VLSI System Design

(Graduate Level)

Fall 2021

HW3 report

Must do self-checking before submission:

Compress all files described in the problem into one tar

All System Verilog 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: 鄭丞祥、陳冠佑

Student ID: N26094891、N26090130

# Problem Description

設計data cache以及instruction cache、DRAM wrapper、ROM wrapper，並將這些元件與CPU系統整合在一起

# Check List

|  |  |  |  |
| --- | --- | --- | --- |
| RTL code | | Synthesis | |
| Prog0 | Pass | Syn0 | Pass |
| Prog1 | Pass | Syn1 | Pass |
| Prog2 | Pass | Syn2 | Pass |
| Prog3 | Pass | Syn3 | pass |
| Superlint | 98.1 % | | |
| AXI\_VIP | PASS | | |
| Cache\_VIP | PASS | | |

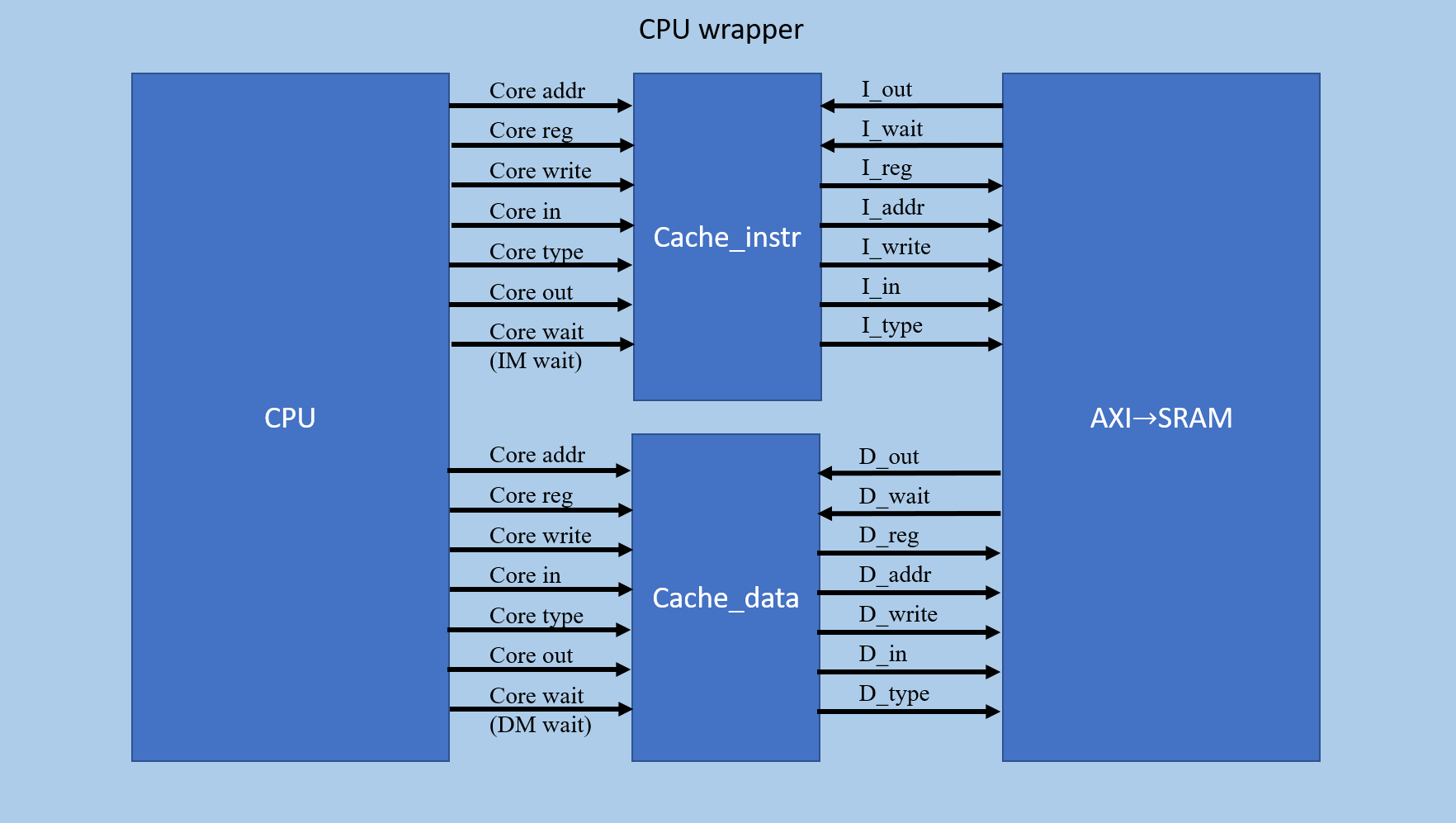
# Contribution

* 鄭丞祥(50%)
* 陳冠佑(50%)

