VLSI System Design (Graduate Level)

Fall 2021

HOMEWORK I

REPORT

Must do self-checking before submission:

Compress all files described in the problem into one tar

All SystemVerilog files can be compiled under SoC Lab environment

All port declarations comply with I/O port specifications

Organize files according to File Hierarchy Requirement

No any waveform files in deliverables

Student name: 王傑世

Student ID: P76091690

**Outline**

1. File structure
2. Summary
3. Lesson learned

1. Pipeline CPU

2. CPU architecture

3. Hazard

1. Instruction

1. R-type

1.1 ADD

1.2 SUB

1.3 SLL

2. I-type

2.1 LW

2.2 XORI

2.3 ANDI

3. S-type

3.1 SW

3.2 SB

3.3 SH

4. B-type

4.1 BEQ

4.2 BNE

4.3 BLT

5. U-type

5.1 AUIPC

5.2 LUI

6. J-type

6.1 JAL

1. Synthesis result

1. Area report

2. Timing report

1. Simulation

1. syn0

2. syn1

3. syn2

4. syn4

1. Superlint

**I. File structure**

P76091690

├── Makefile

├── P76091690.docx

├── StudentID

├── build

├── include

├── script

├── sim

├── src

│ ├── Def.sv

│ ├── inf\_IF\_ID.sv

│ ├── inf\_ID\_EX.sv

│ ├── inf\_EX\_MEM.sv

│ ├── inf\_MEM\_WB.sv

│ ├── IF\_S.sv

│ ├── ID\_S.sv

│ ├── EXE\_S.sv

│ ├── MEM\_S.sv

│ ├── WB\_S.sv

│ ├── Regfile.sv

│ ├── Hazard.sv

│ ├── Forward.sv

│ ├── SRAM\_wrapper.sv

│ ├── CPU.sv

│ └── top.sv

└── syn

**II. Summary**

之前有實作過single cycle CPU，這次為了提升效能，縮短timing，進一步設計成pipeline CPU，同樣是用RISCV的規格去實作。設計過程中會遇到2種Hazards，Data hazard和 Control hazard，我利用Forward加上stall CPU的一些stage來解決問題。

對於Data hazard，如果ID正在讀取的資料是舊的，可以利用Forward直接讀取更新過的資料，比較特別的是遇到load

use data dependency，就必須要stall後面的stage，確保在下個cycle執行正確的指令。

若是遇到B-type的指令，就可能發生Control hazard，PC要跳到哪個位址，我可以在EXE獲得，並將得到的位址送給同個cycle的IF處理，此時就需要清除前面錯誤的指令。

利用forwarding 和 stall 的方法解決 hazard。

**III. Lesson learned**

這次實作pipeline CPU, 共有 5 個 stages, 會遇到2種hazards需要解決, 這裡會利用1個偵測的硬體去檢查是哪一種hazard, 並且利用forwarding 加上stall CPU的方式來解決問題.

