VLSI System Design (Graduate Level)

Fall 2018

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 any waveform files in deliverables

Student name: 王傑世 、 鄭維

Student ID: P76091690、P76091226

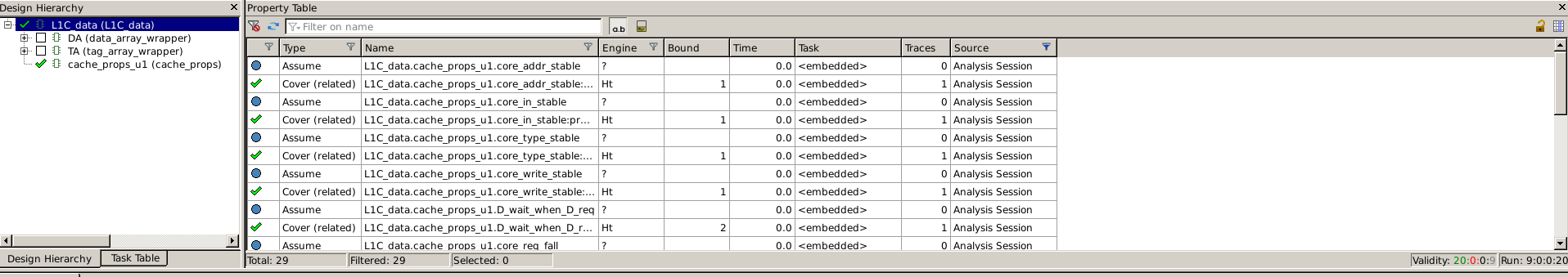
1. **Summary**
   1. **Performance**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Prog0 | Prog1 | Prog2 | Prog3 | area |
| 976320ns | 1848092ns | 11844108ns | 1839447ns | 6454150 |

* 1. **Simulation**

|  |  |  |  |
| --- | --- | --- | --- |
| Prog0 | pass | Syn0 | pass |
| Prog1 | pass | Syn1 | pass |
| Prog2 | pass | Syn2 | pass |
| Prog3 | pass | Syn3 | pass |

