VLSI System Design (Graduate Level)

Fall 2023

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: \_\_\_陳語彤\_\_\_ , \_\_陳俐蓉\_\_\_\_

Student ID: \_\_N26121664\_\_, \_\_N26120113\_\_

**Summary**

這次的作業主要分成兩個部分，第一個部分為更改第三次作業設計的AXI BUS並且可以處理 clock domain crossing，因為在做跨頻處理時需要等待另一個clock domain回傳ready訊號才能將資料送過去，因此資料需要在AXI Interface中做暫存，在這次作業中主要是以FIFO來達成目標。這次的作業總共有五個不同的clock domain (分別為 : CPU clock 、 AXI clock 、ROM clock 、DRAM clock 、SRAM clock)，資料在進出AXI時分別會經過兩次CDC path，因此AXI在master interface和slave interface都要做CDC處理，並且要通過CDC的驗證。

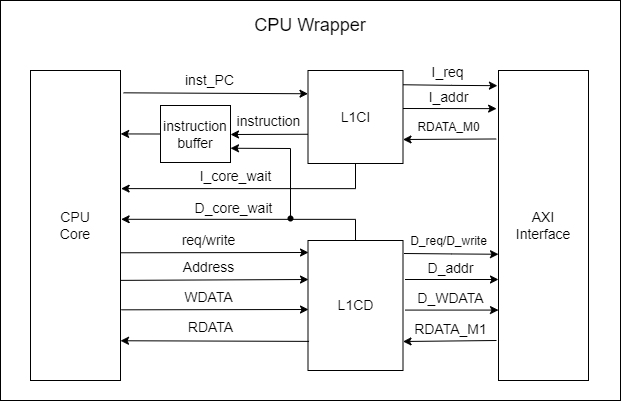
第二部分為Cache，因為AXI和其他Slave的操作頻率都比CPU慢，如果CPU要一直透過AXI去access memory，會導致讀取資料的cycle數大幅增加，如果加上cache的話可以一次從memory連續讀取4筆資料(burst length = 4)，並暫存在cache中，而且cache的操作頻率和CPU相同，CPU讀取cache時也不需要透過AXI傳輸，可以加速資料讀取速度。

|  |  |  |  |
| --- | --- | --- | --- |
| Performance & Area | | | |
|  | RTL | SYN | APR |
| Prog0 time (ns) | 1720088 | 1730496 | 1720496 |
| Prog1 time (ns) | 15972920 | 16977344 | 16972720 |
| Prog2 time (ns) | 1197016 | 1208720 | 1197664 |
| Prog3 time (ns) | 3479048 | 3502952 | 3482072 |
| Area(um^2)  In APR | 6878607.66 | | |
| CPU cycle | 8ns | | |

**Contribution**

|  |  |
| --- | --- |
| 陳語彤 50% | 陳俐蓉 50% |
| AXI CDC 驗證、AXI CDC 、APR、撰寫報告 | CACHE、撰寫報告、APR |

**CPU Wrapper**



上次實作是直接透過AXI去讀取或寫入memory資料，這次因為其他memory的操作頻率都比CPU慢，為了加快CPU讀取資料的速度，在CPU Wrapper中增加Instruction Cache(L1CI)和Data Cache(L1CD)，當CPU要讀取memory資料時，會先去cache中讀取，miss的話才會透過AXI讀取連續4筆資料(burst length = 4)，並存回cache。當CPU要寫入memory時，會直接透過AXI將資料寫進memory，若cache中已經有該memory位址的資料(write hit)，也會同時更新cache中的資料，避免memory和cache的資料不同導致錯誤。

