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: \_\_\_\_\_韋盛然\_\_\_\_\_\_

Student ID: \_\_\_\_N26102262\_\_\_\_\_\_

目錄

[一、 摘要 3](#_Toc85465313)

[二、 設計理念 3](#_Toc85465314)

[1. 指令擷取階段(IF stage) 4](#_Toc85465315)

[2.指令解碼階段(ID stage) 5](#_Toc85465316)

[3.指令執行階段(EXE stage) 8](#_Toc85465317)

[4.記憶體存取階段( MEM stage) 10](#_Toc85465318)

[5.資料寫回階段(WB stage) 11](#_Toc85465319)

[6.CPU架構完成圖 12](#_Toc85465320)

[三、 波形驗證 13](#_Toc85465321)

[prog0(simulation) 13](#_Toc85465322)

[Prog1(simulation) 21](#_Toc85465323)

[Prog2(simulation) 21](#_Toc85465324)

[Prog3(simulation) 22](#_Toc85465325)

[四、 Lessons learned 23](#_Toc85465326)

[五、 附錄 24](#_Toc85465327)

[A. Figures of the CPU 24](#_Toc85465328)

[B. Superlint report 25](#_Toc85465329)

[C. Synthetic report 26](#_Toc85465330)

[D. Area 29](#_Toc85465331)

[E. Suppoted commands 30](#_Toc85465332)

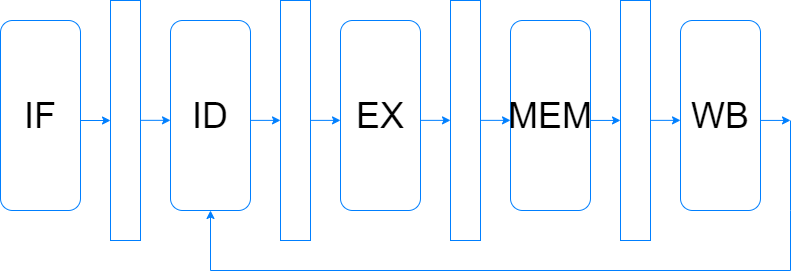
摘要

該project是用 SystemVerilog HDL 編寫用於 risc-v (rv32i) 的五級流水線 CPU，能正確執行作業要求的37個指令，在由助教提供的prog0的test pattern到自己寫的prog1~3的測試中程式都能成功執行、功能正常外，所抓到的MEM中的數值也皆正確，在Superlint中更是達到百分之百的正確性，在此設計中能有效解決各類危障(Hazard)，解決控制危障(Control Hazard)的方式為將判斷前移至ID級做處理，使沖刷的指令條數較少、條件跳躍代價(branch penalty)較小，可以被合成，PA部分Simulation Time (t0+t1+t2+t3) \* Reported Area = 749832 ns \* 5,583,829.182656 µm²，下表為達成要求對照表。

|  |  |  |
| --- | --- | --- |
| Is the Instructions function correct? | **〇** | 37 Instructions all pass. |
| Superlint Is it more than 85 percent? | **〇** | Achieve 100% error-free in Superlint. |
| Mechanism can solve forwarding? | **〇** | Using ID, EX forwarding Unit solve. |
| Can it be synthesized? | **〇** | Yes, successfully. |
| Can rtl0~3、syn0~3 be executed successfully? | **〇** | Yes, successfully. |

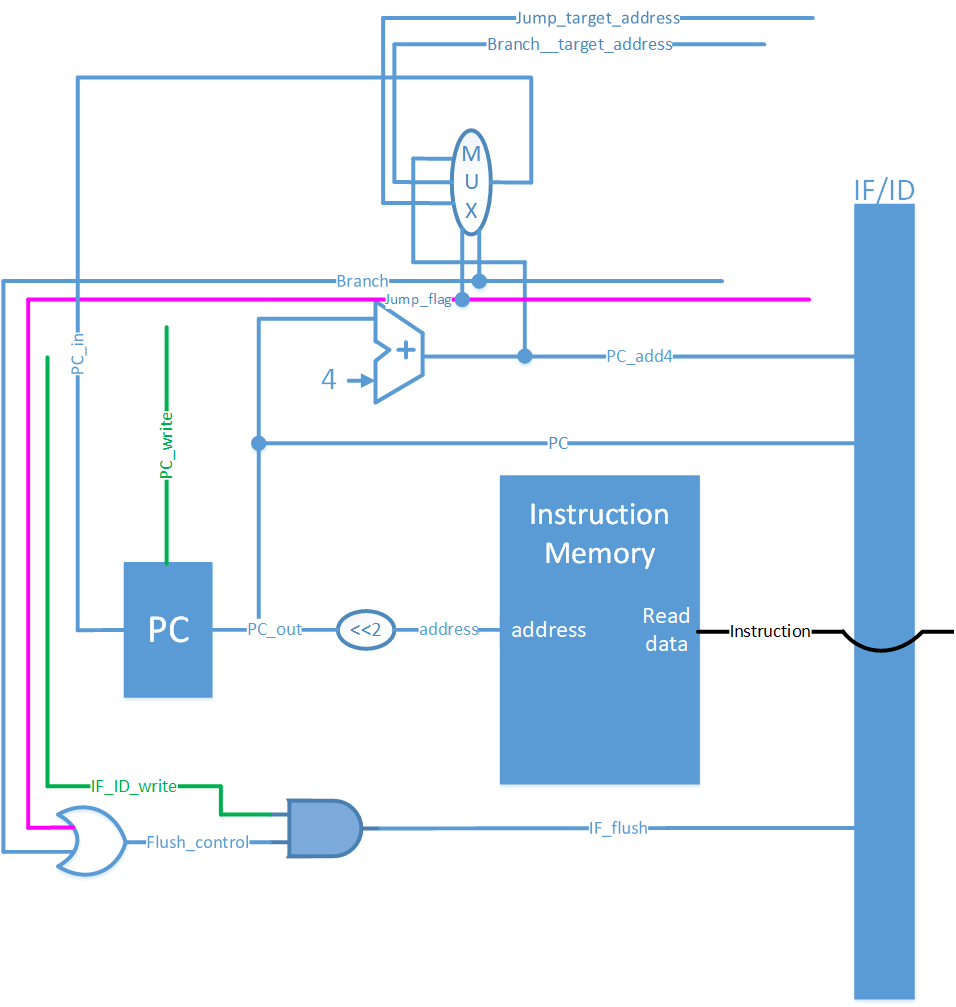
1. 設計理念

圖一為這次設計的五級流水線CPU架構圖，所謂五級流水線即是一個指令的執行需要5個時鐘週期，每個時鐘週期的上升沿來臨時，此指令所代表的一系列數據和控制訊號將轉移到下一級做處理，其中五級分別是Instruction Fetch(IF)、Instruction Decode(ID)、 Execution (EX)、 Memory(MEM)、 Write Back(WB)，不過由於Instruction Memory(IM)與Data Memory(DM)是由助教們提供的SRAM透過SRAM\_wrapper連接至我們的CPU design，雖然在CPU外面，但在此處仍將他分別歸類在IF stage 以及MEM stage中，每一級依照所需做的功能不同，有不同的設計想法以及其補償措施，在下面會逐一介紹我的設計方式。



圖一、CPU架構圖

1. 指令擷取階段(IF stage)



圖二、IF 架構

功能描述:

根據程式計數器(PC)的值從指令記憶體(IM)中取出指令編碼和對PC的值進行更新

在圖二中的電路有: 程式計數器(PC), 指令記憶體(IM), 多工器(mux3to1),加法器(adder) 左移電路(shift left 2),指令沖刷控制(IF flush control).

指令擷取步驟如下: (1)將PC的輸出值送到指令記憶體IM，進行指令提取; (2) 根據stall (即是將PC\_write設為0、IF\_ID\_write設為0)和flush(即是將IF\_flush設為1)控制信號決定是否載入新的PC值與指令至管線暫存器、不做任何動作。

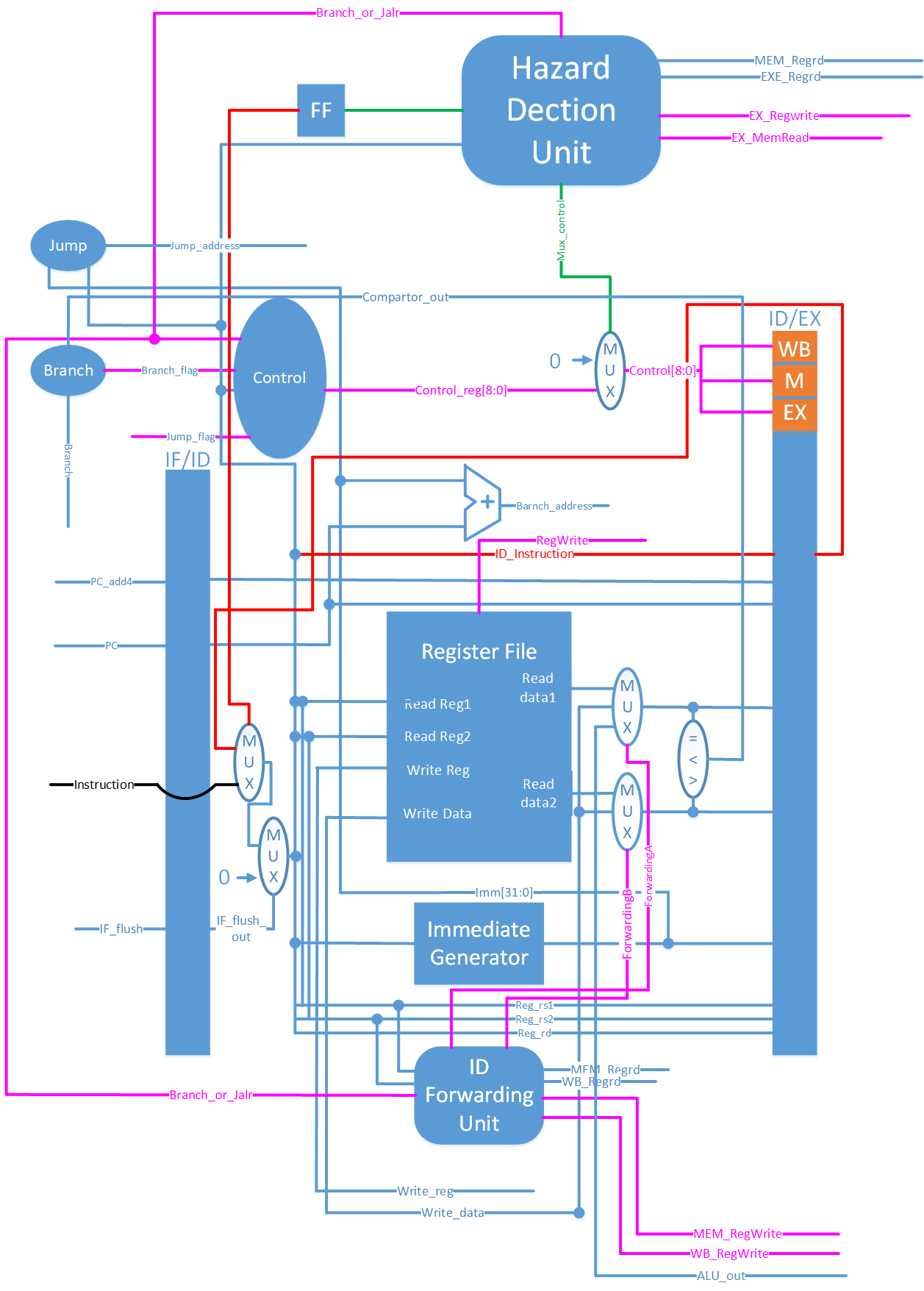
其中PC下一個進來的值分別可能是:PC+4(指令正常執行)、PC(發生hazard需要stall時)、Branch target address、Jump target address，在我程式的設計中PC的輸入是由jump以及branch兩個訊號做mux輸出的選擇，如以下表格所示。

|  |  |
| --- | --- |
| 控制訊號 | 目標位址 |
| Jump=0,Branch=0 | PC+4(正常程式依序執行時) |
| Jump=0,Branch=1 | 選擇分支目標位址(Branch、JAL 的PC\_input = PC + imm [31:0]) |
| Jump=1,Branch=0 | 選擇跳躍目標位址(JALR的PC\_input = rs1 + imm [31:0]) |