1. Pipeline CPU

IF : 從SRAM(IM)讀取資料進來

ID : 將讀進來的instruction做拆解和 register file的讀取

EXE : 根據decode完的結果, 進行相對應的動作

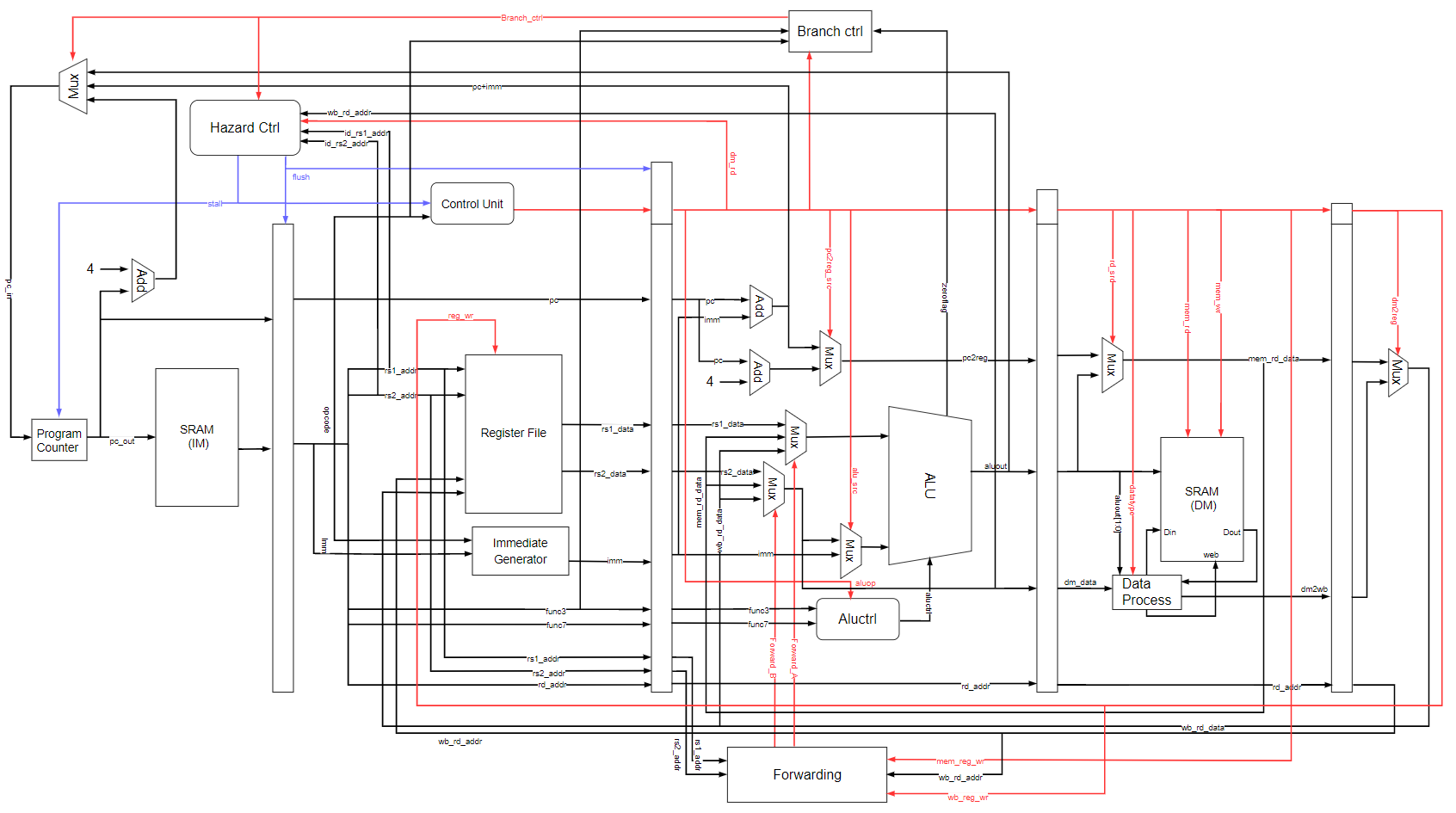
MEM : 若需要從SRAM(DM)進行讀取或寫入, 會在這個階

段進行

WB : 此階段負責將data寫入register file

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Clk0 | Clk1 | Clk2 | Clk3 | Clk4 | Clk5 | Clk6 | Clk7 | Clk8 |
| IF | IF | IF | IF | IF | IF | IF | IF | IF |
|  | ID | ID | ID | ID | ID | ID | ID | ID |
|  |  | EXE | EXE | EXE | EXE | EXE | EXE | EXE |
|  |  |  | MEM | MEM | MEM | MEM | MEM | MEM |
|  |  |  |  | WB | WB | WB | WB | WB |

2. CPU Architecture



3. Hazard

3.1 Structure Hazard

因為這裡只有1個SRAM, 而非分離的2個SRAM, 所以

若是對同1個SRAM進行存取動作, 就會發生.

這次的作業1不會發生

3.2 Data Hazard

執行的instruction 所需的資料還無法獲得, 因為上個

instruction 尚未產生結果

1) Data dependency

/\*RAW\*/

sub $2, $1, $3

add $4, $2, $3

/\*WAR\*/

sub $2, $1, $3

add $1, $3, $4

// 此CPU不會發生, 因為上面的指令先執行, 而且read會在下個cycle

/\*WAW\*/

sub $2, $1, $3

sub $2, $4, $3

// 此CPU不會發生, 因為下面的指令會晚上1個cycle執行,

// 而且 WB在第5個 stage才有動作

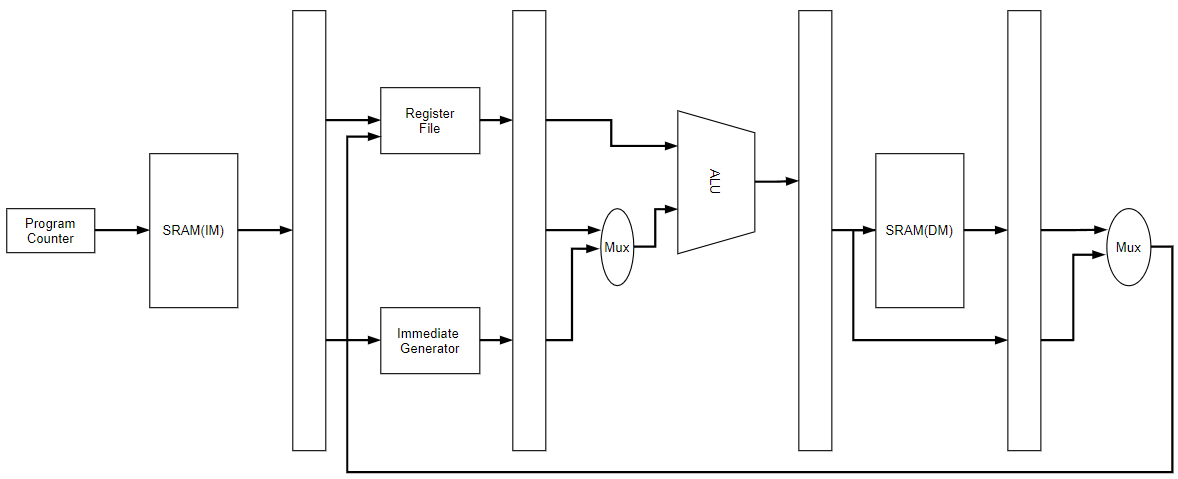
[Solution]

