VLSI System Design (Graduate Level)
Fall 2024

HOMEWORK III
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: _余祐安_ ， _王華昀__
Student ID: _N26134243_, _N26134308_

Summary

<table>
<tr><td colspan="5">Hardware</td></tr>
<tr><td colspan="3"></td><td>RTL</td><td>synthesis</td></tr>
<tr><td rowspan="8">Top</td><td rowspan="2">CPU_wrapper</td><td>CPU</td><td>✓</td><td>✓</td></tr>
<tr><td>New instructions</td><td>✓</td><td>✓</td></tr>
<tr><td colspan="2">SRAM_wrapper (IM & DM)</td><td>✓</td><td>✓</td></tr>
<tr><td colspan="2">ROM_wrapper</td><td>✓</td><td>✓</td></tr>
<tr><td colspan="2">DRAM_wrapper</td><td>✓</td><td>✓</td></tr>
<tr><td colspan="2">AXI</td><td>✓</td><td>✓</td></tr>
<tr><td colspan="2">DMA</td><td>✓</td><td>✓</td></tr>
<tr><td colspan="2">Watch Dog Timer</td><td>✓</td><td>✓</td></tr>
<tr><td colspan="5">Synthesis result</td></tr>
<tr><td colspan="3">Area</td><td colspan="2">Clock cycle(ns)</td></tr>
<tr><td colspan="3">23372</td><td colspan="2">1</td></tr>
<tr><td colspan="5">Firmware & Software</td></tr>
<tr><td></td><td>RTL pass</td><td>syn pass</td><td colspan="2">Execution time(ns)</td></tr>
<tr><td>Booting</td><td>PASS</td><td>PASS</td><td colspan="2">–</td></tr>
<tr><td>Prog 0</td><td>PASS</td><td>PASS</td><td colspan="2">110363</td></tr>
<tr><td>Prog 1</td><td>PASS</td><td>PASS</td><td colspan="2">369841</td></tr>
<tr><td>Prog 2</td><td>PASS</td><td>PASS</td><td colspan="2">2694809</td></tr>
<tr><td>Prog 3</td><td>PASS</td><td>PASS</td><td colspan="2">1350718</td></tr>
<tr><td>Prog 4</td><td>PASS</td><td>PASS</td><td colspan="2">1352848</td></tr>
<tr><td>Prog 5</td><td>PASS</td><td>PASS</td><td colspan="2">55277</td></tr>
<tr><td colspan="5">Spyglass summary(number of inline messages)</td></tr>
<tr><td>Information</td><td>Warning</td><td>Error</td><td colspan="2">Fatal</td></tr>
<tr><td>221</td><td>2</td><td>0</td><td colspan="2">0</td></tr>
<tr><td colspan="5">Superlint(number of inline messages)</td></tr>
<tr><td>Total lines</td><td>Warning</td><td>Error</td><td colspan="2">coverage(%)</td></tr>
<tr><td>13651</td><td>6</td><td>0</td><td colspan="2">99.99%</td></tr>
</table>

Contribution

余祐安 50%	王華昀 50%
CPU、AXI、ROM、DRAM、DMA	CPU、AXI、CSR、WDT、DMA

Hardware Design Description

- System Block Diagram

本次作業中包含3個Master端及6個Slave端，Master分別為CPU(M0、M1)、DMA(M2)，Slave 分別為 ROM(S0)、IM(S1)、DM(S2)、DMA(S3)、WDT(S4)、DRAM(S5)，系統架構圖如下:

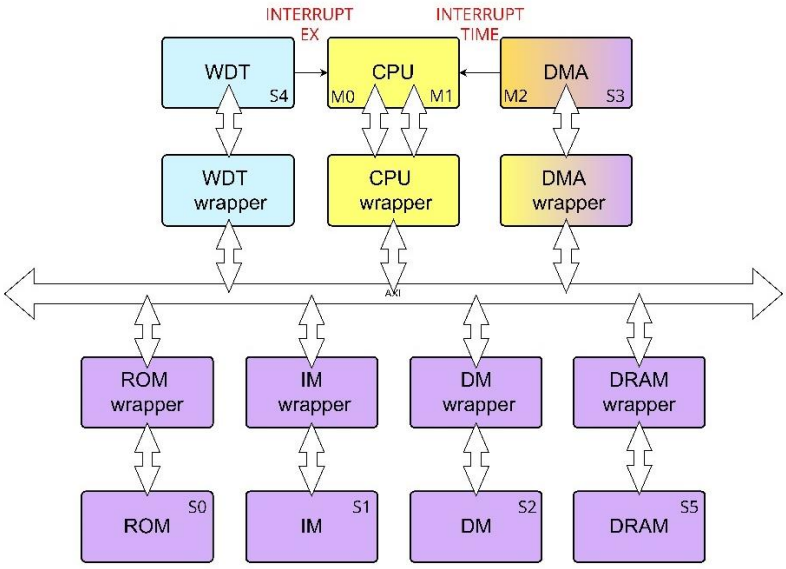


圖 1 系統架構圖

這次的作業與 HW2 相比多加上 ROM 用來儲存 boot 程式，並且使用 DRAM 儲存 instruction 及 data，在 Booting 時會使用 DMA 來將 DRAM 中的資料搬運到 IM 及 DM 中。值得注意的是，系統中 DMA 同時作為 Master 及 Slave 端，以及 WDT 的 clk period 與其他的 clk period 不同。

除此之外，為了處理 Interrupt 的發生，CPU 新增 8 個指令，包括對 CSR register 進行讀寫的 CSRRW、CSRRS、CSRRC、CSRRWI、CSRRSI、CSRRCI，以及應對 Interrupt 的 MRET、WFI。

- Interrupt mechanism description and flow chart

本次作業中在 CPU 內增加一個 Interrupt controller 透過 FSM 來進行 Interrupt 控制。Interrupt 情形主要分為兩種，第一種為 DMA 造成的 External Interrupt，會讓 FSM state 從 IDLE 變為 TRAP，第二種為 WDT 造成的 Timer Interrupt，會讓 FSM state 從 IDLE 變為 TIMEOUT；除此之外，當 CPU 發出 WFI 指令時，會將整個 CPU 暫停運作，等待 Interrupt 的發生。