設計時所遇到的困難之處:

當初在設計此處遇到的困難點是當初沒有考慮到PC每次都是加4(在沒有要分支或是跳躍的情況下)，而這一個加4如果沒有做任何處理的話送進IM，除了原先指到的位置外會再多跳三條指令，將會導致後面取出指令一連串的錯誤，因此解決方式是從PC進去IM中要先將值除4，或者也可以是直接對PC\_out取[15:2]也可以達到相同的效果。

圖二中PC input的值必須先經由rst變成0，是因為當pc遇到第一個negative edge時，IM的DO仍然是don’t care(X)，此處讓PC output在第一個clk觸發時為0而不是4，藉此延後讀出instruction的時間，才能正常讀出第一條instruction。

2.指令解碼階段(ID stage)

圖三、ID架構

功能描述:主要負責指令解碼和暫存器陣列的讀取動作。

在圖二中的電路有:暫存器陣列(Register file)、立即值產生器(Immediate Generator)、加法器、前饋偵測電路(Forwarding Unit)、危障偵測電路(Hazard Detection Unit)、控制單元(Control Unit)、比較單元、跳躍單元(jump)、分支單元(branch)、IF/ID管線暫存器。

原先在MIPS中在ID stage是用sign extend單元做符號擴充，然而在RISC-V中立即值的解碼方式較為複雜，不同opcode的指令都有其相對應的解碼方式，因此在此處使用Immediate Generator的function做處理。

設計時所遇到的困難之處:

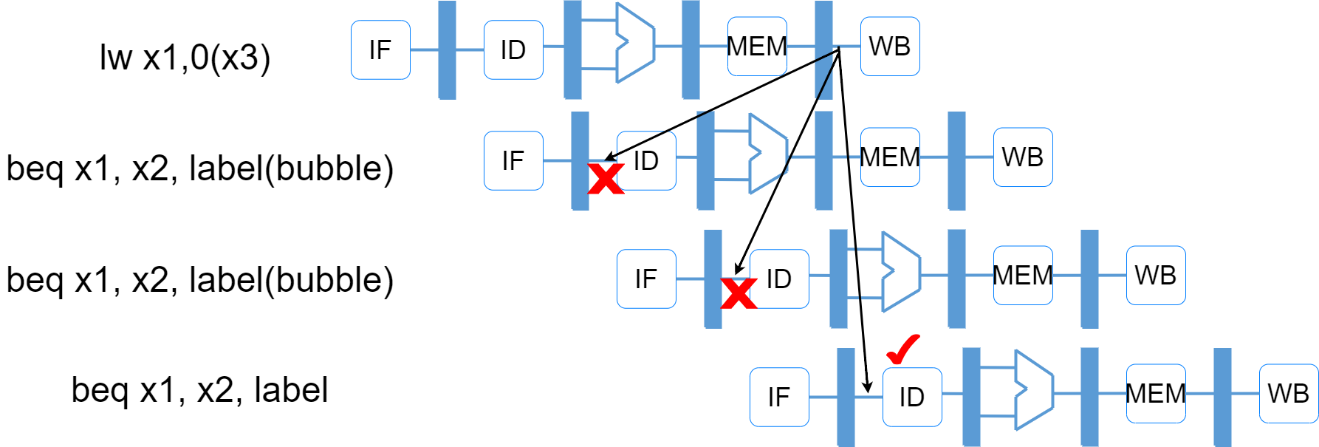
與一般MIPS五級管線的設計相比，由於我的設計是將原先在EX stage的比較單元往前移至ID stage，優點是當遇到需要做branch或是jump指令時，需要沖刷的指令條數較少、條件跳躍代價(branch penalty)較小，整體上執行速度較快，不過缺點是由於在ID將指令解碼而此時要將解碼完的值送到ID的比較單元做比較，就需要有新的在ID的前饋單元將前幾條指令算出的值往前送，另外因為比較單元前移至ID了，所以若是當我ID stage的指令是branch或jalr，且EX stage指令的rd等於ID的rs1或rs2時會需要stall一個時脈週期才能將計算完的結果前饋到ID做比較才能判斷是否要跳躍，如圖四所示

一張含有 文字, 時鐘 的圖片

自動產生的描述

圖四、Hazard by branch

普通的load-use data hazard是stall一條指令之後才可以做前饋(因load指令結果在MEM stage後才讀出)，而當我load指令的下一條是branch or jalr指令時則是stall 2 cycle(因為branch or jalr在ID做而不是在EX做，所以無法只stall一個時脈週期)，如圖五所示。



圖五、Hazard by load-use and branch

再來如果當發生stall時如果在ID級的指令是branch或jalr要怎麼去延遲我的flush訊號，使它等stall完將資料前饋後才決定是否分支或跳躍，寫if-else層級的先後判斷也需特別注意。

此外由於我的設計是將IM讀出的指令穿過IF/ID管線暫存器，也就是指令將不會儲存在管線暫存器中一個時脈週期，優點是執行速度較快(在不將其他訊號做Flip-flop delay的情況下)，這麼做可以讓IF讀出的指令與其他會經過管線暫存器的資料(ex:pc,pc+4 IF\_flush)同一時間抵達到ID階段，缺點是由於讀出的指令並不經過管線暫存器，於是當判斷出要做stall送出IF\_ID\_write=0時，指令並不會暫停，仍會送出到ID階段，我所想到有三種方式可以解決此問題:

第一種是將IF\_ID\_write訊號拿來控制IM的CS，也就是說當需要stall， IF\_ID\_write=0時，CS也設為0，讓其IM沒有輸出，且此時因為PC也被PC\_write控制為再一次輸出同樣PC值，所以當下個時脈周期來的時候，IF\_ID\_write=1 CS=1，IM將剛剛的指令傳出，達到stall的效果。

第二種是在ID設置一個暫存器，平常正常執行時就將指令的儲存進去，有需要 (IF\_ID\_write=0) 的時候，再透過mux控制將指令從中拿出，且暫存器本身就會吃一個時脈週期，因此送出的指令就會在ID重複執行一次，藉此達到stall的效果。

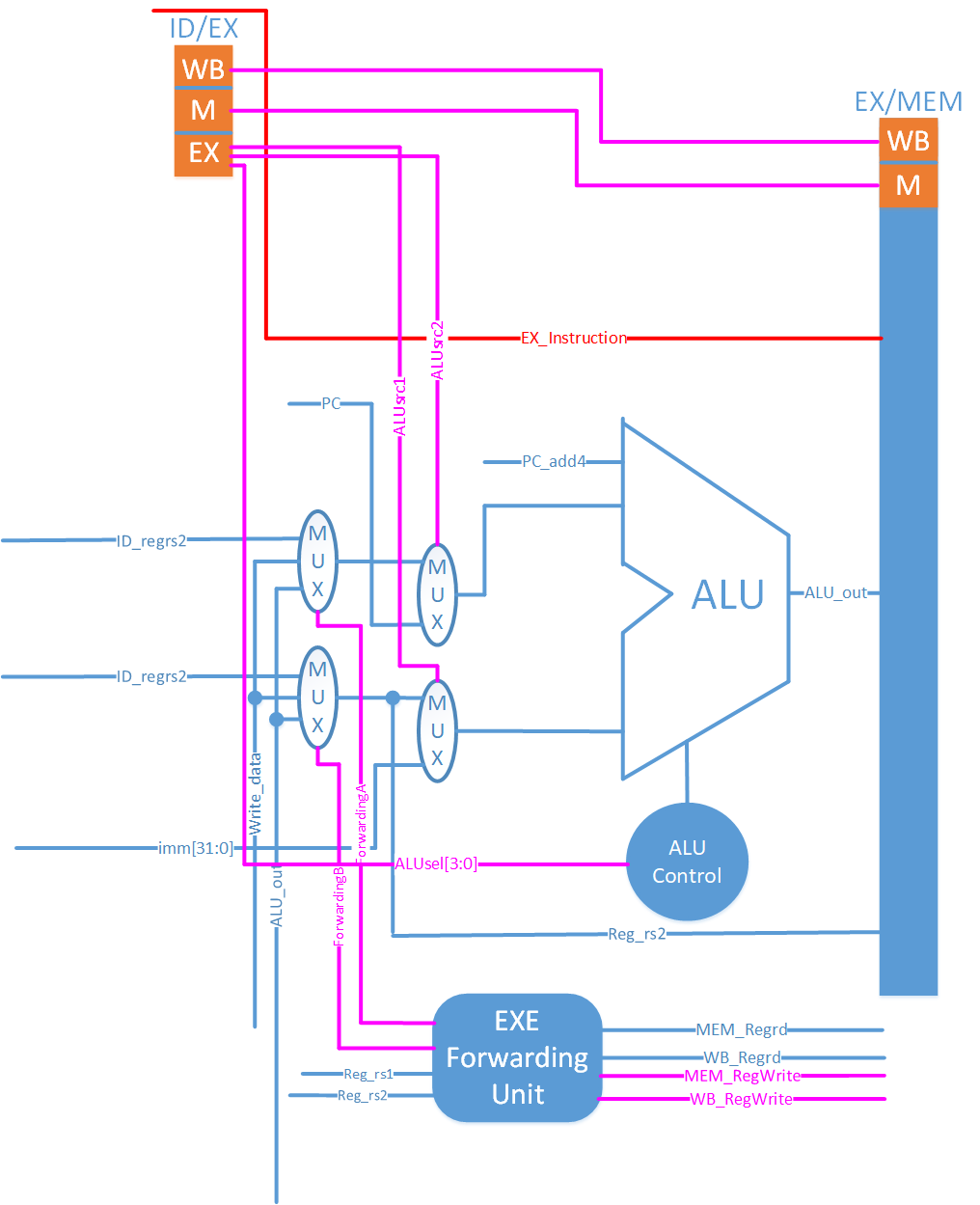
第三種為真正應用在本人設計中的方式:

與第二種大同小異概念相同，只是是拉ID/EX管線暫存器出來的指令讓他在ID stage重新執行一次，同時將到EX stage的control信號全部設為0，一般IF\_ID\_write=1，mux選擇從IM讀出的指令，當IF\_ID\_write=0，mux選擇來自ID/EX管線暫存器的指令，即可以達到ID暫停指令且送出NOP給EX的功能。

最後是ID級讀取暫存器資料的部分是以assign的方式用組合邏輯持續讀出的，不過由於寫入暫存器陣列(Register file)需要一個時脈週期的時間，除非使用正負緣的方式將寫入與讀取錯開，即假設正緣寫入時負緣才讀出，資料就可以來得及送出到EX做，不過由於本人設計為全正緣觸發，而這一個時脈週期將會導致資料來不及被讀出，因此必須在ID做前饋單元將來不及讀出的資料做判斷之後選擇性送出。

最後討論一下當初在設計時因為對flush跟stall兩者在實作上的細節不清楚花費了許多的時間在處理，flush是當我判斷出條件成立或是需要直接跳躍時，同意時脈週期在IF的PC將更新到欲分支或跳躍的地址(PC也需花費一個時脈週期)，之後在將下一個時脈週期進來ID stage的指令送0，即是全部清除為0 (由於將比較單元前移故只需要沖刷一條進ID的指令即可)，，而stall則是保留我這個stage應保留的指令，而傳到EX stage的指令只是在control訊號的部分被全部清除為0，兩者乍看很相似，但在實作上有許多細節須考慮進去。

3.指令執行階段(EXE stage)



