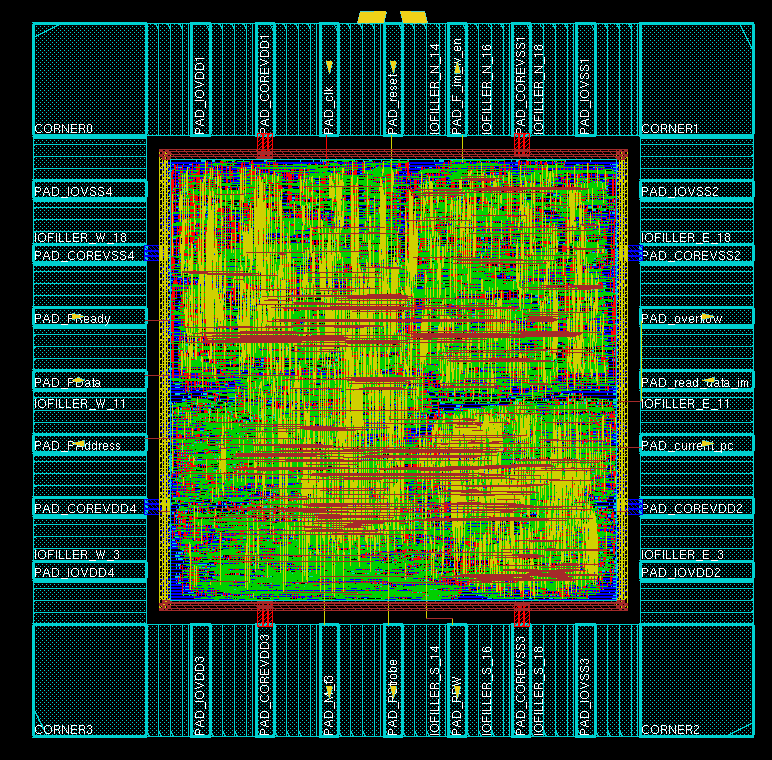
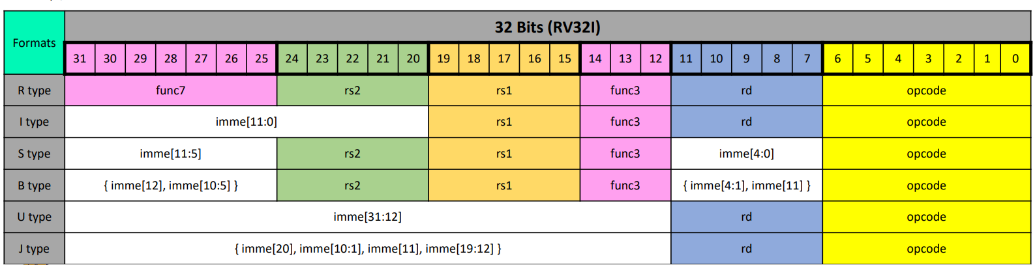
**VLSI系統設計**

**-期末專題報告**



|  |  |  |
| --- | --- | --- |
| **組別:** | | |
| **姓名** | **學號** | **授課教師** |
| 林琮偉 | E24096093 | **李昆忠** |
| 林伯璨 | E24092049 |
| 吳宜哲 | E24096085 |
| 陳正崙 | E24096239 |
| 張根齊 | E24096483 |
| 林軒宇 | E24092057 |

1. **系統簡介**
   1. 指令集格式



* 1. 指令集格式欄位的名稱、長度、說明

|  |  |  |
| --- | --- | --- |
| 指令集格式欄位的名稱 | 長度 | 說明 |
| func7 | 7個bits | 協助opcode判斷指令種類 |
| func3 | 3個bits | 協助opcode判斷指令種類 |
| rs1 | 5個bits | Source register 1 address |
| rs2 | 5個bits | Source register 2 address |
| rd | 5個bits | Destination register address |
| opcode | 7個bits | the operation of the instruction |
| imm | 根據type決定 | 常數 |

* 1. Branch指令與Jump指令的定址

3-1. Branch

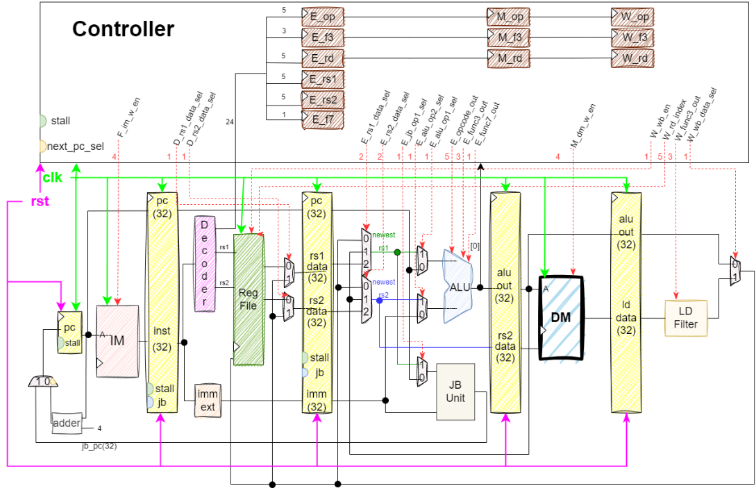
Branch 的定址方法是pc + signed extension{imm, 1’b0}，因為RSICV的IALIGN只有32和16bit，分別是對齊4的倍數以及2的倍數的記憶體地址(byte address)，對其4的倍數則前兩個bit一定是0，對齊2則是前一個bit是0，這兩種情況的imm[0]都是0，所以省略，這樣能跳轉到更遠的地方。

3-2. Jump

jal的定址方法是pc + sext({imm20, 1’b0})，串接一個0的原理和branch一樣

jalr的定址方法是= ( rs1 + sext(imm12) ) & (~32’d1)，和(~32’d1)and的原理和branch一樣

* 1. 架構說明
     1. 架構圖及說明(需自行繪製)



* + 1. 個別元件說明

1.IM : instruction memory，會在裡面存instruction，唯獨

2.Decoder: 將instruction依照指令拆成rs1、rs2、opcode…

3.immext: 提取出指令內的imm，並依照指令種類組出32bit的imm

4.RegFile: 有32個32bits的register，x0~x31，x0永遠是0

5.ALU: 將operand1和operand2依照指令種類算出alu\_out

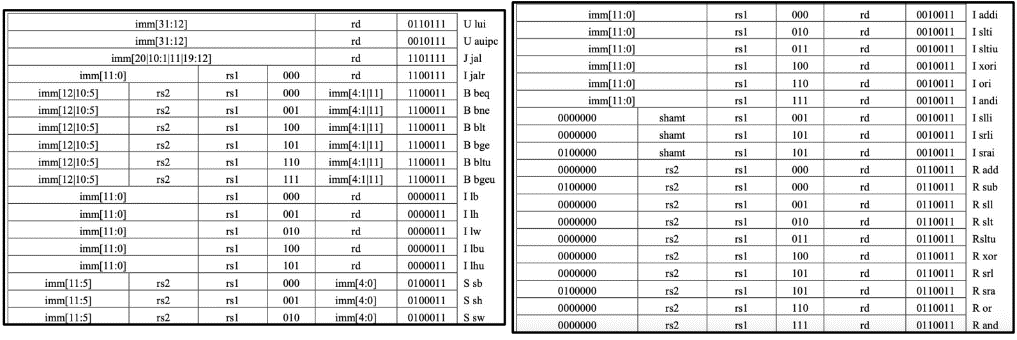
6.JB\_Unit: 計算跳轉地址

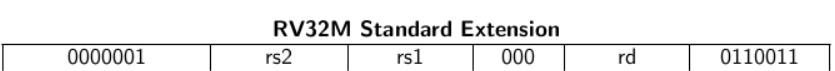
7.DM: Data memory，之後會插入特殊設計cache，這部分參考Cache

8.LD Filter: load有load byte、half word、word，由LD Filter將ld data拆成指定大小

9.Controller: 提供control signal

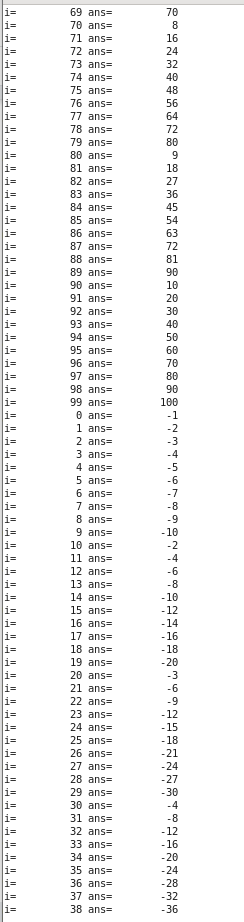
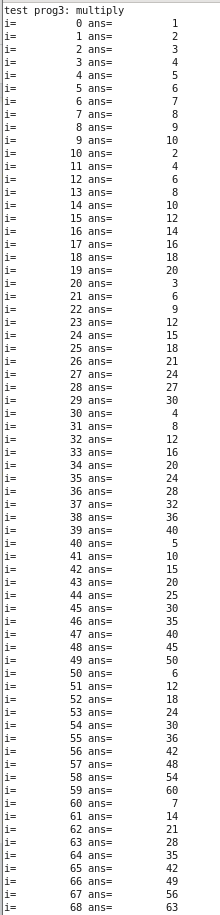
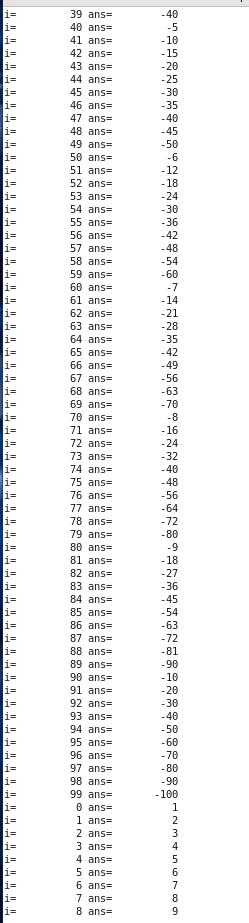
1. **系統目前可執行之指令**

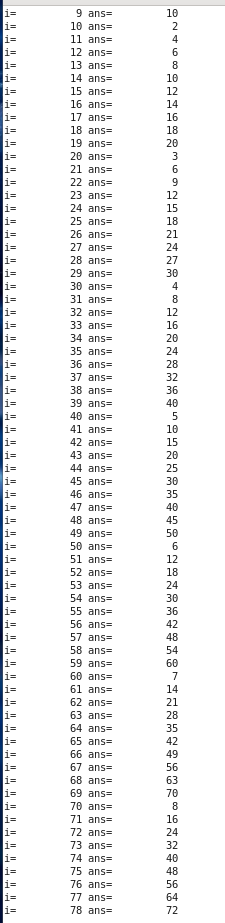
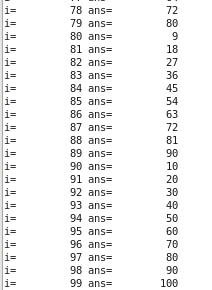
****

MUL：

1. **系統驗證方法與結果分析**
   1. 驗證方法

* 測試乘法:會執行正乘正的1~10的乘法，以及正乘負和負乘負的乘法，並將結果寫入h'100,h'300,h'500來驗證功能是否執行正確，輸出結果如下

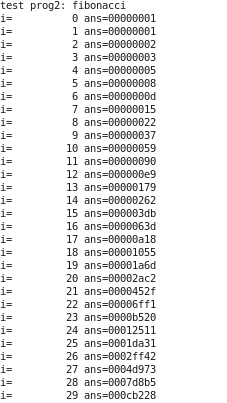




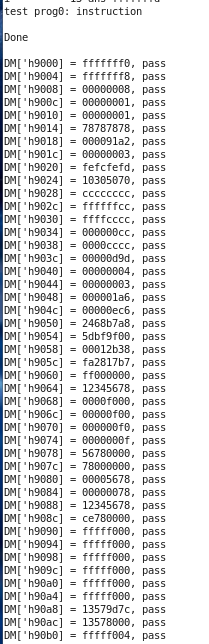
* 驗證浮點數:先讀取1,2,3,4四個數字，再執行1+2和3-4，分別得到答案3和-1再將兩數相乘得到-3，如此一來浮點數的加法減法及乘法都驗證完成，輸出結果如下，i=0~6為浮點數表示法，i=7~13為一般表示法，可以上下對照檢查答案是否正確



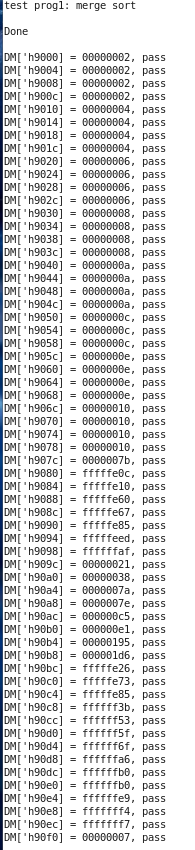
* 驗證費氏數列:會計算出前30項，並將結果存入h'100依序print出來



* 驗證instruction，有golden data來檢查是否正確



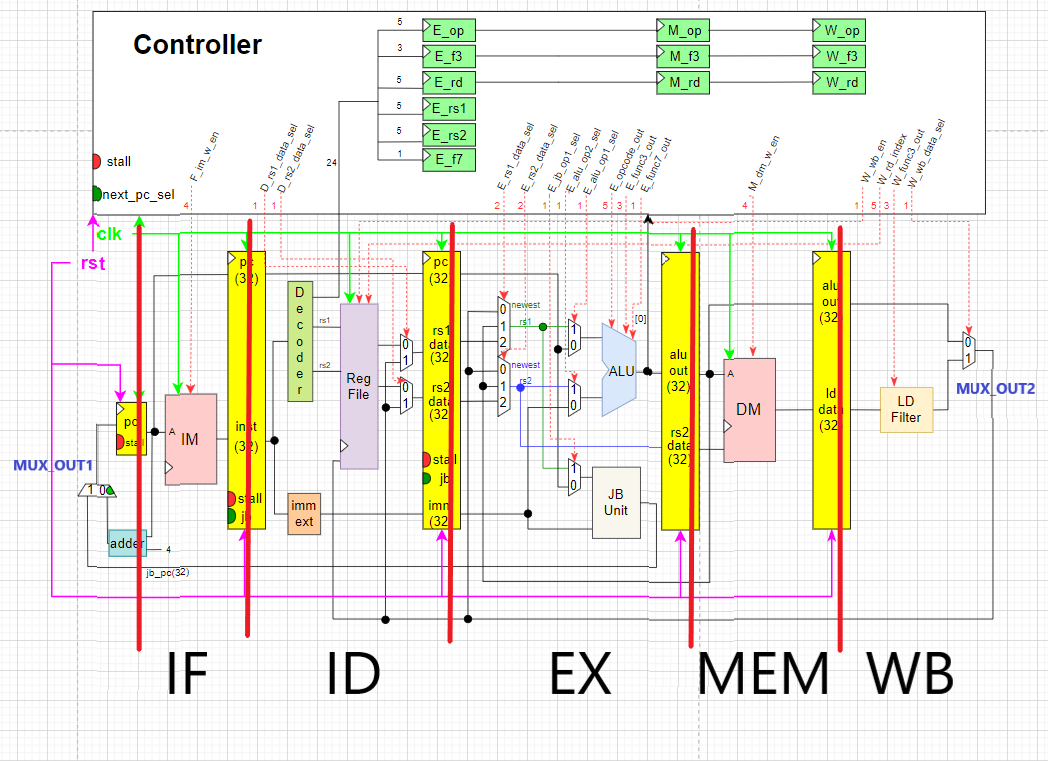
* 驗證merge sort:有golden data做比較



* 用來滿足icc toggle的測資

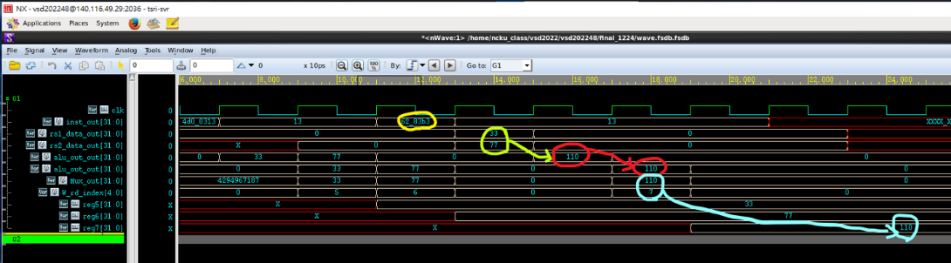


* 1. 結果分析(需附上nWave截圖)
     1. “單一”指令正確性



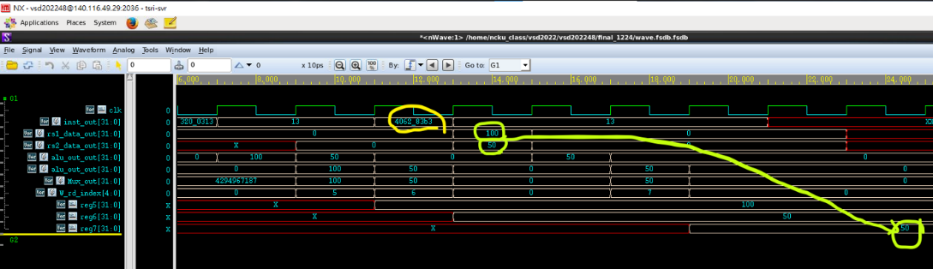
1. R type:

1-1. add rd, rs1, rs2



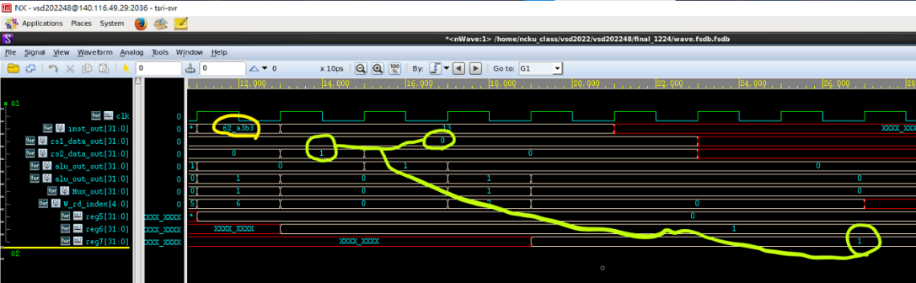
上圖黃色的是add的instr，rs1是33，rs2是77，將兩者相加得到110，最後可看到rd reg7正確存入110。

1-2. sub rd, rs1, rs2



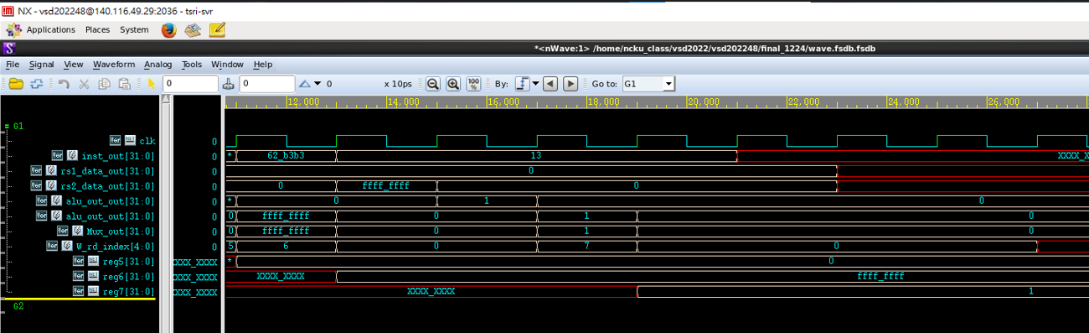
上圖黃色的是sub的instr，rs1是100，rs2是50，將兩者相減得到50，最後可看到rd reg7正確存入50。