為了解決這些hazard, 首先檢查instruction之間是否有

data dependency, 偵測目前instruction是否有寫入

register(rd), 以及目前的instruction和其後instruction的

rd或rs是否相同, 可能會有EX hazard, MEM hazard



在紅色框(ID)要讀取register file的data時, 裡面仍然是

舊的 data, 藍色框(EXE)的才是進行運算完後新的data,

這時候就需要設計一個電路(forwarding), 將新的data送

給現在的instruction, 最後ID的instruction就能拿到最

新的data.

2) Load use data

如果遇到 load的instruction, 由於這個指令的data需要

在MEM結束後才能拿到, 即使用了forwarding仍然會

來不及, 因此暫停1個clock, 清掉control signal並且

stall 住IF和ID的instruction被更新, 避免IF讀取下一

個instruction, 然後再用forwarding才能解決.

3.3 Control Hazard

由於這次的CPU是always not taken, IF的時候會從連續

的地址讀取下次指令, 但是instruction只會在ID被解碼出

來, 確認是否為branch 和 jump, 然後在EXE才能確認是否

跳到相對應的位址在EXE的時候, 就需要將前面錯誤的

instruction 給flush, 才能確保此後的階段不會進行錯誤的

動作.

當有收到 hazard時, Hazard controller就會發出flush 的訊

號給IF 和 ID

如何清除錯誤的instruction?