圖六、EX架構

功能描述:主要負責一般算術邏輯和記憶存取位址的計算。

在圖六中的電路有:ALU(算術邏輯單元)、多工器、ALU control、前饋偵測電路(Forwarding Unit)、ID/EX管線暫存器。

ALU: 算術邏輯單元為整個CPU運算執行核心，此module執行的運算類別有:算術運算、邏輯運算、移位運算。

ALU control:接收來自Control Unit的控制訊號，選擇此處要做的運算類別送至ALU做處理。

在此級Control Unit的控制訊號有要給ALU control做判斷指令型態的ALUsel[3:0]，以及ALUsrc1, ALUsrc2 (ALUsrc1選擇是暫存器或是PC)(ALUsrc2選擇是暫存器或是立即值)。

Forwarding Unit會送出ForwardingA[1:0], ForwardingB[1:0]，下表有詳細描述。

|  |  |
| --- | --- |
| Forwarding A,B[1:0]=00 | 選擇來自暫存器的值 |
| Forwarding A,B [1:0]=01 | 選擇來自WB stage要寫回的資料 |
| Forwarding A,B [1:0]=10 | 選擇來自MEM stage ALU算出的資料拉回 |

ALU\_OP Encoding

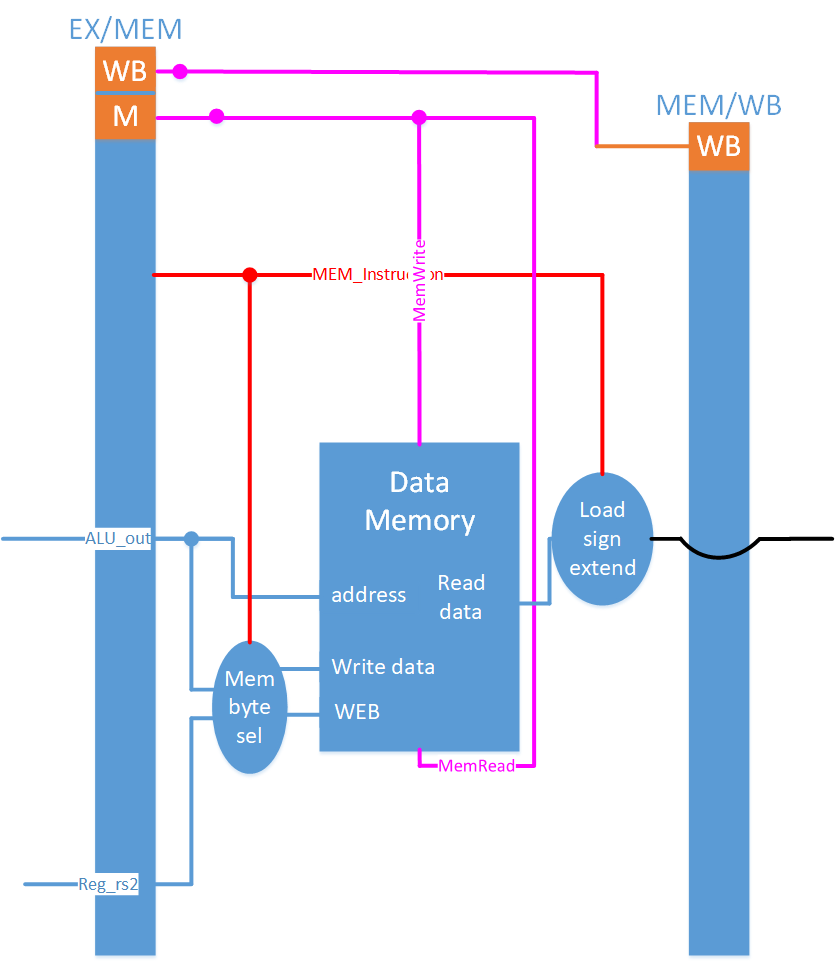
|  |  |
| --- | --- |
| Encoding  Mnemonic or Function | ALU control [3:0] |
| ADD | 0000 |
| SUB | 0001 |
| SLL | 0010 |
| SLT | 0011 |
| SLTU | 0100 |
| XOR | 0101 |
| SRL | 0110 |
| SRA | 0111 |
| OR | 1000 |
| AND | 1001 |
| LUI | 1010 |
| JALR | 1011 |
| Default | Default |

設計時所遇到的困難之處:

Forwarding Unit，在發生危障stall後傳到EX stage的指令由於只是控制信號清除為0(NOP)，不過rs1, rs2的暫存器都還是原先指令中的暫存器，將會導致前饋偵測電路判斷MEM或WB的rd是否等於在EX做NOP指令的rs1, rs2的暫存器，且此時MEM或WB一定有一級的RegWrite或是MemRead為1(因為只有這樣才會被在ID的Hazard Detection Unit給判斷出需要stall指令)，而導致我原先要前饋到ID的指令此時誤給成EX的NOP指令，導致ID拿不到想要的結果，導致一連串的錯誤。

還有要考慮的是連續三條的情況，前兩條的rd暫存器都與第三條的rs1或rs2相等時，第三條的前饋要給誰?此時應該選擇最近的也就是在第三條的前饋需要選擇上一條的rd暫存器，於是在前饋電路中也需考慮此狀況做處理，以上是當初在此處設計思考的點。

4.記憶體存取階段( MEM stage)



圖七、MEM架構

功能描述:主要負責Data Memory的存取動作。

在圖七中的電路有: 資料記憶體(Data Memory, DM)、寫入記憶體位元組選擇器(Memory byte select)、符號擴充單元(for load指令)、EX/MEM管線暫存器。

控制訊號:

Mem Read 當指令為load時為1，其餘指令皆為0

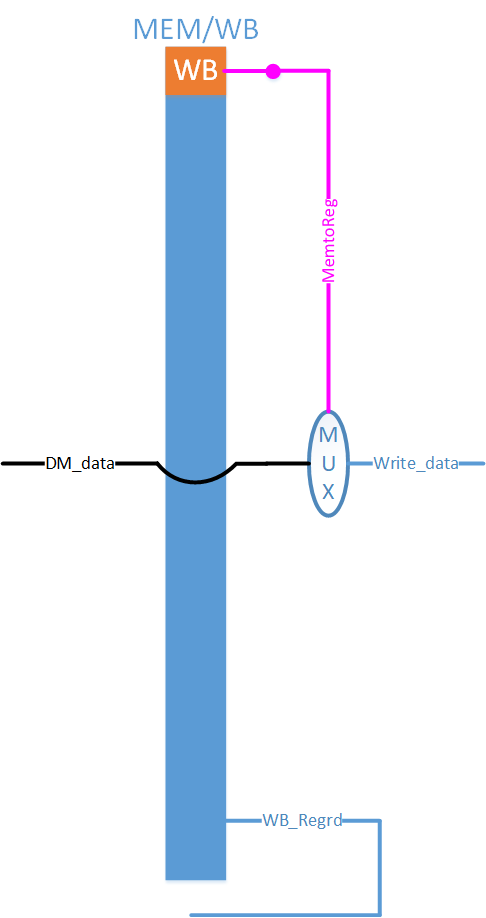
Mem Write 當指令為store時為1，其餘指令皆為0

ALU運算完的結果[15:2]，也就是除4當作記憶體的address輸入，其原理與在IM一樣，在ID stage時有詳細解釋，Memory byte select則是當指令為store類型時會需要判斷是SW、SH或SB，依照不同的指令會有不同的選擇方式寫入，load sign extend則是因為load有有號或無號擴充的區別，分別對其擴充。

設計時所遇到的困難之處:

在當初撰寫Memory byte select的function時，一直是認為用ALU運算的最低兩位元[1:0]選擇Data的byte做寫入，像是指令是SB，且最低兩位元是00就選擇Data的最低一個byte寫入(一筆Data有四個byte)，最低兩位元是10就選擇Data的第3個byte做寫入依此類推，不過最後在看波型debug時發現怎麼樣都對不上，明明要抓進DM的資料應該要是56000000，而我送進去的卻一直是12000000，而後思考可能出錯的原因，在仔細確認過RISC-V的spec後發現它是固定將資料的最低byte讀出(SB)，ALU運算的最低兩位元是用來判斷我要寫入的區塊，00寫入最低，01寫入第2個byte，依此類推，所以在我Memory byte select最後的設計中是將取出的byte或half word按照ALU最低兩位的case做補0後寫入，而不是按照原本整條寫入然後只用WEB控制，經此改動，寫入DM的Data與助教給的test pattern一致。

5.資料寫回階段(WB stage)



圖八、WB架構

功能描述:主要負責將結果寫回Register File中。

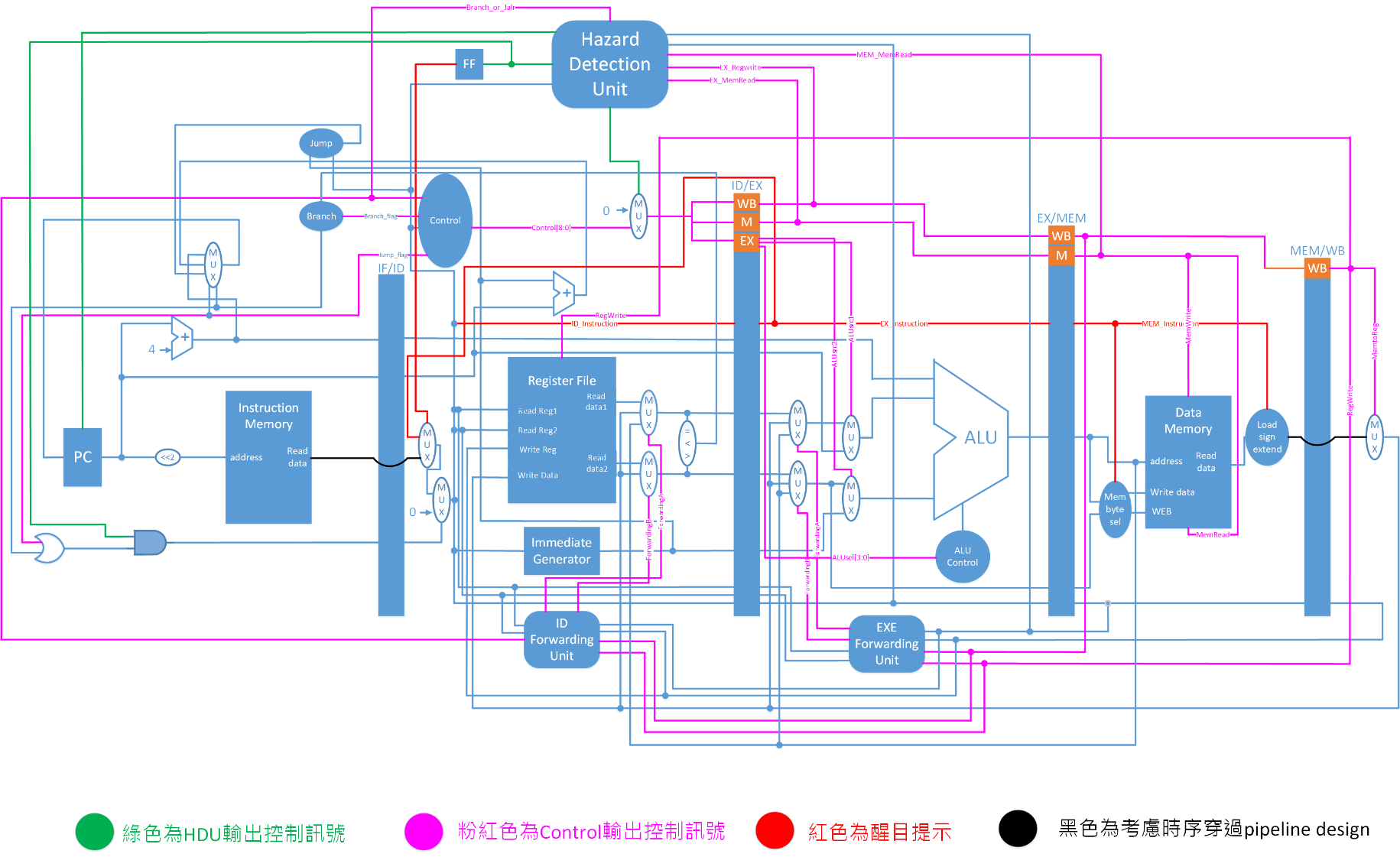
在圖八中的電路有: 多工器、MEM/WB管線暫存器。

控制訊號:

MemtoReg 為1的時候選擇ALU運算完的結果，0選擇來自記憶體讀出的資料。

RegWrite 為1的時候將資料寫回暫存器，0則不寫回。

6.CPU架構完成圖



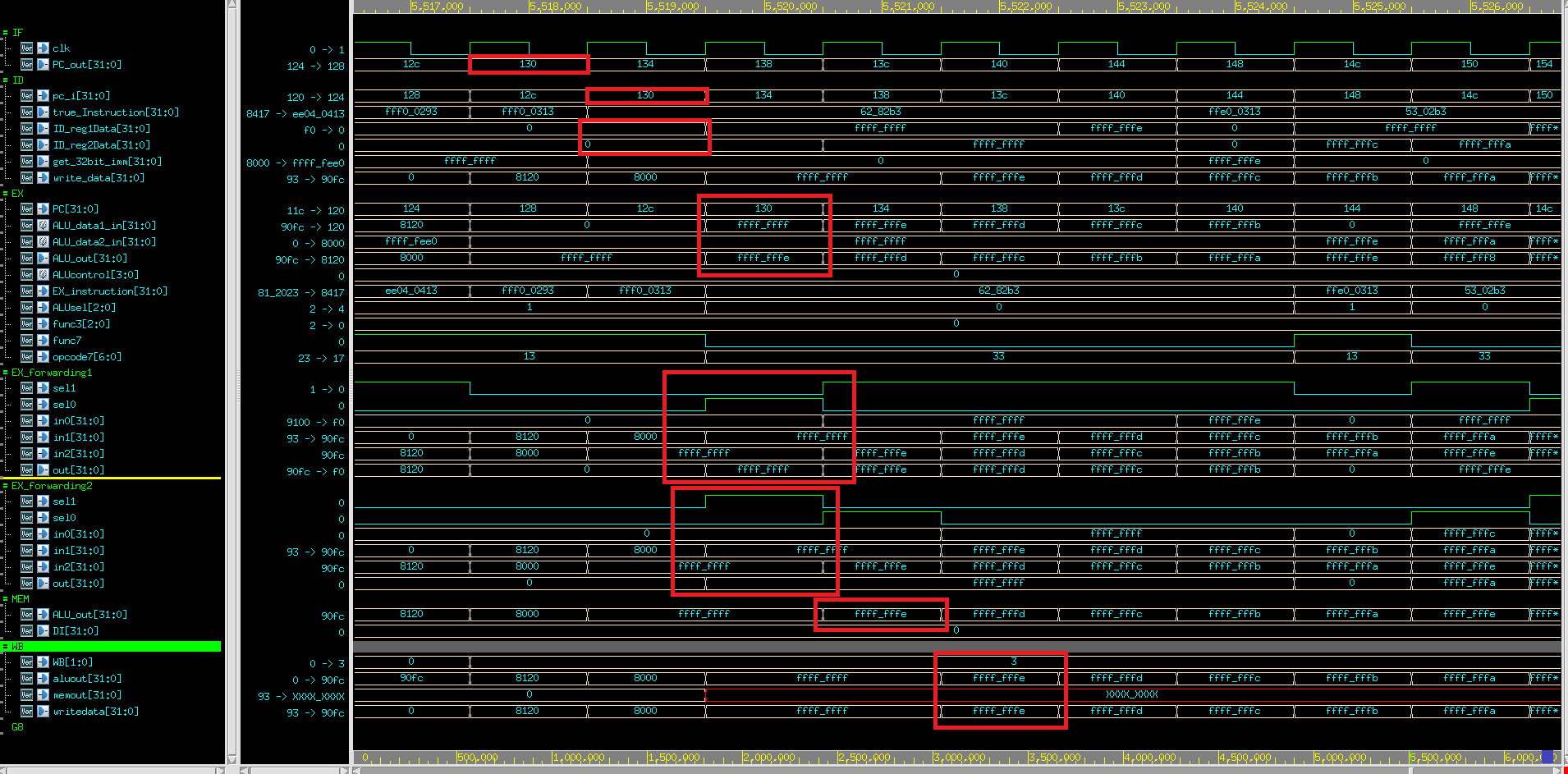
圖九、CPU整體架構

1. 波形驗證

prog0(simulation)

 此項作業要求CPU執行助教提供的test pattern，測試基本的37種指令是否能正常執行。以下為部分波形的截圖，並將說明CPU在不同類型的指令下每一級的運作狀況。

以下信號數值皆是以16進制方式表示



圖九、R-type ADD指令

紅色框框標記的地方即為一條Instruction完整的5 stage pipelined過程。

上圖波形說明

|  |  |
| --- | --- |
| stage | 說明 |
| IF | 抓到PC為130 |
| ID | 從IM讀出的指令為62\_82b3，經過解碼達到的rs1=0, rs2=0 |
| EX | 判斷出Forwarding回傳值，將ffff\_ffff與ffff\_ffff相加得到的ALU out=ffff\_fffe |
| MEM | 不經過MEM stage |
| WB | 讀取來自ALU out=ffff\_fffe寫回Register file |

****

圖十、R-type SRA指令

紅色框框標記的地方即為一條Instruction完整的5 stage pipelined過程。

|  |  |
| --- | --- |
| stage | 說明 |
| IF | 抓到PC為2ec |
| ID | 從IM讀出的指令為4062\_d2b3，經解碼得到的rs1=91a2, rs2=1234\_5678 |
| EX | 判斷出Forwarding回傳值，data1=8765\_4321算術右移4位等於f876\_5432 |
| MEM | 不經過MEM stage |
| WB | 讀取來自ALU out將計算完的結果寫回Register file |



圖十一、R-type OR指令

紅色框框標記的地方即為一條Instruction完整的5 stage pipelined過程。

|  |  |
| --- | --- |
| stage | 說明 |
| IF | 抓到PC為334 |
| ID | 從IM讀出的指令為62\_e2b3，經解碼得到的rs1=1234\_5678, rs2=7777\_7777 |
| EX | 判斷出Forwarding回傳值，data1=1234\_5678與data2=fedc\_ba98做or運算後得到fefc\_fef8 |
| MEM | 不經過MEM stage |
| WB | 讀取來自ALU out將計算完的結果寫回Register file |



圖十二、I-type ADDI指令

紅色框框標記的地方即為一條Instruction完整的5 stage pipelined過程。

|  |  |
| --- | --- |
| stage | 說明 |
| IF | 抓到PC為45c |
| ID | 從IM讀出的指令為7652\_8293，經解碼得到的rs1=cccc\_cccc, rs2= cccc\_cccc |
| EX | 判斷出Forwarding回傳值，data1=ffff\_ffffe與data2=765相加為763 |
| MEM | 不經過MEM stage |
| WB | 讀取來自ALU out將計算完的結果寫回Register file |



圖十三、I-type SLTIU指令

紅色框框標記的地方即為一條Instruction完整的5 stage pipelined過程。

|  |  |
| --- | --- |
| stage | 說明 |
| IF | 抓到PC為4f0 |
| ID | 從IM讀出的指令為fff2\_b293，經解碼得到的rs1=4, rs2=0 |
| EX | 判斷出Forwarding回傳值，data1=ffff\_ffffe與data2=ffff\_ffff做無號數比較，得到1(小於) |
| MEM | 不經過MEM stage |
| WB | 讀取來自ALU out將計算完的結果寫回Register file |



圖十四、I-type LW指令

紅色框框標記的地方即為一條Instruction完整的5 stage pipelined過程。

|  |  |
| --- | --- |
| stage | 說明 |
| IF | 抓到PC為3e0 |
| ID | 從IM讀出的指令為2\_a283，經解碼得到的rs1=90f4, rs2=0 |
| EX | 判斷出Forwarding回傳值，data1=90f8與data2=0做相加，得到90f8 |
| MEM | 取ALUout[15:2]從DM讀出90f4 |
| WB | 讀取來自MEM的值將計算完的結果寫回Register file |



圖十五、S-type SW指令

紅色框框標記的地方即為一條Instruction完整的5 stage pipelined過程。

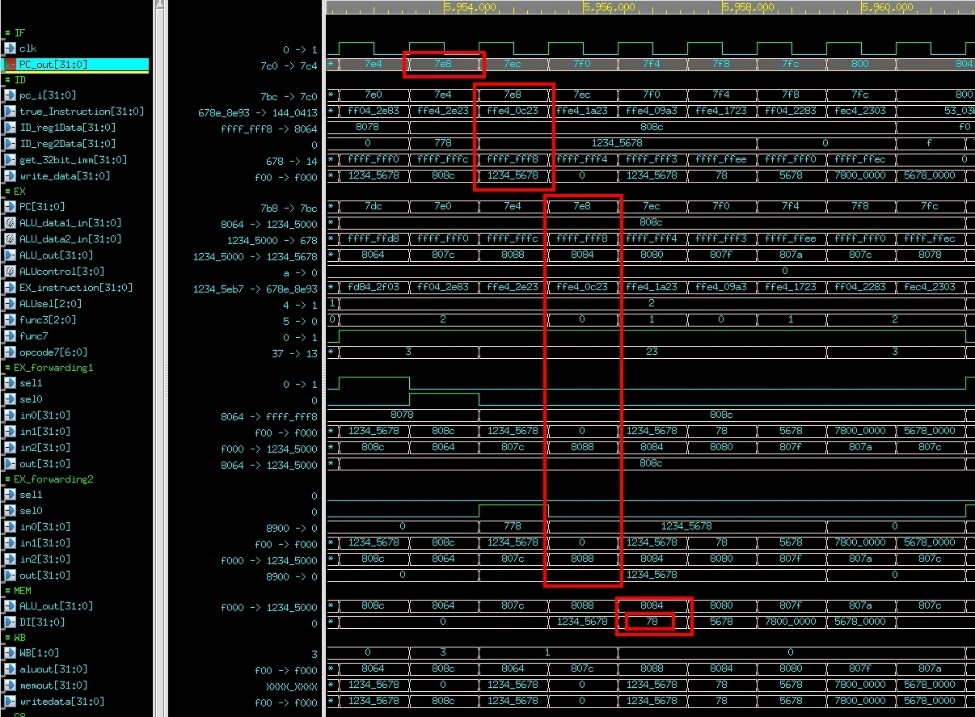
|  |  |
| --- | --- |
| stage | 說明 |
| IF | 抓到PC為7c4 |
| ID | 從IM讀出的指令為fe57\_2e23，經解碼得到的rs1=8064, rs2=f, imm=ffff\_fffc |
| EX | 判斷出Forwarding回傳值，data1=8078與立即值=ffff\_fffc做相加，得到8074 |
| MEM | 取ALU out[15:2]從DM寫入DI=f, 且WEB=1111(store a word) |
| WB | 不寫回Register file |



圖十六、S-type SH指令

紅色框框標記的地方即為一條Instruction完整的5 stage pipelined過程。

|  |  |
| --- | --- |
| stage | 說明 |
| IF | 抓到PC為7ec |
| ID | 從IM讀出的指令為ffe4\_1a23，經解碼得到的rs1=808c, rs2=1234\_5678 |
| EX | 將data1=808c與立即值=ffff\_fff4做相加，得到8080 |
| MEM | 取ALU out[15:2]從DM寫入DI=5678, 且WEB=1100(store a half word) |
| WB | 不寫回Register file |



圖十七、S-type SB指令

紅色框框標記的地方即為一條Instruction完整的5 stage pipelined過程。

|  |  |
| --- | --- |
| stage | 說明 |
| IF | 抓到PC為7e8 |
| ID | 從IM讀出的指令為ffe4\_0c23，經解碼得到的rs1=808c, rs2=1234\_5678 |
| EX | 將data1=808c與立即值=ffff\_fff8做相加，得到8084 |
| MEM | 取ALU out[15:2]從DM寫入DI=78, 且WEB=1110(store a byte) |
| WB | 不寫回Register file |