下圖中說明 Interrupt controller FSM 各級的轉換條件，第一種情況為 DMA 傳送進 CPU 的 interrupt_DMA 跳起並且 CSR 指令中的 mstatus[3](mie) 及 mie[11](meie)為 1 時，說明外部中斷發生，FSM 跳入 TRAP，Interrupt controller 會發出 MEIP_en 訊號通知 PC 及各級暫存器外部中斷發生，需要清除暫存資料，並且將 pc 跳至 mtvec，直到外部中斷結束 CPU 執行到 MRET 指令時，Interrupt controller 會發出 MEIP_end 訊號通知 PC 及各級暫存器外部中斷結束，使 pc 返回 mepc，繼續執行中斷前的指令。第二中情況為 WDT 傳送進 CPU 的 interrupt_WDT 跳起並且 CSR 指令中的 mstatus[3](mie)及 mie[7](mtie)為 1 時，說明時間中斷發生，FSM 跳入 TIMEOUT，Interrupt controller 會發出 MTIP_en 訊號通知 PC 及各級暫存器時間中斷發生，需要清除暫存資料，並直接重新將 CPU 開機，此時 pc 會跳至 mtvec，Interrupt controller 則會等待 interrupt_WDT 結束回到 IDLE。

在實作 TIMEOUT 發生的情形時，我們注意到一個現象，由於 next pc 的決定是由 REG_PC 前的一個 MUX 進行決定，因此會有 Interrupt_WDT 及系統內部的 branch 或 jump 同時發生的情況，當時我們並未注意到這個狀況，因此有時候碰巧在 Interrupt_WDT 結束時，branch 或 jump 訊號尚未落下，導致 pc 在被換為 mtvec 後，原先 next pc 應該要是 pc+4，但 next pc 變為 branch pc，使得系統陷入無限迴圈，因此我們在這次的作業中增加一個 BPU 來進行 pc 的預測，可以使整體 cycle 數變少的情況下，同時控制 pc 的判斷方式，讓 interrupt 發生的時候不會有上述情況發生，才能成功通過 prog3 及 prog4。

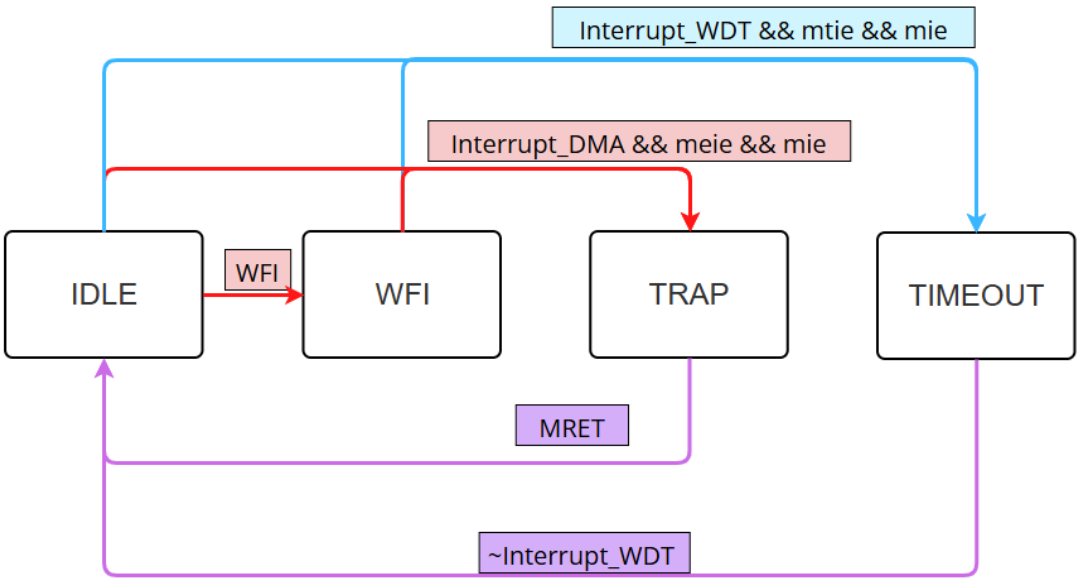


圖 2 Interrupt 機制 FSM 圖

- DRAM wrapper FSM chart

本次作業的程式與資料存放於 DRAM 中。由於 DRAM 的運作不同於 SRAM，資料存取需依序完成 Row Activate 和 Column Activate 操作。當發生 Row Miss 時，需先執行 Row Precharge 再激活新的 Row，以保證正確性。為此，我們設計了一個基於有限狀態機 (FSM) 的 DRAM Controller，負責控制 DRAM 的訪問流程與時序。

- **DRAM 的運作流程可分為以下三個階段：**

Precharge (PRE)：釋放當前激活的 Row。
Activate Row (ROW)：激活目標 Row。
Activate Column (COL)：激活目標 Column，完成資料讀取或寫入操作。

每個階段固定需要 5 個時鐘週期，因此存取延遲的理論值如下：
最佳情況 (Row Hit)：Row 地址匹配時，直接從 COL 狀態完成存取，僅需 5 個週期。
最差情況 (Row Miss)：需依序完成 Precharge -> Activate Row -> Activate Column，總計 15 個週期。

- **FSM 設計與控制邏輯：**

初始化：
控制器啟動時進入 PRE 狀態，執行 Precharge 操作。
狀態切換：
完成 Precharge 後，進入 ROW 狀態以激活目標 Row。隨後進入 COL 狀態，對目標 Column 進行資料操作。
動態監控：
持續監控 AXI 介面的 ARVALID (讀請求有效) 與 AWVALID (寫請求有效) 信號，觸發相應的狀態切換。若目標 Row 地址匹配 (Row Hit)，直接從 ROW 切換至 COL；若不匹配 (Row Miss)，則返回 PRE 狀態重新 Precharge，並激活新的 Row。

- **讀寫控制邏輯:**

讀取 (Read)：
根據 AXI 的 Valid 信號，準確控制 RVALID 信號，確保 AXI Master 正確接收 RDATA。
寫入 (Write)：
使用 3-bit 計數器進行資料控制，生成 BRESP 信號，確保 AXI Master 與 Bridge 間的同步。

