Advanced VLSI System Design

Fall 2024

HOMEWORK IV

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

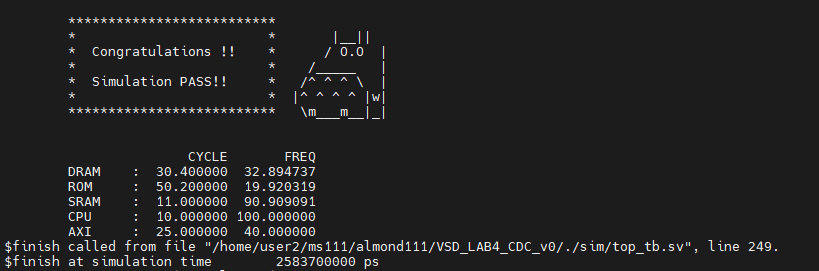
No waveform files in deliverables

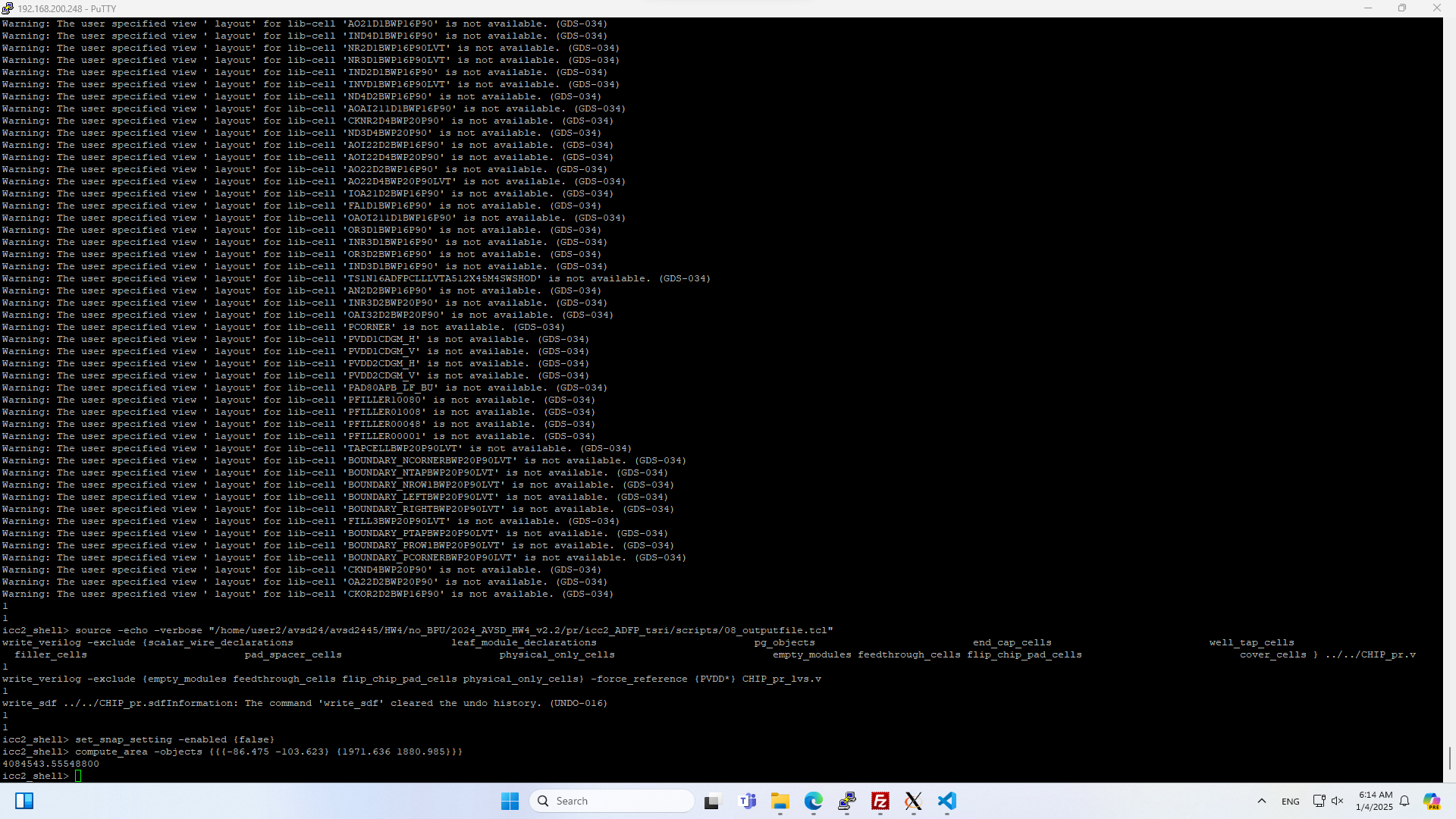
Student name: \_\_余祐安\_\_ \_\_王華昀\_\_

Student ID: \_\_N26134243\_\_ \_\_N26134308\_\_

|  |  |  |  |
| --- | --- | --- | --- |
| Performance & Area | | | |
|  | rtl | syn | pr |
| Prog0 time | 380620000 | 380620000 | 380620000 |
| Prog1 time | 322805000 | 322805000 | 322855000 |
| Prog2 time | 413485000 | 413485000 | 413535000 |
| Prog3 time | 3322745000 | 3322745000 | 3322795000 |
| Area(um^2)  In APR | 4084543.55548800 | | |
| CPU cycle | 5.0 | | |

Refer to the figure to fill in the simulation time !!





Cost 4.084543.55548800 \* 1017000 = 4153980.8 元

// Initiate your report from this point.

# Summary

本次作業中在HW3的基礎上，在CPU wrapper中加入了兩個Level 1 Cache，分別為Inst cache及Data cache，以增加系統運算速度。

除此之外，這次的系統各個IP分別以不同的clk進行，大大增加難度，由於每個IP都和AXI的clk domain不同，因此每個IP與AXI間都需要加上一個AFIFO處理CDC問題。

最後完成RTL後需要on chip進行模擬，合成過後也要使用ICC2來進行APR的設計，並通過所有驗證。

一張含有 文字, 螢幕擷取畫面, 圖表, Rectangle 的圖片

自動產生的描述

系統架構圖

# Design

## Caches (Instruction Cache, Data Cache)

本次作業需要新增Level 1 的cache電路，並採用2-way set associative的mapping方式，須遵循LRU(Least Recently Use)的輪換策略，block size為16 Byte。以下分為Inst Cache及Data Cache來進行介紹:

**INST CACHE:**

**一張含有 文字, 螢幕擷取畫面, 字型, 圖表 的圖片

自動產生的描述**

上圖為Inst Cache的狀態，架構上分為四個狀態，以下將詳細介紹各個狀態行為:

**IDLE**: 在Inst Cache中，由於IM\_read訊號會一直拉起，因此一個cycle就會直接跳到READ狀態。在IDLE狀態時，會將core\_addr直接送入TA及DA中，並且讀出其數值，這樣就可以在進入READ狀態時判斷有沒有hit。

**READ**: 進入READ狀態後，首先會先比較IDLE讀出來的TA與core\_addr判斷hit或miss，若是hit的話則直接把IDLE讀出來的DA送入CPU內，並在下一個cycle回到IDLE狀態。若是miss的話則會進入READ\_MISS狀態。值得注意的是，由於這次是使用2-way set associative的mapping方式，因此在比較是需要同時比較兩個Tag並決定要哪一個資料。而原先的設計是在READ時才將index送入TA，並與core\_addr進行比較，然而這樣的設計在RLT的時候是可以過，但是到post-sim時會因為有delay導致出現unknown，因此才改為在IDLE狀態時就將index送入TA，並在READ的時候比較。

**READ\_MISS:** 在這個狀態，Cache會去向master發出I\_req和讀取地址，這次cahce的block size是16bytes，我們會將I\_addr到I\_addr+4的四筆資料傳送進去cache中，原先在設計的時候一直以為是直接將core\_addr到core\_addr+4的地址的數值，然而仔細思考後才想到core\_addr不一定是填入block中的第一個數值，因此需只能取{core\_addr[31:4],4’b0}這段地址來發出需求，確保這一個block的數值都是同樣的tag。並且為了支援這部分，AXI的burst也要從之前的1調整為4，最後收到RLAST後進入DONE階段，完成更新TA及DA。值得注意的是本次設計並沒有在READ\_MISS狀態直接將數值讀入CPU內，而是需要在等回到IDLE、READ才會讀入CPU中，這部分應該是有優化的空間。

**DATA CACHE:**

一張含有 文字, 螢幕擷取畫面, 字型, 圖表 的圖片

自動產生的描述

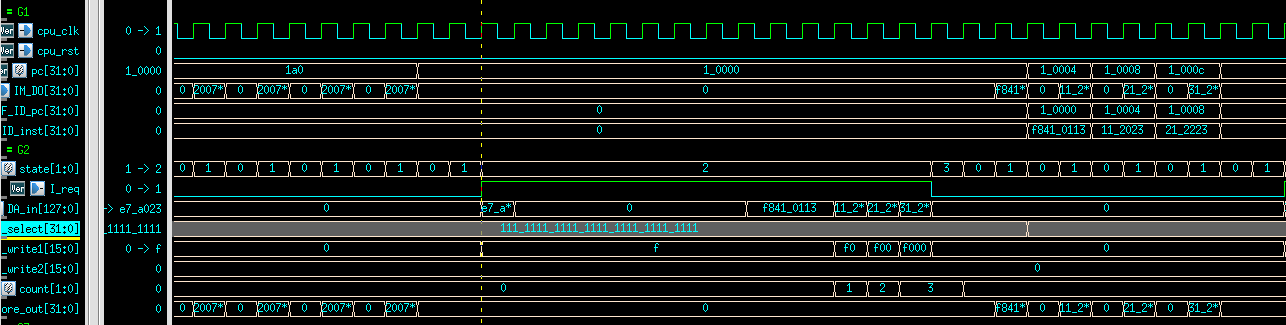
上圖為Data Cache的狀態，架構上分為六個狀態，以下將詳細介紹各個狀態行為:

**IDLE、READ、READ\_MISS:** 這部分的設計與Inst Cache相似，若是兩個Cache同時都miss則會先處理Inst Cache的需求，而Data Cache則會stall住，停在IDLE狀態，等待Inst Cache的需求結束，完成讀取後Data Cache才會開始跳狀態，而CPU則會完全stall住，直到兩個Cache都完成讀取。

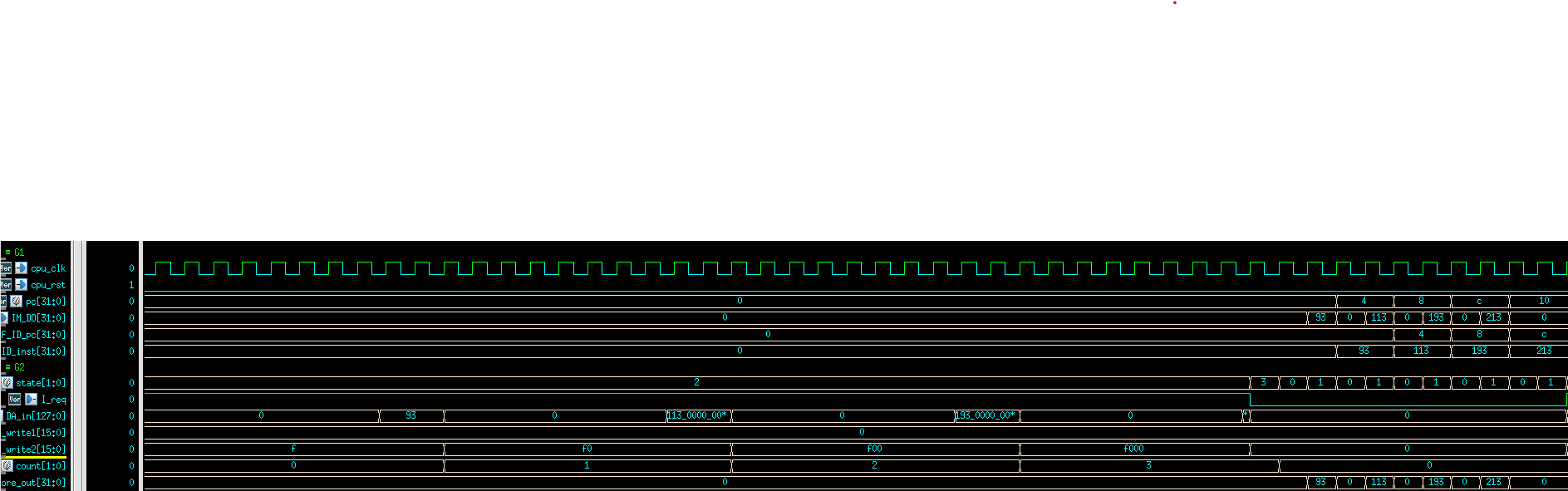
**WRITE:**在IDLE收到write\_req後會進到WRITE狀態，此狀態會將core\_addr及core\_data都進入slave進行寫入，並且由於本次作業採用write through的策略，每一次寫入若是hit都會更新到cache裡面，也就是會同步儲存cache以及slave端，這樣的好處是資料永遠會保持一致，在收到BREADY和BVALID代表slave端寫入完成，則進入到WRITE\_DONE。

