VLSI System Design (Graduate Level)

Fall 2020

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: \_高士鈞\_

Student ID: \_N26094922\_

1. **Problem briefing**

A cpu with:

a. The RISC-V ISA with the specified 33 instructions.

b. The number of pipeline stage is 5.

c. Register file size: 32×32-bit.

d. Program counter with 32-bit.

e. Mechanism to solve data hazard, control hazard and structural hazard.

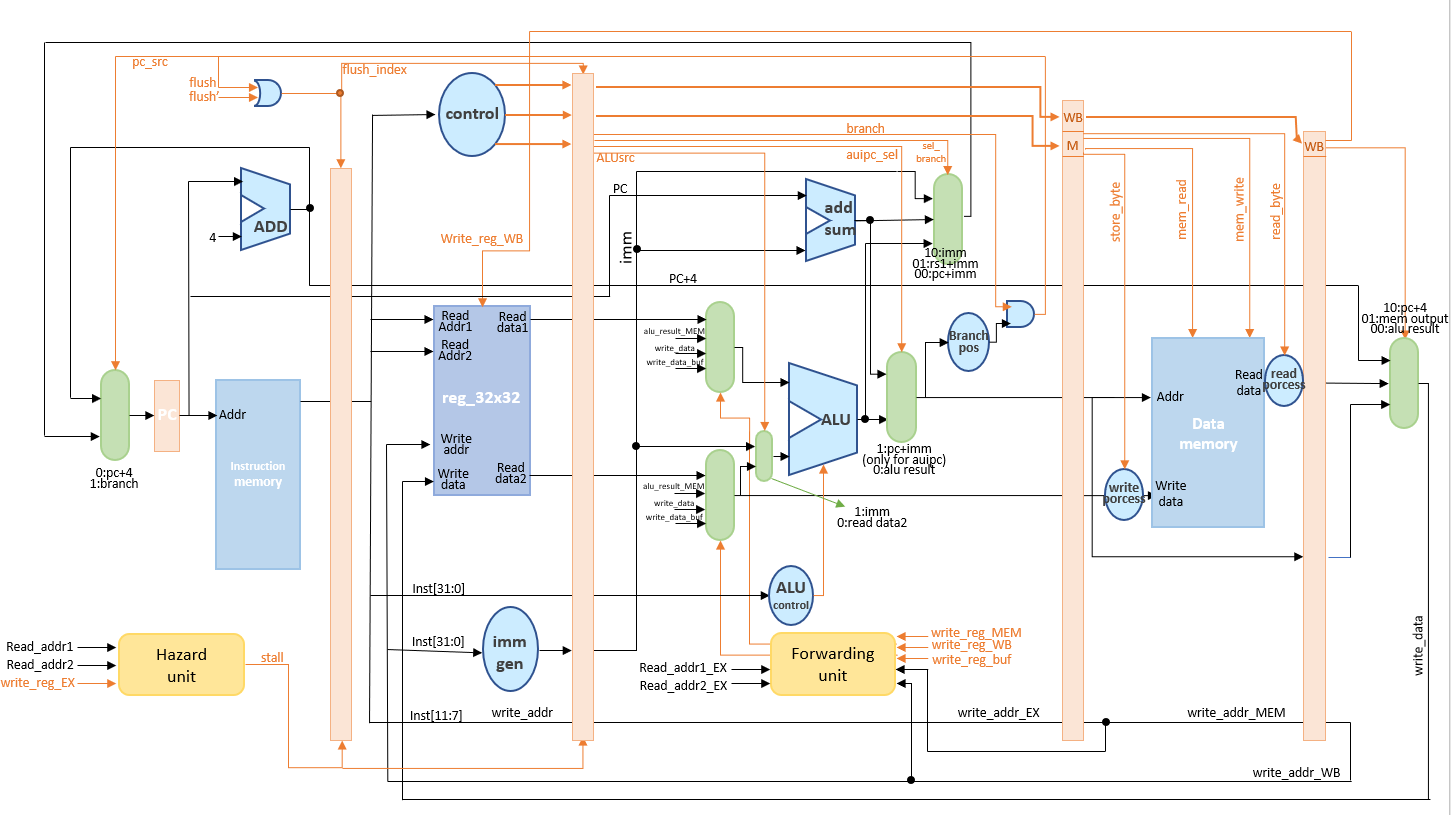
You also need to use two memories outside the CPU with specified size:

a. Instruction memory size: 64KB.

b. Data memory size: 64KB.

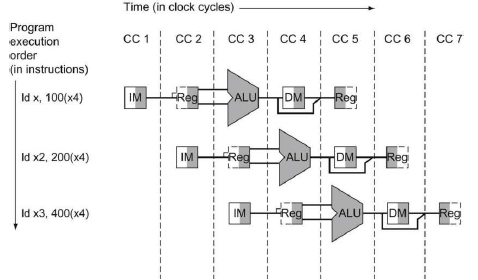
1. **Check list**

|  |  |  |  |
| --- | --- | --- | --- |
| Rtl code | | synthesis | |
| prog0 | v | prog0 | v |
| prog1 | v | prog1 | v |
| prog2 | v | prog2 | v |
| prog3 | v | prog3 | v |
| Post simulation | x | Additional instructions | x |
| superlint | 99% | | |

1. **Block diagram**

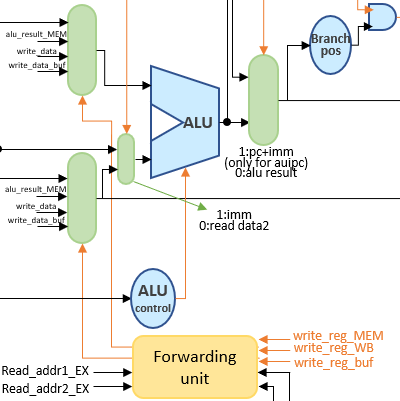
**(黑色線 > 資料傳輸線 橘色線 > 控制訊號線)**

本次的cpu根據典型的五級pipeline cpu架構做延伸，所以在此會特別提出一般架構所不會提到的細節，以及在設計上曾經遇到困難，有特別設計的module。

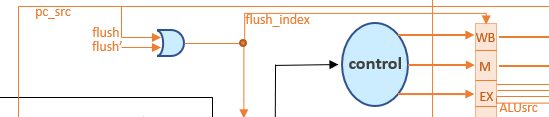
五級pipeline是把cpu的運作流程用flip-flop切成五個階段，分別是IF ID EX MEM WB，這樣的設計可以讓CPU的through put有效的提升，因為後面的指令不需要等前面的指令完成就可以先進到cpu做運算，如右附圖:

1. **Fowarding unit**

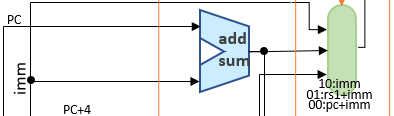
Pipeline雖然可以有效增加cpu效率，但是卻有可能發生資料相依的問題:後者指另要求的資料可能前者指令還沒完成，這時需要設計forwarding unit來將前者資料提早往前拉到ALU運算。



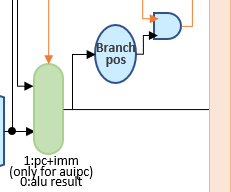
我本次設計的forwarding unit會在幾個狀況下觸發，當前一個指令的值有要寫入reg，且寫入的位置恰等於後一級指令要提取值的位置，則觸發forwarding訊號，把後一級指令需要的值往前拿到ALU運算，特別注意的因為reg本身也是一個sequential circuit，為了避免寫值未完成後面的指令就來提取，所以而外寫了一個buffer來存正在寫入的值以及地址，故本設計共有三條forwarding路徑。

1. **Flush**

RISC-V指令級含有幾個跳躍指令，例如Btype，這些指令往往都要在EX階段處理之後才能決定PC是否要跳躍，但是下一個指令可能在判斷之前就已經被讀進來了，所以我們需要在跳躍發生時對前一級讀進來的錯誤資訊做清除(flush)，在我的設計中，PC在選擇來源時用到PCsrc指令恰好和跳躍發生與否呈正相關，所以我用此訊號來當作判斷flush運作的flag，同時為了確保資料清除正確，對flush訊號多經過一級pipeline成為flush`，flush兩次確保資料流正常。

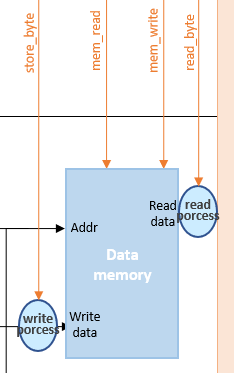
1. **Add sum**

這一階段主要是在處理和PC相關的加法，但除此之外我也把這裡當作是要饋入PC的跳躍位置選擇器，所以在這裡我會根據跳躍指令的類型，以sel\_branch指令選擇出正確的跳躍目的地，理論上add sum和ALU的路徑是獨立的，但在auipc指令時會需要將pc+imm往下傳到寫入reg的路徑，所以才會在ALU之後看到一個多餘的MUX。

1. **Branch pos**

這裡有一個branch pos module，是特別設計給Btype指令的，因為branch指令裡有一些答案需要做inverse才是正確的(例如BEQ和BNEQ，因為他們都是用ALU裡的XOR做運算)，所以這裡放入這個module在需要的時候對result inverse。算出的答案對所有bit 做or後，會和branch signal做and，成為PCsrc訊號。

1. **Load store byte**

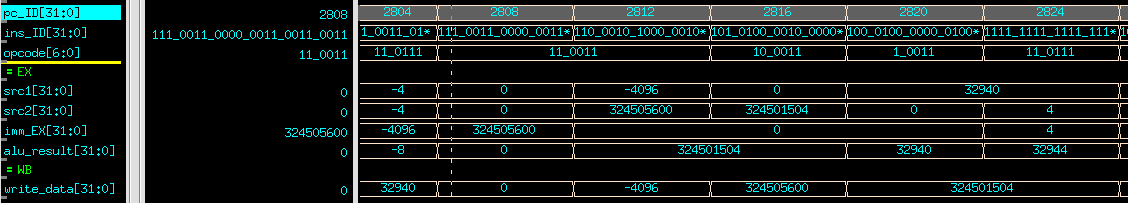
本次作業有SB LB兩個指令，所以我設計兩個模塊做處理，write process部分是拿write addr的LSB做判斷要存入第幾個byte，並改寫write data和DI指令，若是SW則不處理。

LB則比較簡單，當LW時，直接將[31:8]清0，若LW則一樣不處理。

1. **Test programs**

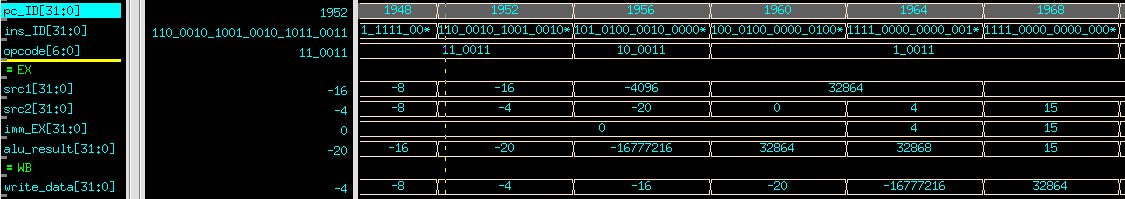
* **Instructions**

為方便講解，這裡取用的PC和instruction都是拿ID階段的訊號，而cursor標注的時間點是指令發生的時間，所以所有運算都會發生在之後的一個cycle。(指令實際時間其實會更早一個cycle，但方便解釋我一律把ID Stage當成first stage)

* 1. **ADD (Rtype)**

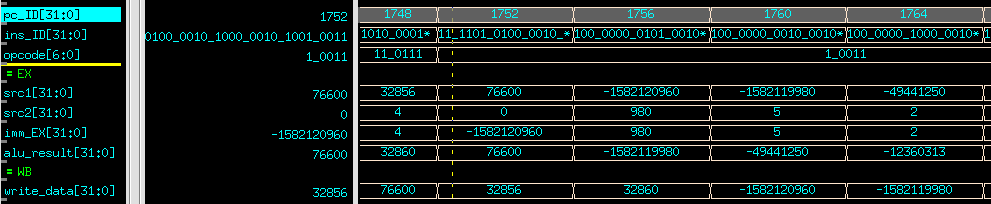
ADD

可以看到在PC=2808時，指令為ADD，所以在下個時間(PC=2812)時，rs1和rs2會進入ALU，並且可以得到相加的答案，並且在兩個cycle之後可以看到該答案被寫入register。