- **設計中的特殊考量:**

信號控制：
精確管理 RAS (Row Address Strobe)、CAS (Column Address Strobe)、WEB (Write Enable Bar) 等 DRAM 控制信號，避免存取違規 (Violation)。
同步性與效率：

FSM 的操作完全與 AXI 協議同步，以實現穩定、高效的資料傳輸。

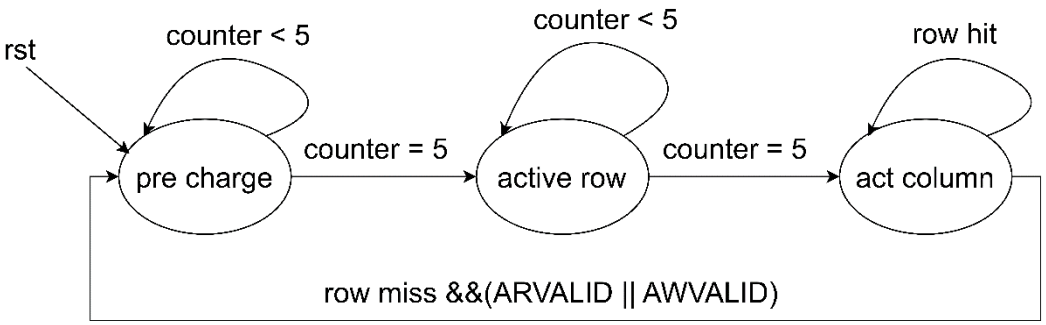


圖 3 DRAM FSM 圖

- WDT & CDC circuit description and diagram

本次作業我們使用常見的 double flip-flop synchronizer，原先我們在 clk 快到慢時使用的是 MUX synchronizer，透過分為 data enable 訊號及 data bus 訊號進行同步處理，架構圖如下圖所示:

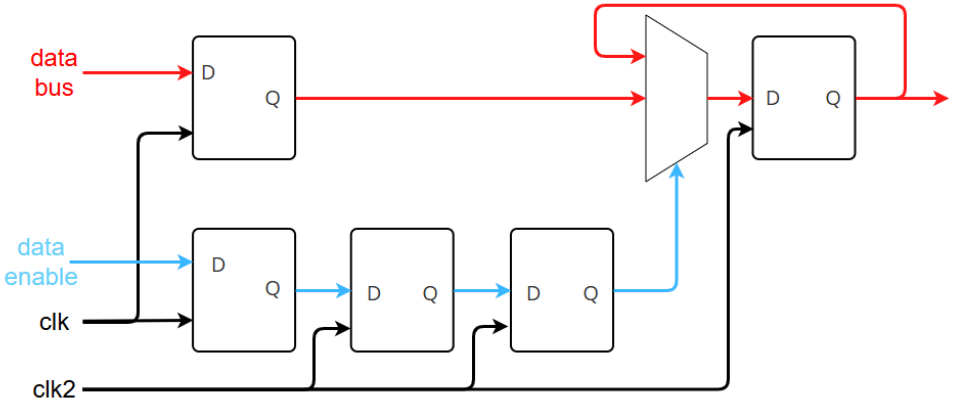


圖 4 CDC 電路圖

在邏輯上此架構與 double flip-flop synchronizer 相同，下方使用 double flip-flop synchronizer 控制 enable 訊號，在用 mux 來控制 data bus 的同步，使用此架構的好處是可以透過調整 data bus 的 bit 數來實現多 bit 的同步，然而在測試 spyglass 時，在 databus 進入第一個 flip-flop 後至進入 mux 前會一直有 clock domain crossing error 產生，說明這段訊號並不穩定，但從設計的角度來看這段訊號原本就不會穩定，因此才需要 mux 來控制 data bus 的同步，因此在不斷嘗試後調整為一般的 double flip-flop synchronizer，但現在再次思考這個架構，或許應該要把 enable 訊號再拉到第一個 data bus 前，並同樣與 data bus out 用一個 mux 多一層過濾，就可以避免掉中間有不穩定的情況發生。但若是未來要針對更大的系統進行 CDC 處理的話，應該還是要使用 aFIFO 就不會有不穩定的情況發生。

而 WDT 的訊號 WDEN、WDLIVE、WTOCNT 則是透過地址進行分類，將 WDATA 分配至個別的暫存器中，當 WDT 內部 counter 計算大於 WTOCNT 並且 WDEN 為 1 時，WTO 訊號拉起，透過 double flip-flop synchronizer 從 clk 慢到快進行同步，再傳送至 CPU 內部，告知時間中斷發生。

Software & Firmware design description

- Prog 1

  在本次作業中的排序法，使用 quick sort，不一樣的地方是在於 short 的資料型態變為 half word，也就是說 CPU 在運算時會將 32bit 的資料拆掉來看，與前幾次作業有些許差異，起初 load data 的 filter 並沒有針對 byte 位置進行對齊，導致比較的資料型態有錯誤，最後修改了 load 資料的 filter 和 store filter 一樣需要對齊地址後得以解決。

- Prog 2

  Prog2 的目標是將圖片轉換為灰階影像。此過程與 Prog1 和 lb 在處理上的邏輯相似，但在演算法上進行了進一步優化。
  如果當前處理的像素中，三個分量 (R、G、B) 的值相等，則直接將目標區域的對應分量都設為該相等值，達成灰階化效果。
  若三個分量的值不相等，則使用以下加權平均公式計算新的灰度值：

  $$Gray=0.3\times R+0.59\times G+0.11\times B$$

  計算後，將目標區域的 R、G、B 分量全部設為該灰度值。

- Booting

  Boot 這個程式會儲存在 ROM，一開始 CPU 會讀取 ROM 的中的 Boot 程式將 Instruction、sdata、data 等資料從 DRAM 搬到 IM 和 DM。這個程式我們使用 3 個 While Loop 搭配_dram_i_end、__sdata_end 和__data_end 來完成 Instruction、sdata、data 等資料的搬運。其中，在搬移資料時，會透過 DMA 來輔助。首先，CPU 會傳送給 DMA 搬移資料的起始地址和終點地址，DMA 會發出 interrupt 使得 CPU 進入 WFI 狀態，DMA 快速把資料從 DRAM 搬入 IM 和 DM，搬移完成後，CPU 會把 DMAEN 拉下，此時 DMA 會關閉 interrupt，讓 PC 回到正常狀態開始執行程序。

