計算機組織期末 Project

110 學年度第 2 學期

老師：朱守禮老師

組別:第 36 組

班別:資訊二乙

學生：資訊三乙 10833230 李郁含

、資訊三乙 10844132 黃中憫

、資訊三乙 10844149 謝宜庭

一、 背景

Project 目的：使用 Verilog HDL 與 Icarus Verilog 模擬器，以 MidtermProject 所設計之 ALU Design 為基礎設計一個 Pipelined MIPS-Lite CPU。

參考：是以計算機組織課程講義為基礎

[1] Chapter 4
[2] 課程講義之 Pipelined Datapath

Project 基本要求：

設計要求:

ALU: 組合邏輯(Combinational Logic)

使用 Midterm Project 所設計之 ALU 完成 add, sub, and, or, sll, slt, ori 指令。

Datapath:

所有指令之執行，須遵守 5-Stage Pipelined CPU 執行指令之行為。

DIVU: 32-bits 無號數除法指令，使用 Midterm Project 所設計之 Division Hardware。

指令要求: 16 道 MIPS 指令

a) Integer Arithmetic: **add, sub, and, or, sll, slt, ori**

b) Integer Memory Access: **lw, sw**

c) Integer Branch: **beq, bne, j**

d) Integer Multiply/Divide: **divu**

e) Other Instructions: **mfhi, mflo, nop**

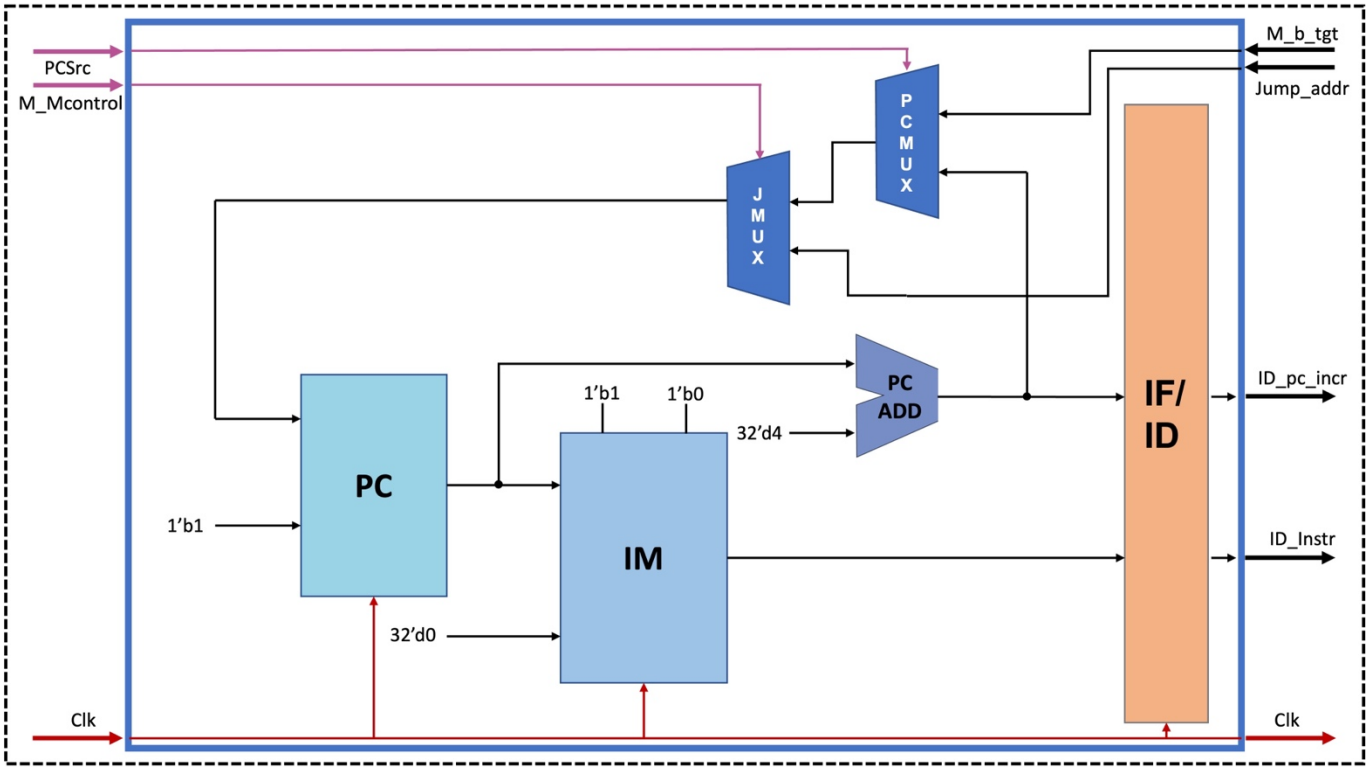
二、 方法

Datapath、架構圖與設計重點說明

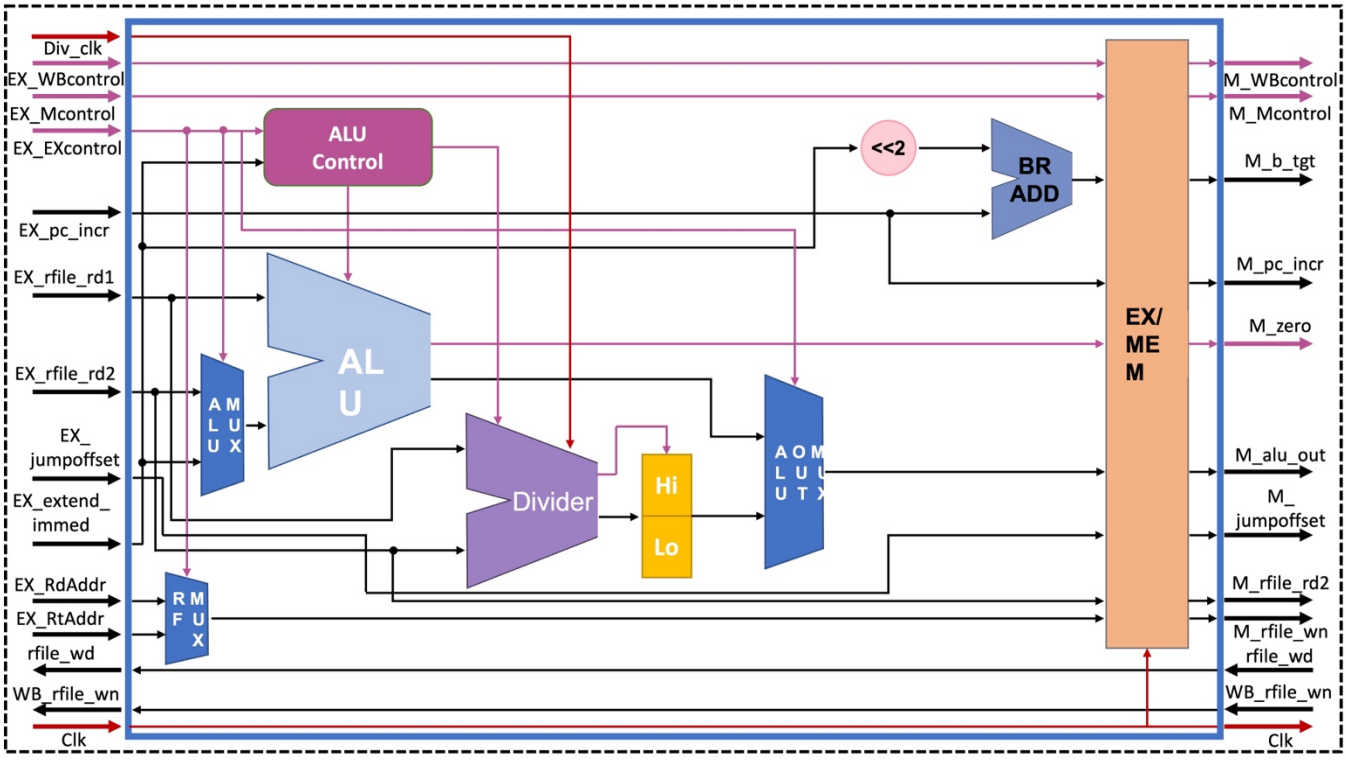
(1) Datapath:

IF_ID、MEM_WB、ID_EX、EX_MEM: pipelined 的五個 stage，在 clk 正緣觸發時會將原本 in 的訊號丟到 out，也就是前往下一個 stage。