* 1. **SLL (Rtype)**

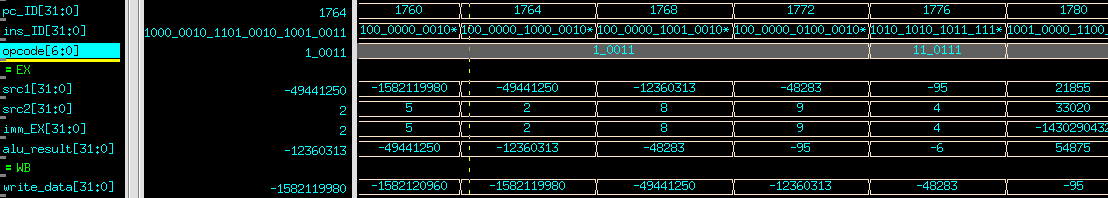
SLL

這裡的SLL指令和ADD類似，只是這裡的ALU行為不一樣了，所以可以看到兩筆資料也都是拿register讀出來的值，然後在ALU做unsigned shift，得到的值在兩個cycle後被寫入reg。

* 1. **ADDI (Itype)**

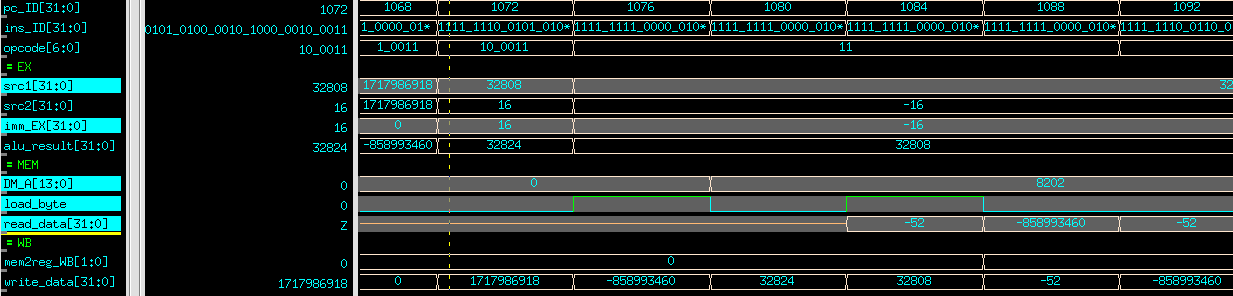
ADDI

Itype指令和Rtype最大差別在運算資料的來源，Itype主要是拿register讀出來的一號資料和imm做運算，所以可以看到這裡的運算對像改成了src1和imm\_EX，其儲存步驟也和Rtype一樣。

* 1. **SRAI (Itype)**

SRAI

這裡要介紹在Itype指令中比較特殊的一個，SRAI，這個指令為位移指令，但要注意其位移量只有imm的LSB五個bits，所以要先對imm處理，以免位移溢位，途中的imm\_EX就是經過處理的值，可以看到位移8是在合理範圍內，位移完後的值也會照程序存回register。

* 1. **LW LB (Itype)**

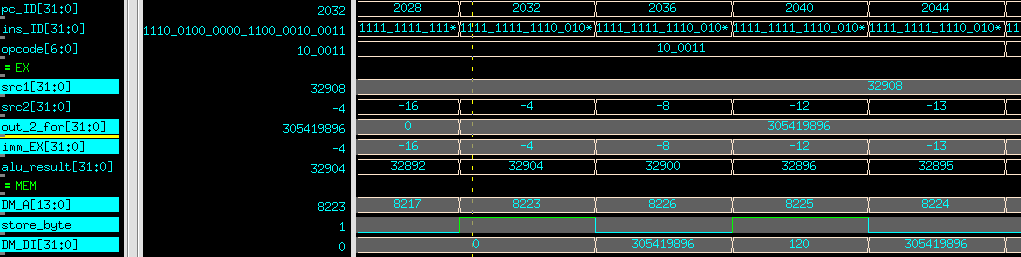
LW

LB

這裡要介紹從data memory讀取資料的LW LB指令。

紅色標注部分是屬於LB，橢圓形地方是從controller發出的load\_byte指令，告訴cpu這裡拿出的data只會用到LSByte，所以待會要把取出的data做後處理(會使用到我設計的read process module)，然後會傳遞ALU計算出的地址給data memory，然後把讀出的資料寫入register(和前面幾個指令寫入方式相同)。

黃色部分是LW，在這個指令下並不會對data memory給出的資料做處理，其餘部分皆和LW相同。

* 1. **SW SB (Stype)**

SW

SB

SW指令用到的會先計算預計存入資料的地址，在這裡的地址和LW一樣是取imm+rs1所以一樣會從ALU取資料，得到地址後，會再把register讀到的另筆資料寫入該地址，也就是虛線標示的路徑，這裡register2的資料剛好碰到forwarding，所以是取out\_2\_for的資料，而這裡一樣有一個store\_byte的資料來控制儲存資料是word or byte，如果是word則不對資料做處理，如果是byte，則會根據address的LSB兩個bits來判斷存在哪個位置。

**LSB**

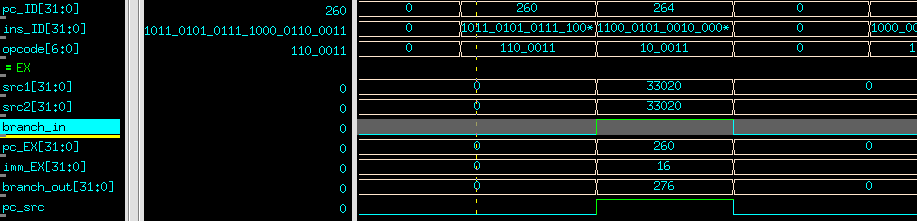
**00**

**01**

**10**

**11**

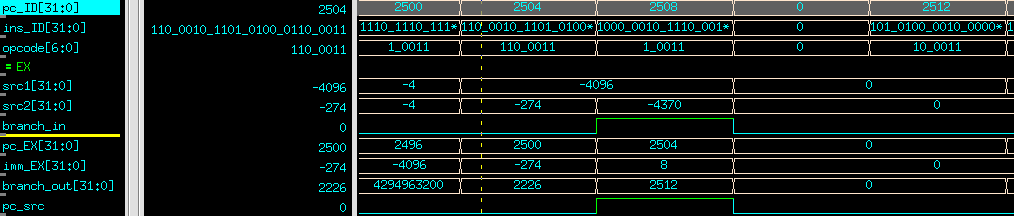
|  |  |  |  |
| --- | --- | --- | --- |
| **24’b0** | | | **reg2[7:0]** |
| **16’b0** | | **reg2[7:0]** | **8’b0** |
| **8’b0** | **reg2[7:0]** | **16’b0** | |
| **reg2[7:0]** | **24’b0** | | |