CPU中有兩個Master 分別為Instruction Master(M0)和Data Master(M1)，當M0無法馬上取回指令時(L1CI miss read)，I-cache會拉起I\_core\_wait訊號，CPU會往後插入stall但不會暫停(但PC會停止計數)， 因此若此時M1已取回資料，CPU可以馬上接受，不需在wrapper中暫存。當M1無法馬上取回資料(L1CD miss read)或需要寫入memory時，D-cache會拉起D\_core\_wait訊號，此時CPU會完全暫停，且中間不會接受資料直到D\_core\_wait轉為low，若此時M0已取回指令，將會無法讀進CPU，所以需要將指令暫存在wrapper中的instruction buffer，等CPU開始運作後再送入CPU。

**Cache**

4

* Cache規格
* Direct map (1 way set associative)
* Cache Size : 1kB (=Byte)
* Block Size (cache line size) : 16Byte (= Byte = 128bit = word)
* Entry (#Block) : = = = 64

由上述可推算出紀錄Tag的bit數 = 32(core address bit) - 6(block index bit) - 2(offset bit) - 2 = 22，因此32bit的Core Address所表示cache中的參數範圍如下:

4

0

1

2

3

9

10

31

|  |  |  |  |
| --- | --- | --- | --- |
| Tag | Block index | Offset |  |

I\_addr[31:4] / D\_addr[31:4]

* Tag : 用來辨識此block所儲存的資料是否為CPU想要讀取的memory位址所儲存的資料。
* Block index : 標示此block的index。共有64個block，所以index範圍為0~63。
* Offset : 因為此處block size為4 word，所以需要2bit來表示block的offset。
* I\_addr / D\_addr : 當cache miss時要去memory讀取資料的起始位址。因為cache會一次讀取連續4筆資料，所以讀取memory的起始位址為原本的byte address (core address[31:2]) 除4。
* Instruction Cache (L1CI)

可以分成下列狀態

1. IDLE: 判斷是否read miss

read hit成立需要同時符合三個條件:

1. CPU讀取訊號(core\_req)拉起
2. Cache中對應block的valid bit為1 (valid[Block index] = 1)，表示此block已存有資料
3. Cache的Tag(TA\_out)和CPU address(core\_addr)相同，表示此block所儲存的資料是CPU想讀取的記憶體位址的資料

在core\_req拉起的同時，block index也會一起送入cache， TA\_out會在下個cycle才會被tag array讀出來。如果core\_req拉起時valid bit為0，那麼在同個cycle就可以知道是miss。若core\_req拉起時valid bit為1，如果此時連續4個cycle都是讀取同個block，則一定會是hit，所以會連續讀出資料，但若此時要讀取的block和上個cycle不同，則要在core\_req拉起的下個cycle才能知道是hit或miss。若miss會進入MISS，若hit會進入HIT。

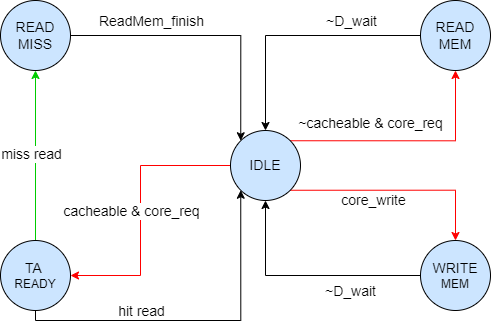
1. MISS

此時cache會去記憶體一次讀取連續4筆資料，並且拉起I\_core\_wait訊號，通知CPU等待cache取回資料，同時也要暫存core\_addr，在讀完4筆資料後(ReadMem\_finish=1)就會hit，接著回到IDLE。

1. HIT

此時cache會直接讀出資料送給CPU，再回到IDLE。

* Data Cache (L1CD)



1. IDLE

等待CPU發出讀寫訊號，可分為三種case:

1. CPU發出讀訊號(core\_req)，但要讀取的位址是uncacheable (core\_addr[31:16] != 16'h1000)，表示cache不會存取此位址的記憶體資料，CPU需要直接去記憶體讀取資料，接著狀態會進入READ MEM。
2. CPU發出讀訊號(core\_req)，且讀取的位址是cacheable，cache會存取此位址的記憶體資料，接著狀態會進入TA READY。
3. CPU發出寫訊號(core\_write)，表示CPU需要直接透過AXI寫入記憶體，狀態會進入WRITE MEM。
4. READ MISS

