

Lab-3 Report

葉承泓 半導體研究學院碩士班(設計部), 112501538

- ✓ Due date: 2023/10/25 23:59

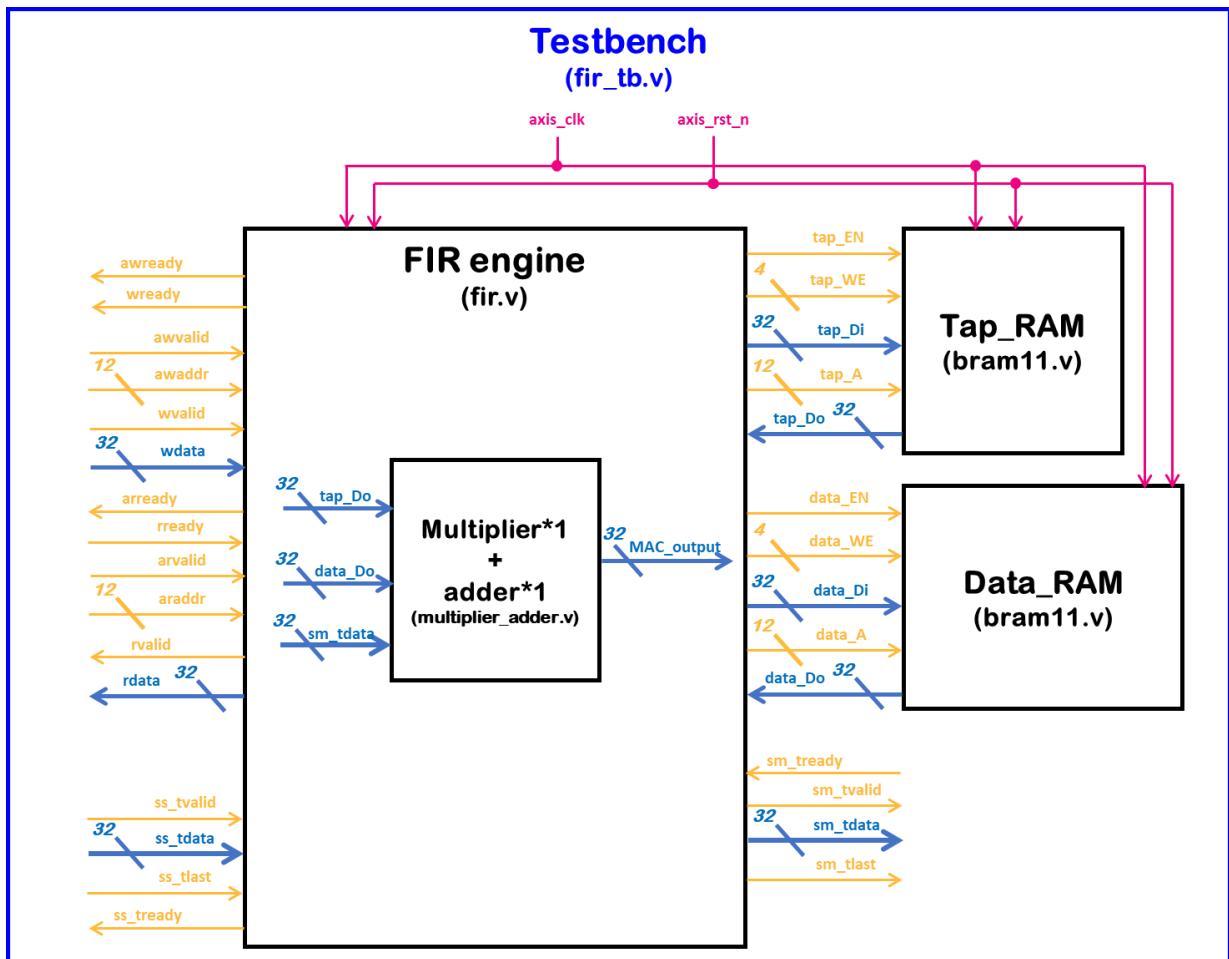
- 繳交至 eeclass 的檔案所對應到 github 上資料夾的相對應檔案：

	eeclass 上的 檔案名稱	Github 上對應路徑	說明
Waveform	fir.vcd	/lab_3/fir/fir.vcd	
Simulation .log	Simulation.l og	/lab_3/fir/xsim.log	使用 makefile 的 方式執行 simulation
Synthesis report	timing_repo rt_Max_frequ ency.txt	/lab_3/vivado/FIR_Verilog/ timing_report_Max_frequency.txt	使用 Vivado 軟體執行 synthesis
	fir_utilizatio n_synth.rpt	/lab_3/vivado/FIR_Verilog/ FIR_Verilog.runs/synth_1/ fir_utilization_synth.rpt	

● Block Diagram

在此我製作了兩種不同表示方法的 block diagram，第一種為 specify 所有 input/output ports 的 block 關係圖；第二種則是顯示 FIR engine 的實作中關於 operation 順序及流程判斷的 block 流程圖（類似 FSM 的 state diagram）。在兩種圖中皆包含了 **datapath/dataflow** 與 **control signals**，分別以相對應的顏色表示。

1. I/O ports 的關係圖(block diagram) :



(其中 datapath/dataflow 使用藍色 (最粗的線條表示主要 data_in/data_out) 、 control signals 使用橘黃色 、 global clock/reset 使用桃紅色)

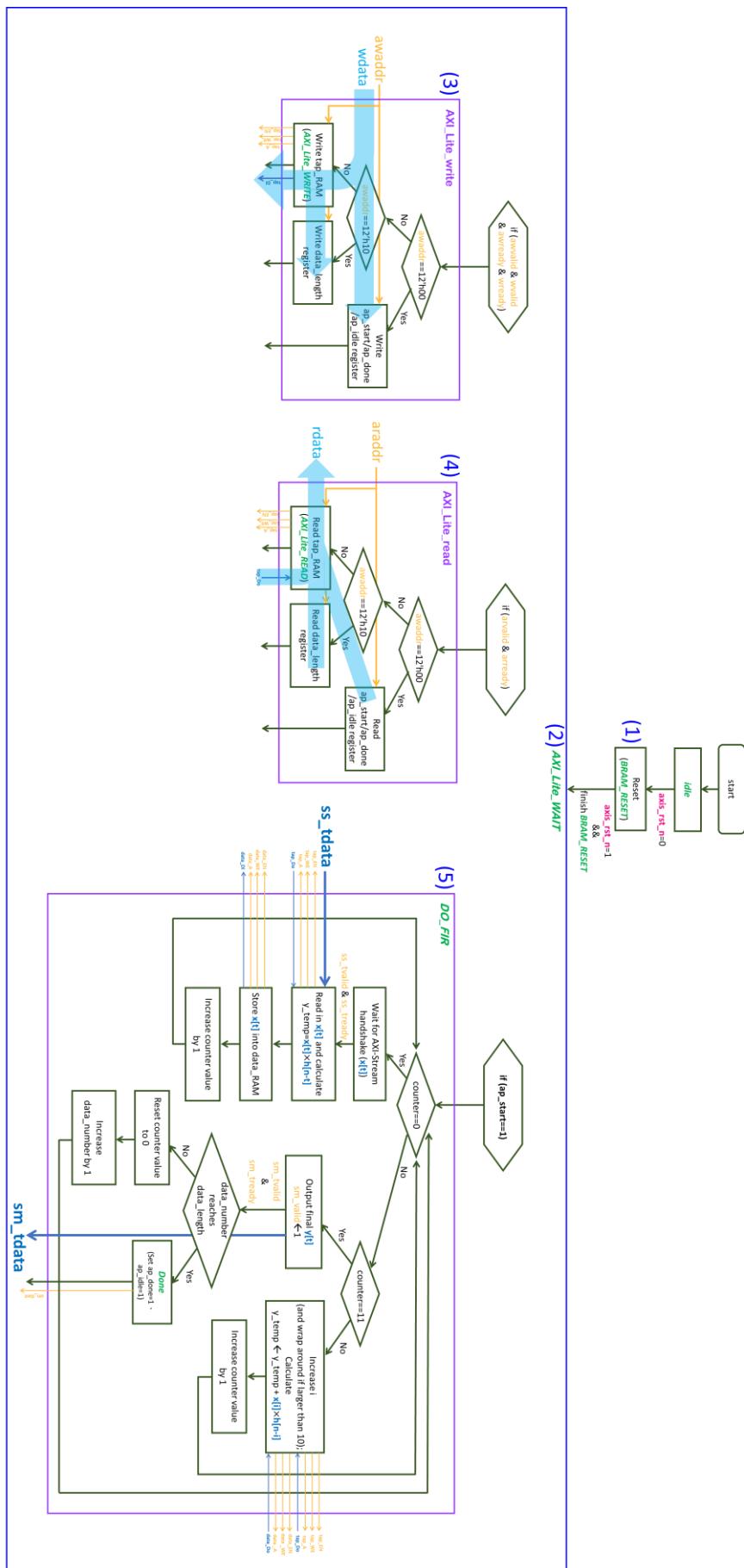
說明：上圖中的 block diagram 描寫各 block (一個 block 表示一個 module) 之間的 I/O 關係，其中我主要實作的是 FIR engine 的部分，以及在其中加入一個乘加器(MAC，即圖中之 multiplier_adder.v 的 module) (使用單獨的一個 module 並且只呼叫一次，以避免合成時自動再被合成出其他的乘法器或加法器) 。

(1) 最外層的 testbench 主要負責讀入 input file 即 golden file 中的檔案，並且以 register 的方式暫存起來，並且 generate 出 input 訊號 (包括 AXI-Stream(data_in、data_out) 、 AXI_Lite(coefficient/tap 、 ap_start 、 ap_done 的讀/寫) 等 protocol 所要求的 handshake 訊號，

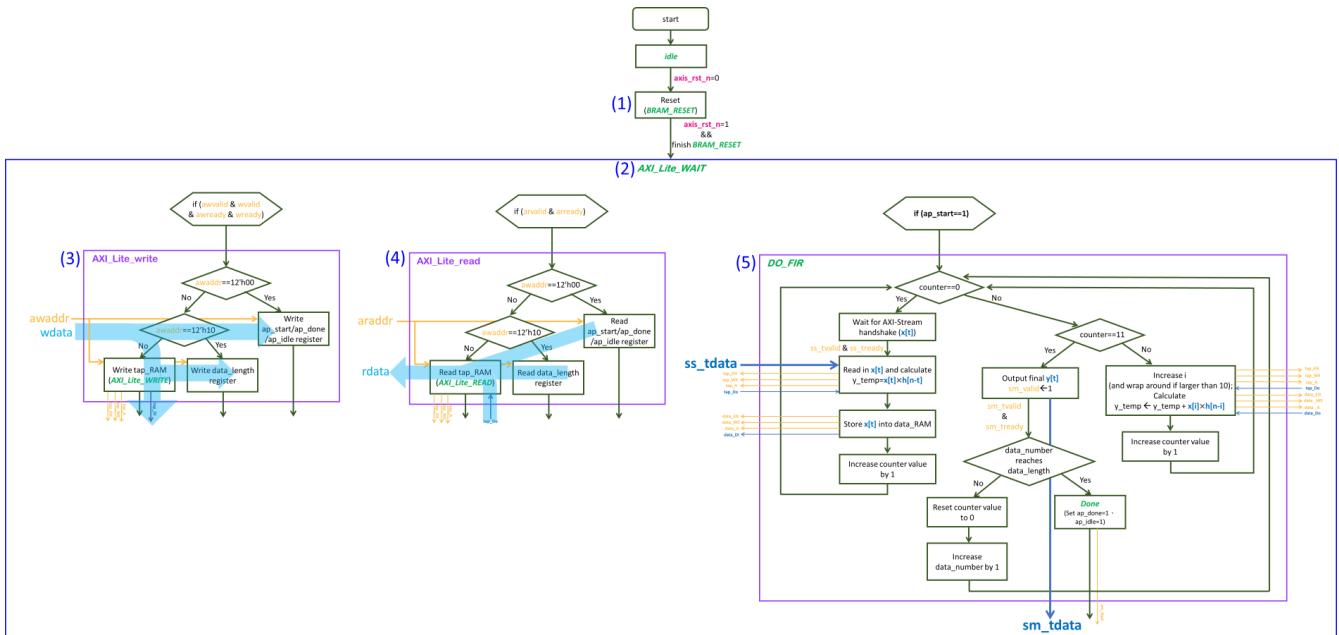
例如當作 master 時就需要 write address valid、write data valid、read address valid、read data ready；而當作 slave 時則需要 write address ready、write data ready、read address ready、read data valid 等 handshake 訊號）給 FIR engine 接收，並且將 FIR engine 的計算結果透過 AXI-Stream 的 interface 接收過來並與 golden value 作比較，也要不時 check ap_idle、ap_done 的值是否正確，最後再 report 到螢幕上。對於這個 module，我以 Github 上的範例為基底，有稍作修改，以幫助 debug（例如有在操作過程中 initialize（也就是寫好 tap_RAM 的值）後將此時 BRAM 的值取出並 print 在螢幕上，以確認有無寫入正確），以及更加合理的操作（例如在我所繳交的 fir_tb.v 中的第 306 行附近，本來的範例中是沒有「awvalid <= 0; wvalid <= 0;」的，但我認為在 tap coefficient 寫入後，應就沒有其他東西要透過 AXI-Lite 寫入，因此 awvalid、wvalid 皆應為 0，否則如果此時又想 read ap_idle 等值，就會產生同時 read/write 的衝突發生，且通常 write 應優先於 read，這將導致在 AXI-Lite 的 interface 上無法 read 到想要的值），並且有將 workbook 中對於 testbench 的要求(requirement)與流程中的每一步所對應到的程式碼區段都標記在其中，以證實有遵照著這些流程。

- (2) Tap_RAM 與 data_RAM 皆是呼叫 bram11.v 的 module 作為 behavior model，以模擬 FPGA 中 SRAM memory 的行為，此部分並無作修改
- (3) FIR engine 為本次實作的重點，其 input/output 的關係以及分組以畫在圖中，並且在 FIR 主要計算部分只使用 1 組 32-bit 的乘法器與加法器來達成計算（將每個乘加運算分配在不同 clock 做），其內部運作方式及原理會在底下做詳細的說明。
- (4) multiplier_adder.v：此為我新增的 module，屬於 combinational logic，內部主要包含一組 32-bit 的乘法器與加法器，會將 in1 與 in2 兩者相乘，並與 sum_in 相加後，形成 sum_out 的 32-bit 輸出。

2. FIR engine 中的運作/判斷流程圖(block diagram) :



(下圖為縮小圖)



(其中 datapath/dataflow 使用藍色 or 淺藍色 (最粗的線條表示主要 data_in/data_out) 、 control signals 使用橘黃色、 global clock/reset 使用桃紅色、 FSM 中的 state 名稱以 綠色斜體 表示， 紫色框框 框起來的部分則表示正在運作的模式)