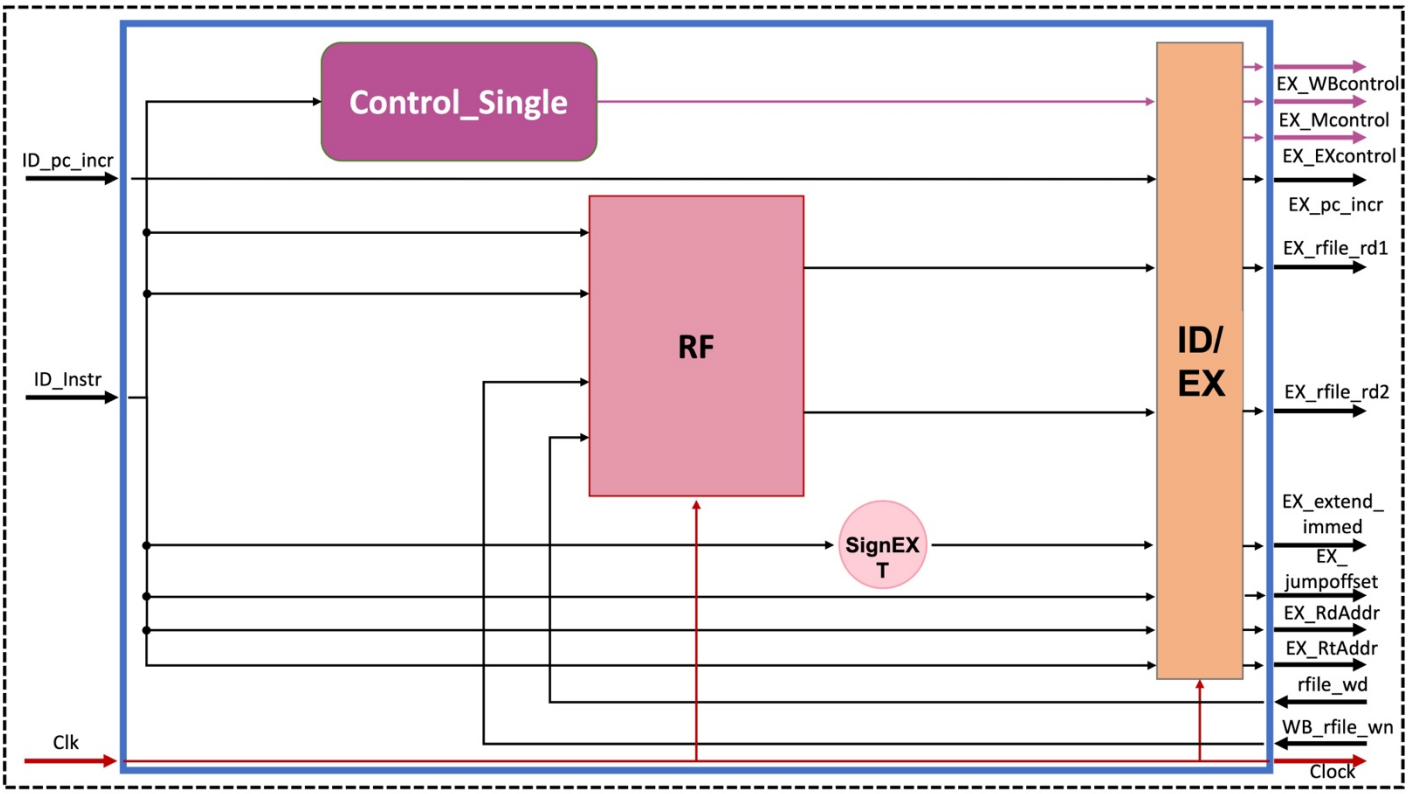
I. Stage_IF



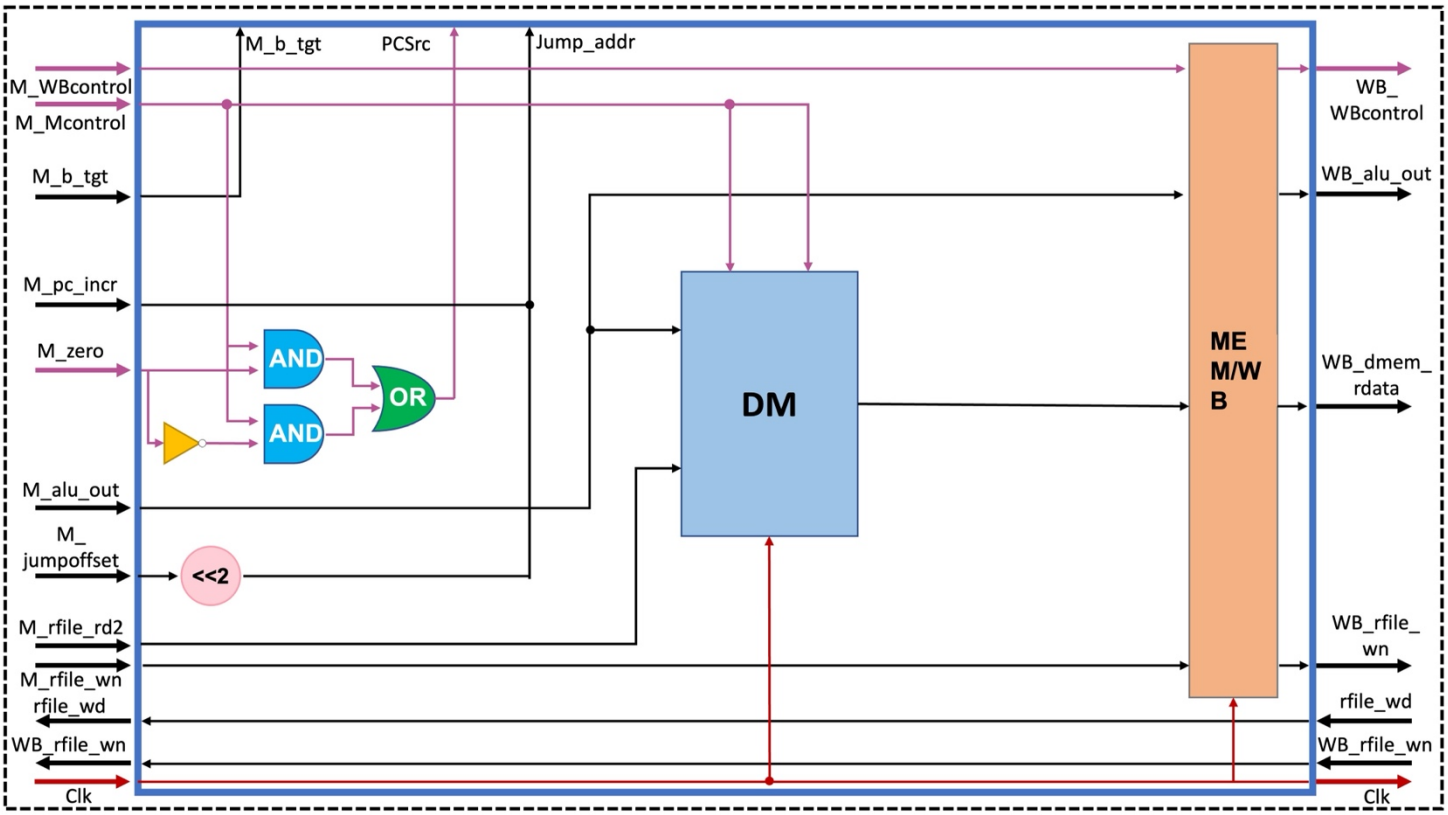
II. Stage_EX



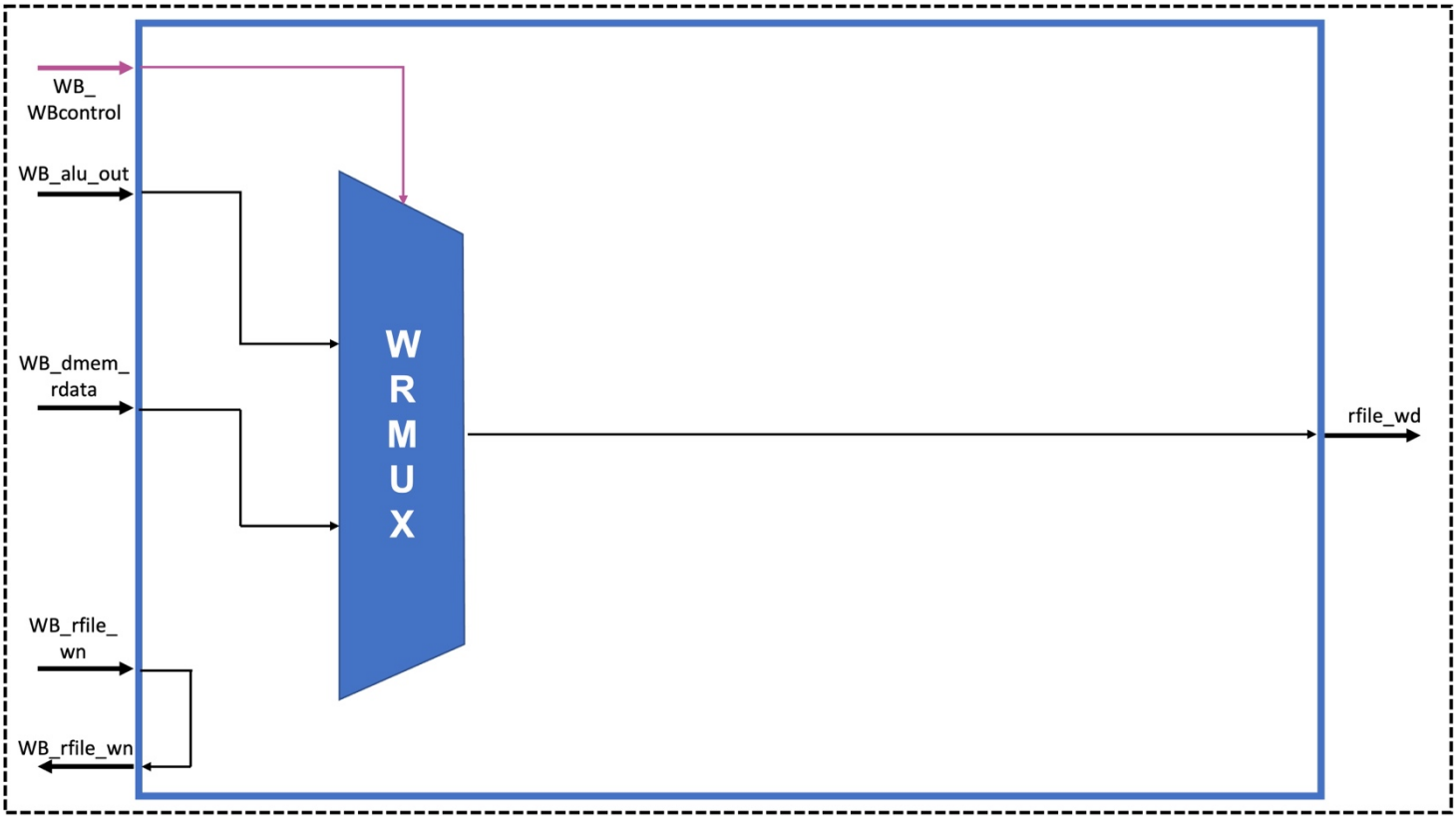
III. Stage_ID



IV.Stage_MEM



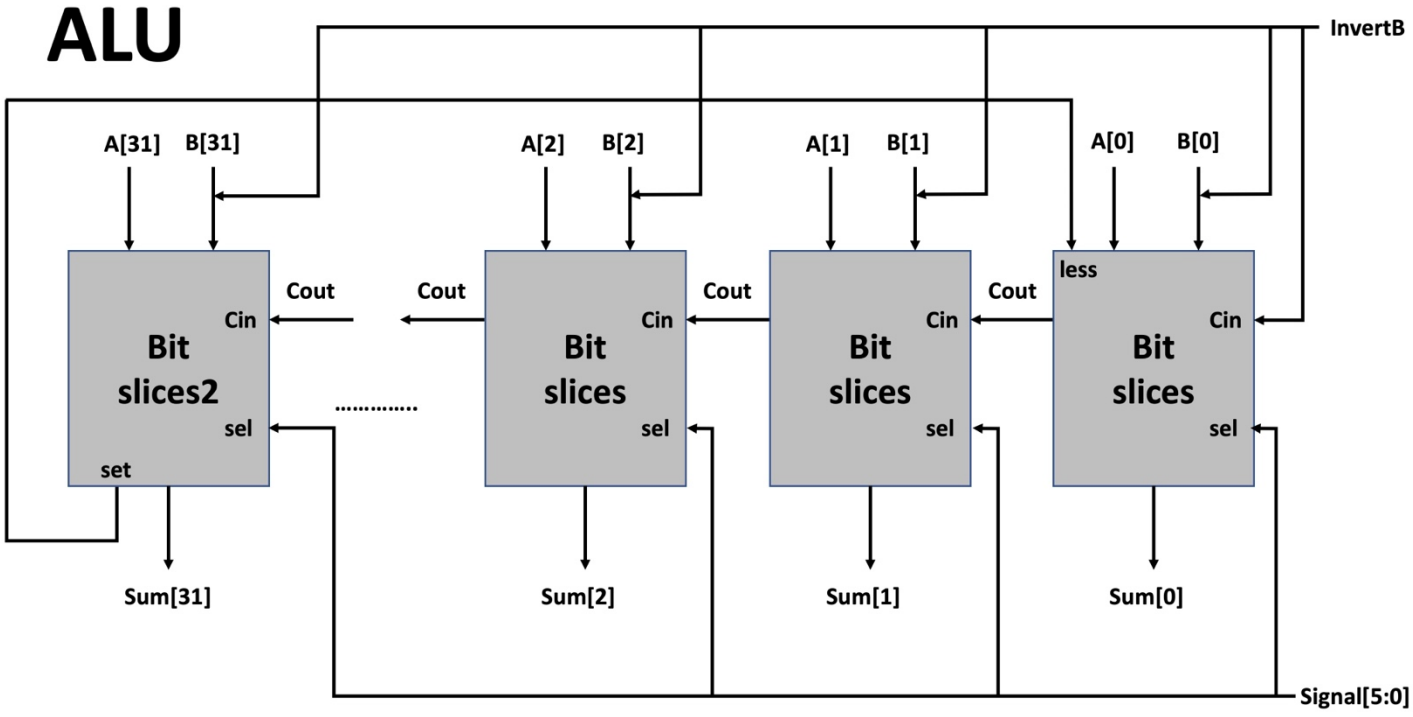
V. Stage_WB

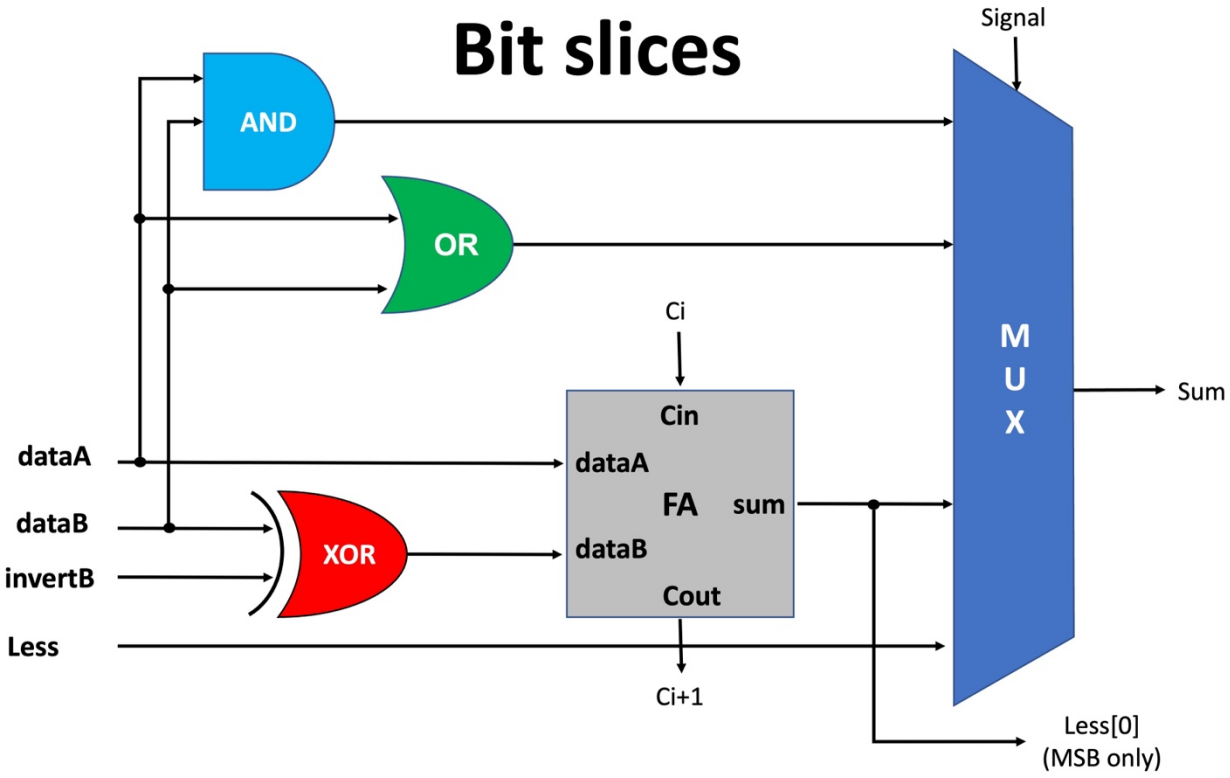


- IF_ID：IF 階與 ID 階間的暫存器，在 clk 正緣觸發時更新其儲存資料，存放 IF 階欲傳送至 ID 階之資料。
- MEM_WB：MEM 階與 WB 階間的暫存器，在 clk 正緣觸發時更新其儲存資料，存放 IF 階欲傳送至 ID 階之資料。
- ID_EX：ID 階與 EX 階間的暫存器，在 clk 正緣觸發時更新其儲存資料，存放 IF 階欲傳送至 ID 階之資料。
- EX_MEM：EX 階與 MEM 階間的暫存器，在 clk 正緣觸發時更新其儲存資料，存放 IF 階欲傳送至 ID 階之資料。

(2) ALU:

ALU 部份我們一共包含六個 module : alu、alu_ctl、Bitslices、Bitslices2、Full Adder、ALU_Shifter





每做一次 ALU 32bits 的運算時，我們會將 input A 跟 input B 以 1bit 進入 Bitslices 32 次(第 32 次進入 Bitslices2:特別多一個 set)分別做 1bit 的 AND、OR、ADD、SUB 跟 SLT (Full Adder 運算)，最後再由輸入的訊號選擇要輸出的結果。

```
//      Signal ( 6-bits)
//      AND    : 36          6'b100100
//      OR     : 37          6'b100101
//      ADD    : 32          6'b100000
//      SUB    : 34          6'b100010
//      SLT    : 42          6'b101010
//      SLL    : 00          6'b000000
//      ORI    : 13          6'b001101
```

只有多一個 ctl 的控制訊號輸入，此訊號為 alu_ctl 的 output ALUOperation，共三位元，最低兩個位元當作 Bitslices 的 sel，最高位元為 invertB 的輸入。

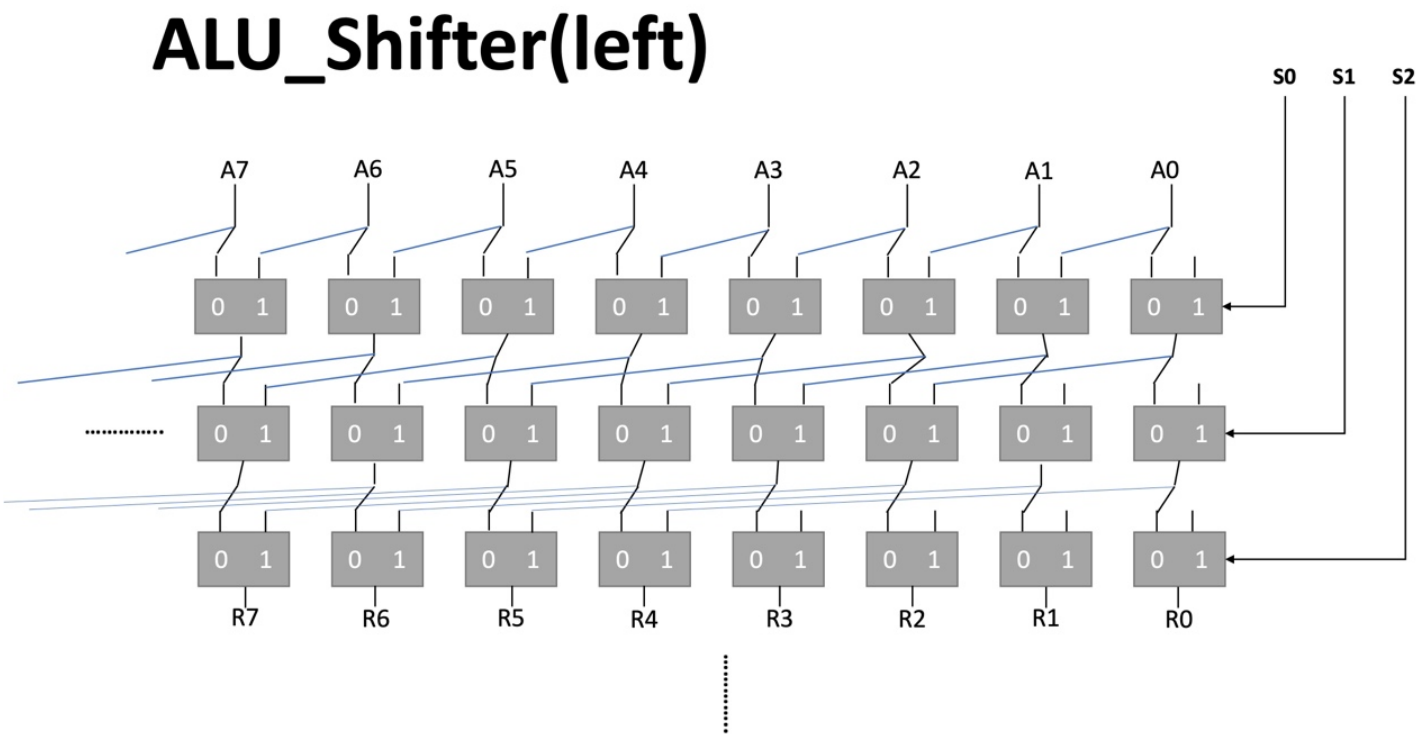
alu_ctl:

根據 ALUOp 及 Funct 訊號決定 ALUOperation 以配送至 ALU。

ALUOp 為 00 時表強制加法，01 為強制減法，若為 10 則靠 Funct 判斷 ALU 的操作(ALUOperation)。

ALU_Shifter:

在期中的 project 中獨立，在期末的 project 我們將他包含在 alu 模組內，將 32bits 的輸入依照二進位的 shift，依照這樣層層堆疊，實現 barrel shifter 的作法，用於 SLL 指令。



- AND :

input A 跟 input B 以每一位元進入 Bit slices 後，用 AND 邏輯閘運算再由輸入的訊號選擇要輸出的結果

- OR :

input A 跟 input B 以每一位元進入 Bit slices 後，用 OR 邏輯閘運算再由輸入的訊號選擇要輸出的結果

- ORI :

類似於 OR，只是 or 是將 rs 的內容 OR rt 的內容寫回 rd，而 ori 則是將 rs 的內容 or immed 寫回 rt，因此只要確保進 ALU 時的 inputA 是 rs、inputB 是 immed、最終 DataMemory 的 WD 是 rt，其餘皆都和 ADD 相同。

- SLL : 往左移的指令，以 160 個二對一多工器(Barrel shifter)實現 32 位元之移位，只是將期中 project 獨立的 shifter 移至 alu 模組內。
- ADD, SUB 跟 SLT (Full Adder 運算) :
  - ADD, SUB :

    在進入 Bit slices 之前會先判斷 Signal (輸入訊號)，如果是 SUB 或 SLT 會先將 invertB 設為一位元 1；其餘設為一位元 0。

    input A 跟 input B 以每一位元進入 Bit slices 後，input B 先和 invertB 用 XOR 邏輯閘運算，再把 1bit 的 carry in、input A 跟 XOR 邏輯閘運算結果帶入 Full Adder 運算做全加法，會得出 1bit 的 carry out（會當作下一個 Bit slices 的 carry in）、sum，最後再由輸入的訊號選擇要輸出的結果。

    如果 Signal (輸入訊號)是 SUB，要把第一個 Bit Slices 所帶入的 carry in 設為 1，因為做 SUB 運算時要把 input B 做 2 的補數(也就是 XOR 邏輯閘運算結果後加一) ，把 1bit 的 carry in、input A 跟 XOR 邏輯閘運算結果帶入 Full Adder 運算做全加法，會得出 1bit 的 carry out（會當作下一個 Bit slices 的 carry in）、sum，最後再由輸入的訊號選擇要輸出的結果。

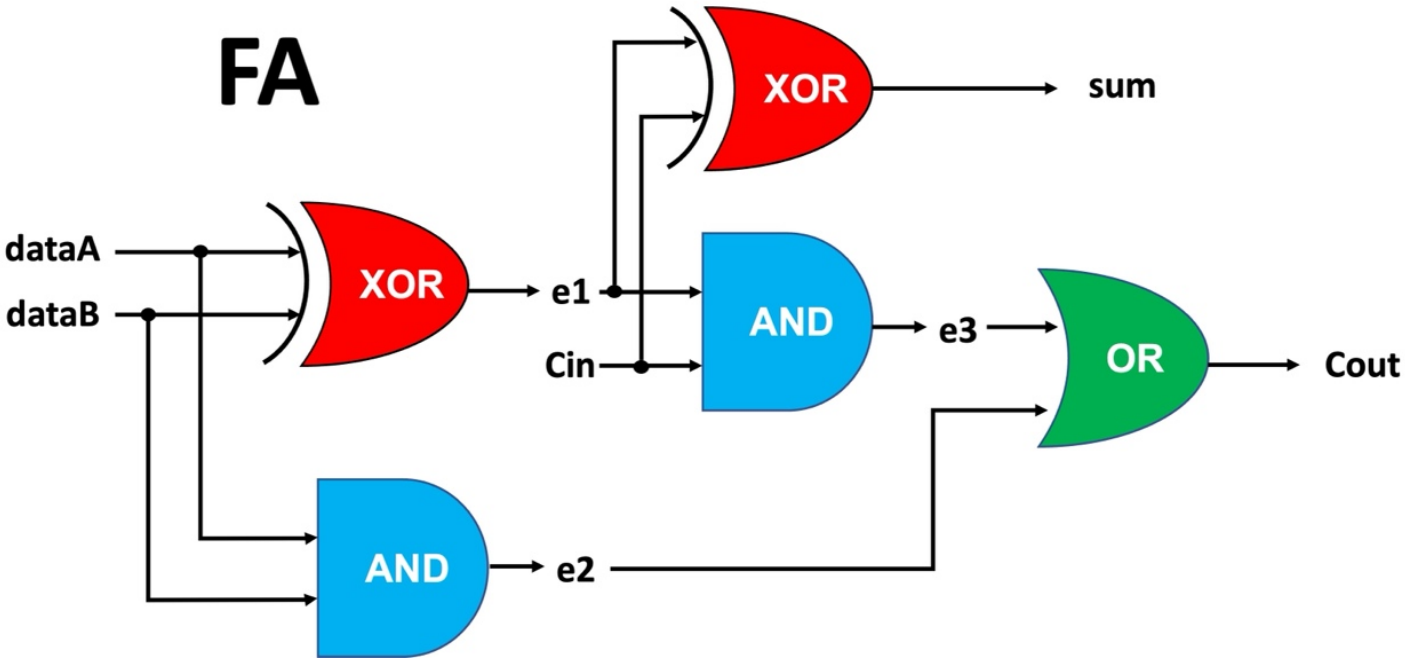
- SLT :