# Problem 1

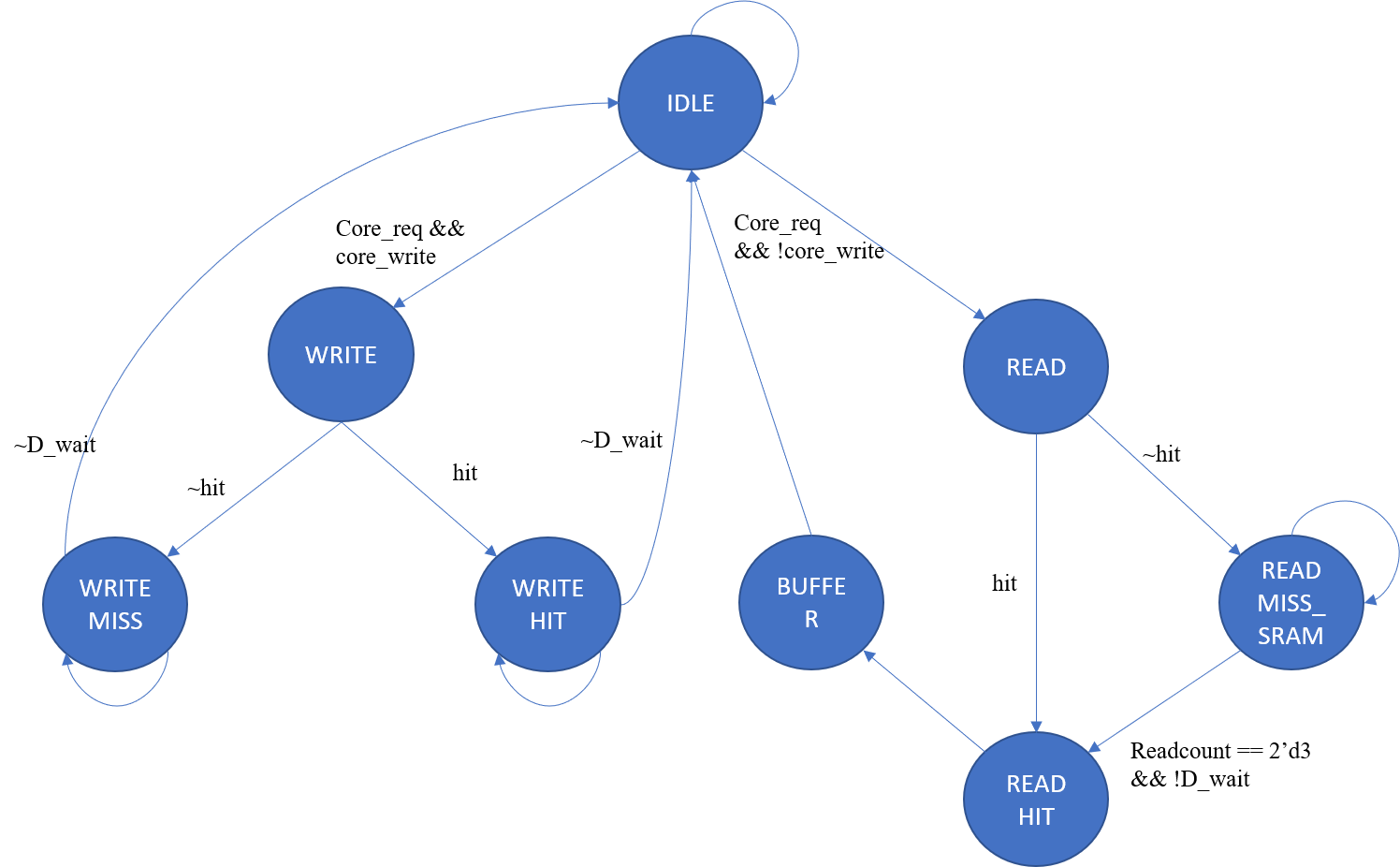
* Block Diagram

Implement a data cache module named “L1C\_data” and an instruction cache module named “L1C\_inst”



在Problem1中目的是要實作出一個cache，以協助CPU因為接上AXI後存取指令與資料速度過慢的問題，也就是加速存取想要資料的效率。而在這個部分中將會以L1C\_data為例子，因為LC1\_instr為LC1\_data的行為子集。

* FSM for Cache

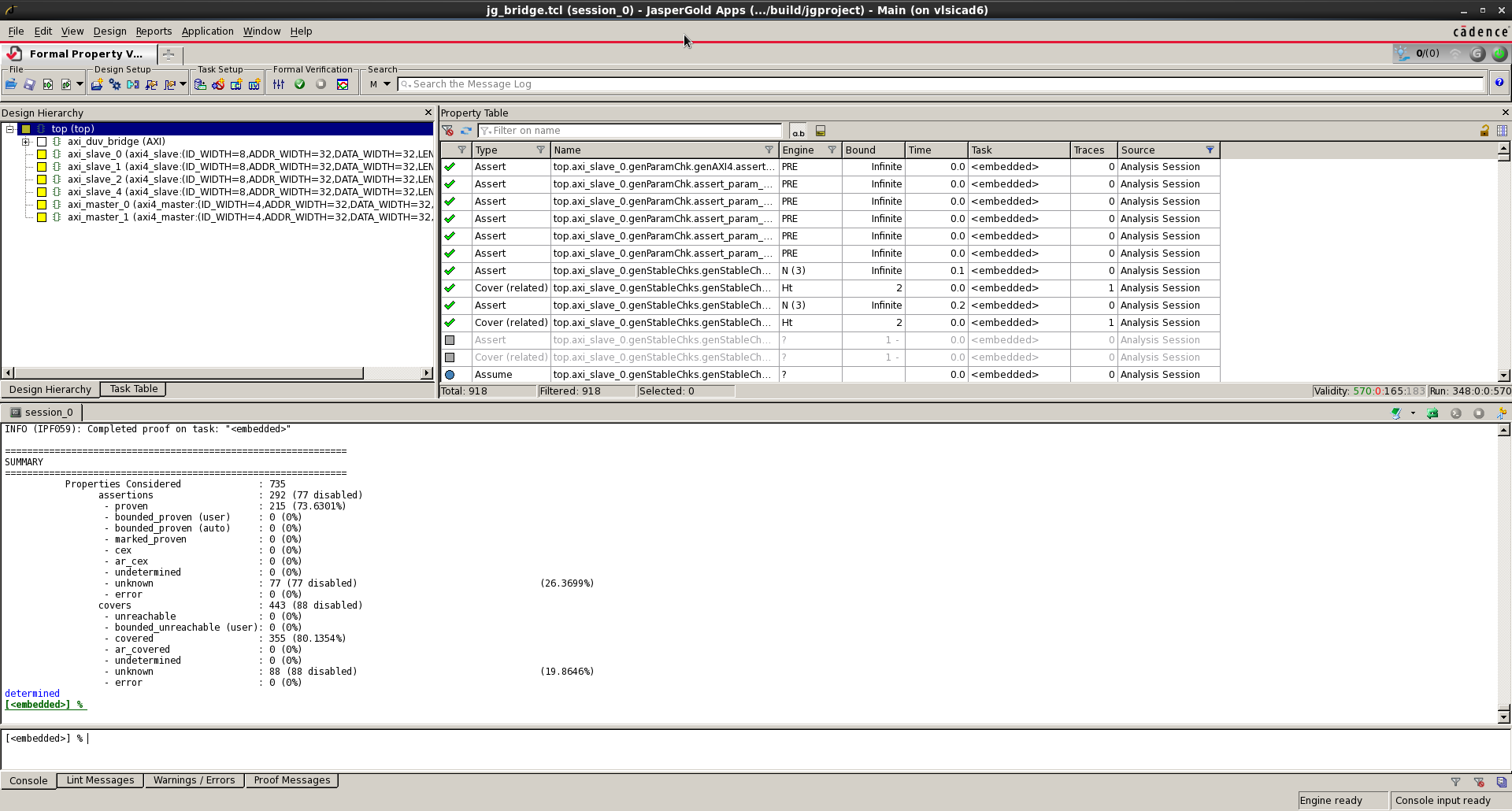


上圖為我們為cache所設計的FSM(以data\_cache為例)，狀態機在reset後首先會先進入到IDLE狀態來等待core\_req訊號，如果core\_req為1時會判斷這時候是否有core\_write訊號，如果core\_req和core\_write同時為1，代表這時候CPU要跟cache互動，而且是做資料寫入。若只有core\_req為1則表示要做資料讀取的動作。