* 1. **BGEU (Btype)**

BGEU

PC+imm->

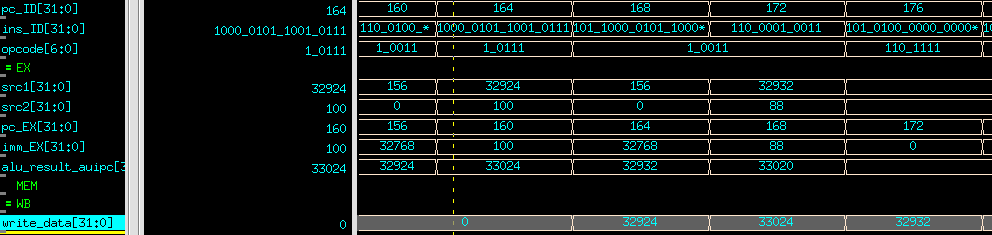
BGEU指令為條件式跳躍的一種，當滿足rs1 ≥ rs2時則跳躍到PC+imm，但實際在ALU運算時是計算rs1 < rs2後再把答案反轉，以此節省電路空間；在這裡可以看到src1恰等於src2，所以觸發跳躍，而跳躍目標正是PC+imm，在右上角也可以看到PC也確實跳躍到我們要求的位置了，而Btype其他幾個指令也都是累死的觸發步驟，只是條件不同而已。

* 1. **BGE (Btype)**

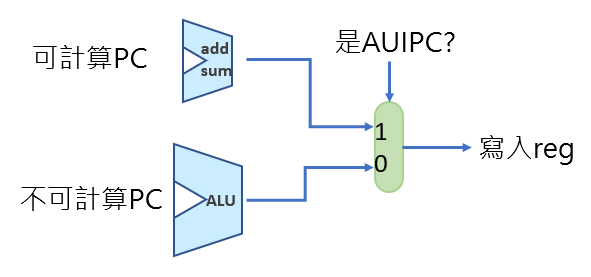
PC+imm->

BGEU

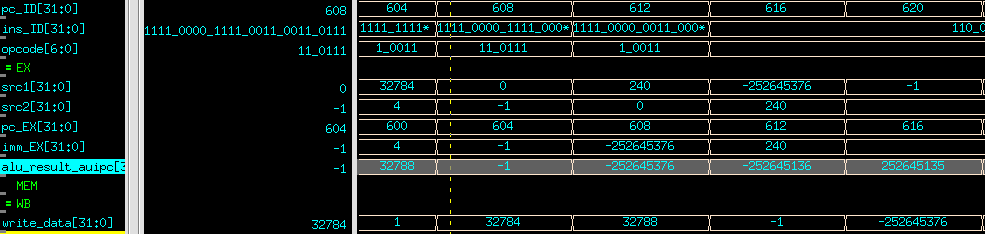
這個指令和前者差不多，差別在這指令比較時是採signed interger，所以在部分情況上結果會有所不同，在這裡同樣滿足條件而觸發了跳躍。

* 1. **AUIPC (Utype)**

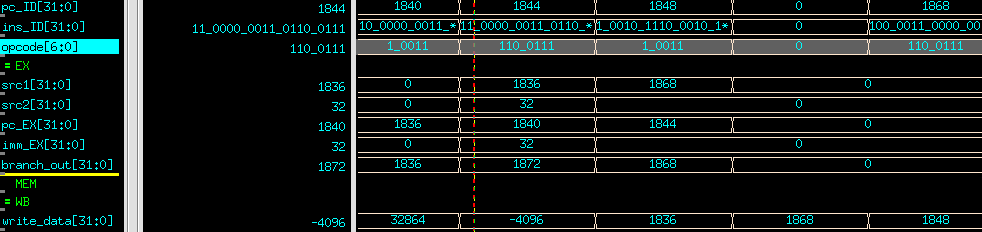
AUIPC

AIIPC是把PC+imm運算結果儲存到register的指令，特別的是我設計架構中，回存reg的路徑上是沒有PC訊號做運算的，所以需要透過add sum module做運算再透過mux把資料輸出給register，也就是波形圖裡的alu\_result\_auipc訊號，這是資料計算比較特別的地方，至於資料回存也就和其他指令大同小異。

* 1. **LUI (Utype)**

LUI的運作流程和AUIPC大同小異，就只是運算完成的值有沒有加上PC的差異，所以不多贅述流程。

LUI

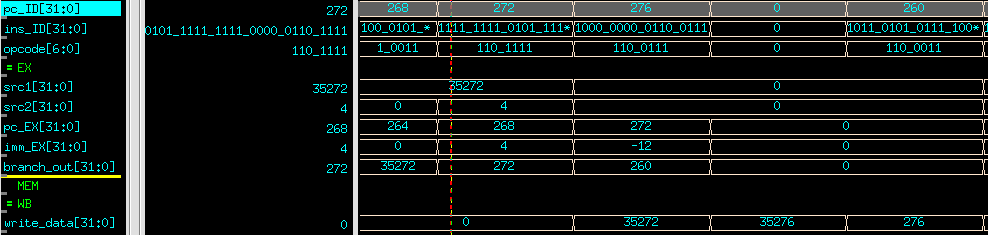
* 1. **JALR (Itype)**

PC+4

JALR

JALR指令為非條件跳躍的一種，PC要跳躍到imm+rs1，所以要先在ALU計算出跳躍目標，在紅框處可以看到目標被正確計算，且在兩個CYCLE之後PC也正確的跳躍；另外寫回部分要寫入PC+4，而在此的PC是指令給與當下的PC，所以在這裡是取1844，非當級往前看的新指令的PC，故寫入值為1848。