說明：對應上圖的操作流程如下：

- (1)一開始 testbench 會先透過 axis_rst_n=0 來讓整個 FIR engine (底下簡稱 FIR) 做 reset ，將 ap_idle 、 ap_done 等訊號分別 reset 為 1 、 0 。接著會將 data_RAM 的值透過 11 個 clock cycle 的 write 依序 reset 成 0 (因為之後開始運作時前幾次運算會需要讀取這些 " 0 " 來當作 initial value) ，此過程是透過一個 counter 將輸入 data_RAM 的 address 依序由 $0 \rightarrow 1 \rightarrow 2 \rightarrow \dots \rightarrow 11$ ，而 write data 則維持為 0 來達到。而 tap_RAM 在此操作中先不用 reset ，因為等等會先寫入外給的 (表 testbench 給 FIR 的) tap coefficients 到這 11 個 32-bit 的 memory 中，才會正式開始進行 FIR 的運算任務。
- (2)當輸入至 FIR 的 axis_rst_n 回到 1 表示 reset phase 結束，並且在 11 個 cycle 將 data_RAM 內部的值做 reset 後，FIR 就會先進入 AXI_Lite_WAIT 的 state 等待 handshake 或是 ap_start 的指令發生，此

時會將 AXI-Lite 的接收端(slave)訊號都打開，也就是將 awready、wready、arready 設為 1。

- (3)若此時接收到 write 的指令，也就是滿足 protocol 要求 (awready、awvalid 同時為 1，即可接收 write address；並且 wready、wvalid 同時為 1，即可接收 write data) 達到 handshake (我在此處的 implementation 為等到 address、data 兩者都同時準備好，才一起進行 handshake 的 data 傳輸，因此此種情況下若有其中一方仍未準備好 (valid+ready)，則另一方會等他準備好再同時傳輸)，則 FIR 會去解析傳進來的 configuration address，若為 0x00 則 map 到 FIR 內部的 ap register (存{ap_idle, ap_done, ap_start})；若為 0x10 則 map 到 FIR 內部的 data_length register；若為其他位址則表示要寫至 tap_RAM 中，因此會將 tap_RAM 的 EN 變成表示要讀/寫、WE 寫為 4'b1111 表示所有 bytes 都要寫、address 為 awaddr-12'h20 (計算得到的)、data_in 就將 wdata 的值寫入，即可成功將 coefficient 值寫入相對應的 BRAM 位置，其 data flow 與 control signal 如上圖所示。
- (4)若此時接收到 read 的指令，也就是滿足 protocol 要求 (arready、arvalid 同時為 1，即可接收 read address) 達到 handshake，則 FIR 會去解析傳進來的 configuration address，若為 0x00 則 map 到 FIR 內部的 ap register (讀出{ap_idle, ap_done, ap_start}的值(擴展成 32bits 輸出))；若為 0x10 則 map 到 FIR 內部的 data_length register；若為其他位址則表示要由 tap_RAM 中取出相對應位址的儲存值，因此會將 tap_RAM 的 EN 變成表示要讀/寫、WE 寫為 4'b0000 表示所有 bytes 都沒有要覆寫、address 為 araddr-12'h20 (計算得到的)，並且進入 AXI_Lite_READ 的 state 等待一個 cycle，即可在 tap_RAM 的 data_out port 得到 data。最後將要輸出的 data 放在 rdata port，並將 rvalid 拉起來後，等待 rready 的 handshake 達成，即完成 data transfer，再返回 AXI_Lite_WAIT state 即可，其 data flow 與 control signal 如上圖所示。

(5)若 testbench 透過 AXI_Lite 的 interface 將 configuration address 中對應到 ap_start 位址的 register 值改填為 1，表示 FIR 已 initial 完畢，要開始正式運作了！因此，當 FIR detect 到 ap_start 為 1，則進入 DO_FIR state，此 state 主要運算 $y \leftarrow y + h * x$ 的 FIR 運算式，但由於只限定使用一個乘法器及加法器執行運算，並且 h coefficient 與 past x value 分別儲存在 tap_RAM 與 data_RAM 中，這些 BRAM 也只能一次最多讀一筆 data，因此我將運算式拆解成 11 個步驟，並且將 temp value of y 儲存起來。由上述可知，在計算過程中會使用到 tap_RAM 的 access、data_RAM 的 access，以及透過 AXI-Stream interface 從外界得到 x 值，以及透過另一個平行運作的 AXI-Stream interface 向外界送出運算結果。在上圖中有將這些 interface 的相關 control 訊號及 dataflow 標記出來，其中最粗的藍線為最主要的 dataflow。運算過程中使用到類似 shift register 的概念，但我是透過改變 BRAM 的 address pointer 來達成，並非真正將所有 data 搬移（這樣要花費太多 cycle 了，因為一個 cycle 只能 read or write only one datum），詳細的做法會在底下的 Describe operation 小節做說明。

● Describe operation

在此我按照 testbench 的操作順序依序介紹我有實作到的重要的幾點操作步驟，並且搭配模擬結果的波形：

1. How to reset data_RAM?

在 axis_rst_n 作 reset 後，會進入 BRAM_RESET 的 state，在這裡會有一個 counter，從 0 開始每次遞增 $(4)_{10} = (100)_2$ ，並將這個 counter 的 value 紿 data_A，也將 data_EN 及 data_WE 全部拉起來、data_in 輸入 0 (reset value)，則就可以依序對 address 0 至 10 的位置做 reset 了。而每次 address 是遞增 $(4)_{10} = (100)_2$ 而非 $(1)_{10}$ 主要是因為在 trace BRAM 的 behavior code 後，會發現輸入的 data_A address 會先被向右 shift 2 bit 後才做為 address (因為其為 byte-address，但我們希望把每 4 個 bytes (即 32bits) 握作一

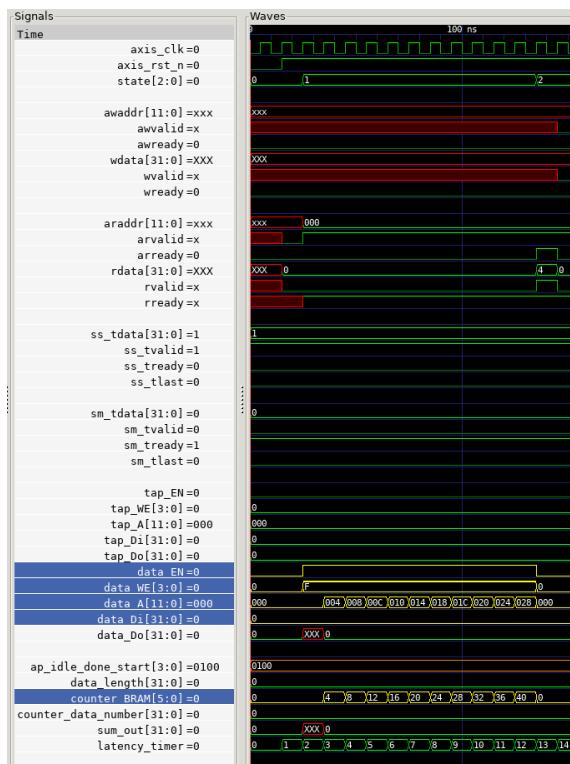
個 data 的儲存位置，故每個 data 的 address 之間就差了 4 個 byte addresses)，因此若希望最終的 address 增加 1，則在輸入端(data_A)需要增加 $(4)_{10}=(100)_2$ 。最後直到 11 個 data 的位置都 reset 好後（即 counter_BRAM== $(40)_{10}$ ），就可以跳出這個 state 了。此部分的程式碼如下所示：

```

if(counter_BRAM==6'd40) begin
    next_state=AXI_Lite_WAIT;
    awready_before_FF=1;
    wready_before_FF=1;
    arready_before_FF=1;
    next_counter_BRAM=0;
end
else begin
    next_state=BRAM_RESET;
    awready_before_FF=0;
    wready_before_FF=0;
    arready_before_FF=0;
    next_counter_BRAM=counter_BRAM+4;
end

```

其模擬波形圖為：



圖中黃色波形即為相關 signal，可看到 counter_BRAM (data_A) 每次逐漸增加 4 的過程，直到輸入的 address=40 為止。在 reset 之後使用 testbench 直接到 BRAM 的內部 memory 取出值（並非透過 interface 的方式），並 print 在螢幕上，截圖如下：

```

Data_RAM[ 0 ] = 00000000
Data_RAM[ 1 ] = 00000000
Data_RAM[ 2 ] = 00000000
Data_RAM[ 3 ] = 00000000
Data_RAM[ 4 ] = 00000000
Data_RAM[ 5 ] = 00000000
Data_RAM[ 6 ] = 00000000
Data_RAM[ 7 ] = 00000000
Data_RAM[ 8 ] = 00000000
Data_RAM[ 9 ] = 00000000
Data_RAM[ 10 ] = 00000000

```

可知 Data_RAM 的確有被 reset 至全為 0。 (因為後面運算過程中會需要用到這些 reset 後的 value 作為 initial state，故必須要有此 state)

2. How to receive tap parameters (coefficients) and place into SRAM?

進入 AXI_Lite_WAIT state 後，testbench 會透過 AXI_Lite 的 write 及 write address channel 寫入 ap_start、data_length、tap_RAM 的值等相關資訊，在這裡我為了滿足 protocol 的要求，實作方式是先等到 awvalid 及 wvalid 都偵測到 HIGH 後，才將 awready 與 wready 拉為 HIGH，如此一來可確保此時可同時接收到 address 及 data (雖然 protocol 並沒有要求需要同時接收，也是可以先接收其中一個 channel 再接收另一個，然後再對這筆(address, data) pair 作處理，但這樣會需要有額外的 FIFO register 暫存，並且甚至可能需要存多個，會比較複雜，因此在這裡我就先使用最基礎的方式——等到確認兩個 valid 皆為 1 後再送出 ready，即可同時收值並立即處理，這種作法也是滿足 protocol 的，只是會需要花比較多 cycle)。當確認 awvalid、awready、wvalid、wready 皆為 1 後，即可開始分析 address 的位置，透過 configuration address map 可知：當 address=0x00 時表示要寫入 FIR 內部的 ap register；當 address=0x10 時表示要寫入 FIR 內部的 data_length register；而其他 address 則是代表要寫入 tap_RAM 中。因此我透過 if else 的 statement 來判斷要存到哪些地方，若要存到 tap_RAM 則進行如下圖的設定：

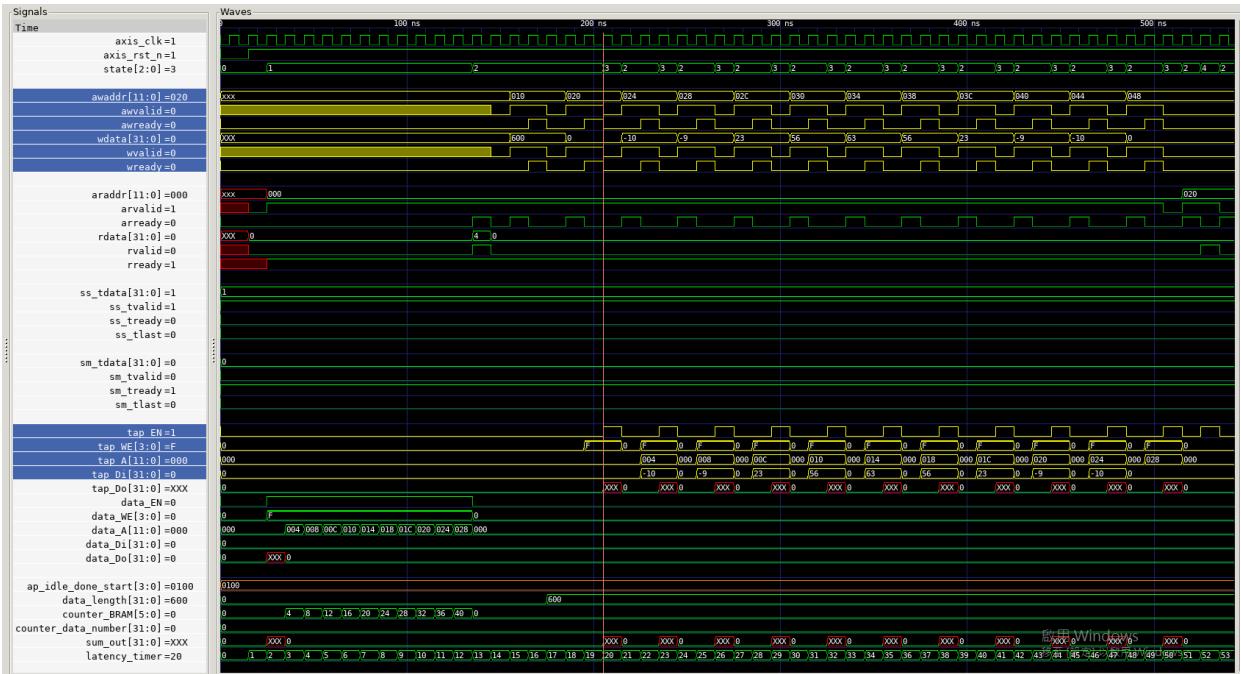
```

next_state=AXI_Lite_WRITE;
tap_EN_reg = 0;
tap_WE_reg = 4'b1111;
tap_Di_reg = wdata;
tap_A_reg = awaddr-12'h20;

next_ap_idle_done_start=ap_idle_done_start;
next_data_length=data_length;

```

其中的 tap_Di 直接接上 wdata 的線，而 address 則是因為 configuration address map 的關係而需要先減去 12'h20。此情形下 ap register 與 data_length register 維持原值。其模擬波形圖如下圖所示：



圖中黃色波形即為相關 signal，可看出 tap 的 write operation 實際是在 handshake 完成後的下一個 cycle (FSM 進入 AXI_Lite_WRITE state) 時完成的，這是因為在 handshake 的前一個 cycle 仍無法確認是否要寫入，必須等 handshake 完畢後才表示確認要寫入，並將相關數據寫入 tap_RAM。在寫入所有的 tap_RAM 之後(約 520ns 處)使用 testbench 直接到 BRAM 的內部 memory 取出值(並非透過 interface 的方式)，並 print 在螢幕上，截圖如下：

```
At 550ns: (for check and debugging)
Tap_RAM[ 0] = 00000000
Tap_RAM[ 1] = ffffff6
Tap_RAM[ 2] = ffffff7
Tap_RAM[ 3] = 00000017
Tap_RAM[ 4] = 00000038
Tap_RAM[ 5] = 0000003f
Tap_RAM[ 6] = 00000038
Tap_RAM[ 7] = 00000017
Tap_RAM[ 8] = ffffff7
Tap_RAM[ 9] = ffffff6
Tap_RAM[10] = 00000000
```