Screen shot of wave forms and simulation results

- **CSRRW**
  - **Mstatus**

    當 CSRRW 讀取 CSR 中 mstatus 的值，寫到 rd 中。並將 rs1 寫入 CSR mstatus。由於本次作業並不會每一個 mstatus bit 都使用到，因此同樣只記錄有用的 bit，分別為 mstatus[3]為 mie、mstatus[7]為 mpie、mstatus[12:11]為 mpp，紀錄結果為 8。

  - **Mie**

    當 CSRRW 讀取 CSR 中 mie 的值，寫到 rd 中。並將 rs1 寫入 CSR mie，值得注意的是，由於這次作業中的 spec 並不會運用到全部的 CSR bit，因此只取有作用的 bit 做紀錄，mie 為 mstatus[3]。

  - **Mip**

    當 CSRRW 讀取 CSR 中 mip 的值，寫到 rd 中。並將 rs1 寫入 CSR mip。與 mie 相同，這次作業的 spec 只會用到 mip[11]的 MEIP、mip[7]的 MTIP，因此只對這兩個 bit 做紀錄。此波型圖剛好紀錄為 0。

  - **Mepc**

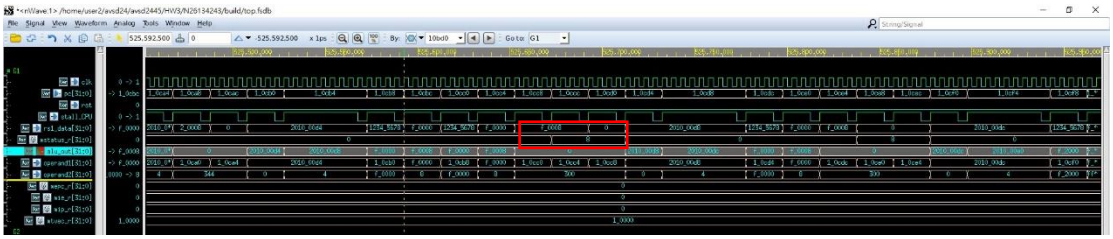
    當 CSRRW 讀取 CSR 中 mepc 的值，寫到 rd 中。並將 rs1 寫入 CSR mepc。Mepc 紀錄 interrupt 前運行到的 pc 位置，但下方情況為 prog0 interrupt 沒有發生，csr_imm 剛好為 MEPC，因此一樣是儲存 rs1 數值。

- **Mtvec**

  當 CSRRW 讀取 CSR 中 mtvec 的值，寫到 rd 中。並將 rs1 寫入 CSR mtvec。值得注意的是，在這次作業中並沒有將數值寫到 mtvec 裡，這是由於這次作業中在 Interrupt 發生時固定要進行 reboot 也就是說 pc 需跳至 0001_0000 重頭開始，因此這個數字是固定的，但還是有寫到 rd 的功能。

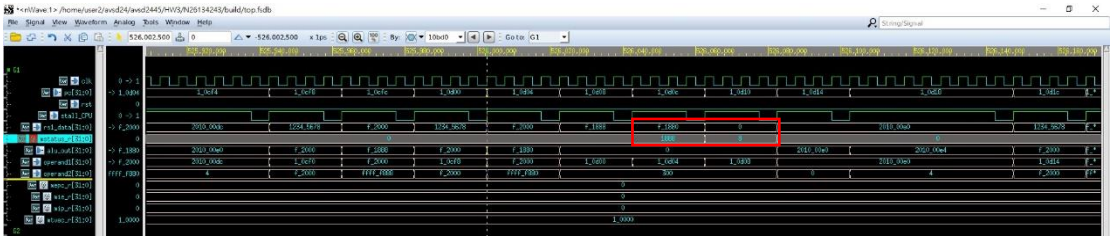
- **CSRRS**

  讀取 CSR 中的值，寫到 rd 中。再將 rs1 對應 bit 為 1 的值在 CSR 中設為 1。在圖中可以看到原先是 f_0008 而由於本次作業並不會每一個 mstatus bit 都使用到，因此同樣只記錄有用的 bit，分別為 mstatus[3]為 mie、mstatus[7]為 mpie、mstatus[12:11]為 mpp，紀錄結果為 8。



- **CSRRC**

  與 CSRRS 正好相反，讀取 CSR 中的值，寫到 rd 中。再將 rs1 對應 bit 為 1 的值在 CSR 中設為 0。同樣只記錄有意義之數值原數值 f_1880 紀錄結果為 8。

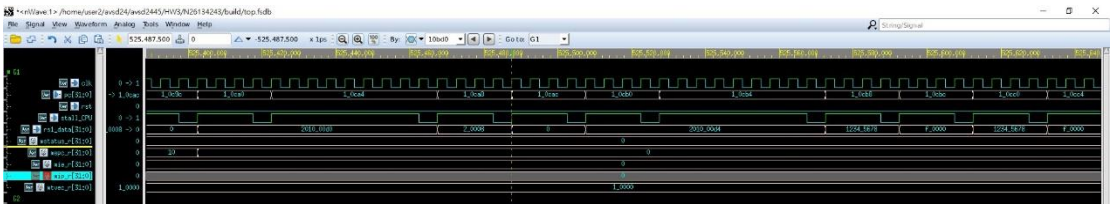


- **CSRRWI**

  與 CSRRW 類似，不同的是讀取 CSR 的值，寫到 rd 中。並將 uimm 寫入 CSR。以下分別為不同的寫入形況。同樣 mtvec 是固定的，不會被寫入但同樣會有讀取的效果，並且每一個都只取對這次作業中有意義的 bit 並不是全部存取。

- **mstatus**

- **mip**



- **mepc**

- **mtvec**

- **CSRRSI**

與 CSRRS 類似，不同的是讀取 CSR 中的值，寫到 rd 中。再將 uimm 對應 bit 為 1 的值在 CSR 中設為 1。同樣 mtvec 是固定的，不會被寫入但同樣會有讀取的效果，並且每一個都只取對這次作業中有意義的 bit 並不是全部存取。

