計算機組織

Final Project

指導老師: 蔡宗漢

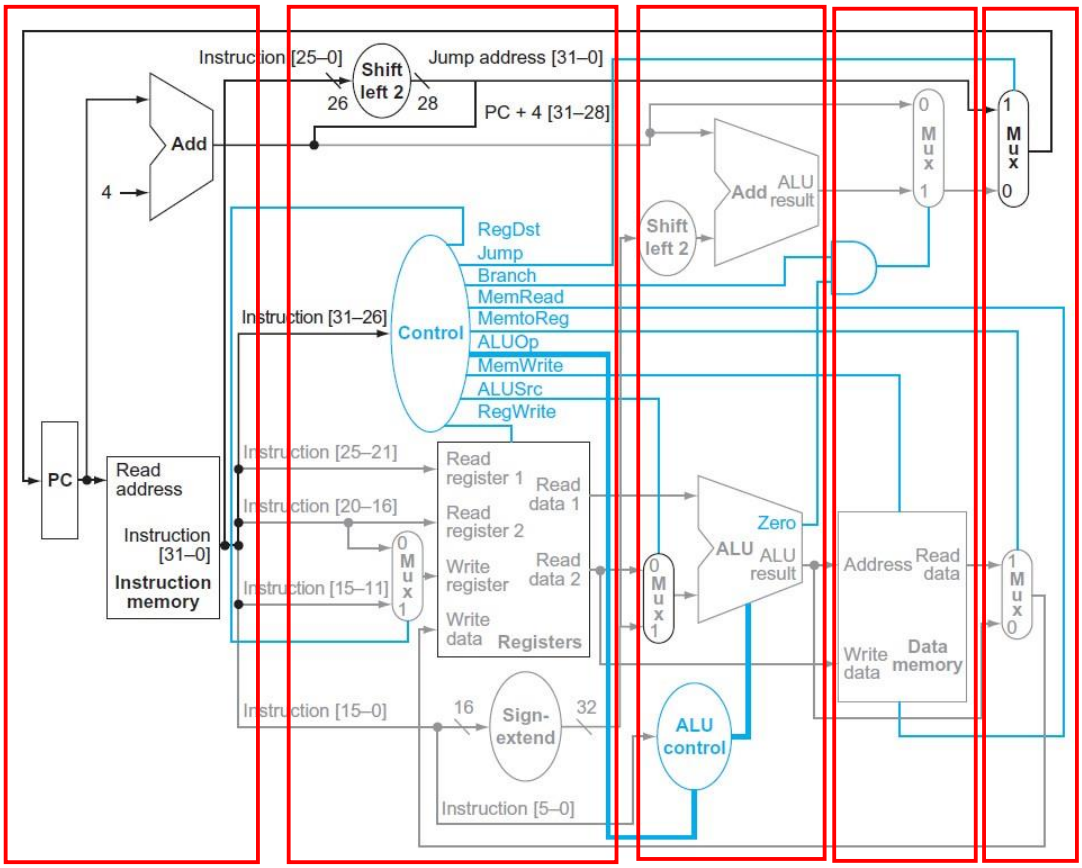
組員: 吳叡青 107501534

劉邦均 107501525

羅世辰 107501526

A. Single Cycle Implementation:

1. Data Path:



Instruction memory

將要執行的 MIPS code 轉成 machine code 並將他放到 instr_mem 的 array 中，並用 PC 來代表 instr_mem 位址。

Registers

一開始給 memory 位址中的初始值並且將 machine code 中的 rs rt 放在 ReadData1 ReadData2 中，利用 control 的 regwrite 來判斷，當 regwrite 為 1 時，將 WriteData 的值放進 memory 中

Control 一開始先將 Control 中的所有值(jump, regdst, alusrc, memtoreg, regwrite, memread, memwrite, branch, aluop)設為 0，再由 OP code 來判斷那些值各是多少，來決定整個程式中需要控制的 logic。

Sign-extend

instruction[15:0]為 offset 或是 immediate，需要將其擴展成 32 bits，因為 ALU 要用 32bit 去做運算，負數的加減才不會有問題(default 是補 0，但負數要補 1)。

ALU Control

利用 machine code 中 R-type 的 function，來判斷這個 ALUcrtl，而 ALUctrl 是用來判斷 ALU 要做的事情(例如:加減乘除)。

ALU 利用 ALU Control 中判斷出來的 ALU ctrl，來判斷加減乘除。

Data_Memory

利用 control 給的 MemWrite 和 MemRead，來判斷是要將 WriteData 的值寫到記憶體的 address，還是要將 address 的值拿出來放到 ReadData。

R-type(Add)

一開始會將 memory 位址中 PC 中儲存的指令拿出來丟到 instruction memory 中，之後將 code 分為 op[31:26] rs[25:21] rt[20:16] rd[15:11] shamt[10:6] func[5:0]。

而 op 丟進 control 中去判斷為何種 code 以及 control 裡面的值應該各為多少，去控制之後的某些子程式，而 rs rt 分別丟進 Read Register1 2 裡面並且將 memory 中地址為 Read Register1 2 的值傳到 Read Data1 2 中，而 rd 就經由 MUX 的選擇丟進 Write Register 中，最後 func 就丟到 ALUcontro 中去控制 ALU，而在這次的 code 中，ALU 拿來當作加法使用。

在 ADD 中，regwrite 為 1 就代表會將最後面算完的結果丟進 memory 中地址為 writereg 的地方，然後儲存起來。而之後將 Read Data1 2 之中的值通過 MUX 的選擇，丟到 ALU 裡面去進行相加。

最後將結果透過 MemtoReg 控制 MUX 使最後的值傳到 WriteData 並將其儲存到 memory 中地址為 writereg 的地方。

I-type(Lw)