進入此狀態表示發生miss read，此時cache會去記憶體一次讀取連續4筆資料，並且拉起D\_core\_wait訊號，通知CPU等待cache取回資料，同時也要暫存core\_addr，在讀完4筆資料後(ReadMem\_finish=1) 就會hit read，並回到IDLE。

1. WRITE MEM

進入此狀態表示CPU需要直接透過AXI寫入記憶體，此時會拉起D\_core\_wait訊號，等待寫入完成後CPU才會繼續執行。此狀態需要另外判斷是否hit write，若為hit write表示cache中已經有該memory位址的資料，需要同時更新cache中的資料，避免memory和cache的資料不同導致錯誤。

hit write成立條件為core\_write & cacheable & (valid[Block index] == 1) & (TA\_out == core\_addr[31:10])。

1. READ MEM

進入此狀態表示CPU需要直接去記憶體讀取資料，此時會拉起D\_core\_wait訊號，等待讀取完成後CPU才會繼續執行。並且把CPU wrapper中的burst length設為1，表示一次AXI傳輸只會讀取1筆資料。

1. TA READY

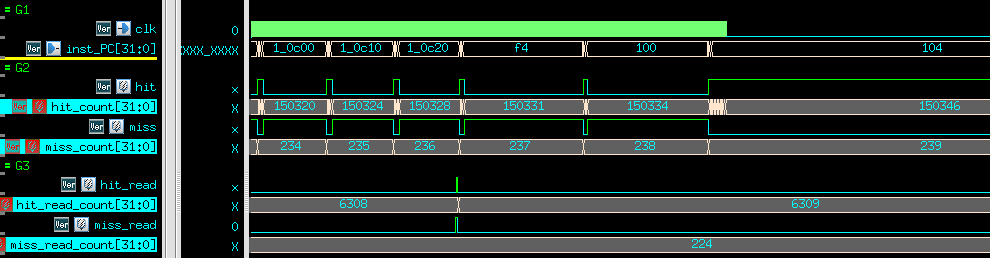
進入此狀態表示tag array已經讀出TA\_out，可以判斷miss read或hit read，若miss read，則狀態會進入READ MISS，若hit read，則狀態會回到IDLE。

miss read或hit read的判斷需core\_req & cacheable才是有效的，若valid[Block index] = 1，且TA\_out和CPU address(core\_addr)相同，則為hit read，反之則miss read。