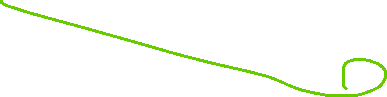
1-3. slt rd, rs1, rs2



上圖黃色的是slt的instr，rs1是0，rs2是1，rs2>rs1，所以最後可看到rd reg7正確存入1。

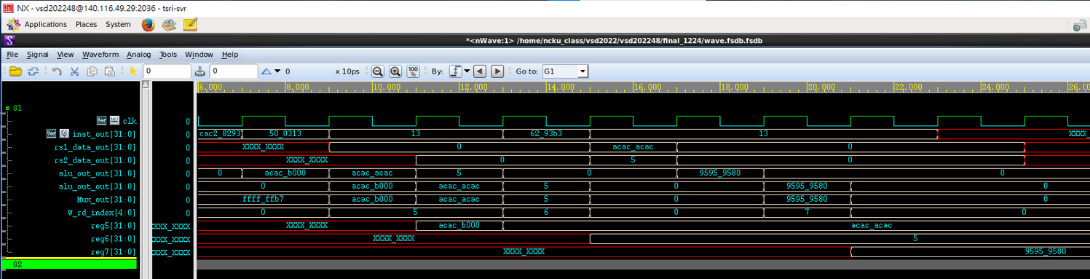
1-4. sltu rd, rs1, rs2

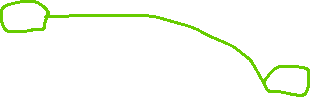




上圖黃色的是sltu的instr，rs1是0，rs2是0hffff\_ffff，unsigned rs2>rs1，所以最後可看到rd reg7正確存入1。

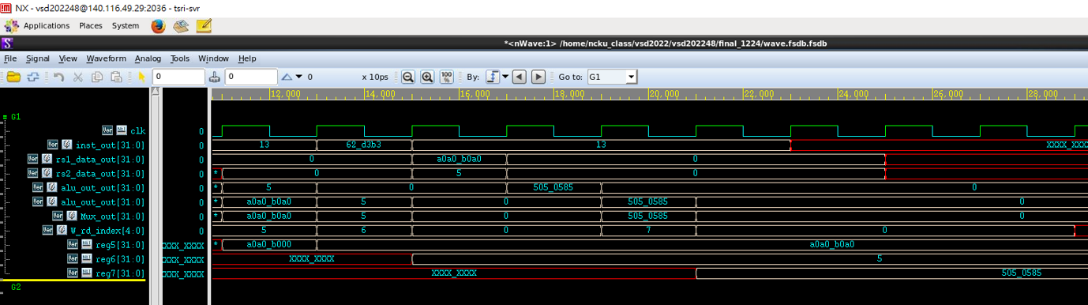
1-5. sll rd, rs1, rs2

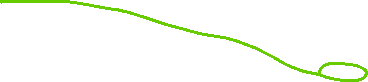




上圖黃色的是sll的instr，rs1是0hacac\_acac，rs2是5，將rs1做邏輯左移，所以最後可看到rd reg7正確存入0h9595\_9580。

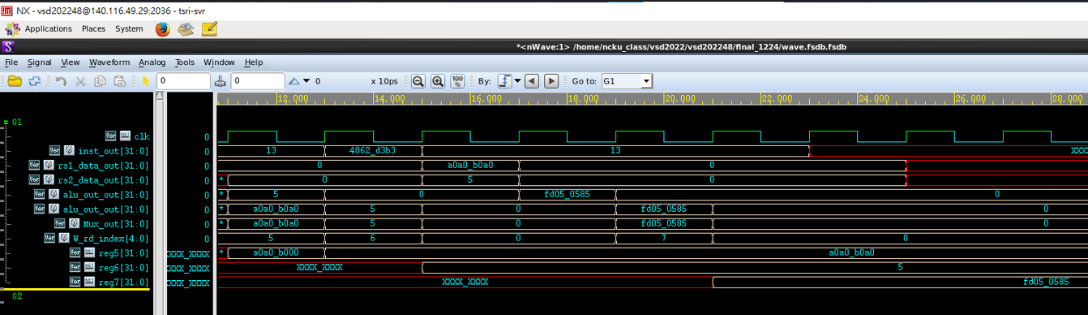
1-6. srl rd, rs1, rs2

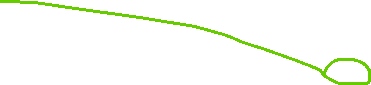




上圖黃色的是srl的instr，rs1是0ha0a0\_b0a0，rs2是5，將rs1做邏輯右移，所以最後可看到rd reg7正確存入0h0505\_0585。

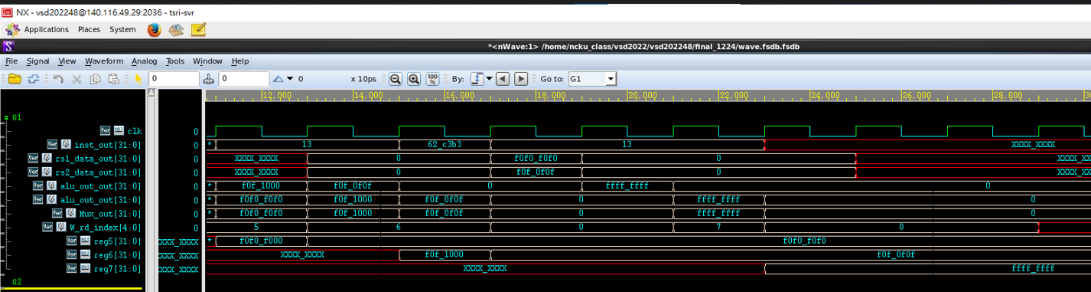
1-7. sra rd, rs1, rs2

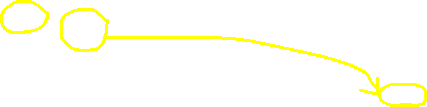




上圖黃色的是sra的instr，rs1是0ha0a0\_b0a0，rs2是5，將rs1做算術右移，所以最後可看到rd reg7正確存入0hfd05\_0585。

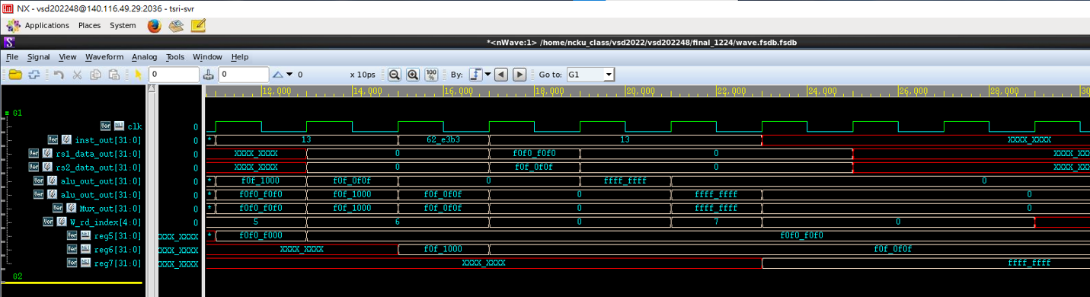
1-8. xor rd, rs1, rs2

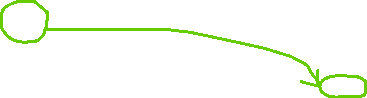




上圖黃色的是xor的instr，rs1是0hf0f0\_f0f0，rs2是0h0f0f\_0f0f，將rs1和rs2做xor，所以最後可看到rd reg7正確存入0hffff\_ffff。

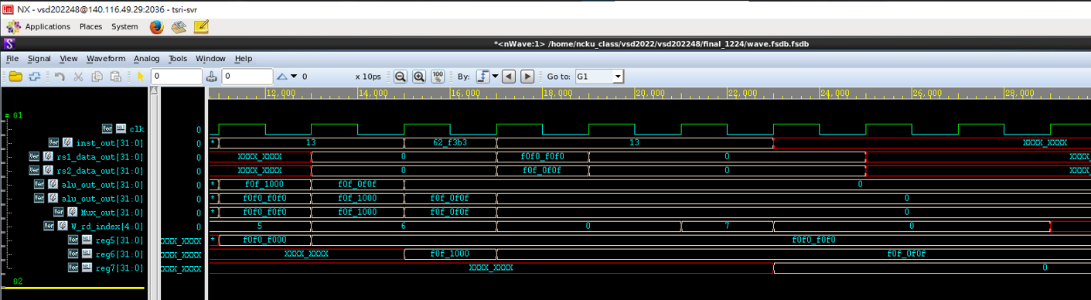
1-9. or rd, rs1, rs2

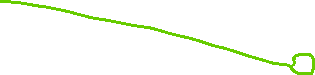




上圖黃色的是or的instr，rs1是0hf0f0\_f0f0，rs2是0h0f0f\_0f0f，將rs1和rs2做or，所以最後可看到rd reg7正確存入0hffff\_ffff。

1-10. and rd, rs1, rs2

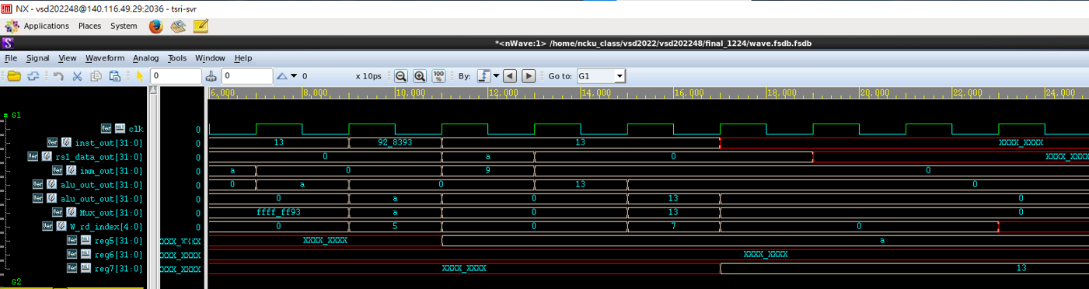


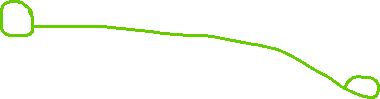


上圖黃色的是and的instr，rs1是0hf0f0\_f0f0，rs2是0h0f0f\_0f0f，將rs1和rs2做and，所以最後可看到rd reg7正確存入0。

2. I type

2-1. addi rd, rs1, imm





上圖黃色的是addi的instr，rs1是0ha，imm是0h9，將rs1和rs2做add，所以最後可看到rd reg7正確存入0h13。

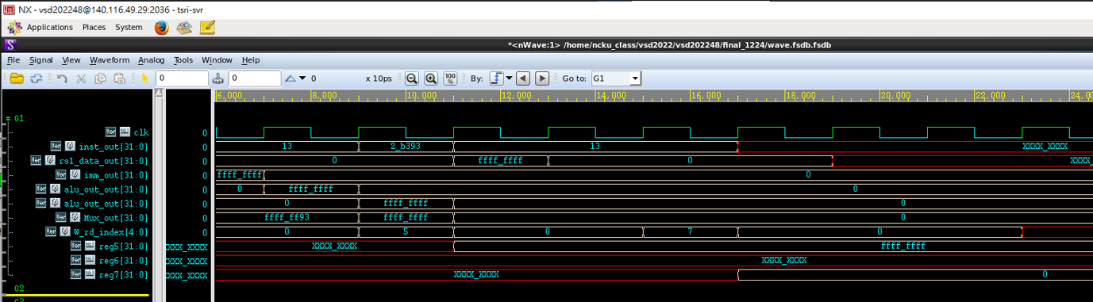
2-2. slti rd, rs1, imm

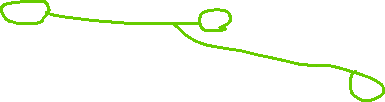
一張含有 文字, 螢幕擷取畫面, 電腦 的圖片

自動產生的描述

上圖黃色的是slti的instr，rs1是0，imm是1，因為rs2>rs1，所以最後可看到rd reg7正確存入1。

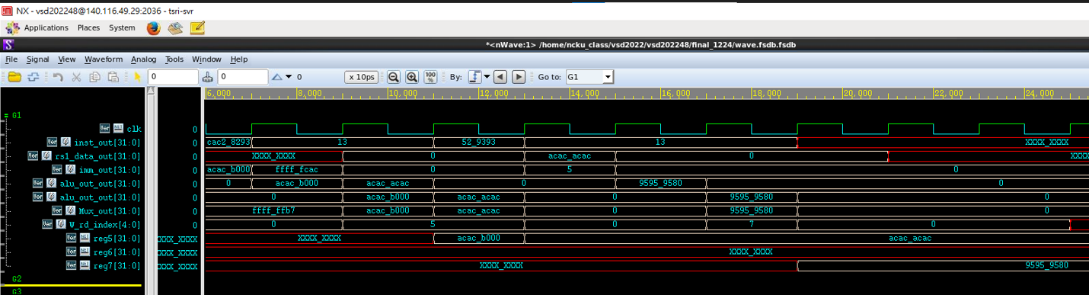
2-3. sltiu rd, rs1, imm

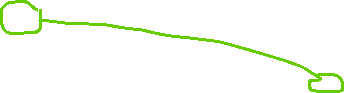




上圖黃色的是slti的instr，rs1是0hffff\_ffff，imm是0，因為unsigned rs1>rs2，所以最後可看到rd reg7正確存入0。

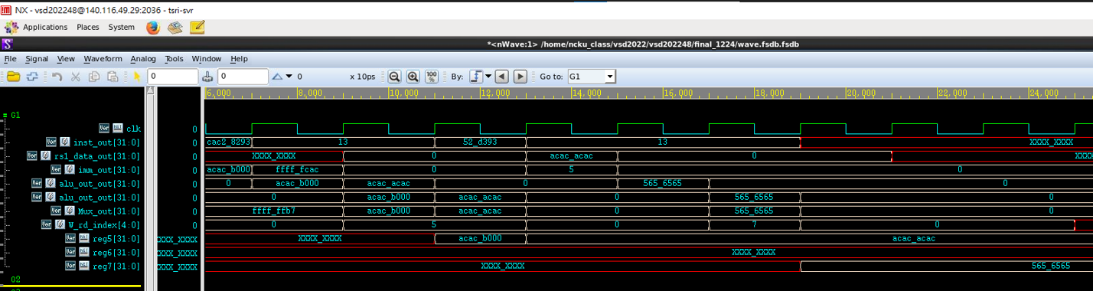
2-4. slli rd, rs1, imm





上圖黃色的是slli的instr，rs1是0hacac\_acac，imm是5，將rs1做邏輯左移5個bits，所以最後可看到rd reg7正確存入0h9595\_9580。

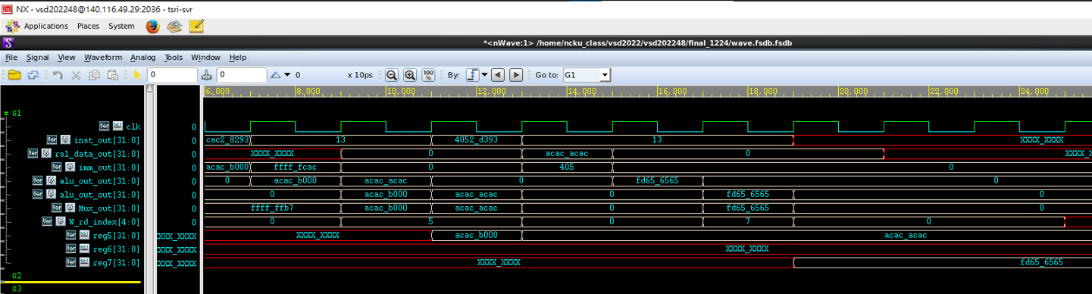
2-5. srli rd, rs1, imm





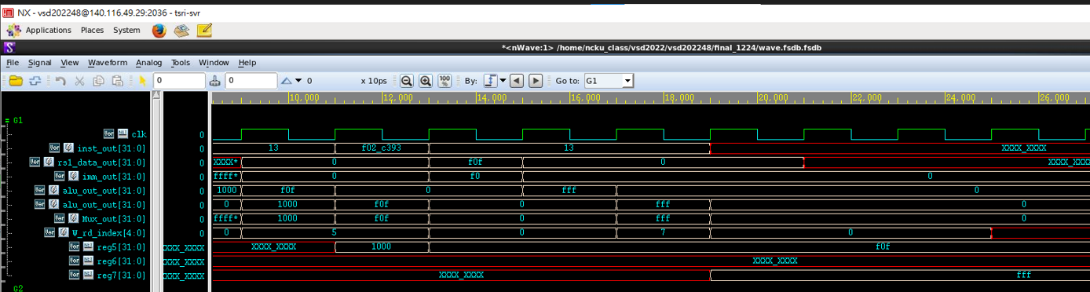
上圖黃色的是srli的instr，rs1是0hacac\_acac，imm是5，將rs1做邏輯右移5個bits，所以最後可看到rd reg7正確存入0h0565\_6565。

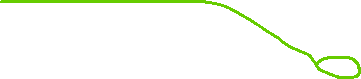
2-6. srai rd, rs1, imm



上圖黃色的是srai的instr，rs1是0hacac\_acac，imm是0h405，這裡原本rs2只要5，但不小心打成405，但因為srai只會取imm的前五個bits，所以還是只會算數右移5個bits，所以最後可看到rd reg7正確存入0hfd65\_6565。

