# VLSI System Design (Graduate Level)
# Fall 2020

# HOMEWORK II
# 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:___周昱佑___ ____杜冠勳____
Student ID:__**N26090180**__ _**N26094883**___

## Summary :

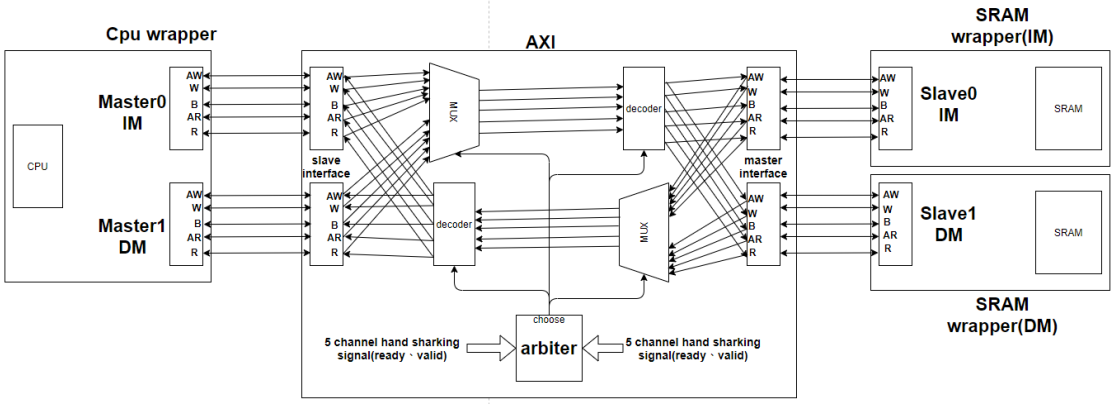
| | | |
|---|---|---|
| 1. | AXI.sv verification(jaspergold) | **DONE** |
| 2. | CPU_wrapper.sv verification(jaspergold) | **DONE** |
| 3. | SRAM_wrapper.sv verification(jaspergold) | **DONE** |
| 3.1 | IM=slave1(ID=0) | **DONE** |
| 3.2 | DM=slave2(ID=1) | **DONE** |
| 4. | Outstanding | 1 (In order) |
| 5. | Burst operation | Single transfer only |
| 6. | Transfer type: | No burst transfer |
| 7. | Synthesis | **DONE** |
| 7.1 | Prog0 before and after Synthesis | **PASS** |
| 7.2 | Prog1 before and after Synthesis | **PASS** |
| 7.3 | Prog2 before and after Synthesis | **PASS** |
| 7.4 | Prog3 before and after Synthesis | **PASS** |
| 8. | superlint | **99.24%** |

## 分工 :

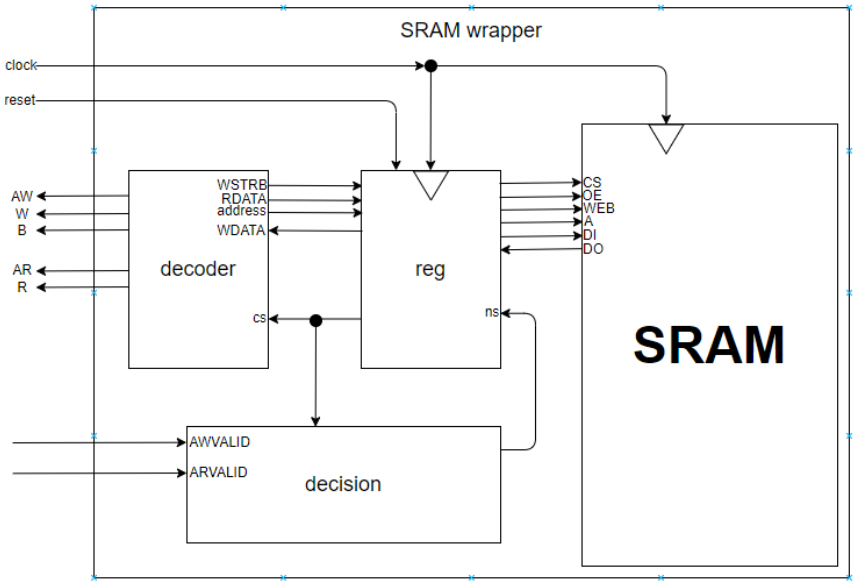
| 姓名 | 周昱佑 **(50%)** | 杜冠勳 **(50%)** |
|---|---|---|
| 負責工作 | **AXI**<br>**SRAM Wrapper** | **AXI**<br>**CPU Wrapper** |

## Architecture :

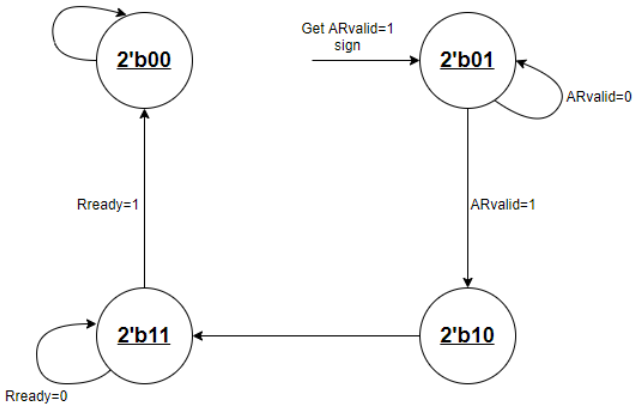
**此次 AXI 的設計架構以狀態機來做訊號線的控制與讀存資料，以這樣設計的原因為較易於做判定是否要維持訊號與 cpu 的 stall，同時在 debug 能較清楚的觀察到是哪個狀態的判定上出了問題，狀態的切換依照 spec 上描述，以 valid 與 ready 為主，在 slave 端會驗證 ID 與 ADDRESS 的正確性給予 RESP 為 OK 或 DECERR，中間 AXI 的部分以 arbiter 來決定優先執行的部分，設計上以 master_dm 優先於 master_im，write 優先於 read。**



➢ Slave :




1. Read FSM




**hand_sharking_signal**

2'b00 : finish (default state)
ARREADY=1'b0
RLAST=1'b0
RVALID=1'b0

2'b01 : AR (get ID or address)
ARREADY=1'b1

2'b10 : access memory (read "Data" from memory)

2'b11 : R (return respon of write)
RRESP=((ID is self_ID)&&(addr is in range))?2'b00:2'b11
RID=ID
RLAST=1'b1
RVALID=1'b1

**memory_data**

2'b00 : keep signal

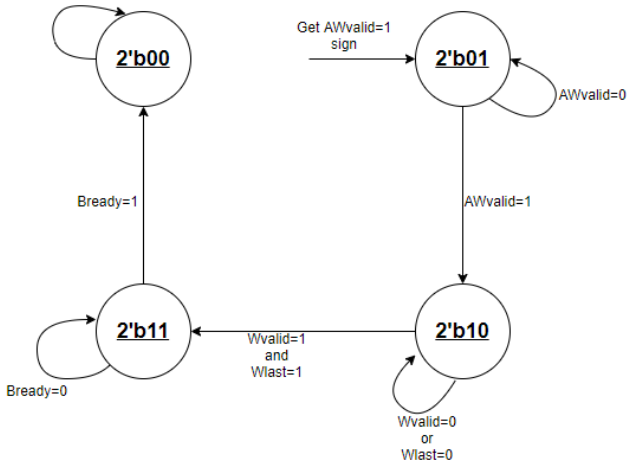
2'b01 : get "address" "ID" from master
addr<=(read or write)?ARADDR:AWADDR
ID<=(read or write)?ARID:AWID

2'b10 : memory control and "write" data to memory
CS<=1'b1
OE<=(read or not)?1'b1:1'b0
WEB<=(write or not)?WSTRB:4'b1111
A<=addr[13:0]