以下用波型圖說明上述狀態行為:

**Inst Cache hit & miss**



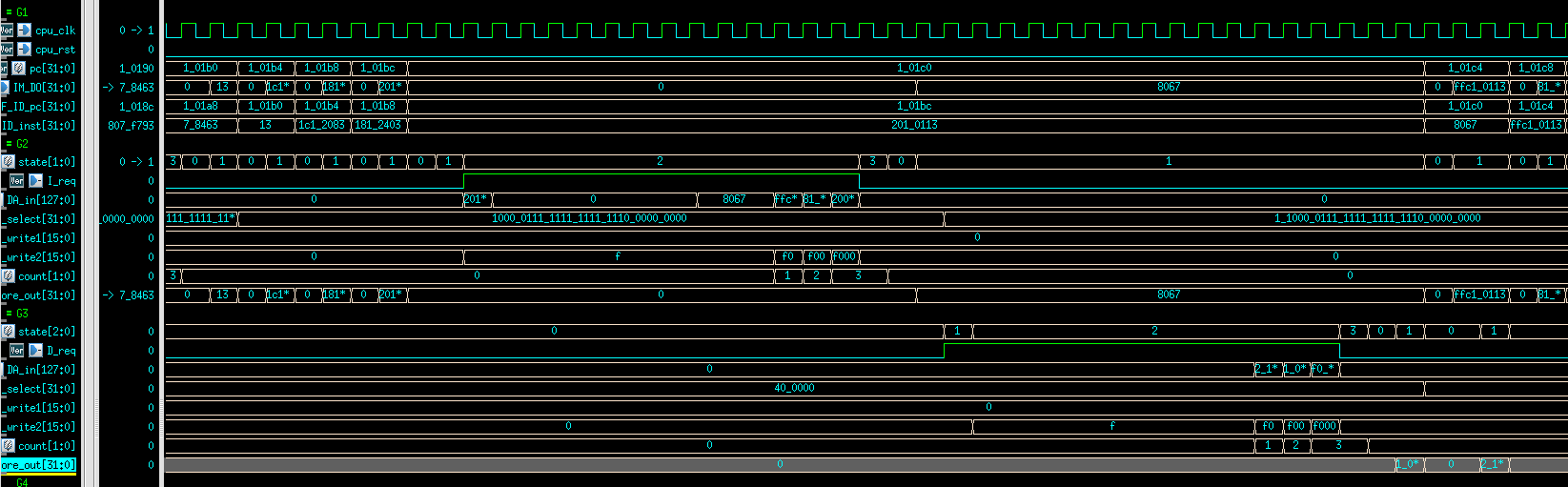
上方波型圖紅色框中可以看到當pc為1\_0000時，發生READ\_MISS，因此state跳到2進行miss的讀取，並透過LRU策略分配到way0，因此可以看到write1從f、f0、f00、f000，進行block的填入，並且在填入完成後狀態須回到0、1才會將數值f841\*讀入CPU，而橘色框中後續三個pc則由於hit發生，因此只需要兩個cycle就完成讀取數值。



上方波型圖同樣為READ\_MISS的發生，不同於第一張波型圖的是，這次的LRU策略將數值分配到way1，因此可以看到write2從f、f0、f00、f000，進行block的填入，並且在填入完成後狀態須回到0、1才會將數值93讀入CPU，而橘色框中後續三個pc則由於hit發生，因此只需要兩個cycle就完成讀取數值。

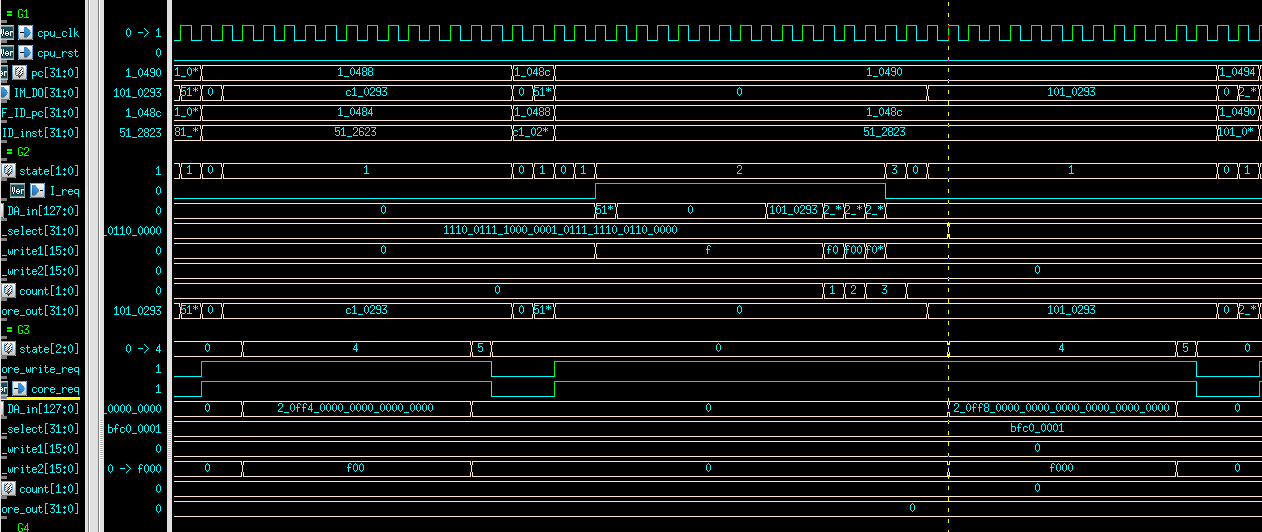
上述兩張波型圖說明READ\_MISS發生時，不同way的選擇，以及在寫入不同way的情況下，hit的發生，實現2-way set associative的mapping方式提高hit rate的特點。

**Inst Cache READ\_MISS & Data Cache READ\_MISS**

****

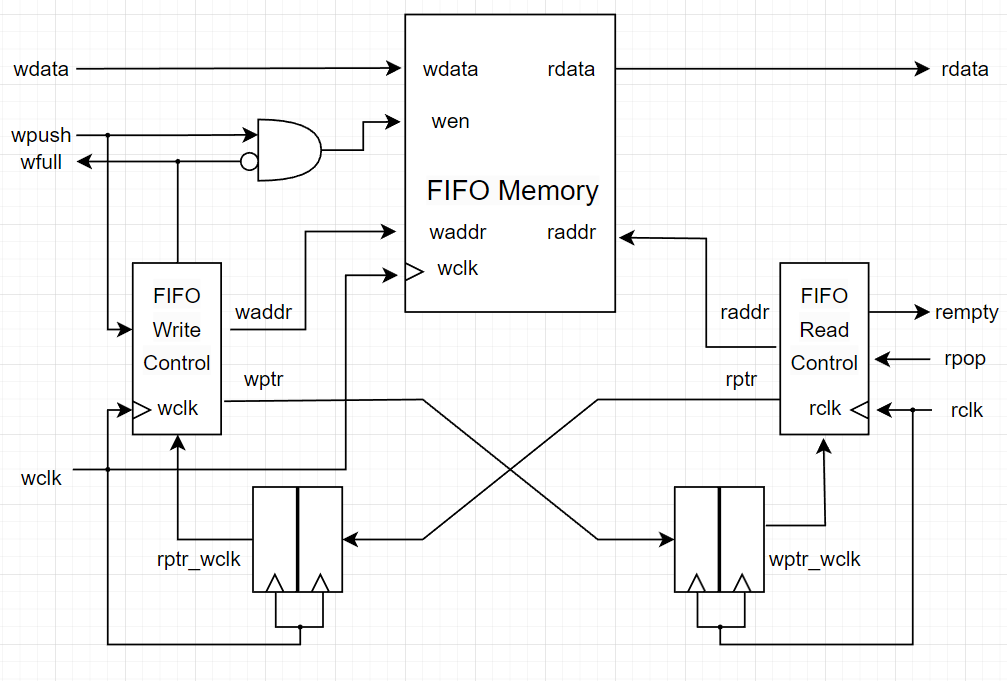
上述波型圖說明當兩個Cache都READ\_MISS時是如何運作，紅色框為Inst Cache而橘色框為Data Cache。從圖中可以看到當兩個都有READ需求時，Inst Cache會先跳狀態，而Data Cache則會stall在IDLE，當Inst Cache完成READ\_MISS並讀出數值後，Data Cache的狀態才會跳，同樣完成READ\_MISS並讀出數值後，CPU的stall才會結束，pc才會繼續跳。

**Inst Cache READ\_MISS & Data Cache WRITE hit**

****

上述波型圖說明當Inst Cache READ\_MISS而Data Cache hit時是如何運作，紅色框為Inst Cache而橘色框為Data Cache。從圖中可以看到當Inst Cache有READ需求時，Inst Cache會先跳狀態，而Data Cache則會stall在IDLE，當Inst Cache完成READ\_MISS並讀出數值後，Data Cache的狀態才會跳至WRITE，並且由於WRITE hit，因此可以看到DA\_in及write2都有跳說明寫入DA，而在寫入完成後跳至WRITE\_DONE狀態，回到IDLE說明寫入完成pc才會繼續跳。

## FIFO



此次系統由多個不同時鐘頻率所驅動，再藉由AXI進行資料傳輸，因此需要再時鐘交接處加上Asynchronous FIFO, AFIFO用來當作資料的緩衝區，以免資料丟失和衝突。

AFIFO主要處理2個不同時鐘頻率分別為控制數據寫入 FIFO 的WCLK和控制數據從 FIFO 讀出的RCLK，其內部由4個部分組成，

1. **數據存儲單元**：存放寫入和讀取的數據。
2. **寫指針 (Write Pointer)** 和 **讀指針 (Read Pointer)**：用於追踪 FIFO 的寫入和讀取位置。
3. **跨時鐘域同步器**：確保在不同時鐘域之間傳遞指針或控制信號時避免元態 (Metastability) 問題。
   * 一般會採用雙觸發器或多級觸發器進行同步。
4. **空滿指標 (Empty/Full Flags)**：指示 FIFO 是否為空或已滿。

在原始的非同步FIFO中就已經有這些組成，但在AFIFO中，讀和寫由不同時鐘驅動再經由pointer進行溝通，其中為了解決Metastability將pointer轉換為GRAY code在使用原先1 bit的打2拍的方式進行同步，即可同步pointer。