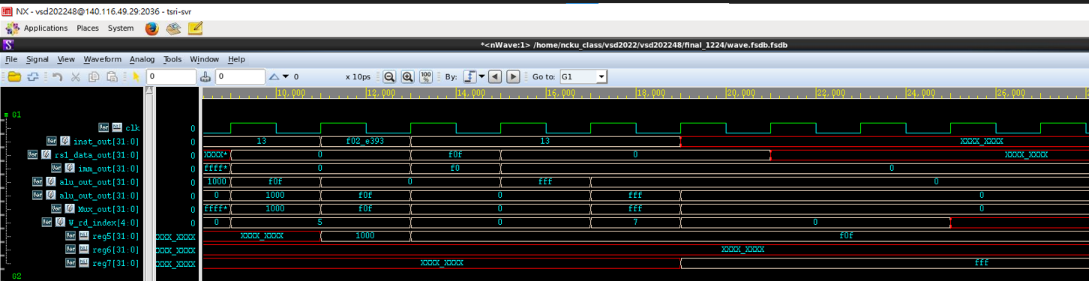
2-7. xori rd, rs1, imm

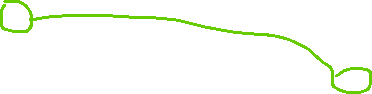




上圖黃色的是xori的instr，rs1是0hf0f，imm是0h0f0，將rs1和rs2做xor，所以最後可看到rd reg7正確存入0hfff。

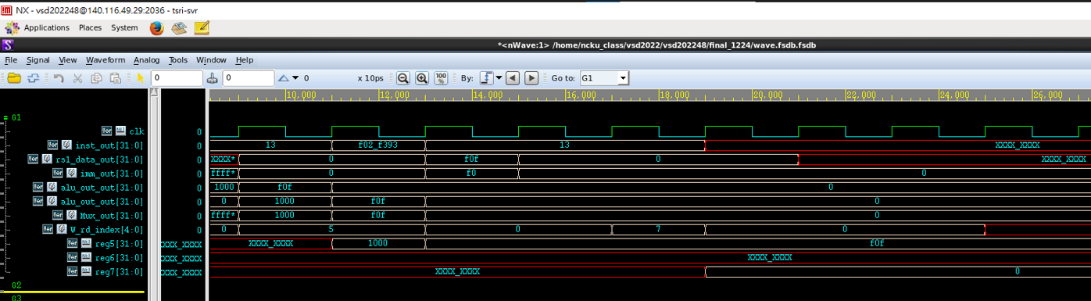
2-8. ori rd, rs1, imm

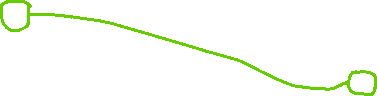




上圖黃色的是ori的instr，rs1是0hf0f，imm是0h0f0，將rs1和rs2做or，所以最後可看到rd reg7正確存入0hfff。

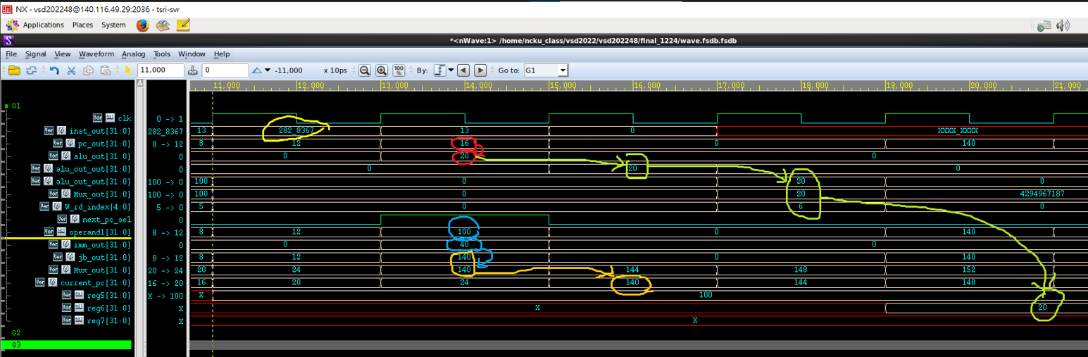
2-9. andi rd, rs1, imm





上圖黃色的是andi的instr，rs1是0hf0f，imm是0h0f0，將rs1和rs2做and，所以最後可看到rd reg7正確存入0hfff。

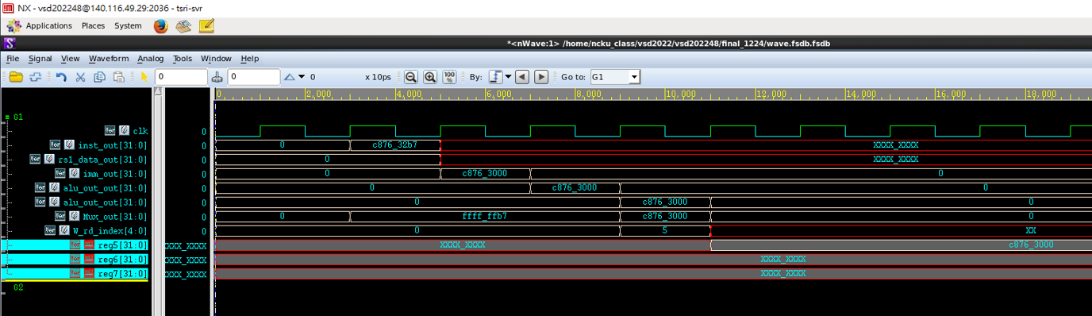
2-10. jalr rd, rs1, imm



上圖黃色的是jalr的instr，jalr會將當前pc加上4並存到rd，所以可以看到上圖綠色的pc加上4也就是20，存到rd\_index reg5，而計算地址則是將rs1加上imm，也就是上圖藍色的100+40=140，所以可以看到下一個cycle current\_pc成功跳到140。

3. U type

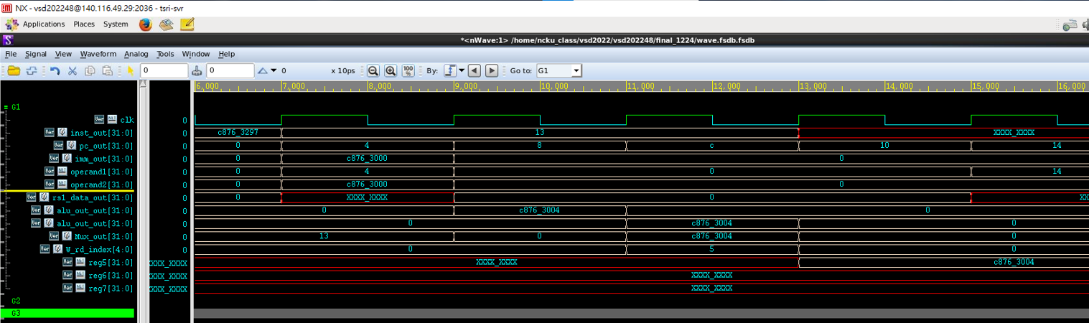
3-1. lui rd,imm





上圖黃色的是lui的instr，imm是0h c8763，到ID stage會把imm串接12個0，也就是0hc876\_3000，經過EX、MEM stage，到WB stage寫入rd\_index的位置，所以最後可看到rd reg5正確存入0hc876\_3000。

3-2. auipc rd,imm

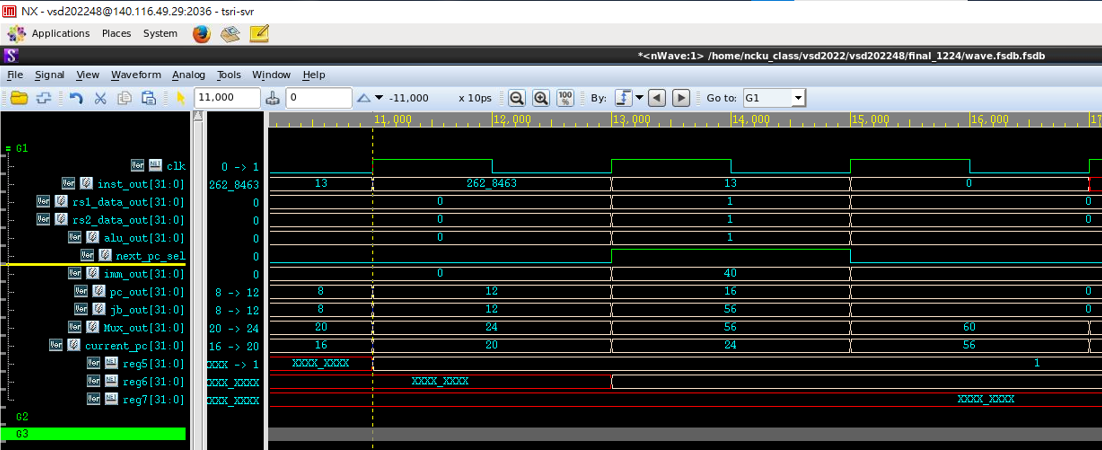




上圖黃色的是auipc的instr，imm是0hc8763，到ID stage會把imm串接12個0，也就是0hc876\_3000，和lui不同的是，auipc會將串接過的imm再加上pc，所以會是0hc876\_3000加上pc的值4，所以會是0hc876\_3004，經過EX、MEM stage，到WB stage寫入rd\_index的位置，所以最後可看到rd reg5正確存入0hc876\_3004。

4. B type

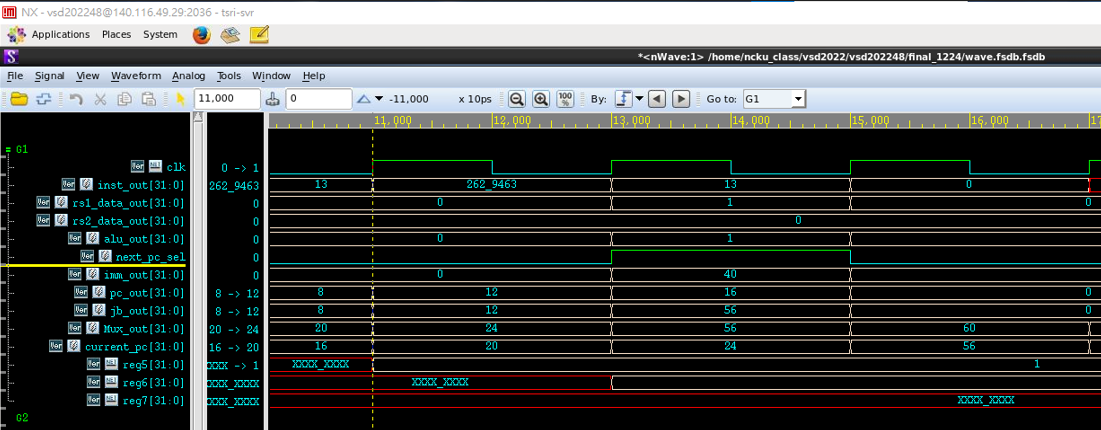
4-1. beq rs1, rs2, imm

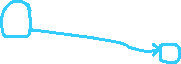




上圖黃色的是beq的instr，rs1和rs2都是1，比對後rs1=rs2，所以可以看到next\_pc\_sel是1，代表要跳，而計算跳轉地址是將imm和pc相加，也就是40+16=56，所以可以看到下一個cycle current\_pc成功跳到56。

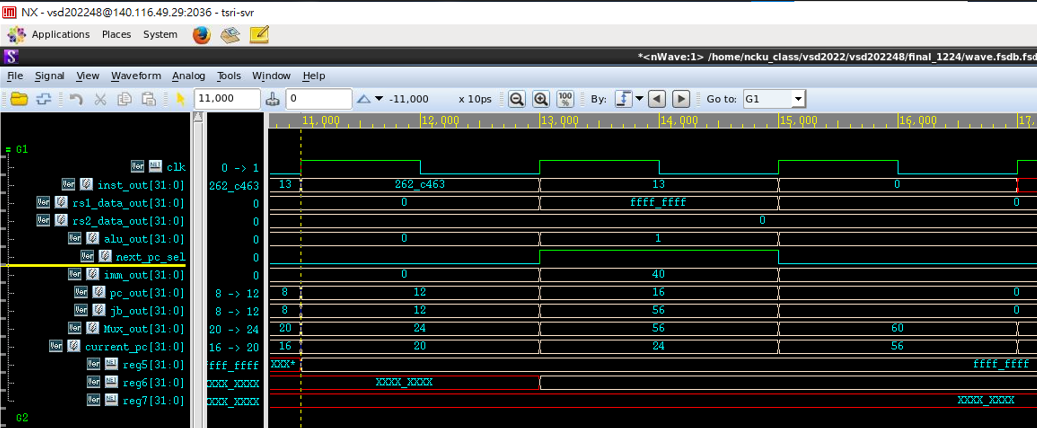
4-2. bne rs1, rs2, imm

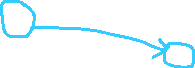




上圖黃色的是bne的instr，rs1是1、rs2是0，比對後rs1!=rs2，所以可以看到next\_pc\_sel是1，代表要跳，而計算跳轉地址是將imm和pc相加，也就是40+16=56，所以可以看到下一個cycle current\_pc成功跳到56。

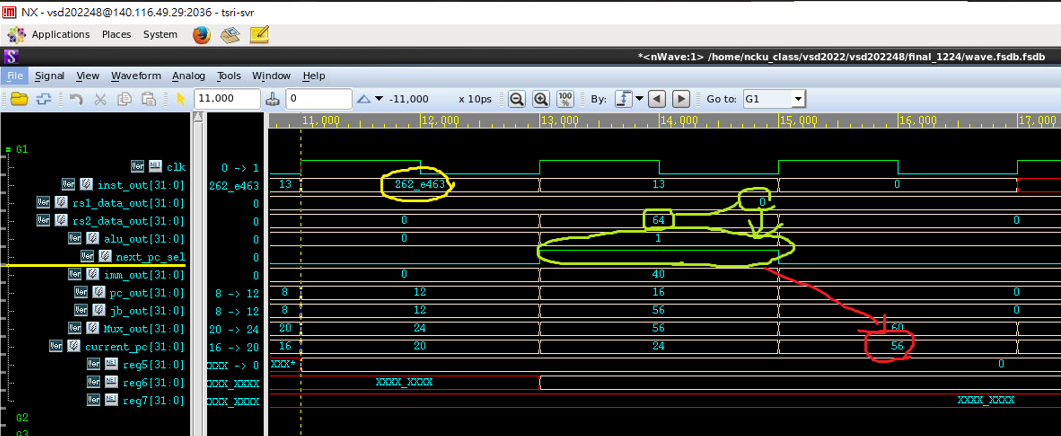
4-3. blt rs1, rs2, imm





上圖黃色的是blt的instr，rs1是0hffff\_ffff(-1)、rs2是0，比對後signed rs1<rs2，所以可以看到next\_pc\_sel是1，代表要跳，而計算跳轉地址是將imm和pc相加，也就是40+16=56，所以可以看到下一個cycle current\_pc成功跳到56。

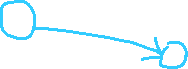
4-4. bltu rs1, rs2, imm



上圖黃色的是bltu的instr，rs1是0、rs2是64，比對後unsigned rs1<rs2，所以可以看到next\_pc\_sel是1，代表要跳，而計算跳轉地址是將imm和pc相加，也就是40+16=56，所以可以看到下一個cycle current\_pc成功跳到56。

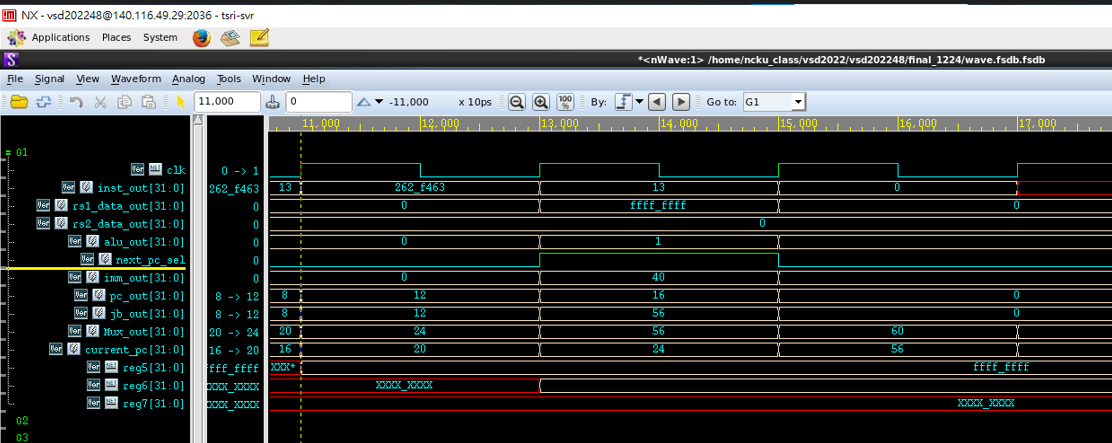
4-5. bge rs1, rs2, imm





上圖黃色的是bge的instr，rs1是64、rs2是0，比對後signed rs1>rs2，所以可以看到next\_pc\_sel是1，代表要跳，而計算跳轉地址是將imm和pc相加，也就是40+16=56，所以可以看到下一個cycle current\_pc成功跳到56。

4-6. bgeu rs1, rs2, imm

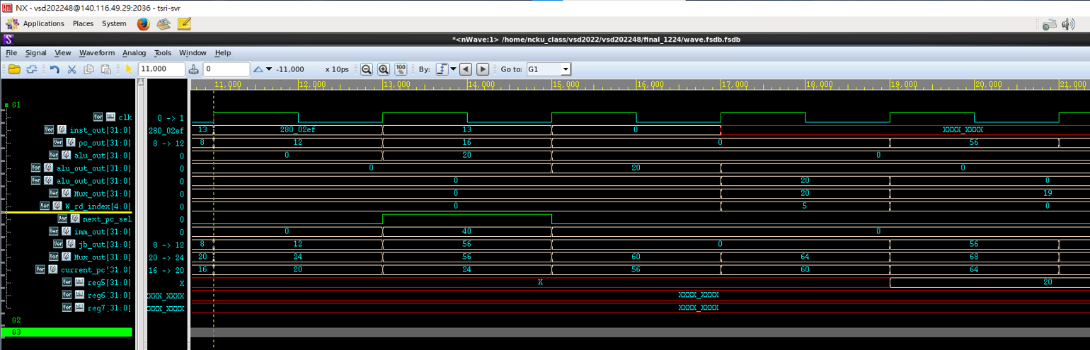


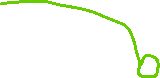


上圖黃色的是bgeu的instr，rs1是0hffff\_ffff、rs2是0，比對後unsigned rs1>rs2，所以可以看到next\_pc\_sel是1，代表要跳，而計算跳轉地址是將imm和pc相加，也就是40+16=56，所以可以看到下一個cycle current\_pc成功跳到56。

5. J type

5-1. jal rd, imm





上圖黃色的是jal的instr，jal會將當前pc加上4並存到rd，所以可以看到上圖綠色的pc加上4也就是20，存到rd\_index reg5，而計算地址則是將pc加上imm，也就是上圖藍色的16+40=56，所以可以看到下一個cycle current\_pc成功跳到56。

6. S type(store) & I type的load

參考Cache

* + 1. 程式正確性
       1. 排序(需自行撰寫組合語言)

組合語言：

.data

test1: .word 16,2,4,16,4,10,12,2,14,8,4,14,6,4,2,10,12,6,10,2,14,14,6,8,16,8,16,6,12,10,8,123

test2: .word 470,405,225,197,126,122,56,33,-81,-275,-379,-409,-416,-496,-500

test3: .word 412,-474,443,171,-23,247,221,7,40,221,-90,61,-9,49,-80,-80,221,-379,-161,-397,-173,276,-197,221,-12,-145,101

TEST1\_SIZE: .word 32

TEST2\_SIZE: .word 15

TEST3\_SIZE: .word 27

.text

.globl main

main:

addi sp, sp, -4

sw s0, 0(sp)

la s0, \_answer

##----------------------------------------------------------------------lab8

##將test1、2、3的data移到0x9000##

la t0,test1

la t1,TEST1\_SIZE

lw t1,0(t1)

mv t2,t1

la t1,TEST2\_SIZE

lw t1,0(t1)

add t2,t2,t1

la t1,TEST3\_SIZE

lw t1,0(t1)

add t2,t2,t1 #t2:test1\_size+test2\_size+test3\_size

li t1,0 #t1=i=0

li t4,0x00009000

loop:

bge t1,t2,end\_loop

lw t3,0(t0)

sw t3,0(t4)

addi t1,t1,1

addi t0,t0,4

addi t4,t4,4

beq x0,x0,loop

end\_loop:

##求test1、2、3的初始位置##

li t0,0x9000 ##t0:test1的初始位置

la t1,TEST1\_SIZE

lw t1,0(t1) ##t1:test1\_size

slli t1,t1,2 ##t1=t1\*4

add t2,t0,t1 ##t2:test2的初始位置

la t1,TEST2\_SIZE

lw t1,0(t1) ##t1:test2\_size

slli t1,t1,2 ##t1=t1\*4

add t3,t2,t1 ##t3:test3的初始位置

##----------------------------------------------------------------------lab8

####callee saved

addi sp,sp,-24

sw s0,0(sp)

sw s1,4(sp)

sw s2,8(sp)

sw s3,12(sp)

sw s4,16(sp)

sw s5,20(sp)