* Hit Rate



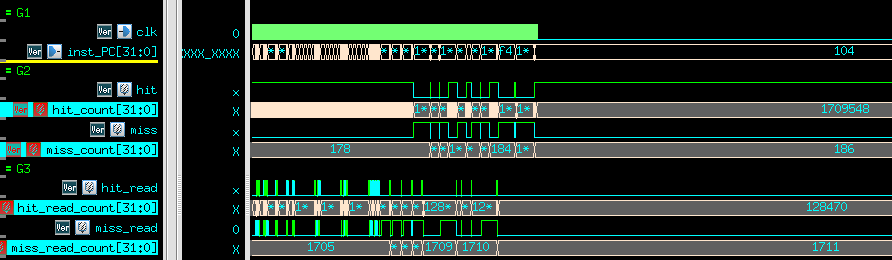
1. prog0



I-cache : hit rate = = 99.8%

D-cache : hit rate = = 96.6%

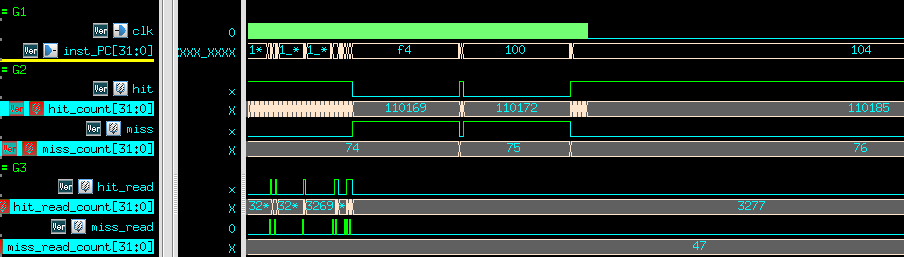
1. prog1



I-cache : hit rate = = 99.9%

D-cache : hit rate = = 98.6%

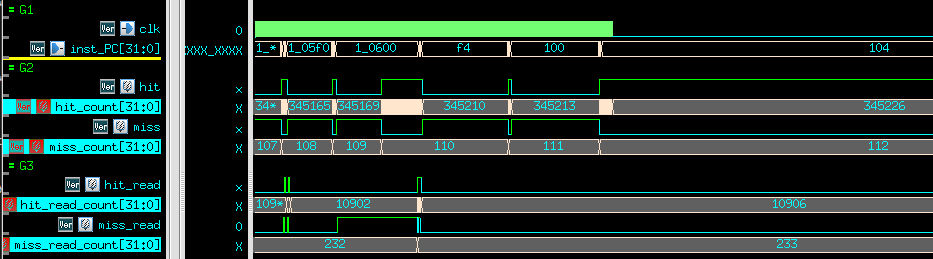
1. prog2



I-cache : hit rate = = 99.9%

D-cache : hit rate = = 98.6%

1. prog3

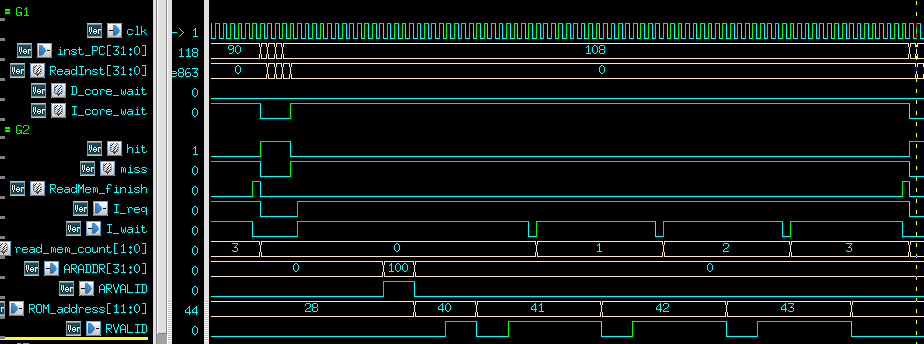


I-cache : hit rate = = 99.9%

D-cache : hit rate = = 97.9%

* Waveform

1. I-cache miss read

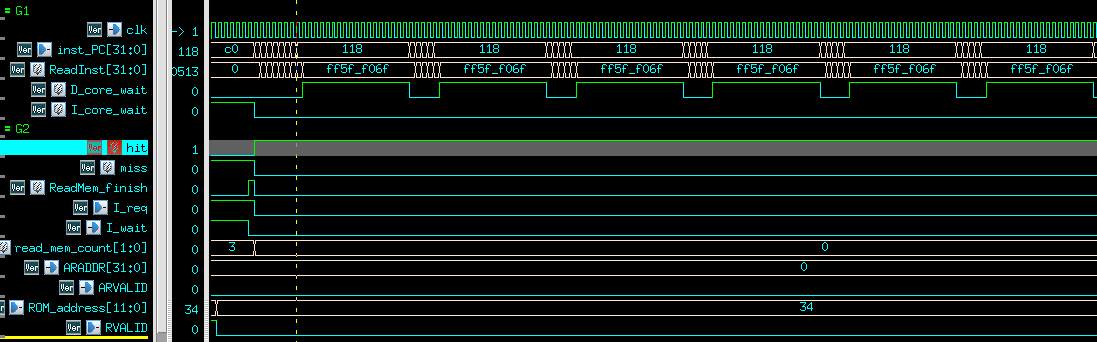


ROM

Wrapper

PC=108~114的指令是一個迴圈，這段波形是program第一次執行到此迴圈，此時該PC的指令尚未被存入cache中，所以會miss，接著cache就會去ROM讀取連續4筆指令(I\_req=1)，這段期間I\_core\_wait會被拉起，通知PC停止計數，讀取完成後(ReadMem\_finish=1)會把4筆指令存入cache中，並轉為hit。