## AXI

因為此次AXI為主要多個時鐘交接處，因此會將AFIFO接在AXI之上，並與每個Wrapper的各通道進行連接，如下圖所示(截圖自助教HW4)。

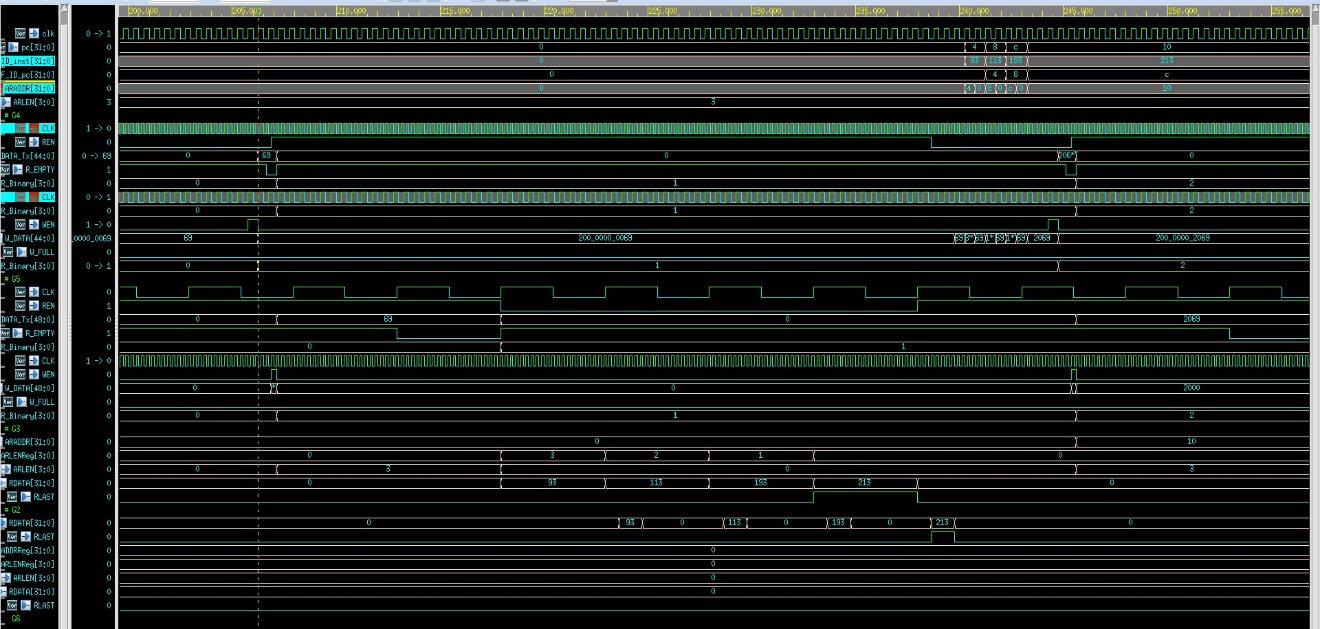


此次與之前較為不同處在於，先前握手是Master和Slave直接進行握手，但礙於clock domain不同，握手變成和FIFO握手，在空時，可以寫入，在滿時，則不可寫入，只可讀取，並透過FIFO做為資料緩衝區。因為這次實做8層AFIFO，不用等待對方拿完才可在寫入，會更有效率些，並且也支持BURST的長度。

**M0 🡸read🡺 S0**

這個階段是CPU讀取ROM的開機指令，M0發送AW訊號，S0回傳W的資料。

首先，AR\_M0 為空，可以寫入資料，M0把地址等資訊寫入，寫完後FIFO空指標關閉，AXI就會讀取資料並經由arbiter把資料導向slave，此時只要slave端的AFIFO未滿，即可把資料寫入，等待slave拿取地址等資訊，即可完成一次AW傳輸。當Slave要回傳data時和先前相同，只是走的通道是R通道，並且有獨立的AFIFO確保資料不衝突。



**AR\_M0**

**|**

**AR\_M0\_AFIFO⬄AXI**

**|**

**AXI ⬄ AR\_S0\_AFIFO**

**|**

**AR\_S0**

**R\_M0**

**|**

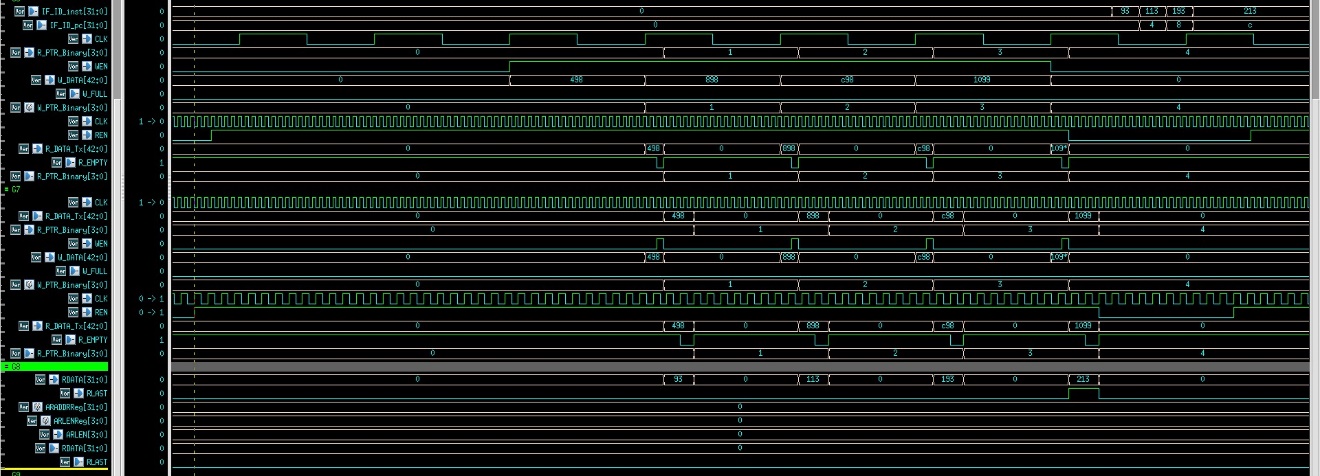
**R\_M0\_AFIFO⬄AXI**

**|**

**AXI ⬄ R\_S0\_AFIFO**

**|**

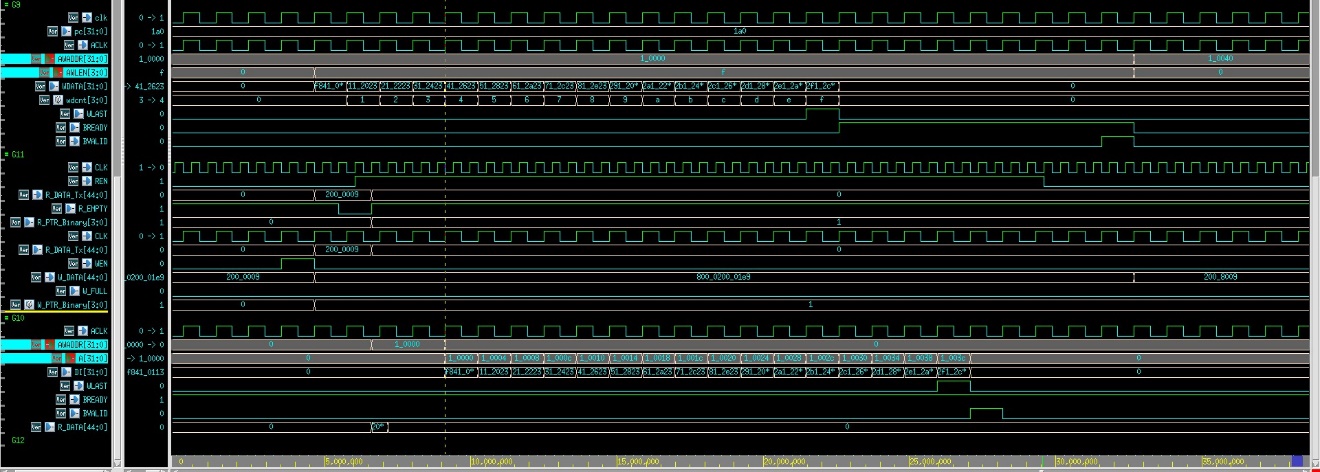
**A\_S0**



**M2 🡸write🡺 S1**

這個階段是DMA從DRAM搬移指令到IM，在此之前M2已經於DRAM讀取完資料，過程如同上個例子，此時需要把資料寫入IM，過程為M2發送AW訊號，並且傳入W的資料待完成後，S1回傳B訊號。

首先，AW\_M0 為空，可以寫入資料，M0把地址等資訊寫入，寫完後FIFO空指標關閉，AXI就會讀取資料並經由arbiter把資料導向slave，此時只要slave端的AFIFO未滿，即可把資料寫入，等待slave拿取地址等資訊，即可完成一次AW傳輸。此時M2把要寫的資料經由W通道傳入AFIFO，滿指摽拉起，M2停止傳輸，待Slave取出資料集完成W通道傳輸，最後Slave要回傳B時和先前相同，只是走的通道是B通道，並且有獨立的AFIFO確保資料不衝突。