####

mv s0,t0 #s0放test1的初始位置地址

la s1,TEST1\_SIZE #s1放TEST1\_SIZE的地址

lw s1,0(s1) #把s1地址存的東西(陣列長度)放在s1

mv s2,t2 #s2放test1的初始位置地址

la s3,TEST2\_SIZE #s3放TEST1\_SIZE的地址

lw s3,0(s3) #把s3地址存的東西(陣列長度)放在s1

mv s4,t3 #s4放test1的初始位置地址

la s5,TEST3\_SIZE #s5放TEST1\_SIZE的地址

lw s5,0(s5) #把s5地址存的東西(陣列長度)放在s1

####caller saved

addi sp,sp,-4 #sp向下4byte

sw ra,0(sp) #Caller saved:這時ra=-1

############################# test1 #############################

####給參數值

mv a0,s0 # a0 =s0 = test1的初始位置

mv a1,x0 # a1 =x0 = start

addi a2,s1,-1 # a2 = TEST1\_SIZE-1 = end

####呼叫function : mergesort

jal ra, mergesort #ra = 下一行

############################# test2 #############################

####給參數值

mv a0,s2 # a0 =s2 = test2的初始位置

mv a1,x0 # a1 =x0 = start

addi a2,s3,-1 # a2 = TEST1\_SIZE-1 = end

####呼叫function : mergesort

jal ra, mergesort #ra = 下一行

############################# test3 #############################

####給參數值

mv a0,s4 # a0 =s4 = test3的初始位置

mv a1,x0 # a1 =x0 = start

addi a2,s5,-1 # a2 = TEST1\_SIZE-1 = end

####呼叫function : mergesort

jal ra, mergesort #ra = 下一行

#################################################################

####Caller saved:restore

lw ra, 0(sp) #Caller saved:拿回一開始呼叫前存的ra=-1

addi sp, sp, 4 #釋放stack空間

####callee saved:restore

lw s0,0(sp)

lw s1,4(sp)

lw s2,8(sp)

lw s3,12(sp)

lw s4,16(sp)

lw s5,20(sp)

addi sp,sp,24

ret #jalr x0, ra, 0

############ mergesort ##############

# t0 : test的初始位置地址

# t1 : start

# t2 : end

# t3 : mid = (start+end)/2

mergesort:

####轉移參數

mv t0, a0 # t0 : test的初始位置地址

mv t1, a1 # t1 : start

mv t2, a2 # t2 : end

#### if(start<end) => if(start>=end) , 跳去mergesort\_end

bge t1, t2, mergesort\_end

####求mid，放在t3

add t3,t2,t1 #t3 = t1+t2 = start+end

srli t3,t3,1 #t3 = t3/2 = (start+end)/2 = mid

#/////////////////////////左邊 : mergesort(arr, start, mid);

####Caller saved : ra ,test的初始位置地址(t0), start(t1), end(t2), mid(t3)

addi sp,sp,-20

sw ra,0(sp)

sw t0,4(sp) #test的初始位置地址

sw t1,8(sp) #start

sw t2,12(sp) #end

sw t3,16(sp) #mid

####給參數值

mv a0, t0 # a0 = t0 = test的初始位置地址

mv a1, t1 # a1 = t1 = start

mv a2, t3 # a2 = t3 = mid

####Call left mergesort

jal ra, mergesort

####Caller saved : restore

lw ra,0(sp)

lw t0,4(sp) #test的初始位置地址

lw t1,8(sp) #start

lw t2,12(sp) #end

lw t3,16(sp) #mid

addi sp,sp,20

#/////////////////////////右邊 : mergesort(arr, mid+1, end);

####Caller saved : ra ,test的初始位置地址(t0), start(t1), end(t2), mid(t3)

addi sp,sp,-20

sw ra,0(sp)

sw t0,4(sp) #test的初始位置地址

sw t1,8(sp) #start

sw t2,12(sp) #end

sw t3,16(sp) #mid

####給參數值

mv a0,t0 #a0 = t0 = test的初始位置地址

addi t3,t3,1 #t3 = mid+1

mv a1,t3 #a1 = t3 = mid+1

mv a2,t2 #a2 = t2 = end

####Call right mergesort

jal ra, mergesort

####Caller saved : restore

lw ra,0(sp)

lw t0,4(sp) #test的初始位置地址

lw t1,8(sp) #start

lw t2,12(sp) #end

lw t3,16(sp) #mid

addi sp,sp,20

#///////////////////////// 最後 : merge(arr, start, mid, end);

addi sp,sp,-20

sw ra,0(sp)

sw t0,4(sp) #test的初始位置地址

sw t1,8(sp) #start

sw t2,12(sp) #end

sw t3,16(sp) #mid

####給參數值

mv a0,t0 #a0 = t0 = test的初始位置地址

mv a1,t1 #a1 = t1 = start

mv a2,t3 #a2 = t3 = mid

mv a3,t2 #a3 = t2 = end

####Call merge

jal ra, merge

####Caller saved : restore

lw ra,0(sp)

lw t0,4(sp) #test的初始位置地址

lw t1,8(sp) #start

lw t2,12(sp) #end

lw t3,16(sp) #mid

addi sp,sp,20

mergesort\_end:

jalr x0, ra, 0

############ merge ##############

# s0 : temp\_size

# s1 : temp的初始位置地址

# s2 : left\_index

# s3 : right\_index

# s4 : left\_max

# s5 : right\_max

# s6 : arr\_index

# s7 : temp\_size\*4

merge:

####callee saved

addi sp,sp,-32

sw s0,0(sp)

sw s1,4(sp)

sw s2,8(sp)

sw s3,12(sp)

sw s4,16(sp)

sw s5,20(sp)

sw s6,24(sp)

sw s7,28(sp)

####轉移參數

mv t0, a0 # t0 : test的初始位置地址

mv t1, a1 # t1 : start

mv t2, a2 # t2 : mid

mv t3, a3 # t3 : end

####int temp\_size = end - start + 1;

add t4,t3,t1 #t4 = t3-t1 = end-start

addi s0,t4,1 #s0 = end-start+1 = temp\_size

####創造長度為temp\_size的array(temp)放在stack

slli t4,s0,2 #t4 = temp\_size\*4

sub sp,sp,t4 #sp向下移temp\_size\*4

mv s7,t4 #把temp\_size\*4存在s7

mv s1,sp #s1存temp的初始位置地址

################# for loop #################

li t4,0 #int i = 0;

for\_loop:

bge t4,s0,end\_for\_loop #if(i>=temp\_size) go to end\_for\_loop

## boby

#取得arr[i+start]的值

add t5,t4,t1 #t5=i+start

slli t5,t5,2 #t5=t5\*4

add t5,t5,t0 #t5=arr[i+start]的地址

lw t5,0(t5) #t5=arr[i+start]的值

#將arr[i+start]的值放在temp[i]

slli t6,t4,2 #t6 = i\*4

add t6,t6,s1 #t6 = t6+s1 = i\*4+temp的初始位置地址 = temp[i]的地址

sw t5,0(t6) #將arr[i+start]的值放在temp[i]

## i++

addi t4,t4,1

beq x0,x0,for\_loop

end\_for\_loop:

################# for loop #################

####初始化

li s2,0 #int left\_index = 0;

sub t4,t2,t1 #t4 = mid-start

addi s3,t4,1 #int right\_index = mid-start+1;

sub s4,t2,t1 #int left\_max = mid-start;

sub s5,t3,t1 #int right\_max = end-start;

mv s6,t1 #int arr\_index = start;

######################## first while loop #############################

first\_while:

bgt s2,s4,first\_while\_end #if left\_index > left\_max, go to first\_while\_end

bgt s3,s5,first\_while\_end #if right\_index > right\_max, go to first\_while\_end

## boby ##

slli t4,s2,2 #t4=left\_index\*4

add t4,t4,s1 #t4=left\_index\*4+temp的初始位置地址

lw t4,0(t4) #t4=temp[left\_index]

slli t5,s3,2 #t5=right\_index\*4

add t5,t5,s1 #t5=right\_index\*4+temp的初始位置地址

lw t5,0(t5) #t5=temp[right\_index]

blt t5,t4,else\_1 #if(temp[left\_index] > temp[right\_index]) go to else\_1

if\_1:

slli t6,s6,2 #t6=arr\_index\*4

add t6,t6,t0 #t6=arr\_index\*4+arr的初始位置地址

sw t4,0(t6) #arr[arr\_index] = temp[left\_index];

addi s6,s6,1 #arr\_index++

addi s2,s2,1 #left\_index++

beq x0,x0,end\_1

else\_1:

slli t6,s6,2 #t6=arr\_index\*4

add t6,t6,t0 #t6=arr\_index\*4+arr的初始位置地址

sw t5,0(t6) #arr[arr\_index] = temp[right\_index];

addi s6,s6,1 #arr\_index++

addi s3,s3,1 #right\_index++

end\_1:

beq x0,x0,first\_while

## boby ##

first\_while\_end: #first\_while條件式不符合，跳到這裡

######################## second while loop #############################

second\_while:

blt s4,s2,second\_while\_end #if left\_index > left\_max, go to second\_while\_end

## boby ##

slli t4,s2,2 #t4=left\_index\*4

add t4,t4,s1 #t4=left\_index\*4+temp的初始位置地址

lw t4,0(t4) #t4=temp[left\_index]

slli t6,s6,2 #t6=arr\_index\*4

add t6,t6,t0 #t6=arr\_index\*4+arr的初始位置地址

sw t4,0(t6) #arr[arr\_index] = temp[left\_index];

addi s6,s6,1 #arr\_index++

addi s2,s2,1 #left\_index++

## boby ##

beq x0,x0,second\_while

second\_while\_end: #second\_while條件式不符合，跳到這裡

######################## third while loop #############################

third\_while:

blt s5,s3,third\_while\_end #if right\_index > right\_max, go to third\_while\_end

## boby ##

slli t5,s3,2 #t5=right\_index\*4

add t5,t5,s1 #t5=right\_index\*4+temp的初始位置地址

lw t5,0(t5) #t5=temp[right\_index]

slli t6,s6,2 #t6=arr\_index\*4

add t6,t6,t0 #t6=arr\_index\*4+arr的初始位置地址

sw t5,0(t6) #arr[arr\_index] = temp[right\_index];

addi s6,s6,1 #arr\_index++

addi s3,s3,1 #right\_index++

## boby ##

beq x0,x0,third\_while

third\_while\_end: #third\_while條件式不符合，跳到這裡

#########################################################################

add sp,sp,s7 #將temp\_size\*4加回來

####callee saved:restore

lw s0,0(sp)

lw s1,4(sp)

lw s2,8(sp)

lw s3,12(sp)

lw s4,16(sp)

lw s5,20(sp)

lw s6,24(sp)

lw s7,28(sp)

addi sp,sp,32

ret

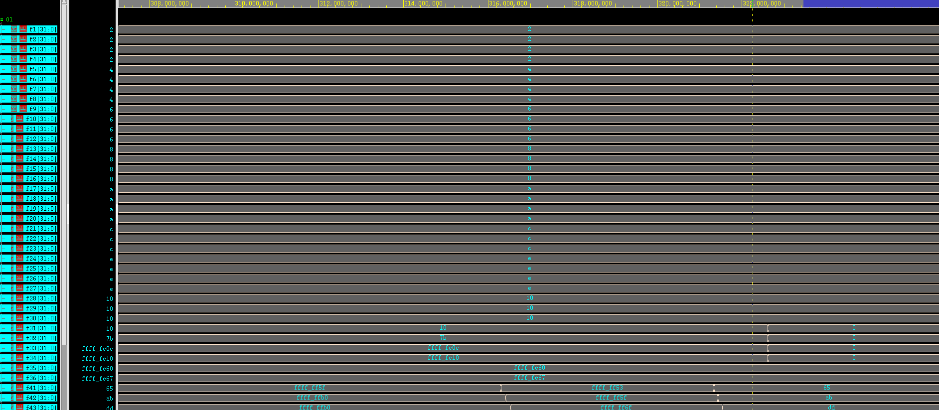
main\_exit:

lw s0, 0(sp)

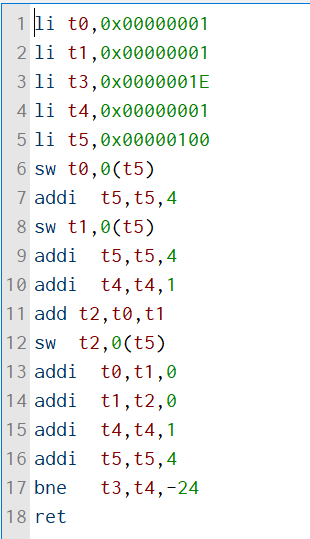
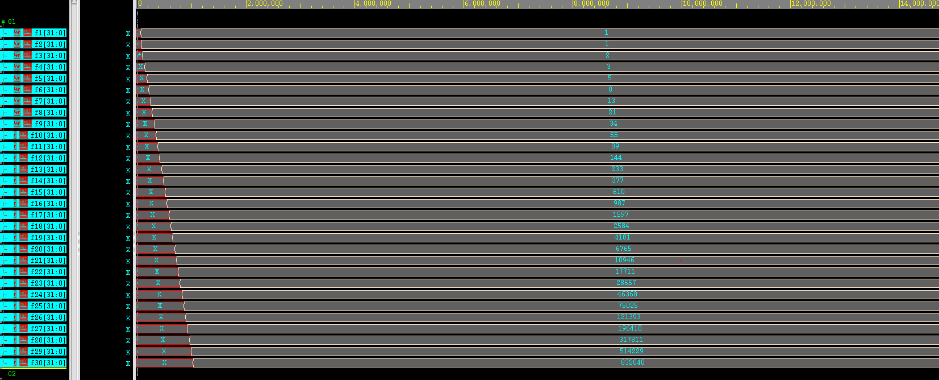
addi sp, sp, 4

ret

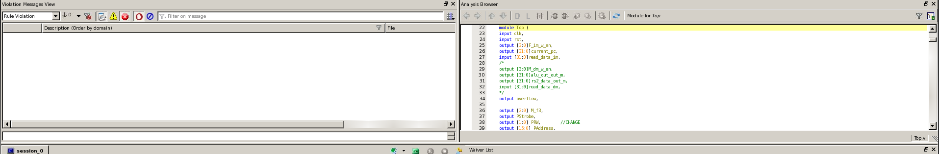
nWave波形：



* + - 1. 費氏數列(需自行撰寫組合語言)



1. **SuperLint與ICC檢查結果**
   1. SuperLint檢查結果(需附上SuperLint截圖)



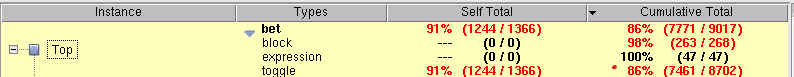
Error = 0

* 1. ICC檢查結果(需附上ICC截圖)

Total Block coverage = 100%(263/268, 5 non-included statement explained below)

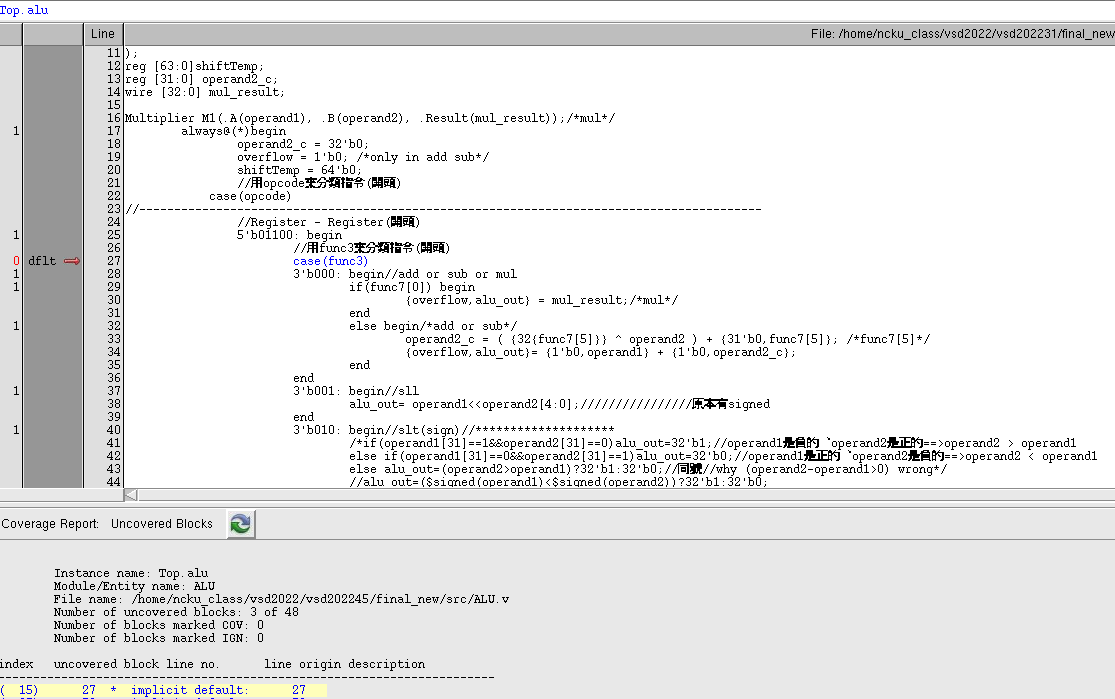
Total expression coverage = 100%

Total Toggle Coverage = 86%

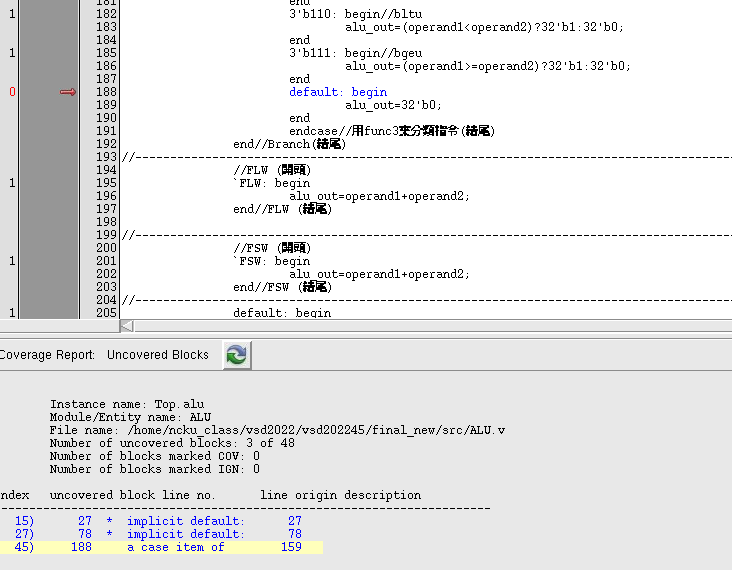


Alu 內三個不合理判定的statement：





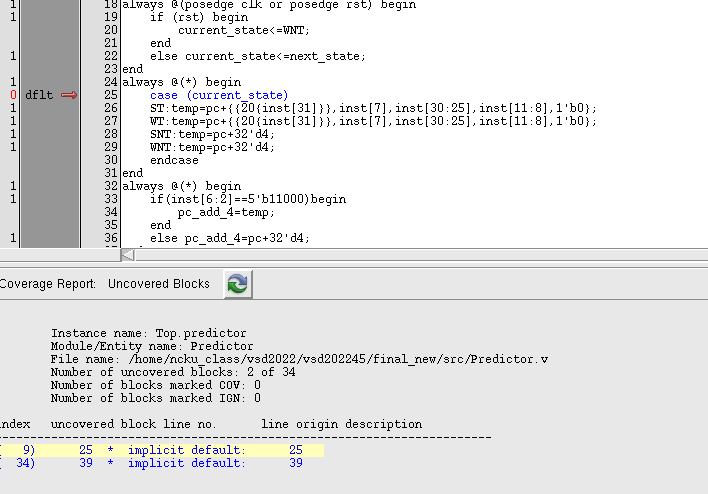


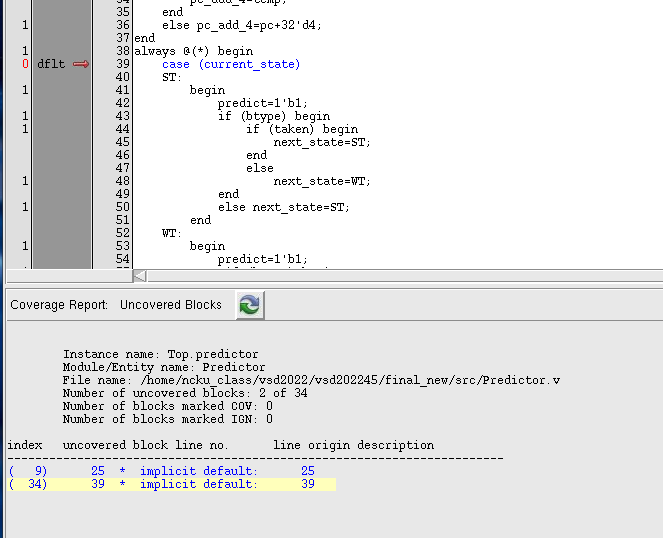


Full case因此不會跑到

predictor 內兩個不合理判定的statement：







1. **合成結果**
   1. 速度(需附上截圖，Setup time和Hold time slack都>0)