- **CSRRCI**

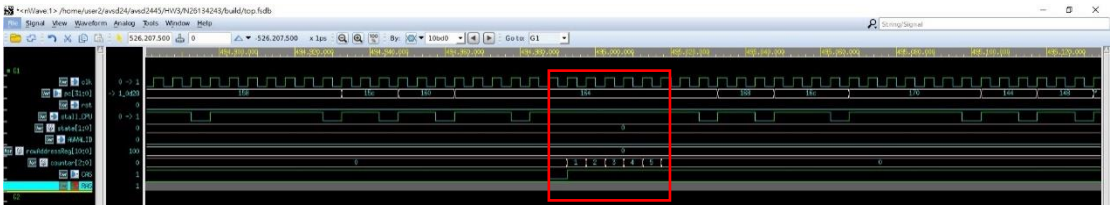
  與 CSRRC 類似，不同的是讀取 CSR 中的值，寫到 rd 中。再將 uimm 對應 bit 為 1 的值在 CSR 中設為 0。同樣 mtvec 是固定的，不會被寫入但同樣會有讀取的效果，並且每一個都只取對這次作業中有意義的 bit 並不是全部存取。

- **DRAM**
  - **Write row not hit**

    以寫入為例，若是發現 row address 和上一次使用的不同，則需要 15 個 cycle 才可完成操作，必須進行 precharge、row active 和 column active，由上圖可以發現 RAS、CAS 和 counter 隨著 AWVALID 成功運作，會將資料寫入到 DRAM。

  - **Read row hit**

    以讀取為例，若是發現 row address 和上一次使用的相同，則只需要 5 個 cycle 即可完成操作，由上圖可以發現 CAS 和 counter 隨著 ARVALID 成功運作，會將資料讀出 DRAM。

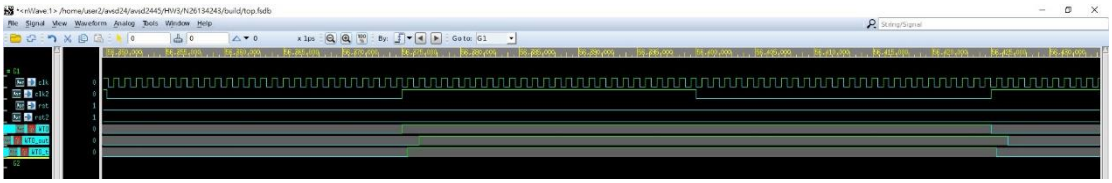


- **WDT**
  - **TIMEOUT CPU reboot**

    從下方波型圖可以看到當 WDT 發出 interrupt 訊號時，CPU pc 跳至 mtvec 進行 reboot，使 pc 跳回 0001_0000，並且 next pc 為 pc+4 重新開機。

- **CDC**

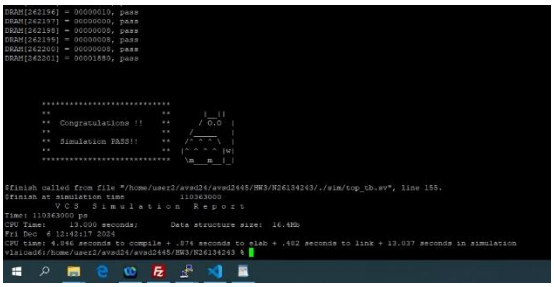
  下方波形圖為系統 CDC 的範例，其中 WTO 為 clk2 domain，經過一個 clk1 domain 的 flip-flop 變為 WTO_t，但此時訊號尚未穩定，因此還需要再經過一個 clk1 domain 的 flip-flop 變為 WTO_out 傳至 CPU 中。