**AW\_M2**

**|**

**AW\_M2\_AFIFO⬄AXI**

**|**

**AXI ⬄ AW\_S1\_AFIFO**

**|**

**AW\_S1**



**B\_M2**

**|**

**B\_M2\_AFIFO⬄AXI**

**|**

**AXI ⬄ B\_S1\_AFIFO**

**|**

**B\_S1**

**W\_M2**

**|**

**W\_M2\_AFIFO⬄AXI**

**|**

**AXI ⬄ W\_S1\_AFIFO**

**|**

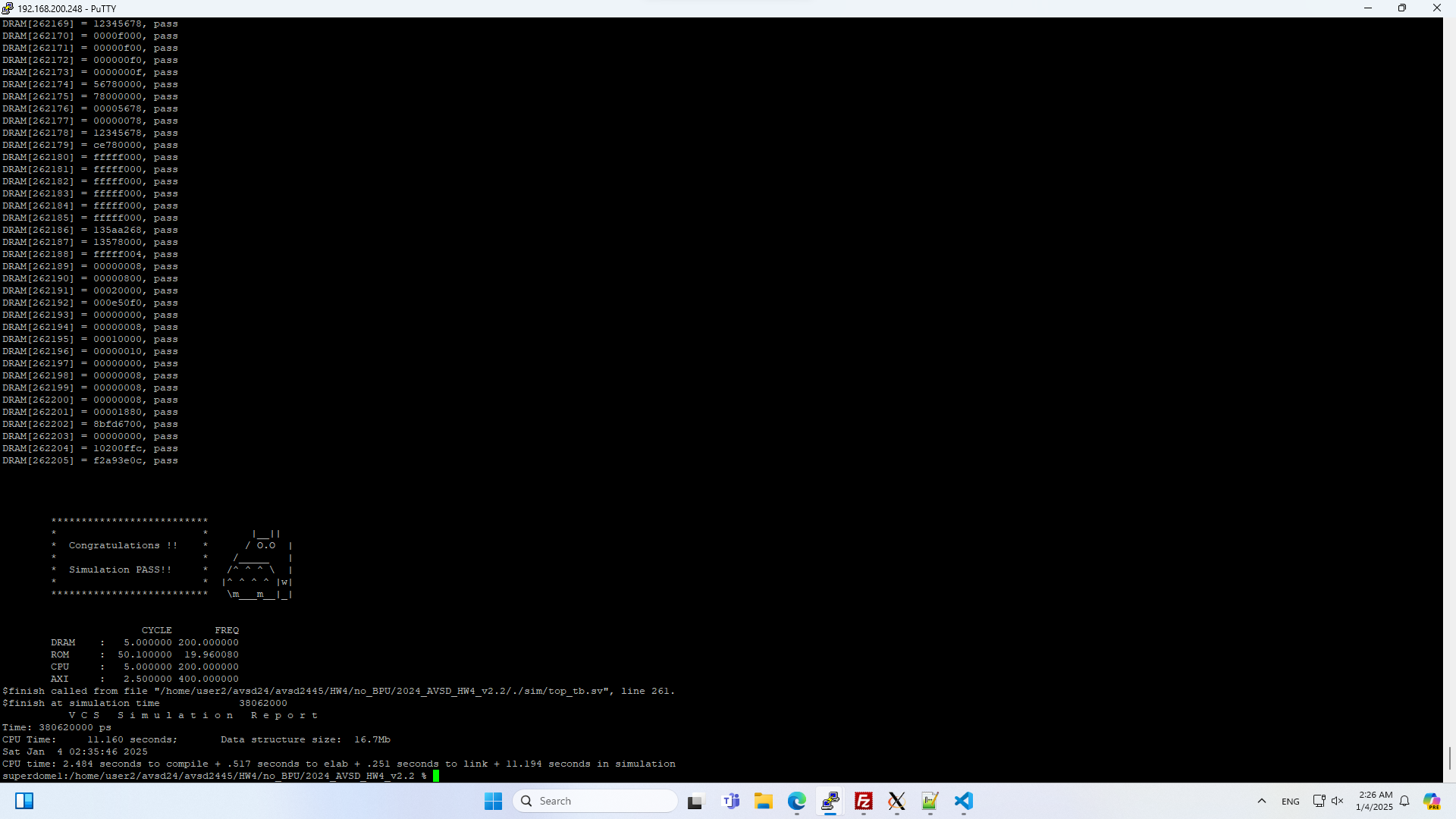
**W\_S1**



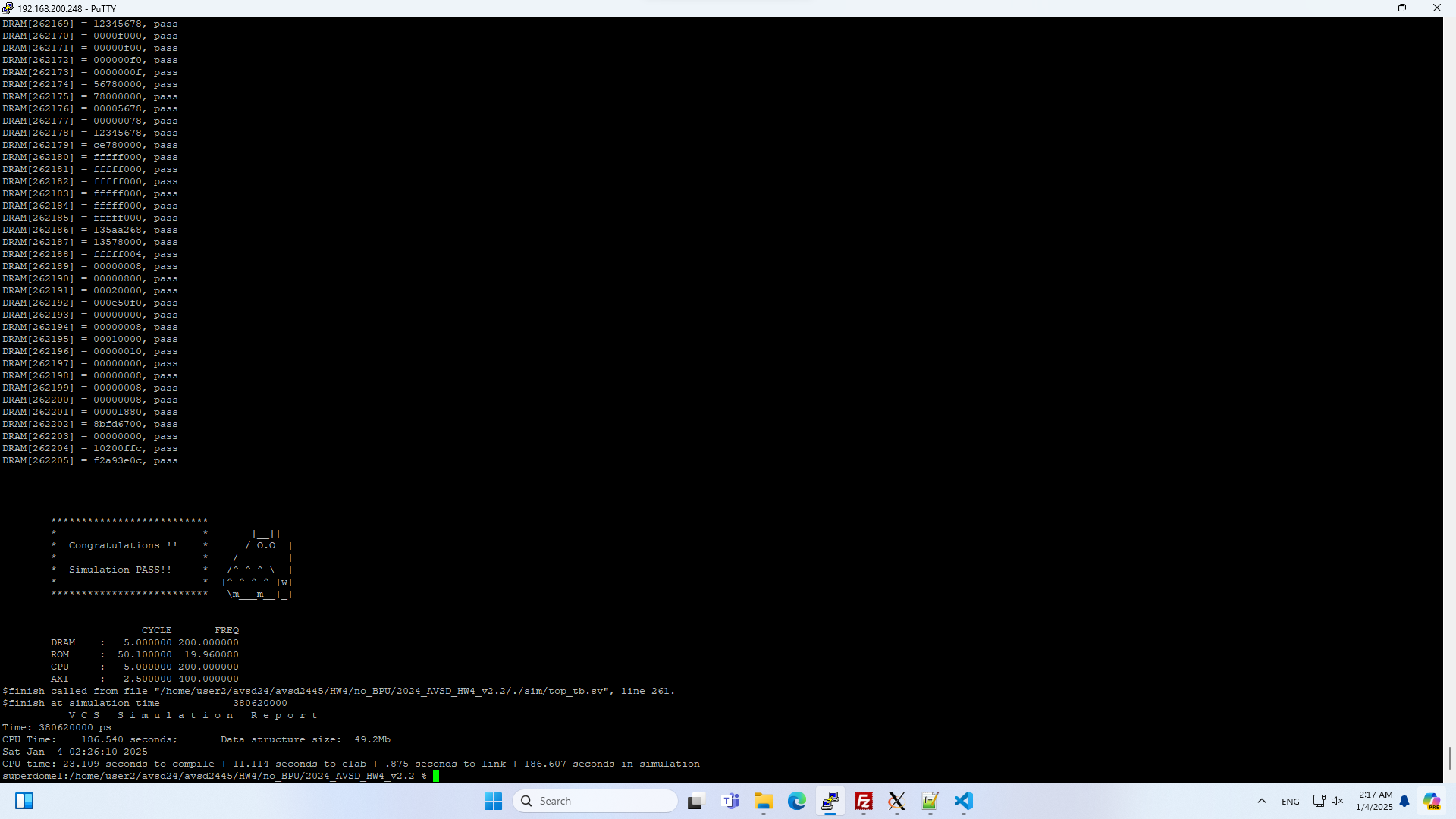
# Simulation Results

## Prog0 pass

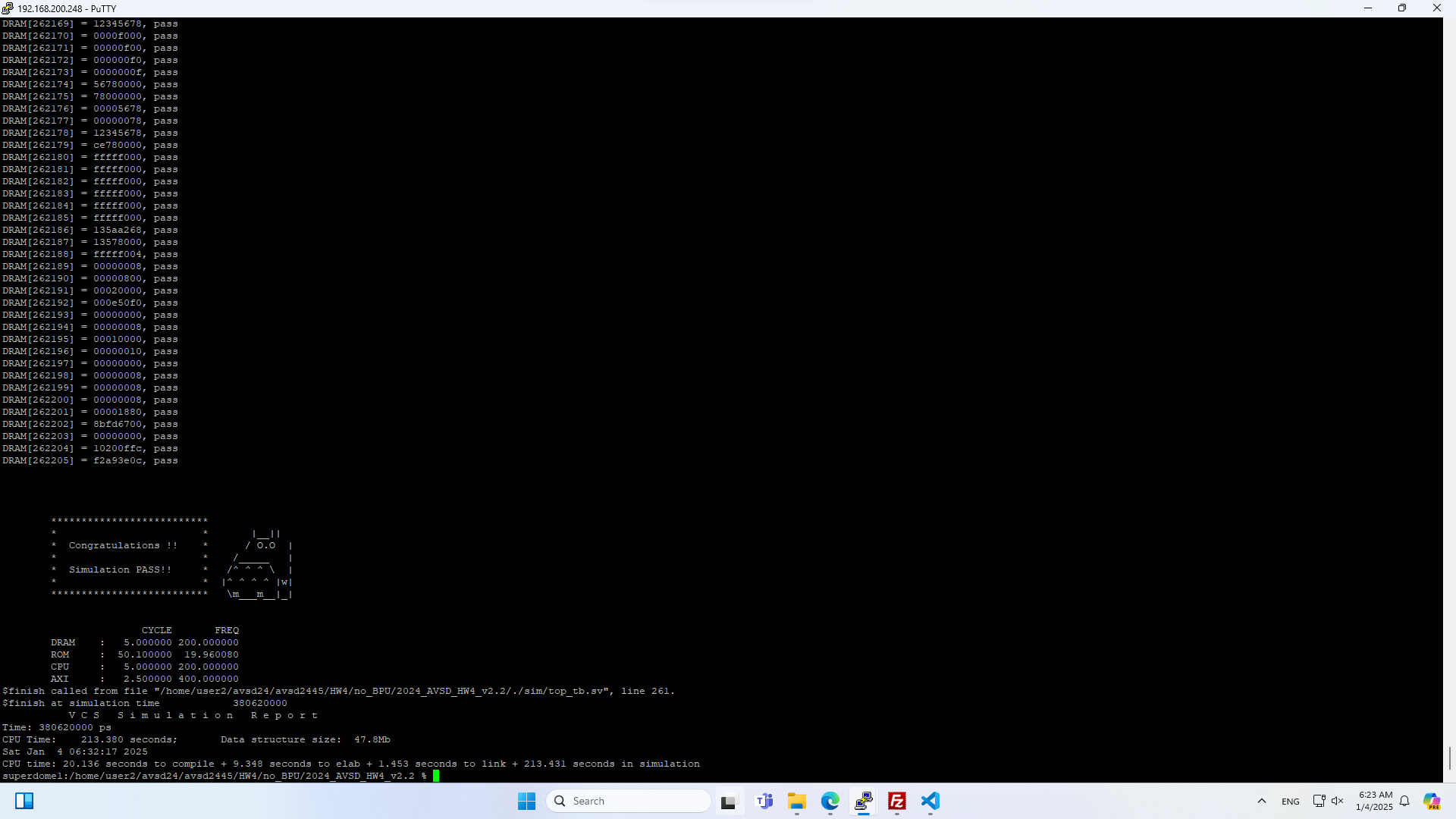
rtl



syn

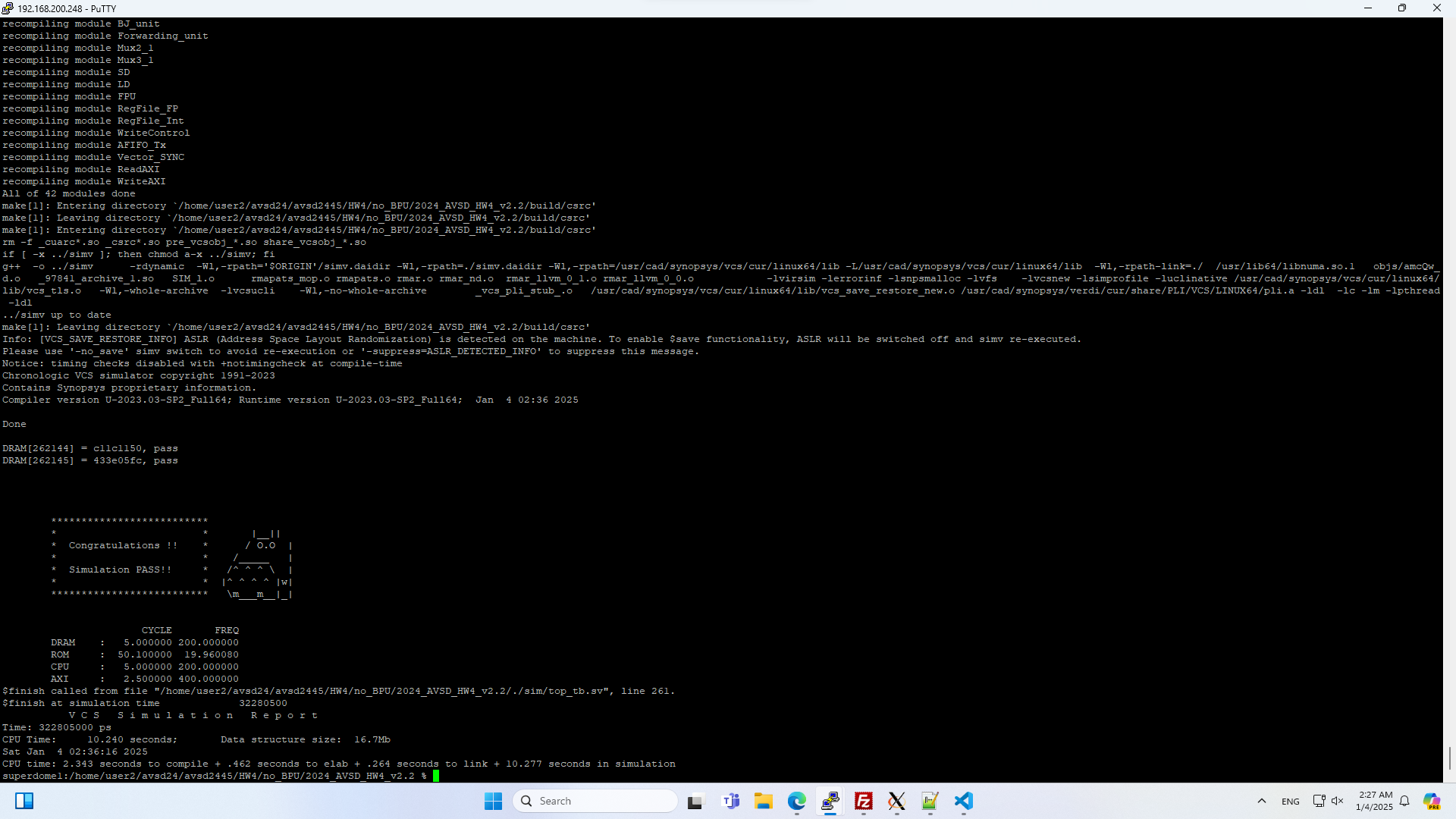