可知這些外界透過 AXI-Lite 所輸入的數據的確有被寫入 tap_RAM 中。

3. How to read back and validate tap_RAM values?

在第 2. 點的最後有使用 testbench「直接到 BRAM 的內部 memory 取出值」的方式來證實 tap_RAM 的確有被寫入想要的值。在此是使用 testbench 透過 AXI_Lite 的 read address channel 來要求 FIR 提供這些 global address (對應到 tap_RAM 中的 address) 所對應的值。在 testbench 中的 config_read_check 這個 task 即負責這個任務，它會將要讀取的 address 放在 araddr port 上，並且拉起 arvalid，並且等待直到 handshake 成立，同時將 rready (因為其為 read data channel 的 slave) 拉起，等待 read data channel 的 handshake，將 FIR 傳出的 data 值與 golden value 進行比較 (會先 mask 掉這 32bit 中沒有要比較的 bits)，以確認 FIR 的正確性。而在 FIR 內部，主要是實作 AXI_Lite 的 read address channel 與 read data channel 的 interface 相關訊號，首先在 AXI-Lite_WAIT state 時，FIR 將 already 拉為 1，並且等待直到 arvalid 拉起來，即可進行 handshake，解析 address 的位置，再透過 configuration address map 可知：當 address=0x00 時表示要讀出 FIR 內部的 ap register (擴充成 32bits 的 width)；當 address=0x10 時要讀出 FIR 內部的 data_length register；而其他 address 則是代表要讀出 tap_RAM 相對應位址的 tap 值。因此我透過 if else 的 statement 來判斷要讀取哪些地方的值，若要讀取 tap_RAM 則進行如下圖的設定：

```

next_state=AXI_Lite_READ;
tap_EN_reg = 0;
tap_WE_reg = 4'b0000;
tap_DI_reg = 0;
tap_A_reg = araddr-12'h20;

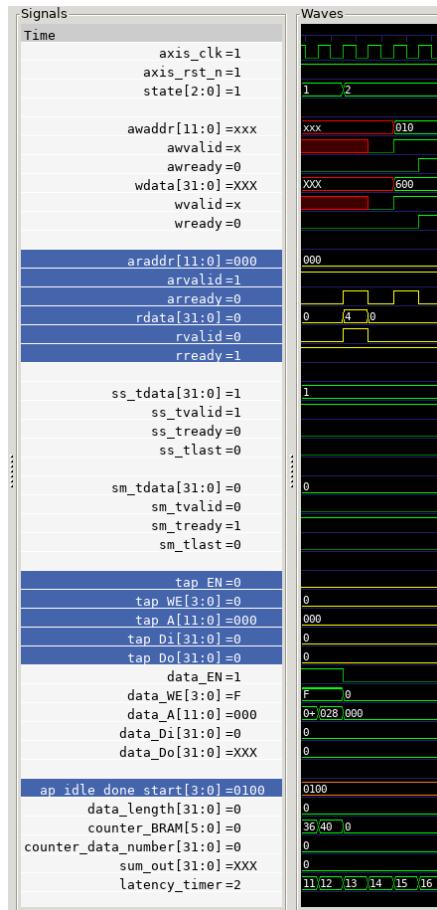
next_ap_idle_done_start=ap_idle_done_start;

rdata_reg=0;
rvalid_reg=0;

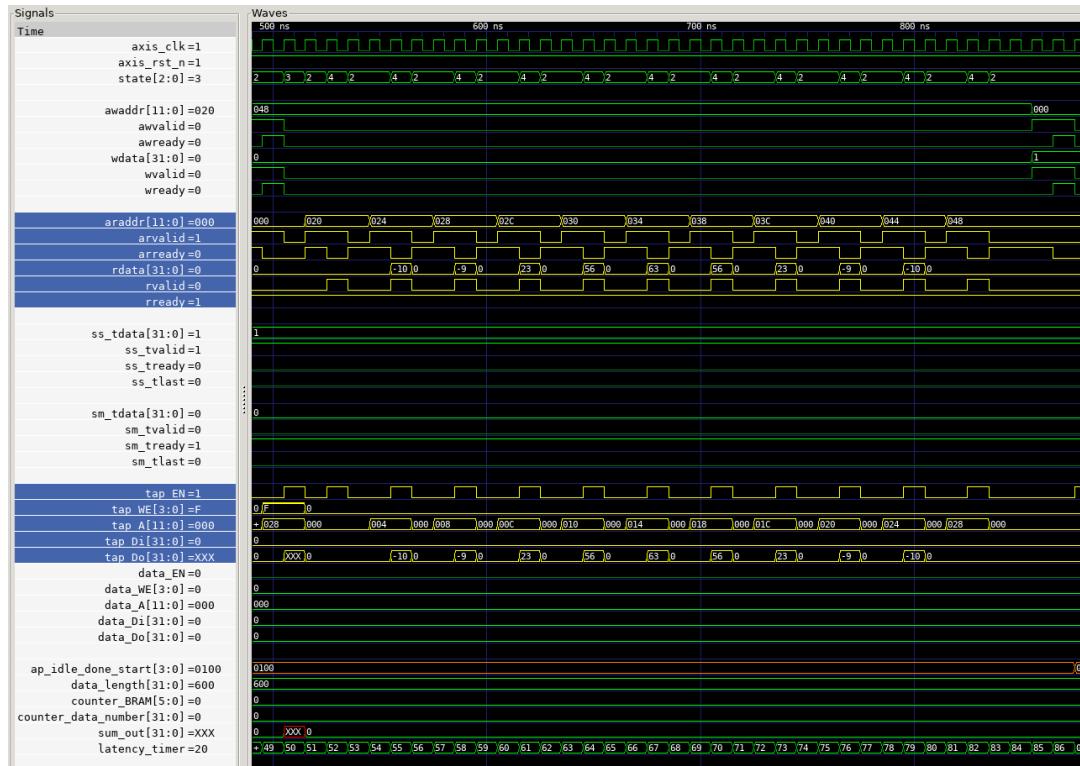
```

其中的 tap_EN 先令為 0 (因為在 address 輸入後的下一個 cycle 才會出現相對應的 data，而查看 BRAM 的 behavior model 後發現 EN 訊號是在 data 出來、要收 data 的那個 cycle 才需要為 1，在輸入 tap_A 時不需將 EN 設為 1)，而 address 則是因為 configuration address map 的關係而需要先減去 12'h20。其模擬波形圖如下圖所示：

(Read address=0x00 的例子)



(Read address \neq 0x00 的例子)



圖中黃色波形即為相關 signal，可看出 tap 的 read operation 主要分成兩種：第一種是要 read ap register，此情況下可以在 handshake 發生時就已準備好 data，這是因為即使 handshake 仍未達成（因為 clock rising edge 尚未到來），但若已偵測到 arvalid 與 already 同時為 1，可預測接下來的 clock rising edge 就會發生 handshake，因此我們可以預先用此時的 address 來找出 register 的值，並且準備好在 data channel，同時將 rvalid 拉到 1，下一個 clock rising edge 到來時即可直接完成 data channel 的 handshake 與 data 傳輸，如圖中所示，此時的 ap register 所存的值為 $(0100)_2 = (4)_{10}$ ，因此在 rvalid 為 1 時的 rdata 值的確為 4；第二種：若是要 access BRAM，則在 handshake 發生時的 cycle 才將 address 傳進 BRAM 中，因此需要等到 handshake 完成後的下一個 cycle (FSM 進入 AXI_Lite_READ state) 時才能放到 data channel 上（此時的 tap_EN 才變為 1，因為此 cycle 下才需要 read 出 data），並完成 data channel 的 handshake。

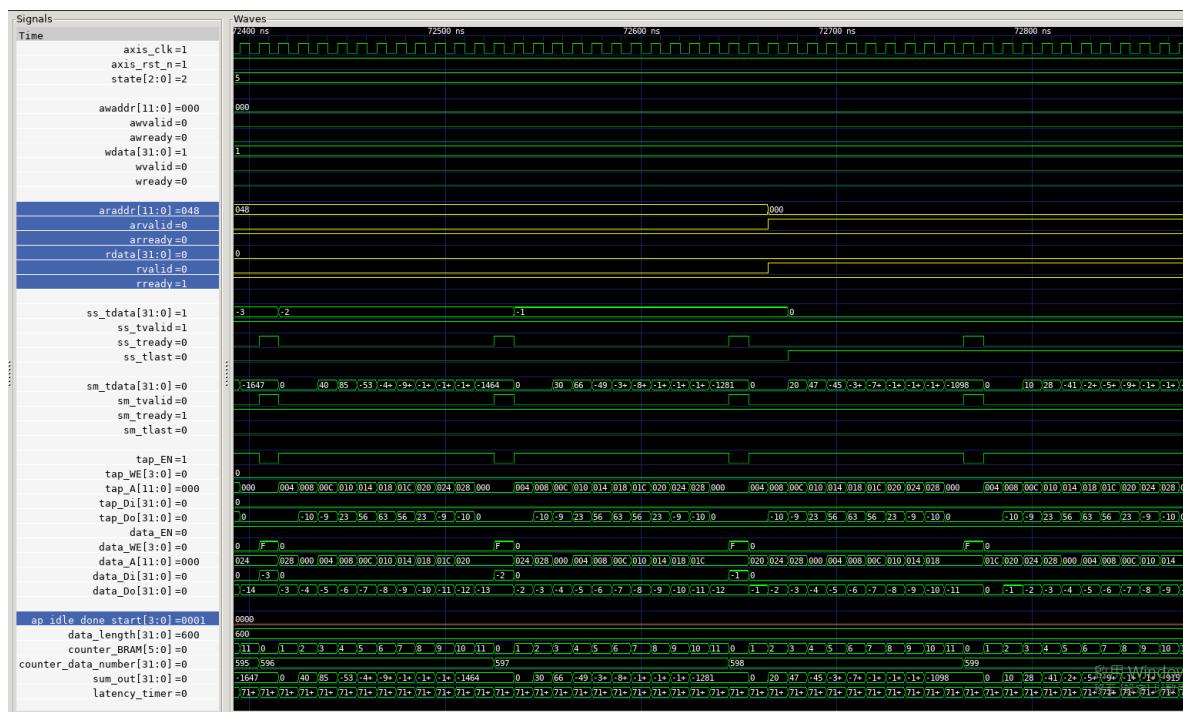
在 testbench 的 config_read_check task 會將 FIR 輸出值與 golden value 的比較結果 print 在螢幕上，截圖如下：（因為分屬在不同區塊(在不同時間 check 這些值)，因此分開截圖）

```
-----Start simulation-----
----Start the data input(AXI-Stream)----
OK: exp =        4, rdata =        4
----Start the coefficient input(AXI-lite)----
Check Coefficient ...
OK: exp =        0, rdata =        0
OK: exp =      -10, rdata =      -10
OK: exp =       -9, rdata =       -9
OK: exp =       23, rdata =       23
OK: exp =       56, rdata =       56
OK: exp =       63, rdata =       63
OK: exp =       56, rdata =       56
OK: exp =       23, rdata =       23
OK: exp =       -9, rdata =       -9
OK: exp =      -10, rdata =      -10
OK: exp =        0, rdata =        0
Tape programming done ...
OK: exp =        0, rdata =        0
[PASS] [Pattern      598] Golden answer:
-----End the data input(AXI-Stream)-----
[PASS] [Pattern      599] Golden answer:
OK: exp =        2, rdata =        6
OK: exp =        4, rdata =        4
```

在圖中可發現倒數第 2 筆資料的理想值(exp.)為 2，而 FIR (rdata) 則輸出 6，但仍算是 OK，這是因為有 mask 的關係——在此只想要測試 ap_done (位於 LSB 算來 bit 2 的位置) 的值是否為 1 (對 ap register 值的 contribution

為 $(2)_{10}$ ），但由於此時的 ap register 值為 $(110)_{10}$ （ap_idle 在此時也為 1，對 ap register 值的 contribution 為 $(4)_{10}$ ），且它們位於相同 address 下（0x00），導致螢幕上 print 出 rdata=6，但因為只有要 check ap_done 的正確性，所以只 mask 在 LSB 算來 bit 2 的位置，故仍為過關！

在 FIR 正在 working 的過程中，由於需要不斷 access tap_RAM 的值出來運算，因此沒空讓 AXI-Lite 的 interface 去 access tap_RAM 的 address，但 ap register 與 data_length register 是放置在 FIR 內部，因此仍可 access 並在 read data channel 中讀出，如下波形圖所示：

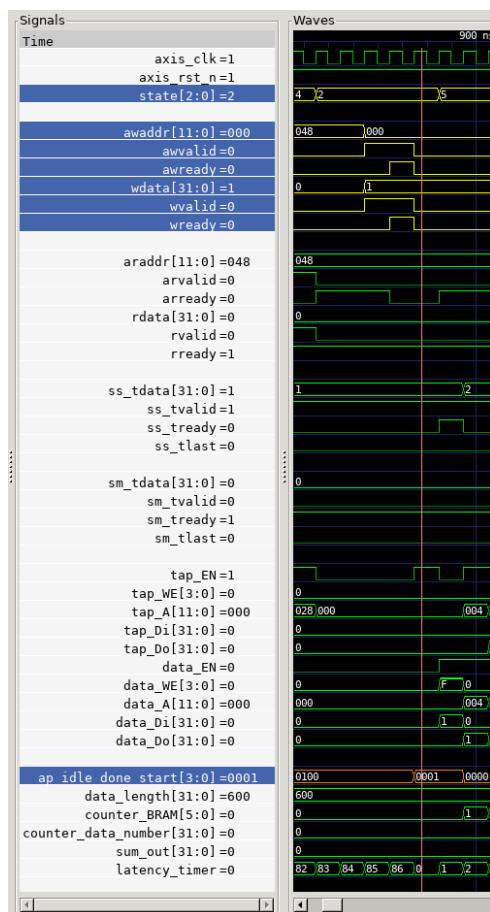


由圖中右側可看出，在 testbench 將 arvalid 拉起來後，形成 handshake，由於 address 為 0x00（表示 ap register），因此可以 access 並將其值（0000）放在 read channel 中，rvalid 也拉為 1，故可正常 access 到 ap_done 的值為 0。

4. How to start the kernel (FIR) operation?

在 testbench 透過 AXI-Lite 的 write channel 將 ap register、data_length register、tap_RAM 都 initialize 好，並且透過 AXI-Lite 的 read channel 確認這些 memory 所存的值正確無誤後，就可以開始進行 FIR 的運算了！因

此，testbench 透過 AXI-Lite 的 write channel 的 handshake 將 ap_start 的值寫為 1，當 FIR 在 AXI-Lite_WAIT state 時發現 ap_start=1（在其他 state 時 ap_start=1 並無法使其開始 operation，例如正在 DO_FIR 又看到 ap_start=1，但還在執行現在的 FIR 任務，因此不會理會它），則會進入 DO_FIR 的 state，開始讀進 data_in 並透過 access tap_RAM 及 data_RAM 來計算出 data_out。ap_start 被 program 為 1 以及導致 state 轉換的過程如下模擬波形圖所示：