Simulation Results

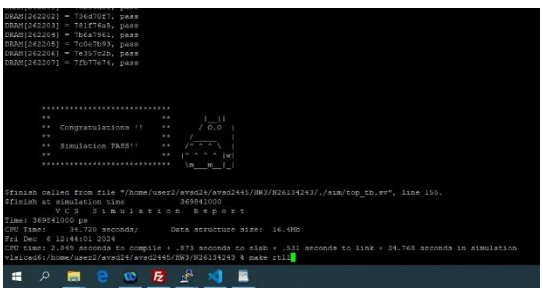
- **Prog0**

  Time : 110363000 ps



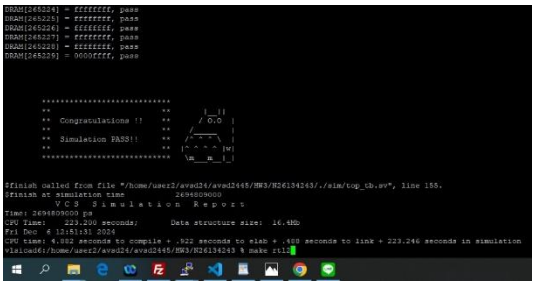
- **Prog1**

  Time : 369841000 ps



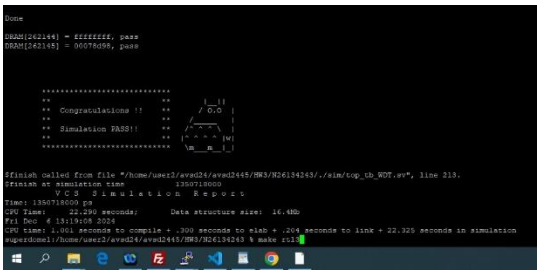
- **Prog2**

  Time : 2694809000 ps



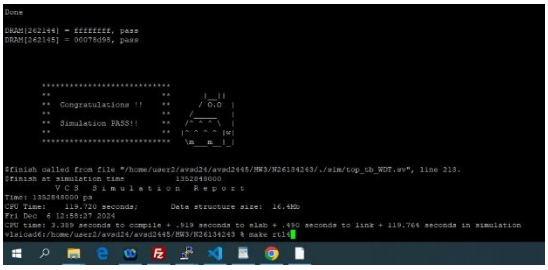
- **Prog3**

  Time : 1350718000 ps



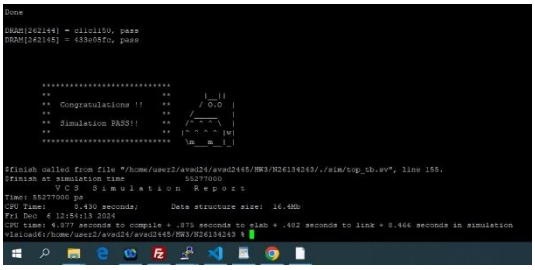
- **Prog4**

  Time : 1352848000 ps



- **Prog5**

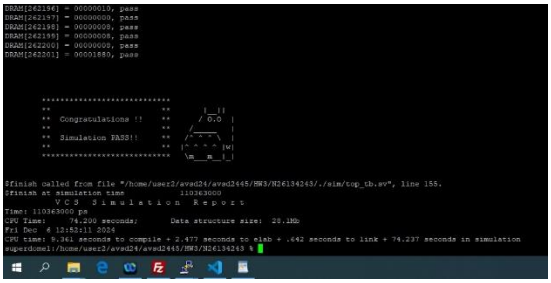
  Time : 55277000 ps



- # Synthesize Results

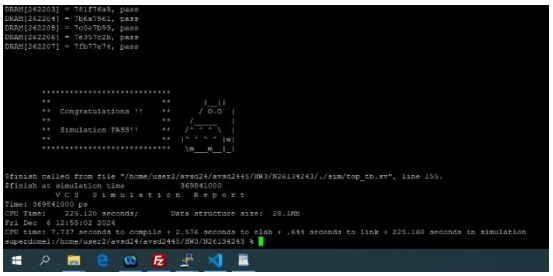
  - ## Syn0

    Time : 110363000 ps



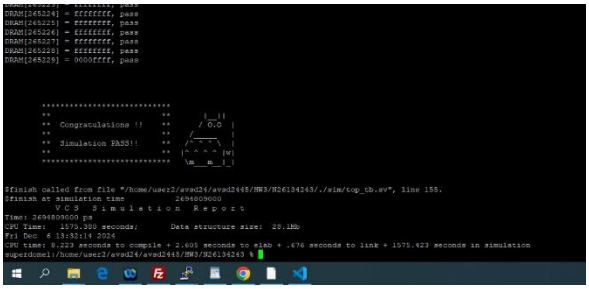
  - ## Syn1

    Time : 369841000 ps



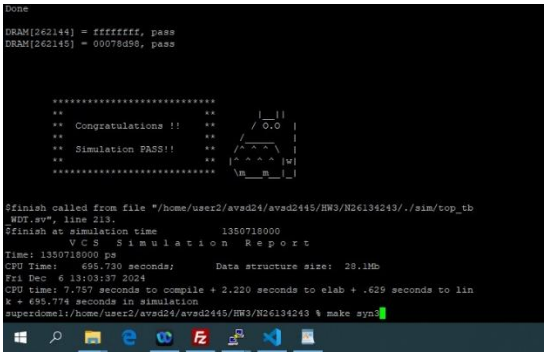
  - ## Syn2

    Time : 2694809000 ps



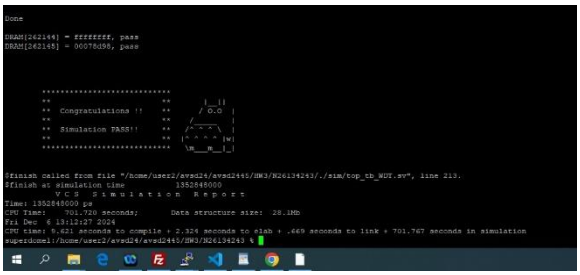
  - ## Syn3

    Time : 1350718000 ps



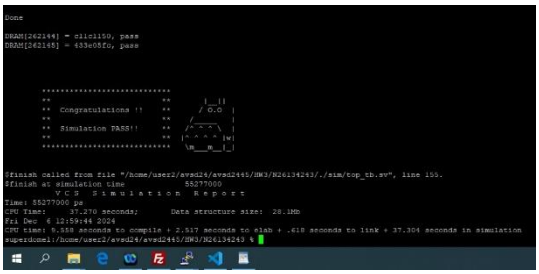
- **Syn4**

  Time : 1352848000 ps



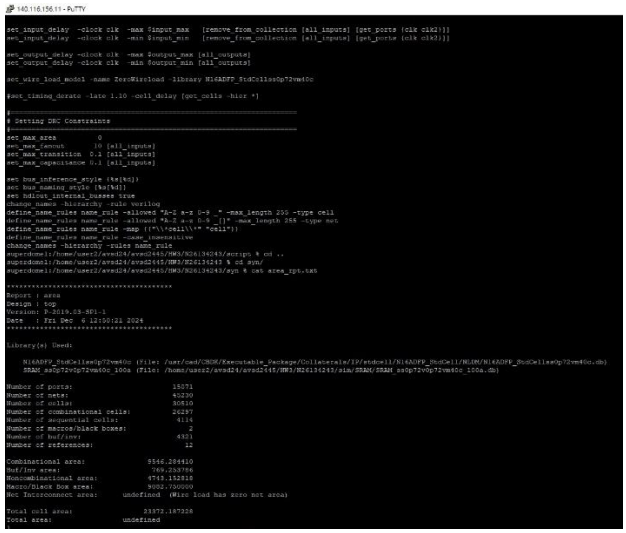
- **Syn5**

  Time : 55277000 ps

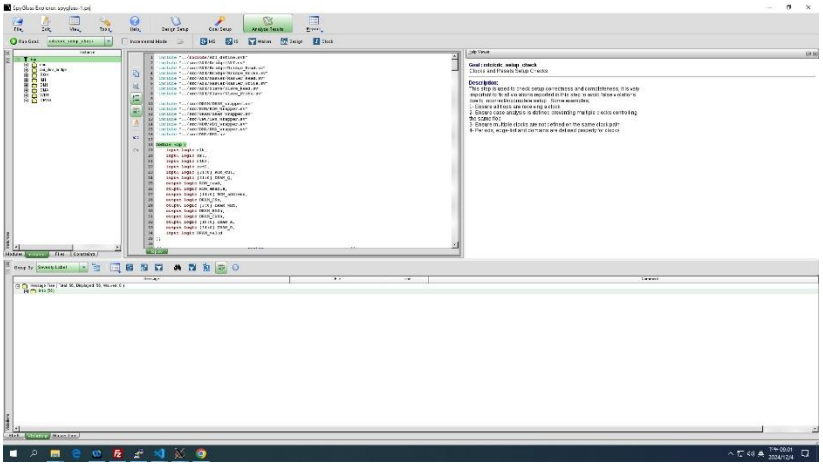


- # AREA REPORT

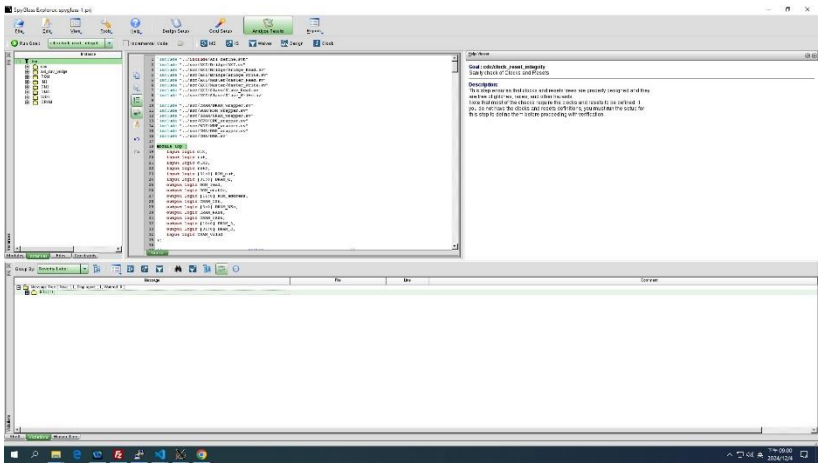
Total cell area : 23372



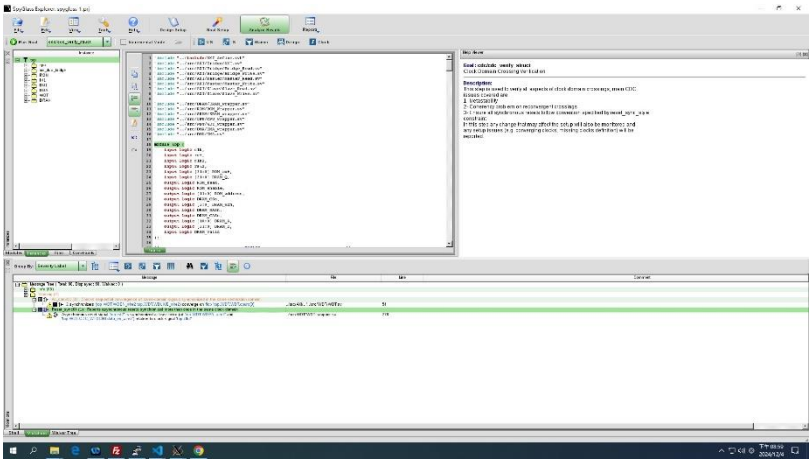
- # SPYGLASS
  - **Set up check**



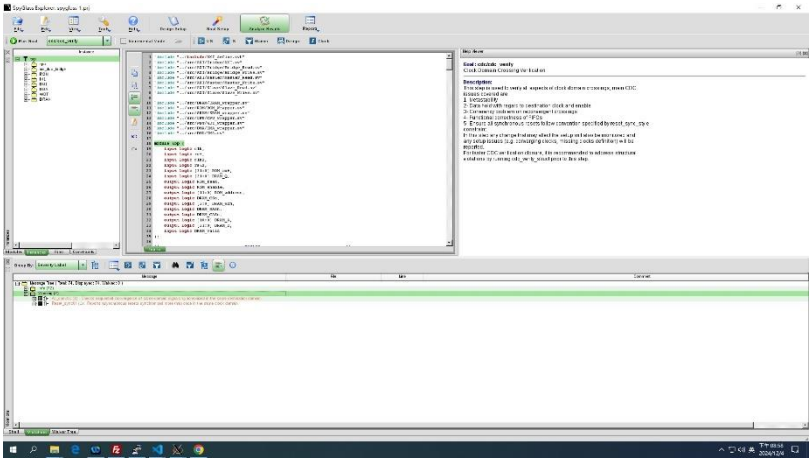