pr

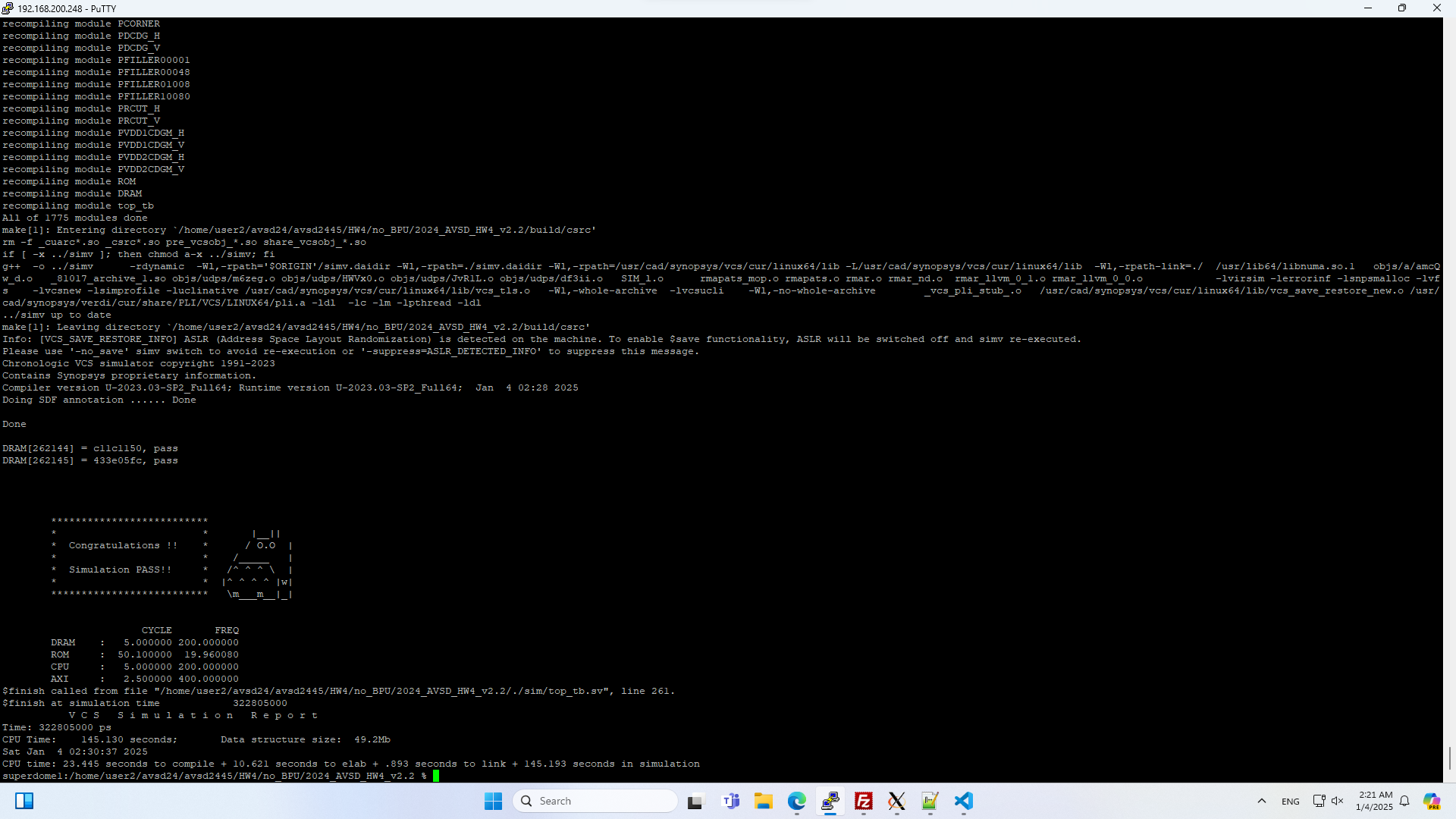


## Prog1 pass

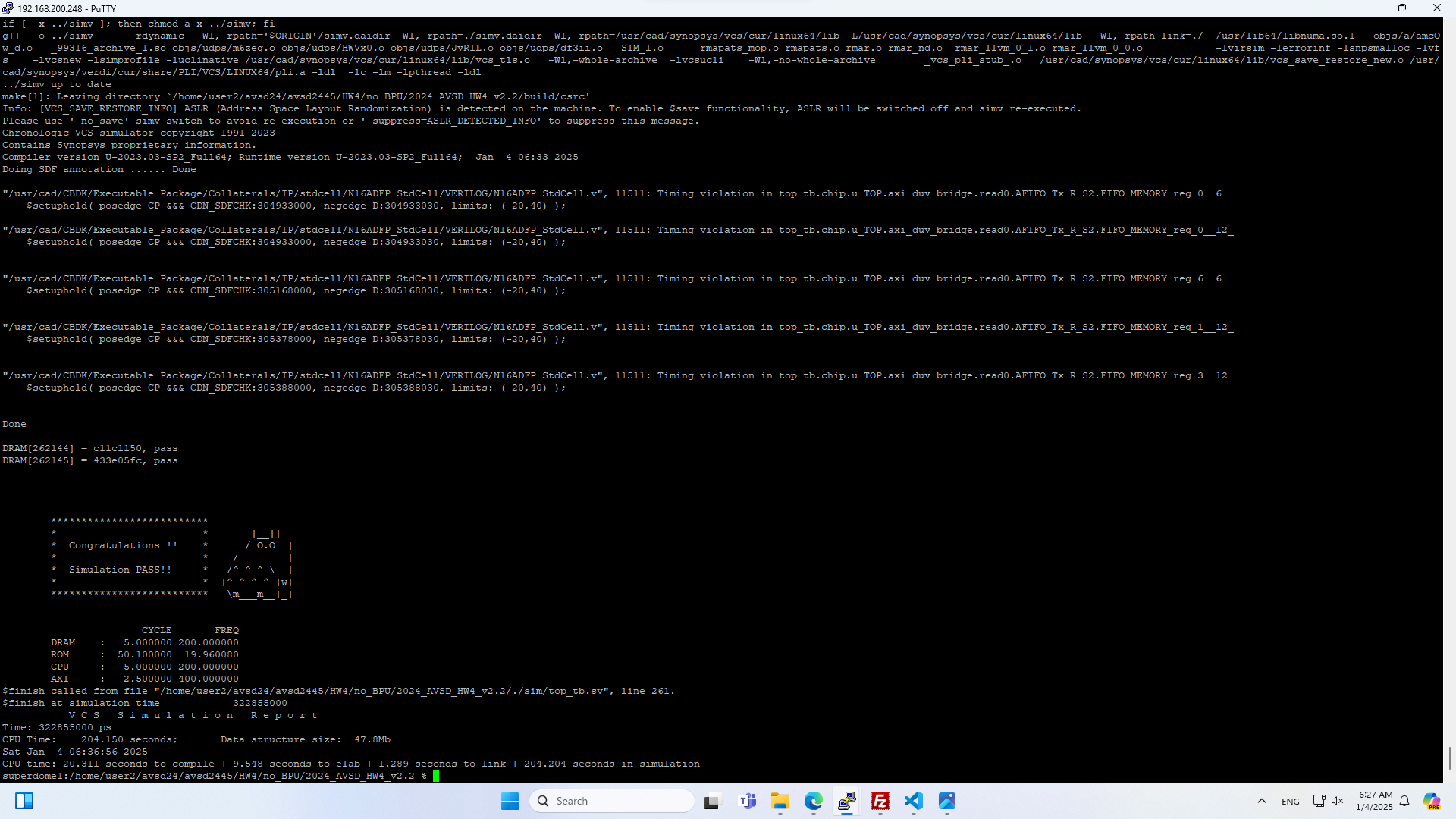
rtl



syn

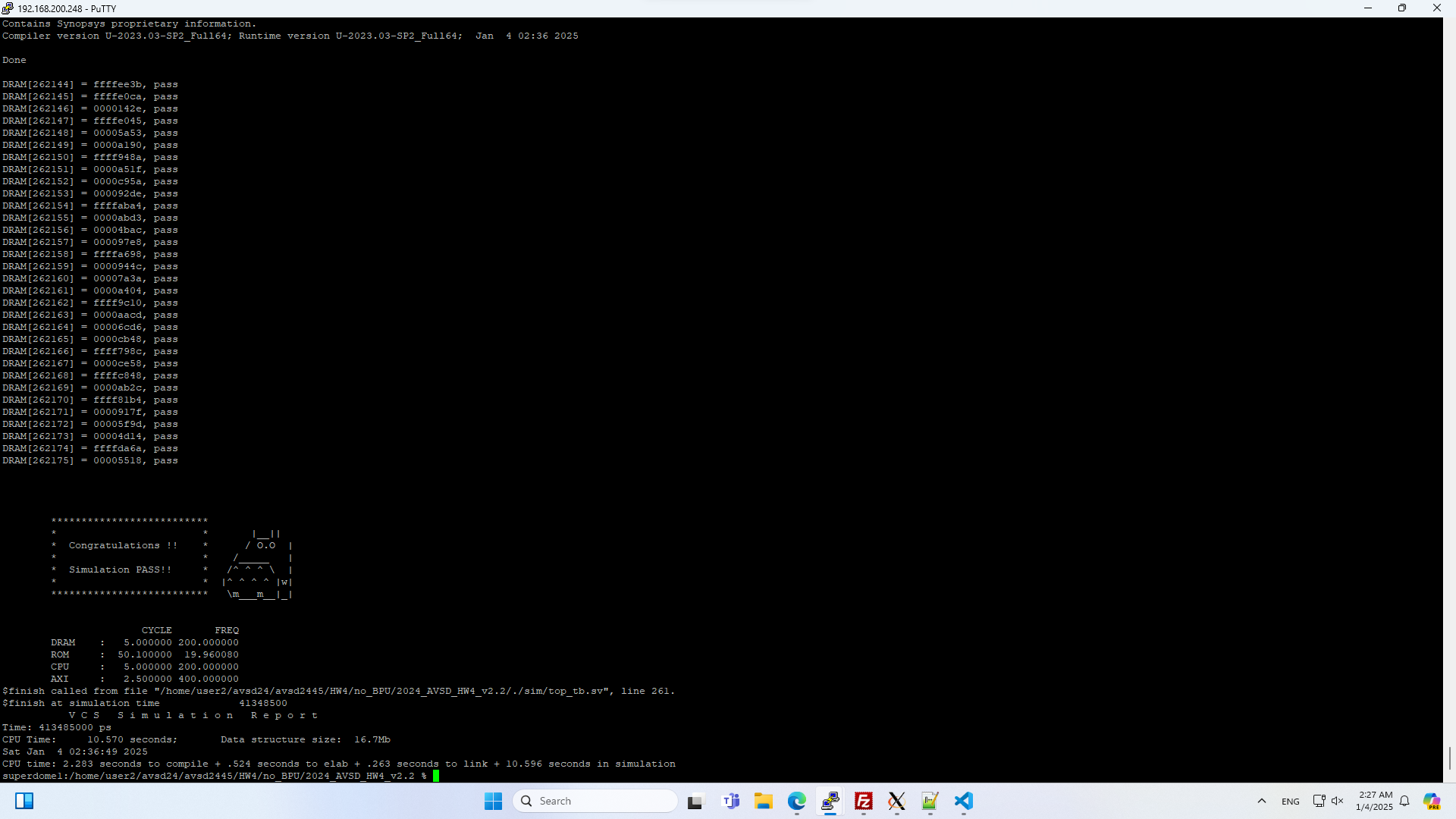


pr

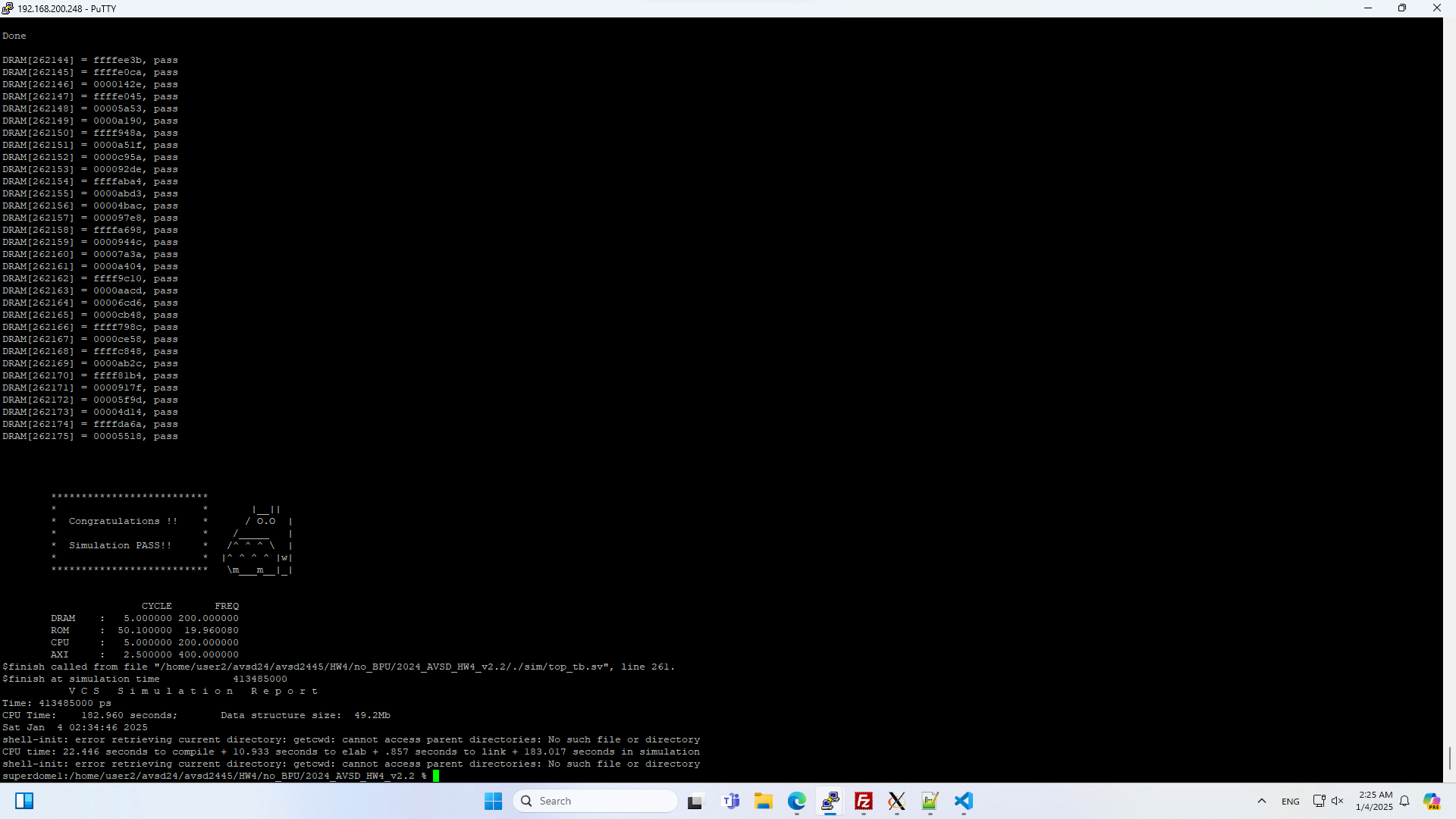


## Prog2 pass

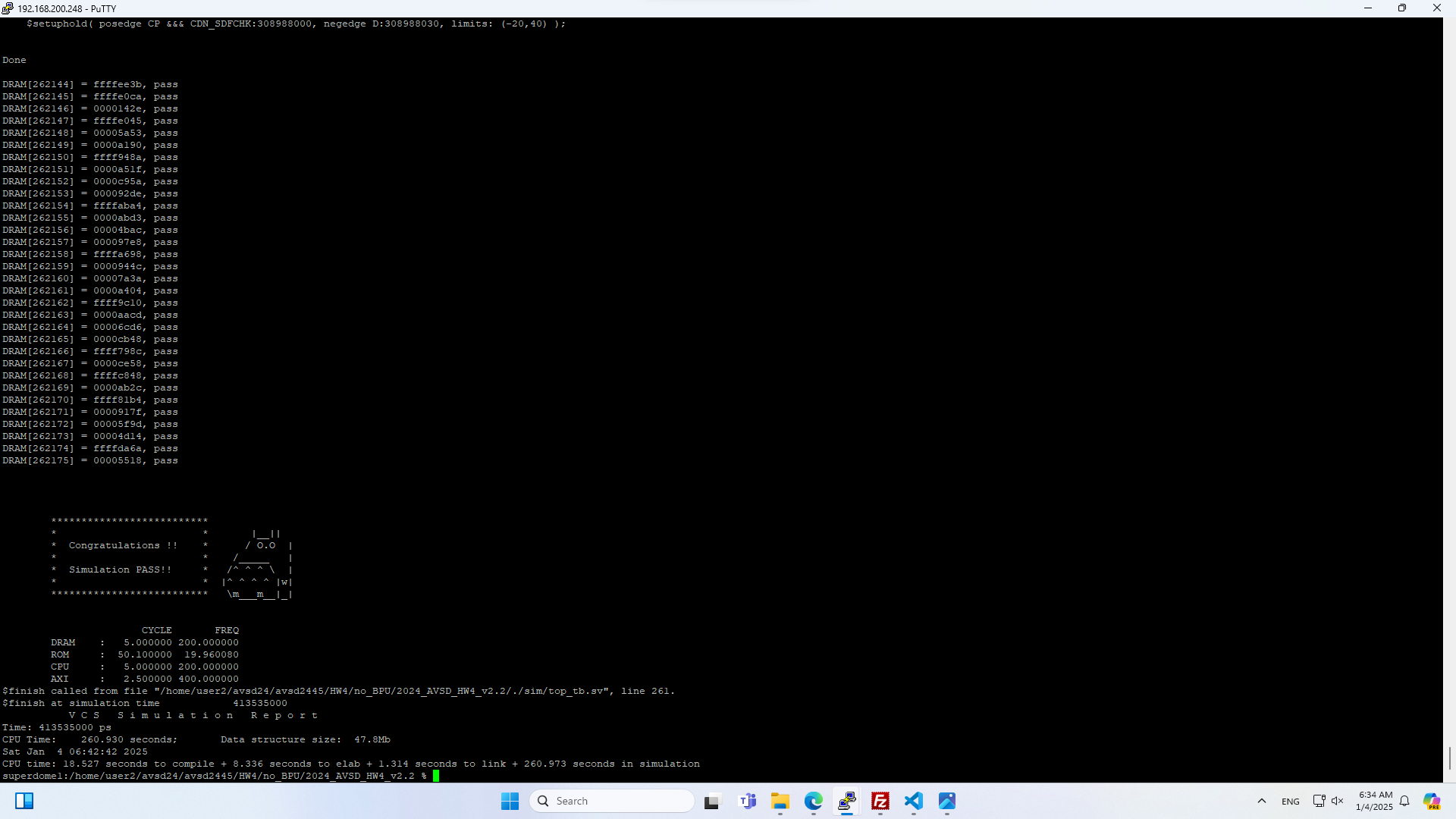
rtl



syn

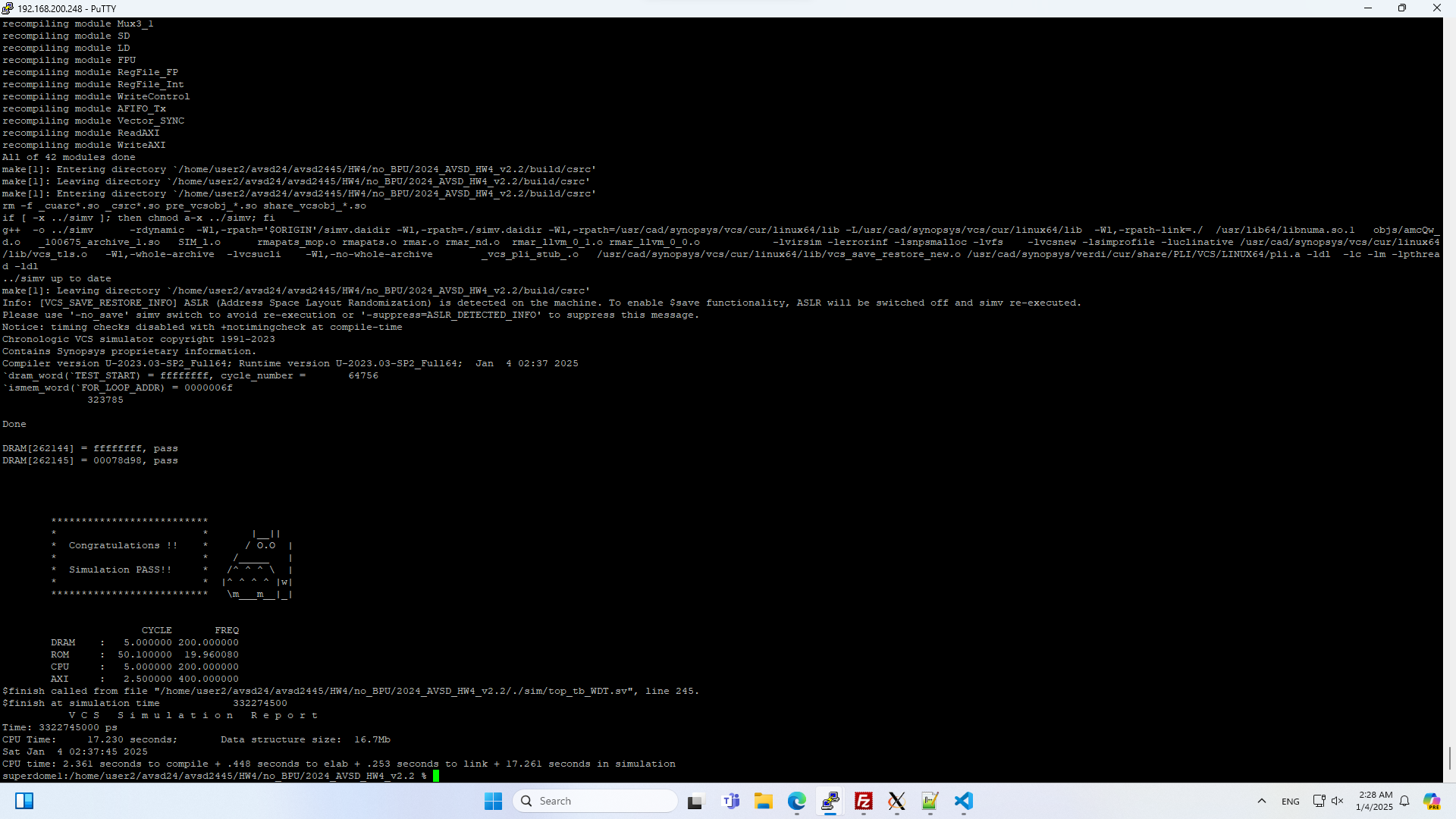


pr

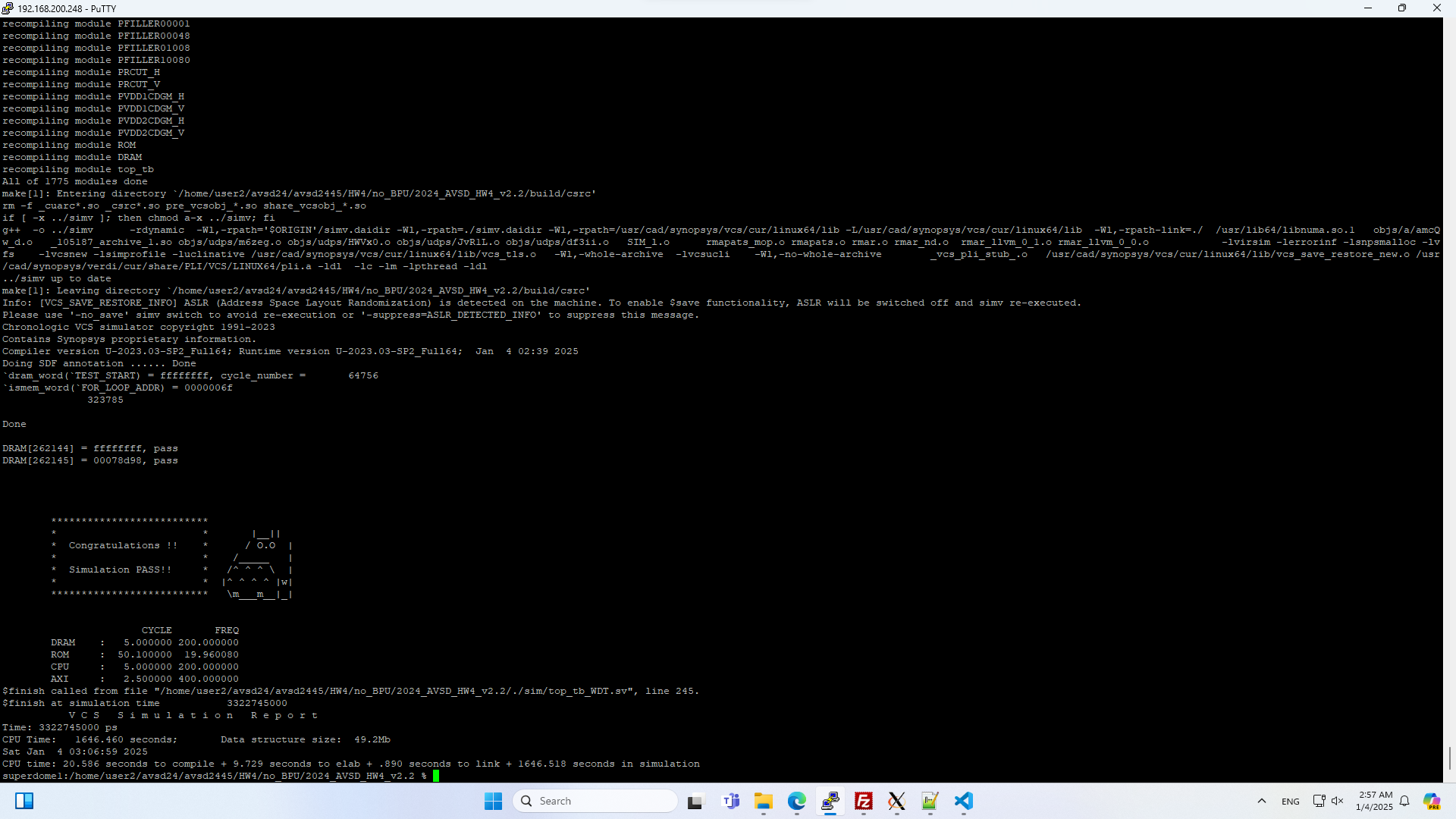


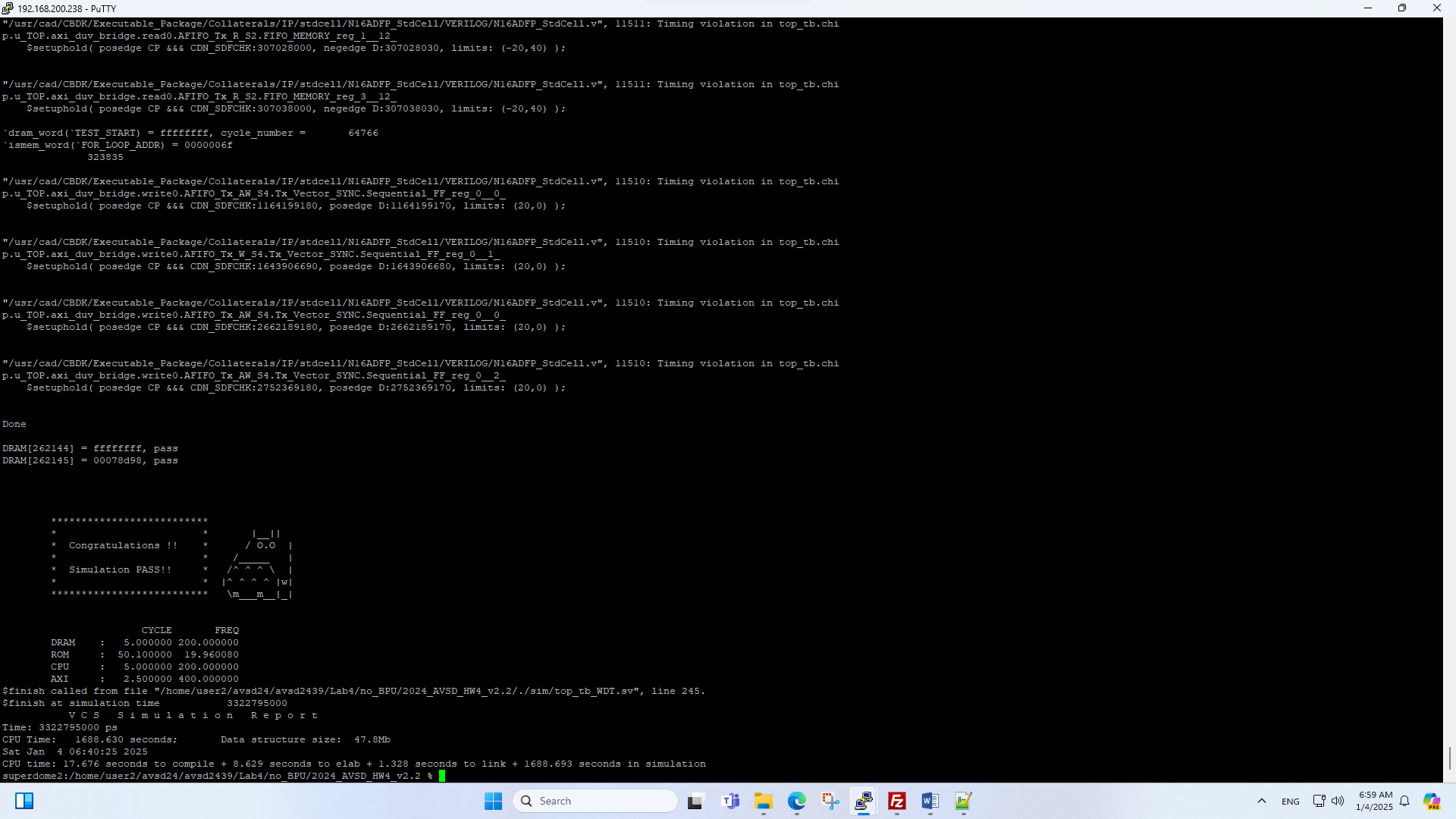
## Prog3 pass

rtl



syn