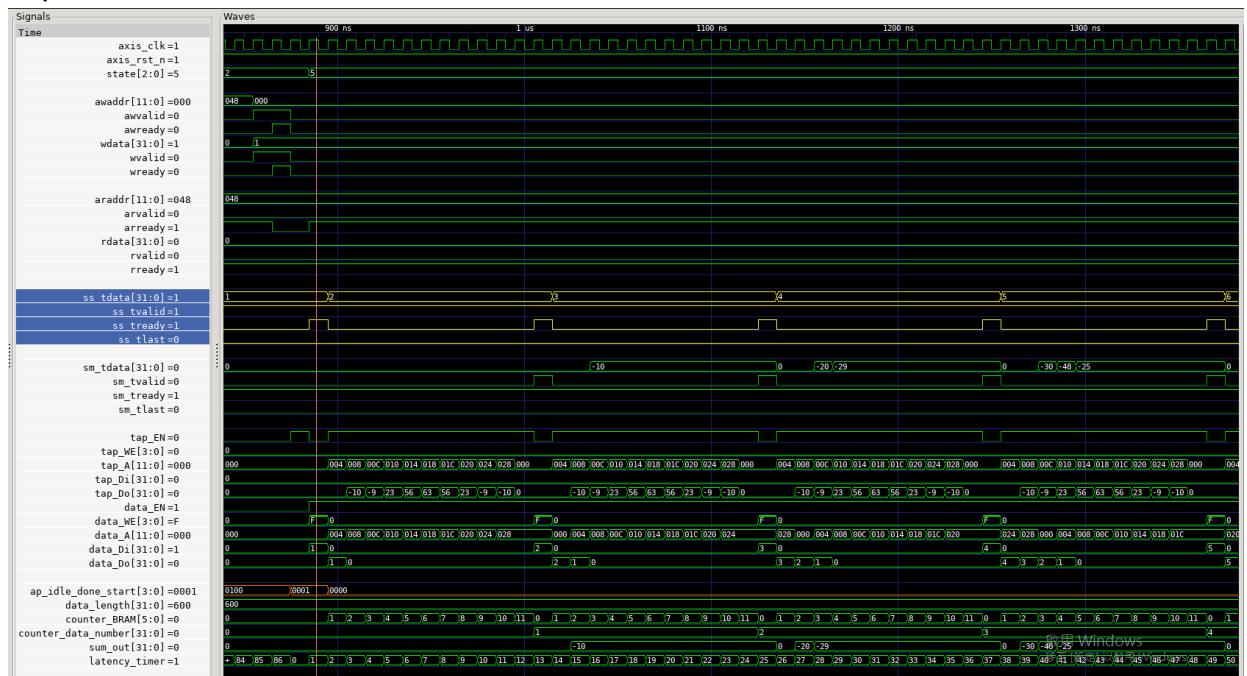


圖中黃色波形即為相關 signal，可看出 AXI-Lite 的 write address 及 write channel 的 handshake 使 ap register 的 ap_start 被 program 成 1，當 FIR detect 到後，state number 由 2（表示 AXI_Lite_WAIT state）轉變為 5（表示 DO_FIR state）。

5. How to get AXI-Stream Data_in?

在 FIR 成功進到 DO_FIR state 後，會將 ap_start 寫回 0，並將 ap_idle 寫入 0，表示正在 working 中，故此時的 ap register 為 {ap_idle, ap_done,

`ap_start}=000`。接著會開始做 FIR 運算，每筆 `data_out` 主要來自底下流程：先從 AXI-Stream input Data_in `x[t]` 進來 → access tap_RAM 拿到 tap coefficient `h[i]`，以及 access data_RAM 拿到 past `x[n-i]` value → 計算 $y[t] \leftarrow y[t] + h[i] * x[t-i]$ → 重複 11 次迭代後得到此筆 output `y[t]` → 透過 AXI-Stream output Data_out `y[t]` 出去，即完成 1 筆資料的運算與輸出。如此流程重複進行 `data_length` 次，即可計算出所需要的 `data_length` 個 output 了。在「AXI-Stream input Data_in `x[t]` 進來」的這個步驟是使用 ASI-Stream 的 interface，FIR 在每筆 output data 的運算過程一開始會先從外界讀入新的 input data `x[t]`，再開始進行迭代運算。過程為先將 `ss_tready` 設為 1，表示要準備讀入新的一筆 data，等到 `ss_tvalid` 也為 1 就表示 handshake 成立，即可將 `ss_tdata` 讀進來了！接著由於要進行 11 個 cycle 的運算，要等這筆 output data 運算完畢再讀入下一筆新的 input data，故需要先將 `ss_tready` 改為 0，先暫時關閉 channel 的 data 流動，以避免一次進來過多 input data 而沒成功接收到。其模擬波形圖如下圖所示：



圖中黃色波形即為相關 signal。由於 DO_FIR 持續了 7000 多個 cycle，因此在此僅放大剛開始的幾個 cycle 以方便觀察，可看出在每次 `ss_tready` 拉起來並完成 handshake 後，過了 11 個 cycle 才又再度拉起 `ss_tready`，中間

的過程為下方第 6.點所述的 output data 運算過程。當每個 output 運算完正在輸出時，才會將 ss_tready 拉起來，開放下一筆 input data 進來。

6. How to access shiftram(data_RAM) and tapRAM to do computation?

此部分為 FIR 的核心運算，也就是

$$y[t] = \sum_{i=0}^{10} h[i] \times x[t-i]$$

我有想出下列的演算法，類似 shift register 的操作方式，但是透過移動 address pointer 的位置而非搬移 data，因此可以省去將 data 在 data_RAM 中搬移的過程（每搬移一筆 data 都需要耗費許多 cycle，因此要只在必要時才 access memory 較好）：

(1) 原始狀態下 BRAM 中的資料為：

tap_RAM	h[0]	h[1]	h[2]	h[3]	h[4]	h[5]	h[6]	h[7]	h[8]	h[9]	h[10]
data_RAM	0 (x[-11])	0 (x[-1])	0 (x[-2])	0 (x[-3])	0 (x[-4])	0 (x[-5])	0 (x[-6])	0 (x[-7])	0 (x[-8])	0 (x[-9])	0 (x[-10])

其中 data_RAM 皆已在 BRAM_RESET 的 state 中 reset 成 0，為了符合上方的 FIR 運算式，將這些值視為 $x[-11] \sim x[-1]$ （也就是假設過去的 input 值皆為 0），並且依照圖中順序擺放。

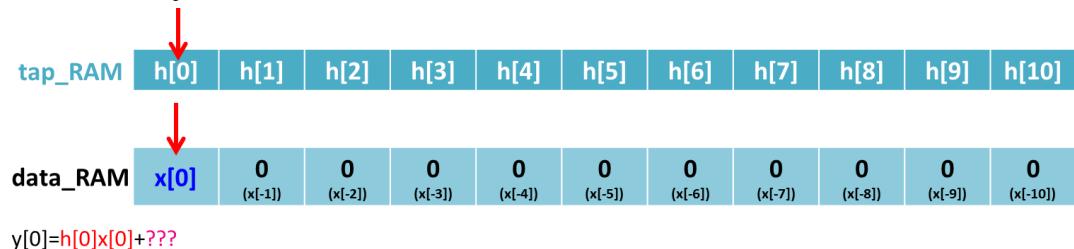
(2) 計算第一筆 output data $y[0]$ ：

A. 先透過 AXI-Stream input $x[0]$ 進來，並將其儲存在 data_RAM 的 address 0 位置（可透過 data_RAM 的 address pointer 當作 data_A port，而 data_Di 則使用 handshake 進來的 ss_tdata port 直接接過去即可），因此 BRAM 存值變為如下圖所示：

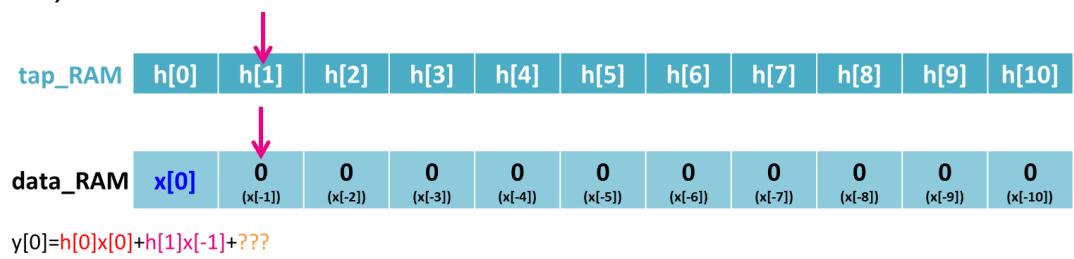
tap_RAM	h[0]	h[1]	h[2]	h[3]	h[4]	h[5]	h[6]	h[7]	h[8]	h[9]	h[10]
data_RAM	x[0]	0 (x[-1])	0 (x[-2])	0 (x[-3])	0 (x[-4])	0 (x[-5])	0 (x[-6])	0 (x[-7])	0 (x[-8])	0 (x[-9])	0 (x[-10])

也要在此時將 output data sm_tdata 的值 reset 到 0。此時的 counter_BRAM reset 為 0，並在下一個 cycle 時增加 1。

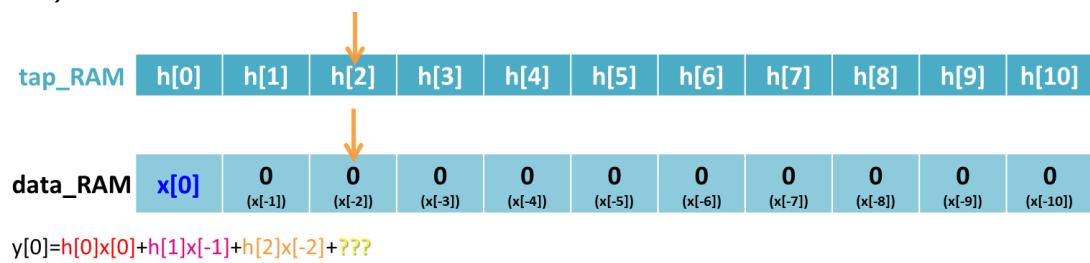
B. 先計算 $sm_tdata + h[0]x[0]$ (利用 multiplier_add.v 的乘加器 module) · 並在 clock rising edge 時將值放回 sm_tdata 中 · 此時的 address pointer 如紅色箭頭所指 : (此時 counter_BRAM 為 1 · 並在下一個 cycle 時增加 1。)



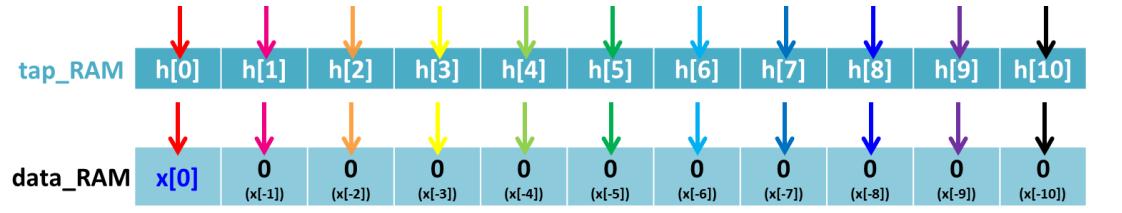
C. 接著將 address pointer 指向下一個 address · 並計算 $sm_tdata + h[1]x[-1]$ (利用 multiplier_add.v 的乘加器 module) · 並在 clock rising edge 時將值放回 sm_tdata 中 · 此時的 address pointer 如桃紅色箭頭所指 : (此時 counter_BRAM 為 2 · 並在下一個 cycle 時增加 1。)



D. 接著將 address pointer 再指向下一個 address · 並計算 $sm_tdata + h[2]x[-2]$ (利用 multiplier_add.v 的乘加器 module) · 並在 clock rising edge 時將值放回 sm_tdata 中 · 此時的 address pointer 如橘色箭頭所指 : (此時 counter_BRAM 為 3 · 並在下一個 cycle 時增加 1。)



E. 重複將 address pointer 再指向 BRAM 的下一個 address，並更新 sm_tdata 的值，直到最後即可得到

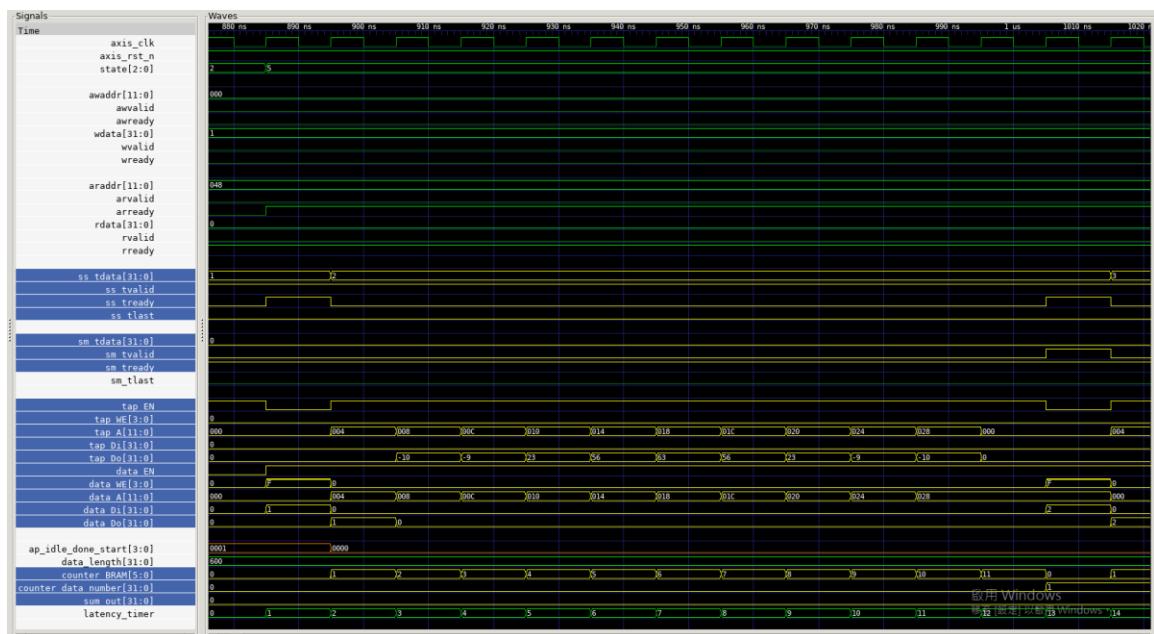


$$y[0] = h[0]x[0] + h[1]x[-1] + h[2]x[-2] + h[3]x[-3] + h[4]x[-4] + h[5]x[-5] + h[6]x[-6] + h[7]x[-7] + h[8]x[-8] + h[9]x[-9] + h[10]x[-10]$$