一樣 input A 跟 input B 以每一位元進入 Bit slices 後做完 32 位元，由於 SLT 結果的 1~31 位元皆是 0，所以直接把 0 帶入 Bit slices 裡的 less，如果 less 為 1 則輸出 1、如果 less 為 0 則輸出 0，只有在第 0 位元的 less 變數帶入 inputA 減 input B 結果的最高位元，如果 A 大於 B，結果為 0，A 小於 B 結果則為 1。

- 全加器 Full Adder：

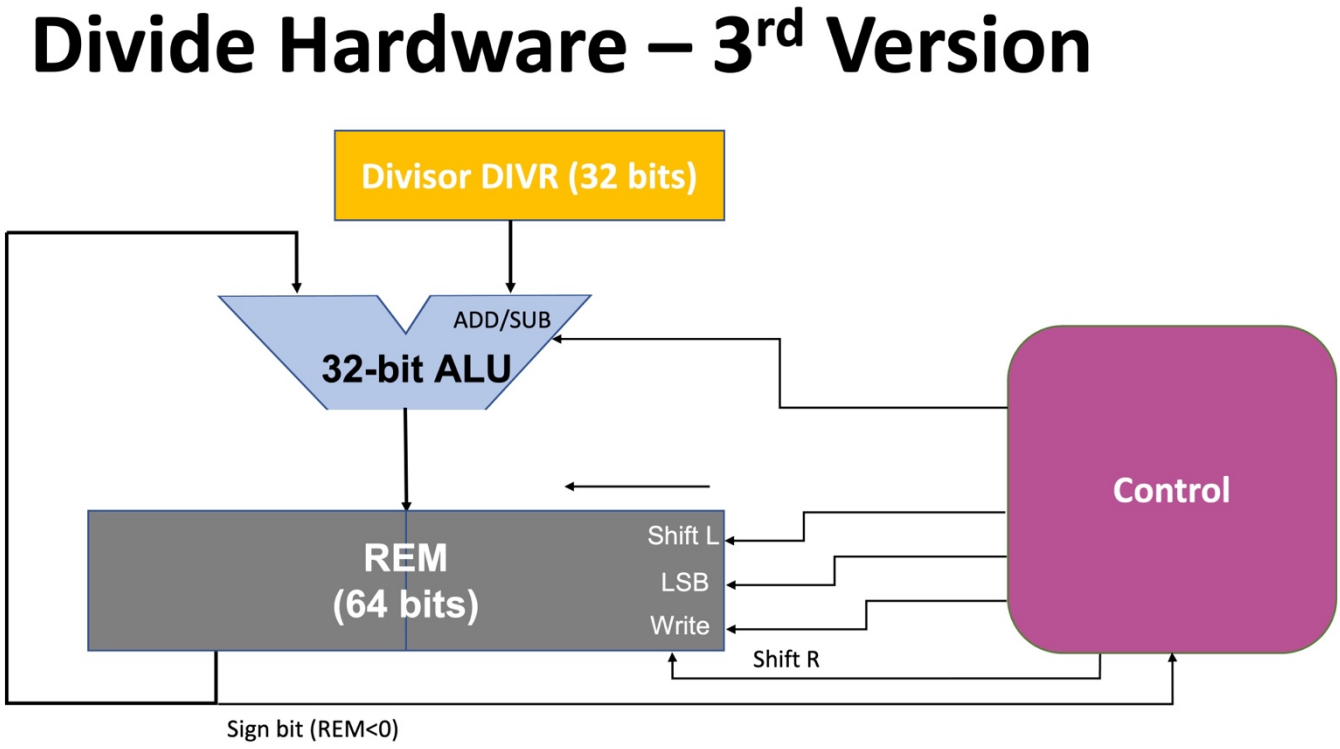
data A 跟 data B 以 1bit 輸入，carry in 為上個運算之進位，sum 是加法運算結果 carry out 為運算之進位位元

- Sum: sum = data A XOR data B XOR data C
- Carry: cout = data A* data B+ data A* data C+ data B* data C

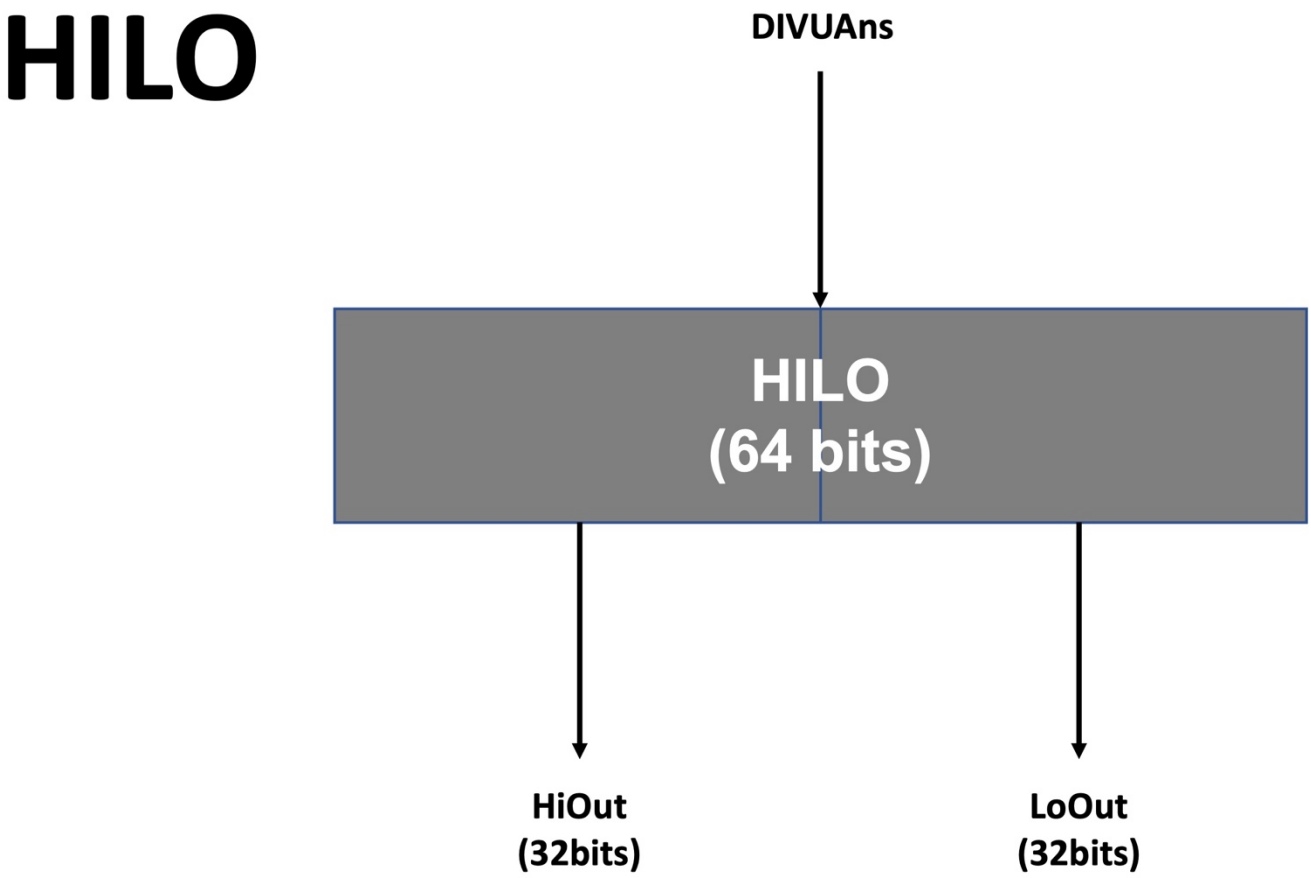


(3) Division Hardware:

DIivider: 跟期中作業一樣，裡面一樣是做第三版的無號數除法，不一樣的地方在於外面的接線。這次外面接了 ALU_Control 和 Div_clk，藉由這兩者來判斷何時要做和要做什麼。當 ALU_ctl 傳入 Funct，他便會依照不同 case 來做。總共有 3 個 case，分別是 DIVU、MFHI 和 MFLO，DIVU 就是做最原本的第三版無號數除法，不過這次將除法做完後右移一位元的動作丟到後面兩個 case，本身沒有將所有事情做完，所以也不會輸出運算結果。而 MFHI 和 MFLO 則是先把除法最後一步驟做完再告訴後面的 HILO 要開始做事情了，並且把算完的結過輸出出去。而裡面的 HILO_signal 會告訴 HILO 要搬的是哪一個部分。



HILO:這部分也是和期中的差不多，只是這次是由 Divider 裡的 HILO_signal 輸出來判斷究竟要做 MFHI 還是 MFLO 的指令。如果 Divider 輸出的 HILO_signal 是 0，那麼就輸出餘數，如果 HILO_signal 是 1，那麼就輸出商。



(4) 其他:

I. add32 :

實現 32 位元加法，此模組用於 PC 及 branch 的位址加法。

II. control_single :

根據 6 bits 的 opcode 產生對應的控制訊號：

	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Jump	ALUOp
R_FORMAT	1	0	0	1	0	0	0	10
ORI	0	1	0	1	0	0	0	00
LW	0	1	1	1	1	0	0	00
SW	x	1	x	0	0	1	0	00
BEQ	x	0	x	0	0	0	0	01
BNE	x	0	x	0	0	0	0	01
J	x	0	x	0	0	0	1	01

III. mips_single：

宣告元件類型跟名稱，整合所有 module 以及處理訊號傳送，然後依照課本的架構圖將線拉好。

IV. Memory：

依據MemRead及MemWrite控制Memory的寫入或讀取，若(MemRead, MemWrite) = (0, 1)，執行寫入的功能，將 wd 寫入至 mem[addr]。

若(MemRead, MemWrite) = (1, 0)，執行讀取的功能，表需寫回 Register File，將 mem[addr]傳給 rd。

V. Mux2：

二對一多工器，根據訊號擇一輸出，主要用在選擇 PC 要 fetch 下一道指令的位址、ALU 的 operand2 輸入、欲寫回的暫存器以及內容。

VI. reg32:：

32 位元的暫存器，他會從 input 的 en_reg 判斷可不可以寫入，之後將所讀取的資料寫入打算寫進的暫存器位址。

VII. reg_file：

32 個 32 位元的暫存器。一開始會先記錄 source1、source2 的暫存器編號還有目的地的暫存器編號和欲寫入的資料。之後先將從 source1 和 source2 所讀取的資料存起來，然後將資料寫入目的地位址的地址。