  - **Reset integrity**



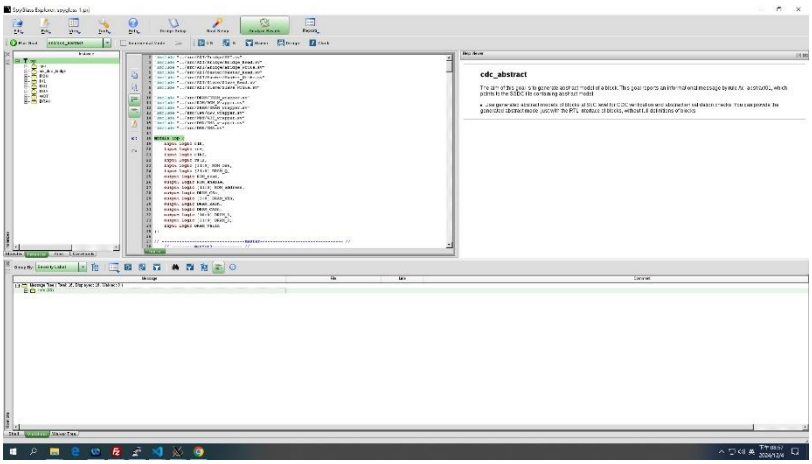
■ **Verify struct**



■ **Verify**

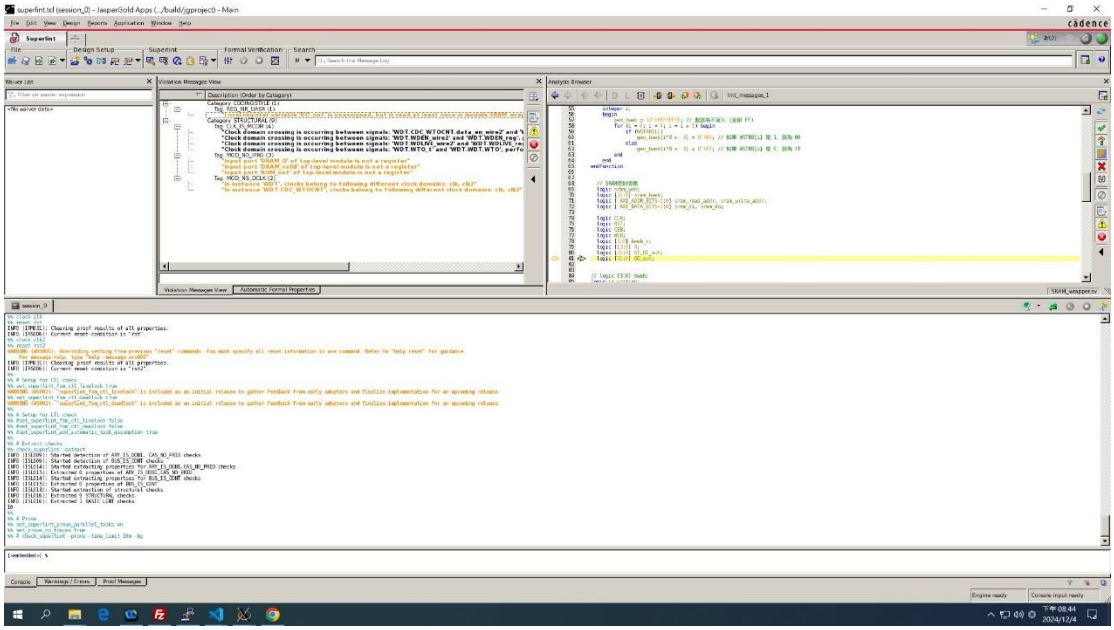


■ **Abstract**



Spyglass 中有兩個 warning 重複出現，分別為 AC_conv01 以及 Reset_sync04，這兩個 warning 即是上面 CDC 解說時說到的兩個 clk domain flip-flop 中有不穩定的情況發生時他檢測出的警告，然而那兩個 flip-flop 原本就不會穩定，因此才需要使用 double flip-flop 的架構來穩定訊號，所以判斷出這兩個 warning 不會對系統造成影響。

- **SUPERLINT**



Superlint 中並沒有甚麼 warning，有出線的 warning 幾乎都是 top level module 宣告問題，然而那部分是直接使用助教提供的 DRAM、SRAM、ROM 會跳出的宣告問題，並不是在我們撰寫的 RTL 檔案中出現的 warning，除此之外是提醒 clock domain crossing 的發生，這也是這次作業的內容，因此 superlint 並沒有對我們的撰寫的 RTL 報錯。

Problems to answer

1. What is the deference between mcycle and timer?

When is mcycle used?

在 RISC-V 架構中，mcycle 用於紀錄 CPU 執行的時鐘周期數，通常用來判斷此 CPU 實行 program 時所需總周期數，可以來做為性能分析的評估，在同樣的 program 中若使用較少的 mcycle 完成即為效能較好。而 timer 則是一般的計時裝置，通常在 CPU 外部設置，如同本次作業的 WDT，主要用途是用來與內部的 CPU 做區隔，用 timer 來進行判斷 CPU 是否正常運行，若出現異常時則要通知 CPU 重新開機，因此 mcycle 為 CPU 內部評估效能或計算功號的時鐘週期總數，timer 為外部計時器，用來判斷 CPU 是否正常運行，並控制其 TIMEOUT 的發生。

2. What is “Potential Qualifier” in Spyglass?