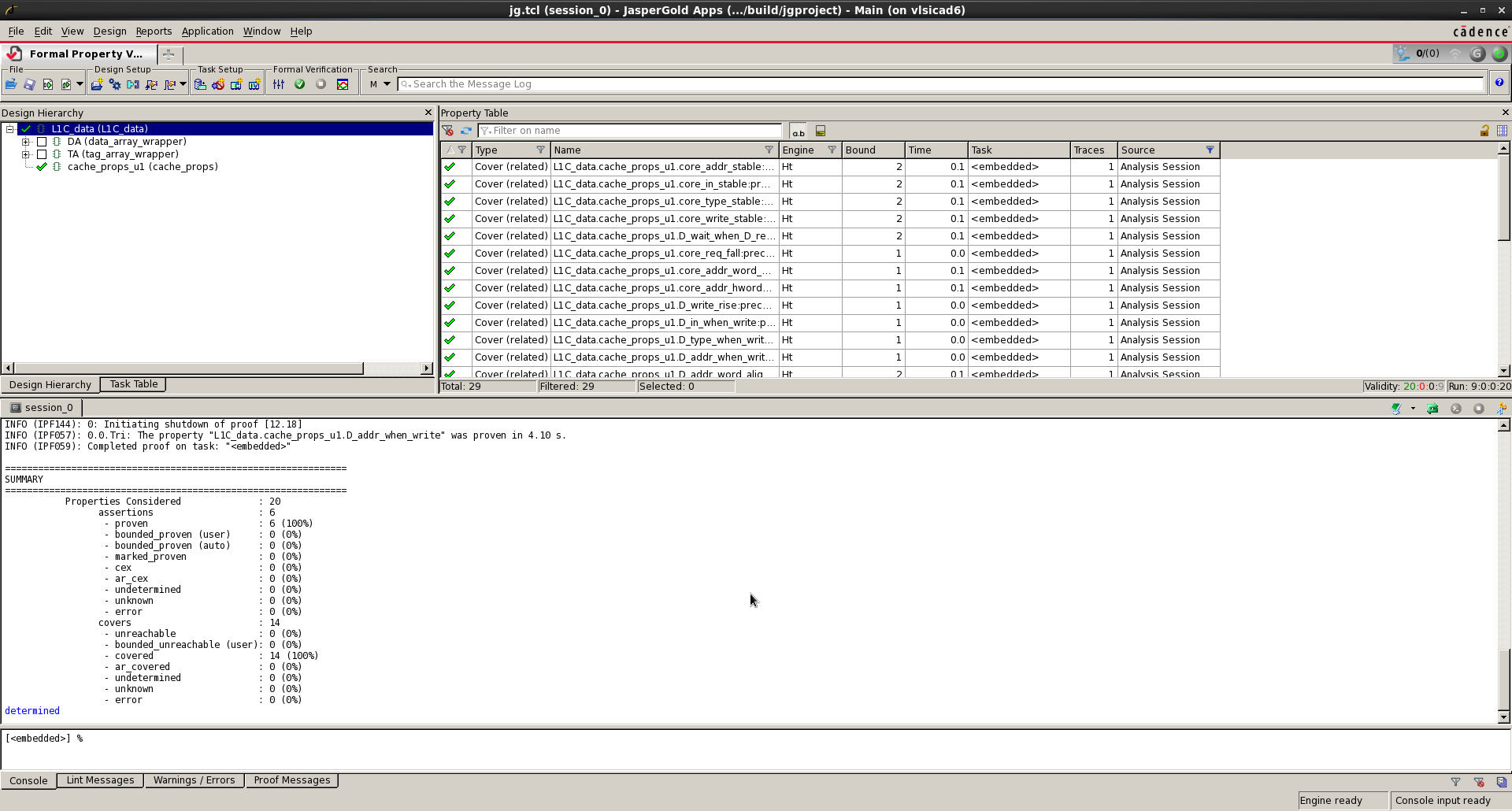
關於資料讀取的部分，首先會先在READ狀態判斷該筆資料的讀取是hit or miss，如果是hit的話，就會直接將DA\_read訊號拉為1，從data array讀取我們想要的資料給到CPU，並在下一個cycle進到buffer狀態，而進到buffer狀態後的下一個cycle會在無條件回到IDLE，完成這次的讀取。若是read miss，則此時會進入到READMISS state，在此state會讓cache從SRAM讀取一個line的資料到cache，作法是透過會在進入到這個state後，cache會發出所希望的存取的那條line的base address，並搭配上一個readcount來去依序讀出四筆資料。可以利用readcount來讀出所希望的四筆資料的概念是因為當D\_wait為0時，表示cache已經從SRAM讀完一筆資料回cache，而因為我們要一次搬四筆資料，所以只要我們還在READMISS的階段，並且D\_wait為0時，就會讓readcount+1，直到readcount == 2’d3，也就是讀了四次的時候，就會跳狀態到READHIT階段，讓CPU去data array進行資料的讀取，把剛剛我們寫入cache的資料讀出，之後再到BUFFER，回到IDLE等待下次的動作。

關於資料的寫入，會在write階段做write hit or write miss的斷判，如果今天是write miss的話會因為設計為write around的關係只寫入SRAM而不寫入cache，若是write hit的話，就會因為write through的關係寫入cache同時也寫入SRAM之後再回到IDLE狀態。

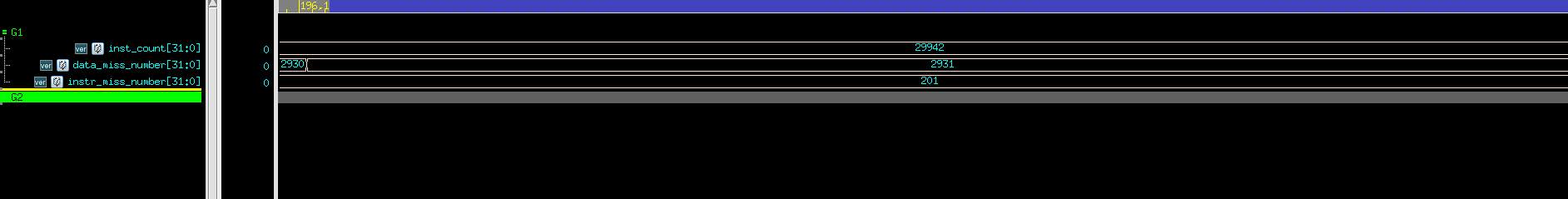
* VIP B verification: AXI的assertion全數通過



* VIP Cache verification: Cache的assertion全數通過



* Cache hit rate



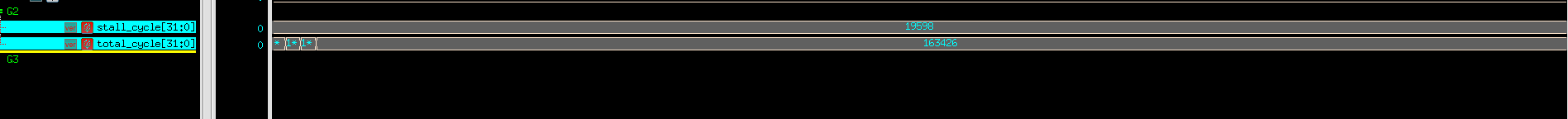
由上圖可以看到，在這次的作業中總共會用到的指令(已將stall和flush造成的影響給去除掉)共為29942個指令，而在instruction cache中會總共會miss 201次，而data cache 共會miss 2931次，因此

Instruction cache hit rate =

Data cache hit rate =

Cache hit rate會有這麼高的百分比原因在於此次的cache設計的大小共有1KB大小，足夠容納夠多的資料。指令部分會特別高的原因是因為指令大多是連續指令，今天cache一次從SRAM搬連續四筆資料到cache時很容易讓下面三個指令跟著hit，因此才能高達99%的Hit rate，data部分的話會因為branch、Jump之類的跳躍指令，導致資料讀取不連續，而使得hit rate下降。

* IPC



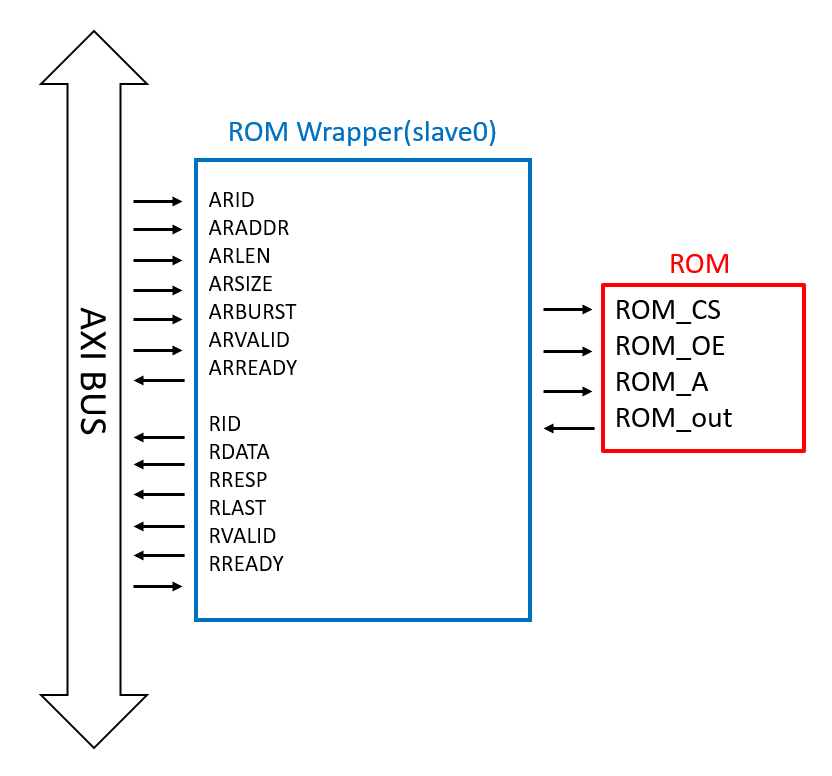
IPC = (total cycle – stall cycle) / total cycle =