VIII. sign_extend：

位元擴充。將 16 位元的輸入變成 32 位元，在輸入資料的前面補上 16 個 0。

三、結果

1.Icarus Verilog 驗證結果：

```
C:\Users\work>cd Downloads

C:\Users\work\Downloads>cd mips_single

C:\Users\work\Downloads\mips_single>vvp alu.vvp
VCD info: dumpfile mips_single.vcd opened for output.
WARNING: tb_SingleCycle.v:33: $readmemh(instr_mem.txt): Not enough words in the file for the requested range [0:1023].
WARNING: tb_SingleCycle.v:34: $readmemh(data_mem.txt): Not enough words in the file for the requested range [0:1023].
WARNING: tb_SingleCycle.v:36: $readmemh: The behaviour for reg[...] mem[N:0]; $readmemh("...", mem); changed in the 1364-2005 standard. To avoid ambiguity, use mem[0:N] or explicit range pa
rameters $readmemh("...", mem, start, stop);. Defaulting to 1364-2005 behavior.
                  0, reading data: Mem[         x] => xxxxxxxx
                  0, reading data: Mem[         0] => 8e2f0000
                  9, PC:         0
                  9, LW

                 10, reading data: Mem[         4] => 12310004
                 10, reg_file[17] =>         2 (Port 1)
                 10, reg_file[15] =>        21 (Port 2)
                 19, PC:         4
                 19, BEQ

                 20, reading data: Mem[         8] => 16310003
                 20, reg_file[17] =>         2 (Port 2)
                 29, PC:         8
                 29, BNE

                 30, reading data: Mem[        12] => 16320002
                 30, reading data: Mem[         2] => 00000100
                 39, PC:        12
                 39, BNE

                 40, reading data: Mem[         0] => 01000000
                 40, reading data: Mem[        16] => 02509020
                 40, reg_file[18] =>         3 (Port 2)
                 50, reg_file[15] <=       256 (Write)
                 49, PC:        16
                 49, wd:       256
                 49, ADD

                 50, reading data: Mem[        24] => 08000009
                 50, reg_file[18] =>         3 (Port 1)
                 50, reg_file[16] =>         1 (Port 2)
                 59, PC:        24
                 59, J
```

```
 60, reading data: Mem[        28] => 02509025
 60, reg_file[ 0] =>          0 (Port 1)
 60, reg_file[ 0] =>          0 (Port 2)
 69, PC:        28
 69, wd:         x
 69, OR

 70, reading data: Mem[        24] => 08000009
 70, reg_file[18] =>          3 (Port 1)
 70, reg_file[16] =>          1 (Port 2)
 79, PC:        24
 79, J

 80, reading data: Mem[        28] => 02509025
 80, reg_file[ 0] =>          0 (Port 1)
 80, reg_file[ 0] =>          0 (Port 2)
 90, reg_file[18] <=          4 (Write)
 89, PC:        28
 89, wd:         4
 89, OR

 90, reading data: Mem[        36] => 36310f0f
 90, reg_file[18] =>          4 (Port 1)
 90, reg_file[16] =>          1 (Port 2)
 99, PC:        36
 99, ORI

100, reading data: Mem[        40] => 00000000
100, reg_file[17] =>          2 (Port 1)
100, reg_file[17] =>          2 (Port 2)
110, reg_file[18] <=          3 (Write)
109, PC:        40
109, wd:         3
109, NOP

110, reading data: Mem[        36] => 36310f0f
110, reg_file[ 0] =>          0 (Port 1)
110, reg_file[ 0] =>          0 (Port 2)
119, PC:        36
119, ORI

120, reading data: Mem[        40] => 00000000
120, reg_file[17] =>          2 (Port 1)
120, reg_file[17] =>          2 (Port 2)
130, reg_file[18] <=          5 (Write)
129, PC:        40
129, wd:         5
129, NOP
```

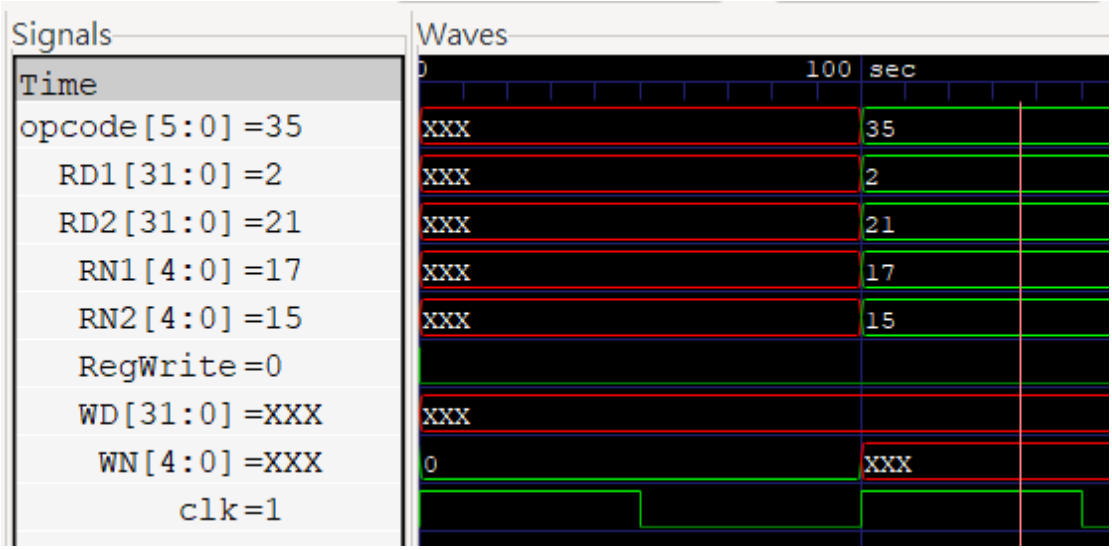
```
130, reading data: Mem[        44] => 02529080
130, reg_file[ 0] =>          0 (Port 1)
130, reg_file[ 0] =>          0 (Port 2)
139, PC:         44
139, wd:       3855
139, SLL

140, reg_file[17] <=       3855 (Write)
140, reading data: Mem[        48] => 02d7982a
140, reg_file[18] =>          5 (Port 1)
140, reg_file[18] =>          5 (Port 2)
149, PC:         48
149, wd:          0
149, SLT

150, reading data: Mem[        52] => 0232001b
150, reg_file[22] =>          7 (Port 1)
150, reg_file[23] =>          8 (Port 2)
159, PC:         52
159, wd:       3855
159, DIVU

160, reg_file[17] <=       3855 (Write)
160, reading data: Mem[        56] => 00009810
160, reg_file[17] =>       3855 (Port 1)
160, reg_file[18] =>          5 (Port 2)
169, PC:         56
169, wd:          0
169, MFHI

170, reading data: Mem[        60] => 00009812
170, reg_file[ 0] =>          0 (Port 1)
170, reg_file[ 0] =>          0 (Port 2)
179, PC:         60
179, wd:         20
179, MFLO
```