在IF將錯誤的instrucion 清除

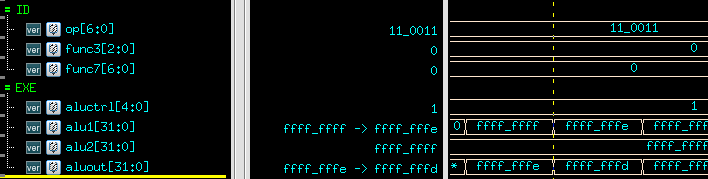
在ID將control signal設定為預設值, 因此當為預設值

時, 就可以將此instruction當作NOP

**IV. Instruction**

1. R-type

1.1 ADD



op = 7’b011\_0011

func3 = 3’b000 // ADD 或 SUB

func7 = 7’b000\_0000 // 由func7第6個bit得知為 ADD

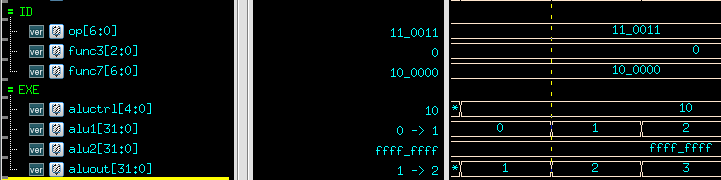
aluctrl = 5’b0\_0001 // 動作為ADD

alu1 = 32’hffff\_fffe

alu2 = 32’hffff\_ffff

aluout = 32’hffff\_fffd // alu1+alu2

1.2 SUB



op = 7’b011\_0011

func3 = 3’b000 // ADD 或 SUB

func7 = 7’b010\_0000 // 由func7第6個bit得知為 SUB

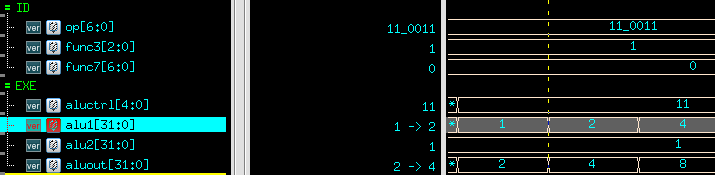
aluctrl = 5’b0\_0010 // 動作SUB

alu1 = 32’h0000\_0001

alu2 = 32’hffff\_ffff

aluout = 32’h0000\_0002 // alu1-alu2

1.3 SLL



op = 7’b011\_0011

func3 = 3’b001

func7 = 7’b000\_0000

aluctrl = 5’b0\_0011 // 動作為SLL

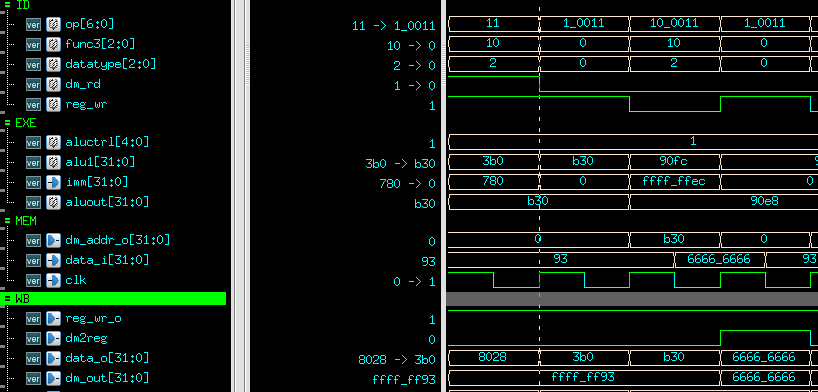
alu1 = 32’h0000\_0010

alu2 = 32’h0000\_0001

aluout = 32’h0000\_0100 // alu1向左移alu2個bit

1. I-type

2.1 LW



op = 7’b000\_0011

func3 = 3’b010

datatype = 3’b010 // 根據func3=010, 所以設定為word

dm\_rd = 1’b1 // 要讀取DM

reg\_wr = 1’b1 // lw 要將讀取的data存在register

aluctrl = 5’b0\_0001 // 動作為ADD

alu1 = 32’h0000\_0b30

imm = 32’h0000\_0000

aluout = 32’h0000\_0b30 // alu1+imm

dm\_addr\_o = 32’h0000\_0b30 // `EXE` aluout的結果,要讀取SRAM的address

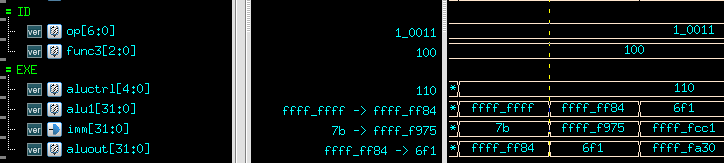
data\_i = 32’h6666\_6666 // 在負緣的clk,順利讀到SRAM的值