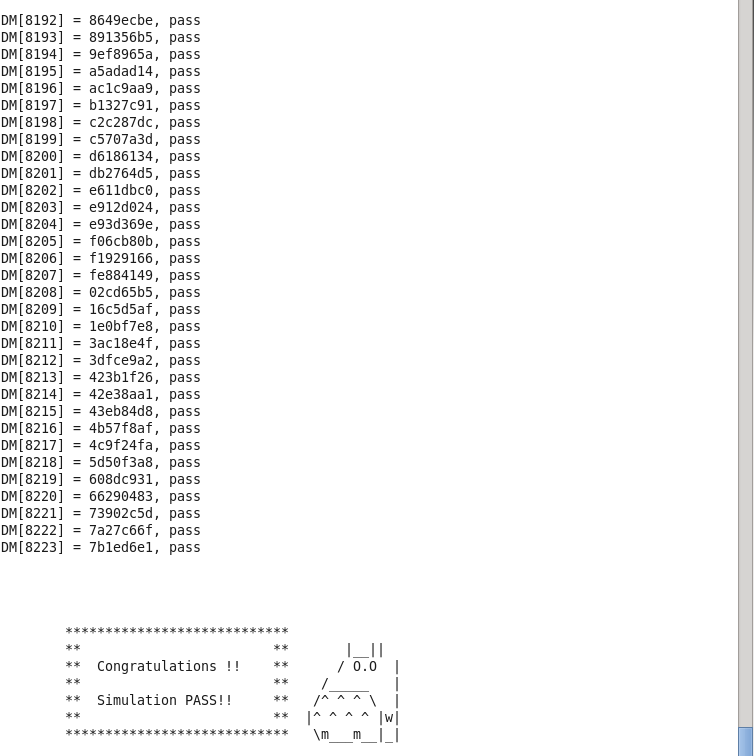
* 1. **JAL (Jtype)**

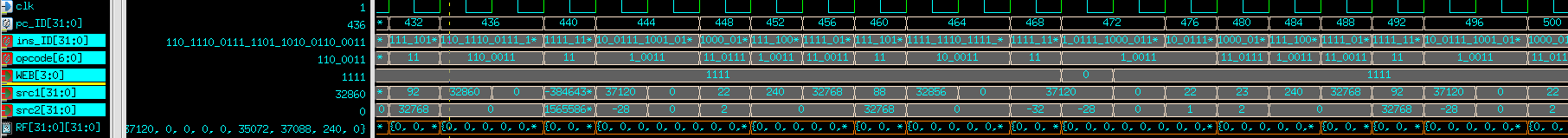


JAL

PC+4

JAL指令為非條件跳躍的一種，首先，PC要跳躍到PC+imm，所以我們需要先計算這部分，在紅框部分可以看到add sum把這兩者相加後輸出訊號為branch\_out，且兩個cycle後PC更新為正確的值；另一個部分，寫回reg的為PC+4，這裡我是直接從PC端拉一條線過來，所以沒有運算步驟，但依然可以看到寫回的值為正確的PC+4(黃色部分)。

* **Prog0**
* **Prog1**

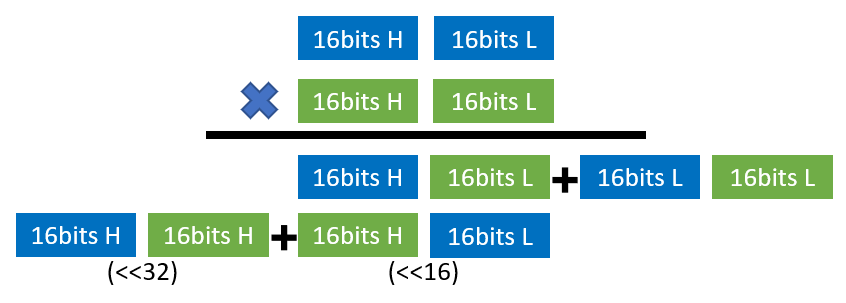
Bubble sort基本原理就是透過不斷和自己隔壁的人比大小，如果比較大就換位，所以在波形圖裡也可以看到我們的CPU會一直出現LW指令把資料讀到reg裡，再提出來做比大小並根據結果換位存值。

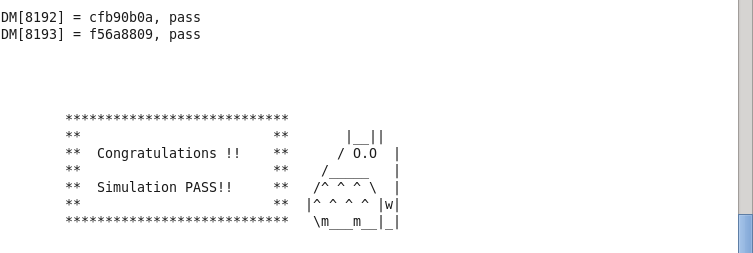
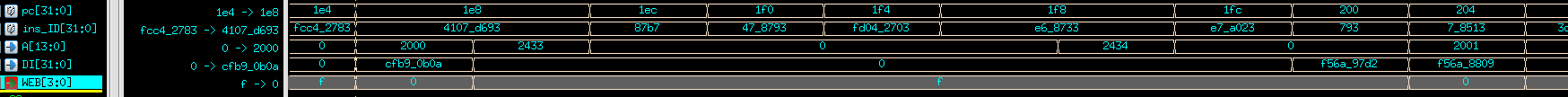
比大小

LW

LW

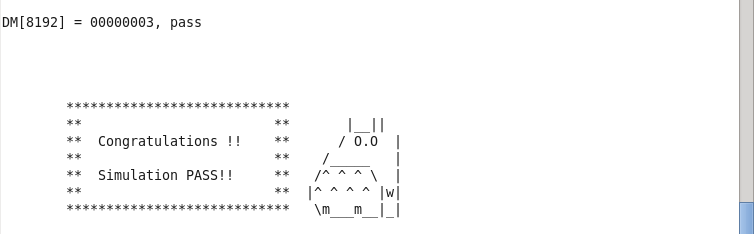
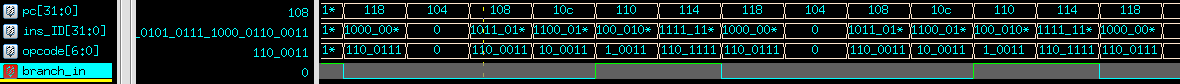
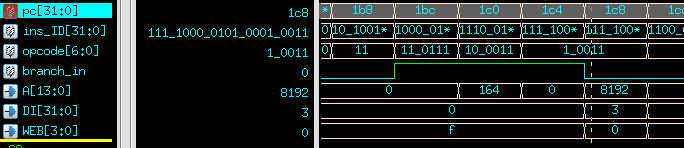
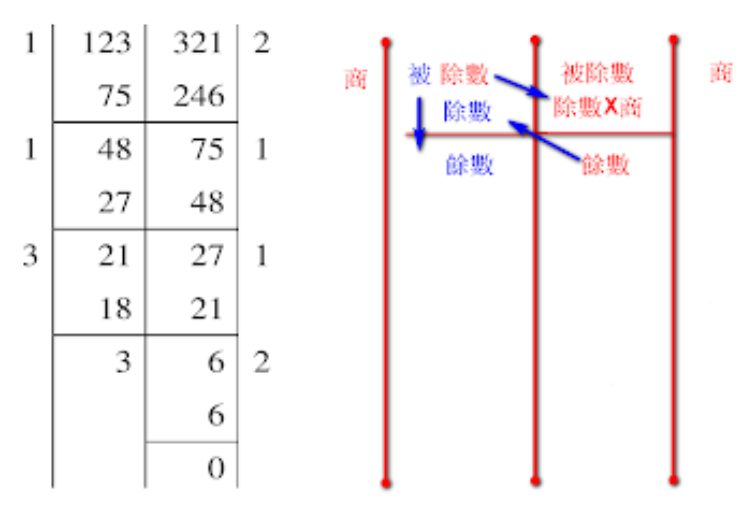
SW