# Problem 2

* Block Diagram

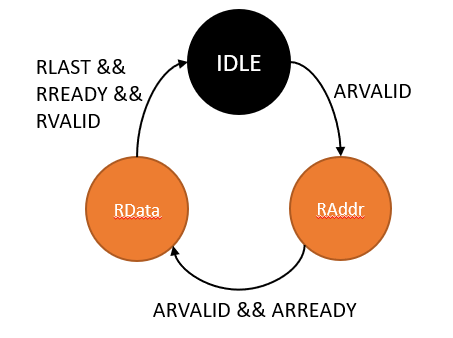
在problem2中我們必須實做出ROM wrapper、DRAM wrapper，並且完成booting透過執行ROM的指令將data從DRAM搬到IM以及DM放，並且執行IM之指令。  
以下會各自說明我們的ROM wrapper以及DRAM wrapper怎麼設計。

* ROM wrapper

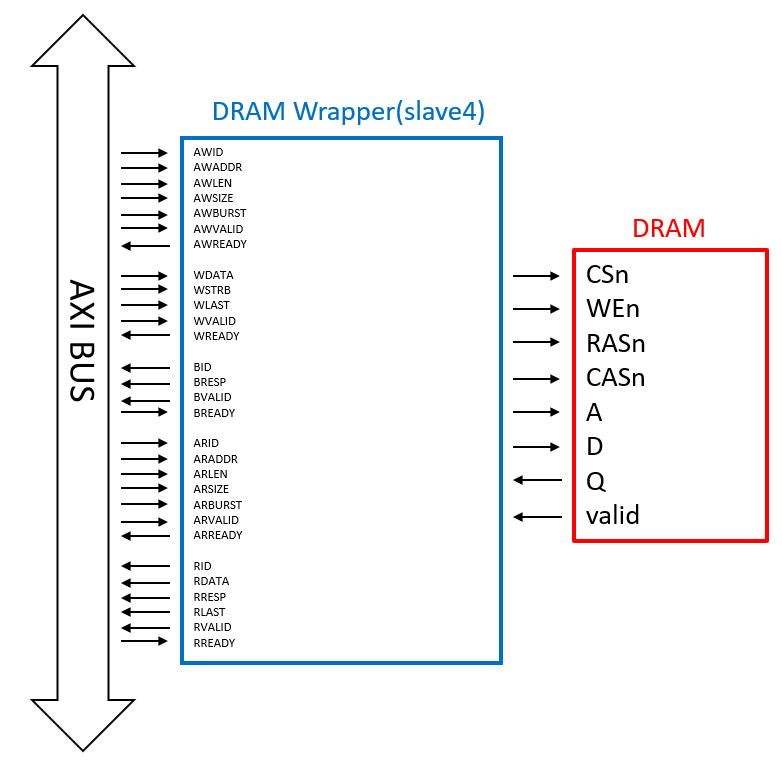


ROM的行為相較於DRAM簡單許多因為ROM為read only memory，只用來儲存booting怎麼搬資料的程式，(即把booting.c的C code轉為組合語言存在ROM中)，可以把他想為簡化版的SRAM去設計wrapper。

以FSM 來看可以簡單分為3個state，IDLE、RAddr、RData即可。因為是slave端的wrapper因此我的設計是用bus傳過來的一些訊號線做判斷來進行state的轉換。