## Cache Hit Rate

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Hit Rate | Prog0 | Prog1 | Prog2 | Prog3 |
| Instruction | 98.5% | 99.5% | 99.5% | 99.9% |
| Data | 89.4% | 90.2% | 98.9% | 99.9% |

當計算出這個hit rate的時候覺得很不合理，不應該那麼高，但回頭看波型圖時發現到，由於每個prog中間fill block都佔很大一部分，所以導致cache一直hit，因此這份數據我認為還是有參考性。

Prog0:



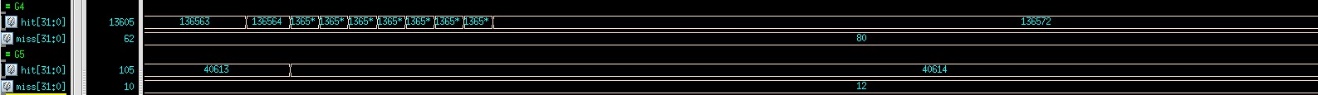
Prog1:

****

Prog2:



Prog3:



紅框與橘框分別為Inst Cache及Data Cache的hit與miss數。

# Spyglass CDC check

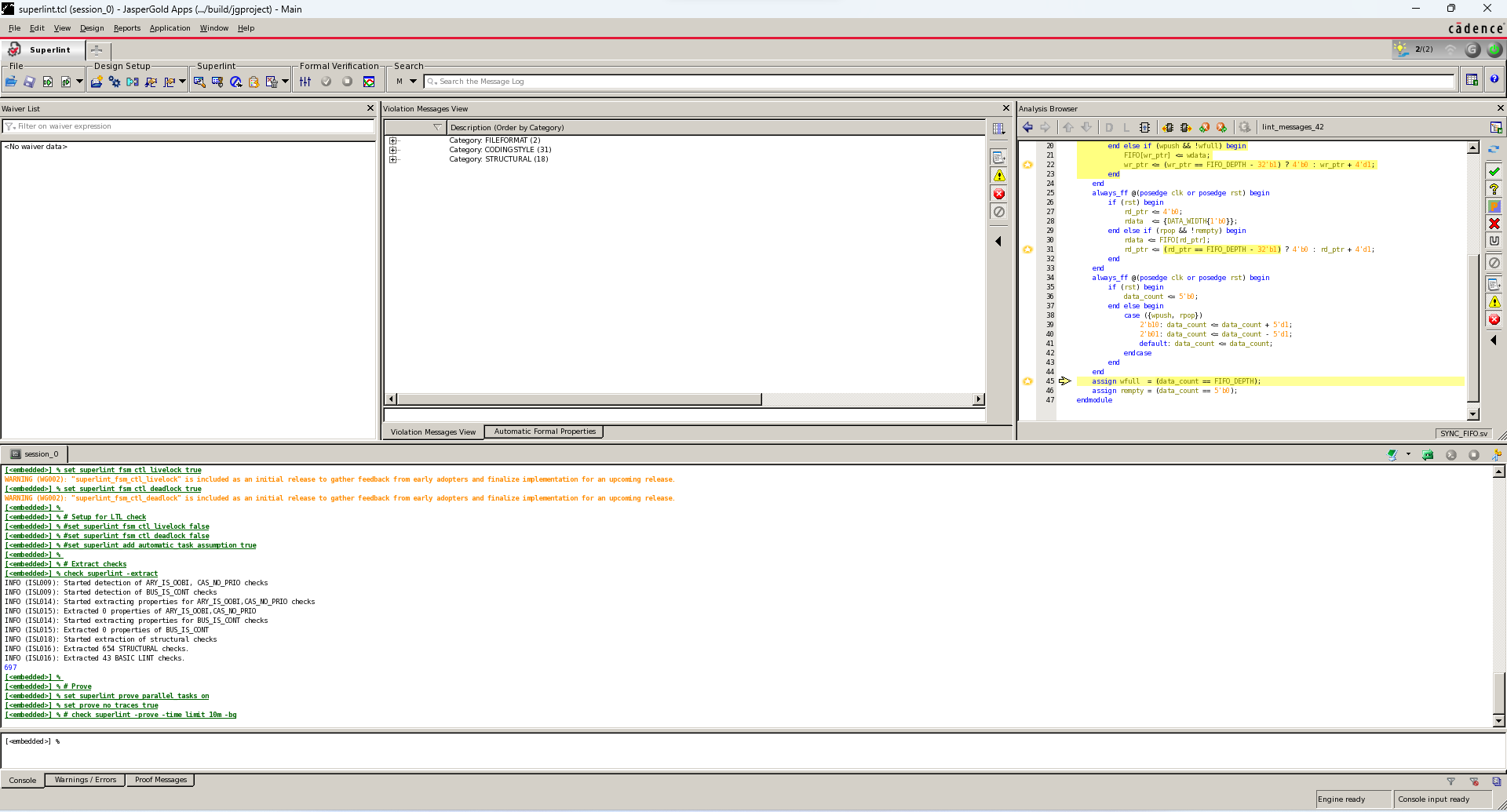