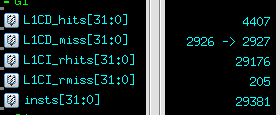
* 1. **make cache (pass)**



* 1. **IPC、HIT rate、MISS rate**

\*L1IC只有read hit和 read miss

* + 1. **Prog0**



D-cache hits = 4407

D-cache miss = 2927

D-hit rate = 4407/7334 = 0.600

I-cache hits = 29176

I-cache miss = 205

I-hit rate = 29176/29381 = 0.993

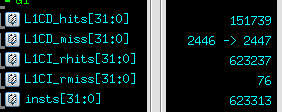
-

instructions = 29381

cycles = 108480

IPC = 0.270

* + 1. **Prog1**



D-cache hits = 151739

D-cache miss = 2447

D-hit rate = 151739/154186 = 0.984

I-cache hits = 623237

I-cache miss = 76

I-hit rate = 623237/623313 = 0.999

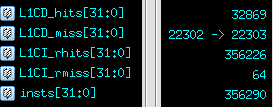
-

instructions = 623313

cycles = 2053388

IPC = 0.304

* + 1. **Prog2**



D-cache hits = 32869

D-cache miss = 22303

D-hit rate = 32869/55172 = 0.596

I-cache hits = 356226

I-cache miss = 64

I-hit rate = 356226/356290 = 0.999

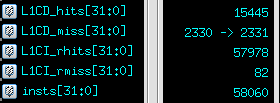
-

instructions = 356290

cycles = 1303569

IPC = 0.273

* + 1. **Prog3**



D-cache hits = 15445

D-cache miss = 2331

D-hit rate = 15445/17776 = 0.869

I-cache hits = 57978

I-cache miss = 82

I-hit rate = 57978/58060 = 0.999

-

instructions = 58060

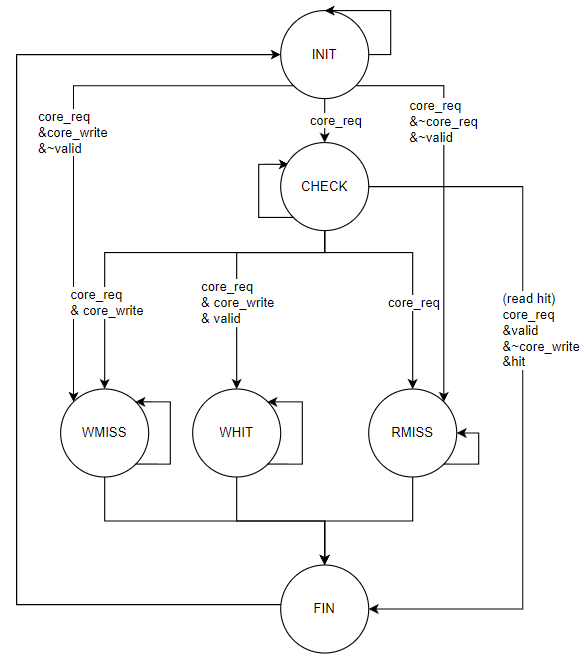
cycles = 204383

IPC = 0.284

1. **Lesson Learned**
   1. **Cache**

address: 32bit

|  |  |  |  |
| --- | --- | --- | --- |
| Tag | Index | Block offset | Byte offset |
| 22 | 6 | 2 | 2 |



* + 1. **INIT**

在這個state，會確認core\_req(CPU是否要讀或寫)，CPU沒有請求的話就會維持在INIT這個state下。CPU若是有請求，就會檢查這個address的index是否valid，core\_req=1的前提下，valid=1就會進入CHECK，若為0就代表miss了，再根據core\_write這條訊號決定進入WMISS或者是RMISS。

另外，因為data\_array和tag\_array都是SRAM，所以這時送進去index，在下個cycle才吐出我們需要比對的data，才可以確認是否為hit，因此多了一個CHECK的state做檢查。

* + 1. **CHECK**

進到這個state表示，這個index的valid為1，但是還需要確認address的tag和原本在tag\_array的tag是否相同。相同代表hit，就會read hit或 write hit，反之，進入RMISS或WMISS。

read hit的話，在這階段跟data\_array要data，並在下個cycle

* + 1. **WMISS**

在這個state，採取write around的方式，也就是只會將data寫入SRAM，並不會寫在cache

* + 1. **WHIT**

如果write hit的話，採取write through的方式，寫入SRAM的同時也會寫入cache，避免 cache coherency的問題。

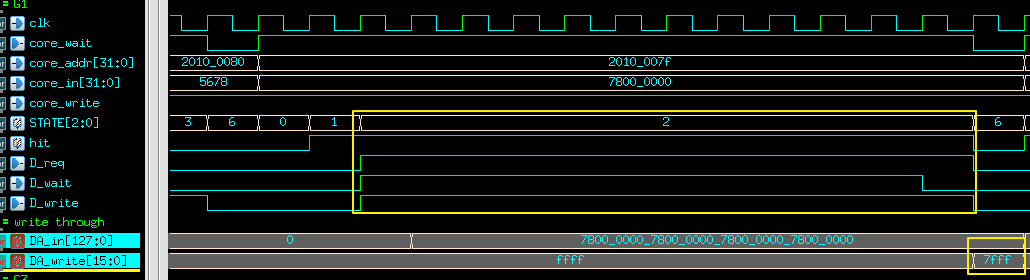
* + 1. **RMISS**

這次的line為4個word，所以如果read miss的話，cache會透過Master發送請求，再根據address的位址透過AXI傳送給SRAM或是DRAM，其中之一接受後，就會連續讀取4筆data經過AXI傳給Master再給cache，cache再將拿到的4個word存起來，並且根據address的block offset，把正確的data給CPU

* + 1. **FIN**

在前面的階段由於cache都在動作，所以會發出wait的訊號，告訴CPU現在不能動作，這個state就是當cache處理完後要把wait拉下來，讓CPU繼續後面的動作。

**Example1: (write hit)**



INIT(0): 因為index valid，所以會進入下個state(CHECK)

CHECK(1): 這邊會檢查cache的tag是否和core\_addr的相同，這裡hit=1且core\_write=1，表示write hit，進到下個state(WHIT)