一開始會將 memory 位址中 PC 中儲存的指令拿出來丟到 instruction memory 中，之後將 code 分為 op[31:26] rs[25:21] rt[20:16] offset[15:0] 。

而 op 丟進 control 中去判斷為何種 code 以及 control 裡面的值應該各為多少，去控制之後的某些子程式，而 rs 丟進 Read Register1 裡面並且將 memory 中地址為 Read Register1 的值傳到 Read Data1 中，而 rt 就經由 MUX 的選擇丟進 Write Register 中，最後 offset 就透過 MUX 的選擇與 ReadData1 一起放到 ALU 裡面做運算，而這次的 code 中，ALU 拿來做加法使用，用來計算要 load 的位址。

最後將 ALU 結果丟進 Data memory 的 Address 中，由於 MemRead 為 1，將 memory 中位址為 Address 裡面的值丟到 Read Data 中，最後將 Read Data 的值傳回 WriteData 裡面，然後獎 WriteData 的值儲存回 memory 中位址為 Write Register 中。

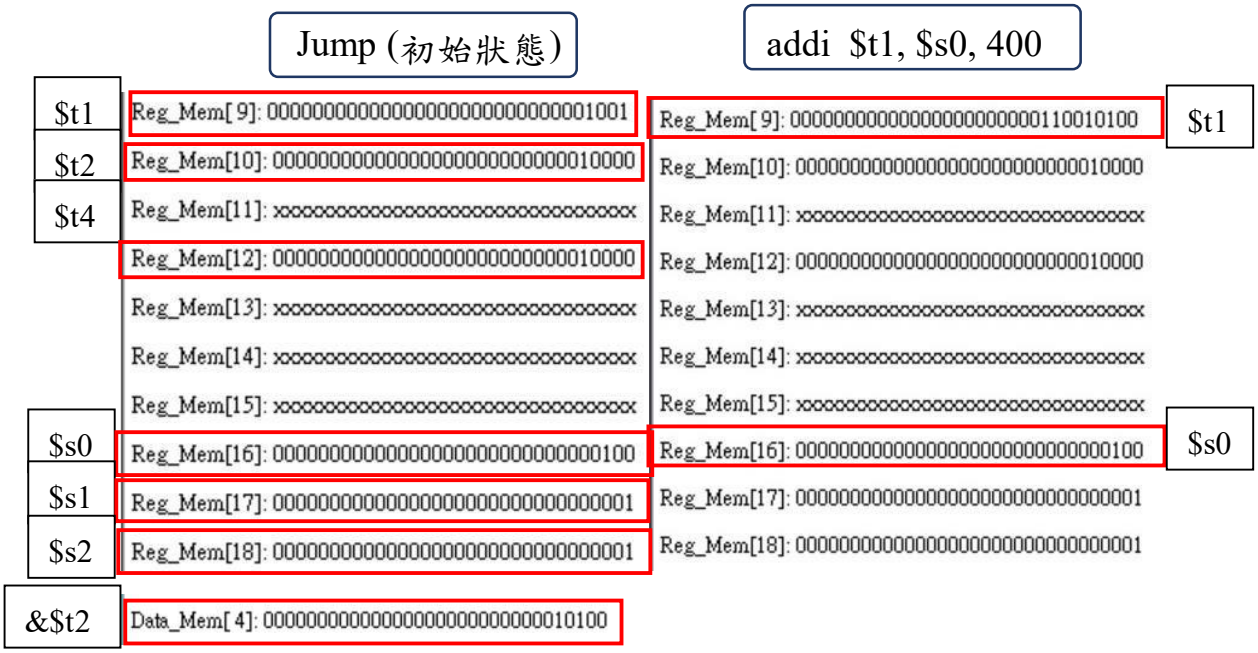
Jump/Branch Equal

一開始會將 memory 位址中 PC 中儲存的指令拿出來丟到 instruction memory 中，之後將 code 分為 op[31:26] rs[25:21] rt[20:16] constant or address[15:0] 。

而 op 丟進 control 中去判斷為何種 code 以及 control 裡面的值應該各為多少，去控制之後的某些子程式，而 rs rt 分別丟進 Read Register1 2 裡面並且將 memory
中地址為 Read Register1 2 的值傳到 Read Data1 2 中，最後 constant or address 要先進行擴充，使他從 16bits 轉到 32bits，然後再向左 shift 後傳到 ADD 中。而這次的 code 中，ALU 拿來做減法使用，將 Read Register1 2 丟進 ALU 裡面進行相減，若為 0 的話 zero 為 1，而會將 zero 拿去與 branch 做 and 來決定 MUX，若 and 完結果為 0，則 branch equal 成立代表會將 address 和本次的 PC 相加，決定下一次 PC 的位置，也就是要跳到另一個指令，若 and 完結果不為 1，則代表不成立，下一次 PC 的位置仍然為 PC+4。