圖十八、B-type BEQ指令

紅色框框標記的地方即為一條Instruction完整的5 stage pipelined過程。

|  |  |
| --- | --- |
| stage | 說明 |
| IF | 抓到PC為81c |
| ID | 從IM讀出的指令為ff03\_0313，經解碼得到的rs1=ffff\_f000, rs2=1000, imm=38,Hazard Detection Unit判斷出需要stall, stall後ID\_reg2Data變為ff0,comparator=010(data1<data2),不滿足跳躍條件，PC=PC+4 |
| EX | 無(實際上沒有執行任何行為) |
| MEM | 無(實際上沒有執行任何行為) |
| WB | 無(實際上沒有執行任何行為) |



圖十九、B-type BNE指令

紅色框框標記的地方即為一條Instruction完整的5 stage pipelined過程。

|  |  |
| --- | --- |
| stage | 說明 |
| IF | 抓到PC為87c |
| ID | 從IM讀出的指令為ffff\_f337，經解碼得到的rs1=ffff\_f000, rs2=ff0 imm=38,Hazard Detection Unit判斷出需要stall,stall後ID\_reg2Data變為ffff\_f000, comparator=001(data1=data2),不滿足跳躍條件，PC=PC+4 |
| EX | 無(實際上沒有執行任何行為) |
| MEM | 無(實際上沒有執行任何行為) |
| WB | 無(實際上沒有執行任何行為) |



圖二十、B-type BLTU指令

紅色框框標記的地方即為一條Instruction完整的5 stage pipelined過程。

|  |  |
| --- | --- |
| stage | 說明 |
| IF | 抓到PC為9dc |
| ID | 從IM讀出的指令為262\_ea63，經解碼得到的rs1=ffff\_f000, rs2=ffff\_ff00 imm=34,Hazard Detection Unit判斷出需要stall, stall後ID\_reg2Data變為ffff\_effc ,comparator=110(data1>data2)unsign, 不滿足跳躍條件，PC=PC+4 |
| EX | 無(實際上沒有執行任何行為) |
| MEM | 無(實際上沒有執行任何行為) |
| WB | 無(實際上沒有執行任何行為) |



圖二十一、U-type AUIPC指令

紅色框框標記的地方即為一條Instruction完整的5 stage pipelined過程。

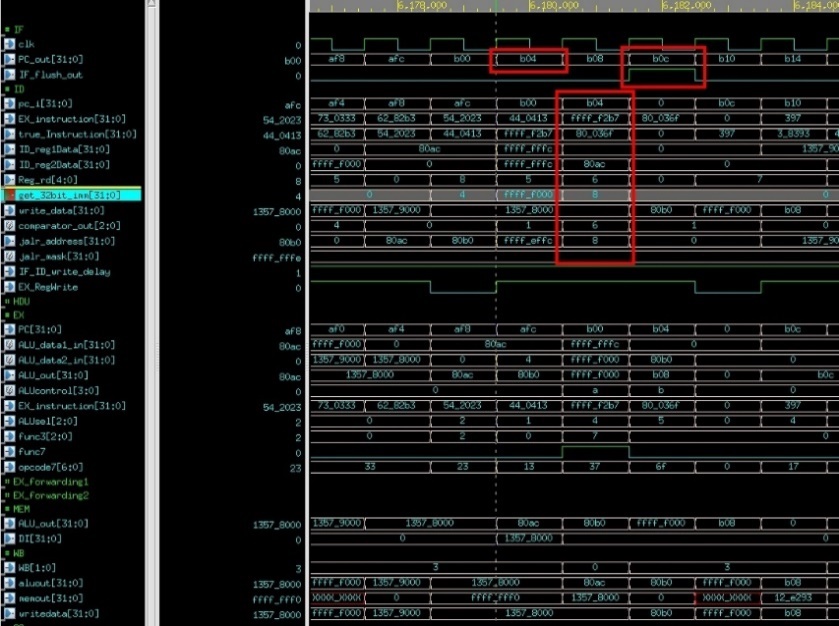
|  |  |
| --- | --- |
| stage | 說明 |
| IF | 抓到PC為acc |
| ID | 從IM讀出的指令為ffff\_f317，經解碼得到的imm=ffff\_f000 |
| EX | ALUdata1=acc, ALUdata2=ffff\_f000, rd = PC + imm = ffff\_facc |
| MEM | 不經過MEM stage |
| WB | 讀取來自ALU out將計算完的結果寫回Register file |



圖二十二、U-type LUI指令

紅色框框標記的地方即為一條Instruction完整的5 stage pipelined過程。

|  |  |
| --- | --- |
| stage | 說明 |
| IF | 抓到PC為ae8 |
| ID | 從IM讀出的指令為ffff\_f337，經解碼得到的imm=ffff\_f000 |
| EX | ALUdata2=ffff\_f000, rd = imm = ffff\_f000 |
| MEM | 不經過MEM stage |
| WB | 讀取來自ALU out將計算完的結果寫回Register file |

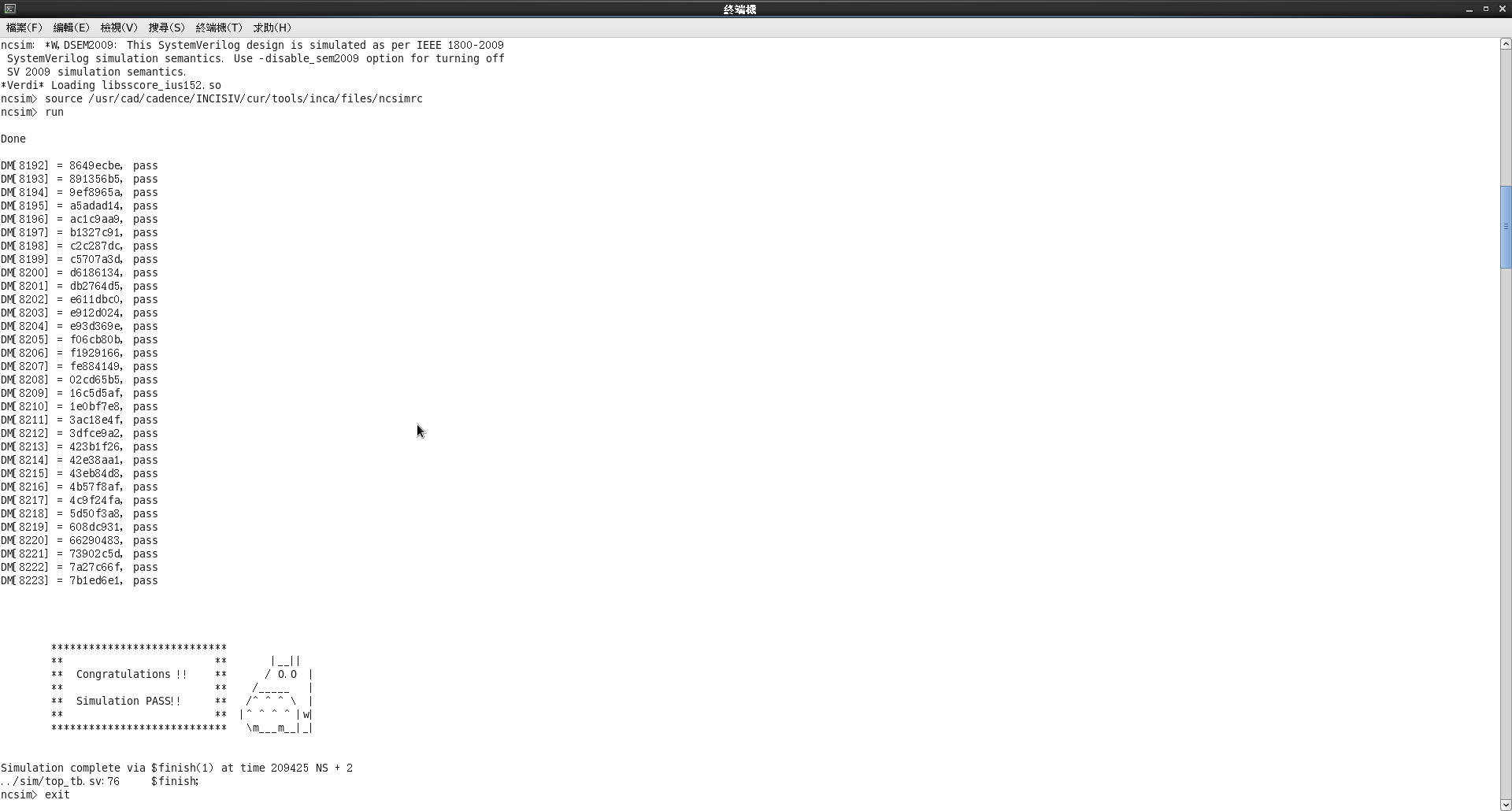


圖二十三、J-type JAL指令

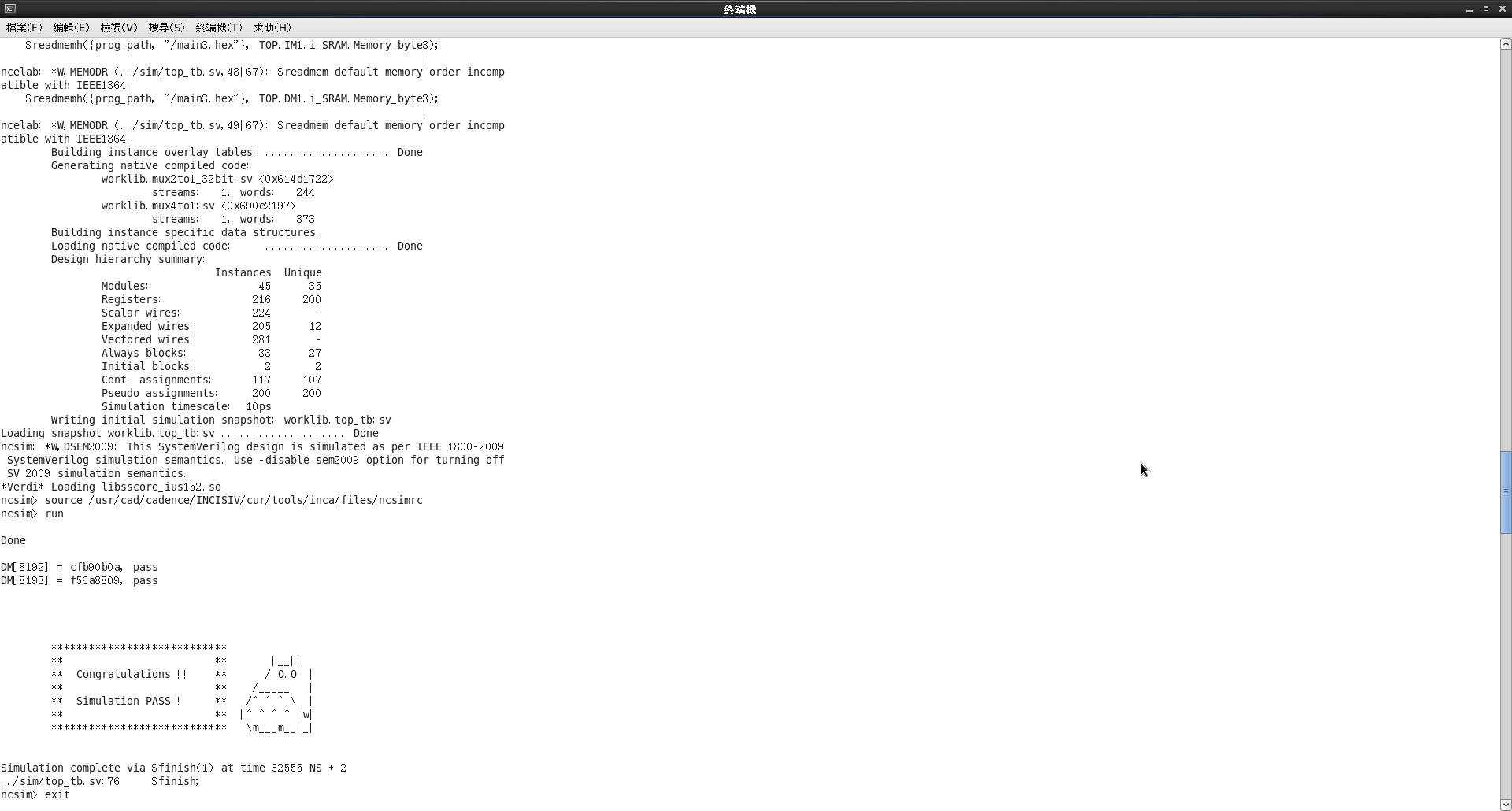
紅色框框標記的地方即為一條Instruction完整的5 stage pipelined過程。

|  |  |
| --- | --- |
| stage | 說明 |
| IF | 抓到PC為b04 |
| ID | 從IM讀出的指令為80\_036f，經解碼得到的imm=8,沖刷掉PC=b08後跳到PC=b0c |
| EX | 無(實際上沒有執行任何行為) |
| MEM | 無(實際上沒有執行任何行為) |
| WB | 無(實際上沒有執行任何行為) |