1. I-cache hit read

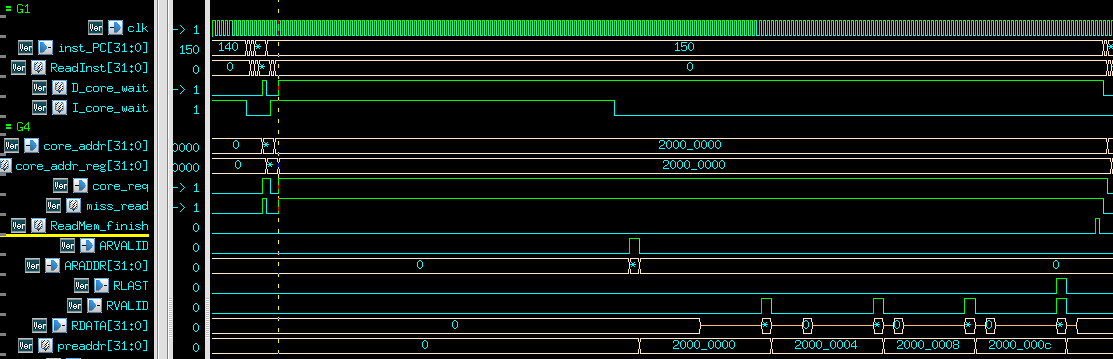


ROM

Wrapper

這段波形是program第二次以上執行第(1)點的迴圈，因為此時cache中已經儲存了此段PC的指令，所以會hit，不需再次到ROM中讀取指令，直接從cache讀取指令即可，可以加快讀取資料的速度。

1. D-cache miss read



DRAM

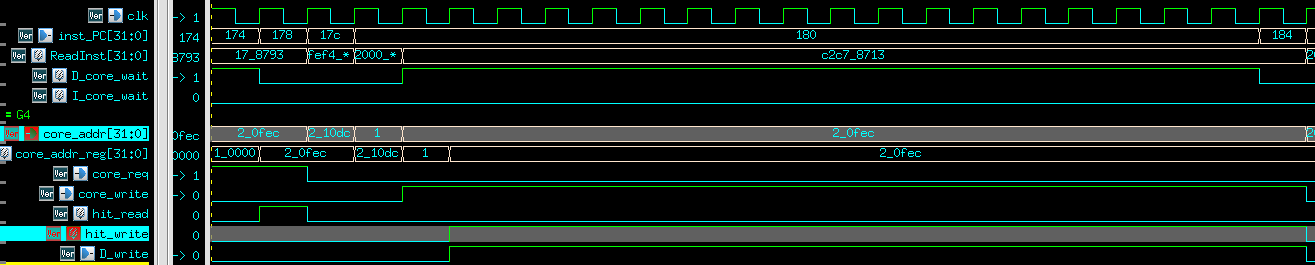
Wrapper

此波形是booting時CPU會把DRAM 中program的指令搬進IM中，此時是CPU第一次access 此位址的DRAM，所以會read miss，接著cache會去DRAM中一次讀取4筆資料並存起來。