* **Prog2**

32bit \* 32bit乘法最大問題在結果的64bit答案會造成溢位，所以我們必須把兩個成數拆成高位及低位，並用直式乘法的概念把答案重新合併成64bit，大概的概念如下圖。

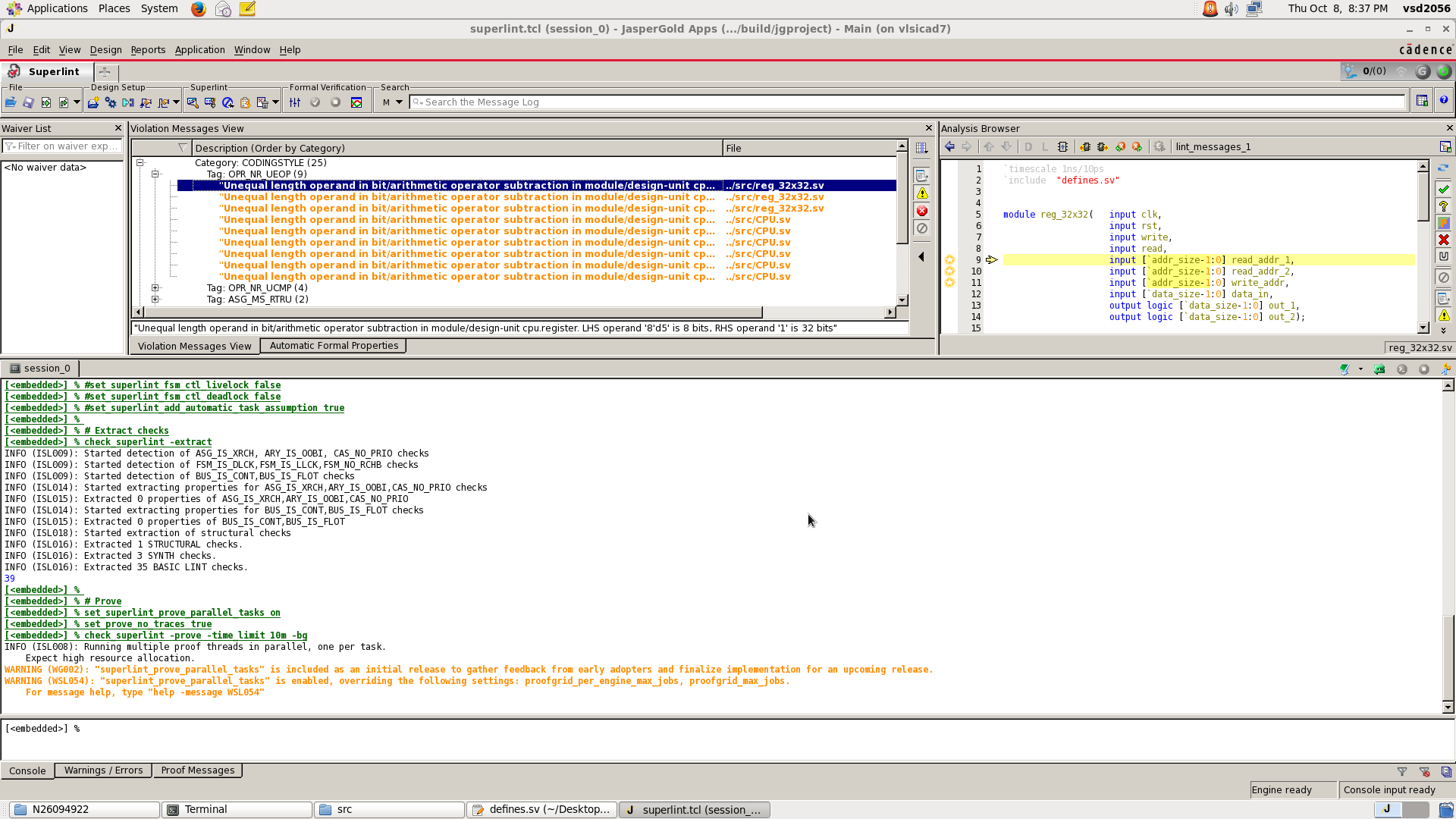
高低位的分別存入memory中

* **Prog3**

****最大公因數(GCD)可以用輾轉相除法的方式得到，所以在這裡的波形會看到CPU一直在比較大小並且觸發跳躍，直到答案出現(有其中一個數字變成0)。

寫入最終答案

* **Superlint warnings**