Clock period: 34ns

Setup time slack = 0.01

一張含有 桌 的圖片

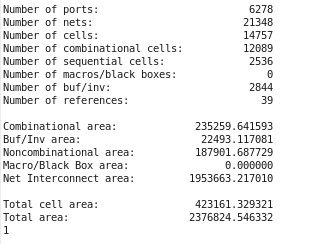
自動產生的描述

Hold time slack = 0.49

一張含有 文字, 螢幕擷取畫面, 收據 的圖片

自動產生的描述

* 1. 面積(需附上截圖)



Total cell area = 423161.329321 um2

* 1. 功耗(需附上截圖)

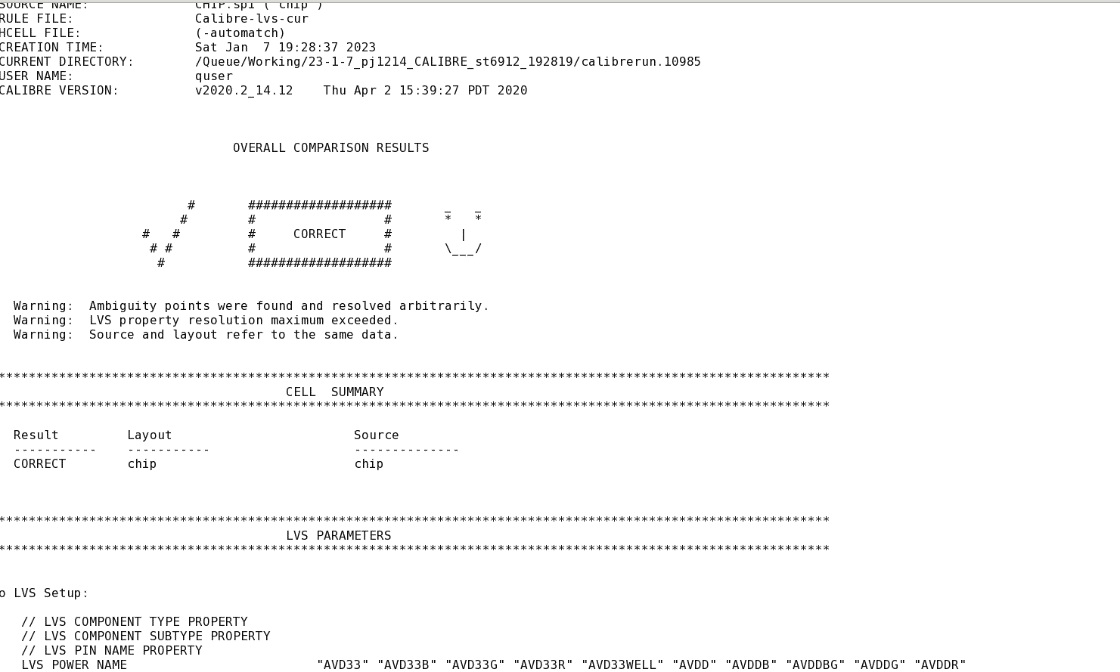
一張含有 文字, 收據, 螢幕擷取畫面 的圖片

自動產生的描述

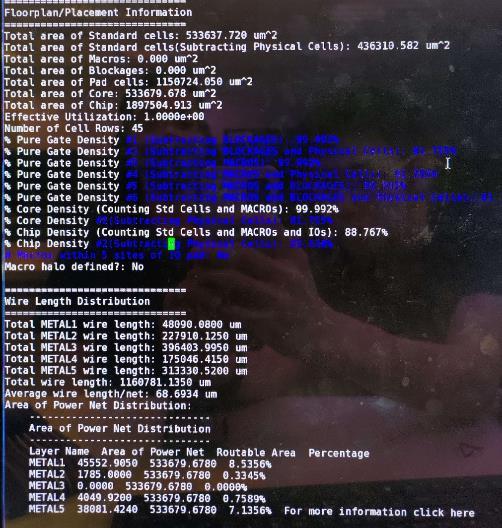
Total Dynamic Power = 5.3572 mW

1. **Layout結果**

LVS result:



DRC report (NO error)



Verify geometry

****

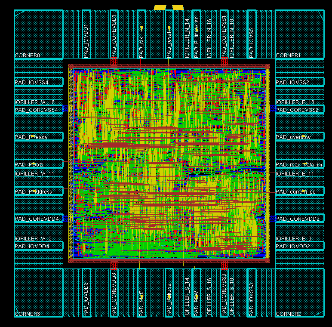
Verify connectivity

****

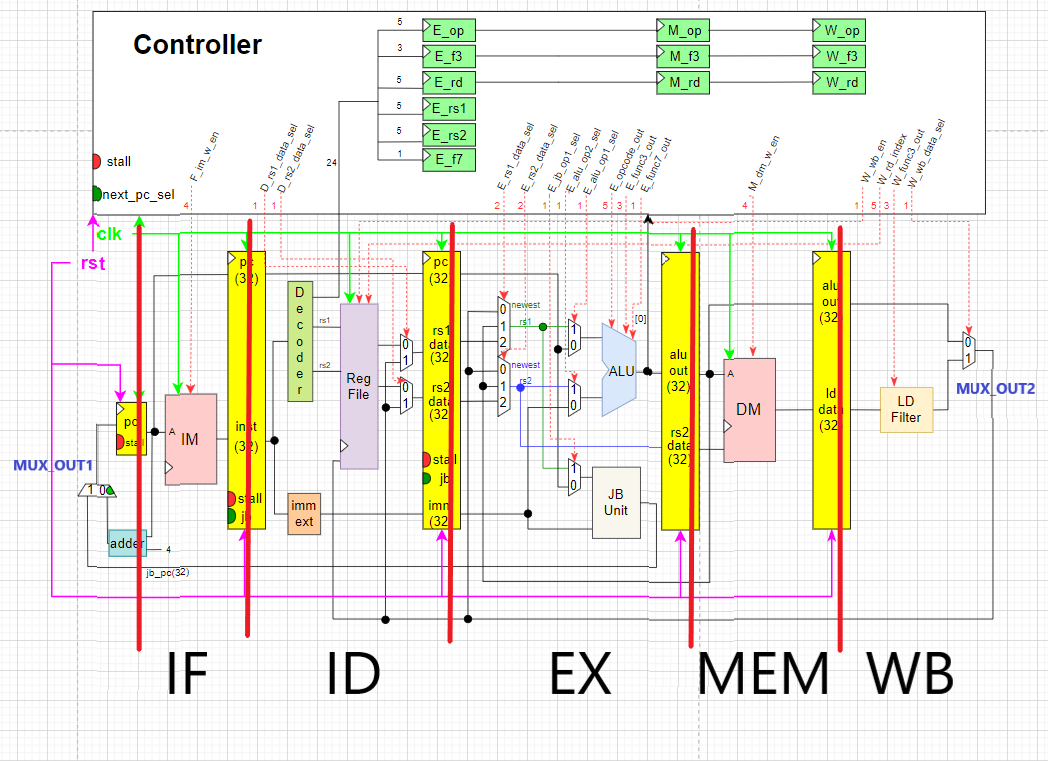
Verify ANTENNA

****

Layout 結果



1. **管線化**
   1. 設計說明



* 1. 階段的劃分

IF：Instruction fetch，會透過pc從Instruction memory 取出現在要執行的instruction，並且會透過一個多工器選擇下一個要執行的instruction的pc是上一個pc加4還是由JB Unit決定。

ID：Instruction decode，會將instruction 經過decoder解碼出上方指令集格式中的各欄位，並去讀取Register File。

EX：Execute，會根據opcode、func3、func7決定要做什麼指令，並根據指令來決定ALU和JB Unit輸入和處理方式。

MEM：Memory access，若M\_dm\_w\_en為1時，會將ALU計算完的memory address和要存入的write data輸入Data memory。

WB：Write back，將alu\_out或是Data memory的load data寫回Register File。

* 1. 管線化的危障(需說明解決方法)

管線化會遇到以下危障: 資料危障、控制危障與結構危障。成因與解決方法可以以下方表格說明

|  |  |  |
| --- | --- | --- |
| 危障類型 | 成因 | 解決方法 |
| 結構危障 | Fetch階段取指令與write階段寫入資料對於同個記憶體會有衝突。 | 將記憶體分為im與dm，分別存放指令與資料 |
| 控制危障 | 指令在execute階段決定要跳轉，然而fetch與decode已經開始執行錯誤的指令 | Controller會接收跳轉訊號並且flush掉原本要執行的指令。 |
| 資料危障 | 如果運算結果需要寫回register更新，那麼後面的指令若是需要馬上用到就會來不及取到最新的資料 | 將write back與mem階段的資料提前回授到instruction decode與execute階段 |

1. **特殊設計(例如cache、支援浮點數運算等額外設計都可提出)**
   1. Branch predict
2. **predictor 種類**

bimodal predictor



1. **IO and state explanation**

**Input:**

clk clock

rst reset

pc program counter

inst instruction

btype 是否現在ALU是否是在做B type的運算

taken 用來檢視branch指令是truth or false

**output:**

pc\_add\_4 下一個pc需要的數值

predict 用來告訴controller現在是何種stage，使controller

知道之後是否要refresh data

**predictor state:**

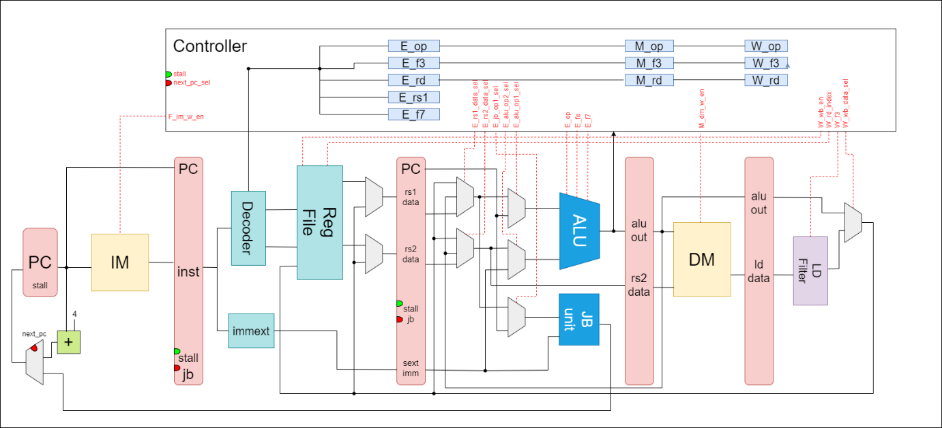
ST = 2'b00 strong taken

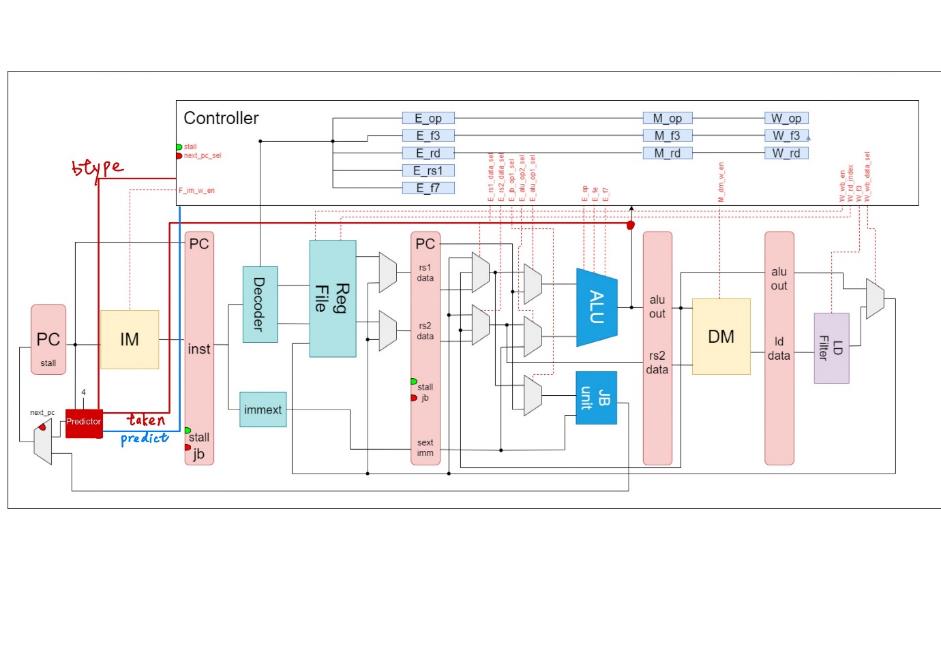
WT = 2'b01 weak taken

SNT = 2'b10 strong not taken

WNT = 2'b11 weak not taken

Default=WNT weak not taken

1. **硬體架構**

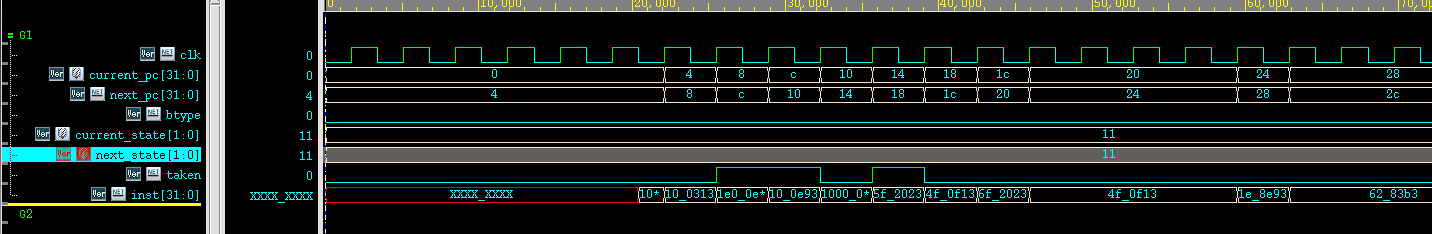


After

Before

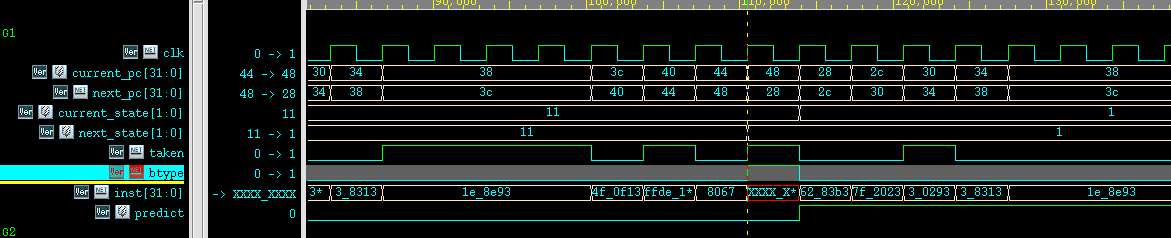
1. **驗證結果(費氏數列)**

可以看到沒有讀取到b type指令，不改變狀態

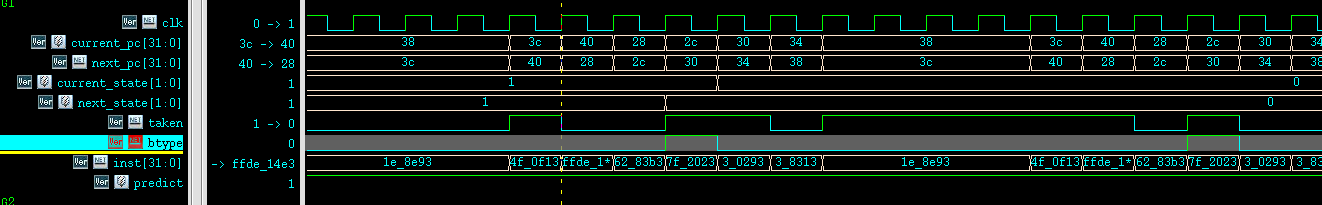


可以看到因為現在是WNT stage，所以pc會繼續pc+4抓下兩個指令才branch，且因為8067已經為ret，所以在fetch會有xxxxx出現，現在btype and taken都是1 所以下一個stage是WT

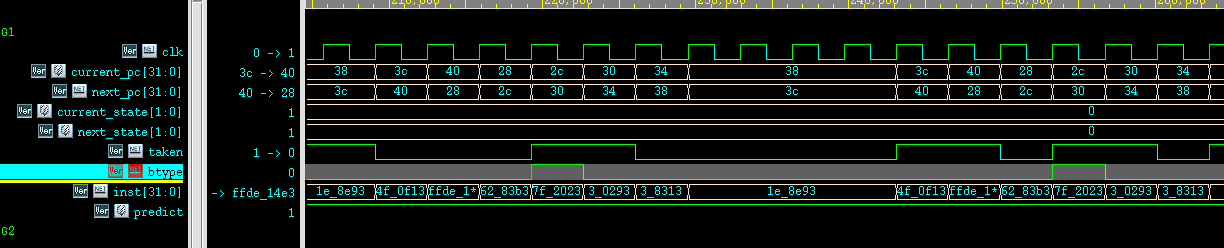
Branch 指令



當又一次遇到branch指令，可以看到因為現在的stage是WT所以不會再執行pc+4，而是會直接取到branch 後指令(pc=40後下一個直接是pc=28)，且因為branch taken 下一個stage會是ST



當再次遇到branch因為已經ST所以會直接取到branch 後指令，且因為branch taken 下一個stage會是ST，保持直到計算完全部的費氏數列



結論:可以看到branch predictor確實有成功predict

* 1. Cache

1. Component spec and some definition of particular signal:

因為Memory 設計為8bits x 216 個 (Address: 16bits)