1. D-cache hit write

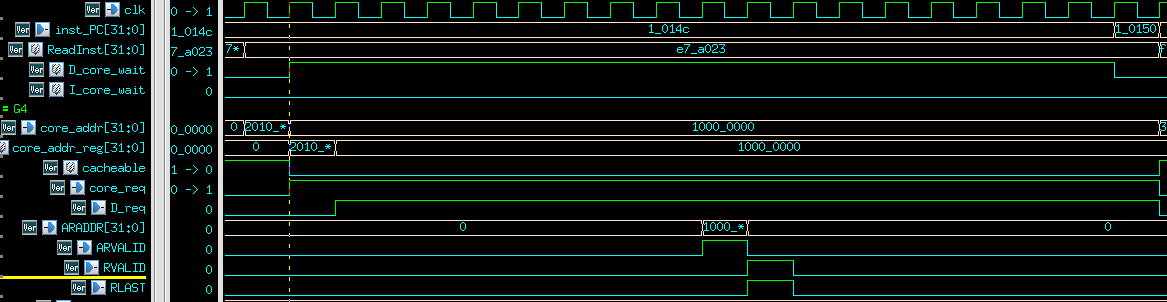
Hit write

Hit read



可以看到在CPU想寫入20fec這個位址時，在更早之前其實已經將此位址的資料讀進cache中，因此在寫入此位址記憶體的同時，也要同步更新已儲存在cache中的資料。

1. D-cache uncacheable



SCtrl

Wrapper

這段波形是CPU要去讀取sensor control，sensor control的位址是uncacheable，cache不會存取此位址的資料，所以CPU需要直接透過AXI讀取sensor control的該筆資料，且讀回來的資料也不會存進cache中。

**AXI clock domain crossing**

|  |
| --- |
|  |
| AXI CDC |
| 在先前的作業中AXI與CPU、SLAVE的clock domain是相同的，故不會有clock domain crossing的情形發生，所以在資料傳輸的過程中只需要handshake一次就好了，但此次有5個clock domain而且AXI與CPU是不同clock domain，所以在資料從master到slave總共需要經過三個clock domain，此時就必須要有三次的handshake才能避免clock domain crossing的問題。  第一次的handshake是發生在master與slave interface之間，這次是從cpu clock domain跨到axi clock domain，第二次的handshake是AXI BUS，此時不需要做CDC，在實作時這裡的handshake跟前幾次的作業相同，而最後一次的handshake是發生在master interface與slave之間，這次是從axi clock domain跨到slave clock domain。 |

|  |
| --- |
|  |
| VALID與READY訊號(master to slave interface) |
| 在資料傳輸的過程中VALID與READY訊號是不需要經過FIFO的，舉ARVALID為例，在這裡的ARVALID\_M\_i是從master發出的VALID訊號，而ARREADY\_M \_i是由FIFO發出的READY訊號，只要FIFO不是滿的情況下就會回傳1的訊號(可以由下圖的黃框中的波形看出) ，而ARVALID\_M是FIFO發出的VALID訊號只要FIFO不是空的情況下就會發出1的訊號，而ARREADY\_M 是由AXI handshake所發出的訊號(可以由下圖的紅框中的波形看出)。 |
|  |

|  |
| --- |
|  |
| VALID與READY訊號(master interface to slave) |
| 在這裡的ARVALID\_S是由AXI handshake發出的VALID訊號，而ARREADY \_S是由FIFO發出的READY訊號，只要FIFO不是滿的情況下就會回傳1的訊號(可以由下圖的黃框中的波形看出)，而ARVALID \_S\_O是FIFO發出的VALID訊號只要FIFO不是空的情況下就會發出1的訊號，而ARREADY\_ S\_i 是由SLAVE端所發出的訊號，代表可以到此SLAVE去做資料存取的動作(可以由下圖的紅框中的波形看出)。 |
|  |