dm2reg = 1’b1 // 選擇要存入register file的data

// 這裡選擇從SRAM讀進來的data

data\_o = 32’h6666\_6666 // 將SRAM讀到的data存入register file

2.2 XORI



op = 7’b001\_0011

func3 = 3’b100

aluctrl = 5’b0\_0110 // 動作為XOR

alu1 = 32’hffff\_ff84

imm = 32’hffff\_f975

aluout = 32’h0000\_06f1 // alu1 xor imm

2.3 ANDI 

op = 7’b001\_0011

func3 = 3’b111

aluctrl = 5’b0\_1010 // 動作為AND

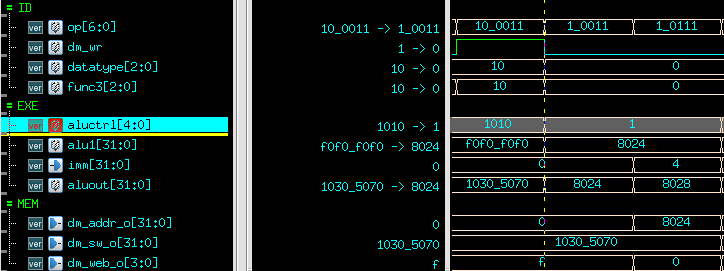
alu1 = 32’h0000\_0200

imm = 32’hffff\_fe87

aluout = 32’h0000\_0200 // alu1 and imm

3. S-type

3.1 SW



op = 7’b010\_0011

func3 = 3’b010 // sw

datatype = 3’b010 // 根據func3=010, 所以為word

dm\_wr = 1’b1 // 要寫入DM

aluctrl = 5’b0\_0001 // 動作為ADD

alu1 = 32’h0000\_8024

imm = 32’h0000\_0000

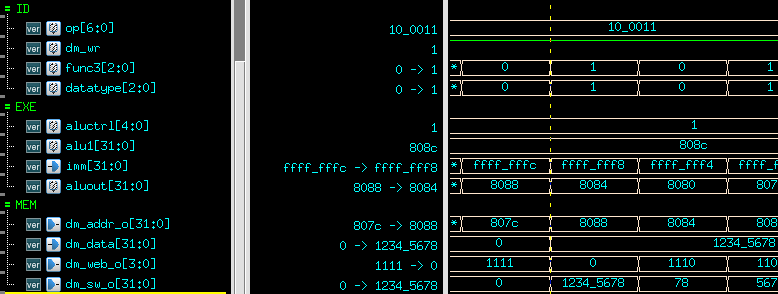
aluout = 32’h0000\_8024 // alu1+imm

dm\_addr\_o = 32’h0000\_8024 // `EXE` aluout的結果,要寫入SRAM的address

dm\_sw\_o = 32’h1030\_5070 // 寫入SRAM的值

dm\_web\_o = 4’hf // 寫入的方式,因為SW所以寫入整個word

3.2 SB



op = 7’b010\_0011

func3 = 3’b000 // sb

datatype = 3’b000 // 根據func3=000, 所以為byte

dm\_wr = 1’b1 // 要寫入DM

aluctrl = 5’b0\_0001 // 動作為ADD

alu1 = 32’h0000\_808c

imm = 32’hffff\_ffff

aluout = 32’h0000\_8084 // alu1+imm

dm\_addr\_o = 32’h0000\_8084 // `EXE` aluout的結果,要寫入SRAM的address

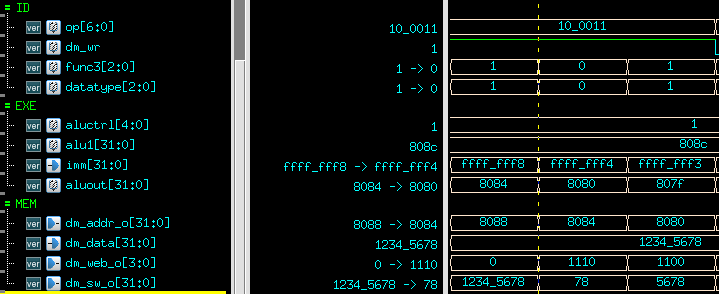
dm\_data = 32’h1234\_5678 // 要寫入SRAM的值

dm\_web\_o = 4’b1110 // 根據dm\_addr\_o[1:0]決定寫入SRAM的方式

// 4’b1110, 寫入lowest byte

dm\_sw\_o = 32’h0000\_0078 // 根據dm\_web\_o,最後寫入32’h0000\_0078

3.3 SH



op = 7’b010\_0011

func3 = 3’b001 // sh

datatype = 3’b001 // 根據func3=001, 所以為half word

dm\_wr = 1’b1 // 要寫入DM

aluctrl = 5’b0\_0001 // 動作為ADD

alu1 = 32’h0000\_808c

imm = 32’hffff\_fff4

aluout = 32’h0000\_8080 // alu1+imm

dm\_addr\_o = 32’h0000\_8080 // `EXE` aluout的結果,要寫入SRAM的address

dm\_data = 32’h1234\_5678 // 要寫入SRAM的值

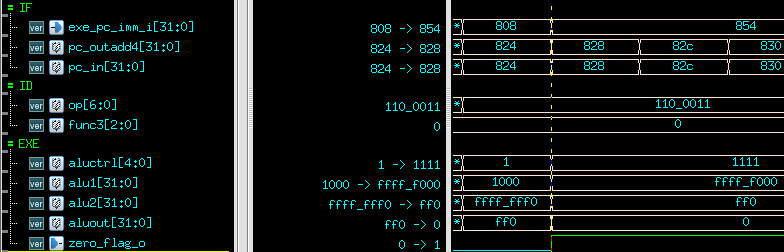
dm\_web\_o = 4’b1100 // 根據dm\_addr\_o[1:0]決定寫入SRAM的方式

// 4’b1100, 寫入lowest half word

dm\_sw\_o = 32’h0000\_5678 // 根據dm\_web\_o,最後寫入32’h0000\_5678

1. B-type

4.1 BEQ



op = 7’b110\_0011

func3 = 3’b000 // BEQ

aluctrl = 5’b0\_1111 // 動作為BEQ,檢查2者是否相等

alu1 = 32’hffff\_f000

alu2 = 32’h0000\_0ff0

aluout = 32’h0000\_0000 // alu1==alu2不成立,所以結果為0

zero\_flag\_o = 1 // 因為aluout=0,所以被拉為1

exe\_pc\_imm\_i = 32’h0000\_0054 // pc+imm

pc\_out\_add4 = 32’h0000\_0828 // pc+4

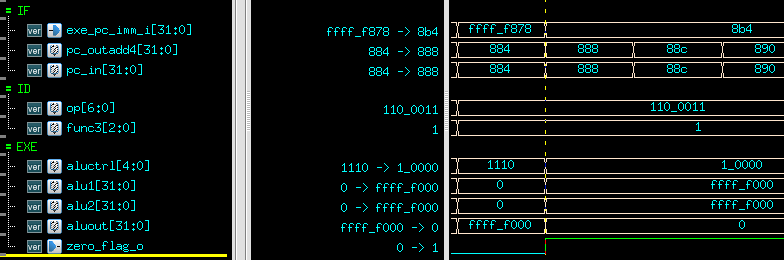
pc\_in = 32’h0000\_0828 // 根據zero\_flag\_o的結果決定選擇pc的動作