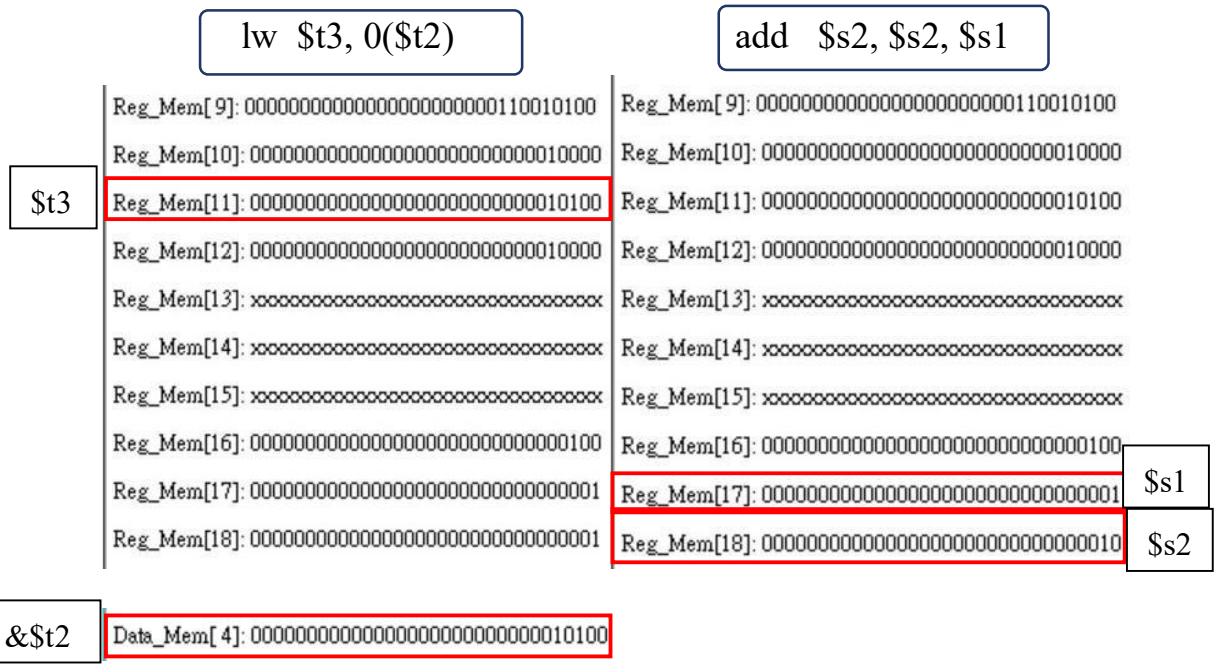
2. Example:

```
Start:   jump   Next
Next:    addi  $t1, $s0, 400
         lw    $t3, 0($t2)
         add   $s2, $s2, $s1
         addi  $t1, $t1,  -4
         beq     $t2, $t4, Start
```



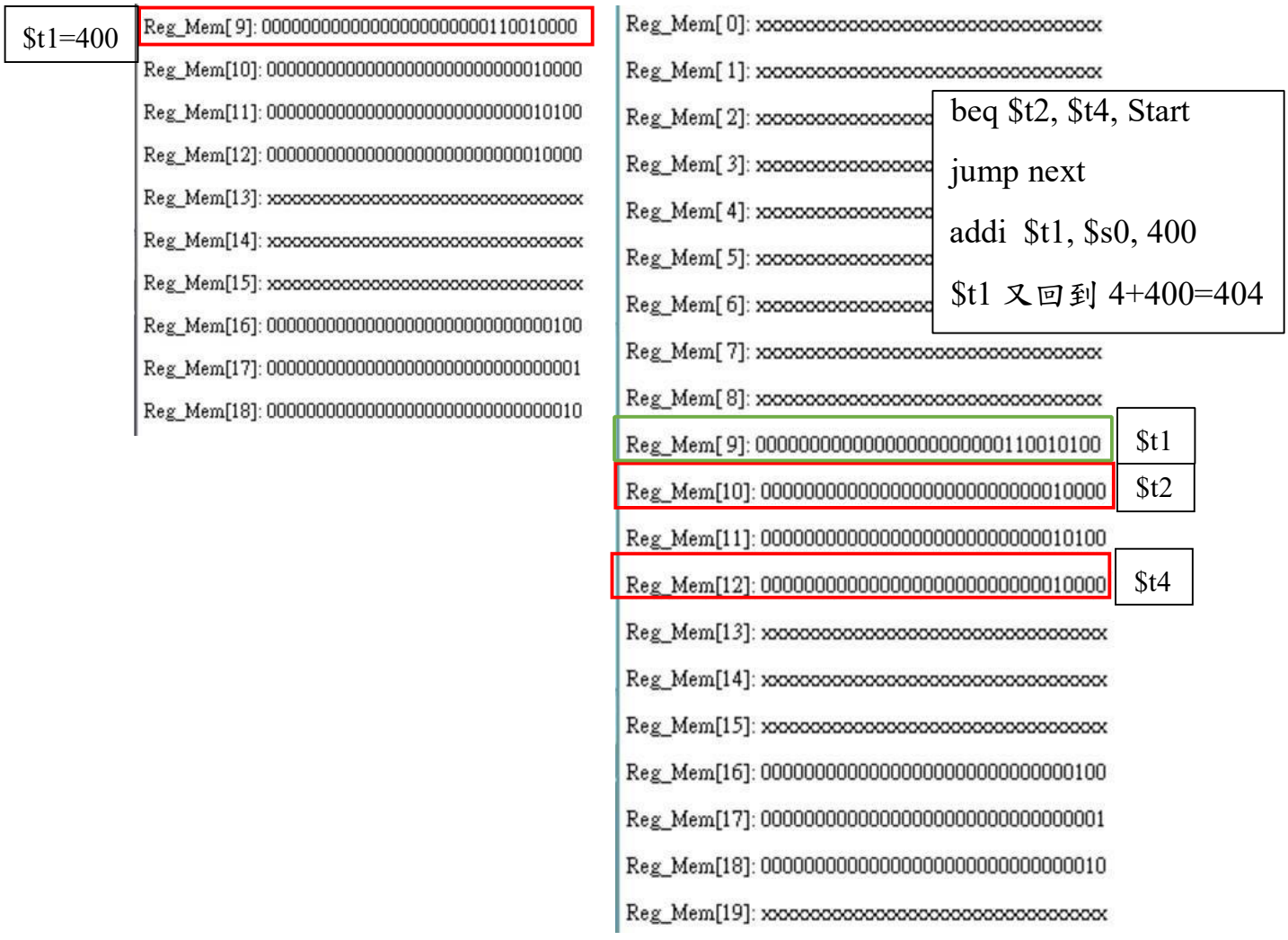
lw $t3, 0($t2)