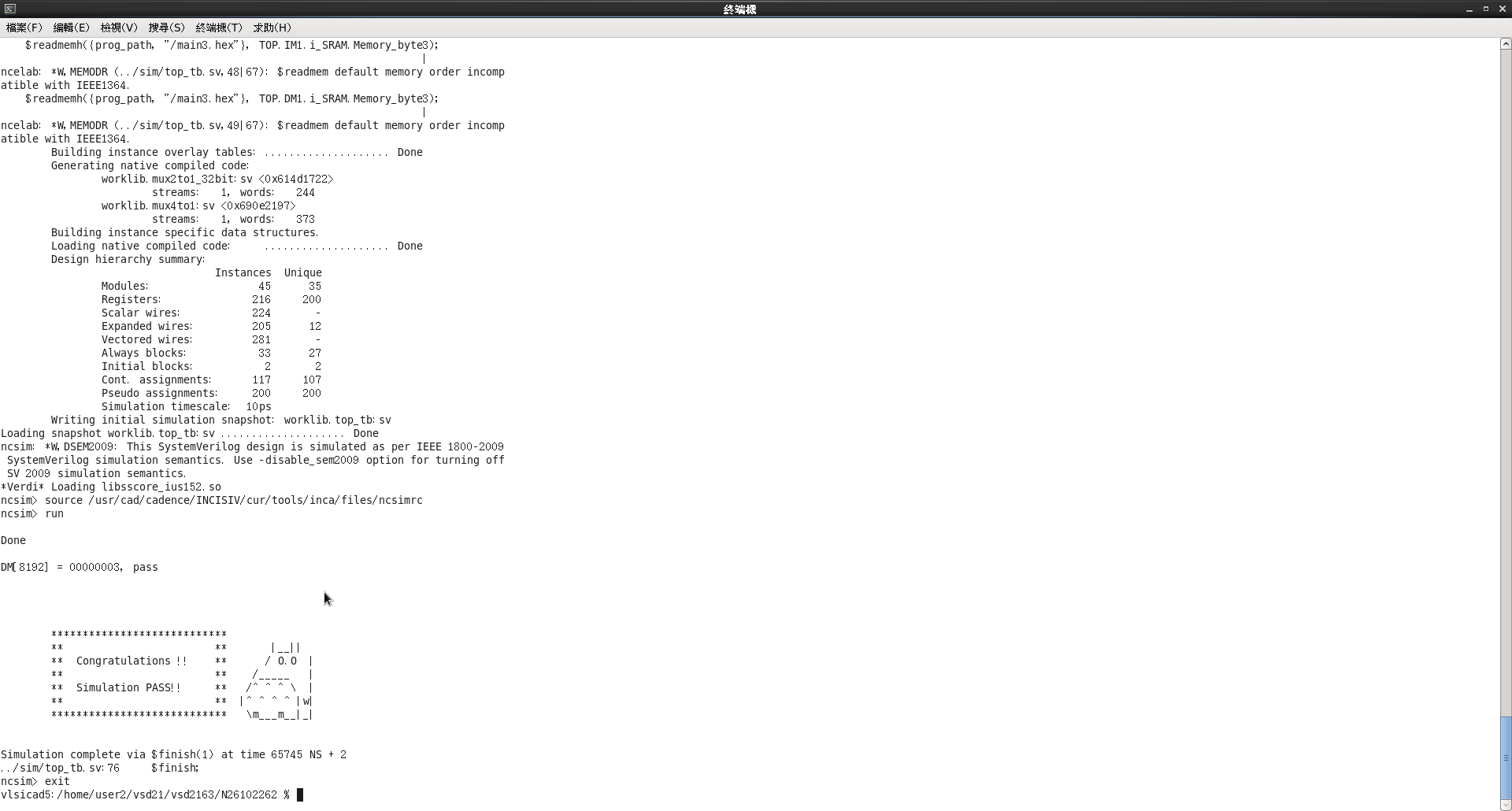
Prog1(simulation)



Prog2(simulation)



Prog3(simulation)



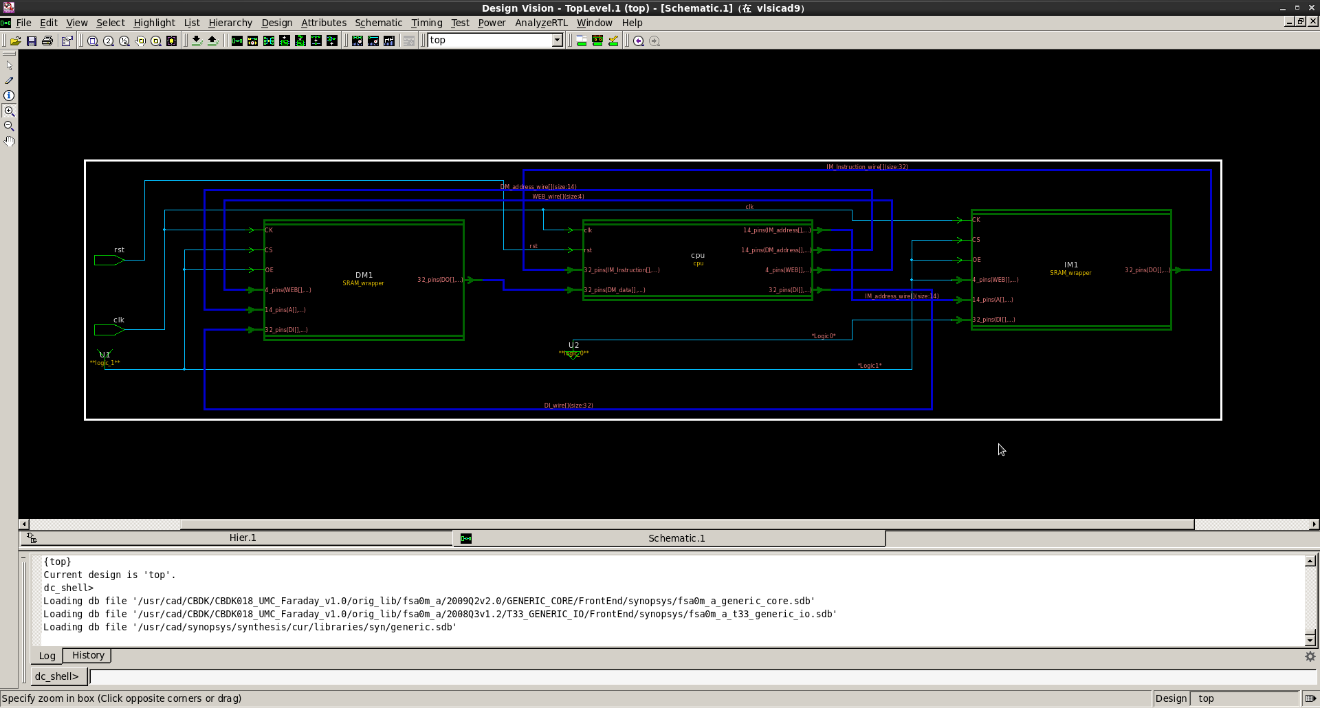
1. Lessons learned

這是我第一次寫CPU的RTL code，從一開始毫無方向不知道要做什麼很迷茫，到前中期的時後跟同學討論以及看看Github上面其他人的mips架構大概是怎麼樣去描述的，之後才慢慢開始有想法要怎麼寫，一開始先按照之前大學時計算機組織上的架構圖刻，接著在寫的過程中也常常遇到挫折有些東西不知道怎麼去用硬體的方式描述他，因為一般在寫軟體的思考方式不完全能套用到硬體上面做執行，與同學討論找學長請教後才比較了解到該怎麼解決，而我設計的架構是先從一個一個小的module開始，之後做完這些小的module後在將它包在我每一級的sv檔中，最後再將五個stage以及四個pipeline register一起放在cpu.sv中，最後按照作業格式拉出相對應腳位接給記憶體做輸出入，在一開始的時候光是nWave都還不太會用，熟悉tool怎麼用之後就是波型搭配prog0中的main.log and main.s開始debug，之後debug後才發現比較大的問題是還有時序的問題要考慮進去，一級與一級間資料的傳遞時間，怎麼讓組合邏輯與循序邏輯(管線暫存器、暫存器陣列、IM、DM)間到下一級的時間一致，都是當初與同學們討論之後試著實踐，想法錯了就改，錯了就改，直到最後時序上都沒有問題才終於放心，在此過程中雖然老實說蠻痛苦的，但最後看到全部都pass的時候心中的成就感真的爆棚，也碰到了很多在書上沒有去探討過原來實作上會遇到這些困難，該怎麼解決這些問題，都一再提醒著我下次不論是學習任何領域的新知，可以怎麼換個角度去思考、去學習，也許就能看得更多更廣。

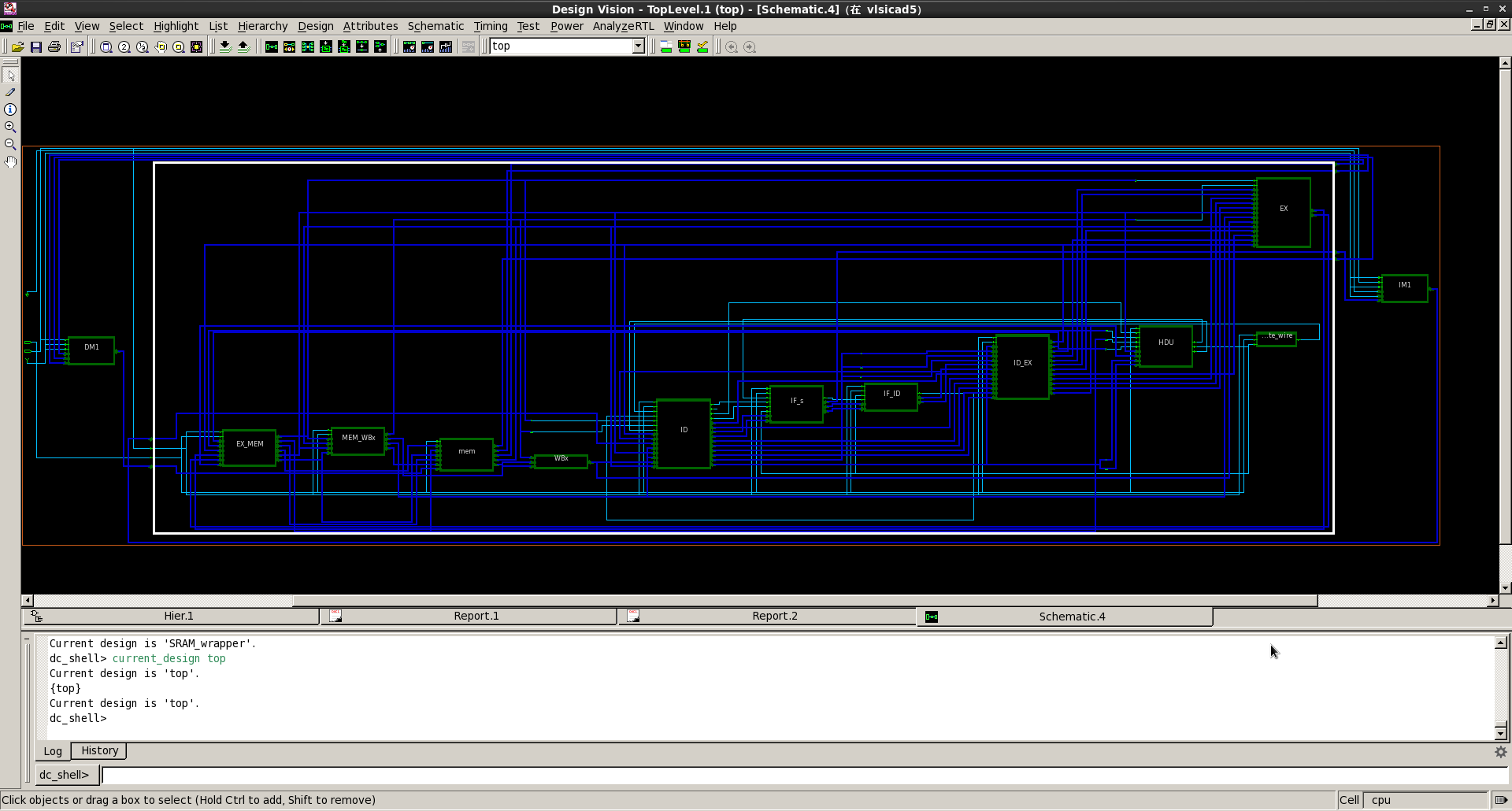
最後老實說這真的是我第一次寫這麼多verilog code，不得不說訓練相當紮實，怪不得聽學長們以及網路上都說這是一門可以學到很多收穫滿滿的課程，果然名不虛傳!