即有第一筆 output data $y[0]$ 的正確算式。此時 counter_BRAM 為 11。

F. 當 counter_BRAM=11 時，表示已經迭代 11 次運算，因此此時的 y 值已算完可以輸出了。由於此時 $y[0]$ 已存在 sm_tdata port 上（因為在運算過程中 sm_tdata 為 idle，故剛好能直接拿來當作運算過程中的暫存器使用，且如此一來最後就可以直接由此 port 輸出），故此時只需要將 sm_tvalid 改為 1 即可與外界進行 AXI-Stream 的 handshake 輸出此值。並將 counter_data_number 的值增加 1，表示已完成一筆數據。

G. 第一筆數據的相對應波形圖如下：



圖中黃色波形即為相關 signal，而圖中的 sum_out 即為乘加器 module 的 output port 訊號。

(3) 計算第二筆 output data $y[1]$ ：

- A. 先透過 AXI-Stream input $x[1]$ 進來，並將其儲存在 data_RAM 的 address 10 位置（此位置為前一筆資料運算過程中 data_RAM 的 address pointer 最後所在的位置，也就是前面圖中的黑色箭頭處），因此 BRAM 存值變為如下圖所示：

tap_RAM	$h[0]$	$h[1]$	$h[2]$	$h[3]$	$h[4]$	$h[5]$	$h[6]$	$h[7]$	$h[8]$	$h[9]$	$h[10]$
data_RAM	$x[0]$	$0_{(x[-1])}$	$0_{(x[-2])}$	$0_{(x[-3])}$	$0_{(x[-4])}$	$0_{(x[-5])}$	$0_{(x[-6])}$	$0_{(x[-7])}$	$0_{(x[-8])}$	$0_{(x[-9])}$	$x[1]$

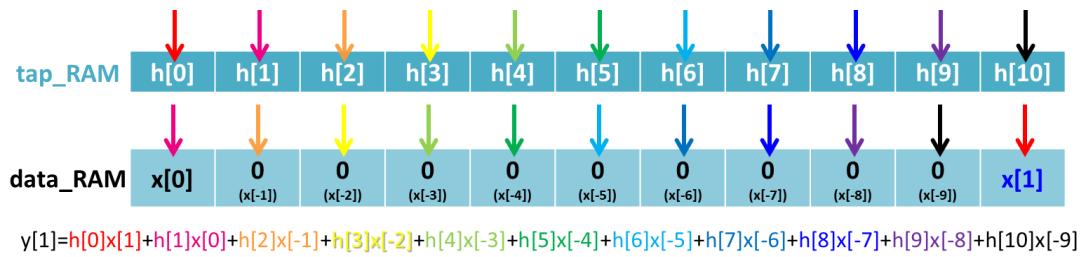
另外，要在此時將 output data sm_tdata 的值 reset 到 0。此時的 counter_BRAM reset 為 0，並在下一個 cycle 時增加 1。

- B. 接著將 tap_RAM 的 address pointer 指向下一個 address，但 data_RAM 的 address pointer 維持在原位（前面圖中的黑色箭頭處），並計算 $sm_tdata + h[0]x[1]$ （利用 multiplier_add.v 的乘加器 module），並在 clock rising edge 時將值放回 sm_tdata 中，此時的 address pointer 如紅色箭頭所指：（此時 counter_BRAM 為 1，並在下一個 cycle 時增加 1。）

tap_RAM	$h[0]$	$h[1]$	$h[2]$	$h[3]$	$h[4]$	$h[5]$	$h[6]$	$h[7]$	$h[8]$	$h[9]$	$h[10]$
data_RAM	$x[0]$	$0_{(x[-1])}$	$0_{(x[-2])}$	$0_{(x[-3])}$	$0_{(x[-4])}$	$0_{(x[-5])}$	$0_{(x[-6])}$	$0_{(x[-7])}$	$0_{(x[-8])}$	$0_{(x[-9])}$	$x[1]$

$$y[1]=h[0]x[1]+???$$

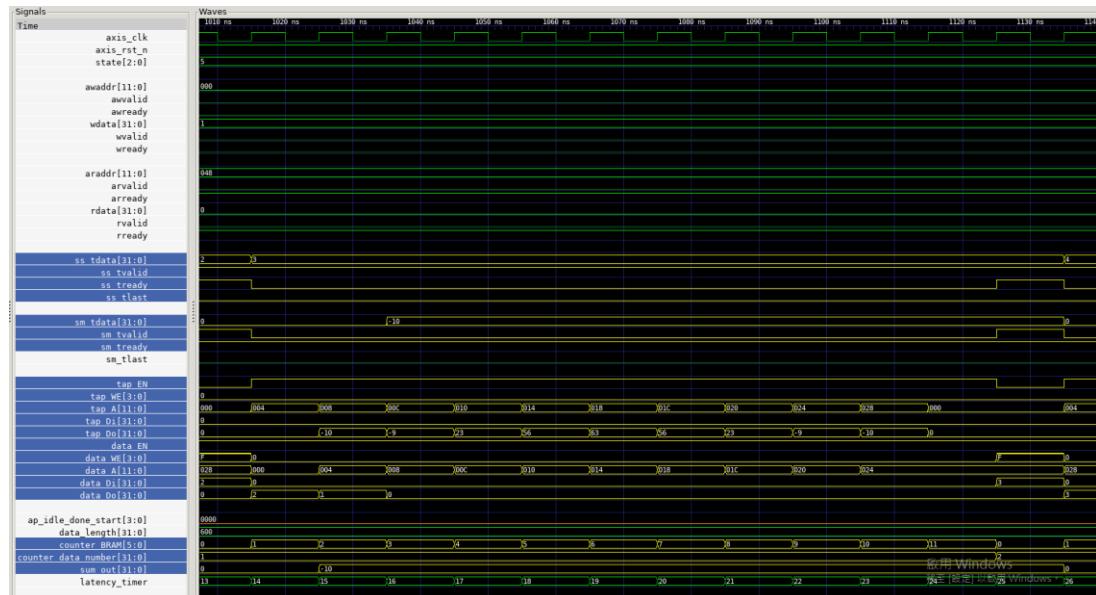
- C. 接下來不斷將 address pointer 指向下一個 address (tap_RAM、data_RAM 皆是)，按照「桃紅色箭頭 → 橘色箭頭 → 黃色箭頭 → 淺綠色箭頭 → 綠色箭頭 → 淺藍色箭頭 → 藍色箭頭 → 純藍色箭頭 → 紫色箭頭 → 黑色箭頭」的順序依序運算並更新 sm_tdata 的值，即可得到下圖：



即有第二筆 output data $y[1]$ 的正確算式。此時 counter_BRAM 為 11。

- D. 當 counter_BRAM=11 時，表示已經迭代 11 次運算，因此此時的 y 值已算完可以輸出了。接著只需要將 sm_tvalid 改為 1 即可與外界進行 AXI-Stream 的 handshake 輸出此值。並將 counter_data_number 的值增加 1，表示已完成一筆數據。

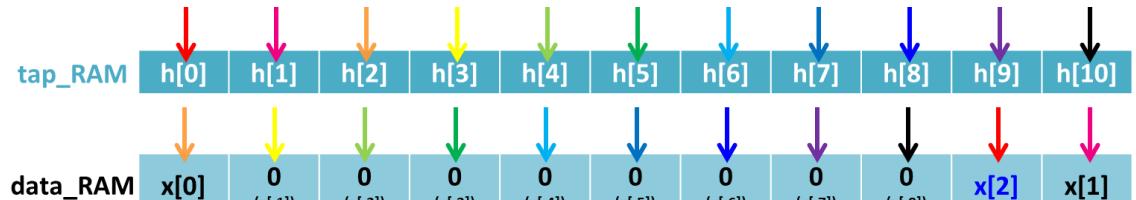
E. 第二筆數據的相對應波形圖如下：



(4) 計算第三筆 output data $y[2]$ ：

與前面類似，先 input $x[2]$ 進來，並將其儲存在 data_RAM 的 address 9 位置（前前圖的黑色箭頭處）→接著將 tap_RAM 的 address pointer 指向下一個 address，但 data_RAM 的 address pointer 維持在原位（前

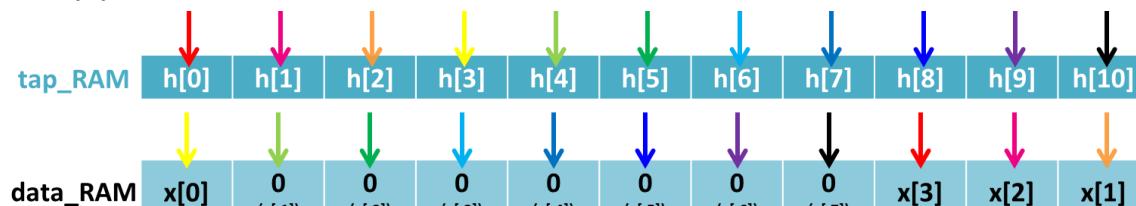
前圖中的黑色箭頭處) →接著才是不斷將 address pointer 指向下一個 address (tap_RAM 、 data_RAM 皆是) · 按照「 紅色箭頭 → 桃紅色箭頭 → 橘色箭頭 → 黃色箭頭 → 淺綠色箭頭 → 綠色箭頭 → 淺藍色箭頭 → 藍色箭頭 → 純藍色箭頭 → 紫色箭頭 → 黑色箭頭 」的順序依序運算並更新 sm_tdata 的值 · 即可得到下圖 :



$$y[2] = h[0]x[2] + h[1]x[1] + h[2]x[0] + h[3]x[-1] + h[4]x[-2] + h[5]x[-3] + h[6]x[-4] + h[7]x[-5] + h[8]x[-6] + h[9]x[-7] + h[10]x[-8]$$

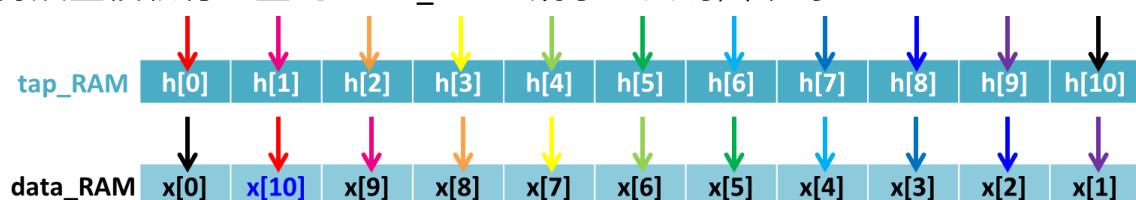
(5) 計算第四筆 output data $y[3]$:

與第(4)點同理 · 可得到下圖 :



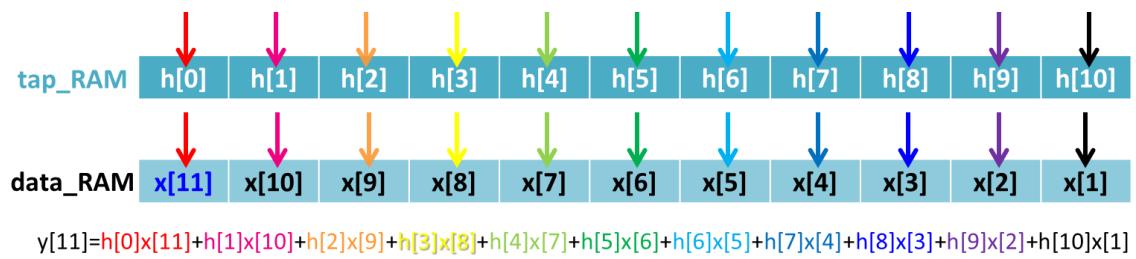
$$y[3] = h[0]x[3] + h[1]x[2] + h[2]x[1] + h[3]x[0] + h[4]x[-1] + h[5]x[-2] + h[6]x[-3] + h[7]x[-4] + h[8]x[-5] + h[9]x[-6] + h[10]x[-7]$$

(6) 持續重複執行 · 直到 data_RAM 繞了一圈寫回來時 :

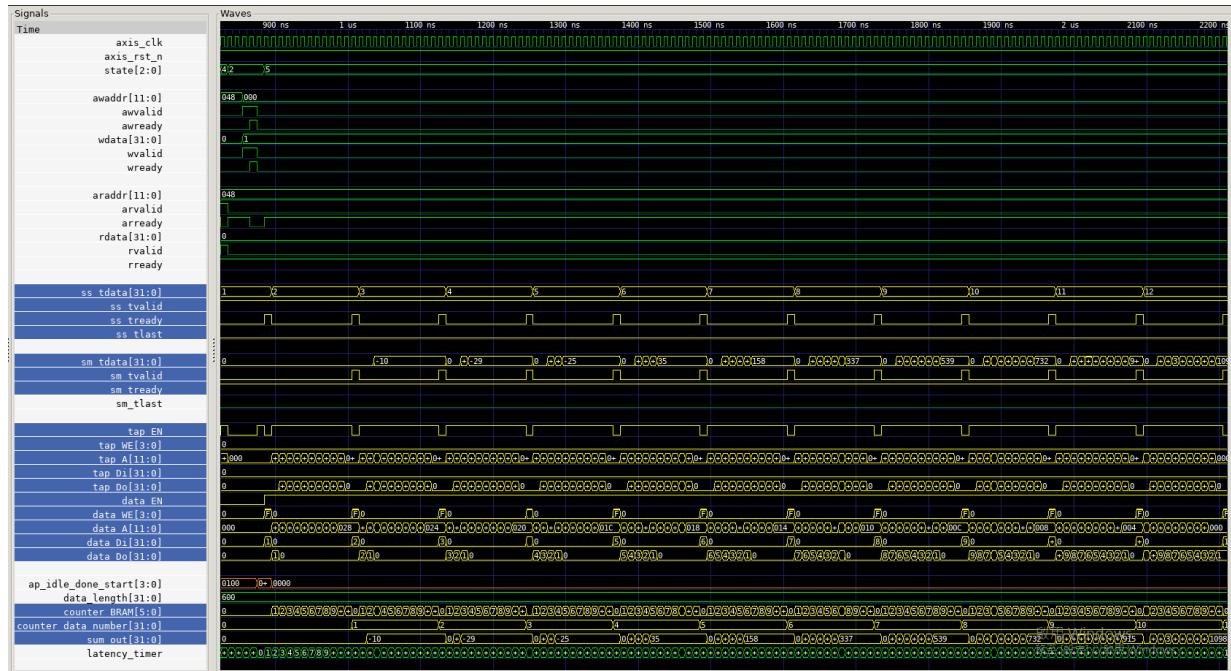


$$y[10] = h[0]x[10] + h[1]x[9] + h[2]x[8] + h[3]x[7] + h[4]x[6] + h[5]x[5] + h[6]x[4] + h[7]x[3] + h[8]x[2] + h[9]x[1] + h[10]x[0]$$

