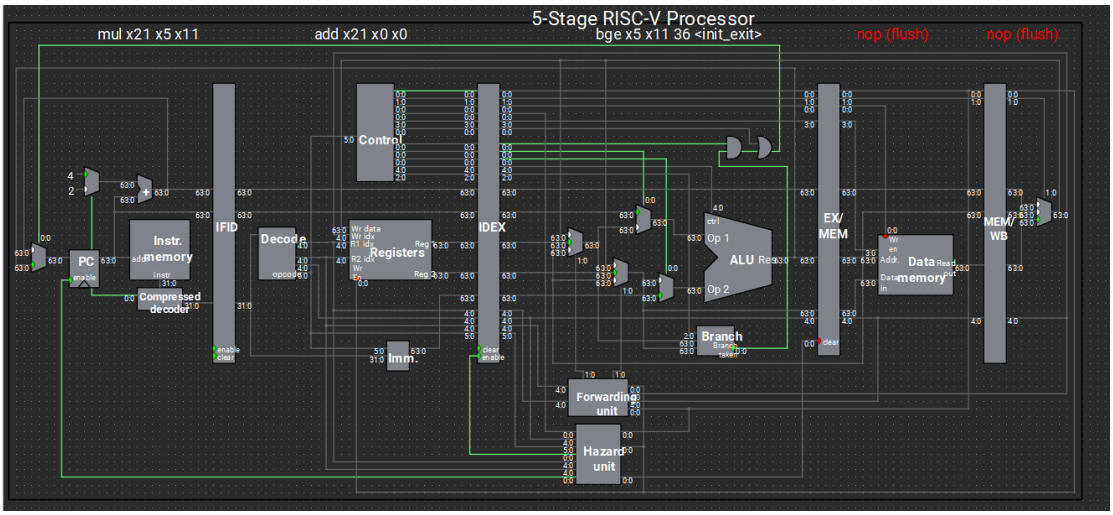


Type(5):

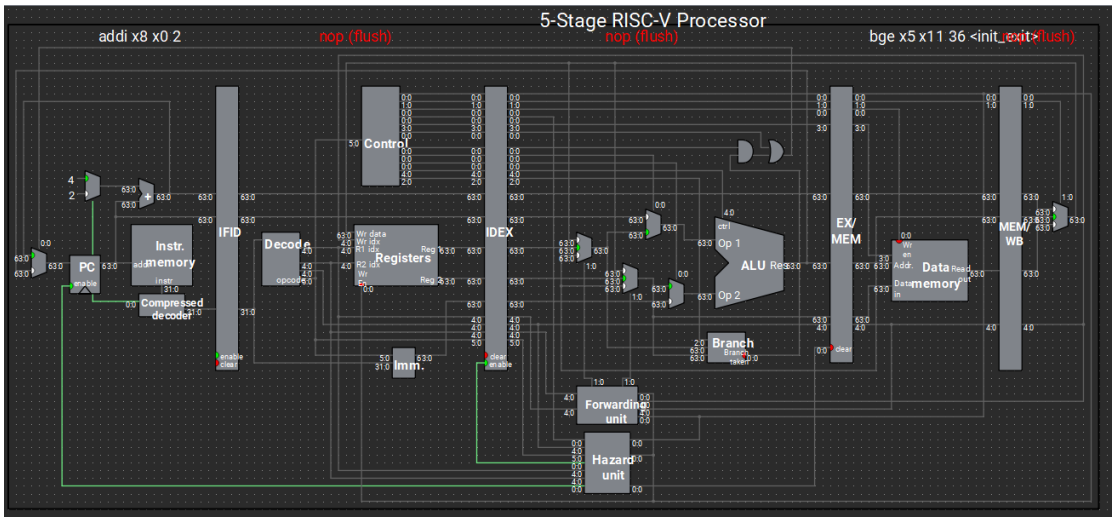
```
51  init:
52      bge x5, x11, init_exit
53
54      # m[i][i] = m[i*n + i]
55      add x21, x0, x0      #se
56      mul x21, x5, x11     #x2
57      add x21, x21, x5     #x2
58      slli x21, x21, 2     #ea
59      add x21, x12, x21    #x2
60      sw x0, 0(x21)        #m[
61
62      addi x5, x5, 1       #i
63      jal x0, init         #re
64
65  init_exit:
66      addi x8, x0, 2       #L
```

Type 5 hazard 發生在上圖的紅框與藍框部分。

下圖是 x5 已經大於等於 x11 時所截的圖，這時候的 processor 仍然繼續將 init block 裡藍框的 code pipeline 進 processor 的 IF,ID stage，但事實上因為 init block 的條件(x5 < x11)已不再成立，所以 processor 就得如圖二所示，將已經不符合條件的 add,mul 指令 flush 掉，再將 init_exit 的 addi pipeline 到 processor 裡。



(圖一)


5-Stage RISC-V Processor
addi x8 x0 2
nop (flush)
nop (flush)
bge x5 x11 36 <init_exit>
PC
Instr. memory
Compressed decoder
IFID
Decode
Control
Registers
Imm.
IDEX
Op 1
Op 2
ALU
Branch
Forwarding unit
Hazard unit
EX/ MEM
Data memory
MEM/ WB

(圖二)