add $s2, $s2, $s1



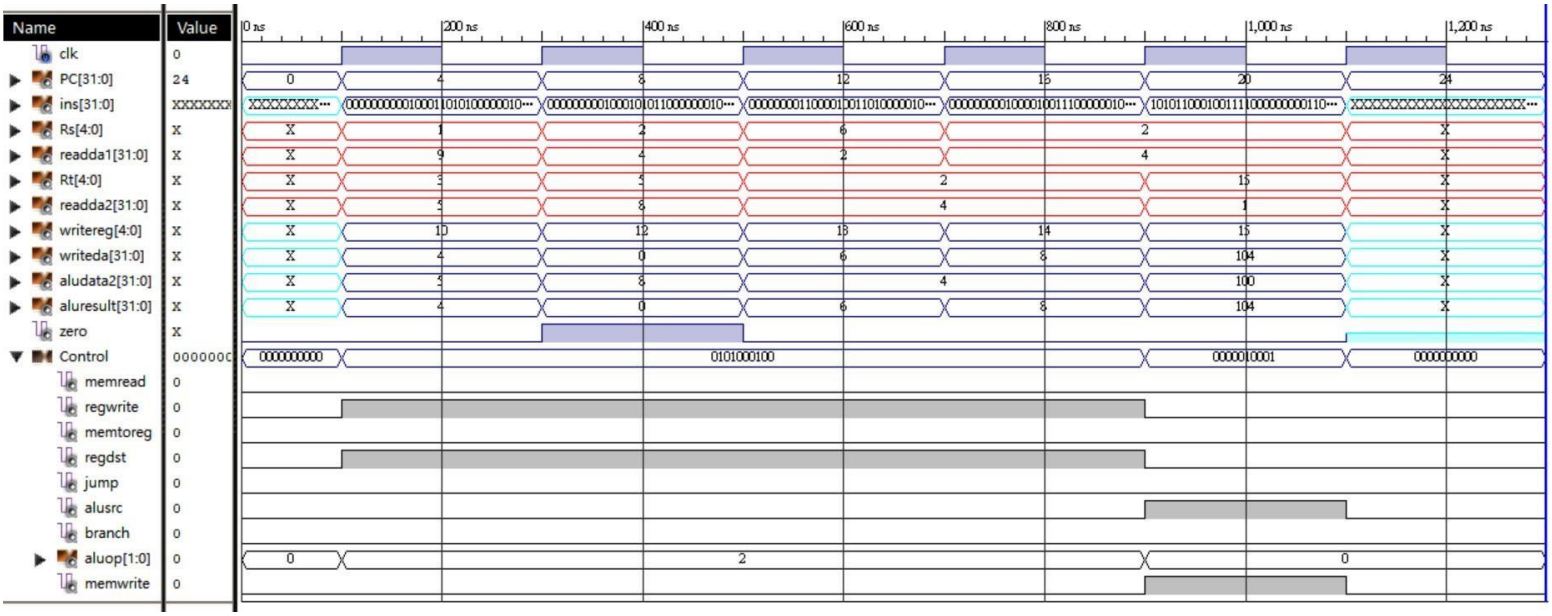
addi $t1, $t1, -4

beq $t2, $t4, Start

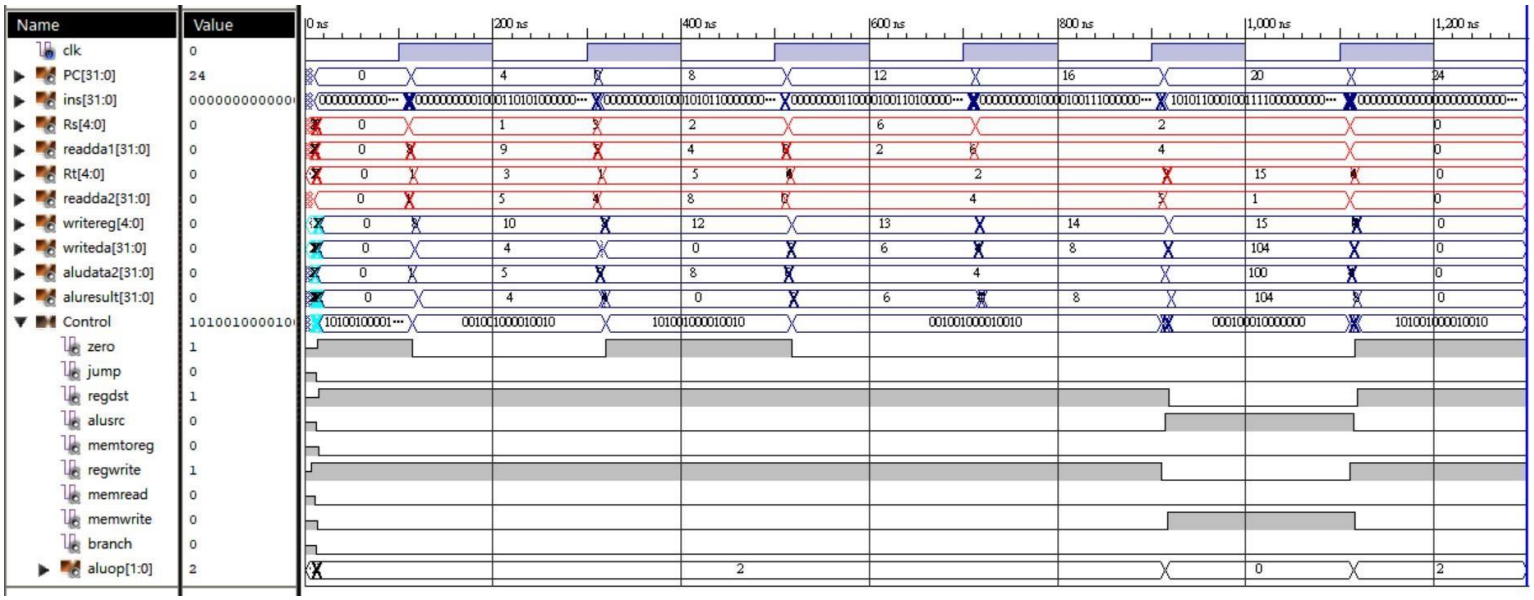


3. Waveform:

(1) Behavioral:

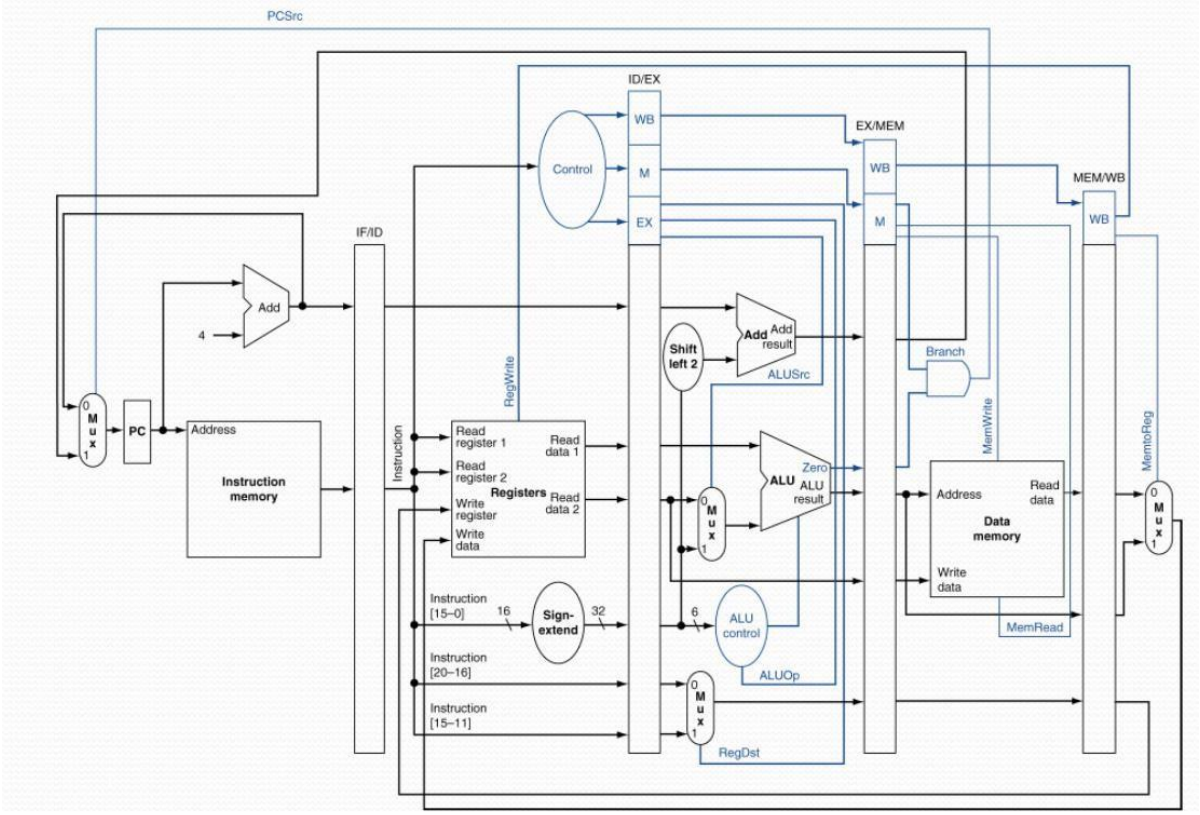


(2) Post-Route:



B. Pipeline Implementation:

1. Data Path:



IF_ID / ID_EX / EX_MEM / MEM_WB

這四個子程式都是利用一個 register 來儲存 stage 的一些 output，包括 control、ALU 運算的結果、從 memory 讀出來的資料等等，同一指令的資料會跟著指令到每個 stage，不同指令間不互相干擾，若下一 stage 不需要該資料則不用往下傳，以此可構成 pipeline 的形式，大幅縮短 clock cycle time。

2. Example:

```
sub  $10, $1, $3
 and  $12, $2, $5
or   $13, $6, $2
add  $14, $2, $2
sw  $15, 100 ($2)
```

初始狀態

Reg_Mem[1]: 00000000000000000000000000001001
Reg_Mem[2]: 00000000000000000000000000000100
Reg_Mem[3]: 00000000000000000000000000000101
Reg_Mem[4]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[5]: 00000000000000000000000000001000
Reg_Mem[6]: 00000000000000000000000000000010
Reg_Mem[7]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[8]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[9]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[10]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[11]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[12]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[13]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[14]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[15]: 00000000000000000000000000000001