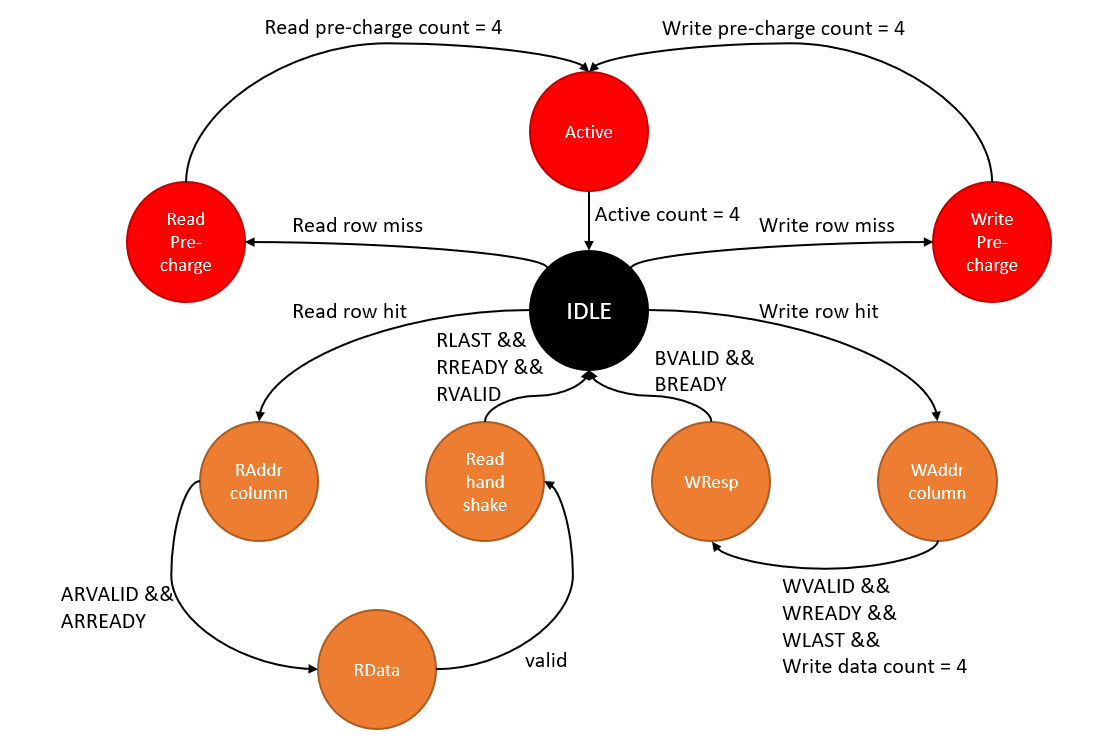


* DRAM wrapper

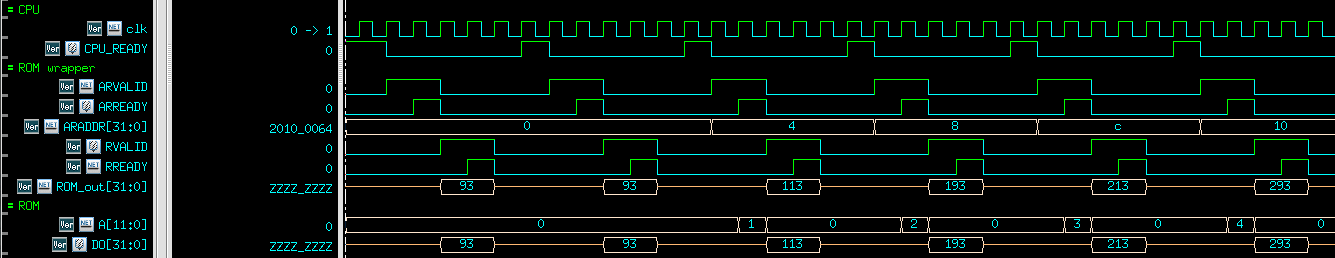


DRAM的行為相較於ROM以及SRAM就複雜許多，因為DRAM本身的特性並不是都使用電晶體兜成，因此會有許多電容充放電的問題，如TA跟老師上課所描述。

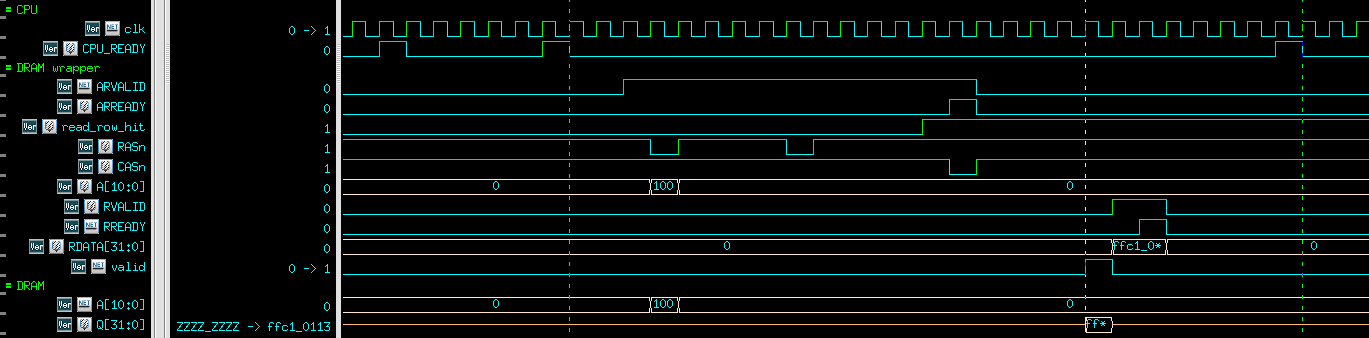
DRAM的地址是分row及column寫入，因此會有若這次讀寫的row跟上次不一樣的話，會先經過pre-charge state將原本的row釋放掉，在進入active state將新的要讀寫的row active，才可以在進入給column的state，反之若今天要讀寫的state與上一次讀寫的row一樣即可直接給column address進行讀寫，因此會需要read hit以及write hit來判斷此次讀寫row是否跟上次讀寫的row一樣進行state的判斷，除此之外因為此次作業有規定pre-charge active以及寫data要5個cycle因此在我的設計中state之間的轉換也有加入counter去數。而讀data的部分則是用valid訊號去判斷資料是否讀到來進行state的轉換。另外write stage會沒有write data state是因為DRAM的特性，會在給column address的時候給入要寫之DATA與Wen，因此我的設計將WAddr column跟write data state並在一起。



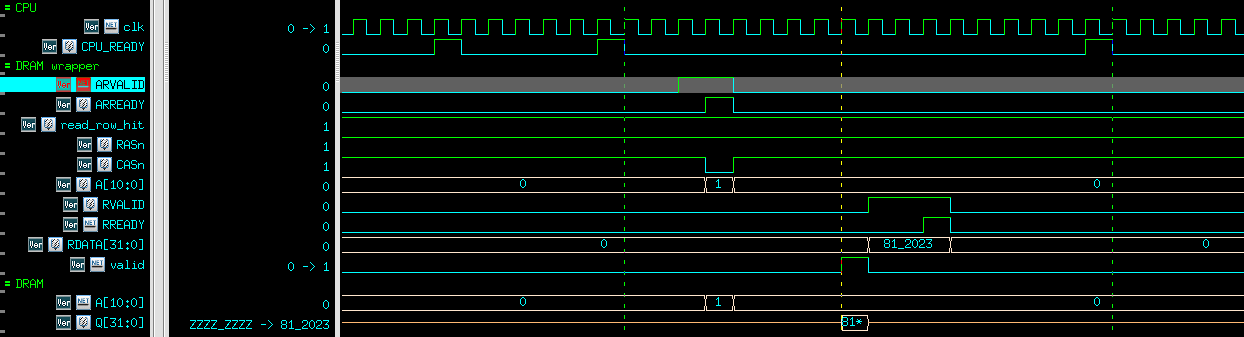
* Waveform

ROM Read 波形:

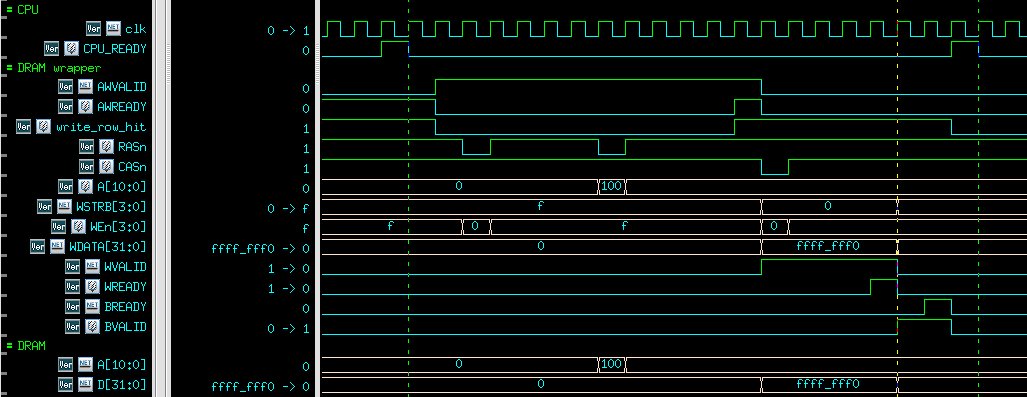
由波形圖中可以看出ROM的行為相對簡單，紅色框框為Read Address Channel的handshake，而data也可以馬上在下一個cycle讀出，黃色框框為Read Data Channel的handshake。

DRAM Read row miss波行:

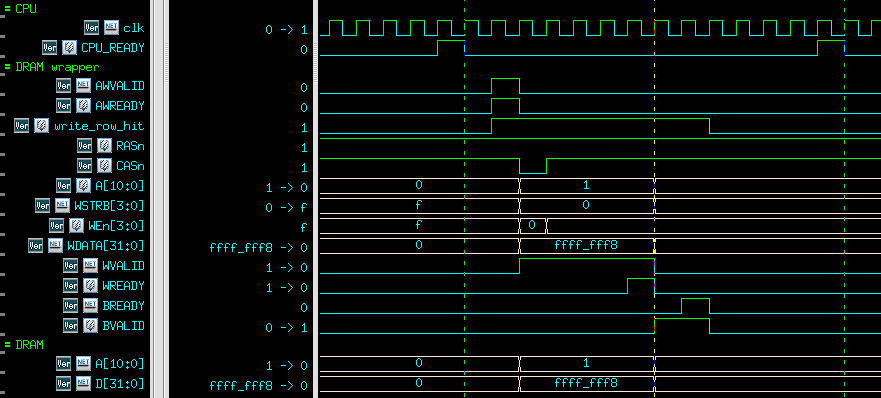
由波行可以看出這是read row miss的case紅色框框為read row miss因此需要先pre charge 前一個 row address 100，再經過5個cycle後黃色框框active 新的row address 0，再經過5個cycle藍色框框，給column address 0，再看到紫色框框valid訊號跟data(ffc1\_0113)一起出來，才會到橘色框框進行Read channel的handshake將資料給回master端。