🡪因此Cache對應Memory的設計取210 個

1. DataRAM: 8bits x 210
2. TagRAM: 6bits x 210
3. ValidRAM: 1bit x 210
4. Tag: PAddress[15:10] Address: PAddress[9:0]
5. LoadSelect [1:0] : 透過從CPU輸入的func3判斷目前是要Store/Load Byte, Half-Word, 還是Word。

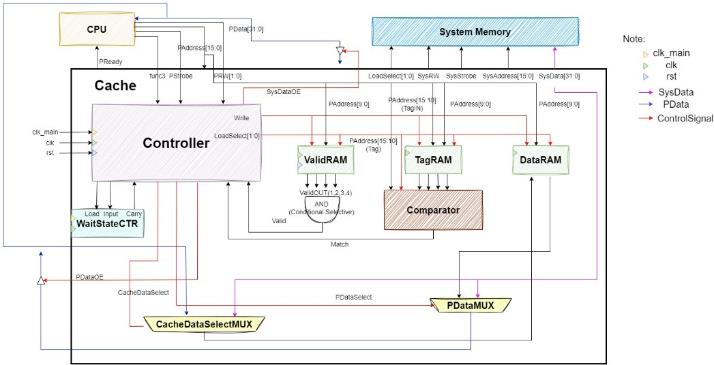
🡪 `define BYTE 2’d0

🡪 `define HALFWORD 2’d1

🡪 `define WORD 2’d2

1. PStrobe, SysStrobe: = 1 when trying to access Cache and System Memory respectively.
2. SysData and PData are a pair of inout port controlled by SysDataOE and PDataOE.
3. clk period = clk\_main period / 2

2. Structure Diagram of Cache:



3. Design Concept of Particular Module

**TagRAM:**

做為儲存Cache中Tag的儲存單元，在posedge的時候透過判斷指令的type是Byte, Half-Word, Word (LoadSelect) 來送出對應的Tag1, Tag2, Tag3, Tag4。並在negedge的時候同理判斷LoadSelect來寫入從CPU傳送過來的Tag。🡪若為Byte則僅寫入Address的位置，Half-Word寫入Address, Address+1的位置，以此類推。

**ValidRAM:**

做為判斷該資料是否存在Cache中的Valid訊號的儲存單元，與TagRAM的操作方式相似都是在posedge讀，negedge寫入並都是透過判斷LoadSelect決定寫入與輸出的數量。但與TagRAM不同的是ValidRAM會吃rst訊號因為當一開始開啟CPU時若沒有吃Valid訊號則有可能出現資料比對到存在於Cache中的亂碼進而出現非預期的Cache Hit的狀況因此需要在一開始將Valid RAM清空使當沒有寫入過該位置但卻比對到的時候要輸出Not Match。

**DataRAM:**

做為儲存Cache資料的RAM，操作方式與TagRAM相同，唯一差異的是輸入的資料不再是Tag而是Data本身。這邊傳入的Data並不特指一個位置傳送過來的而是要在下面介紹的CacheDataSelectMUX中做選擇。

**AND Gate (Conditional Selective):**

這邊放這個AND Gate的原因是因為Valid RAM傳送出來的signal是4個不同位置的Valid訊號因此要把這4個訊號透過判斷LoadSelect來分別將對應的Valid1, Valid2, Valid3, Valid4等等AND起來形成輸入給Controller的Valid訊號。

**Comparator:**

透過LoadSelect去判斷TagRAM的output Tag1, Tag2等等是否跟CPU傳送過來的Tag相同。

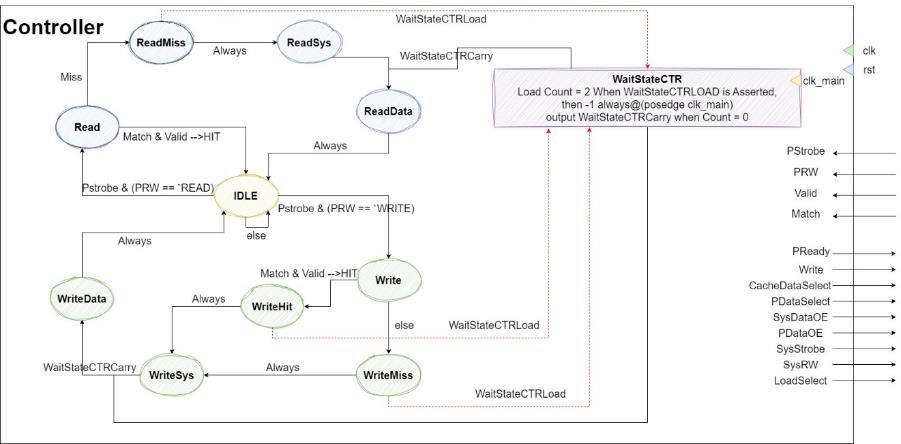
**PDataMUX:**

輸入訊號為DataRAM的output以及SysData，透過控制訊號PDataSelect來決定PDataout。

**CacheDataSelectMUX:**

輸入訊號為SysData以及PData，透過判斷CacheDataSelect訊號來決定輸入的DataRAM的Input Data。

4. Controller & WaitStateCTR



🡪Controller主要分為10個State來做出ReadHit, ReadMiss, WriteHit, WriteMiss四種狀況。透過TagRAM的Tag比對得到的Match以及ValidRAM通過AND之後的Valid訊號做Hit, Miss的判斷並透過WaitStateCTR模擬Memory需要多個Cycle才可以取用的狀況。(在此模擬要等待兩個clk\_main cycle)。

使用兩個不同Period的clk做成Cache的原因是因為Cache的單一clk操作十分簡單只是純粹的讀寫或條件判斷因此我認為不應該受CPU中的Critical Component ALU的運算速度形成的clk period影響。使用獨立出來的clk period去實現Cache才不會造成多數時間都在等待clk assertion。

每個State的output signal以及一些Internal的控制訊號表列如下

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| State | WaitStateCTRLoad | PReadyEnable | Ready | Write | SysStrobe | SysRW | CacheDataSelect | PDataSelect | PDataOE | SysDataOE |
| IDLE | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Read | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| ReadMiss | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| ReadSys | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| ReadData | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| Write | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| WriteHit | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| WriteMiss | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| WriteSys | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| WriteData | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

🡪PReady = (PReadyEnable && Match && Valid) || Ready

5. Waveform Explanation (Operation Correctness):

**Define State:**

0:IDLE

1: Read

2: ReadMiss

3: ReadSys

4: ReadData

5: Write

6: WriteHit

7: WriteMiss

8: WriteSys

9: WriteData

Read Write Signal: Read: 1, Write: 0

**Load: (ReahHit, ReadMiss)**

Assembly Code:

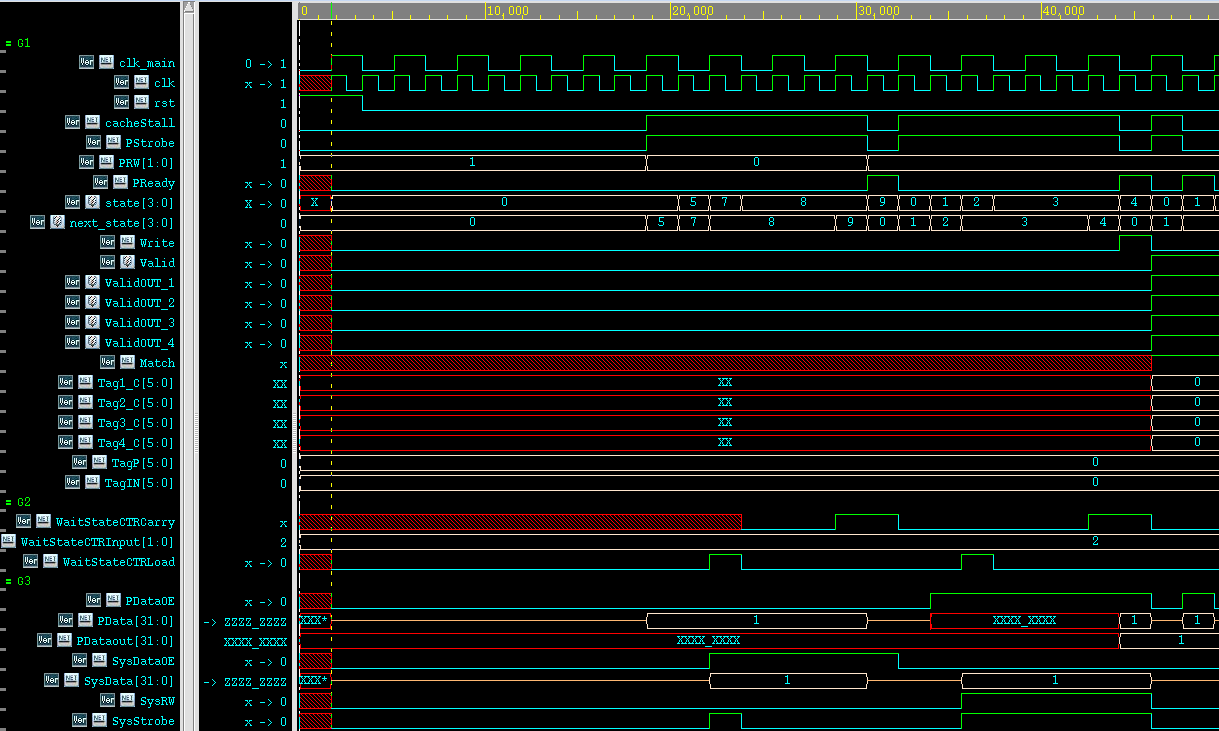
addi x5 x0 1

addi x6 x0 40

sw x5 0 x6

lw x7 0 x6

lw x8 0 x6

****ReadHit and ReadMiss Waveform

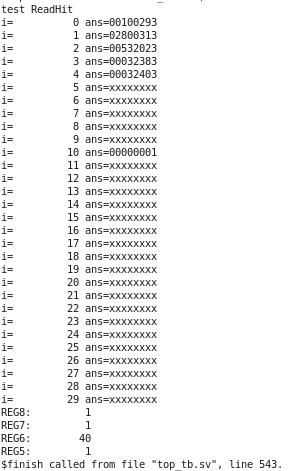
從波形中可以看到前面cachestall沒有起來的部分就是在做addi的部分此時做完之後x5 = 1, x6 = 40。進入下一個operation Store Mem[40]的位置為x5的值，此時正在取用Cache因此cachestall為high表示停住CPU運轉，同時PStrobe=1和PRW=0表Write。這部分因為著重在Read的部分就先不解釋Store的Flow，此時Mem[40]的位置=1 (x5的值)。

**ReadMiss:**

接著做第一次的Load將Mem[x6]的值存到x7裡面。波形中可以看到State的變化為0 🡪 1 🡪 2 🡪 3 🡪 4表示走的路徑為ReadMiss的路徑。並且可以看到當state = 2 (ReadMiss)的時候WaitStateCTRLoad為High表示將WaitStateCTR中的Count設為WaitStateCTRInput = 2。並在下一個min\_clk進入ReadSys state等待WaitStateCTR倒數，可以看到state hold在3也就是ReadSys。當Counter變為0時WaitStateCTRCarry變為high此時在下一個min\_clk，FSM偵測到Carry = 1 state變為ReadData(4)。在這個State中可以看到PReady跟Write都變為high代表讓CPU知道我已經拿完資料了可以繼續執行運算並且Write訊號使DataRAM，TagRAM，ValidRAM都寫入對應的資料，也就是ReadMiss中更新Cache中資料的部分。此時可以看到一直為Don’t Care 的PDataout在這個State中變為1且PDataOE = 1(Tristate Buffer控制inout port)代表選擇PData = PDataout = 1而不是CPU傳送過來的資料也就代表成功Load出Mem[x6]中的值。

**ReadHit:**

接著在執行一次Load的指令，這次是將Mem[x6]的值存到x8裡面。同樣可以從波形中看到State的變化只有從0變到1就馬上變回來了表示為ReadHit的路徑。從波形中可以看到ValidOUT\_1, 2, 3, 4都為1代表在上一個operation有確實更新ValidRAM中的資料。同時Tag1, 2, 3, 4都為0且TagP也就是從CPU傳送過來的Tag也為0代表TagRAM也有被確實更新，因此Match訊號為1代表Tag有對到。由於Match跟Valid都為1因此可以從上面的Controller FSM附圖中看到當為Read State且 Match & Valid = 1時會代表ReadHit回到IDLE State。此路徑不會進入WaitStateCTR所以WaitStateCTRLoad不會變為High。而在下半部的inout port處理上可以看到因為ReadHit了所以System Memory的assert訊號都沒有變為high代表沒有取用到Memory成功實現ReadHit。

****

從Terminal接線出來看各Register中的值 (與Assembly Code 的結果一致)

**Store: (WriteHit, WriteMiss)**

Assembly Code:

lui x5 0xaaaab

addi x5 x5 -1366

lui x6 0xccccd

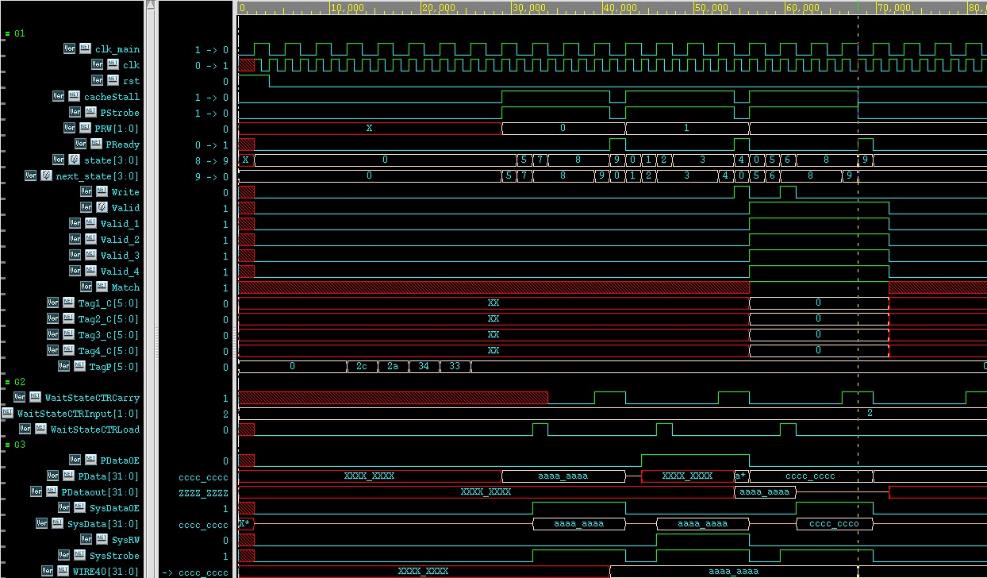
addi x6 x6 -820

addi x7 x0 40

sw x5 0 x7

lw x9 0 x7

sw x6 0 x7



WriteHit and WriteMiss Waveform

從波形中可以看到前面幾個cycle Cache Stall都沒有變為high表示沒有取用到Cache此時當完成前面5個運算operation後x5 = aaaa\_aaaa，x6 = cccc\_cccc，x7 = 40。

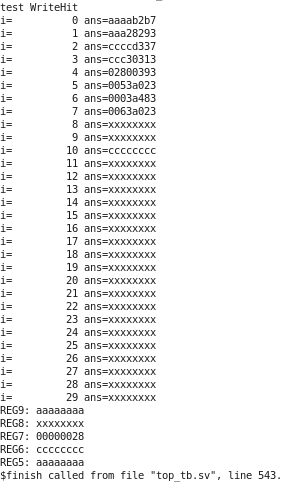
**WriteMiss:**

此時會執行將x5的數值存入Mem[x7]的位置因此可以看到此時PStrobe = 1，Cachestall = 1，以及PRW = 0代表Write動作。同時可以看到State的變化為0 🡪 5 🡪 7 🡪 8 🡪 9表示WriteMiss的途徑。由於目前TagRAM，ValidRAM中都沒有有效值因此Match以及Valid都為Low表示Miss狀況。進入State7( WriteMiss後)可以看到此時會啟用WaitStateCTRLoad使WaitStateCTR被Initial成2開始並進入State8 (WriteSys)等待倒數，同時啟用SysStrobe開始access SystemMemory且在兩個main\_cycle之後Carry會= 1使State變為State9(WriteData)。此時透過波形可以看到在State9的時候SysData = aaaa\_aaaa，SysStrobe = 1，SysDataOE = 1，SysRW = 0 (Read)且已經等完模擬的Memory取用時間了所以DataMEM[x7]的位置會被寫入aaaa\_aaaa。結果可以從波形圖的最下面Wire40也就是從testbench接線出來的DataMEM[40]的位置的Data變化看出來確實在WriteData的State值被更新了。另外也可以用assembly code的flow以及下圖terminal的結果驗證。(sw x6 0 x7之後接lw x9 0 x7 若x9的位置有aaaa\_aaaa就是成功)

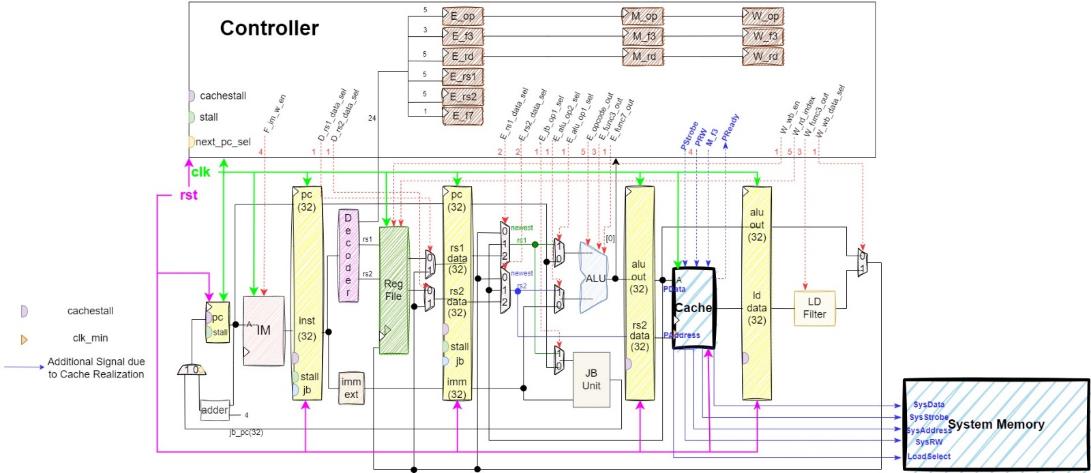
**WriteHit:**

做完WriteMiss之後會先做一次lw，此lw的目的不僅僅是檢查上面所說的WriteMiss有沒有成功，更因為是我們設計的Policy是WriteThrough因此當WriteMiss的時候是不會更新Cache中的Data的因此我們需要做一次的ReadMiss去讓Cache中有這個Tag的資料才能模擬出這次的WriteHit。當lw完成之後可以看到在lw的ReadData State中Write訊號有變為high且TagP也有Data傳入代表有更新Cache中的資料。因此我們在執行最後一個指令sw x6 0 x7的時候就會首先看到state的變化為

0 🡪 5 🡪 6 🡪 8 🡪 9代表WriteHit的路徑。接著看到Valid的訊號全部為high代表儲存的資料為Word且4個Tag都Valid且因為Tag1, 2, 3, 4都 = TagP所以為WriteHit。接著在下一個State時WaitStateCTRLoad變為high表示進入等待SysteMemory取用時間。此處與WriteMiss較為不同的是WriteHit的狀況會更新Cache中的資料因此可以看到在等待期間Write訊號有變為high代表更新了Cache中的Data。最後，同樣可以看到在WriteData的State時DataMEM[40]的位置變為cccc\_cccc代表成功寫入System Memory。