Data_Mem[25]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Data_Mem[26]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Data_Mem[27]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Data_Mem[28]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

sub $10, $1, $3

Reg_Mem[1]: 00000000000000000000000000001001
Reg_Mem[2]: 00000000000000000000000000000100
Reg_Mem[3]: 00000000000000000000000000000101
Reg_Mem[4]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[5]: 00000000000000000000000000001000
Reg_Mem[6]: 00000000000000000000000000000010
Reg_Mem[7]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[8]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[9]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[10]: 00000000000000000000000000000100
Reg_Mem[11]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[12]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[13]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[14]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[15]: 00000000000000000000000000000001

and $12, $2, $5

Reg_Mem[1]: 00000000000000000000000000001001
Reg_Mem[2]: 00000000000000000000000000000100
Reg_Mem[3]: 00000000000000000000000000000101
Reg_Mem[4]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[5]: 00000000000000000000000000001000
Reg_Mem[6]: 00000000000000000000000000000010
Reg_Mem[7]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[8]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[9]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[10]: 00000000000000000000000000000100
Reg_Mem[11]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[12]: 00000000000000000000000000000000
Reg_Mem[13]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[14]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[15]: 00000000000000000000000000000001

or $13, $6, $2

Reg_Mem[1]: 00000000000000000000000000001001
Reg_Mem[2]: 00000000000000000000000000000100
Reg_Mem[3]: 00000000000000000000000000000101
Reg_Mem[4]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[5]: 00000000000000000000000000001000
Reg_Mem[6]: 00000000000000000000000000000010
Reg_Mem[7]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[8]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[9]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[10]: 00000000000000000000000000000100
Reg_Mem[11]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[12]: 00000000000000000000000000000000
Reg_Mem[13]: 00000000000000000000000000000110
Reg_Mem[14]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[15]: 00000000000000000000000000000001

add $14, $2, $2

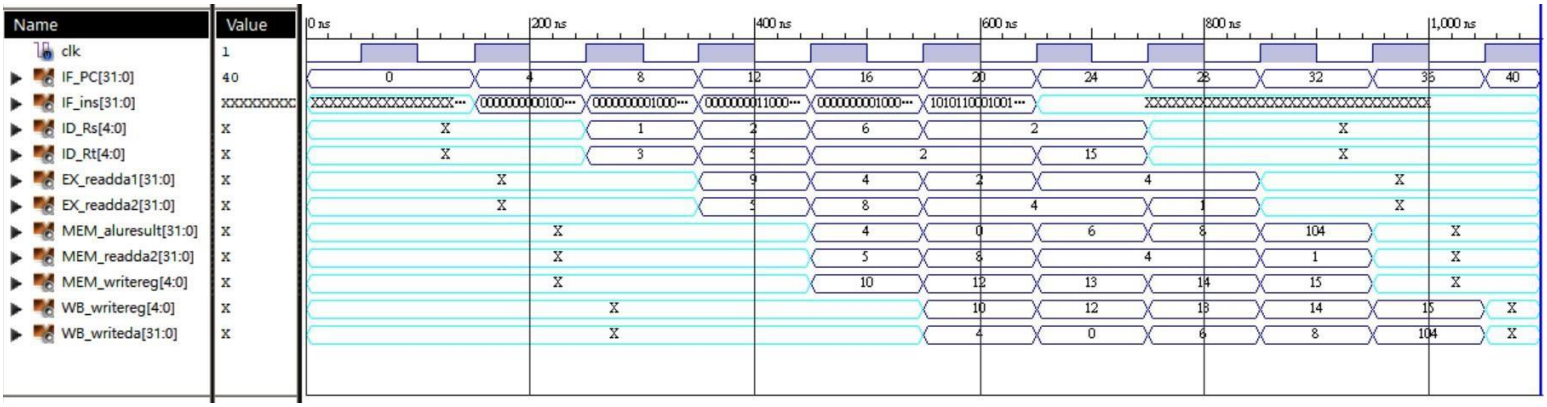
Reg_Mem[1]: 00000000000000000000000000001001
Reg_Mem[2]: 00000000000000000000000000000100
Reg_Mem[3]: 00000000000000000000000000000101
Reg_Mem[4]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[5]: 00000000000000000000000000001000
Reg_Mem[6]: 00000000000000000000000000000010
Reg_Mem[7]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[8]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[9]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[10]: 00000000000000000000000000000100
Reg_Mem[11]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[12]: 00000000000000000000000000000000
Reg_Mem[13]: 00000000000000000000000000000110
Reg_Mem[14]: 00000000000000000000000000001000
Reg_Mem[15]: 00000000000000000000000000000001

sw $15, 100 ($2)

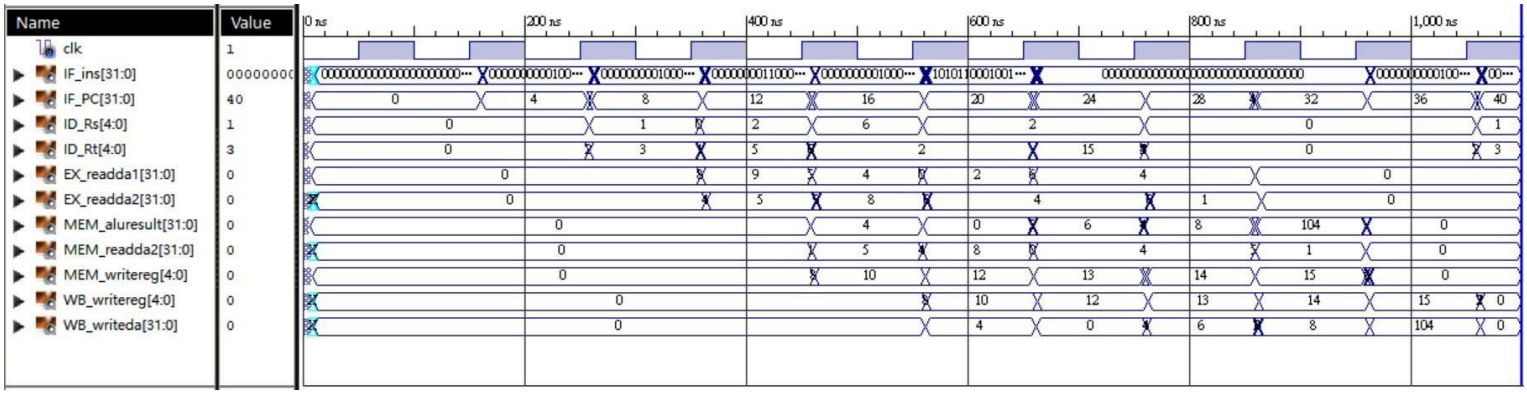
Data_Mem[25]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Data_Mem[26]: 00000000000000000000000000000001
Data_Mem[27]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Data_Mem[28]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