// 1:pc+4, 0:pc+imm

// 因為zero\_flag\_o=1,所以最後pc要跳到pc+4

// 32’h0000\_0824 跳到 32’h0000\_0828

4.2 BNE



op = 7’b110\_0011

func3 = 3’b001

aluctrl = 5’b0\_1111 // 動作為BNE,檢查2者是否不相等

alu1 = 32’hffff\_f000

alu2 = 32’hffff\_f000

aluout = 32’h0000\_0000 // alu1!=alu2不成立,所以結果為0

zero\_flag\_o = 1 // 因為aluout=0,所以被拉為1

exe\_pc\_imm\_i = 32’h0000\_08b4 // pc+imm

pc\_outadd4 = 32’h0000\_0888 // pc+4

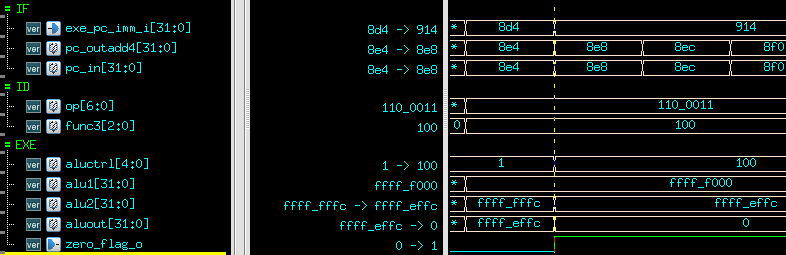
pc\_in = 32’h0000\_0888 // 根據zero\_flag\_o的結果決定選擇pc