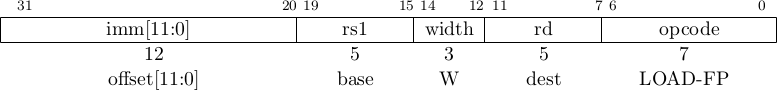
從Terminal接線出來看各Register中的值 (與Assembly Code 的結果一致)

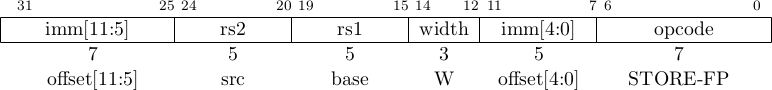
**** 6. New Structure of PipeLine CPU with Cache Installed:

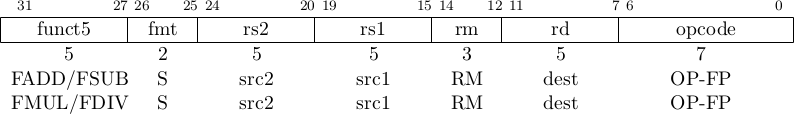
🡪上圖為新增Cache之後改變的Structure圖，主要改變的點為在每個Stage Register都新增了一個cachestall的控制訊號，用來在等待Cache把資料丟回來的時候把指令Hold住。另外就是原本跟DataMEM的溝通只需要ReadWrite訊號就好現在多了一些用來給Cache做判斷的訊號。另外就是除了把SystemMemory丟到Flow的外面獨立出來跟Cache作互動。中間各Signal的功能與改變時機都在波形以及Design Concept解釋過了。

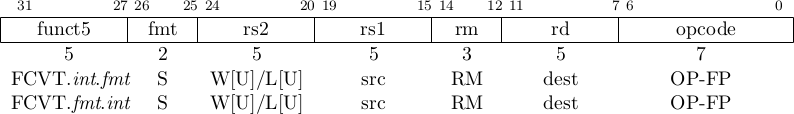
* 1. Single-Precision Floating-Point Processor

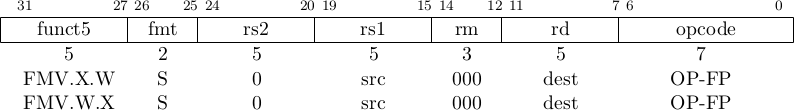
指令集架構：RV32F (modified)











備註：Let fmt=2’b00(32bit單浮點數), RM=3’b001(無條件捨棄法)

備註：rs1, rs2, rd為regFile一般暫存器; frs1, frs2, frd為fregFile浮點數暫存器

FLW: 將rs1的值+simm12作為地址，讀取DM[addr]存入frd中

FSW: 將rs1的值+simm12作為地址，讀取frs2存入DM[addr]中

FADD: 將frs1, frs2的值做加法，以無條件捨棄法存入frd中

FSUB: 將frs1, frs2的值做減法，以無條件捨棄法存入frd中

FMUL: 將frs1, frs2的值做乘法，以無條件捨棄法存入frd中

FCVT.W.S: 將frs1的值，以無條件捨棄法轉換至rd中

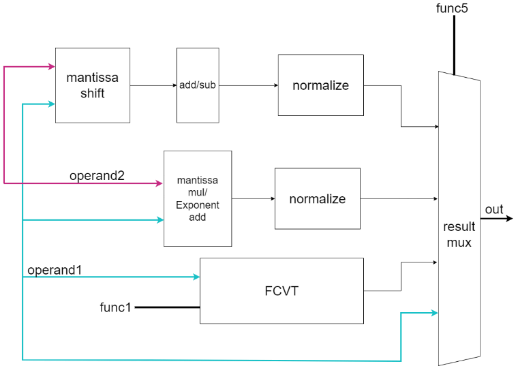
FCVT.WU.S: 將frs1的值視為unsigned，以無條件捨棄法轉換至rd中

FCVT.S.W: 將rs1的值，以無條件捨棄法轉換至frd中

FCVT.S.WU: 將rs1的值視為unsigned，以無條件捨棄法轉換至frd中

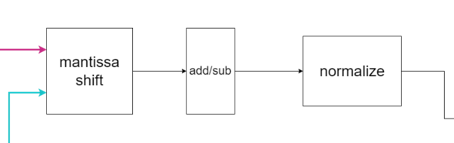
FMV.X.W: 將frs1的值，原封不動地搬入rd中

FMV.W.X: 將rs1的值，原封不動地搬入frd中



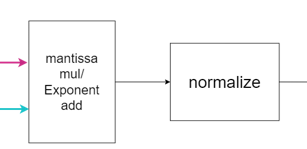
上圖是這次浮點數運算使用的FPU內部架構。 除了FLW與FSW兩個指令以外的浮點數運算都會由他進行。接下來就來說明各個指令會使用到的部分。

ADD/SUB: 加減法是我們這次支援的指令中最繁雜的，也是這個FPU的critical path。

****

浮點數加法必須先將兩數的exponent統一並將mantissa補上1做對應的位移才能作加減，因此mantissa shift這個組件必須與加減法器串聯。做完加減法後還必須將結果的mantissa正規化成1. ……，由於加減法第一個1所在的位置不能確定，因此這裡我們用了許多的mux去判斷第一個1在哪裡。 決定好後再把exponent加或減去對應的值並將mantissa做移位完成normalize。

MUL:



浮點數的乘法運算法則是將Exponent相加，mantissa1補1做相乘後再做正規化。而Exponent與mantissa的運算可以平行做，因此相較於ADD/SUB會省去相當多的時間。 最後是normalize，從小數的乘法運算可以知道兩個有23位小數的數字相乘一定會有46位小數，而且由於整數部份一定會有1，因此判斷第一個1的位置只需要判斷乘法運算結果最前面兩個位元就夠了。 一樣決定好後再把exponent加上對應的值並將mantissa做移位完成normalize。

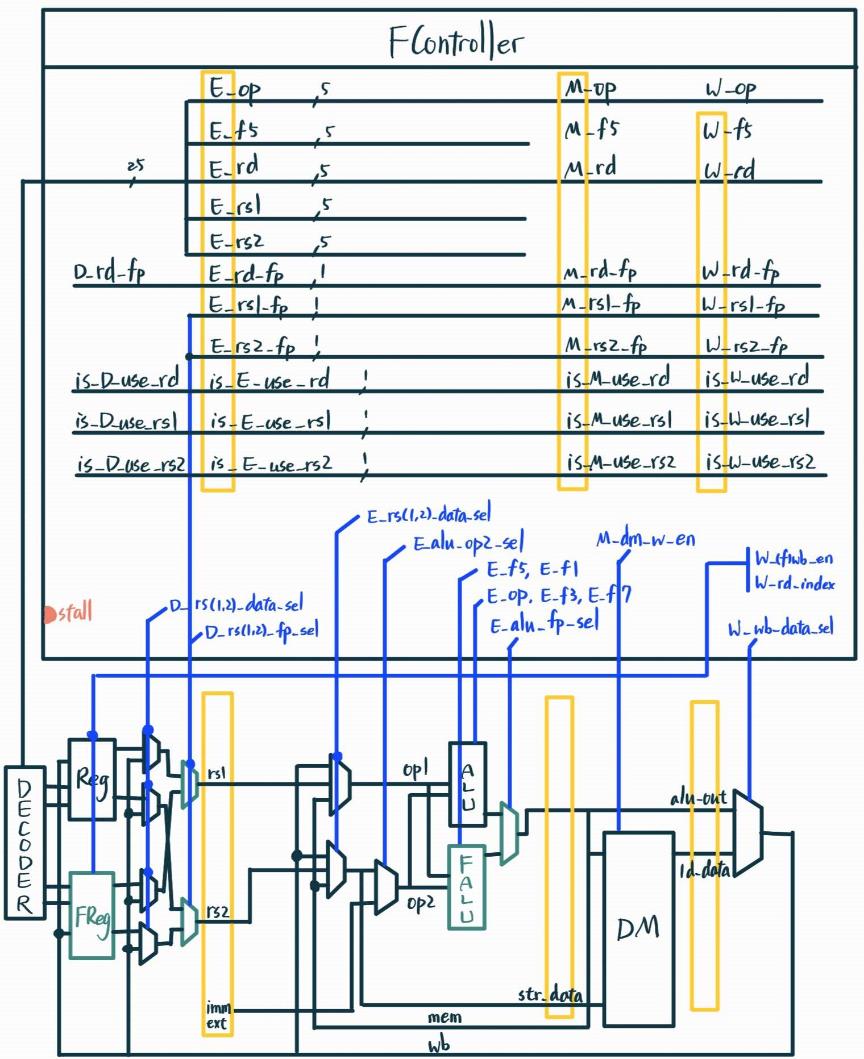


FCVT主要吃兩個訊號，operand1與func1，func1會決定要以signed或unsigned來轉換資料，FCVT內部又分為兩種 將一般register資料轉為浮點數或是將浮點數轉換為整數。 以FCVT W.S來說，FPU會將1.5123轉為整數型態1; FCVT S.W 則會將整數1轉成1.0000000……。

FMV: 這系列指令比較特別，他是直接將資料完全不做處理將freg傳至reg或將reg傳至freg，因此直接拉線到result mux就行。

FLW與FSW這兩個指令是會將freg內的資料儲存至記憶體，因此運算記憶體位置交由ALU做運算，與FPU無關。

浮點數架構圖：



上圖，是單浮點數系統裝在Non-pipeline CPU上的架構圖。

若不影響原本架構則不在圖中(如reg\_pc)。其中深綠色的module為新加入的module，黑色的module則非。黃色長方形為register，藍色線為Controller的輸出訊號。

原先考慮將原Controller和Floating Point Controller分開，做成平行架構，再使用high Z控制訊號來源。但考慮到面積和效能的影響，最終還是選擇將兩者合而為一，最終變成上圖的架構。

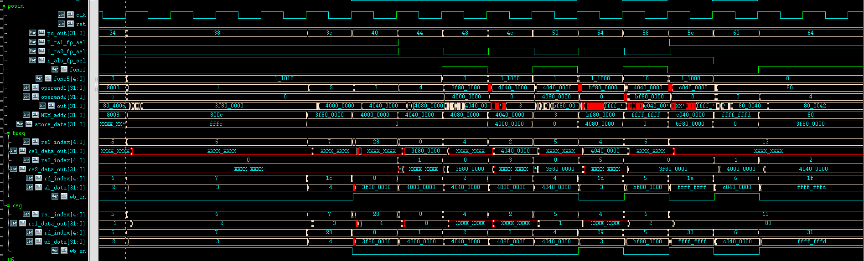
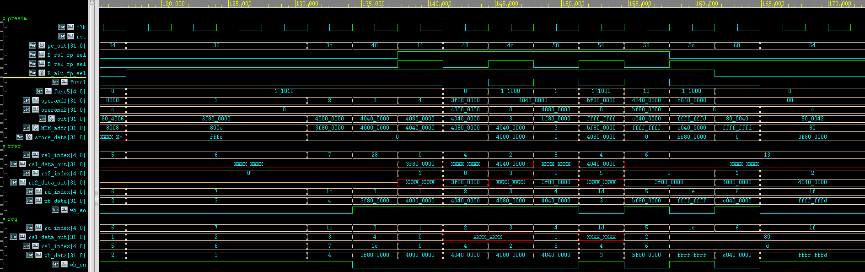
再來看到fregFile，這是跟regFile極為相似的module，唯一的差別是fregFile的f0(ft0)不同於regFile的x0，不會一直為0。

而FPU又或者FALU，是專門做浮點數運算的單元，負責的指令為FSW, FLW外的浮點數指令集。因為FSW和FLW的運算是將rs1的值+simm12作為地址，皆為整數運算，因此交給ALU同load/store處理。

剩餘新增的module都是多工器，共三個。其存在的目的都是決定應該取整數還是浮點數，被controller根據不同stage的opcode所控制。如E\_reg的rs1輸入，就會先被D-stage的opcode決定應該取regfile的rs1還是fregfile的rs1；rs2同理。而ALU和FALU後方的多工器則是由E-stage的opcode決定取何者的計算結果。

其中控制mux的select訊號線，這邊統稱fp\_sel，會由FlipFlop不斷傳遞下去。原因是很多訊號線都需要知道來源為何，如處理data hazard的forwarding訊號。與其每個stage都用一次combinational電路，不如將rs1, rs2, rd的fp\_sel不斷傳下去。

波型解釋：

****

上圖為前模、後模的擷取波型一

將t0, t1, t2, t3分別存1,2,3,4後，要做fcvt.s.w，將方才的數值轉換存入ft0, ft1, ft2, ft3。再讓ft4 = ft0+ft1, fcvt.w.s將方才的數值轉換至一般暫存器t4。ft5 = ft2-ft3, fcvt.w.s將方才的數值轉換至一般暫存器t5。ft6 = ft4\*ft5, fcvt.w.s將方才的數值轉換至一般暫存器t6。即pc = 38到58在做的指令。

可以注意到這段期間，E\_alu\_fp\_sel一直為high，代表這部分都交由falu計算，也就是fcvt和加減乘法。而E\_rs1\_fp\_sel為D\_rs1\_fp\_sel的下一個FlipFlop，會在fcvt.s.w的過程中為low，因為要的是一般暫存器的值。而E\_rd\_fp\_sel為D\_rd\_fp\_sel的下一個FlipFlop，會在fcvt.s.w的過程中為high，因為要存入的是浮點數暫存器。

相反的，E\_rs1\_fp\_sel在fcvt.w.s的過程中為high，因為要的是浮點數暫存器的值。而E\_rd\_fp\_sel會在fcvt.w.s的過程中為low，因為要存入的是一般暫存器。

而加減乘法的過程中，E\_rs1\_fp\_sel, E\_rs2\_fp\_sel, E\_rd\_fp\_sel皆為high，因為都是浮點數暫存器的計算和存取。

可以看到out(falu\_out)的計算結果分別為

+1=0x3f80\_0000=(0)\_8’d127\_(1.)000…=+1.0\*2127-127

+2=0x4000\_0000=(0)\_8’d128\_(1.)000…=+1.0\*2128-127

+3=0x4040\_0000=(0)\_8’d128\_(1.)100…=+1.5\*2128-127

+4=0x4080\_0000=(0)\_8’d129\_(1.)000…=+1.0\*2129-127

+3=0x4040\_0000=(0)\_8’d128\_(1.)100…=+1.5\*2128-127

-1=0xbf80\_0000=(0)\_8’d127\_(1.)000…=-1.0\*2127-127

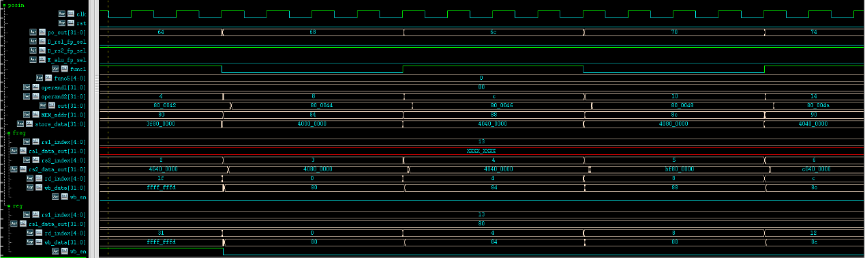
-3=0xc040\_0000=(0)\_8’d128\_(1.)100…=-1.5\*2128-127

也分別在計算後兩個cycle後(E->M->W)寫入ft0-ft6中。也將t4, t5, t6正確寫入。

前後有cycle數較多的情況，如第一個fcvt，是因為有load/store的影響，每當read/write miss時，cache就會stall所導致。

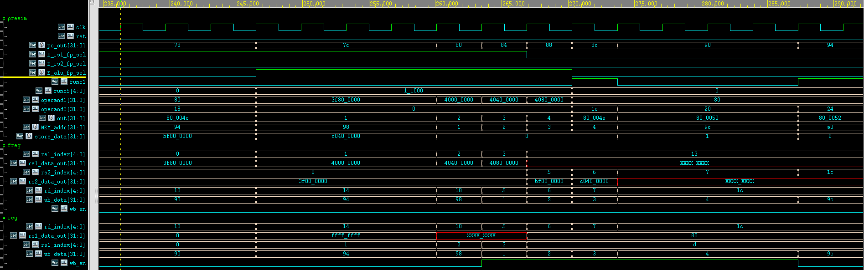
**一張含有 文字, 電視, 監視器, 室內 的圖片

自動產生的描述**

****

上圖為前模、後模的擷取波型二

接下來是將方才的ft0-ft6用fsw全部存入MEM[0x80到0x98]。可以看到這裡E\_alu\_fp\_sel為low，因為load/store都由ALU處理。可以注意到此時E\_rs2\_fp\_sel為high，因為fsw不是使用rd而是rs2。

****

**一張含有 文字, 室內, 顯示 的圖片

自動產生的描述**

上圖為前模、後模的擷取波型三

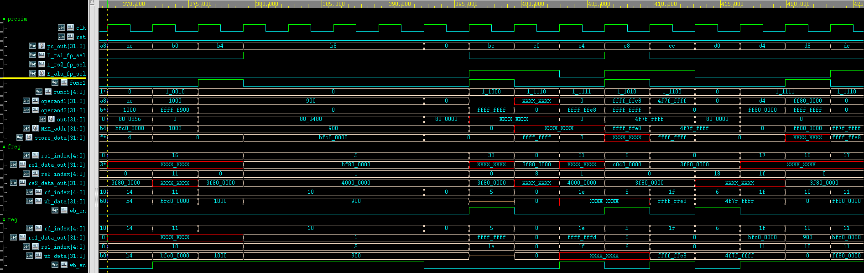
**一張含有 文字, 電視, 公寓 的圖片

自動產生的描述一張含有 文字, 室內, 顯示 的圖片

自動產生的描述**

上圖為前模、後模的擷取波型四

再將ft0-ft3用fcvt.w.s轉換至t0-t3，和上述ft4-ft6訊號同理。緊接著pc=8c到a4皆為sw，由alu控制，所以E\_alu\_fp\_sel為low。

****

**一張含有 文字, 室內, 差異, 數個 的圖片

自動產生的描述**

上圖為前模、後模的擷取波型五

pc=b8時，要將1.5用flw存入ft5，可以看到此時E\_alu\_fp\_sel為low。而E\_rd\_fp\_sel為high，因為flw不是使用rs2。

pc=bc時，做fcvt.wu.s，與fcvt.w.s同理。可以看到falu\_out把1.5無條件捨去為1。

pc=c0時，做fmv.w.x，將t5的值，原封不動地搬入ft5中。而這部份是交由falu負責，所以E\_alu\_fp\_sel為high。

pc=c8時，做fcvt.s.wu，與fcvt.s.w同理。可以看到

freg[6]=-24做轉換=0x4f7f\_ffff=(0)\_8’d158\_(1.)111…，有誤差

pc=cc時，做fmv.x.w，將ft6的值，原封不動地搬入t6中。而這部份一樣是交由falu負責，所以E\_alu\_fp\_sel為high。

剩下為ICC的不足測資，沒有新的指令就不贅述了。

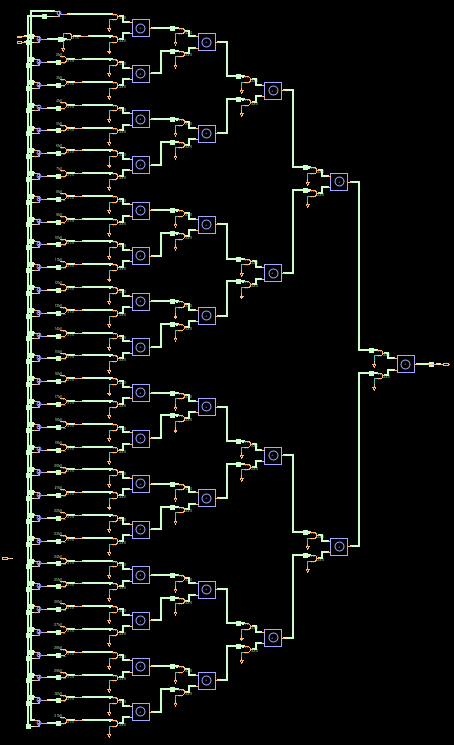
* 1. Accelerated Multiplier
     1. 測試方法