DI<=(write valid)?WDATA:32'd0

2'b11 : get "read" data
RDATA=(read or not)?DO:RDATA

2. Write FSM




**hand_sharking_signal**

2'b00 : finish (default state)
AWREADY=1'b0
WREADY=1'b0
BVALID=1'b0;

2'b01 : AW (get ID or address)
AWREADY=1'b1

2'b10 : W (write "Data" to memory)
WREADY=1'b1

2'b11 : B (return respon of write)
BRESP=((ID is self_ID)&&(addr is in range))?2'b00:2'b11
BID=ID
BVALID=1'b1

**memory_data**

2'b00 : keep signal

2'b01 : get "address" "ID" from master
addr<=(read or write)?ARADDR:AWADDR
ID<=(read or write)?ARID:AWID

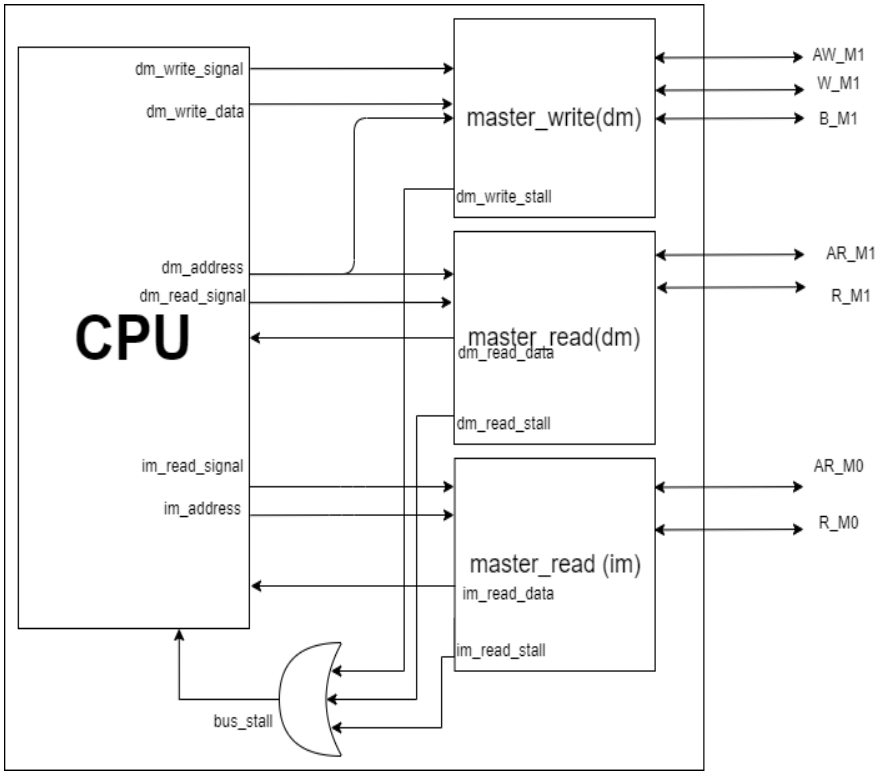
2'b10 : memory control and "write" data to memory
CS<=1'b1
OE<=(read or not)?1'b1:1'b0
WEB<=(write or not)?WSTRB:4'b1111
A<=addr[13:0]

DI<=(write valid)?WDATA:32'd0

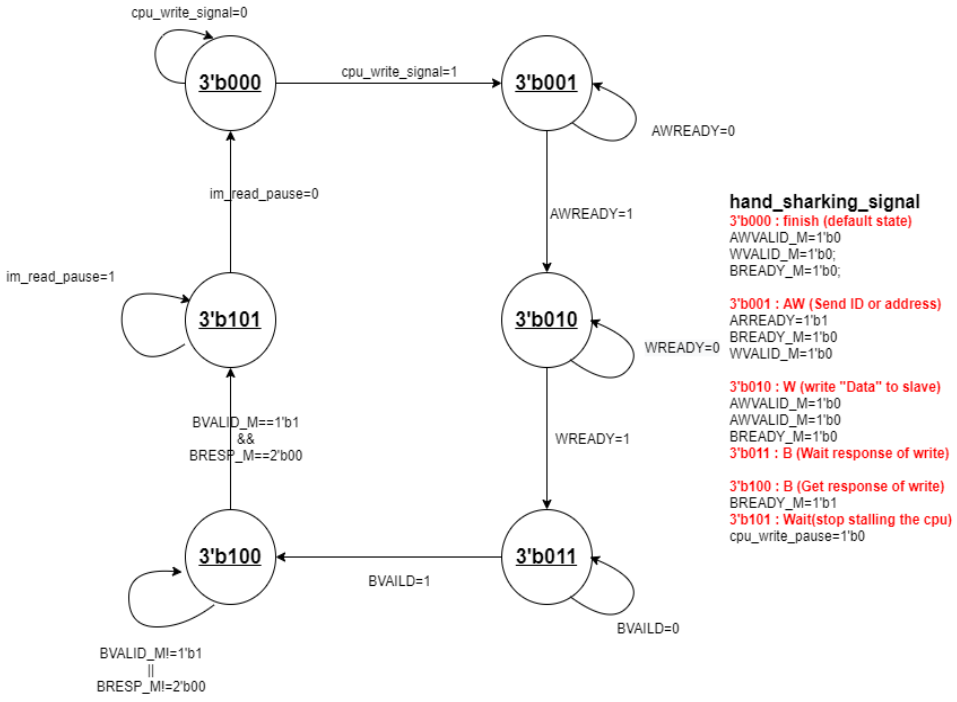
2'b11 : get "read" data
RDATA=(read or not)?DO:RDATA

在設計上，有 hand sharking 訊號才會進入下一個 state，抓取 master 端送過來的 data 則也是以達到 ready 與 valid 同時拉為 1 時才抓取，並存入 reg 中保持穩定，state 01 為 AW 或 AR 抓取 data，state 10 為 W 或 R 抓取 master 送過來要寫入 memory 的 data(LW、LB)與訪問 memory 取得正確 address 上的 data(SW、SB)，當為 read 時，則會在 state 11 時把 state 10 取得的 data 送出並拉起 RREADY 訊號，並回傳 RRESP，在 write 時，state 10 則會寫入 data，並直接進入 state 11 傳送 BVALID 訊號與 BRESP。

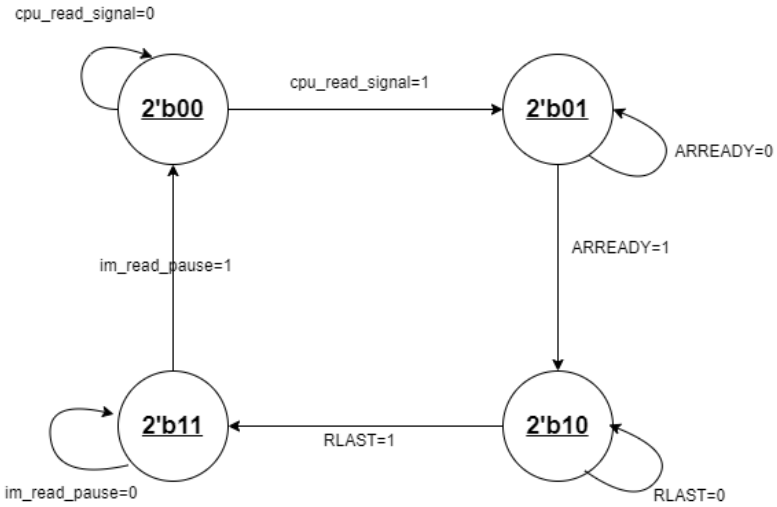
- Master:




## 1. Master_write



## 2. Master_read




**hand_sharking_signal**

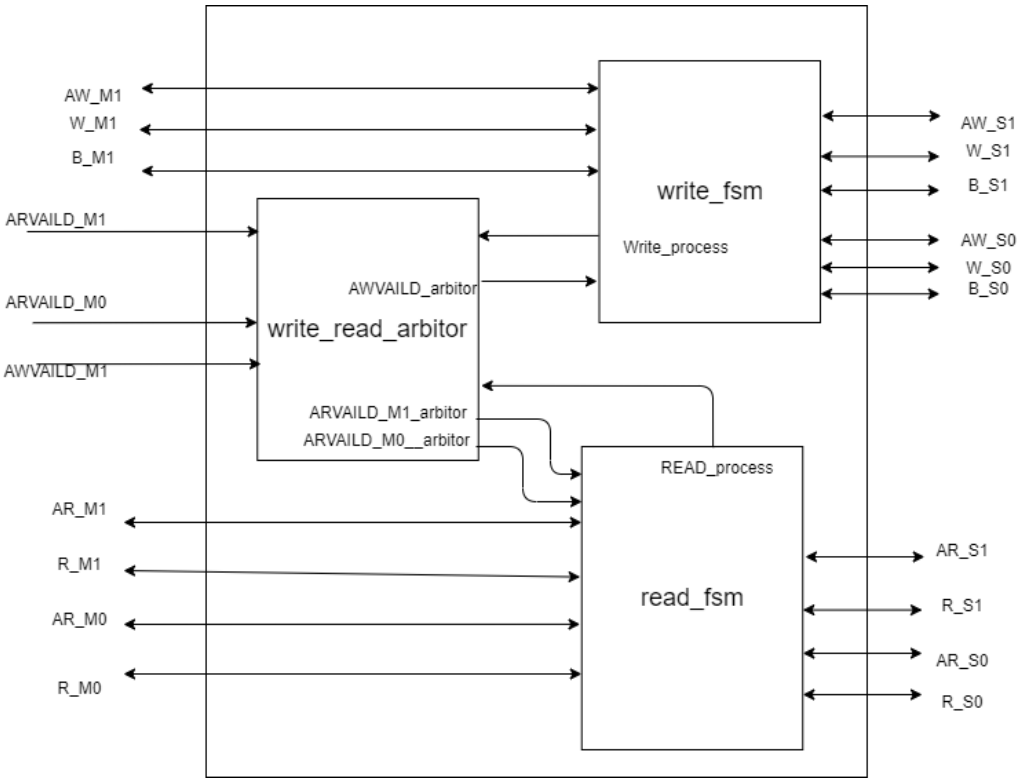
**2'b00 : finish (default state)**
ARVALID_M=1'b0
RREADY_M =1'b0

**2'b01 : AR (Send ID or address)**
ARID_M =slaveid
ARADDR_M =address
ARVALID_M=1'b1
RREADY_M =1'b0

**2'b10 :R( Get data from Slave)**
ARID_M =slaveid
ARADDR_M =address
ARVALID_M=1'b0
RREADY_M =1'b1
read_data= (RRESP_M==2'b00 && RVALID_M==1'b1)?RDATA_M:32'd0

**2'b11 : Wait(CPU get data and stall the cpu)**
read_pause_cpu=1'b0;

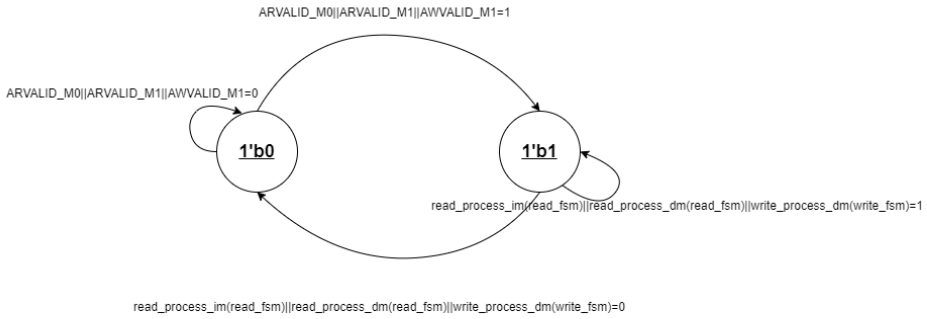
- AXI:



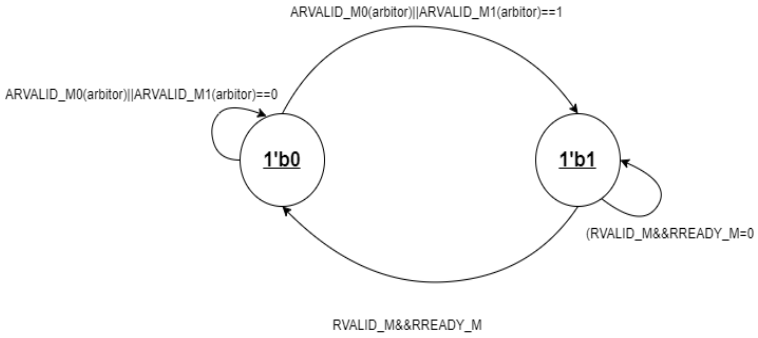
# 1. write_read_aribitor

**Arbitor_control_signal**
**1'b0 : Arbitration**
situation={ARVALID_M0,
ARVALID_M1,
AWVALID_M1
};
case(situation)
3'b000:
begin
ARVALID_M_0stage1=1'b0;
ARVALID_M_1stage1=1'b0;
AWVALID_M_1stage1=1'b0;
end
3'b001:
begin
ARVALID_M_0stage1=1'b0;
ARVALID_M_1stage1=1'b0;
AWVALID_M_1stage1=AWVALID_M1;
end
.
.
.
.
.

**1'b1 : Transfer Processing**
keep the signal

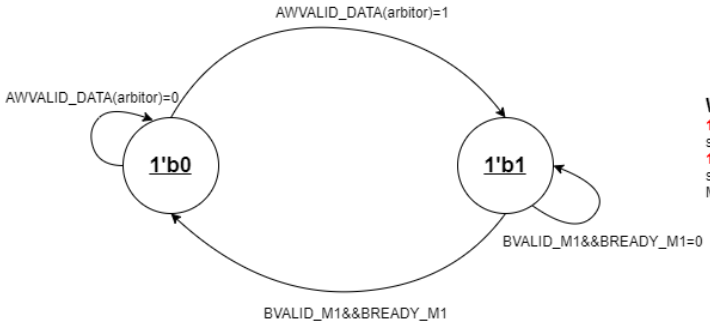



# 2. read_fsm




**Read_control_signal**
**1'b0 : Initializing**
signal all set to 0
**1'b1 : Transfer Processing**
slave_sel=(situation_decode=2'b01)?ARADDR_M1[31:16]:ARADDR_M0[31:0]
Master connect the signal with corresponding slave

# 3. write_fsm




**Write_control_signal**
**1'b0 : Initializing**
signal all set to 0
**1'b1 : Transfer Processing**
slave_select=AWADDR_M1[31:16]
Master connect the signal with corresponding slave

## Major problem & resolutions :

### Slave :

1. RVALID、ARREADY、AWREADY等訊號線為unstable?
   <sol.>等待master端送出相對應的hand sharking訊號才進入下一個狀態。
2. RID、RDATA、BID等資料為unstable?
   <sol.>在轉換與接收data的state時才做改變並將資料栓鎖進reg中，其他時候則輸出栓鎖的數值。
3. 輸入訊號線出現亂跳的現象，port與線的數值不相同?
   <sol.>檢查top.sv的接線，發現有誤，下次接完後要再細心的檢查。
4. 當訊號線讀跟寫同時送入slave端?
   <sol.>多設定一個flag訊號來做優先度的判定，以write為優先再來才是read，達到防止狀態機亂跳狀態而送錯訊號。
5. Read data時，資料沒有在cpu stall時回傳，反而多延後一個stall區間才收到?
   <sol.>當memory在讀出資料時，address送入為1個clock，送出資料(RDATA)為1個clock，所以在狀態機state10時，多停留1個clock讓資料被讀出並在state11時將資料送出。
6. 收到的address在某些時候讀錯data?
   <sol.>由於我們原先處理address為cpu做完除以4後才傳出給slave，但是在做prog0時發現master_dm會去讀slave_im的數值，造成我們事先處理的address會使AXI的abiter開錯通道，使資料讀錯並回傳，再改成由slave端統一處理時已經解決。

### AXI :

1. Output訊號unstable(aw、arvalid與aw、arready)?
   <sol.>由於在沒有選擇到要開啟通道的slave與master，jaspergold仍然會給予訊號，所以需要將其輸出的訊號做栓鎖，若有出現hand sharking的時候才能改變其訊號。

2. valid與ready訊號的先後順序?