DRAM Read row hit波行:

由波行可以看出這是read row hit的case紅色框框為因為hit可以直接給 column address 1，等待黃色框框valid訊號跟data(81\_2023)一起出來，才會到藍色框框進行Read channel的handshake將資料給回master端。

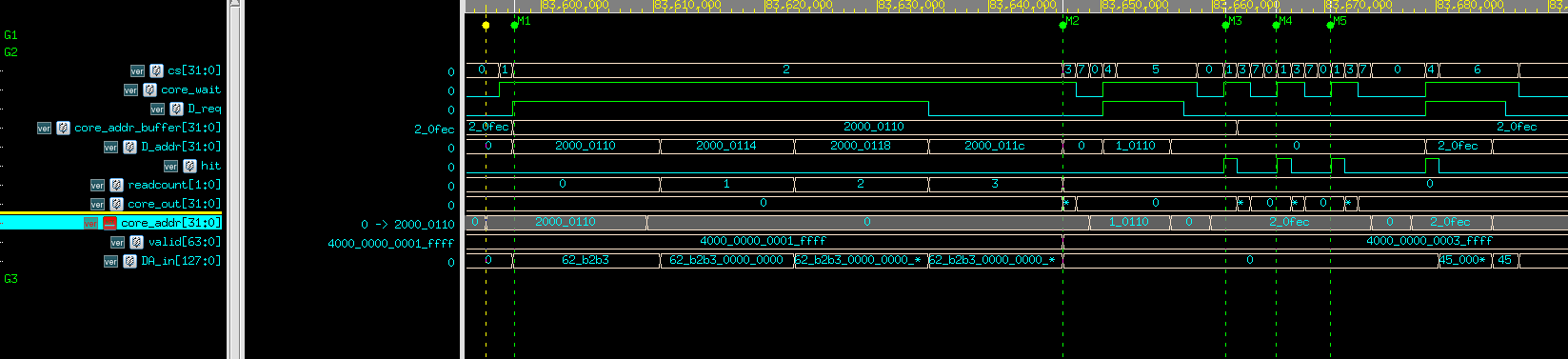
DRAM Write row miss波行:

由波行可以看出這是write row miss的case紅色框框為write row miss因此需要先pre charge 前一個 row address 0，再經過5個cycle後黃色框框active 新的row address 100，再經過5個cycle藍色框框，給column address 0並同時將要寫入之data(ffff\_fff0)以及WEn送入，再經過5個cycle看到紫色框框才進行write data channel的handshake，才會到橘色框框進行Write response channel的handshake完成寫入的動作。

DRAM Write row hit波行:

由波行可以看出這是write row hit的case紅色框框為因為hit可以直接給 column address 1並同時將要寫入之data(ffff\_fff8)以及WEn送入，再經過5個cycle看到黃色框框才進行write data channel的handshake，才會到藍色框框進行Write response channel的handshake完成寫入的動作。

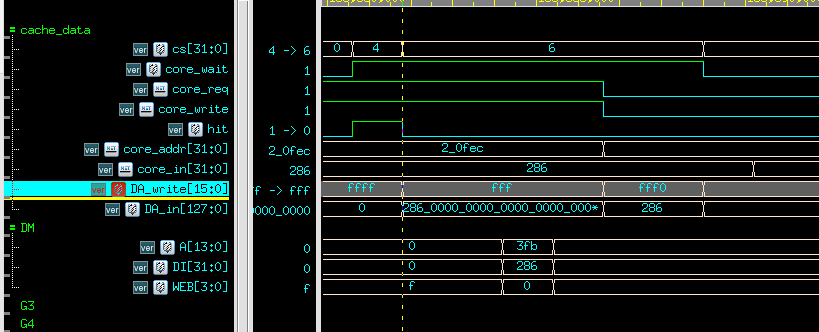
Case: Read hit / miss



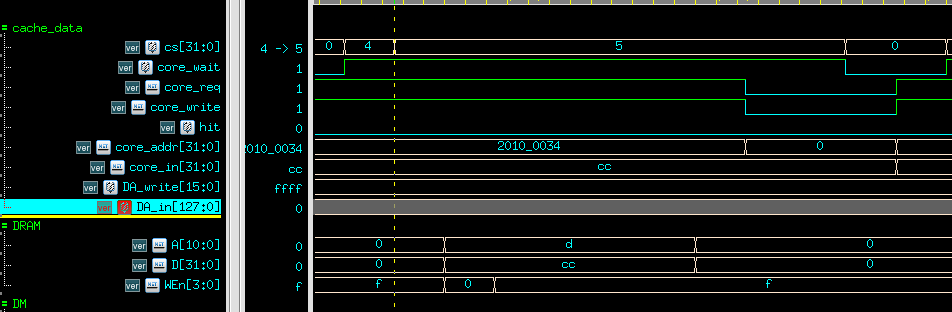
如上圖所示，可以看到在M1~M2區間，此時的state為READMISS，因此我們透過core\_addr\_buffer，持續地給予初始位置，並配合上readcount來達到讀出SRAM連續四筆資料到cache，也就是圖中的DA\_in 4個數值。

而在圖中的M3~M5階段，state 1因為READ階段，此時因為hit=1，所以會直接來到READHIT階段，將資料讀出(黃框部分)，來達到加快資料存取速度的作用。能不用再透過AXI bus到SRAM要相對的資料。

Case: Write hit / miss



Write hit: 當今天core\_req = 1 且 core\_write = 1時，表示CPU要寫入資料到cache裡面，如果果此時發生了hit，也就是要寫入的資料其位置與cache有重複，則當下這筆資料不僅會寫入cache，將同個位置的資料做更新，也同時會寫入到DM裡面。上圖中可以看到state=6的時候為我們的write hit state，在這當下cache會在位置20fec被更新成data=286，而此筆data在同一個state會更新進DM裡面。



Write miss: 今天假設是發生了write miss的情況，也就是寫入資料的位置噢cache沒有重複，則狀態機會進入到state 5，在這個state資料會直接被寫入到DRAM裡面而不會一起更新cache相對位置的資料，如上圖黃框所示。

* Prog0測試結果:



* Prog1: Sort Algorithm of half word

將助教給的half word從MEM中讀出以後再利用bubble sort由小排到大後在寫回MEM中



* Prog2: Gray scale

將memory中bmp檔格式的照片轉成灰階再以bmp的格式存回memory。

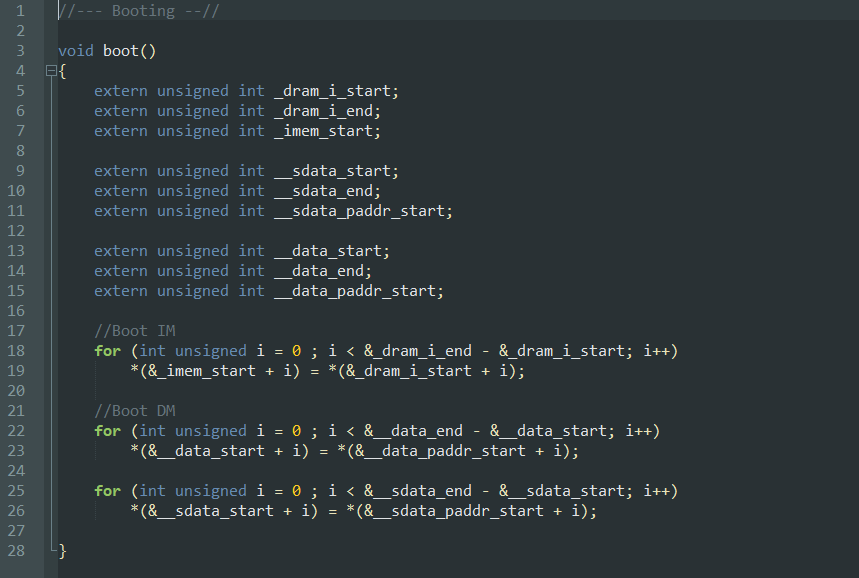


* Prog3: Matrix multiplication

將兩筆二維矩陣透過三層for迴圈來達到矩陣相乘的目的

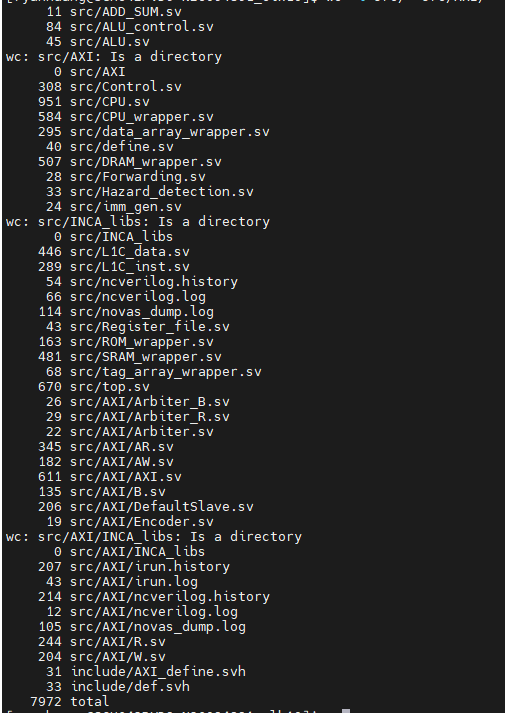


* Boot.c

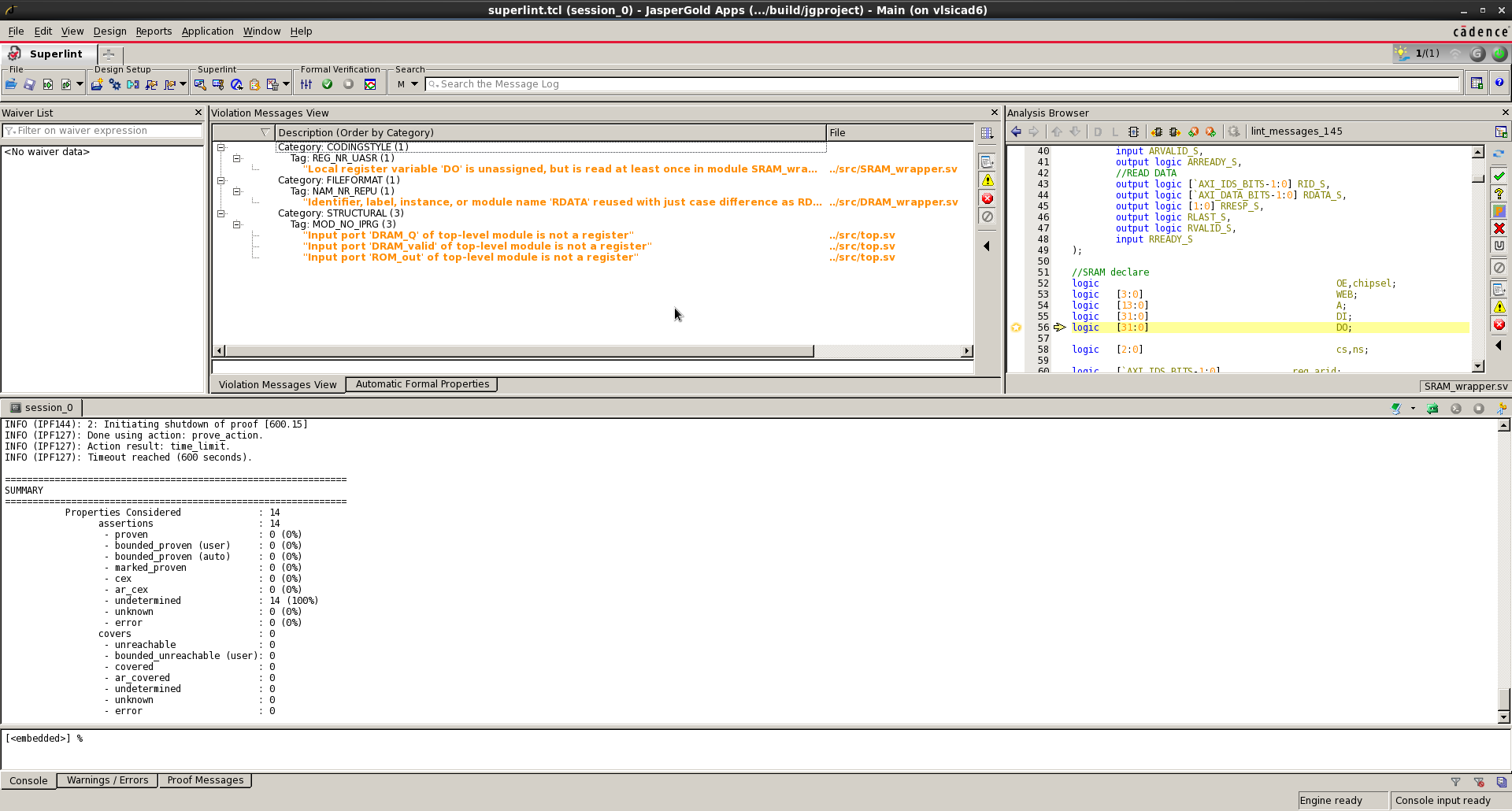


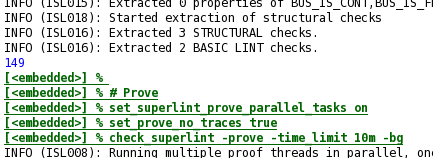
Boot.c其實就如同助教在投影片講述的，大部分都被行為都被定義得很清楚了，所以只要清楚知道要從哪裡開始搬運資料，要把資料搬運到哪邊就可以，例如\_dram\_i\_start就是DRAM的指令起始位置，會把牠搬運到\_imem\_start的相對位置。