(7) 接著讀入 $x[11]$ 後同樣是按照上述方式 · 將 $x[0]$ 覆蓋掉 (因為之後不會需要用到 $x[0]$ 的值來運算了)



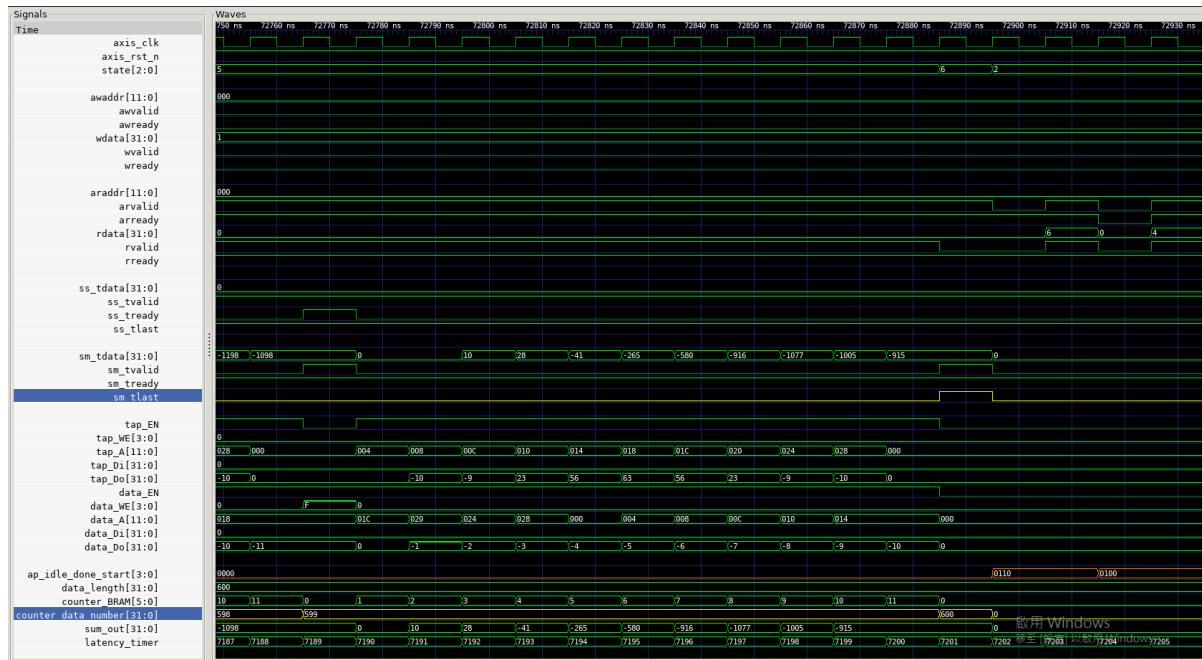
如此一來即可達成 FIR 的 function，並且每筆 data 只需要 12 個 cycle 即可完成。此部分的波形圖如下圖所示（由於持續好幾千個 cycle，故只放大一部份波形以方便觀察）：



7. How to give AXI-Stream Data_out?

在第 6. 點中有說明過，由於此時 $y[0]$ 已存在 sm_tdata port 上（因為在運算過程中 sm_tdata 為 idle，故剛好能直接拿來當作運算過程中的暫存器使用，且如此一來最後就可以直接由此 port 輸出），故此時只需要將 sm_tvalid 改為 1，並等待 sm_tready 為 1，即可與外界進行 AXI-Stream 的 handshake 輸出此值（但由於下個 data 也要用到 sm_tdata 來暫存運算中資料，故必須等這次 output data 成功傳輸後才可 reset 此值在給下筆 data 運算使用）。在前面的波形圖中已有顯示 sm channel 相關的波形。

若 counter_data_number 已 reach data_length，表示正在輸出最後一筆 data，故此時的 sm_tlast 訊號應拉升為 1，如下圖黃色部分波形所示：



8. How ap_done is generated?

ap_done 訊號在 axis_rst_n 做 reset 時會 reset 為 0。在 FIR working 時，每當運算完/輸出一筆 data 後，會將 counter_data_number 的值增加 1，表示已多完成一筆數據。若 counter_data_number 已 reach data_length，表示輸出筆數已達要求，此時可將 FSM 的 state 跳至 FIR_LAST_ONE，表示正在輸出最後一筆 data，此時仍在等待 sm channel 的 handshake。待 handshake 結束後，表示所有 data 傳輸完成，FIR 的任務也就到此告一段落。故按照 workbook 中關於 ap_done protocol 的說明：

ap_done is asserted when engine completes last data processing and data is transferred.

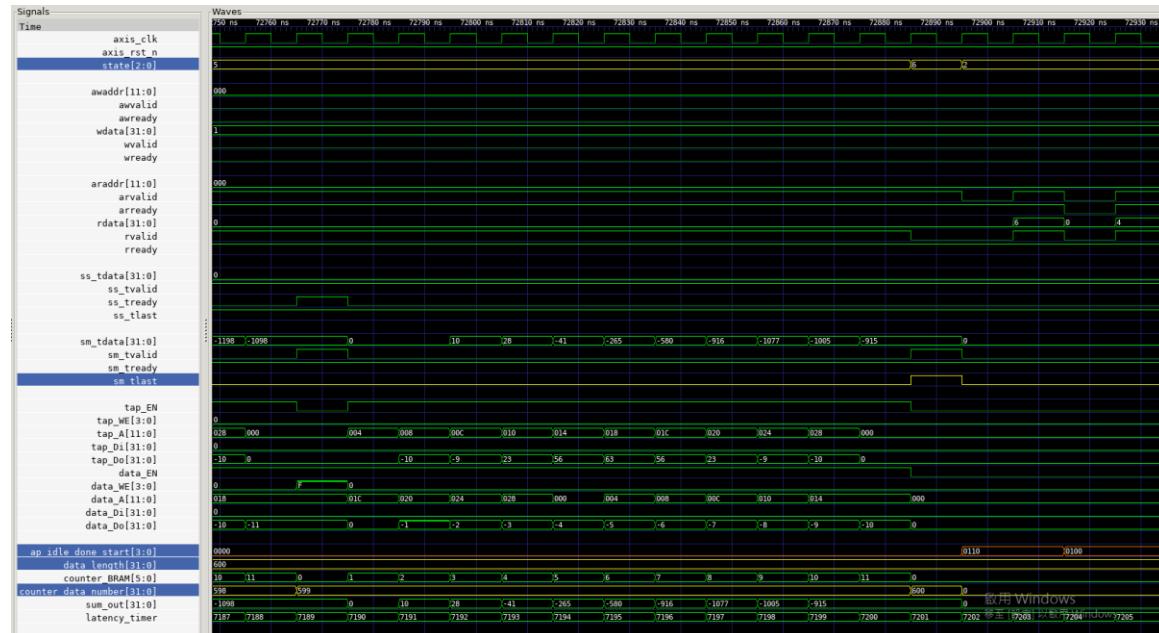
ap_done is reset when ap_done is read, i.e. address 0 is read.

由於此時已完成 last data processing 且 data 已傳輸完畢，故將 ap_done 拉為 HIGH (之後當 testbench 透過 AXI-Lite read channel 讀取到 0x00 的

address 時，表示 ap_done 已被讀取，故到時候需 reset 回 0)。而此時由 workbook 中關於 ap_idle protocol 的說明：

ap_idle is set to 1 when FIR engine processes the last data and last data is transferred.

故此時亦須將 ap_idle 設為 1。故波形圖為：



圖中黃色波形即為相關 signal · 而橘色波形則為 ap register 所存的值 · state number 為 6 表示在 FIR_LAST_ONE state 。

9. How many clock cycles have been used from ap_start to ap_done?

在 testbench 透過 AXI-Lite interface 去 program ap_start 為 1 時將計數器 reset 為 0 · 而 FIR 運作過程中每過一個 clock cycle 計數器就會 +1 。在 ap_done 被讀取到之後就會將計數器的值存在 FIR_final_latency 中 · 並且 report 到螢幕上 · 下圖為我在 testbench 中所加入的相關程式碼：

(每過一個 cycle 就加 1)

```
// Added by me
integer latency_timer;
initial begin
    latency_timer = 0;
    #5;
    forever begin
        #10 latency_timer = latency_timer+1;
    end
end
```

(ap_start program 為 1 後 timer 做 reset · 從 0 開始計數)

```
$display(" Tape programming done ...");
$display(" Start FIR");
////// 3. Program ap_start -> 1 (workbook p.19)
////// 2. Execution phase: (1)Program ap_start (workbook p.20)
@(posedge axis_clk) config_write(12'h00, 32'h0000_0001); // ap_start = 1
/////////////////// Added by me /////////////////////
////// 2. Execution phase: (2)Start latency timer (workbook p.20)
latency_timer=0;
```

(偵測到 ap_done 後 · 將此時 timer 的值 latch 起來)

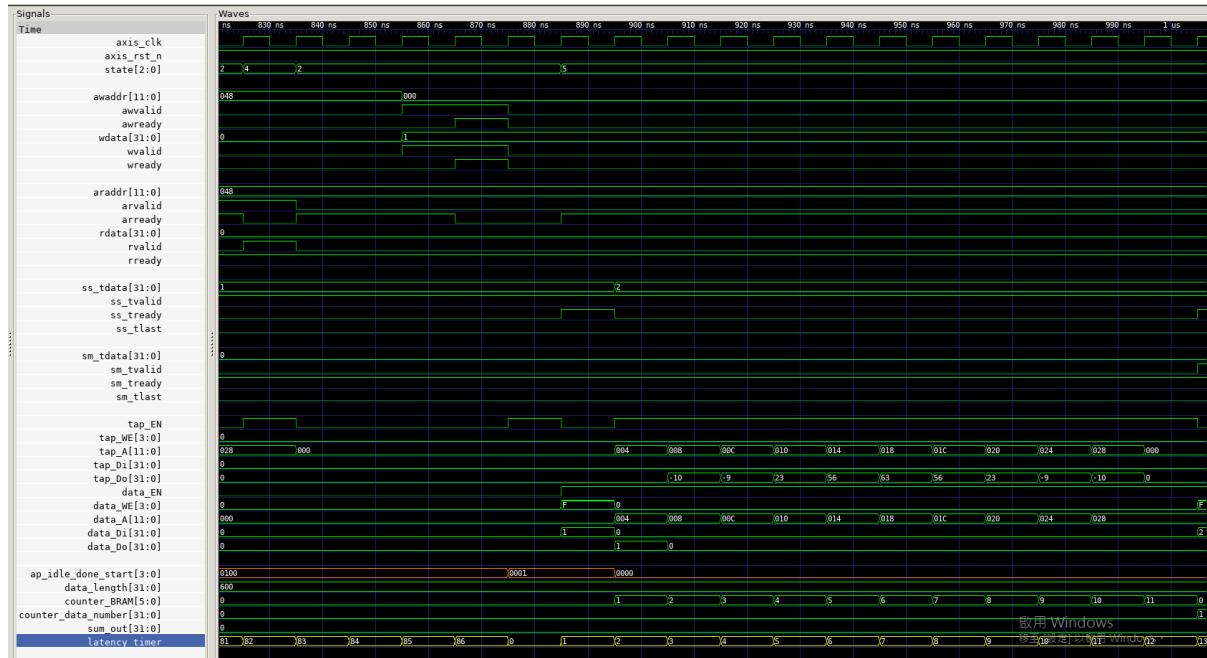
```
config_read_check(12'h00, 32'h02, 32'h0000_0002); // check ap_done = 1 (0x00 [bit 1])
/////////////////// Added by me ///////////////////
FIR_final_latency=latency_timer;
///////////////////
```

(最後輸出到螢幕上)

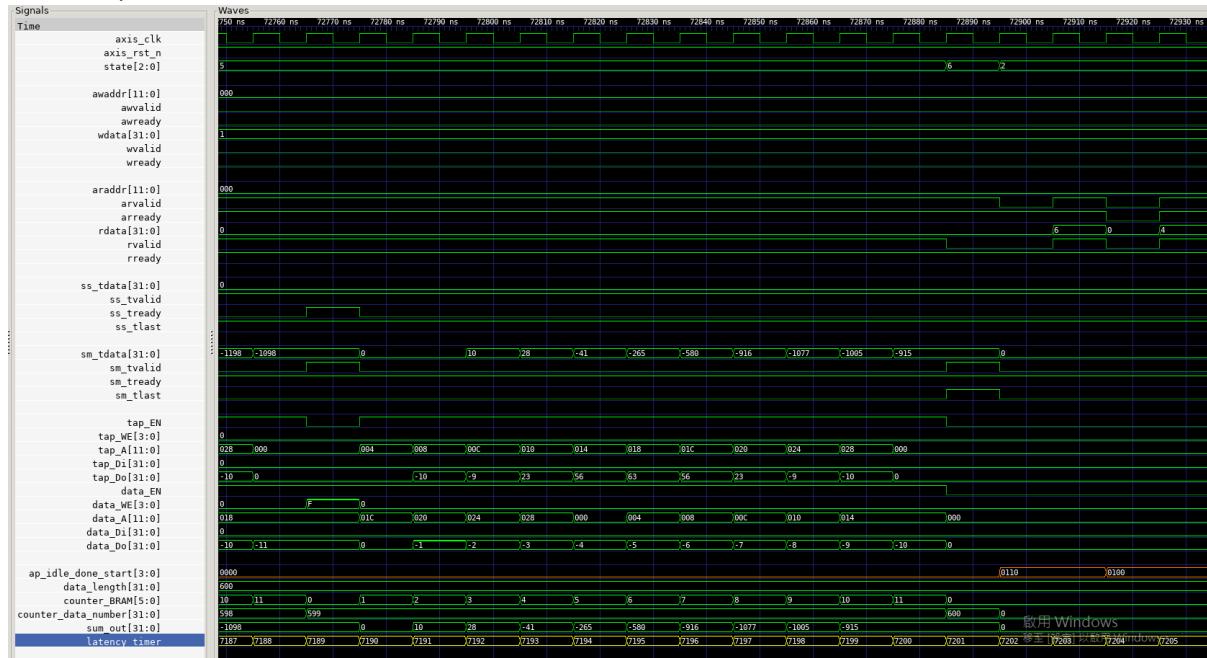
```
/////////////////// Added by me ///////////////////
////// 3. Checking phase: (1)Report latency (workbook p.20)
$display("FIR engine latency = %d clock cycles", FIR_final_latency);
///////////////////
```

此 timer 在波形圖中的行為 (每過 1 個 cycle 就加 1) :

(ap_start 變為 1 後 timer 就 reset 為 0)



(偵測到 ap_done 變為 1 時的 timer 值為 7204)