   <sol.>測試 jaspergold時，valid與ready訊號並沒有指定先後，若valid訊號先送出，則要保持與其一同送出的資料，所以加上一個flip-flop與狀態機來做資料的栓鎖。

3. AR、R與AW、W的先後順序問題?

   <sol.>由於AW和W可以同時送出，但是AR與R有先後送出的問題，由於我們在設計通道上，但是jaspergold會提前拉起B與R的hand sharking訊號，如此，則會出現通道過早關閉，所以將read與write的hand sharking訊號放入arbiter內來做通道的開關判定。

4. Decoder條件沒有寫滿?

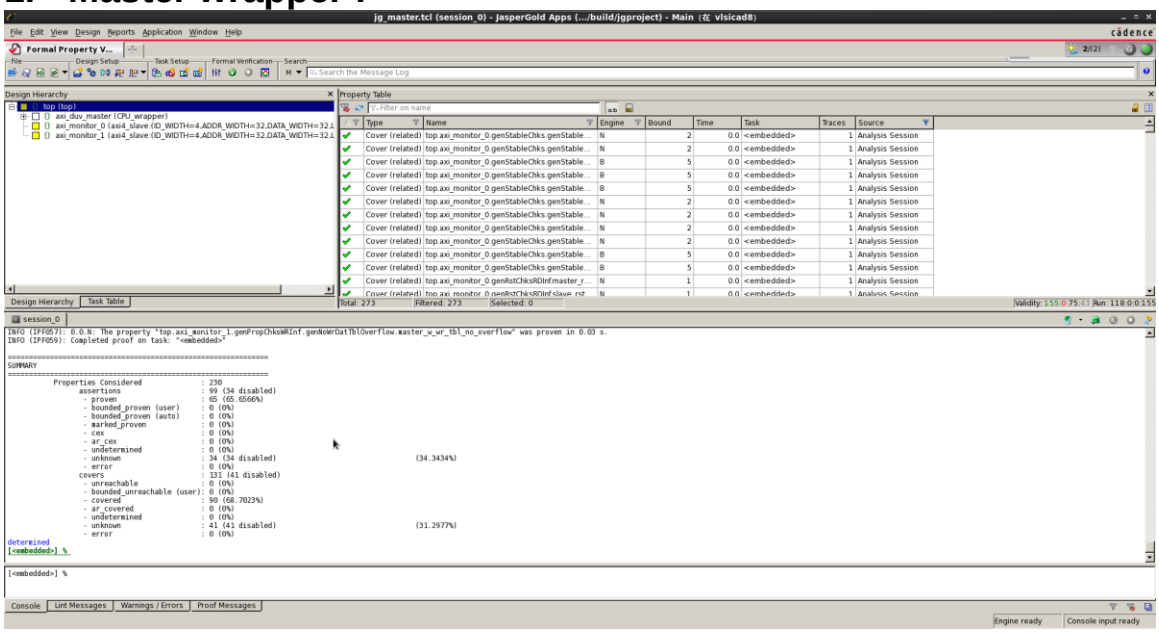
   <sol.>造成latch產生與訊號線的值可能亂跳，在問題1.有提到的栓鎖問題有解決但是仍要把沒用到的線補上，防止產生latch。