3. Waveform:

(1) Behavioral:



(2) Post-Route:



4. Timing Report:

A. Single Cycle:

```
Timing Summary:
---------------
Speed Grade: -3

  Minimum period: 25.403ns (Maximum Frequency: 39.366MHz)
  Minimum input arrival time before clock: No path found
  Maximum output required time after clock: 15.732ns
  Maximum combinational path delay: No path found
```

B. Pipelined:

```
Timing Summary:
---------------
Speed Grade: -3

  Minimum period: 8.586ns (Maximum Frequency: 116.467MHz)
  Minimum input arrival time before clock: No path found
  Maximum output required time after clock: 5.193ns
  Maximum combinational path delay: No path found
```

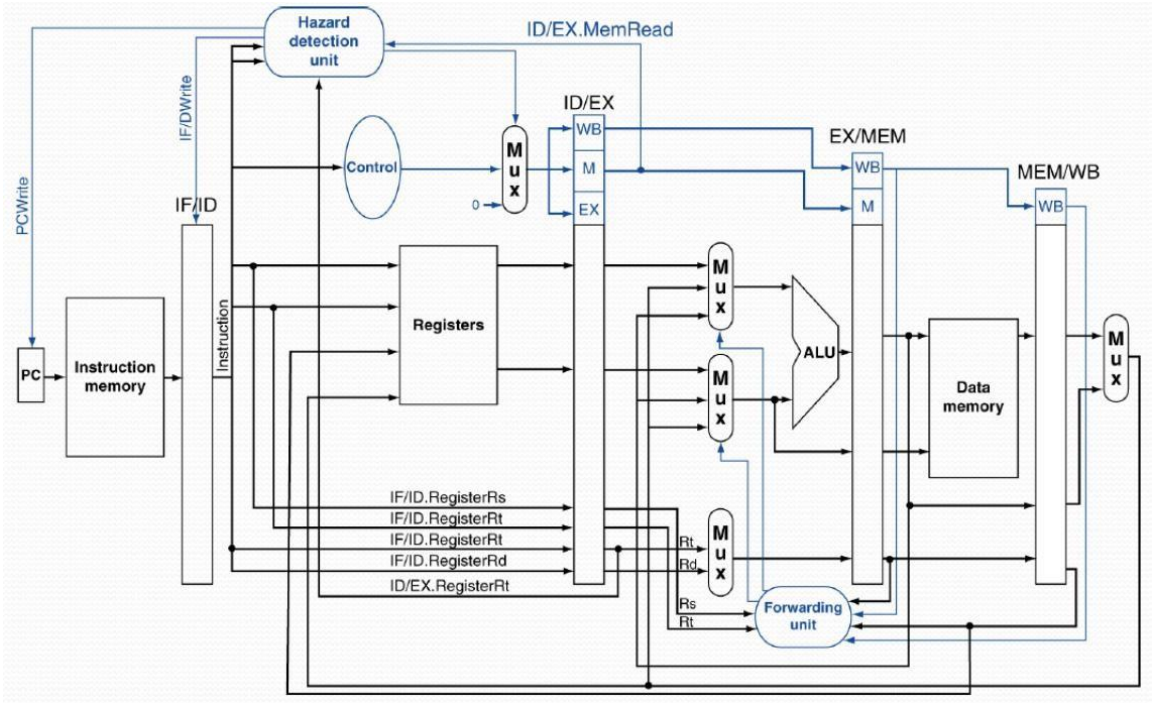
:

C. Hazard

```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 10.959ns (Maximum Frequency: 91.253MHz)
   Minimum input arrival time before clock: No path found
   Maximum output required time after clock: 7.283ns
   Maximum combinational path delay: No path found
```