輸出到螢幕上的結果為：

```
-----Congratulations! Pass-----
FIR engine latency = 7204 clock cycles
```

故 FIR 花費了約 7204 個 clock cycle 執行 600 筆 output data 的運算！

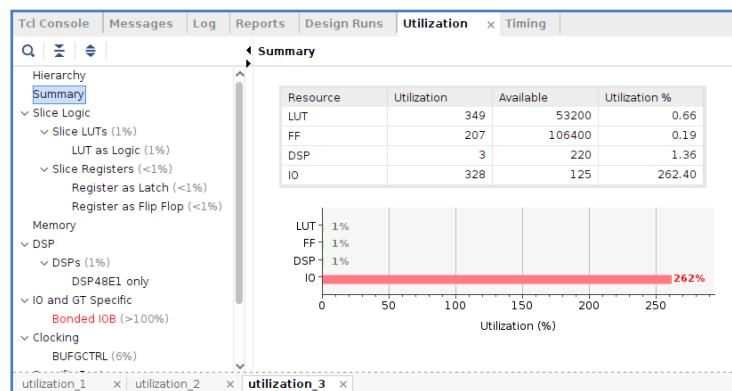
● Resource usage (including FF, LUT, BRAM):

使用 Vivado 匯入 design 後，執行 synthesis，可得到合成後的 utilization report，即 fir_utilization_synth.rpt。開啟此檔案可得到下圖的資源使用情形：

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	390	0	0	53200	0.73
LUT as Logic	390	0	0	53200	0.73
LUT as Memory	0	0	0	17400	0.00
Slice Registers	207	0	0	106400	0.19
Register as Flip Flop	174	0	0	106400	0.16
Register as Latch	33	0	0	106400	0.03
F7 Muxes	0	0	0	26600	0.00
F8 Muxes	0	0	0	13300	0.00

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	0	0	0	140	0.00
RAMB32/FIFO*	0	0	0	140	0.00
RAMB18	0	0	0	280	0.00

另外在 Vivado 中按下 " Report Utilization " 後會產生下圖（更為視覺化的整理）：



由上面三張圖可得知此 design 合成後所使用的

1. FF 個數為 207 個，而此 FPGA 板中共有 106400 個，故 utilization 為 0.19%
2. LUT 個數為 349 個，而此 FPGA 板中共有 53200 個，故 utilization 為 0.66%
3. BRAM 使用 0 個 → 合理，因為沒有將 external BRAM 拿去合成，故 utilization 為 0%

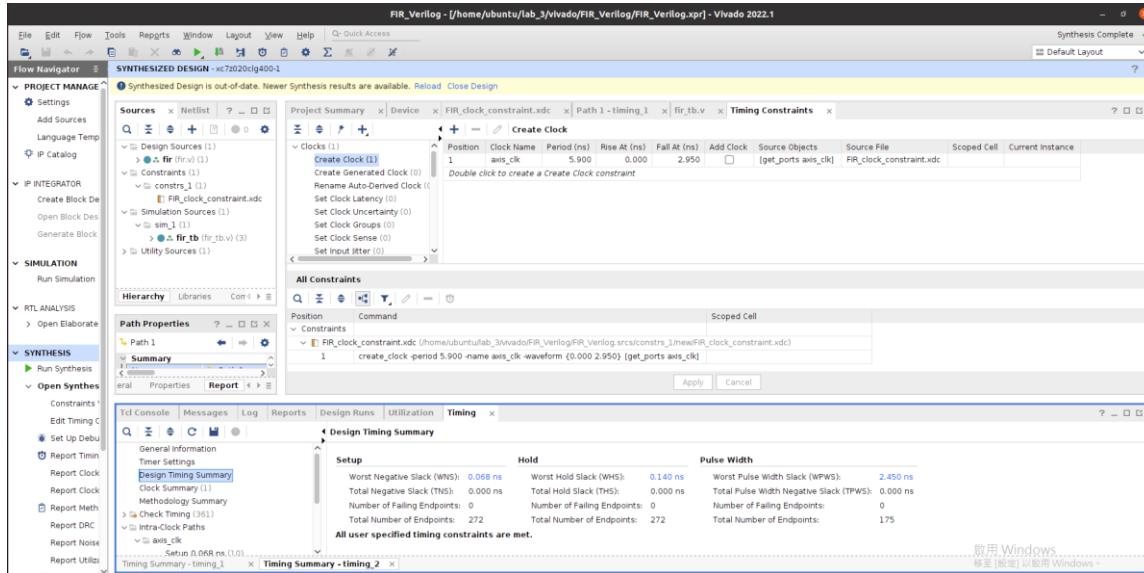
● Timing Report

使用 Vivado 匯入 design 後，執行 synthesis，並盡量設定最高速度（我一開始先使用 clock period=10ns 合成，發現 slack 為+4 點多，因此可再加快操作速度：此 design 所能達到的最快速度約為 clock period=5.9ns，換算下來 maximum frequency 約為 169.5MHz）。再按照作業區的 SYN_Workflow.pptx 的操作流程，可得到合成後的 timing report，即 timing_report_Max_frequency.txt，其中有一部份寫到關於 longest path：（由於文件較長，故分多次截圖）

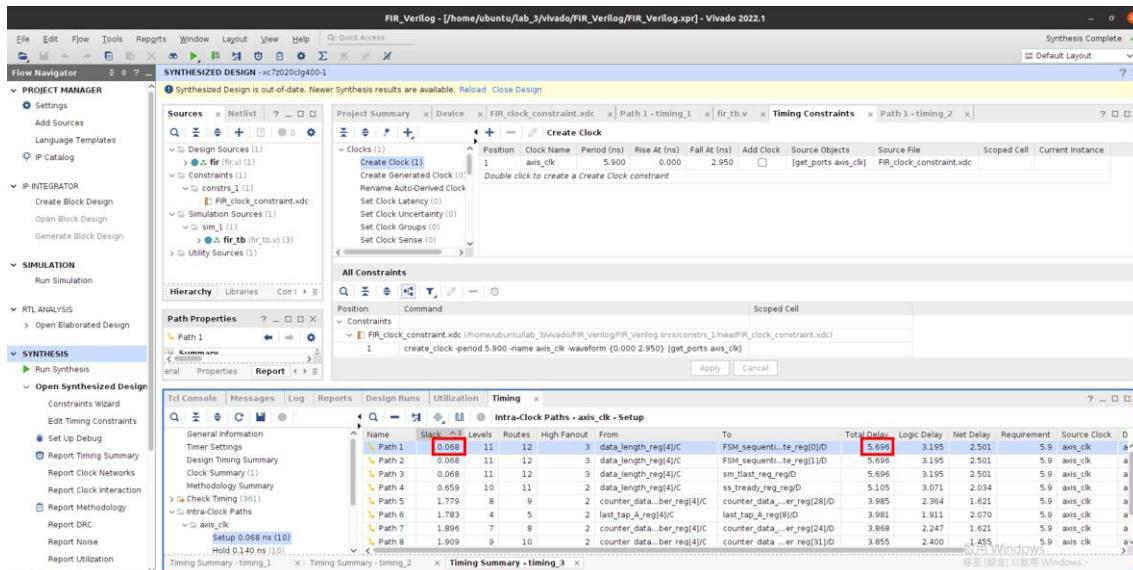
Max Delay Paths				
Location	Delay type	Incr(ns)	Path(ns)	Netlist Resource(s)
	(clock axis_clk rise edge)			
		0.000	0.000 r	
		0.000	0.000 r axis_clk (IN)	
net (fo=0)		0.000	0.000 r axis_clk	
			r axis_clk_IBUF_inst/I	
TBUF (Prop_ibuf_I_0)		0.972	0.972 r axis_clk_IBUF_inst/O	
net (fo=1, unplaced)		0.880	1.771 axis_clk_IBUF	
			r axis_clk_IBUF_BUFG_inst/I	
BUFG (Prop_bufg_I_0)		0.101	1.872 r axis_clk_IBUF_BUFG_inst/O	
net (fo=174, unplaced)		0.584	2.456 axis_clk_IBUF_BUFG	
			r data_length_reg[4]/C	
FDCE (Prop_fdcce_C_Q)		0.478	2.934 r data_length_reg[4]/Q	
net (fo=2, unplaced)		0.057	3.591 data_length[4]	
			r ss_tready_reg_reg_i_35/O[3]	
CARRY4 (Prop_carry4_DI[3].CO[3])		0.567	4.158 r ss_tready_reg_reg_i_35/CO[3]	
net (fo=1, unplaced)		0.009	4.167 ss_tready_reg_reg_i_35_n_0	
			r ss_tready_reg_reg_i_34/CI	
CARRY4 (Prop_carry4_CI_CO[3])		0.117	4.284 r ss_tready_reg_reg_i_34/CO[3]	
net (fo=1, unplaced)		0.000	4.284 ss_tready_reg_reg_i_34_n_0	
			r ss_tready_reg_reg_i_23/CI	
CARRY4 (Prop_carry4_CI_CO[3])		0.117	4.401 r ss_tready_reg_reg_i_23/CO[3]	
net (fo=1, unplaced)		0.000	4.401 ss_tready_reg_reg_i_23_n_0	
			r ss_tready_reg_reg_i_22/CI	
CARRY4 (Prop_carry4_CI_CO[3])		0.117	4.518 r ss_tready_reg_reg_i_22/CO[3]	
net (fo=1, unplaced)		0.000	4.518 ss_tready_reg_reg_i_22_n_0	
			r ss_tready_reg_reg_i_21/CI	
CARRY4 (Prop_carry4_CI_CO[3])		0.117	4.635 r ss_tready_reg_reg_i_21/CO[3]	
net (fo=1, unplaced)		0.000	4.635 ss_tready_reg_reg_i_21_n_0	
			r ss_tready_reg_reg_i_16/CI	
CARRY4 (Prop_carry4_CI_CO[3])		0.117	4.752 r ss_tready_reg_reg_i_16/CO[3]	
net (fo=1, unplaced)		0.000	4.752 ss_tready_reg_reg_i_16_n_0	
			r ss_tready_reg_reg_i_15/CI	
CARRY4 (Prop_carry4_CI_O[2])		0.256	5.008 r ss_tready_reg_reg_i_15/O[2]	
net (fo=1, unplaced)		0.005	5.913 next_state[27]	
			r ss_tready_reg_i_7/I0	
LUT6 (Prop_lut6_I0_O)		0.301	6.214 r ss_tready_reg_i_7/I0	
net (fo=1, unplaced)		0.000	6.214 ss_tready_reg_i_7_n_0	
			r ss_tready_reg_reg_i_2/S[1]	
CARRY4 (Prop_carry4_S[1].CO[2])		0.574	6.788 f ss_tready_reg_reg_i_2/CO[2]	
net (fo=2, unplaced)		0.463	7.251 ss_tready_reg_reg_i_2_n_1	
			f FSM_sequential_state[1].i_3/I0	
LUT6 (Prop_lut6_I0_O)		0.310	7.561 r FSM_sequential_state[1].i_3/I0	
net (fo=3, unplaced)		0.467	8.028 FSM_sequential_state[1].i_3_n_0	
			r FSM_sequential_state[0].i_1/I3	
LUT6 (Prop_lut6_I3_O)		0.124	8.152 r FSM_sequential_state[0].i_1/I0	
net (fo=1, unplaced)		0.000	8.152 FSM_sequential_state[0].i_1_n_0	
			r FSM_sequential_state[0]/D	

	(clock axis_clk rise edge)	5.900	5.900 r	axis_clk (IN)
net (fo=0)	0.000	5.900 r	axis_clk	
IBUF (Prop_ibuf_I_0)	0.838	6.738 r	axis_clk_IBUF_inst/0	
net (fo=1, unplaced)	0.768	7.498 r	axis_clk_IBUF	
BUFG (Prop_bufg_I_0)	0.091	7.589 r	axis_clk_IBUF_BUFG_inst/0	
net (fo=174, unplaced)	0.439	8.028 r	axis_clk_IBUF_BUFG	
FDCE			r	FSM_sequential_state_reg[0]/C
clock pessimism	0.184	8.211		
clock uncertainty	-0.035	8.176		
FDCE (Setup_fdc_C_0)	0.044	8.226		FSM_sequential_state_reg[0]
required time		8.226		
arrival time		-8.152		
slack	0.068			

另外，也可在 Vivado 中看到較為視覺化的 timing report：



其中也有 report 出 critical path (為下圖 path 1)：



此 critical path 中圖所經過的路徑也會顯示出：



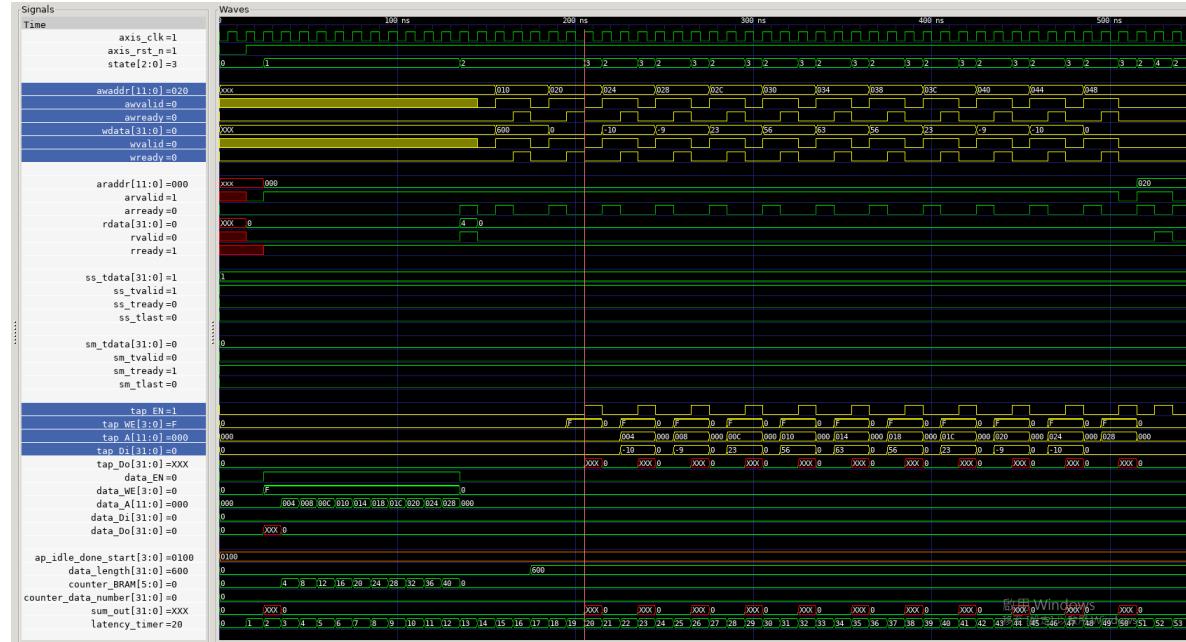
由前面幾張圖中的紅色方框可知：此 longest path 的 timing/delay 為 5.696ns · 而其 slack 為 0.068 · 為正的值。