|  |
| --- |
|  |
| AXI CDC (FIFO) |
| 這是這次作業處理clock domain crossing的FIFO，在這次的設計當中我將FIFO深度設計為1，因為不會有一次多筆資料要通過FIFO，所以深度是1就足夠了，接著是下面兩個DFF的部分，也就是助教在講義上GRAY code的部分，左邊的DFF主要是將RCLK domain的rptr訊號經過兩個D-Flip-Flop傳到WCLK的domain，而右邊的DFF則是將WCLK domain wptr訊號經過兩個D-Flip-Flop傳到RCLK的domain。  接著是說明FIFO wptr與FIFO rptr兩個module在做的事，FIFO wptr主要是在當資料寫進FIFO時(也就是VALID為1的時候)必須去判斷FIFO是否是滿的，如果FIFO是滿的話就會將READY訊號降為0(同時也不會讓winc升起)，而FIFO rptr主要做的事是當資料要從FIFO讀出時(也就是VALID為1的時候)必須去判斷FIFO是否是空的，如果FIFO是空的話就會將READY訊號降為0(同時也不會讓rinc升起) 。  最後則是FIFO的部分，因為在這次作業時做的過程中不會有同時好幾筆資料需要被存進FIFO的情形發生，所以FIFO深度為1，因此不會有需要判斷index的情形發生，處理起來相對較輕鬆，只需在winc為1的時候將資料存進data memory中，同時將wptr做反向的動作(為了判斷FIFO空跟滿)，而在rinc為1的時候將資料讀出，在這部份為了避免錯誤我還是有在判斷一次當FIFO為空的時候讀出的資料將會為0，雖然只要FIFO是空的話rinc基本上是不會被拉起的。 |

|  |
| --- |
| FIFO Simulation Result |
|  |
| 以AR傳輸地址為例，當ARVALID\_M\_i為1時代表master要做傳地址的動作，此時只要FIFO是空的就會回傳ARREADY\_M同時winc(FIFO的write enable)就會為1，並且在下一個cycle將資料寫進FIFO同時也會更新wptr的值，而只要wptr的值不等於rptr\_wq2，wfull的訊號就會被拉起(表示FIFO已經滿了)(上述動作可以在上圖黃框中看出)，但同時rempty不會同時放下因為判斷空是read clock domain要做的是，所以需等待wptr的值經過clock domain crossing處理後傳到read clock domain此訊號為下圖的wptr\_rq2，只要此訊號與rptr不同rempty就會被拉起，同時只要ARREADY\_M 為1 rinc (FIFO的read enable)就會為1 ，同時將資料讀出，而此時讀出的資料應該要與剛剛寫進FIFO的值相同。  以AR傳輸地址為例，在master與slave interface之間需要寫入FIFO的資料包含ARID、ARLEN、RDATA、ARSIZE、ARBURST，而如果是R傳輸資料的時候，在slave與master interface之間則是需要將RID、 RDATA、 RRESP、RLAST存入FIFO裡，由上述例子可以得知除了VALID與READY訊號其餘需要傳輸的資料都必須存進FIFO裡。 |

**CDC Spyglass**

|  |
| --- |
| CDC Setup Check ( 0 Error、0 Warning) |
|  |
| Clock Reset Integrity ( 0 Error、0 Warning) |
|  |
| CDC Abstract ( 0 Error、0 Warning) |
|  |

|  |
| --- |
| CDC Verify Struct ( 0 Error、62 Warning) |
|  |
|  |
|  |
|  |

|  |
| --- |
| CDC Verify ( 0 Error、97 Warning) |
|  |
|  |
| CDC Learned |
| 在這次cdc驗證的過程中讓我發現其實我AXI的clock domain crossing，其實有很多的問題，在一開始一直有100多個error，因為上次的問題較簡單所以在驗證cdc的時候沒有出現太多的問題，而這次狀況較複雜導致出現了很多我沒看過的error，對於spyglass的工具使用也不太清楚，所以這次特別回去讀了上次作業的講義，才發現其實可以看到simplified schematic只需要按Incremental Schematic有了這張圖讓我debug的過程中更加的順利，雖然最後的問題是因為我忘了把AXI Arbiter的clock從cpu clock domain換成axi clock domain，還有把ready 訊號m0跟m1接反了，但在這次debug的過程中讓我對於spyglass的tool使用更加的熟悉，對於cdc也有更深入的了解。 |