C. Data Path with Hazard Detection:

1. Data Path:



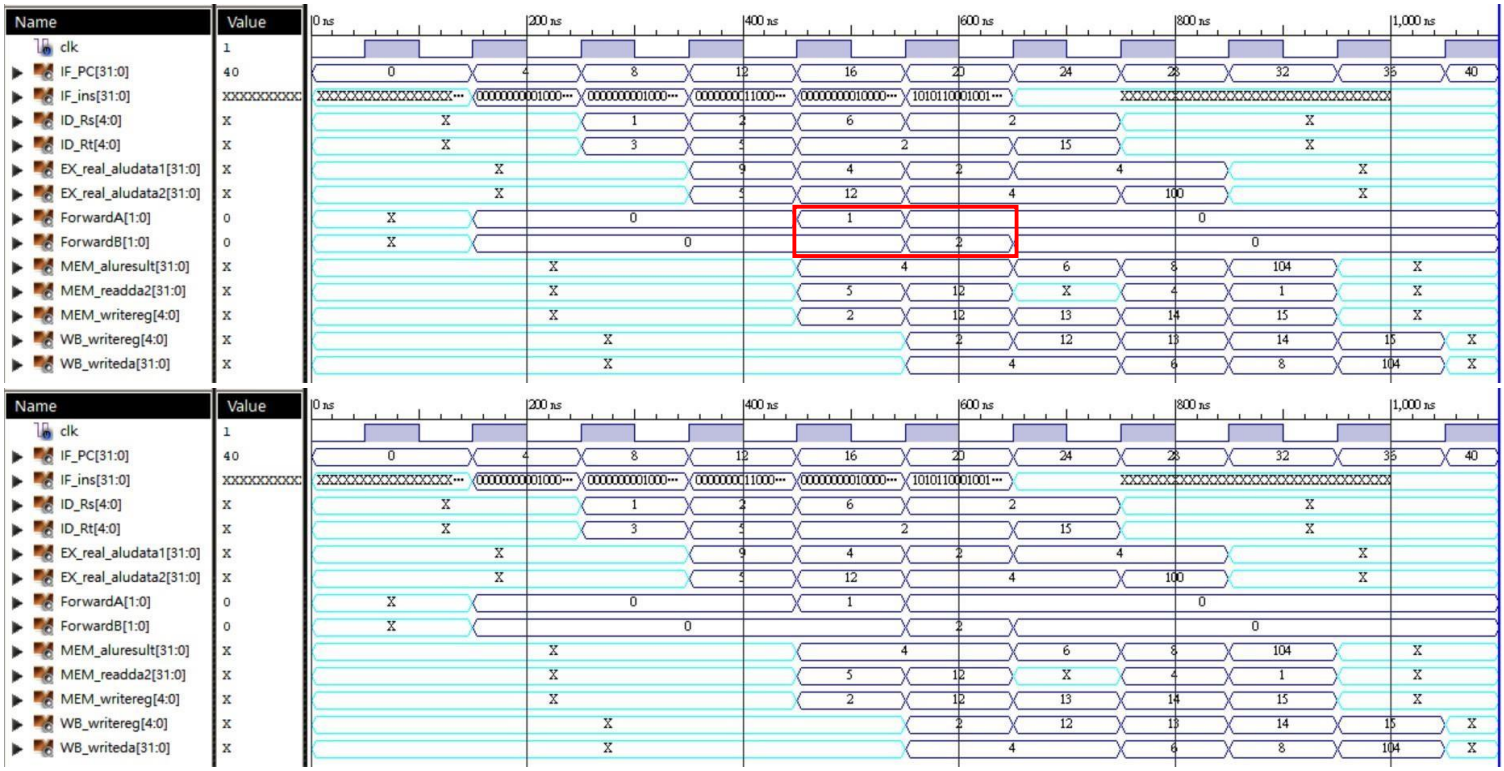
2. Example:

sub $2 , $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100 ($2)

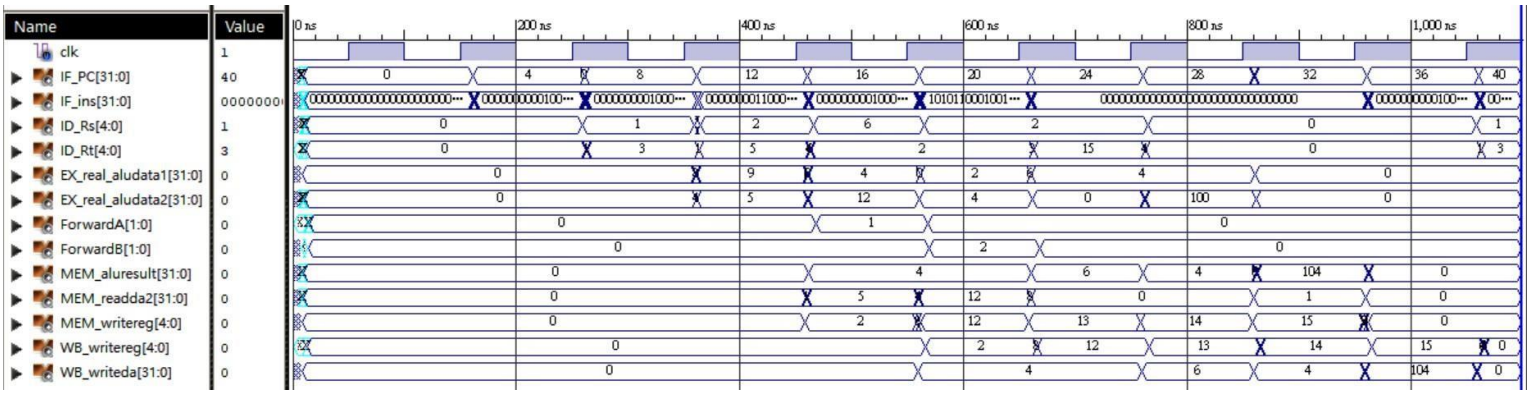
```
Reg_Mem[ 0]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[ 1]: 00000000000000000000000000001001
Reg_Mem[ 2]: 00000000000000000000000000000100   $2
Reg_Mem[ 3]: 00000000000000000000000000000101
Reg_Mem[ 4]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[ 5]: 00000000000000000000000000001100
Reg_Mem[ 6]: 00000000000000000000000000000010
Reg_Mem[ 7]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[ 8]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[ 9]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[10]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[11]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Reg_Mem[12]: 00000000000000000000000000000100   0100 & 1100 = 0100
Reg_Mem[13]: 00000000000000000000000000000110   0100 | 0010 = 0110
Reg_Mem[14]: 00000000000000000000000000001000   0100 + 0100 = 1000
Reg_Mem[15]: 00000000000000000000000000000001

Data_Mem[25]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Data_Mem[26]: 00000000000000000000000000000001   4+100 = 104 = 26th word
Data_Mem[27]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Data_Mem[28]: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

3. Waveform:

(1) Behavioral:



(2) Post-Route:



工作分配**:**

	吳叡青	劉邦均	羅世辰
工作分配	負責各自副程式與主程式的接線	負責各自副程式與主程式的接線	負責各自副程式與主程式的接線
	撰寫不同功能的副程式	撰寫不同功能的副程式	撰寫不同功能的副程式
貢獻比例	33%	33%	33%

參考資料**:**

1. 課本 Chapter 4
2. 老師講義 Chapter 4