以上為 16 道指令於 terminal 輸出的結果，能觀察出指令之暫存器數值與輸出結果相符合。

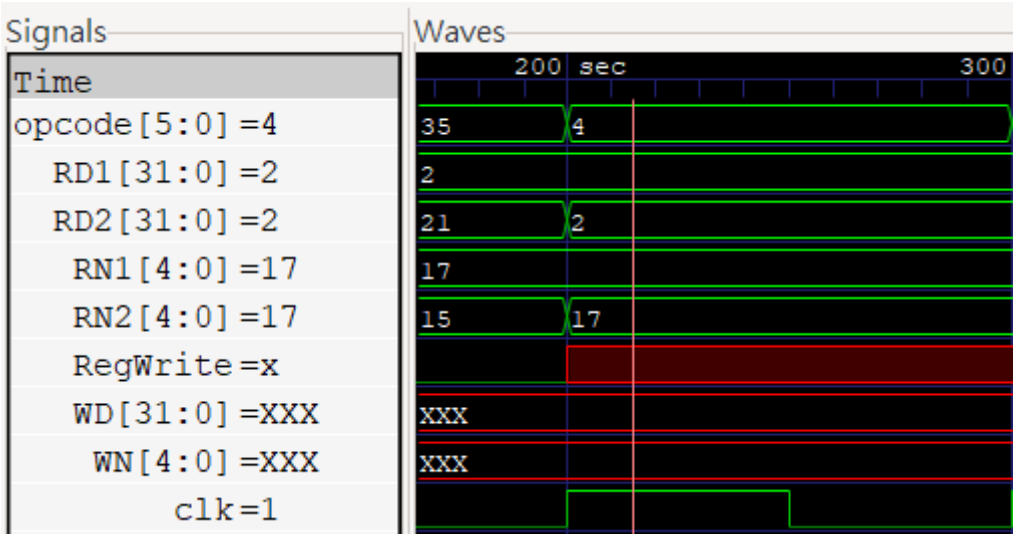
2.Waveform 輸出圖形：

分成 16 道指令

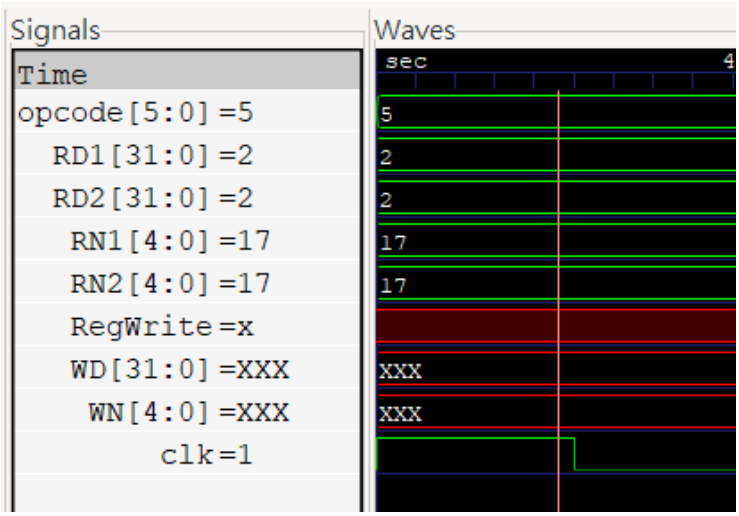
(1)LW



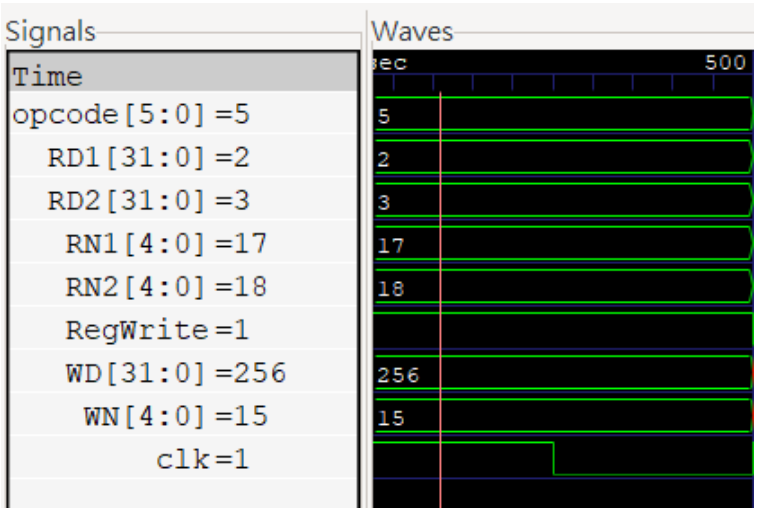
(2)BEQ



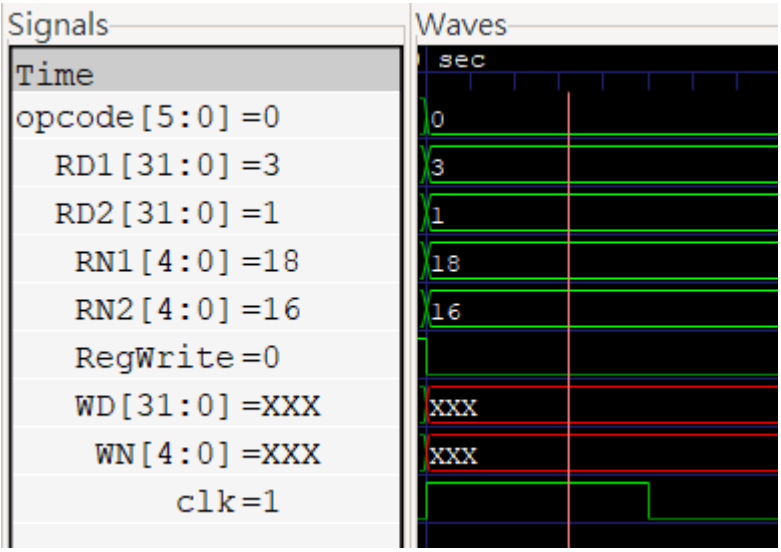
(3)BNE



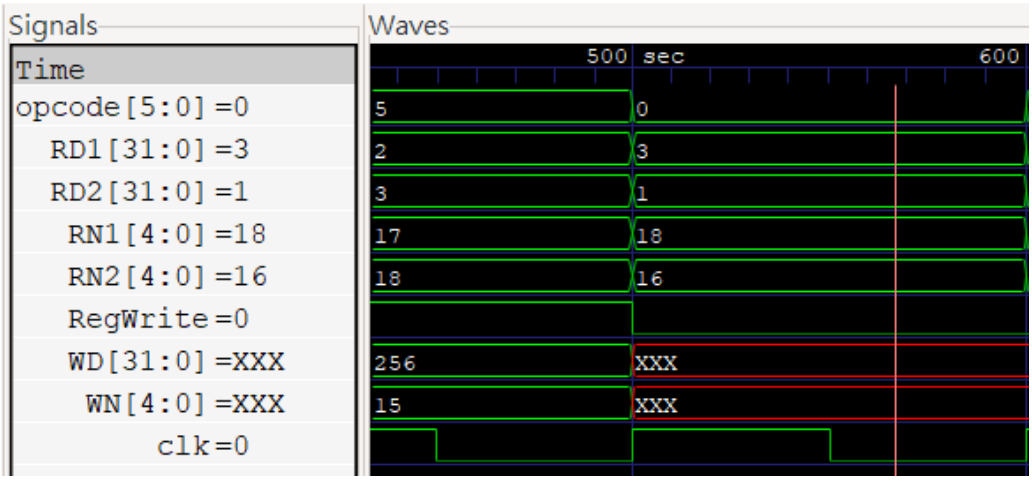
(4)BNE



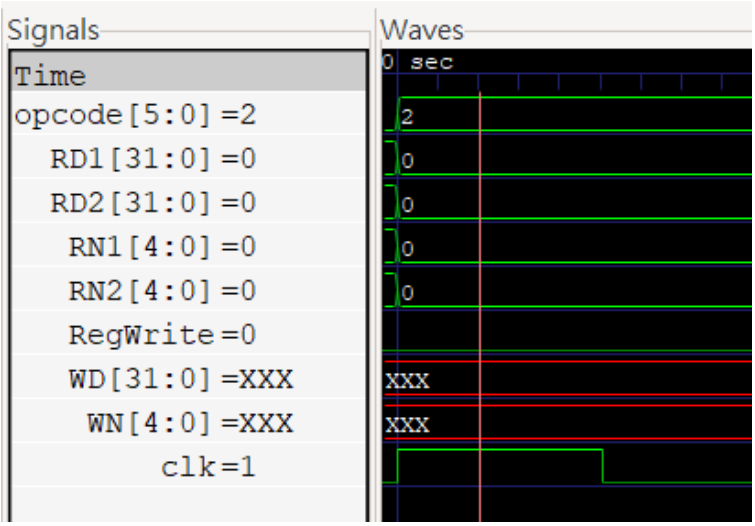
(5)ADD



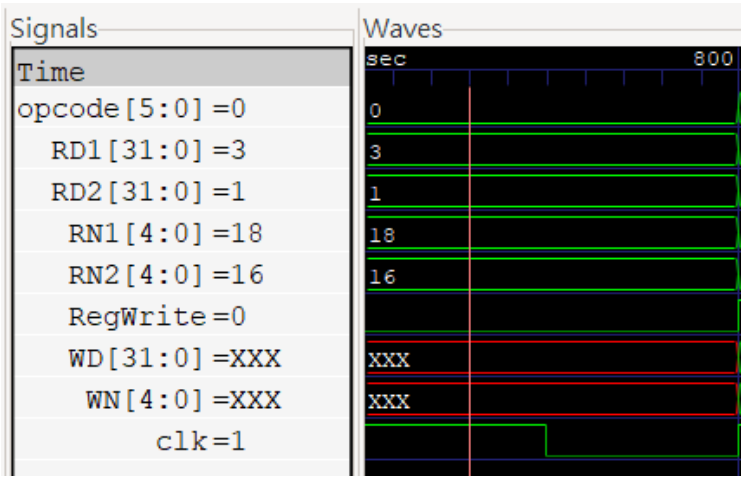
(6)SUB


Signals
Waves
Time
opcode[5:0]=0
RD1[31:0]=3
RD2[31:0]=1
RN1[4:0]=18
RN2[4:0]=16
RegWrite=0
WD[31:0]=XXX
WN[4:0]=XXX
clk=0
500 sec
600
5
0
2
3
3
1
17
18
18
16
256
XXX
15
XXX

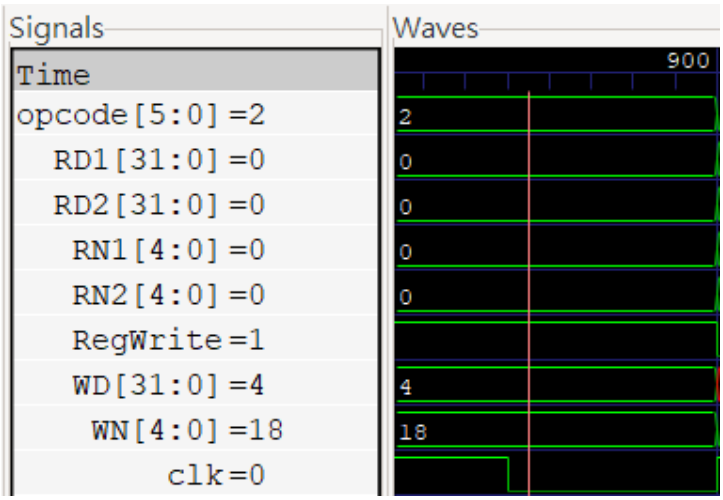
(7)J


Signals
Waves
Time
opcode[5:0]=2
RD1[31:0]=0
RD2[31:0]=0
RN1[4:0]=0
RN2[4:0]=0
RegWrite=0
WD[31:0]=XXX
WN[4:0]=XXX
clk=1
0 sec
2
0
0
0
0
XXX
XXX

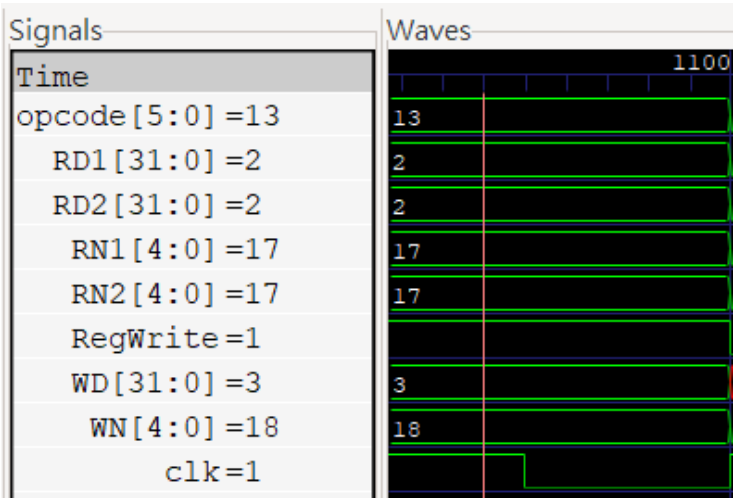
(8)OR


Signals
Waves
Time
opcode[5:0]=0
RD1[31:0]=3
RD2[31:0]=1
RN1[4:0]=18
RN2[4:0]=16
RegWrite=0
WD[31:0]=XXX
WN[4:0]=XXX
clk=1
sec
800
0
3
1
18
16
XXX
XXX