# ● Results of verification :

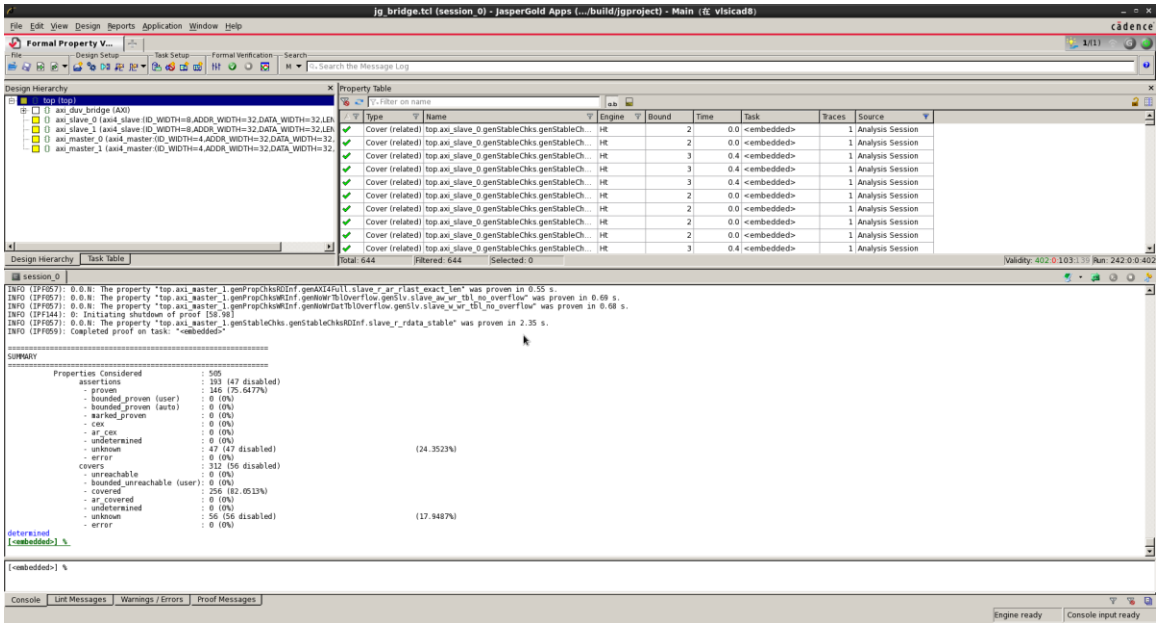
➢ Max pending number : **1**

## ✧ Jaspergold for AXI 4.0 protocol :

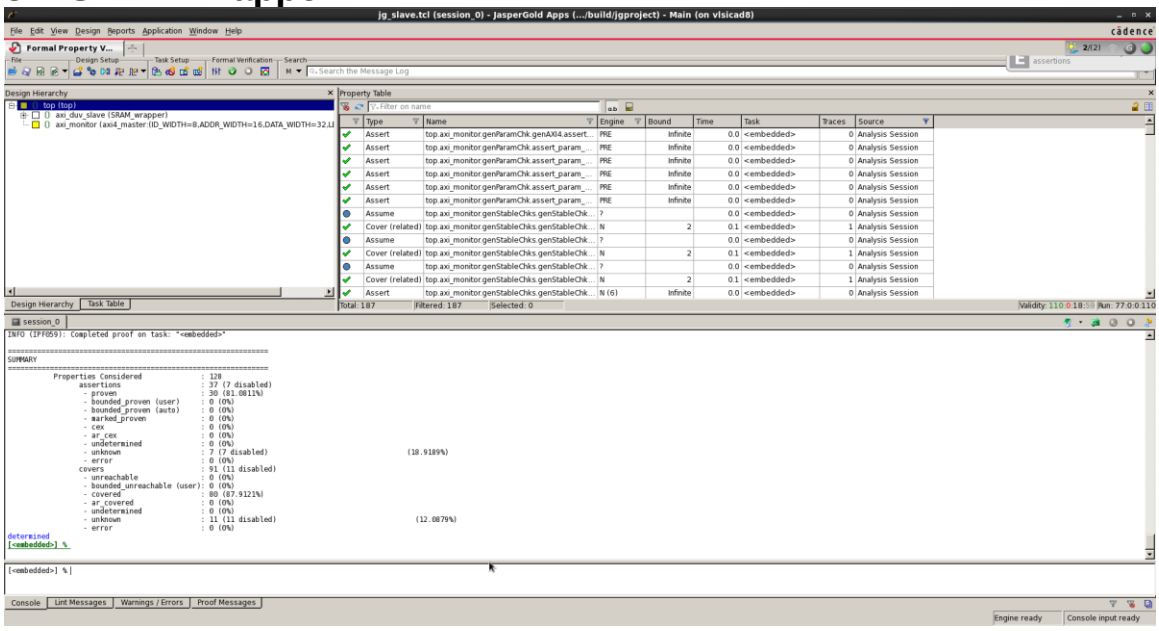
### 1. Master wrapper :



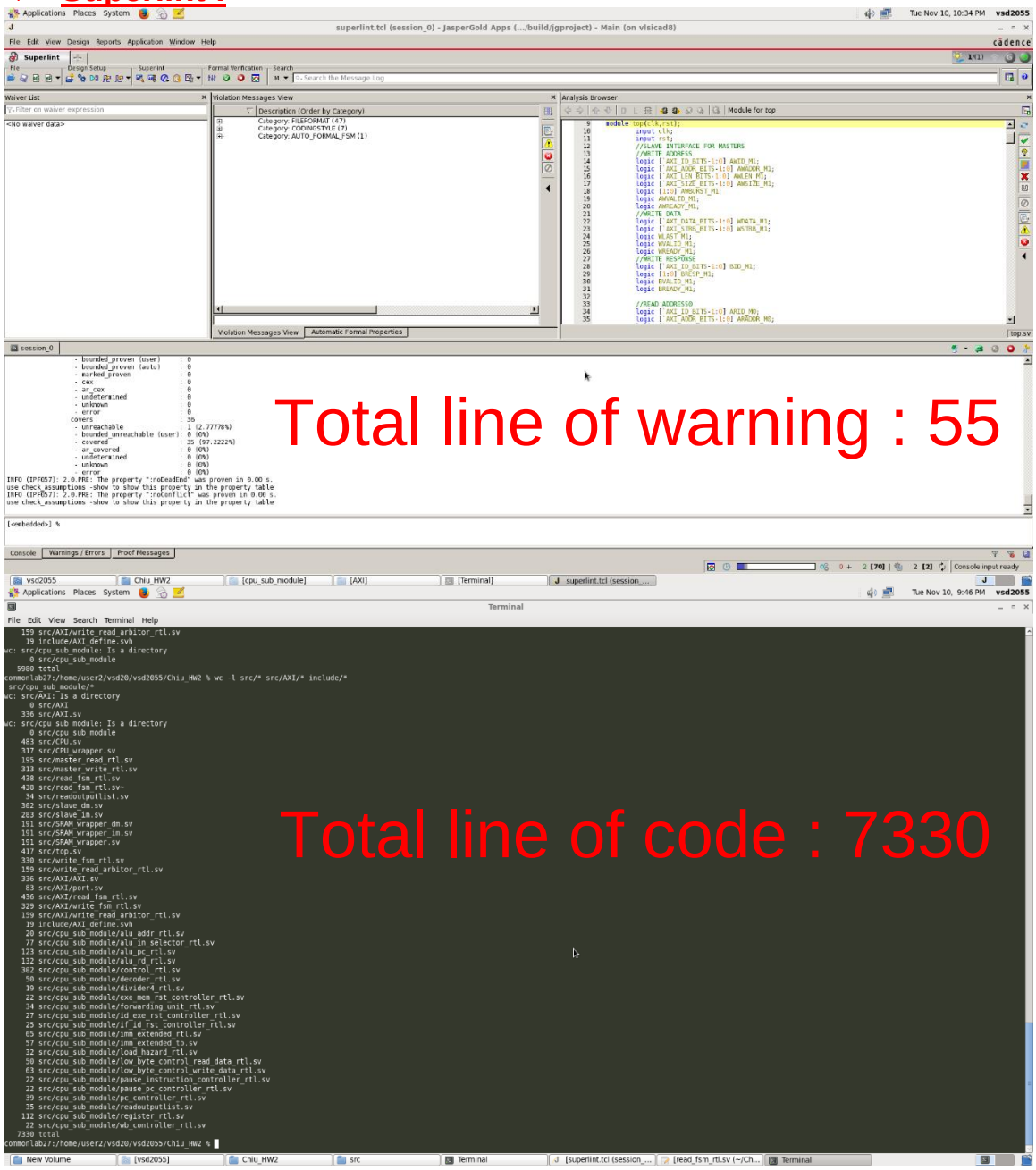
## 2. AXI :




## 3. SRAM wrapper :



## ✧ Superlint :




Total line of code : 7330
Total line of warning : 55 (47+7+1)
※Correct rate of code : (7330-55)/7330=0.992496　(99.24%)

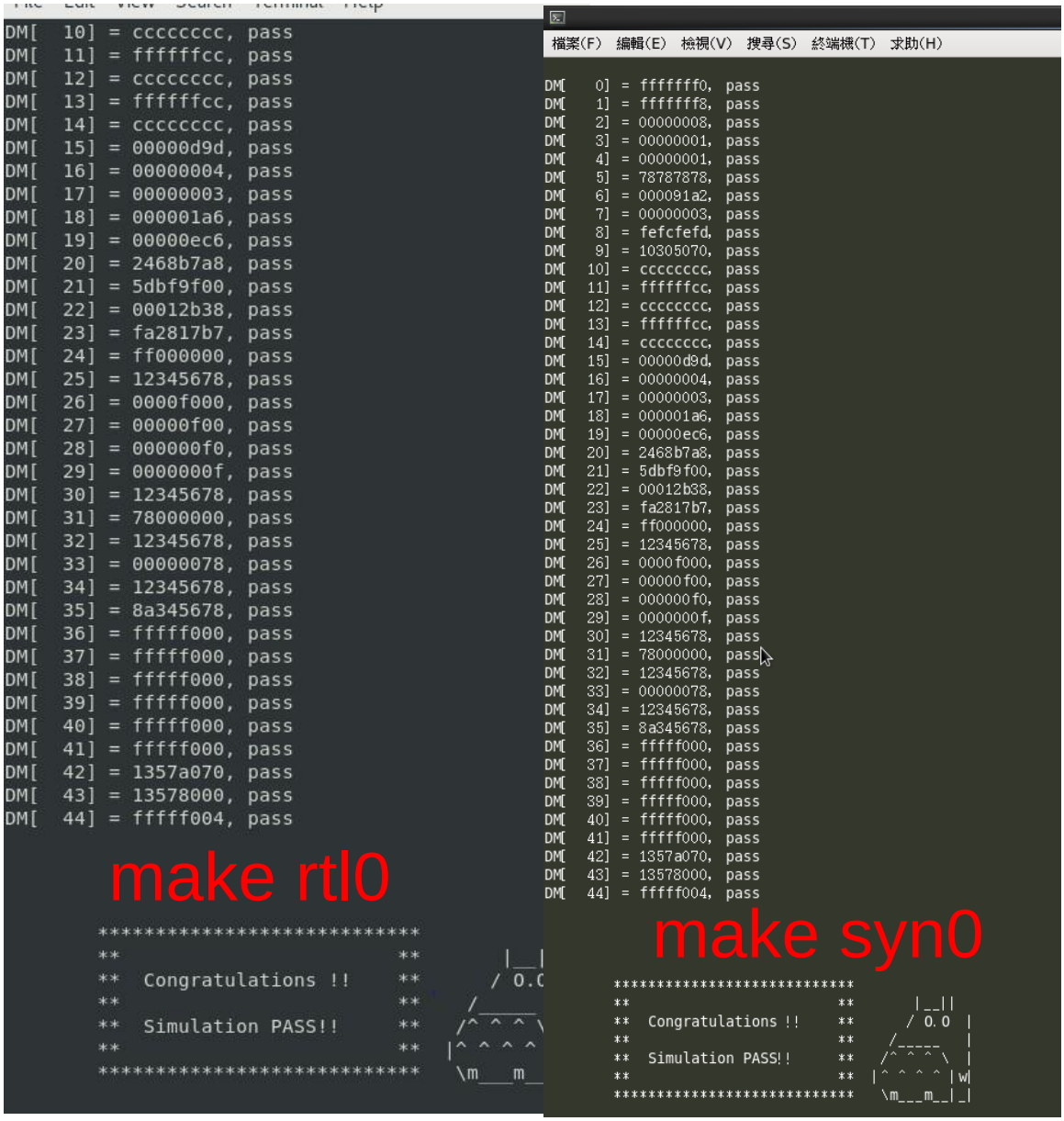
修正錯誤與 warning:

1. Master 端的送出 hand sharking 訊號有 latch 產生，有 53 條 warning，補齊訊號線在 always_comb 後，已被修正。

2. 上圖所示的 47 條 warning，為 file formate 上的問題，檔案的命名上為該 warning 出現的原因，由於檔案命名不影響，故按照其設計命名沒有做修正。

## ✧ Test for Prog0~3

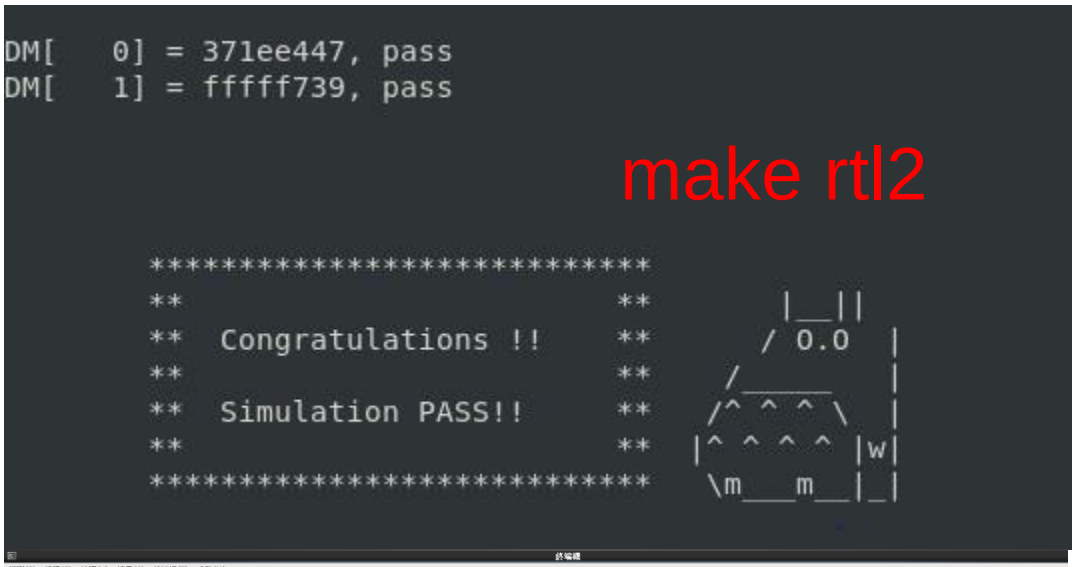
### 1. Prog0

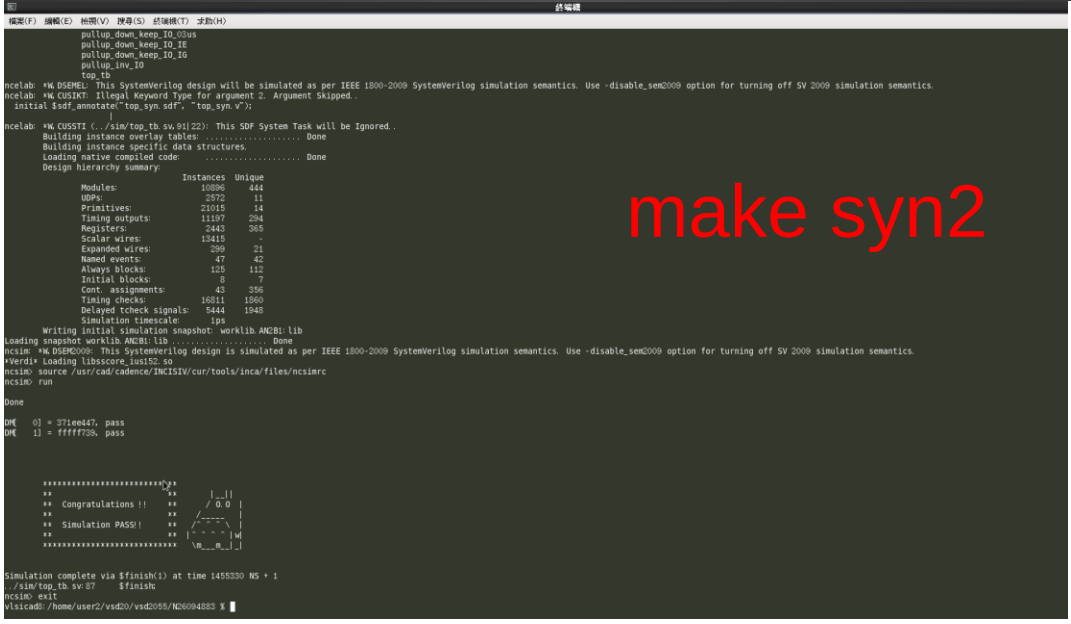


## 2. Prog1

目前仍然在觀察波行查看哪個部份的 code 有出現錯誤的現象，memory 內所存的數值處於後半段 pass 的狀態，但是前半段有排序錯誤的現象，推測其錯誤發生的原因為 master 端的收值存回出現了點問題，狀態機會回到 default state 使回送的資料為 32’ d0，下一個 clock 使 reg 寫入錯的資料 0，而排序的 instruction 抓錯比較的數值而沒有做到正確數值寫入 memory。