1. 附錄
2. Figures of the CPU

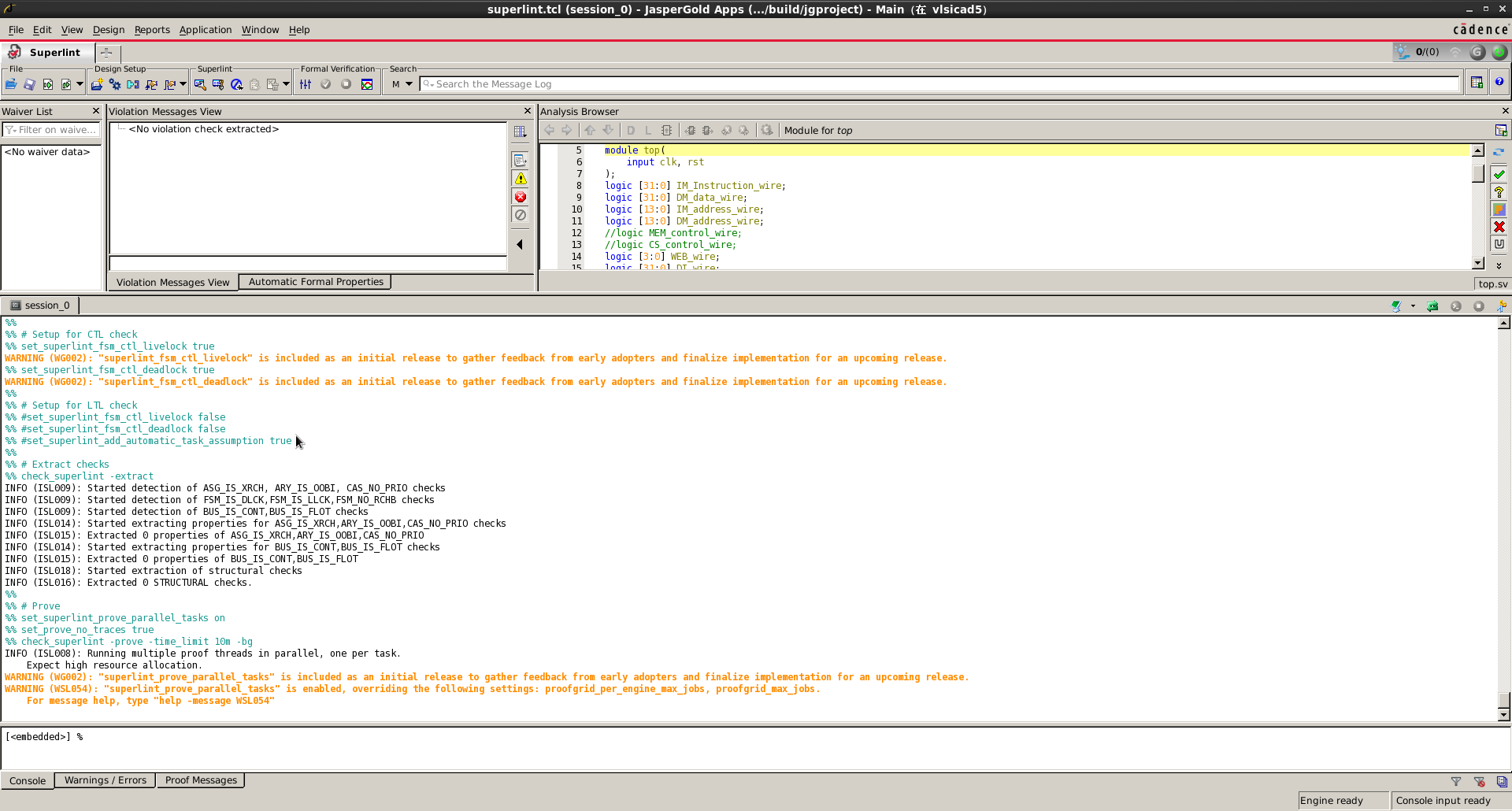
**1.TOP**

****

**2.CPU**

****

1. Superlint report



****

**Superlint within 0/2194.**

**Superlint常見語法錯誤:**

1.CodeStyle, Tag:REG\_NO\_USED

**"Local register variable 'LH\_tmp00' is unused (neither read nor assigned) in module 'LOAD\_signextend'"**

Solve:當初宣告的logic最後沒有使用到，就應該刪掉或是註解掉。

2.CodeStyle, Tag: REG\_NO\_READ

**"Local register variable 'y[63:32]' is not read, but is assigned at least once in module 'ALU'"**

Solve: 當初宣告的logic有一部份的bit數沒有用到，下次可以直接宣告欲使用的bit數就好，不用多宣告。

3.CodeStyle, Tag:ASG\_MS\_RPAD

**Unequal length operands in assignment in design-unit/block 'ALU'. Length of RHS is less than LHS:…**

Solve:等式的左邊(LHS)的bit數與右邊(RHS)的不相等，當初在撰寫時有時候明明要宣告成32bit的logic但忘記打bit數，此時又將做完運算的數值assign給宣告成1bit或是錯誤的bit數時產生的錯誤。