WHIT(2): 黃框的部分

這裡會向我們的slave發送請求(D\_req =1)，在AXI根據core\_addr做解碼，得到要寫入的地方(這裡寫入DRAM)，這時D\_RAM會發出D\_wait的訊號要求cache不要有動作，等到D\_wait拉下，此時寫入也完成了，進入下個state(FIN)。

另外，由於這裡採write through的策略，所以會將core\_in也寫入cache中，依據block offset、byte offset、core\_type，決定DA\_write的行為，這裡DA\_write = 16’h7fff

FIN(6): cache一連串動作完成後，就可以把core\_wait的訊號拉下來

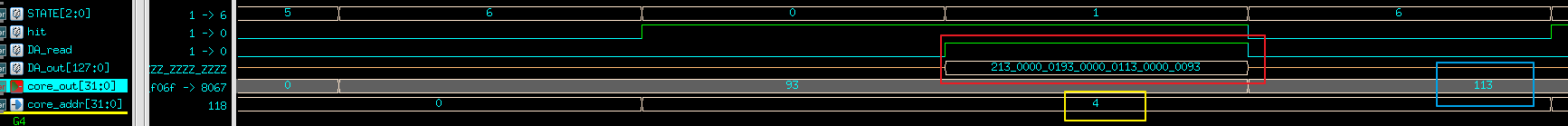
**Example2: (write miss)**

INIT(0): 因為valid=0(miss)、core\_req=1、core\_write=1，所以跳過state(CHECK)，直接進到state(WMISS)

WMISS(3): write miss的話，採write around，只會寫入SRAM或DRAM而已，不會寫入data\_array，因此這裡DA\_write=4’hffff(黃框)都是拉高的，表示不會寫入

FIN(6): D\_wait拉下後的下個cycle，就會進到這個state，表示完成。

**Example3: (read hit)**



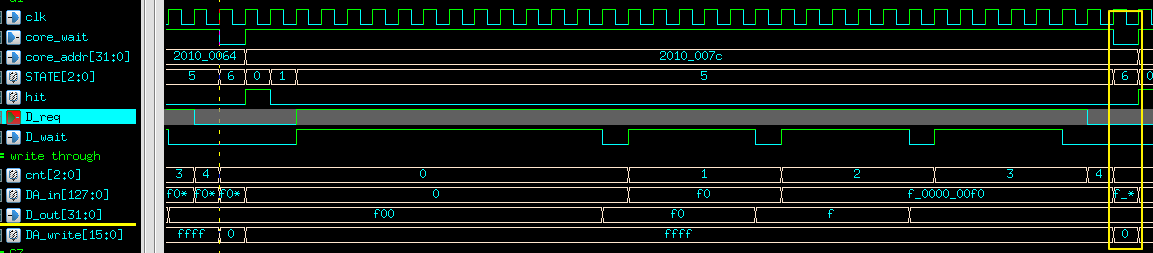
INIT(0): index valid，所以會進入state(CHECK)

CHECK(1): 這裡tag相同，發生hit=1，並且不需要寫入，所以read hit了。

這時候就可以拿cache中data\_array的值，把DA\_read拉高，表示讀取，會回傳一筆128bit的data(紅框)，再根據core\_addr[3:2](黃框)block offset，決定哪個word是我們需要的，這裡core\_addr[3:2] = 2’b01，所以會從其中選第一個block，32’h113當作我們要的data

FIN(6): 把有效的data存到core\_out，回傳給CPU。

**Example4: (read miss)**



INIT(0): index valid，所以會進入state(CHECK)