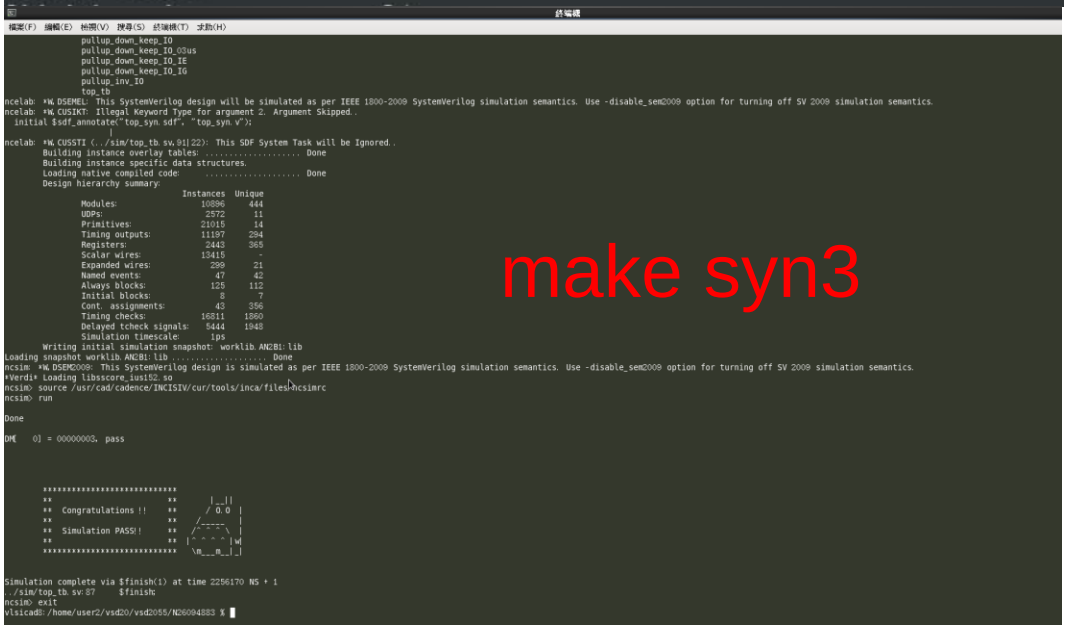
## 3. Prog2



## 4. Prog3


DM[ 0] = 00000003, pass
make rtl3
Congratulations !!
Simulation PASS!!
Simulation complete via $finish(1) at time 1158105 NS + 2
../sim/top_tb.sv:87 $finish;
ncsim> exit


make syn3
Done
DM[ 0] = 00000003, pass
Congratulations !!
Simulation PASS!!
Simulation complete via $finish(1) at time 2256170 NS + 1
../sim/top_tb.sv:87 $finish;
ncsim> exit

## ✧ Synthesis :

合成上我們使用的 clock cycle 為 20ns，結果如下附圖

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
        -sort_by group
Design : top
Version: O-2018.06
Date   : Tue Nov 10 21:23:54 2020
****************************************

 # A fanout number of 1000 was used for high fanout net computations.

Operating Conditions: BCCOM   Library: fsa0m_a_generic_core_ff1p98vm40c
Wire Load Model Mode: enclosed

  Startpoint: cpu/CPU1/stage3_register_out_reg[130]
              (rising edge-triggered flip-flop clocked by clk)
  Endpoint: cpu/CPU1/stage3_register_out_reg[95]
              (rising edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  top                enG5000K              fsa0m_a_t33_generic_io_ff1p98vm40c
  CPU                enG30K                fsa0m_a_generic_core_ss1p62v125c
  forwarding_unit    enG5K                 fsa0m_a_generic_core_ss1p62v125c
  alu_in_selector    enG5K                 fsa0m_a_generic_core_ss1p62v125c
  alu_rd             enG10K                fsa0m_a_generic_core_ss1p62v125c
  alu_rd_DW01_sub_0  enG5K                 fsa0m_a_generic_core_ss1p62v125c

  Point                                                   Incr       Path
  --------------------------------------------------------------------------
  clock clk (rise edge)                                  10.00      10.00
  clock network delay (ideal)                             0.00      10.00
  cpu/CPU1/stage3_register_out_reg[130]/CK (QDFFRBN)      0.00 #    10.00 r
  cpu/CPU1/stage3_register_out_reg[130]/Q (QDFFRBN)       0.30      10.30 f
  cpu/CPU1/fwu/exe_mem_rd_addr[2] (forwarding_unit)       0.00      10.30 f
  cpu/CPU1/fwu/U4/O (NR3)                                 0.11      10.40 r
  cpu/CPU1/fwu/U3/O (AN3B2S)                              0.13      10.54 r
  cpu/CPU1/fwu/U24/O (AN4B1S)                             0.07      10.60 f
  cpu/CPU1/fwu/U15/O (AN4B1S)                             0.15      10.75 f
  cpu/CPU1/fwu/rs2_exe_hazard (forwarding_unit)           0.00      10.75 f
  cpu/CPU1/ais/rs2_exe_hazard (alu_in_selector)           0.00      10.75 f
  cpu/CPU1/ais/U44/O (BUF1CK)                             0.38      11.13 f
  cpu/CPU1/ais/U43/O (AN2B1S)                             0.18      11.31 r
  cpu/CPU1/ais/U28/O (BUF1CK)                             0.21      11.52 r
  cpu/CPU1/ais/U55/O (AO222)                              0.19      11.71 r
  cpu/CPU1/ais/src2_data[1] (alu_in_selector)              0.00      11.71 r
  cpu/CPU1/U140/O (BUF1CK)                                0.10      11.81 r
  cpu/CPU1/ard/src2[1] (alu_rd)                           0.00      11.81 r
  cpu/CPU1/ard/U530/O (INV1S)                             0.17      11.99 f
  cpu/CPU1/ard/U529/O (BUF1CK)                            0.15      12.13 f
  cpu/CPU1/ard/U159/O (INV2)                              0.41      12.55 r
  cpu/CPU1/ard/sub_38/B[1] (alu_rd_DW01_sub_0)            0.00      12.55 r
  cpu/CPU1/ard/sub_38/U4/O (INV1S)                        0.10      12.65 f
  cpu/CPU1/ard/sub_38/U2_1/CO (FA1S)                      0.25      12.90 f
  cpu/CPU1/ard/sub_38/U2_2/CO (FA1S)                      0.21      13.11 f
  cpu/CPU1/ard/sub_38/U2_3/CO (FA1S)                      0.21      13.33 f
  cpu/CPU1/ard/sub_38/U2_4/CO (FA1S)                      0.21      13.54 f
  cpu/CPU1/ard/sub_38/U2_5/CO (FA1S)                      0.21      13.75 f
  cpu/CPU1/ard/sub_38/U2_6/CO (FA1S)                      0.21      13.97 f
  cpu/CPU1/ard/sub_38/U2_7/CO (FA1S)                      0.21      14.18 f
  cpu/CPU1/ard/sub_38/U2_8/CO (FA1S)                      0.21      14.39 f
  cpu/CPU1/ard/sub_38/U2_9/CO (FA1S)                      0.21      14.61 f
  cpu/CPU1/ard/sub_38/U2_10/CO (FA1S)                     0.21      14.82 f
  cpu/CPU1/ard/sub_38/U2_11/CO (FA1S)                     0.21      15.03 f
  cpu/CPU1/ard/sub_38/U2_12/CO (FA1S)                     0.21      15.24 f
  cpu/CPU1/ard/sub_38/U2_13/CO (FA1S)                     0.21      15.45 f
  cpu/CPU1/ard/sub_38/U2_14/CO (FA1S)                     0.21      15.67 f
  cpu/CPU1/ard/sub_38/U2_15/CO (FA1S)                     0.21      15.88 f
  cpu/CPU1/ard/sub_38/U2_16/CO (FA1S)                     0.21      16.09 f
  cpu/CPU1/ard/sub_38/U2_17/CO (FA1S)                     0.21      16.30 f
  cpu/CPU1/ard/sub_38/U2_18/CO (FA1S)                     0.21      16.51 f
  cpu/CPU1/ard/sub_38/U2_19/CO (FA1S)                     0.21      16.72 f
  cpu/CPU1/ard/sub_38/U2_20/CO (FA1S)                     0.21      16.94 f
  cpu/CPU1/ard/sub_38/U2_21/CO (FA1S)                     0.21      17.15 f
  cpu/CPU1/ard/sub_38/U2_22/CO (FA1S)                     0.21      17.36 f
  cpu/CPU1/ard/sub_38/U2_23/CO (FA1S)                     0.21      17.58 f
  cpu/CPU1/ard/sub_38/U2_24/CO (FA1S)                     0.21      17.79 f
  cpu/CPU1/ard/sub_38/U2_25/CO (FA1S)                     0.21      18.00 f
  cpu/CPU1/ard/sub_38/U2_26/CO (FA1S)                     0.21      18.22 f
  cpu/CPU1/ard/sub_38/U2_27/CO (FA1S)                     0.21      18.43 f
  cpu/CPU1/ard/sub_38/U2_28/CO (FA1S)                     0.21      18.65 f
  cpu/CPU1/ard/sub_38/U2_29/CO (FA1S)                     0.21      18.86 f
  cpu/CPU1/ard/sub_38/U2_30/CO (FA1S)                     0.20      19.06 f
  cpu/CPU1/ard/sub_38/U2_31/O (XOR3)                      0.14      19.20 f
  cpu/CPU1/ard/sub_38/DIFF[31] (alu_rd_DW01_sub_0)        0.00      19.20 f
  cpu/CPU1/ard/U926/O (AO222)                             0.21      19.41 f
  cpu/CPU1/ard/U1231/O (OR3)                              0.12      19.53 f
  cpu/CPU1/ard/alu_rd_data[31] (alu_rd)                   0.00      19.53 f
  cpu/CPU1/U123/O (AO22)                                  0.13      19.66 f
  cpu/CPU1/stage3_register_out_reg[95]/D (QDFFRBN)        0.00      19.66 f
  data arrival time                                                 19.66