# 一張含有 文字, 螢幕擷取畫面, 陳列, 軟體 的圖片 自動產生的描述

# 一張含有 文字, 螢幕擷取畫面, 軟體, 陳列 的圖片 自動產生的描述

# 我們的 CDC 採用了 AFIFO 架構，經過 Spyglass 的 CDC 分析後，結果顯示僅有 warning，並未發現 error。這些 warning 可能是由於 AFIFO 的結構與週邊訊號較為複雜，使得工具無法全面檢測潛在問題。然而，電路已達成部分驗證（partially proved），這說明設計具備足夠的穩定性，未導致 Spyglass 檢測出任何錯誤。

# Superlint check

## Coverage



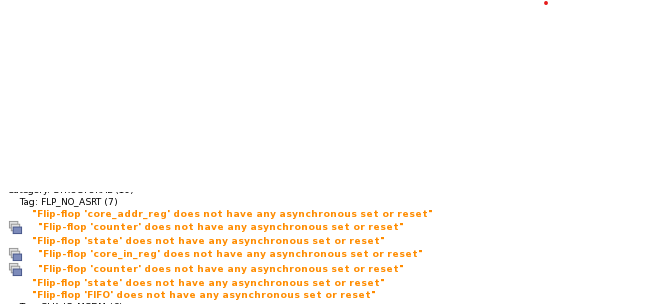
Total :16361, warning:51, ratio:99.9%

## 3 most frequent warning/errors

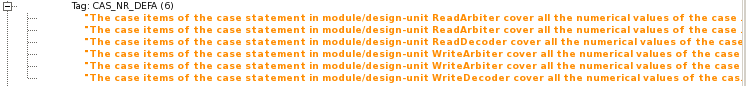
1.



2.



3.