還有一個好像是在自己實驗室跑Superlint才會有的錯誤是

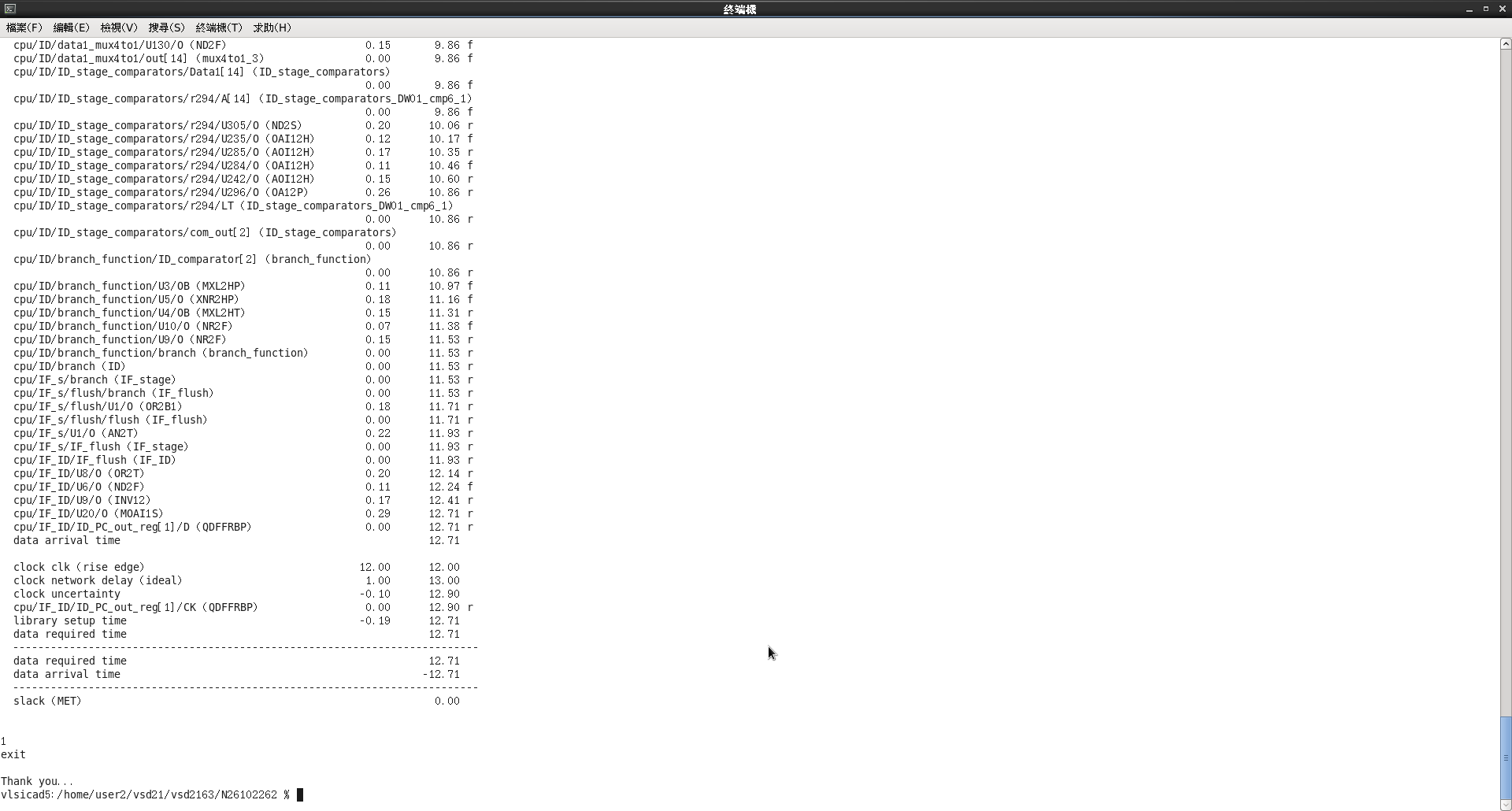
Category:FILEFORMAT, Tag: FIL\_NR\_CLTC

**"RTL source line contains control characters"**

它會去抓換行或者是為了對齊的Tab，但是在soc lab這一問題並不會發生，但在自己實驗室這個會抓出2000多個錯誤…

1. Synthetic report
2. **Performance:**

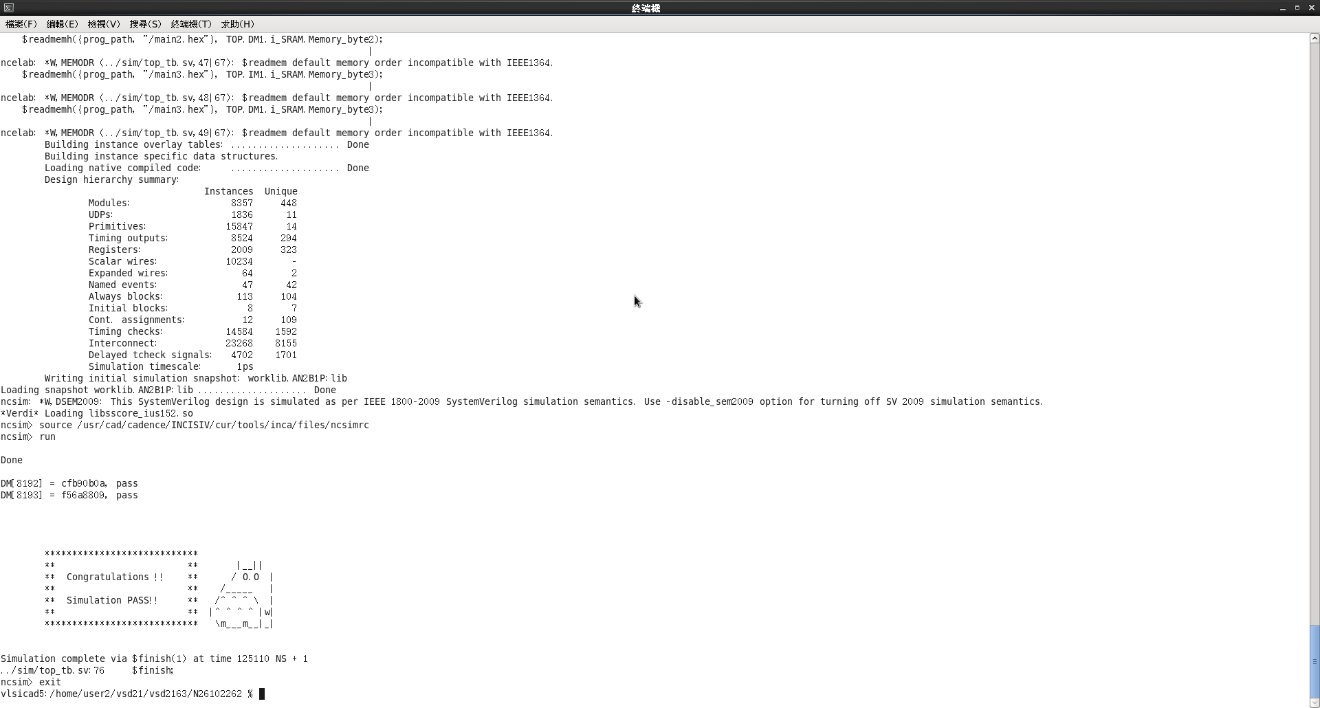
clk -period 12下slack為0 (make synthesize)

****

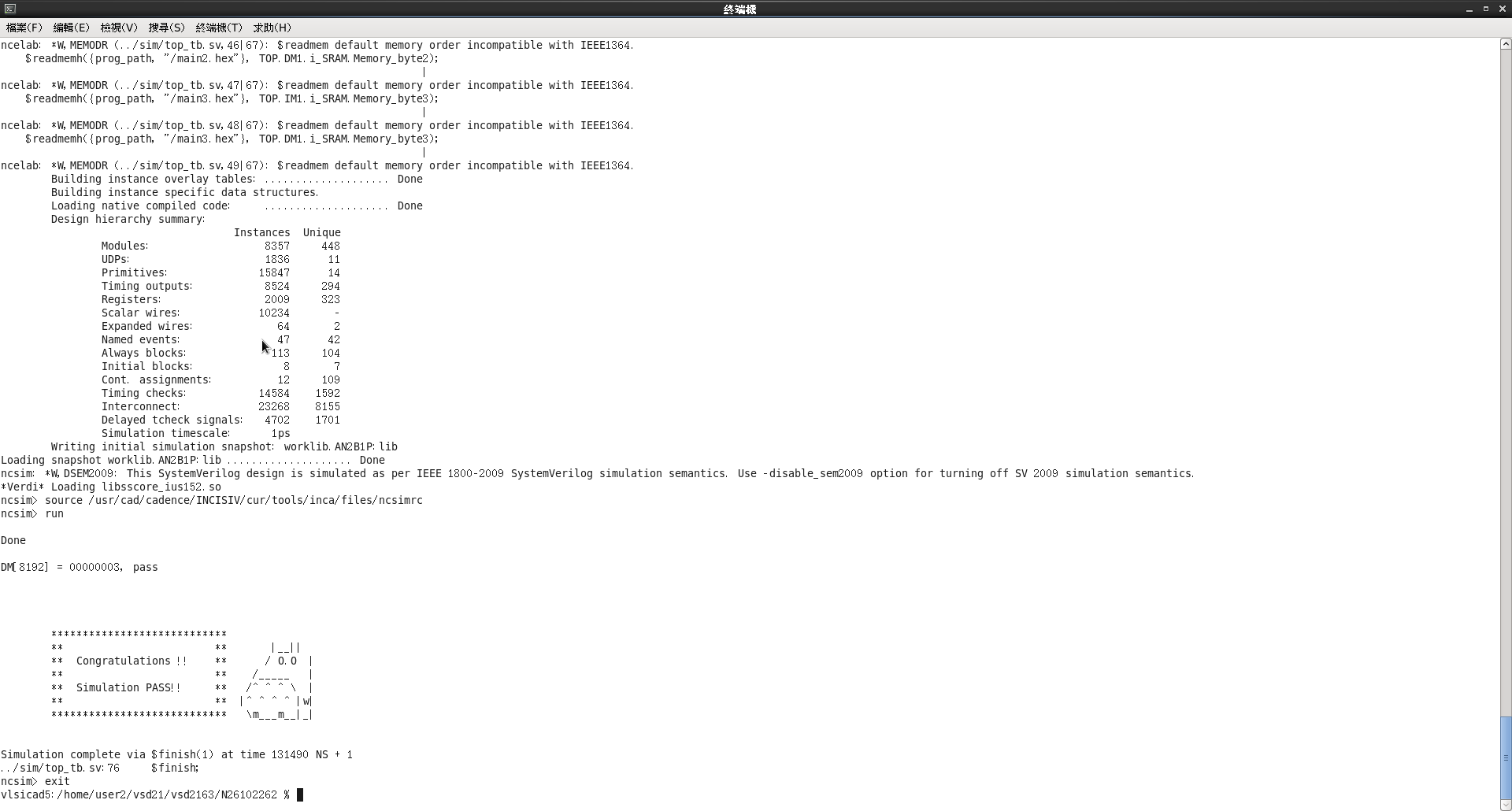
**Simulation time after actual synthesis(make synX)**

**1.syn0 2.syn1**

****

**3.syn2**

**4.syn3**

****

t0=74382 ns +

t1=418850 ns +

t2=125110 ns +

t3=131490 ns = 749832ns

Reported Area = 面積數據(Total Cell Area)為5,583,829.182656 µm²

Simulation Time (t0+t1+t2+t3) \* Reported Area =

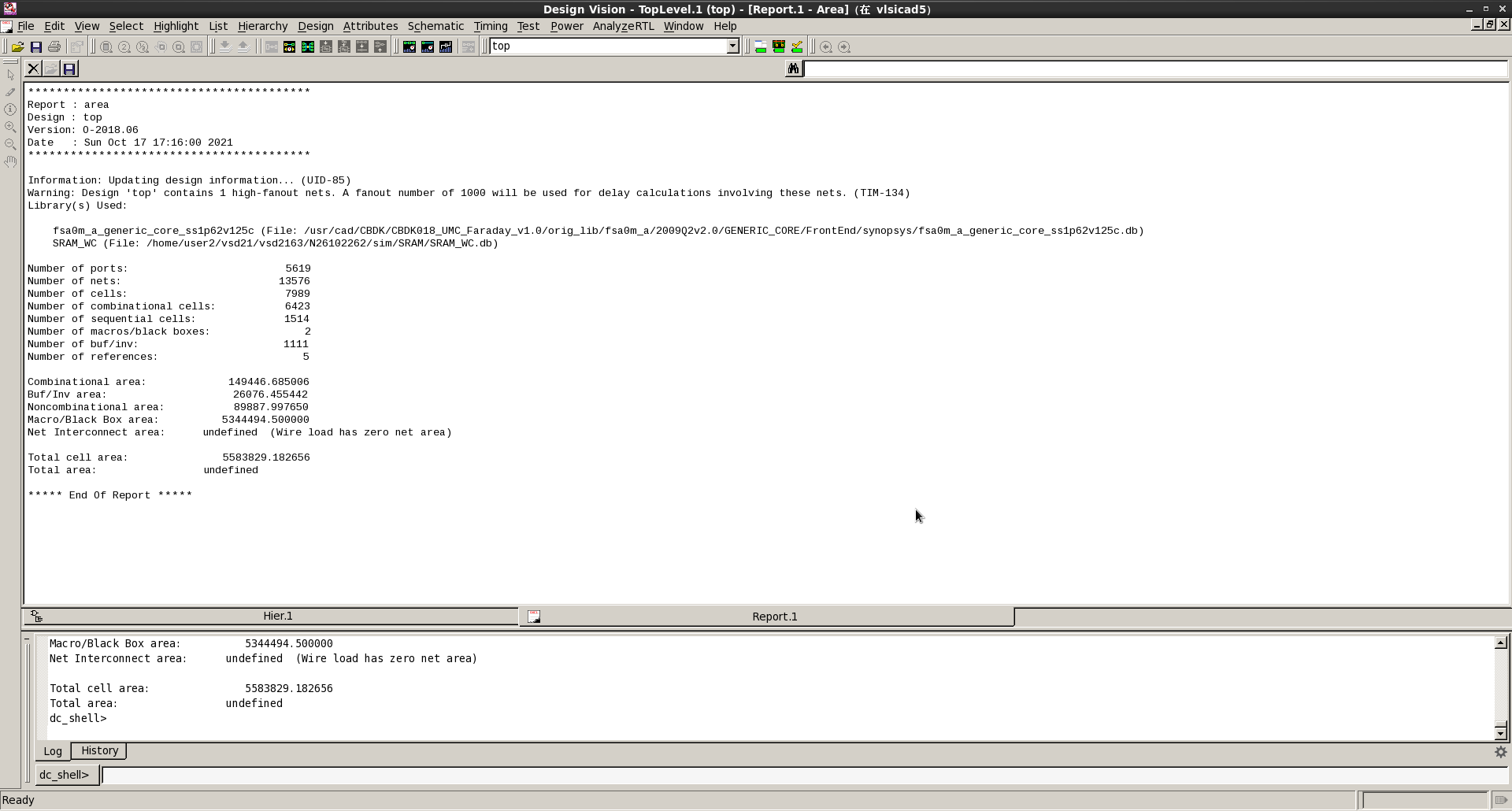
749832 ns \* 5,583,829.182656 µm²

1. Area

面積數據(Total Cell Area)為5,583,829.182656 µm²

扣掉助教所給的SRAM是5,344,494.5µm²

最後為 239,334.682656 µm²(無計算繞線)

****

1. Suppoted commands

|  |  |  |
| --- | --- | --- |
| Command | Support | Type |
| ADD | **〇** | R-type |
| SUB | **〇** |
| SLL | **〇** |
| SLT | **〇** |
| SLTU | **〇** |
| XOR | **〇** |
| SRL | **〇** |
| SRA | **〇** |
| OR | **〇** |
| AND | **〇** |
| LW | **〇** | I-type |
| ADDI | **〇** |
| SLTI | **〇** |
| SLTIU | **〇** |
| XORI | **〇** |
| ORI | **〇** |
| ANDI | **〇** |
| LB | **〇** |
| SLLI | **〇** |
| SRLI | **〇** |
| SRAI | **〇** |
| JALR | **〇** |
| LH | **〇** |
| LHU | **〇** |
| LBU | **〇** |
| SW | **〇** | S-type |
| SB | **〇** |
| SH | **〇** |
| BEQ | **〇** | B-type |
| BNE | **〇** |
| BLT | **〇** |
| BGE | **〇** |
| BLEU | **〇** |
| BGEU | **〇** |
| AUIPC | **〇** | U-type |
| LUI | **〇** |
| JAL | **〇** | J-type |