# Number of lines of RTL code: 7972



# Superlint result: 149



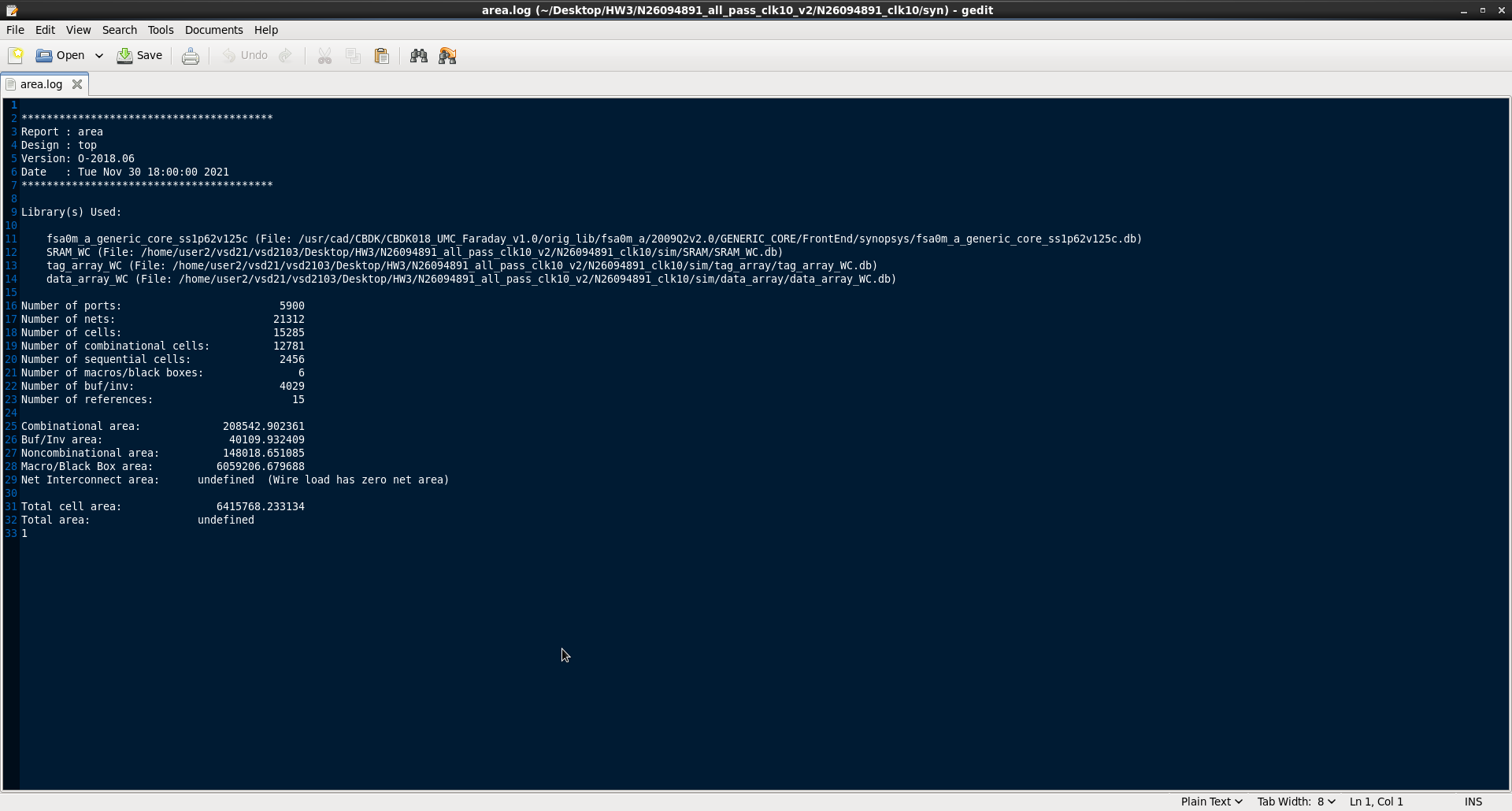


第1個warning為上次就有的warning，原因為DO命名的關係，並沒有assign給他值但卻把它的值給別人，但其實我們是有給DO值的在SRAM中，只是寫法為DO[i]，賦予每一個bit值，不是直接給DO一個值，所以系統出現這個warning提示，並不影響。

第2個warning為module定義的腳位RDATA與我們的狀態機的一個狀態命名一樣，但並不影響code的執行，所以並沒有特別修改。  
第3~5個warning為DRAM與ROM外接的腳位，但因為外接的DRAM跟ROM不是我們寫的，因此這裡我們也沒特別做修改。

# Synthesis:

* CLK -> 10
* Synthesis Area: 6415768.233



* Simulation time:
  + Prog 0: 1634270 NS



* + Prog 1: 22041380 NS



* + Prog 2: 42804690 NS



* + Prog 3: 2089550 NS



* PA:
  + Our: (t0 + t1 + t2 + t3) \* Area = (1634270+ 22041380+ 42804690+2089550) \* 6415768.2 = 4.39 \*
  + TA: (458680 + 4096390 + 10990320 + 452970) \* 6530417 =
  + Credit our get: (1.044 \* ) / () \* 15 = 3.567

# Problems encountered

關於在設計cache的會遇到的問題一開始應該算是data array以及tag array要怎麼給相對應的控制訊號線，因為一開始其實對於cache的行為沒有很熟悉，所以畫出狀態機以後就照著直觀的想法去給訊號線，但真正的問題是將cache整進去含有DRAM ROM的CPU系統裡面，會發生許多timing上的問題，像是此次可能需要做讀寫的動作，因為wrapper和cache算是擁有各自不同狀態的狀態機，因此即使我一開始訊號線給對，但可能會因為讀寫相關訊號線或是地址關係而無法順利地讓cache如期運作，也就是前面講的許多timing problem。

在設計DRAM wrapper時遇到最大的問題就是一開始我們不太懂DRAM的操作原理，因此我們花了不少時間才真正理解到若是讀寫miss的話，我們必須先pre charge 舊的row address才能active新的row address，才能再給column address，以及這些階段都必須等5個cycle等待電容充放電，我們有遇到data一直沒寫進去的問題，後來才發現是因為我們沒有等5個cycle讓電容確時充放電。  
而在最後做系統整合的時候也遇到蠻多問題的，因為這次CPU wrapper必須將兩顆cache包進去，但因為我們的設計中原本的CPU wrapper以及cache都有自己的FSM，所以在將兩個FSM整合的時候就會遇到不少的timing的問題，像是可能送到CPU的資料必須再buffer一個cycle之類的問題。或是像是在連續store word時也有遇到問題，也是我們再設計狀態之間的傳換多加一些訊號線讓他更嚴謹的判斷才解決的。

# Lesson learned

鄭丞祥: 此次作業中設計cache時一開始會遇到的問題是在計算機組織中所學過的cache概念完全忘光了，所以在一開始其實對於該怎麼設計cache的行為完全沒有想法，花上了一陣子的時間回憶cache的知識以及與同學討論，在撰寫過程中也曾因為一開始以為cache的行為是如此(TA以及DATA ARRAY)，但寫到後面才發現搞錯而重寫的，但如果將Cache的整體運作有了輪廓以後其實寫起來不難，就只是個FSM給訊號線而已，最困難的還是將cache整進去系統裡面與其他人溝通，常常因為彼此之間的狀態差一個cycle導致要不到想要的資料或是資料寫不進去，因此幾乎有六七成的時間都在debug。

陳冠佑: 這次作業我主要是延續上次作業寫wrapper的部分，以及修改CPU wrapper的部分，在寫wrapper的過程中，DRAM wrapper花了我比較久的時間，ROM wrapper因為只有讀相對簡單，而DRAM wrapper因為一開始不太熟悉DRAM的運作原理，所以花了一些時間理解，了解到若row miss必須先pre charge舊的row address再active新的row address，才能餵入column address，也是此次作業才讓我了解到DRAM真正的操作原理，另外在最後做系統整合時，因為CPU wrapper相較於上次作業要包上cache因此在debug的過程中花了我們不少時間，因為在整合cache的FSM以及原本CPU wrapper的FSM時會有許多timing的問題必須看波行遇到才會想到，因此整個系統兜起來後debug才是我們這次作業花最多時間的大宗，也讓我再次理解到老師上課說的debug是我們在做硬體架構設計的人花做多時間的階段。