**問題1**

**原因:** 在define變數時define addr\_size為8 bits，但實際運用時寫了addr\_size-1，其中1預設為32 bits造成warning。

**解法:** 重新define addr\_size為32 bits。

**問題2**

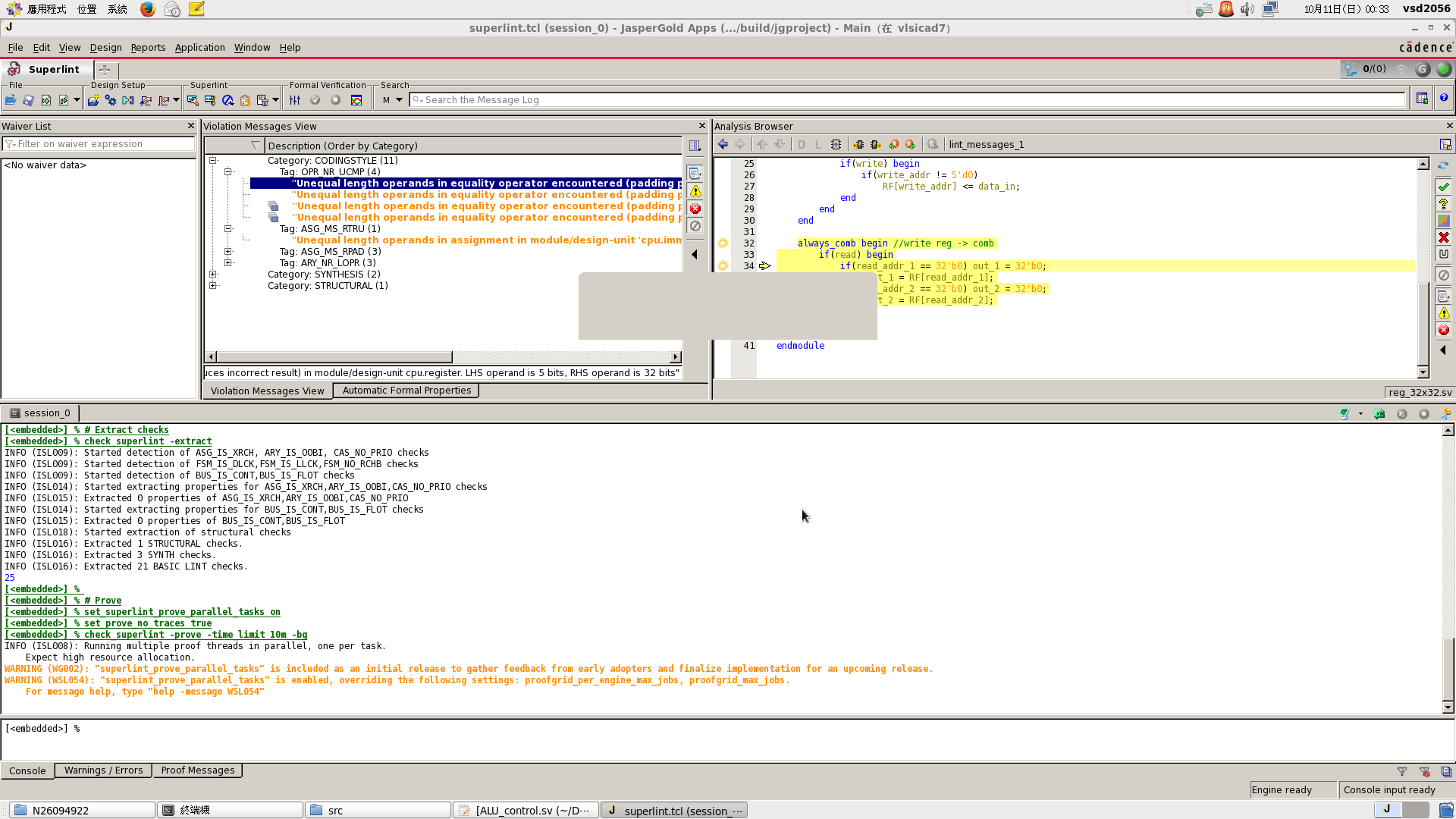
**原因:** 寫control訊號寫了case卻沒加default，出現latch。(沒截到圖)

**解法:** 加入default case。

**問題3**

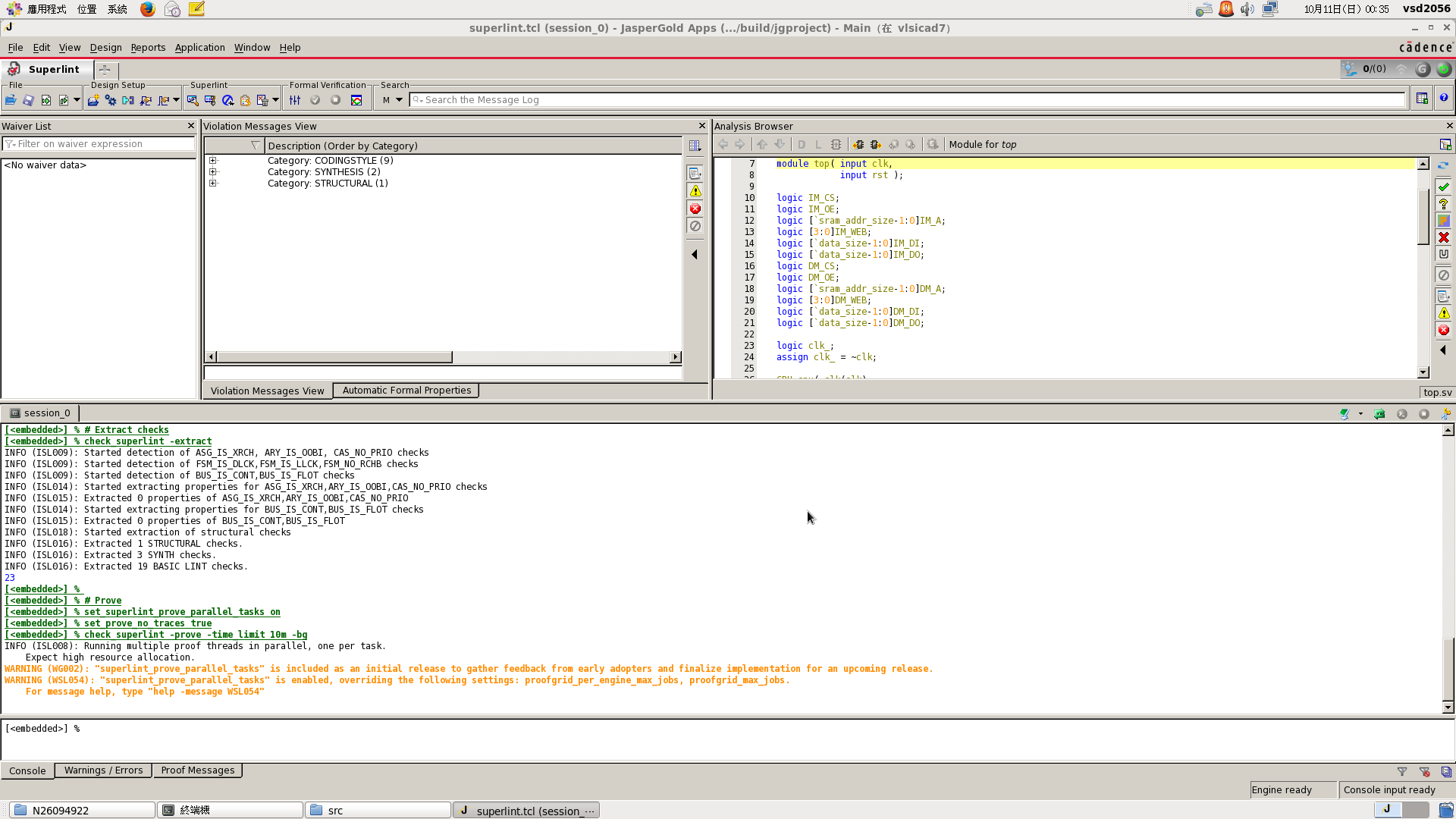
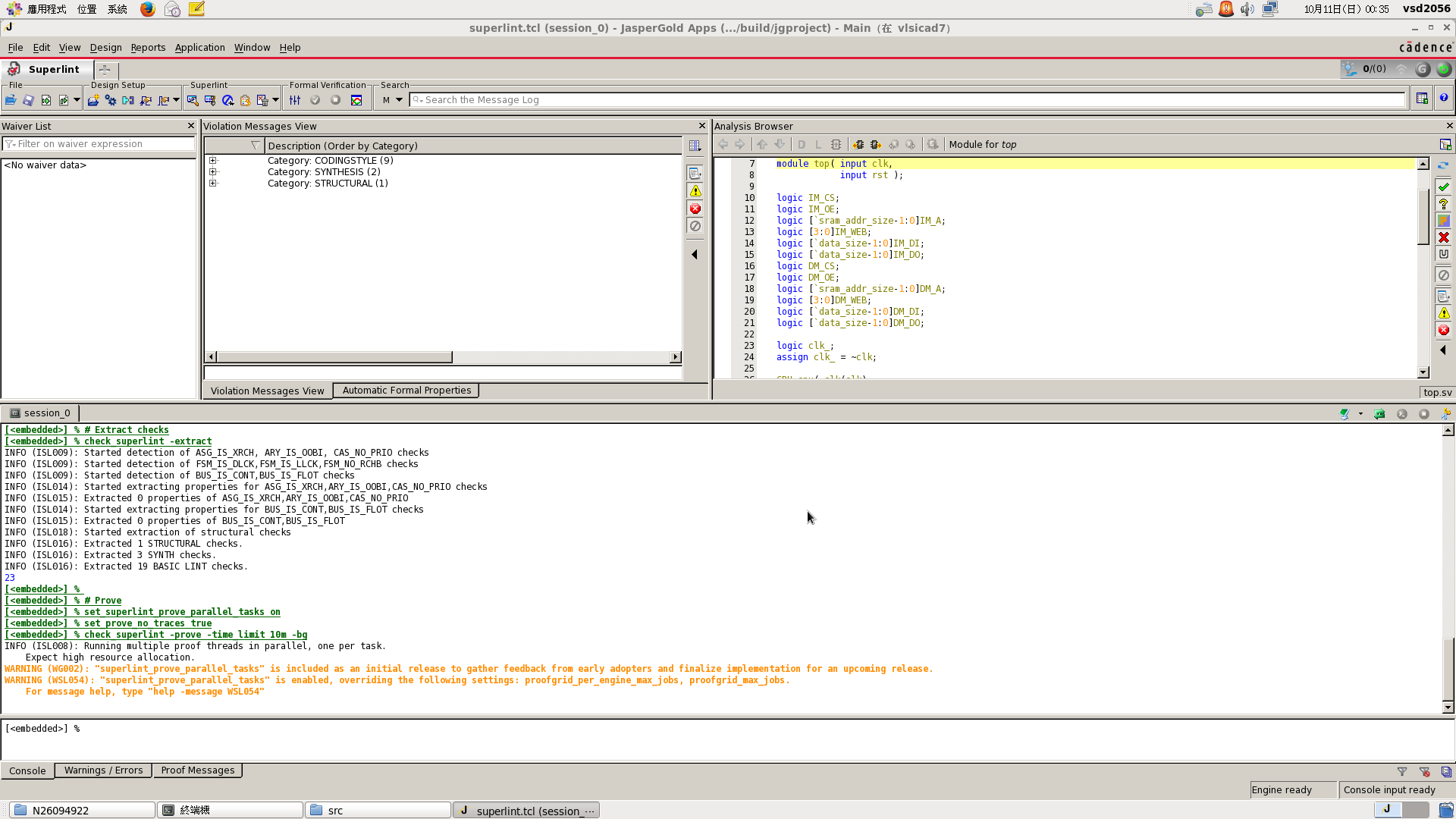
**原因:** 警告同時使用了clk和~clk。