**Simulation Result**

|  |  |
| --- | --- |
| Prog0 | Prog1 |
|  |  |
| Prog2 | Prog3 |
|  |  |

**Synthesize result**

|  |  |
| --- | --- |
| Prog0 | Prog1 |
|  |  |
| Prog2 | Prog3 |
|  |  |

**APR result**

|  |  |
| --- | --- |
| Prog0 | Prog1 |
|  |  |
| Prog2 | Prog3 |
|  |  |
| Floorplan | |
|  | |

|  |
| --- |
| Area |
|  |
| Power |
|  |

**Superlint result**

|  |  |  |  |
| --- | --- | --- | --- |
| Superlint(number of inline messages) | | | |
|  | | | |
| Total lines | Warning | Error | coverage(%) |
| 9629 | 207 | 0 | 97.9% |

**Lesson learned**

1. 在這次作業實作之前，我對於clock domain crossing的概念其實不太清楚，雖然在上課的過程中有提到過，不過大多數都是在說1 bit的情形，只需經過1-bit Synchronizer或是Pulse Synchronizer，上述兩種情形是我們比較熟悉的處理方式，而上次作業雖然WDT的32 bit 變數WTOCNT需要做clock domain crossing，但我們並沒有用到fifo，我們是直接做Synchronizer handshake(很像是深度為1的fifo)，雖然概念差不多但此次作業的難度相較起來較困難，因為我一開始實作的fifo深度為15，在gray code的處理及fifo 的滿跟空上需要相對複雜的過程，而在實作完成後，通過simulation後才發現其實只會用到深度為1的fifo，所以最後又將深度調整為1，寫起來就相對簡單很多。

在這次AXI clock domain crossing的處理過程中也有遇到另外一個問題， 就是VALID跟READY訊號要如何傳遞，因在HW4公布的時候助教說VALID跟READY訊號是不需要進到FIFO裡面的，一開始還沒確實了解該怎麼做時確實讓我們一頭霧水，不過在經過我們兩個人30分鐘的討論後才有了初步的想法，雖然還是有部分是沒搞清楚的，但在多次錯誤後終於弄懂，原來空跟滿就是處理clock domain crossing一部分，因為滿是由wptr 不等於 rptr\_wq2判斷，其中wptr為write clock domain的訊號而rptr\_wq2是read clock domain的訊號，而空是由rptr 不等於wptr \_rq2判斷，其中wptr \_rq2為write clock domain的訊號而rptr是read clock domain的訊號。

1. 在這次作業中我們也修正了一些上一次所做的wrapper，因為在之前時做的時候沒有cache不會有一次需要讀四筆資料的時候，加上因為還要處理AXI clock domain crossing的關係讓wrapper的地址出現了一點狀況，在某些情形下無法將資料寫進正確的位置裡，因為AXI是不會暫存資料的，他只是一個傳輸的Bus，所以在data從FIFO被讀出後下一個cycle資料就會被歸0，導致當資料到slave要做寫入動作的時候地址是錯誤的，所以這次我們在wrapper在讀寫資料的時候都有將傳進wrapper的地址去做暫存，等到此次的資料皆傳輸完成。
2. 這次做cache花最多時間的地方就是在對時序，因為TA\_out會在block index送進去的下個cycle才會讀出來，導致判斷hit或miss需要跟著一起晚一個cycle，如果沒有用狀態機切好狀態，容易導致時序錯亂，而且因為我一開始沒有用狀態機寫，為了對時序導致在core\_wait訊號的判斷在合成的時候出現combinational loop，因此又回去修改I-cache的設計，讓core\_wait單純用miss去判斷。
3. 這次因為cache會一次讀取4筆資料的關係，有把wrapper裡一次讀取資料的次數改成4次，實際做法就是把傳進wrapper裡的位址每讀一次就加4，這樣就可以讀出連續4筆的資料。