A為32bit的被乘數，B為32bit的乘數，Result為乘法結果的最後32bit。

會輸入一個clk，每過一個clk period 輸入的被乘數和乘數就會改變，而在下次被乘數或乘數改變1ns前會去驗證乘法結果是否正確(Booth’s multiplier是多等一個cycle再去驗證)，K訊號為應該顯示的結果，p訊號為Result和K的差值，若有錯誤error訊號會變為1。

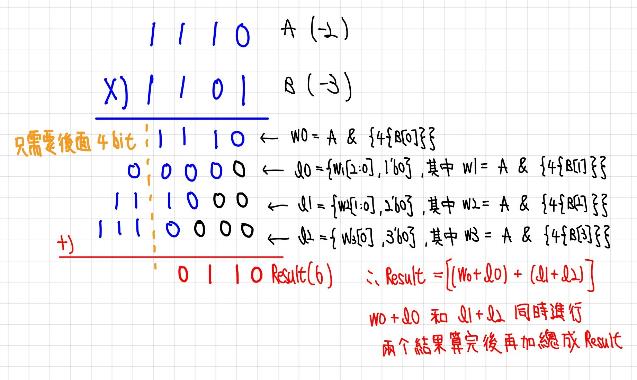
測資包括A為-128~127，B也為-128~127內的所有可能結果。

* + 1. hierarchy multiplier
       1. 架構



* + - 1. 原理

由於Result只需要乘法結果最後的32bit，所以不需要管是否進位到第33bit，因此先將wire w0~31分別設為A和{32{B[0]}}~{32{B[31]}}做bitwise and，再分別將wire l0設為{w1[30:0],1’b0}、wire l1設為{w2[29:0],2’b0}、...、wire l30設為{w31[0],31’b0}，再將w0和l0~l30依序做hierarchy兩兩相加，因此就只需要大約4個加法器的運算時間就能完成乘法的運算。下圖用4bit的乘法器:A為4’b1110、B為4’b1101為來進行舉例。



* + - 1. 速度、面積和功耗
         1. 速度

clk period為7.4ns

一張含有 桌 的圖片

自動產生的描述

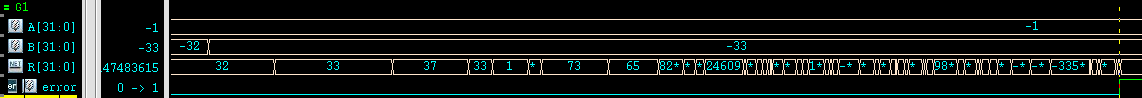
取其中一段測資：



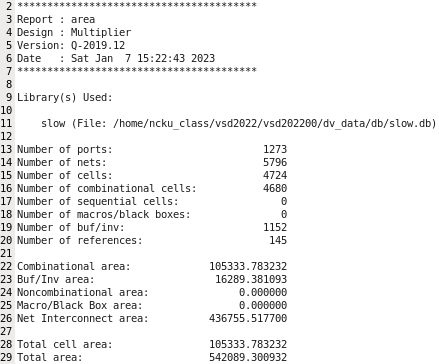
clk period為7.2ns(會有error)

一張含有 桌 的圖片

自動產生的描述



* + - * 1. 面積：(105333.783232 um2)

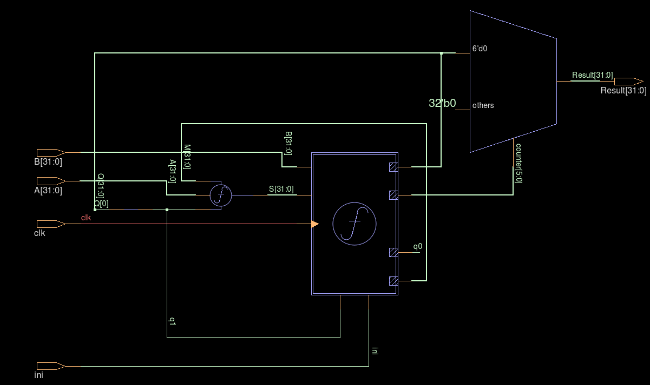


* + - * 1. 功耗：(11.3208 mW)

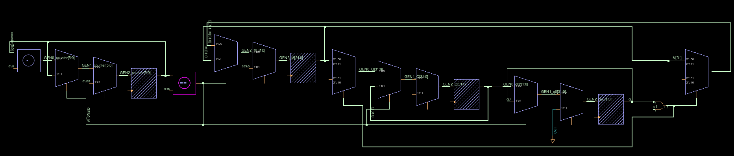
一張含有 桌 的圖片

自動產生的描述

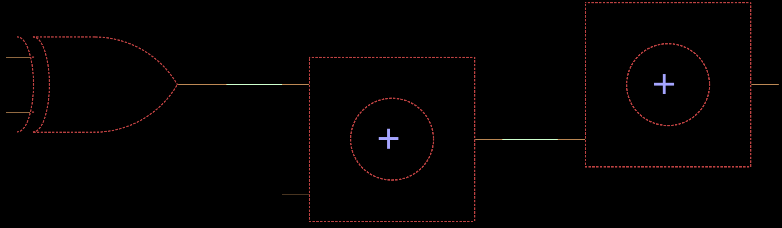
* + 1. booth’s multiplier
       1. 架構



上方大的f block：



上方小的f block：



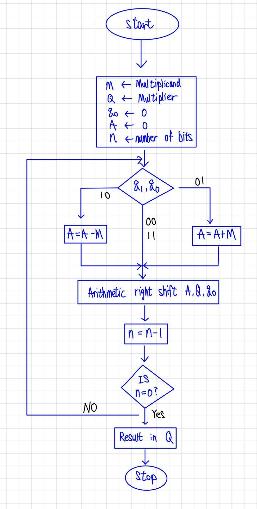
* + - 1. 原理

若有一個含有連續1的二進位乘數，例如00111110，M為被乘數，那乘法的Result就可以表示為：

M \* (00111110) = M \* (25+24+23+22+21) = M \* 62

也可以表示為：M \* (010000-10) = M \* (26-21) = M \* 62

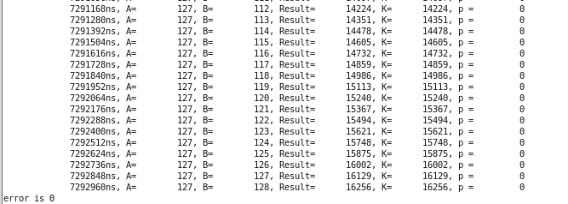
所以任何二進位連續的1都可以表示為兩個二進位數的差，q1設為Q[0]，所以下方流程中當{q1,q0}為00時，表示為連續的0，可以不用管，{q1,q0}為11時，表示為連續的1，也可以不用管，{q1,q0}為01時，表示為連續1的開頭，所以要執行加法，{q1,q0}為10時，表示為連續1的結尾，所以要執行減法，每判斷並執行完一次後就執行算術右移，詳細流程如下圖，並且用4bit的乘法器:被乘數為4’b1110、乘數為4’b1101為來進行舉例：

一張含有 桌 的圖片

自動產生的描述

* + - 1. 速度、面積和功耗
         1. 速度

clk period為3.2ns



取其中一段測資：



clk period為3ns(會有timing violation，且Result為don’t care)

一張含有 文字 的圖片

自動產生的描述

* + - * 1. 面積：(19303.099460 um2)

一張含有 桌 的圖片

自動產生的描述

* + - * 1. 功耗：(5.8598 mW)

一張含有 桌 的圖片

自動產生的描述

* + - * 1. 缺點

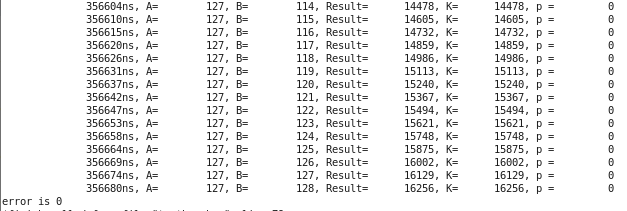
需要有internal clk

需要有ini訊號(在CPU內需要有判斷A和B是否穩定的訊號)

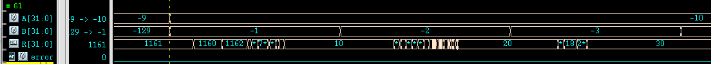
總共需要34個cycle才能算完一組乘法(要先花1個cycle reset counter，再花32個cycle運算，最後還需1個cycle穩定輸出結果)

* + 1. synopsys multiplier
       1. 速度、面積和功耗
          1. 速度

clk period為5.4ns



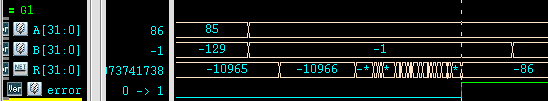
取其中一段測資：



clk period為5.2ns(會有error)

一張含有 桌 的圖片

自動產生的描述



* + - * 1. 面積：(57560.026422 um2)

一張含有 桌 的圖片

自動產生的描述

* + - * 1. 功耗：(10.7982 mW)

一張含有 桌 的圖片

自動產生的描述

* + 1. 比較：(但hierarchy可以利用pipelined來處理連續的乘法)

|  |  |  |  |
| --- | --- | --- | --- |
|  | min period | area(um2) | power(mW) |
| hierarchy | 7.4 ns  (1ns hold time) | 105333.783232 | 11.3208 |
| booth’s | 3.2 ns  (need 34 cycle) | 19303.099460 | 5.8598 |
| synopsys | 5.4 ns  (1ns hold time) | 57560.026422 | 10.7982 |

1. **心得與問題討論**
   1. 林琮偉

這是我第一次和那麼多人一起實作一個那麼大的project，在這次final project中我主要負責加速乘法器和superlint的處理，加速乘法器我嘗試了兩種架構，一種是booth’s multiplier，另一種hierarchy multiplier，在處理booth’s multiplier的時候我遇到了一些問題，首先是在CPU中送進ALU的data並不是立刻穩定，所以需要有辦法知道訊號何時穩定，並送給booth’s multiplier一個initial訊號，並且booth’s multiplier需要有一個counter來決定計算是否完成，所以它自已需要一個period比較小的clock，這些問題使得我最後選擇了另一種hierarchy的設計，只不過這兩種設計最後都無法比synopsys自己合成的multiplier來得快，但這兩種設計都可以實現pipelined，所以在很多個連續乘法指令應該就可以比synopsys的來得快，做完這次project後我感覺synopsys真的非常厲害，它不但速度基本上不會因為輸入data不同而影響到速度，而且面積和功率都可以比hierarchy multiplier來得小，希望未來能夠持續精進自己，做出比synopsys更好的multiplier。

* 1. 陳正崙

這次負責實作的部分為Cache的設計與實現，我認為雖然Cache本身的操作不是很複雜，僅僅只是資料上的讀寫與一些條件判斷而已，但在這次的Project中我認為最困難的點在與Sequential 電路的設計需要精密的控制clk以及各式的控制訊號使Data的傳遞正確也能被正確讀寫到。其中又屬因為我在設計的時候採用的是two phase 的clk，不僅Cache內部的FSM要依照internal clk去走又因為System Memory吃的是CPU的main\_clk所以我需要在對的時間點傳出Data讓main\_clk能夠在對的posedge上吃到Data。另外一點就是PData和SysData的inout port的控制我認為也是我遇到的一部分困難。尤其是在Debug的時候當把Cache加入原本的CPU架構後會遇到許多問題例如當取用Cache的時候需要把其他的Stage Register的資料與operation hold住，因此就需要去調整CPU controller並新增Cache Stall訊號等等。總的來說這次Cache的設計我認為雖然Debug的過程真的很痛苦但最後能夠看到Cache Hit那些的實現縮短cycle的取用mem真的很有成就感。比較可惜的是時間上不太夠不然我想試試看把Cache跟CPU吃同一個clk去合成看看能不能縮短clk period，因為在加入Cache之前最短period大概是16ns但在加入Cache之後變為34ns，我認為主要的原因是因為我的Cache的Period是main\_clk的一半因此在進入Output Data到CPU的State的時候是只有存在半個main\_cycle的因此alu實際上運算的時間可能只有半個cycle，但可惜沒有時間去驗證我的猜想，可能會之後再找時間去測試看看是不是與我預期的一樣。這次Project做的雖然辛苦但最後成果出來的時候很有成就感，謝謝組員與助教的努力與幫忙。

* 1. 林伯璨

這次的作業是我第一次與其他人合作完成一個大型的project，我從這次的經驗中學到了許多東西。首先是對於我一開始負責的部分-浮點數運算有了在硬體上的認知，浮點數運算的法則有學過計組都會知道， 然而在硬體上實現我遇到的第一個問題就是如何有效率地將兩個運算資料做有效的處理。尤其是加減法運算。再來是精度的問題，之前就有耳聞單精度浮點數很容易出現誤差，我一直覺得這個問題不太會遇到，然而在我自己實作硬體時就會發現真的要捨棄一些數據才不會過度浪費硬體，尤其是當兩數的數量級有落差時會更加明顯。再來是icc，我們要想辦法去發想要如何更有效的測過所有的狀況與越多的位元，我們用了許多技巧: 放指令在超遠的地方、設計互補測資等…，當仔細思考後就會知道不是盲測很多測資就有效，另外我為了要生測資還意外知道linker腳本要怎麼讀，挺酷的。最後是layout，這個雖然有很多設定我都不太能夠完全理解實際意義，但是我看他給我的數據我覺得他的最佳化真的超級強。另外看layout的圖我覺得挺紓壓的，雖然最後因為工作站當掉沒截到圖。除此之外我也從組員身上學到不少，多人project最麻煩的就是要合併，而組員良好的coding style與詳盡的註解超級加速合併的過程。我也是在這次才特別注重註解的標記，其實真的很有效。

* 1. 吳宜哲

這次的作業我十分享受，因為選到可以設計架構工作。浮點數運算接觸不深，起初覺得我不一定能負擔如此重任。但隨著資料查詢、之前上課的印象、與組員的討論，我漸漸看清整個架構。用一個下午將整個架構畫出來後，反覆參考和修改spec，總算完全清楚架構，也發現自己應該能勝任。我將FALU的module交給另一位組員，同時在晚上各自完成+驗證，並將二者合併。最終只找一次typo的bug就完成浮點數系統了。能如此快速的完成+debug真的是非常感動，之後設計電路應該都會先畫架構圖。

因為要與其他組員的系統合併，就會需要合併他人的修改，這個體驗是前所未有的。當兩人同時對原架構作修改時，就要討論怎麼修改才會不影響成果，而當兩個系統成功合併時的喜悅感真是無與倫比。

做浮點數驗證ICC時，原以為會相當複雜，但其實不然，只要將對應的bit讀取，和需要的opcode做計算，即可完成。總覺得是否過於簡單了，但其實應該是系統較簡單才能如此輕易地完成。寫對應的assembly code也不難，較為麻煩的是找assembler，還要擔心轉換會有bug，只能多次驗證。

做layout時，心情很複雜。基於相信自己努力的結晶，我不斷催眠自己要相信會成功。而且一旦失敗，會有一種一切徒然無功的挫折感。所以每個error和violation顯示時都膽戰心驚，好在最後結果是好的。

這次算是相當成功的體驗，希望未來我能更精進自己的實力，以更快的速度完成更好的電路。可惜的是每個人的coding style皆不太相同，例如只有我使用到definition.v。但有這次經驗後，應該會提醒其他組員要注意的coding style，讓後續的電路合併更順利。

* 1. 張根齊

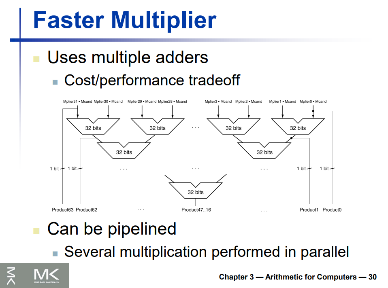
這次期末專題是我第一次和這麼多人合作，每個人都有不同的分工，而我負責的部分是最原始的RISC-V Pipeline CPU和修改原始CPU的ICC部分，因為我們要做的CPU是計組教過的CPU，要完成可以前模的CPU問題不大，但當要合成時就會碰到一些問題，有一些語法沒辦法合成，像是$signed()的數字比較就要用額外的判斷完成，還有一些verilog的寫法會合成出效能不太好的cpu，要用比較硬體的觀點去寫才會有比較好的效能。另外一部分就是ICC，我們ICC一開始toggle coverage很多都沒到70%，主要的原因是我們的pc address雖然有32bits，但實際上我們的Memory只有2^16，因為verilog不支援2^32的二維矩陣，所以pc後面16個bit根本不會toggle，為了達到70%，只能把前面16bit都toggle到，所以我們特地寫一個test program，讓pc直接跳去0hff00再跳回來0h00fc，這樣就可以讓pc前15個bit有toggle到(第一個不會，因為不會有奇數address)，讓有用到pc訊號的module達到70% coverage。這次期末專題花了大家很多時間和心力，雖然過程很不容易，但當作出來的時候還是很有成就感，真的很感謝每一位組員的幫忙。

1. **分工**

|  |  |  |  |
| --- | --- | --- | --- |
| 姓名 | 學號 | 負責項目 | 貢獻度(%) |
| 林琮偉 | E24096093 | 加速乘法器、superlint、word美編、整合 | 20 |
| 林伯璨 | E24092049 | FPU、ICC、產生測資、Layout、Dataflow | 20 |
| 吳宜哲 | E24096085 | FPU、ICC、產生測資、Layout | 20 |
| 陳正崙 | E24096239 | Cache、PPT美編、Dataflow、合成 | 20 |
| 張根齊 | E24096483 | CPU基本架構、ICC、產生測資 | 20 |
| 林軒宇 | E24092057 | Branch predict、ICC、費式數列 | (退選) |

1. **參考資料**

林輝堂老師計算機組織CH3講義P.30



Booth’s algorithm 維基百科：[布斯乘法算法 - 維基百科，自由的百科全書 (wikipedia.org)](https://zh.wikipedia.org/zh-tw/%E5%B8%83%E6%96%AF%E4%B9%98%E6%B3%95%E7%AE%97%E6%B3%95)

Booth’s algorithm 示範影片：[Booth's Algorithm for Signed Multiplication - YouTube](https://www.youtube.com/watch?v=QFXaddi-Ag8)

RISC-V 指令集架構介紹 - F Standard Extension：<https://tclin914.github.io/3d45634e/>

RV32F ISA：<https://five-embeddev.com/riscv-isa-manual/latest/f.html>

RV32F ISA Manual：<https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>