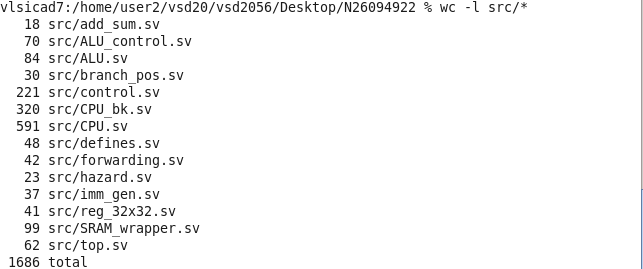
**解法:** 架構設計問題無法更改，詢問之後應該是沒問題的。

**問題4**

**原因:** reg file中address的bit數定義錯誤。

**解法:** 更正bit數量即可。

**Superlint result**

****

**總行數 1686**

**Warning 12**

**Complete = >99%**

* **Problems encountered**

在本次作業中我遇到最大的困難是時序問題，尤其是跟記憶元件有關的部分，包含我自己設計的register和助教提供的兩個memory，最主要的問題是發生在讀取資料(LW LB指令)時，我並不是很確定資料確切就緒的時間點是何時，尤其是助教提供的兩個memory，因為他們不是我所設計的所以我並不是很能掌握他們的運作原理，在我試著了解之後，發現這類儲存元件都是需要而外的CYCLE才能取出資料，這樣的而外CYCLE會有點類似自己已經走過一次flipflop，時間已經dealy過了，但這樣就會和我當初參考書本上的概念有所不同，且這件事會影響整個cpu架構，所以我使用clk的負緣觸發memory，讓資料提早半個CYCLE讀取完成，避免了而外CYCLE產生而能夠延續原始的設計，但是這樣的做法在Superlint會被警告，之後需要特別注意這樣設計會不會衍生其他問題。

另外在data hazard部分，我一開始也是參考書本的架構，但之後卻發現有些狀況是書本沒有提到的，像是在做forwarding的時候，書本上指設計了兩條forwarding路徑，也就是資料還卡在MEM和WB階段時的兩個狀態，但時測後發現其實”資料正在寫入register”的這個階段也需要被考慮進去，因為對後面指令來說這裡的資料也是無法取得的，所以我覺得這些都是我們需要自己思考，無法完全依賴書本的小細節。

* **Lessons learned**

在這堂課之前，我使用verilog的經驗只有在FPGA課程中寫過一些程式，真正要都出一顆具有完整功能的CPU我是完全沒有經驗的，以前是要寫出某個具有特定功能的電路，但現在卻是要想辦法透過控制訊號把各個module兜起來，所以在這次作業中我深刻了解block diagram對於完成一個系統的重要性，有了block diagram我才能清楚知道各個部件之間互相溝通的方式；另外我以前對於數位電路要看波形是非常排斥的，但經過這次作業之後，我已經能掌握一些看波形的技巧，我覺得這對我來說是個意外的收穫，也是非常有價值的經驗。