  clock clk (rise edge)                                  30.00      30.00
  clock network delay (ideal)                             0.00      30.00
  cpu/CPU1/stage3_register_out_reg[95]/CK (QDFFRBN)       0.00      30.00 r
  library setup time                                     -0.11      29.89
  data required time                                                29.89
  --------------------------------------------------------------------------
  data required time                                                29.89
  data arrival time                                                 -19.66
  --------------------------------------------------------------------------
  slack (MET)                                                       10.23
```

**※於報告的附檔中有截錄 time、power、area 的 txt 檔報告**

在執行合成時會看到一些 warning 或 error 的出現，代表有些無法合成的部份或多餘的 code，例如 : 原先在 master wrapper 內 always_ff(posedge clk)begin end，這種原先在寫 code 時沒有刪除掉的 code，進行修正。

## Lesson learned :

- 周昱佑 :

在這次的作業中對於 slave 與 master 之間的溝通介面有了更多的了解，這次我們所學的為 AXI4.0 的協定，在這之中從 hand sharking 的了解是最主要的設計要點，當 valid 與 ready 同時為 1 時才能將資料當作正確的讀入或送出，所以我以這兩種訊號作為狀態機進下一個 state 的條件，這個條件設計原本我並沒有考慮到，是因為在測試 jaspergold 時，發現到這兩個訊號如果其中一者沒有拉為 1，則另一者不能放下原本拉起的訊號，由於原本由狀態機的設計是依照 valid 來在給一個 clock time 的 ready，所以在這裡修正進下一個 state 的狀態，並依照 valid 來判定資料抓取的確認訊號，並用 address 與 AW、ARID 來判定訪問正確的 slave，由於我負責設計 slave 端的部分，同時考慮到要留一個 clock time 給 memory 作讀寫的動作，所以考慮到之後合成不能有 multi-edge clock 所以給一個 stage 作 memory 訪問用，另外一個修正點在 AW、ARvalid 同時送入時要先執行哪一個，所以多用了一個變數 flag 來作判斷，總結來說最後我將送至 slave 的訊號分為 hand sharking(控制、確認)與 data(要以 register 栓鎖)，由於一開始對這個協定不是很了解，所以花了不少時間跟實驗室的同學一起研究了一下 spec 並討論中間 AXI 的設計，並了解到其中並不能用純組合電路的方式來做設計，需要考慮到 master 訪問的優先度、valid 與 ready 訊號先後問題、未被選擇的 interface 訊號保持的問題等，基本上都要加上 register 來作資料得儲存，在 AXI 最後的設計上才知道 AW 與 W 是有可能同時送出資料的，所以在 arbiter 上判斷條件又作了點修正。

在做 prog0 的 debug 時也發現很多沒注意到的，例如:address，若是在 cpu 優先處理好除以 4 再送出，會使 AXI 有判定通道開啟上的錯誤，進而讀錯資料回傳給 master，另外在 master 端送資料過來時，我的 slave 能否在我所設計的 state 栓鎖住 data 並且保持穩定，同時在 default 時也保持住直到下一次訪問才能更新栓鎖的 data，這些設計上的小缺漏在整合之後才會看到的。

同時也感謝助教的協助解答了我不少關於 AXI protocol 的問題，讓我對這次陌生的作業學習到不少。

- 杜冠勳:

此次作業學習到到了 AMBA AXI 的溝通架構 是以 handshake 的基礎上完成的而這次作業中我設計了 AXI 架構 以及 CPUWrapper 架構

在設計 AXI 架構時 ,使用了三個狀態機互相送出訊號去做溝通,三個狀態機分別為 arbitor ,讀取 ,以及寫入 ,arbiter 收到來自 master 的訊號時,會仲裁出通道使用權,而沒被選中的通道則是被控制在讀 0, 讀取 ,以及寫入狀態機 接收來自 arbitor 狀態機的訊號,線決定 slave 端口是否接上。

在設計 AXI 架構時因為過去不好的習慣導致在使用狀態機時產生了 latch 以及一些 recursive 的電路 經過這次的練習修正了許多觀念 也成功設計出來 。

在設計 CPUWrapper 架構時 因為要考慮到 cpu stall 住的問題 所以一開始就有將 CPU 放入 wrapper 中 設計 只是沒想到我把 reset 接為 0 導致等於沒有接上,但當時候只測 wrapper 時 jg 是有全過的,如附圖,但當跑完 4 個 prog 回去測 jg 時發現不會過了,原因可能如下

我們的 CPU 設計與 Wrapper 溝通 除了 read write 訊號 還有來自 wrapper 的 bus_stall 訊號線,而 bus_stall 訊號線的設計是 當 read write 訊號來時就會拉起,而 dm 優先做的關係 導致在設計 讀寫 dm 的狀態機必須多一個 state 來做為判斷是否回狀態 0(等候 read write 訊號狀態)的依據

因為如果回到狀態 0 由於 IM 在讀取指令中 造成 pipeline register 會 stall 住而造成那段時間的 dm_read 或者是 dm_write 為 1 會讓 dm 端口 一直送出 request 造成無窮迴圈 故在這次的設計中 兩個 master 端口有互傳控制訊號的情形出現導致某一些 state 無法被測到(uncover )像是 dm 端口先拉 ARVaild AWVaild 的情形 就不會有

- 原因一 CPU 會一直讀取指令 故 im_read 當初設計為恆拉為一

- 原因 2 儘管後來有改成在 load_hazard 避免再向 IM 讀取指令 但是考量到設計的架構,我們將 bus stall 統一在 wrapper 做處理, wrapper 僅會送一個控制訊號給 cpu 使用不用多作處理 才是符合 wrapper 設計 這導致我們不用在 cpu 去協調 送出 IM DM Request 的優先順序

另一種寫法則是使用控制 cpu 讀寫的順序進 wrapper ,但控制流程較為複雜,而 wrapper 還是要送訊號回來給 cpu 只是狀態機不用多一個 state 判斷 優先順序,而

是 CPU 端口處理,由於我們的設計是選前者,故會讓 CPU 只會有 IM DM 端口同時送出訊號的情形出現,或是單獨 IM 送出請求訊號的情形,造成 jg 有 uncover。
在與 CPU 整合上 須注意 bus_stall 情形必須優先於 其他影響 pipeline register 的情形出現 像是 load word 或是 jump 的情況
另外 write_data 回 register 時為了確保資料安全性,亦可在 stall 後一刻再拉起訊號 下一個 posedge clk 即可寫回。