// 1:pc+4, 0:pc+imm

// 因為zero\_flag\_o=1,最後pc跳pc+4

// 32’h0000\_0884 跳到 32’h0000\_0888

4.3 BLT



op = 7’b110\_0011

func3 = 3’b100

aluctrl = 5’b0\_0100 // 動作為SLT,檢查前者是否小於後者

alu1 = 32’hffff\_f000

alu2 = 32’hffff\_effc

aluout = 32’h0000\_0000 // alu1<alu2不成立,所以結果為0

zero\_flag\_o = 1 // 因為aluout=0,所以被拉為1

exe\_pc\_imm\_i = 32’h0000\_0914 // pc+imm

pc\_outadd4 = 32’h0000\_08e8 // pc+4

pc\_in = 32’h0000\_08e8 // 根據zero\_flag\_o的結果決定選擇pc的

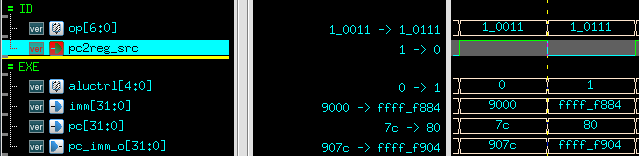
// 1:pc+4, 0:pc+imm

// 因為zero\_flag\_o=1,最後pc跳pc+4

// 32’h0000\_08e4 跳到 32’h0000\_08e8

1. U-type

5.1 AUIPC



op = 7’b001\_0111

pc2reg\_src = 1 // 會選擇 pc+4 或 pc+imm 存入register,

// 1:pc\_imm, 0:pc+4

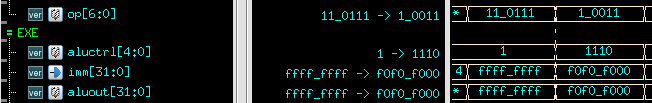
aluctrl = 5’b0\_0001 // 動作為 ADD

imm = 32’hffff\_f884

pc = 32’h0000\_0080

pc\_imm\_o = 32’hffff\_f904 // pc+imm

5.2 LUI



op = 7’b011\_0111

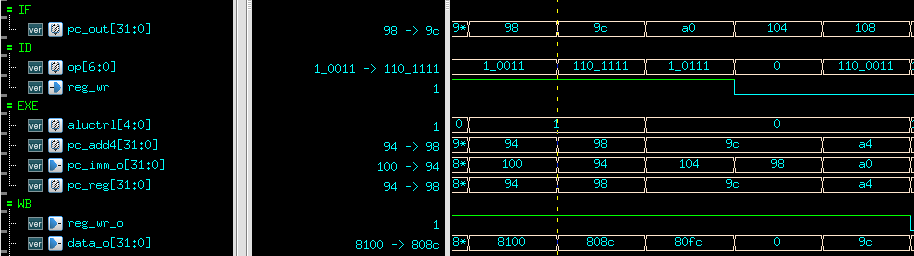
aluctrtl = 5’b0\_1110 // 動作為 IMM

imm = 32’hf0f0\_f000

aluout = 32’hf0f0\_f000 // aluout=imm

1. J-type

6.1 JAL



op = 7’b110\_1111

pc\_out = 32’h0000\_0104 // JAL等`EXE`完成後決定要跳的位址

reg\_wr = 1’b1 // 要將PC+4的值存入register file

aluctrl = 5’b0\_0000 // 沒有動作

pc\_imm\_o = 32’h0000\_0104 // pc+imm

pc\_add4 = 32’h0000\_009c // pc+4

pc\_reg = 32’h0000\_009c // pc+4,因為reg\_wr\_o=1,所以需要存

// 入registerfile

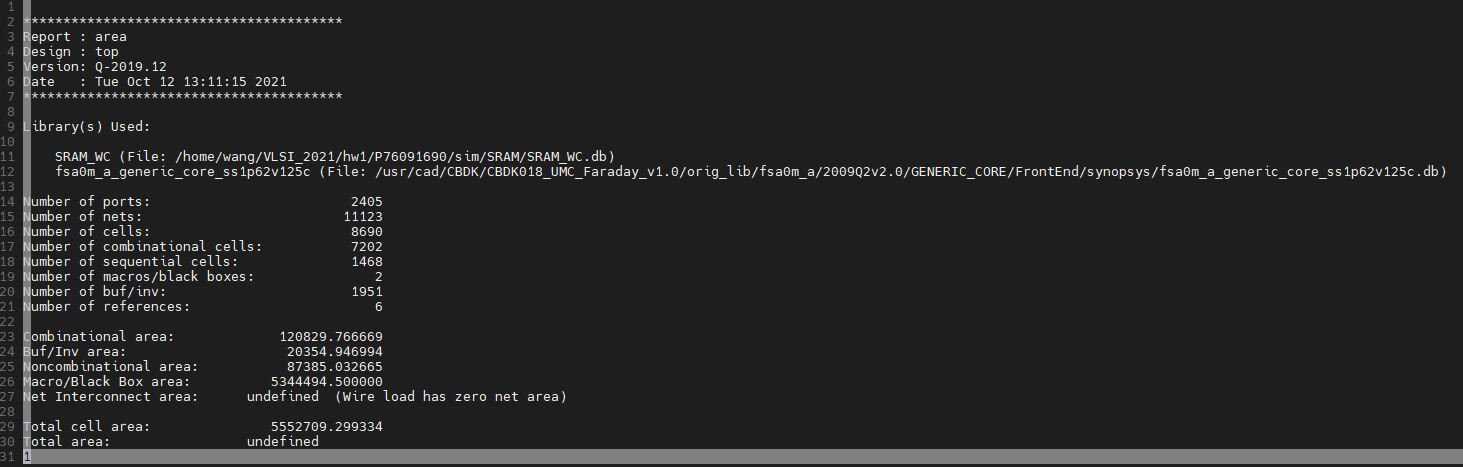
**V. Synthesis result**

create\_clock -name clk -period 15.0 [get\_ports clk]

set\_input\_delay  -max 7.5   -clock clk [remove\_from\_collection [all\_inputs] [get\_ports clk]]