在 Superlint 檢查中，我們的設計僅留下少數幾個 warning，其中大多數與使用 synchronous reset 有關。選擇 synchronous reset 是為了符合 Spyglass 的規範。然而，使用 synchronous reset 會讓 Superlint 無法準確檢測某些錯誤。因此，我們採取了先將 reset 暫時改為 asynchronous，執行 Superlint 檢查並修正錯誤後，再將 reset 改回 synchronous 的策略。

此外，針對 CAS\_NR\_DEFA 的問題，我們覺得這不會影響程式功能，並且是個好習慣，能確保case都有對應的default。

剩餘的 warning 幾乎都是與位元對齊有關的小問題，影響不大。

# ICC2 APR Layout flow

# 01\_design\_setup

先設定APR初始環境。圖片中為已經定義好的模組sram,TA和DA。

# 02\_design\_planning

這步是在為CHIP的placement做事前的設計準備，包括擺置IO腳位，定義間距和起始位置並插入填充單元等設置。最終完成Floorplan的布局。

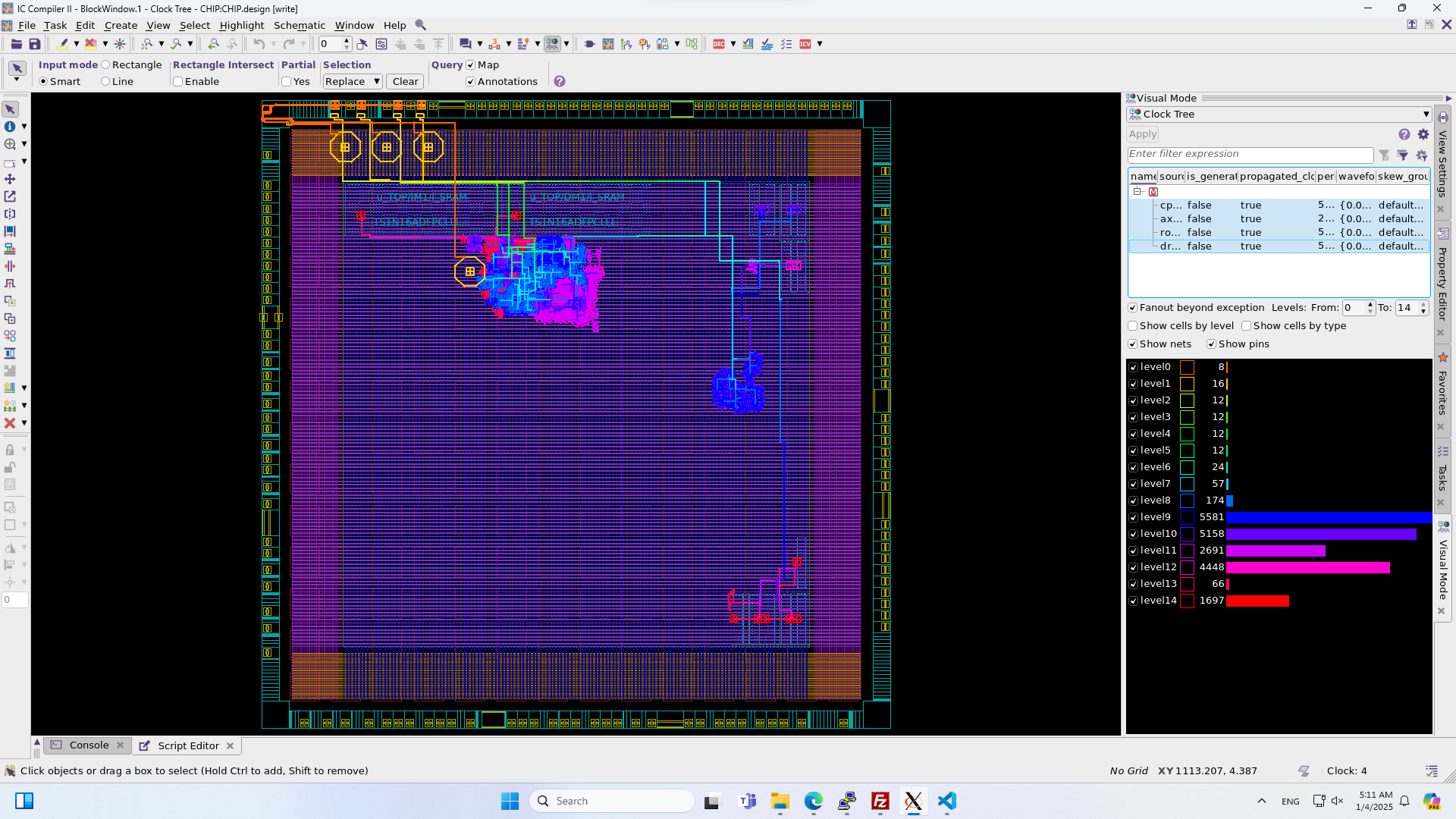
# 03\_placement

這主要在對CHIP的place優化，希望能夠完成整體布局，時序與功耗並保證設計符合規則。

# 

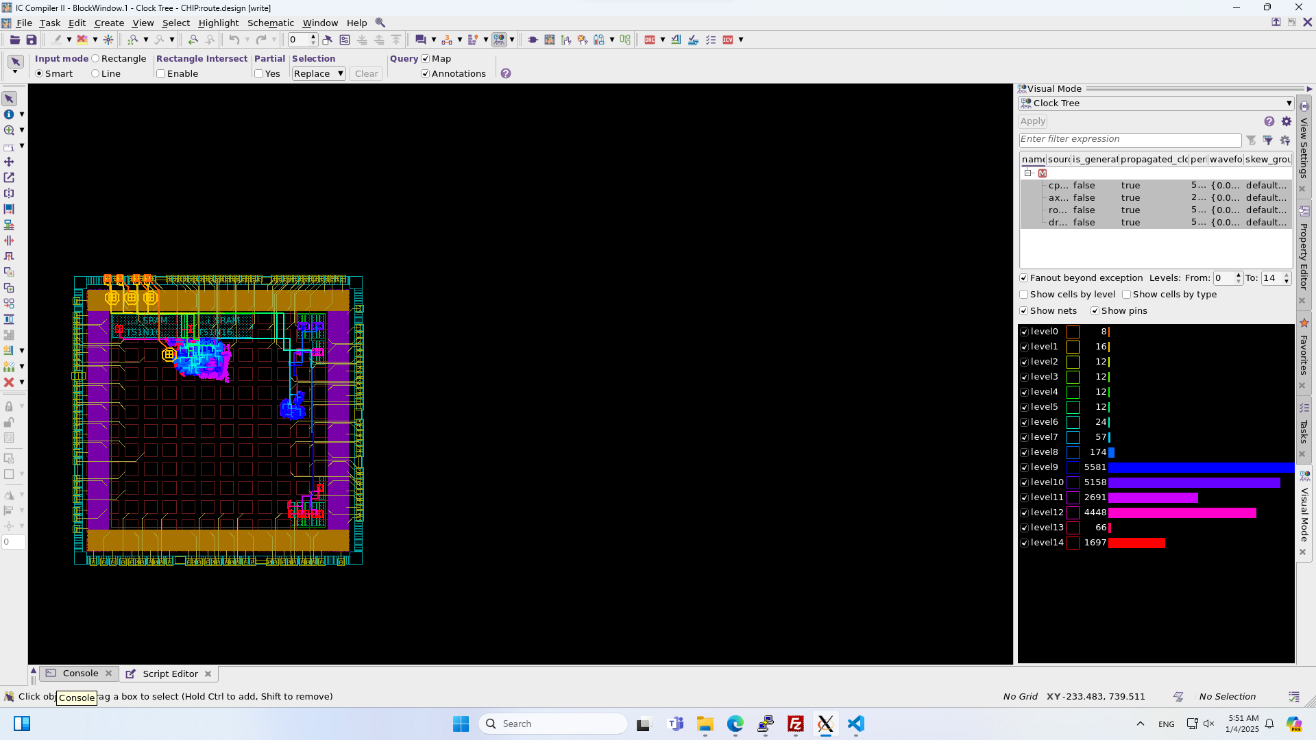
# 04\_cts

這步是生長clock tree，主要是要讓clk不會因為線路的長短而影響到clk的同步性。因此需要優化減少clk的delay。



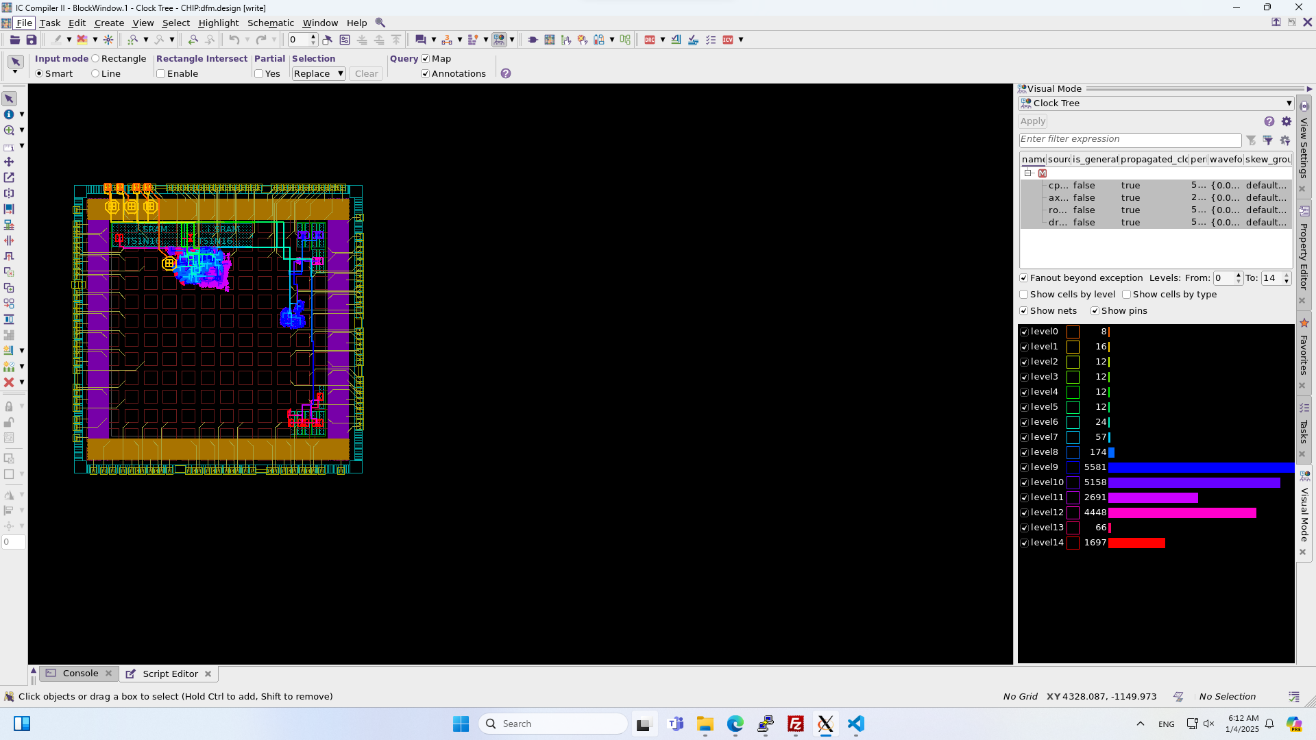
# 05\_route

這步在進行自動繞線，主要完成信號路徑布局優化和驗證。確保設計符合要求。



# 06\_dfm

這步在執行(Design for Manufaturability)，和最終驗證工作，包含填充，路徑檢查，時序分析，power檢查。



# 07\_streamout

這步在儲存前面的結果。

# 08\_outputfile

這步在輸出結果，完成SDF。

# Lessons Learned

# 余祐安:

# 這次主要負責AXI和FIFO的設計，主要要解決CDC的問題，透過AFIFO連接2個不同clock domain，以此確保資料的穩定性，了解到其中如何透過gray code的同步，利用指標來同步資料是蠻有趣的方法。在debug的時候發現會有資料不見，主要原因是之前的clk都是相同，所以所有元件同步，但這次因為時間不同，導致資料會丟失，因此需在原先AXI的基礎上多保留狀態以確保資料能夠延續，直到狀態改變。同時，起初在設計AFIFO的時候指考慮pointer和如何判斷滿或空，但沒有注意到FIFO內的register在讀取結束後需要清0，導致指針回來時讀取到錯誤的資料，後來修改程式後順利運行。這四次作業紮實的學習如何撰寫SV code並且利用工具除錯，和組員和助教討論，十分熱心解決問題，希望FINAL可以順利完成。

# 王華昀:

# 這次的作業在HW3上新增cache及AFIFO，而我是負責設計cache部分，剛開始在理解cache的mapping方式及替換方法就花不少時間，把計結複習了一遍。在了解cache的原理後很快就把各個狀態訂好，然而各個狀態的行為需要非常謹慎的設計，一不小心就會讓cache無法運作。除此之外，當pre-sim通過後，post-sim一直無法通過，原因是2 way在比對TA時一直有unknown的產生，當下真的很挫折，因為只要每合成一次就需要花費兩個小時，並且結果還不確定是否正確。並且中間有好幾次因為工作站空間不足無法產生.v檔，好不容易解決cache unknown的問題後，又發現cpu內部在branch的時候有問題，好在問題不難解決，經過一波三折後才成功通過post-sim。這次作業中比較可惜的是在APR方面，由於所剩時間不足，因此在APR沒辦法好好理解各個sript的內容。這次的作業真的讓我學到很多，也花費超級多時間，非常謝謝助教們常常被我到實驗室打擾，以及組員祐安花費很多時間一起完成作業。

# Contribution

|  |  |  |
| --- | --- | --- |
| Name | 余祐安 | 王華昀 |
| Contribution percentage | 50% | 50% |
| Contribution parts | AXI/FIFO | CACHE |