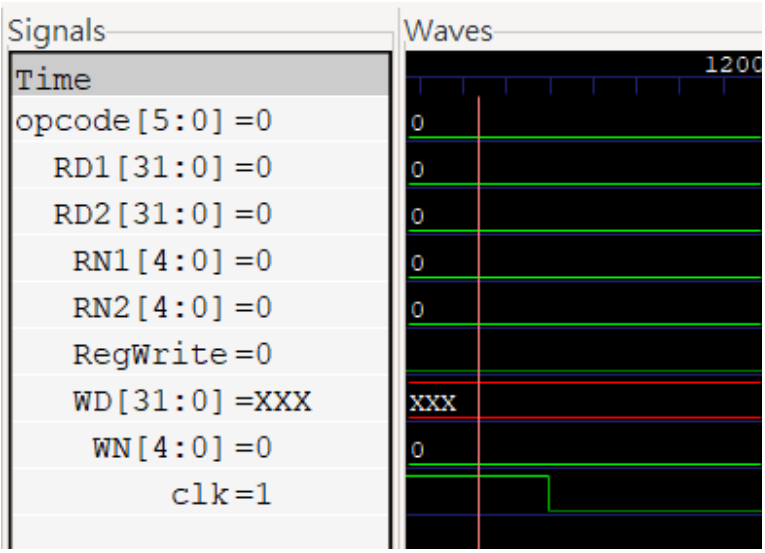
(9)SW



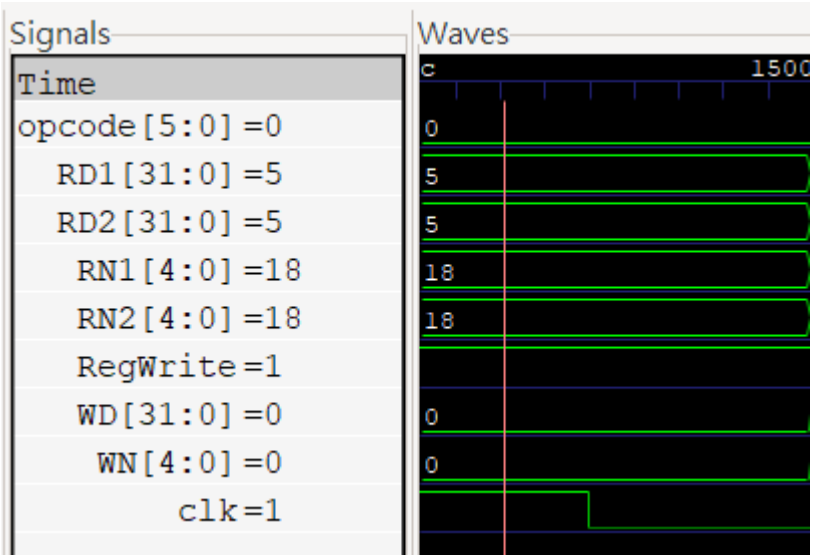
(10)ORI



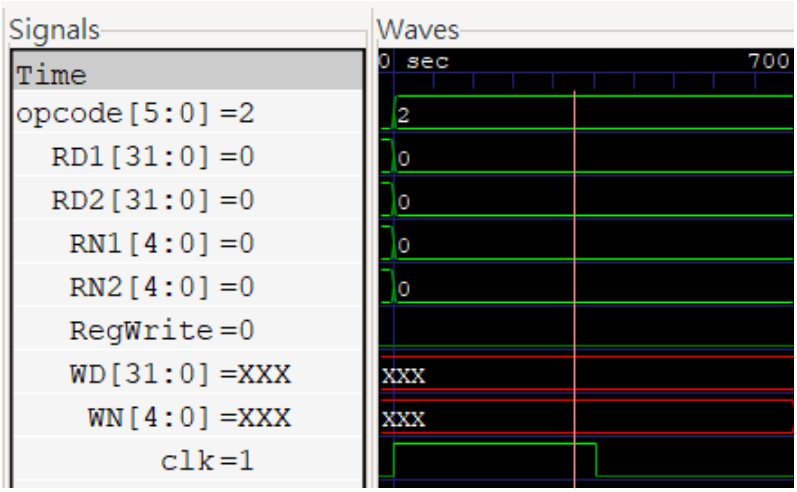
(11)NOP



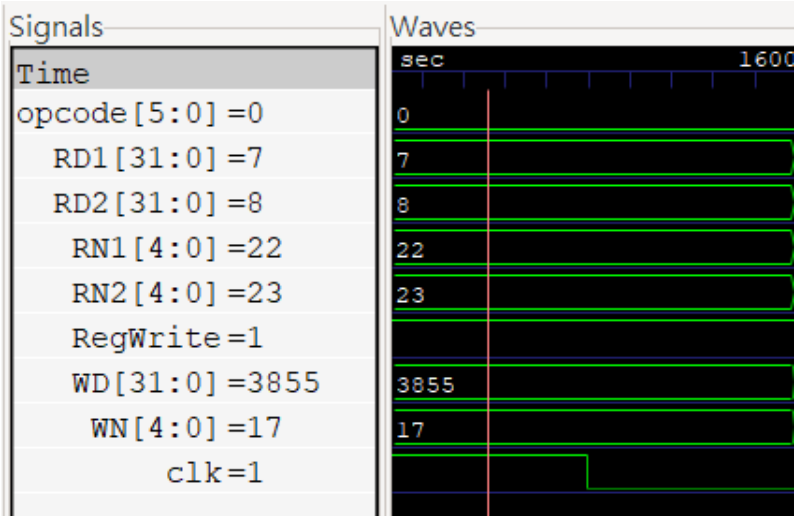
(12)SLL



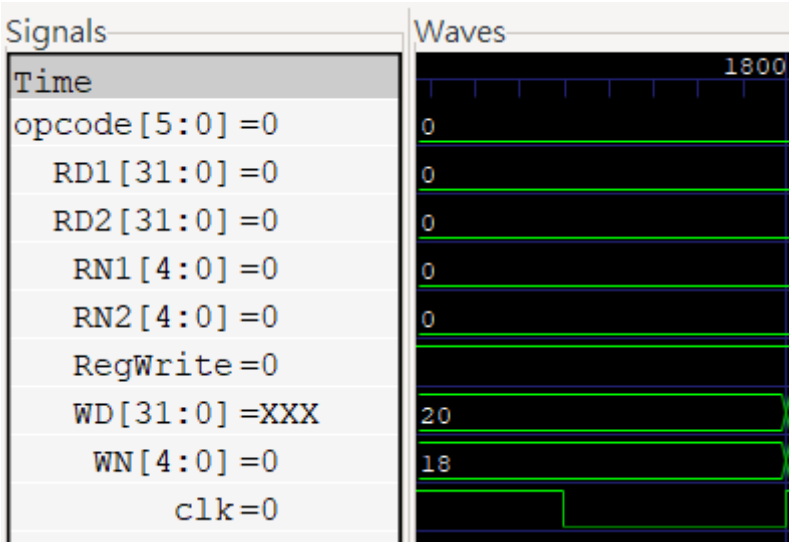
(13)SLT



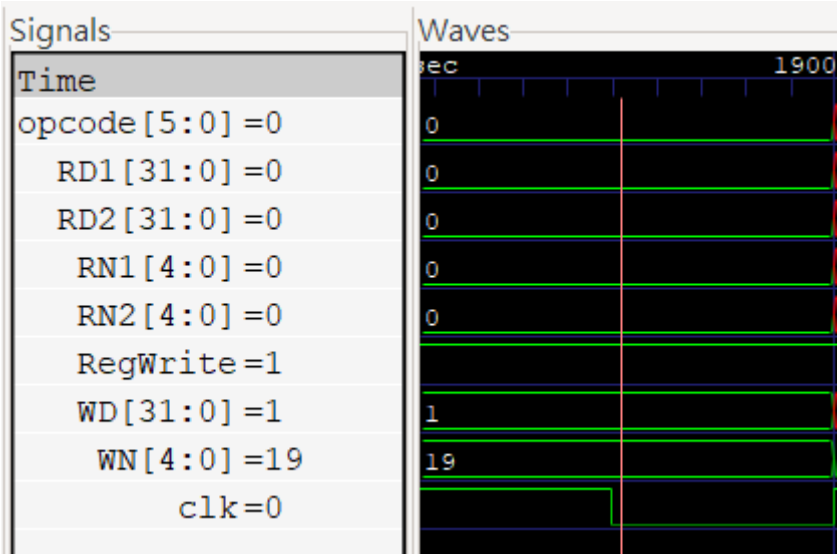
(14)DIVU



(15)MFHI

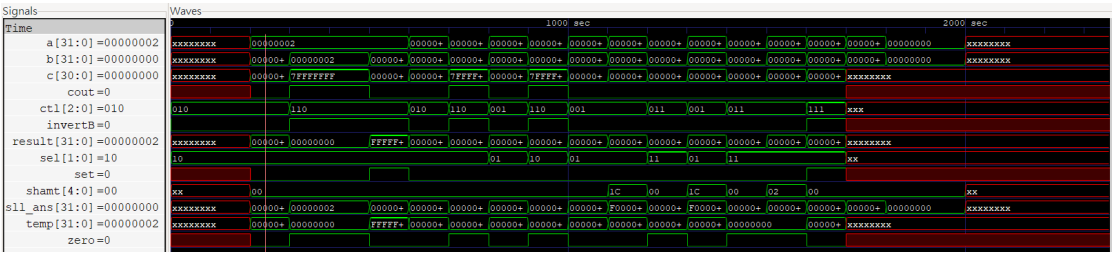


(16)MFLO

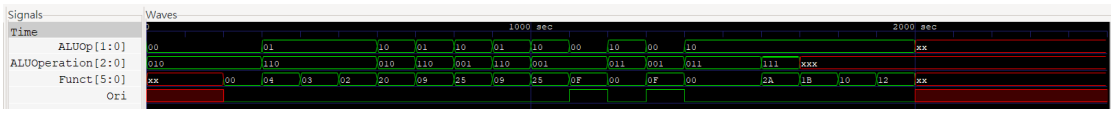


分成 24 個 Module

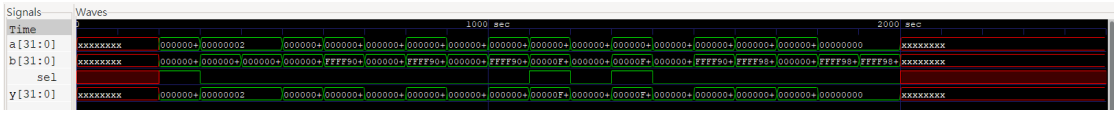
(1)ALU



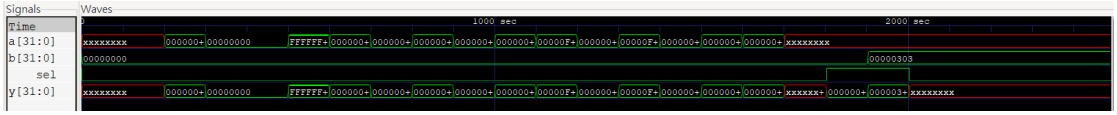
(2)ALUCTL



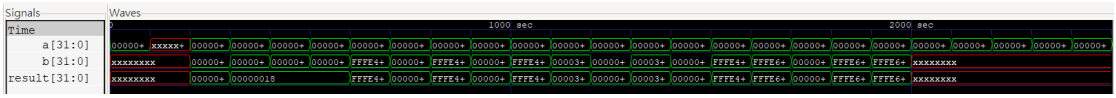
(3)ALUMUX



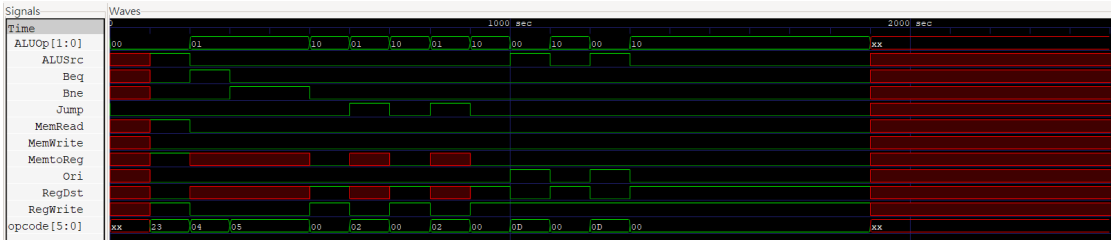
(4)ALUOUTMUX



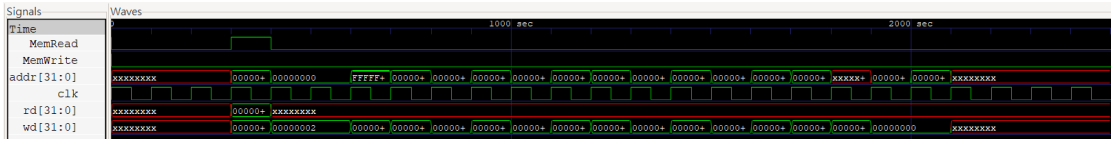
(5)BRADD



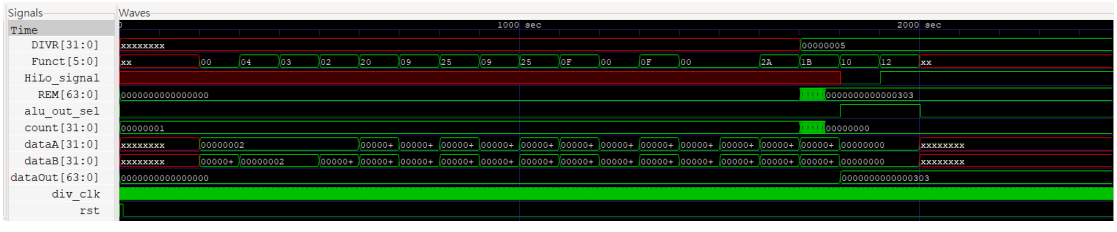
(6)CTL



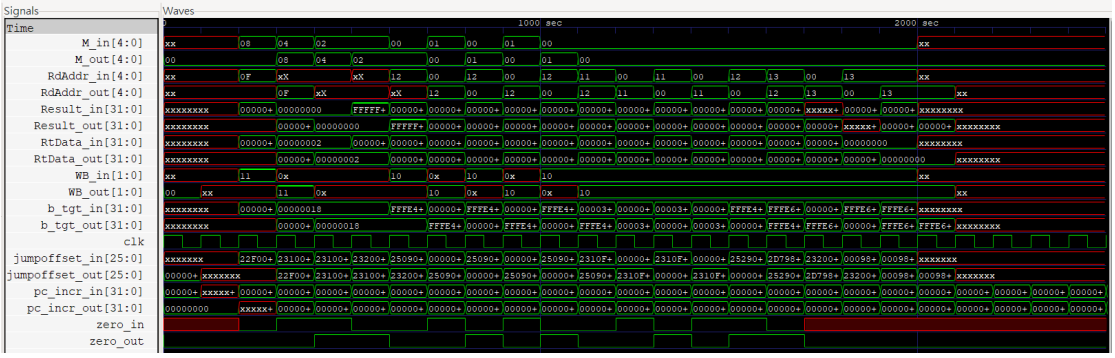
(7)DatMem



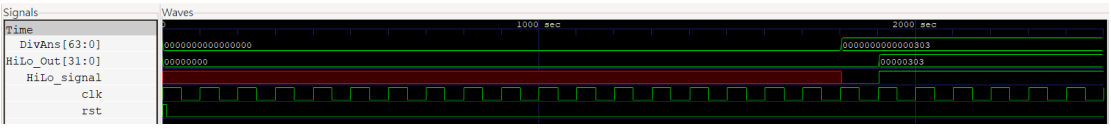
(8)Divider



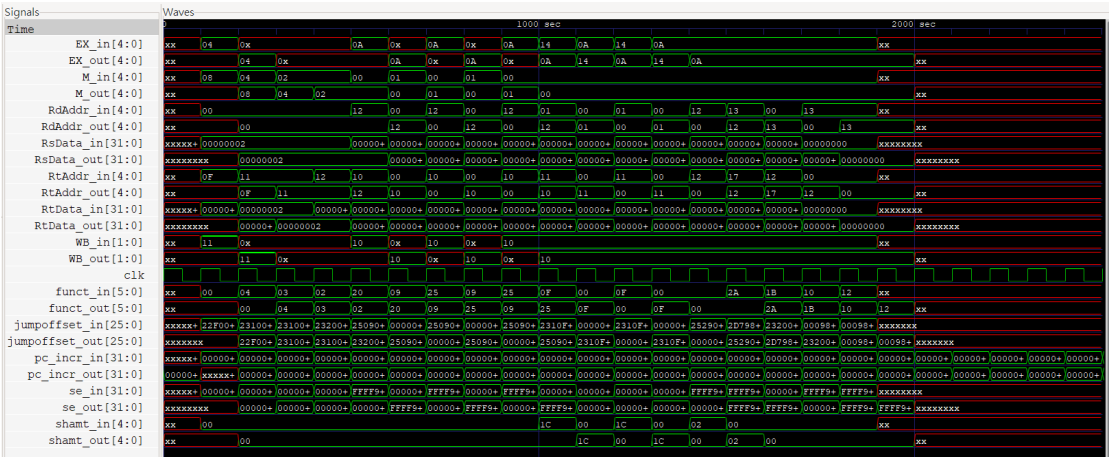
(9)EX_MEM



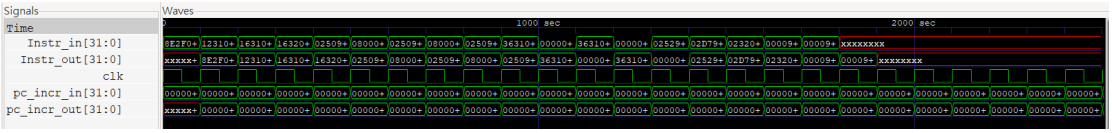
(10)HiLo



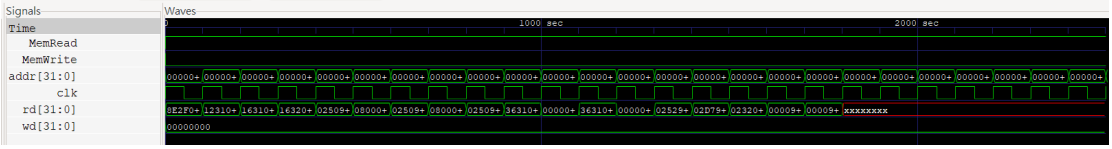
(11)ID_EX



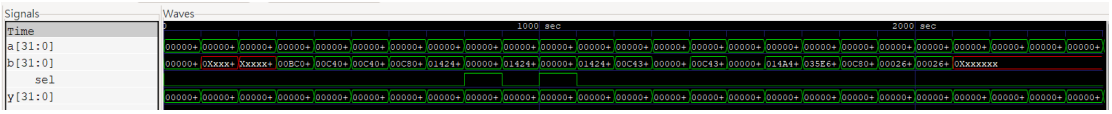
(12)IF_ID



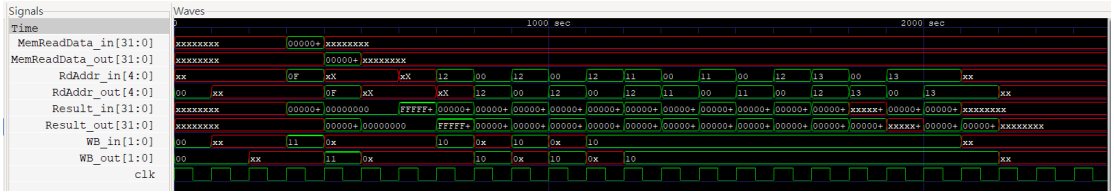
(13)InstrMem



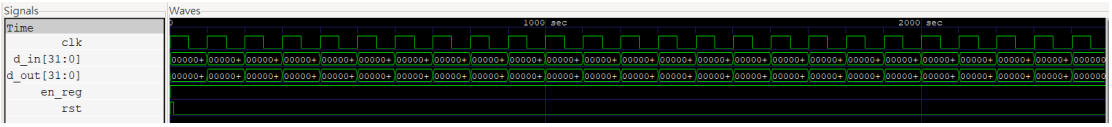
(14)JMUX



(15)MEM_WB

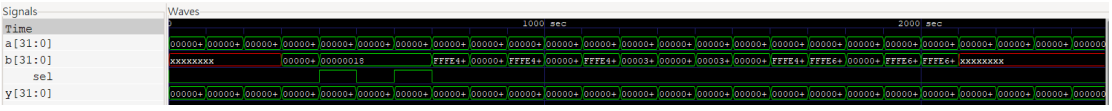


(16)PC

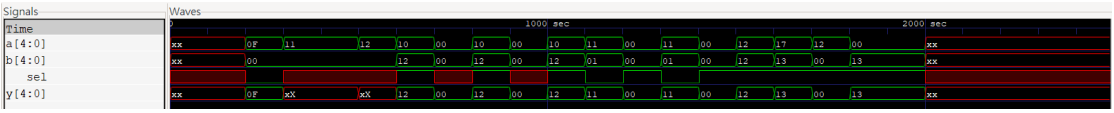


(17)PCADD

(18)PCMUX

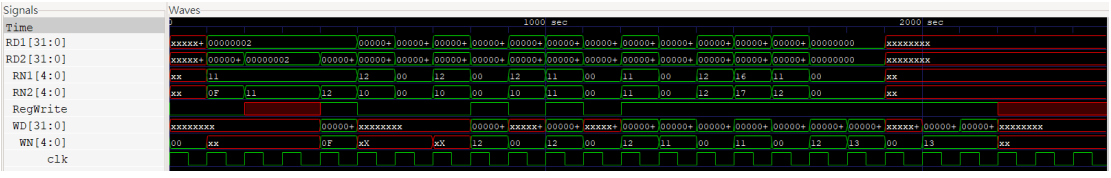


(19)RFMUX

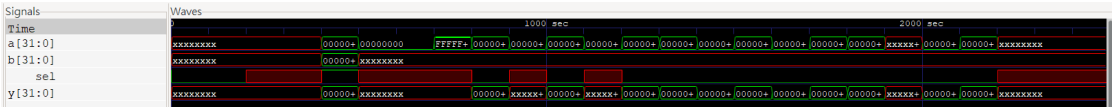


(20)RegFile

(21)SignExt



(22)WRMUX



四、討論

各組員分工方式與負責項目:

10833230 李郁含

alu、alu_ctl、Bitslices、Bitslices2、Full Adder、ALU_Shifter 撰寫、製作計算機組織成果報告

10844132 黃中惘

reg32 、reg_file、mux_2to1、sign_extend、Divider、HILO 撰寫、撰寫之程式說明

10844149 謝宜庭