CHECK(1): 這邊會檢查cache的tag是否和core\_addr的相同，這裡沒有 hit且core\_write=0，所以read miss

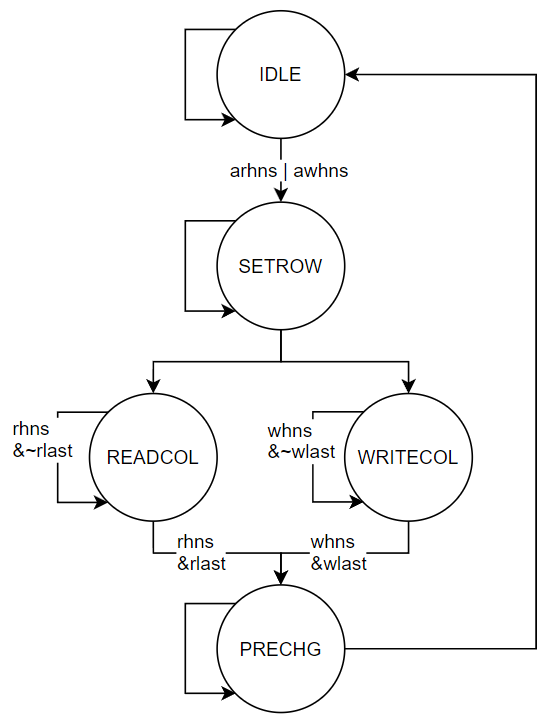
RMISS(5): cache每個line可以存4個word，所以cache會透過master向slave 要求。

這裡會向DRAM讀取data，每當DRAM運作完後，D\_wait就會降下來，表示data有效，讀完4次後，這4筆data共128個bit會用register (DA\_in)存起來，在下個state(FIN)存到DA\_array

FIN(6): DA\_write會設成4’h0，表示寫入整個128bit的data

* 1. **DRAM\_wrapper**

DRAM決定row，col和充電的時候都至少需要5個cycle，所以會用counter去做累加



* + 1. **IDLE**

如果有aw\_channel或ar\_channel發生handshake的話就會進入SETROW的state，由於這個state不會做任何事情，所以DRAM的每個訊號都會被拉高，做等待的動作

* + 1. **SETROW**

RASn = 1’b0;

CASn = 1’b1;

WEn = 4’hf;

act row之後，會根據AXI送來的訊號，決定要read col 或者是 write col

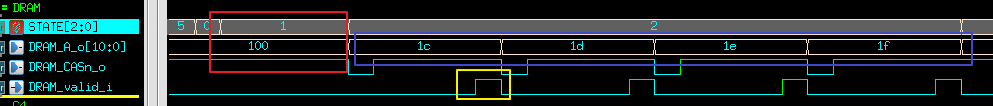
* + 1. **READCOL**

RASn = 1’b0;

CASn = 1’b1;

WEn = 4’hf;

在read miss的壯況下，由於會讀取4筆data，並且這4筆都會在同一個row上，所以不需要重新充電、重新選row，繼續在同一個row上執行，只需要改變col即可，可以減少大量的cycle，加快時間



此圖情況為read miss。

SETROW(1): 選定好row後(row=11’h100) READCOL(2): 在一開始丟出CASn=1’b0的訊號，並將選定的col送到DRAM，DRAM運作完後會DRAM\_valid(黃框)的訊號會被拉高，表示data有效，因為read miss需要讀取4筆data，而且都是在同一個row上，只是col會不同而已(依序為11’h1c, 11’h1d, 11’h1e, 11’h1f)(籃框)

* + 1. **WRITECOL**

RASn = 1’b0;

CASn = 1’b1;

WEn = 4’h0; (a word)

WEn主要根據write strob和byte offset，決定要寫入多少，若是要完整寫入一個data，WEn就設成4’h0。

* + 1. **PRECHG**

RASn = 1’b0;

CASn = 1’b1;

WEn = 4’h0;

因為DRAM的讀取是破壞性的，所以需要先precharge，才能重新設定row，然後再選col

* 1. **ROM\_wrapper**

用SRAM\_wrapper做修改。由於ROM只會讀取，並不會作寫入的動作，awready, wready永遠為0，所以將SRAM\_wrapper中write channel都設為不會動作即可。

1. **Program**
   1. **boot.c**

一開始boot.c的指令都會存在ROM裡面， 這些指令的目的就是將DRAM裡面的data搬到我們的SRAM裡面，在正式開始執行prog時，才可以從instruction memory、data memory，這兩個SRAM中讀取到data。

指令都放在(\_dram\_i\_end - \_dram\_i\_start)這個範圍內，必須將這些指令依序移到(\_mem\_start)之後，

也就是搬到SRAM(IM)。

要搬到SRAM(DM)，第一個範圍是(\_sdata\_end-\_sdata\_start)，依序移到(\_\_sdata\_paddr\_start)，第二個是(\_\_data\_end-\_\_data\_start)，一樣依序搬到(\_\_data\_paddr\_start)。