Signal that fails to synchronize the source signal of data crossing due to the presence of invalid logic at the point of convergence with source, or the signal is not synchronized itself。以上為 spyglass 白皮書中說明 Potential Qualifier 的描述文字。Potential Qualifier 是在我們發現錯誤時，spyglass 根據訊號的 clk domain、source signal 去標記出有價值的訊號，讓我們在 UI 介面可以進行判斷，幫助 debug。

Lesson learned

余祐安:

在這次作業中，需要理解一個小型系統並設計，包括 DRAM 的搬運原理、Boot 開機順序、CSR 中斷處理（Interrupt）、以及跨時脈域（CDC）的問題，這些都是我第一次實作的內容。同時，透過這次作業，我也重新複習了計算機組織與作業系統的相關知識，對整體電腦系統的運作有了更深的理解。

這次作業的挑戰主要在於系統規模相較於以往更為龐大，當遇到 Bug 時常不知從何下手，問題一度讓我感到困擾。回顧這些困難，我認為原因在於對系統內部機制的熟悉度不足，導致 Debug 時效率較低。

在組合系統的初期，花費了最多時間來整合各個 module。特別是 Prog0、Prog1、Prog2 的功能驗證，以及中斷處理（Interrupt）和 CDC 問題的解決，都需要反覆嘗試與調整。

中斷問題是在仔細研讀網上教材後，找到了靈感，成功解決。CDC 問題則採用了 2 Flip-Flop 的 1-bit 同步方法，因為我們判斷資料不需要頻繁讀寫，若用 async FIFO 反而顯得過於複雜且不必要。透過設計與整合，我更熟悉如何將零散的元件組合成一個完整的系統，並了解到系統各部分協同工作的原理。中斷與 CDC 是過去未曾深入實作的部分，這次作業讓我理解到它們在硬體系統設計中的重要性，並學會了實現方法。在當面對龐大系統中的 Bug 時，學到了如何有條理地逐步拆解問題，並在資料收集與嘗試中尋求解答。最後，謝謝華昀一起討論和解決問題，並且把各測資完成。電腦教室好冷。

王華昀:

這次的作業相較於 HW2 一次增加很多 IP 需要實作，從 DRAM 的搬運、Interrupt 的控制訊號、CDC 問題幾乎是把計算機組織的重點都實作一次，這一次作業我覺得很困難的部分是在看完題目後不知道從何處去下手，再加上是兩個人一起實作，在溝通上變得格外重要，在發現問題所在後需要很細心的每個 IP 去進行檢查才能知道問題出在何處。除此之外由於增加 CSR 指令，需要進入到 CPU 去調整內部的線路，很常一個不注意就讓 CPU 內部有錯誤。在通過 prog0 後以為會一路順利，又發現到 prog1 與之前的測資都不同，是需要存取 Half Word，並依照正確的位置去擺放，這是之前沒有測試到的部分，因此也有針對這部分去進行 CPU 的修改。這次的作業讓我學習到整合多個 IP 時要注意很多事情，並且讓我更熟悉課本上提到的內容，謝謝祐安跟我在電腦教室一起討論無數個下午及夜晚。