整個 project 接線 code 之撰寫、撰寫之程式說明、畫架構圖

五、結論 六、未來展望

心得：

10833230 李郁含

這次的期末 project 跟期中比難度簡直是直線上升，課本講義上的架構圖複雜了 100 倍，光是一開始理解題目、老師原本的 code 跟作業要求就花了將近一個禮拜，更別說後面的擴增指令的撰寫和各個暫存器的接線更是讓我們頭痛不已，我們遠距後的時間幾乎都花在研究 5-stage-pipeline cpu 上了。

雖然在這個艱難關卡的 project 我們遇到了不少災難和關卡，比如說跟期中的 ALU 比多了幾道指令該怎麼解決、還有此次作業 shifter 要包在 alu 模組裡、pipeline 的暫存器需要什麼訊號、除法器訊號要怎麼給跟最後整個 project 的接線，不過這次的 project 也讓我們對 pipeline cpu 的運作又有更進一步的認識也讓我們對於 verilog 的撰寫更大幅度地進步，且最後因為大家分散各地分 part 寫的寫完的程式碼們一起執行肯定會有大量的 bug 需要 de，不過大家一起同心協力討論 debug ，最後看到成果出來的那刻真的相當欣慰及感動，這次的作業其實不好分 part 因此我們大家都相當配合，每人都付出了努力在這份作業上，也才能即時並順利地完成作業。

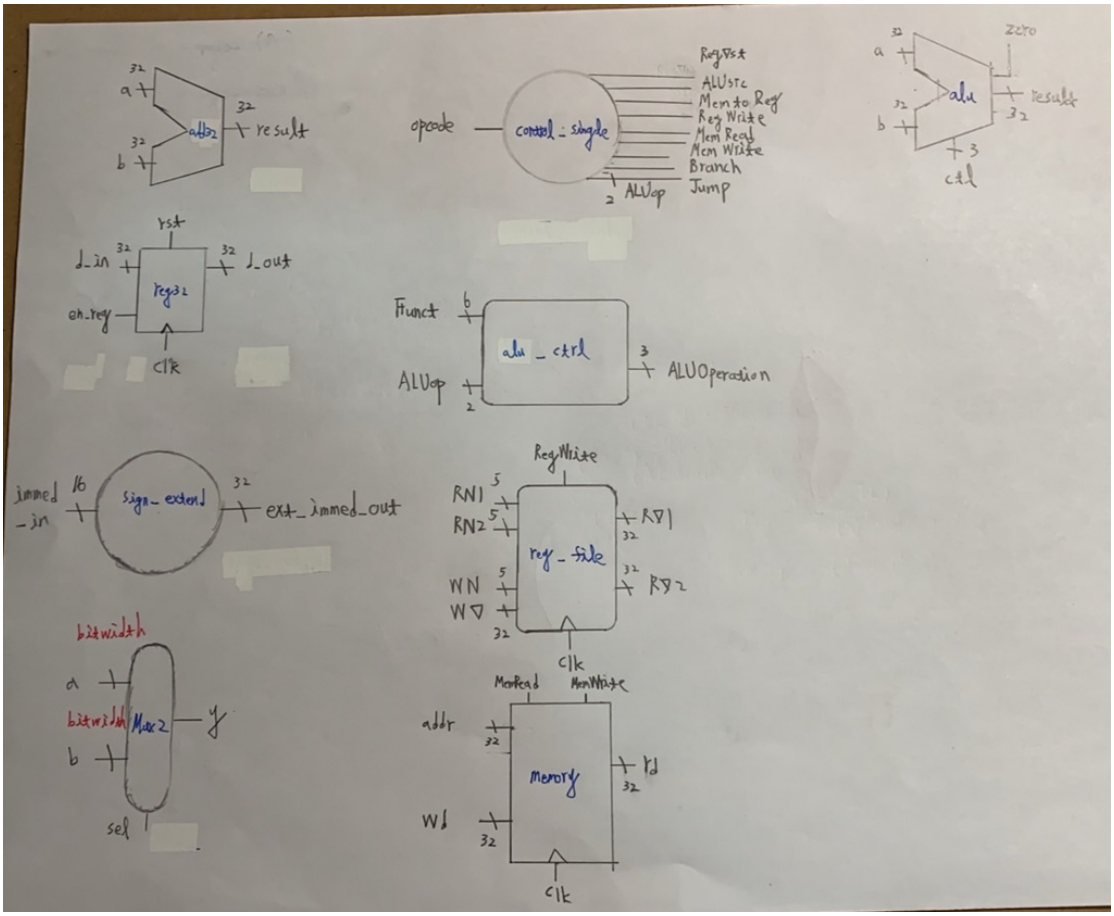
10844132 黃中惘

這次的作業真的好難，期中作業和這次的比根本就是小巫見大巫……這次真的是多虧我的組員才能在截止日前將這份作業完成。我負責的只有 DIVU 的部分，但光這一部分就讓我頭痛了很久，雖然裡面內容沒有變很多，但是實際上完全不一樣，外面的接線和指令輸入輸出，只是多了這兩個部份就差了很多……因為要考慮到除法器他要接在什麼東西上，指令是從哪邊傳進來，後面輸出時要怎麼告訴後面的 HILO 要做 MFHI 或者 MFLO 這兩道指令。當所有東西都和在一起時，整體的難度就上了好幾階。不過真的是還好我的組員願意幫助我，陪我一起寫和找 Debug，不然我不覺得我有辦法做出這次的作業。

10844149 謝宜庭

這次的作業比期中困難很多，光是理解好 datapath 就花了一些時間，接線的部分更是錯綜複雜。為了順利完成接線，在這之前也畫了各個元件的實際樣貌(in out 線路)，這部分一定要完全理解架構圖才能順利完成。整合的部分也遇到好多困難，因為當時跟大家約了一天要將自己的部分寫完，但實際理解才發現，我的部分需要用到其他人的線路，因此接完固有的指令後，等組員寫完再修改線路與、重新拉線。還有將 Divu 與 Alu 丟到 datapath 電路也花很多心力，因為用的參數名稱都不同整合起來就很麻煩。雖然真的很趕，最後我們還是在期限前完成，而且因為這次的作業，我們也學到很多關於設計 CPU 的相關資訊。

附圖是各個元件的樣貌


32
a
32
b
add32
32
result
opcode
control_single
RegDst
ALUSrc
MemtoReg
RegWrite
MemRead
MemWrite
Branch
Jump
2
ALUop
32
a
32
b
alu
zero
result
32
3
ctl
rst
32
d_in
32
d_out
reg32
en_reg
clk
Funct
6
ALUop
2
alu_ctrl
3
ALUOperation
immed_in
16
sign_extend
32
ext_immed_out
RegWrite
5
RN1
5
RN2
5
WN
WD
32
reg_file
32
RD1
32
RD2
clk
bitwidth
a
bitwidth
b
Mux2
y
sel
MemRead
MemWrite
addr
32
memory
32
RD
WD
32
clk