1. Area Report

Total cell area: 552709.29334

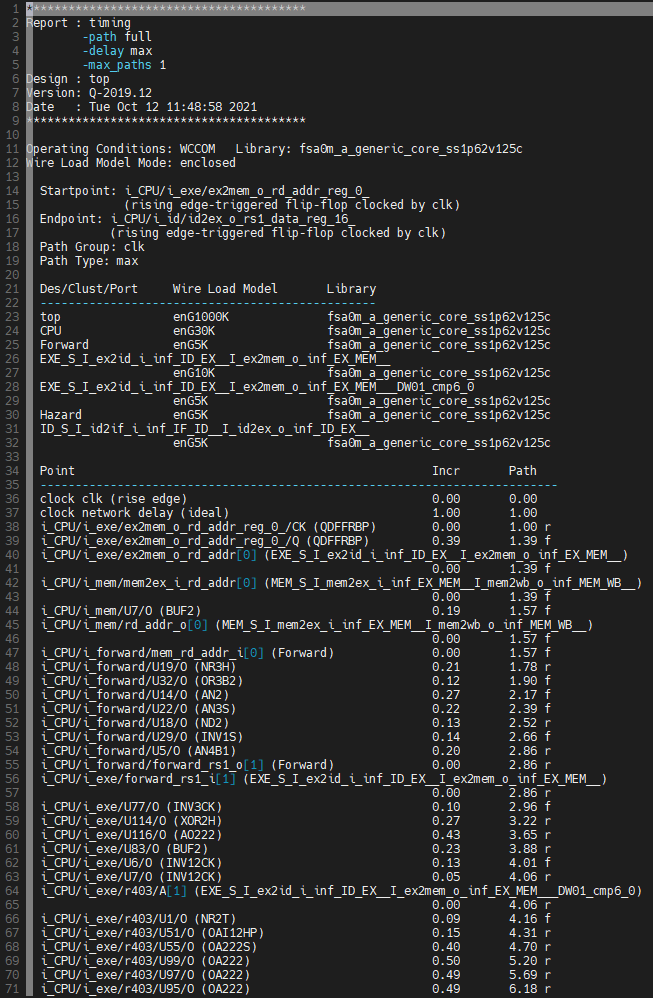
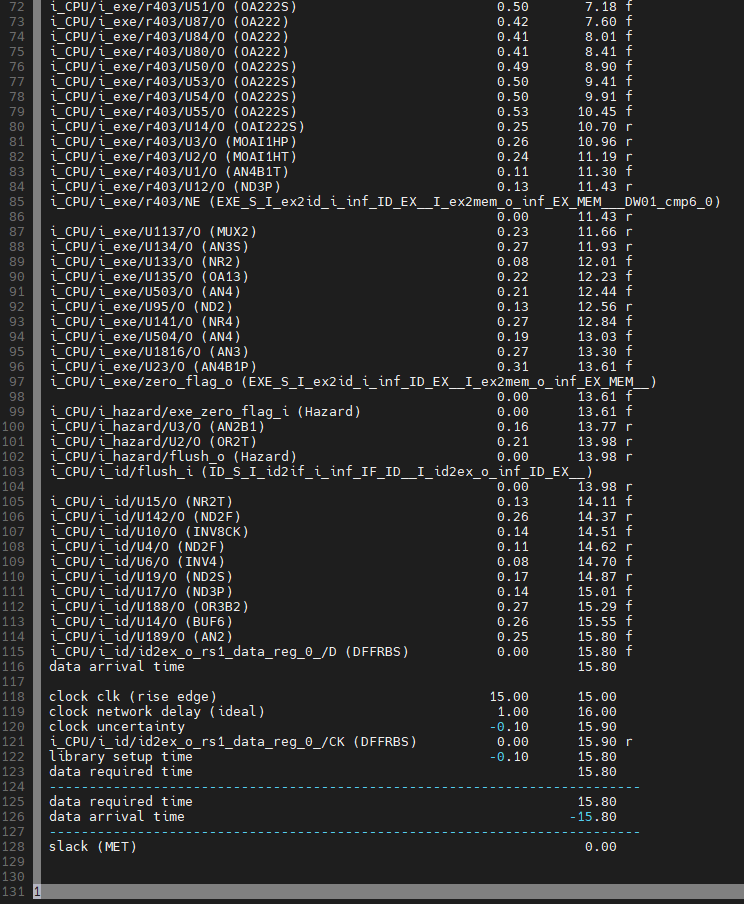


2. Timing Report

data required time 15.80

data arrival time -15.80

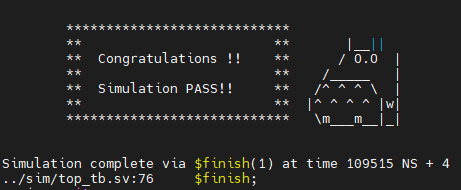
slack(NET) 0.00

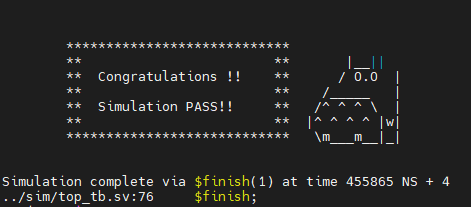
**VI. Simulation**

|  |  |  |  |
| --- | --- | --- | --- |
| rtl0 | pass | syn0 | pass |
| rtl1 | pass | syn1 | pass |
| rtl2 | pass | syn2 | pass |
| rtl3 | pass | syn3 | pass |

1. syn0 (6類37條指令)



1. syn 1 (bubble sort in ascending order)

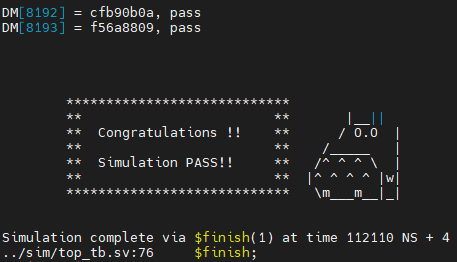


1. syn 2 (multiply)

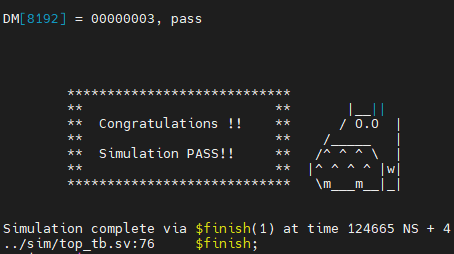
0x1a2b3c4d: 64’h000000001A2B3C4D (mul1)

0x98765432: 64’hffffffff98765432 (mul2)

mul1 \* mul2 = 64'hf56a8809\_cfb90b0a



1. syn 3 (greatest common divisor)



**VII. Superlint**

wc -l src/\* include/\*