以上只要用個for loop，將我們的來源相等於目標的位址，就可以達到搬移的效果。

**Example:**

for (i=0; i<( (&\_dram\_i\_end) - (&\_dram\_i\_start));i++){

(&\_imem\_start)[i] = (&\_dram\_i\_start)[i];

}

* 1. **Prog1**

bubble sort，為了簡短執行時間，對2個要交換位置的data做XOR，連續做3次就可以達到swap的目的

* 1. **Prog2**

將RGB圖片轉成灰階，3種channel分別要乘上0.11、0.59、0.3，因為在CPU中並沒有針對浮點數做處理，所以需要對這些小數做近似值。先對這3個數做轉換，如下:

0.11 = (1110000101001000)2 \*2-20

0.59 = (10010111000010100100)2 \* 2-20

0.3 = (1001100110011001101)2 \* 2-20

轉乘2進位後，只需要對我們的目標值進左移，就可以達到\*2、\*4...，乘上2的次方的效果，最後再向右移20個bit(\*2-20)，就可以得到結果。有了0.11\*b、0.59\*g、0.3\*r的積之後，三者相加就可以拿到灰階值。

* 1. **Prog3**

矩陣相乘中，乘法佔相當大的比例，所以加快乘法的運算，就可以大大縮短整個執行的時間。利用bitwise做相乘，就可以達到這個目的。

**Example:** 3\* 17

17 = (10001)2

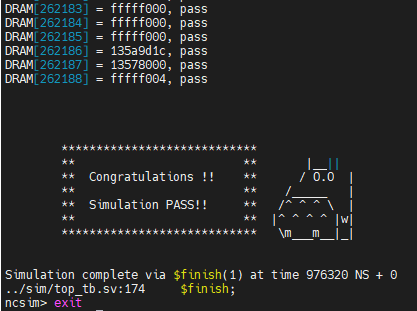
= 16\*1+ 8\*0+ 4\*0+ 2\*0+ 1\*1

3\*17 = 1\*1\*3 + 0\*2\*3+0\*4\*3+0\*8\*3+1\*16\*3

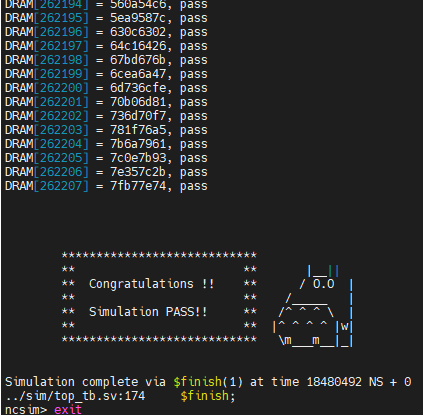
因為都是和2的N次方做相乘，所以只需要用shift 左移即可。

reference: https://stackoverflow.com/questions/3722004/how-to-perform-multiplication-using-bitwise-operators

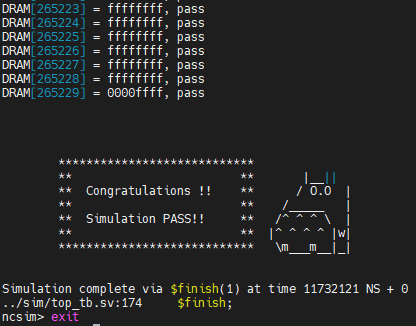
1. **Simulation**
   1. **SYN**
      1. syn0



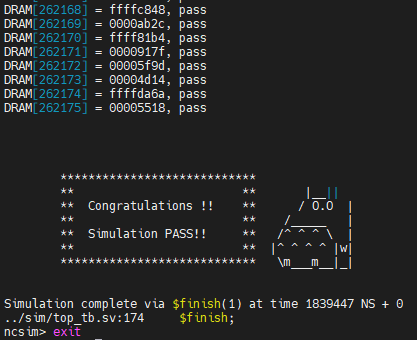
* + 1. syn1



* + 1. syn2



* + 1. syn3



1. **Superlint**