● Simulation Waveform

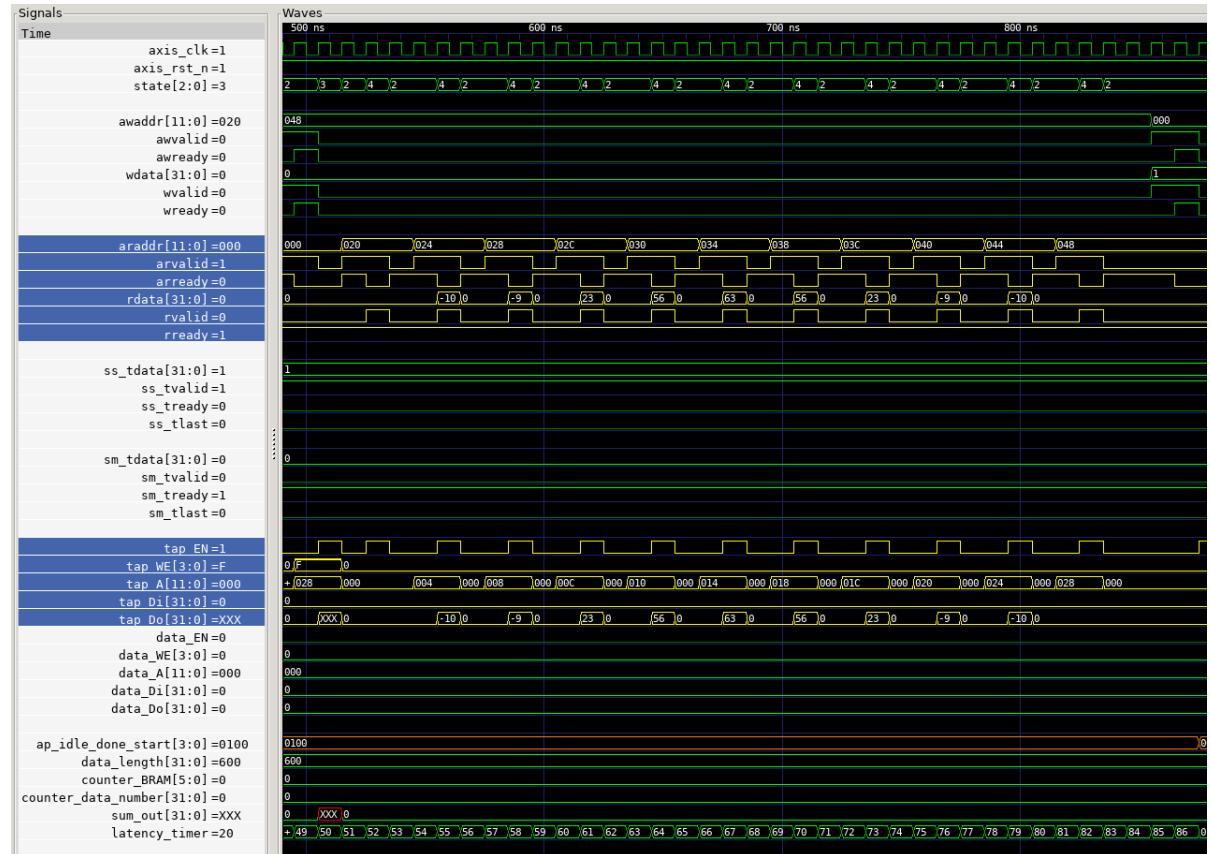
在前面介紹時已有放過波形圖以及做相關 operation 的說明了，在此將題目所需的相關波形與圖形列出來：

1. Coefficient program, and read back

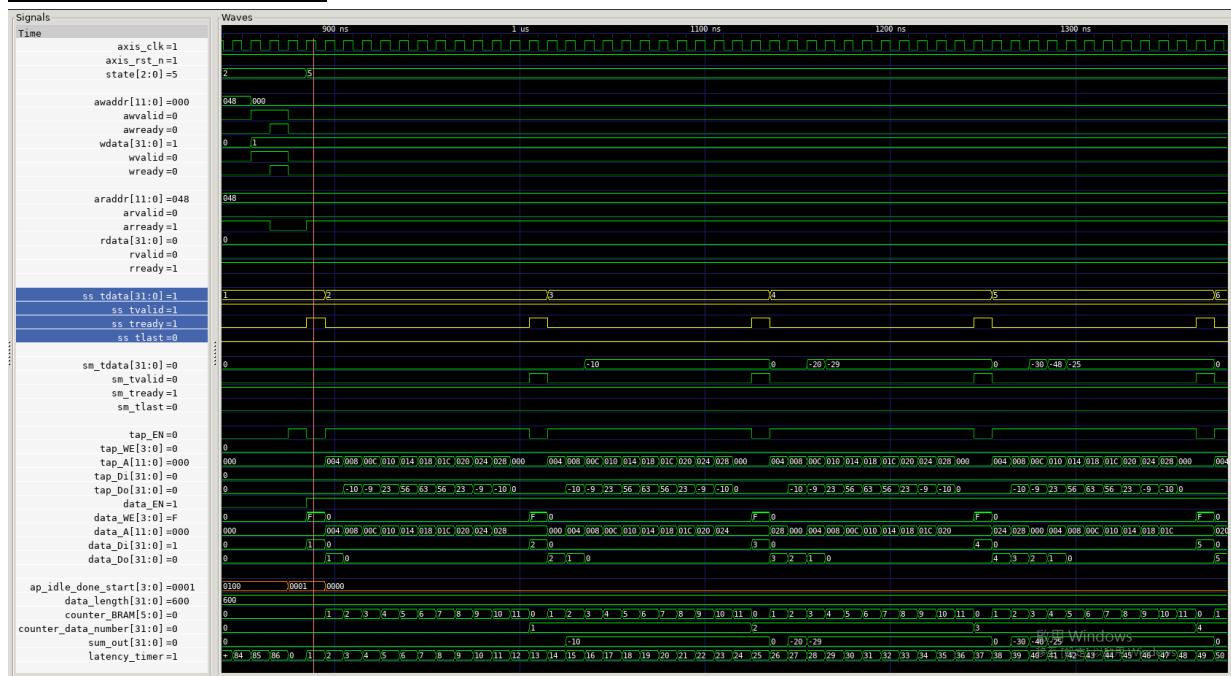
(coefficient program)



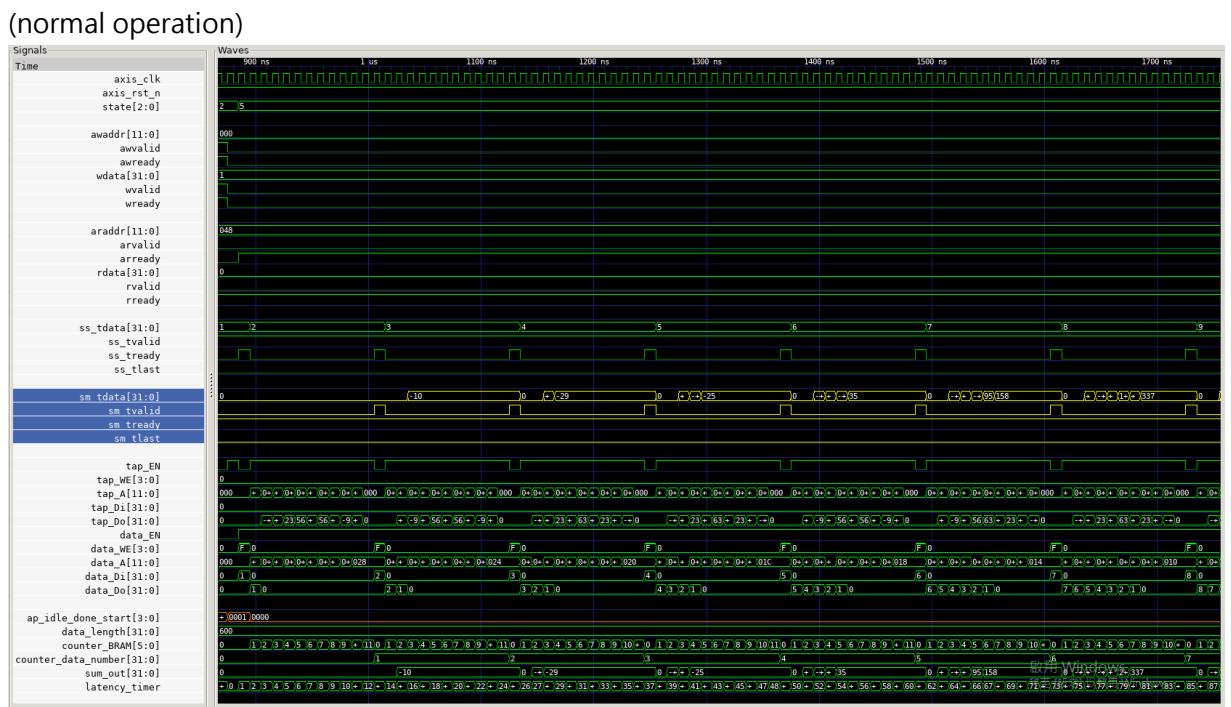
(coefficient read back)



2. Data-in stream-in



3. Data-out stream-out

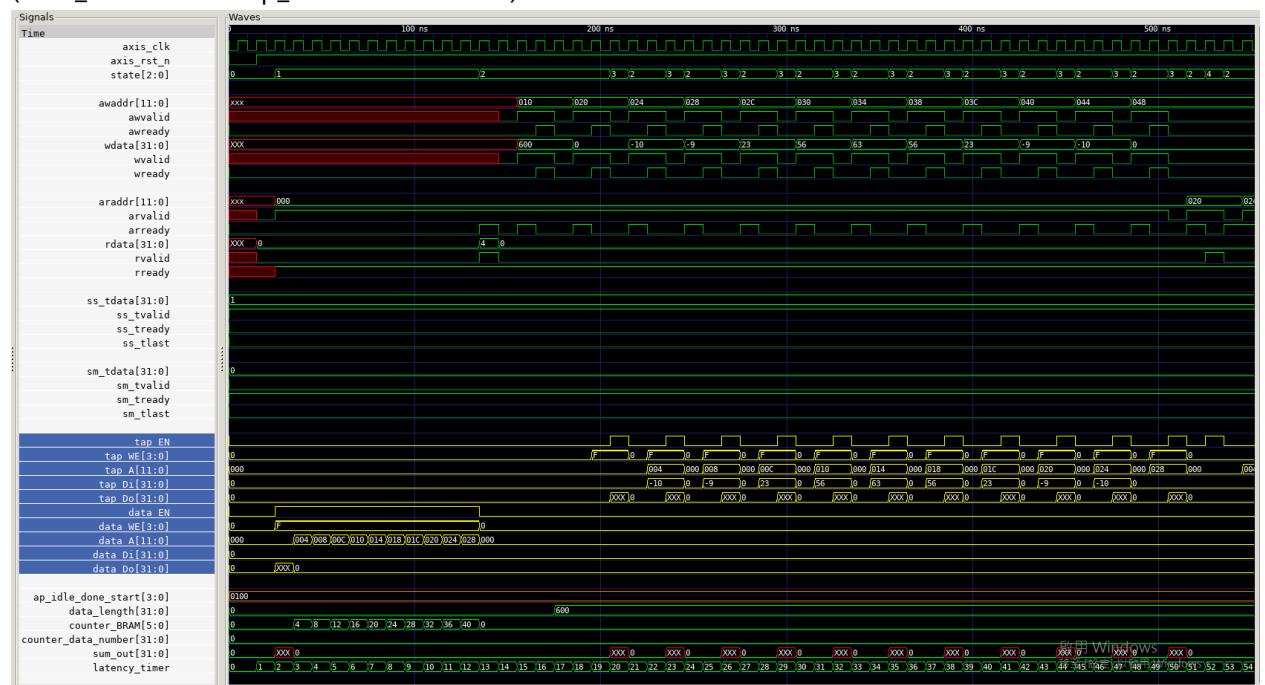


(last one output data)

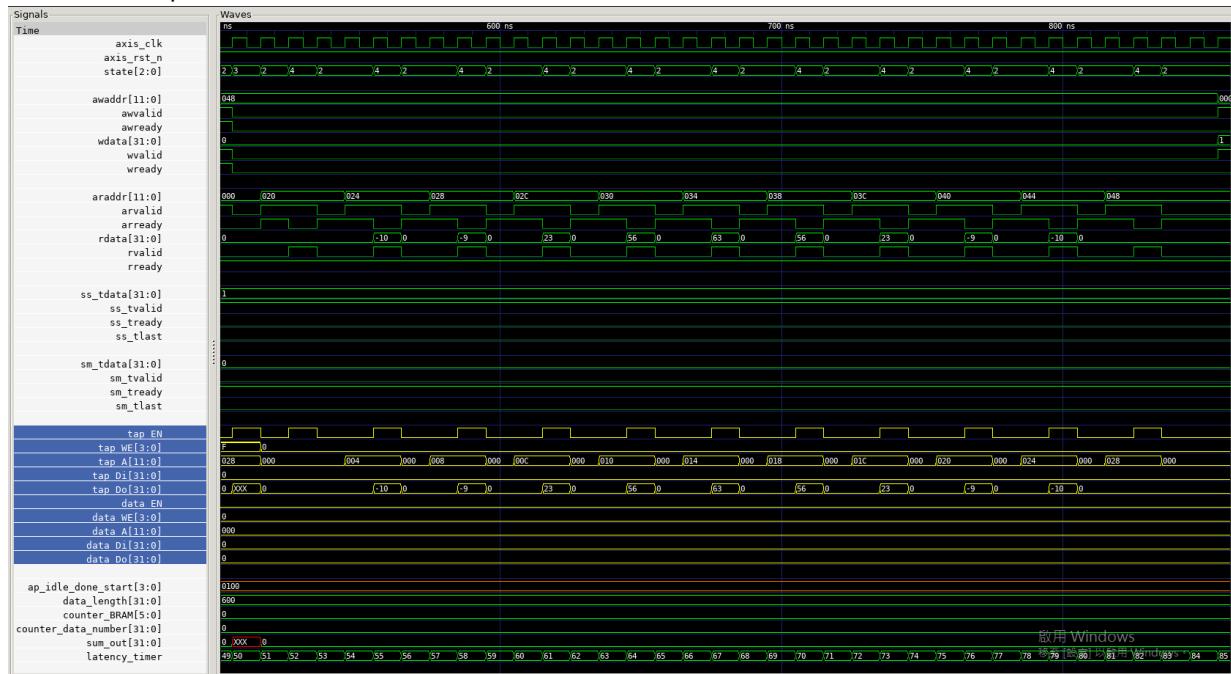


4. RAM access control

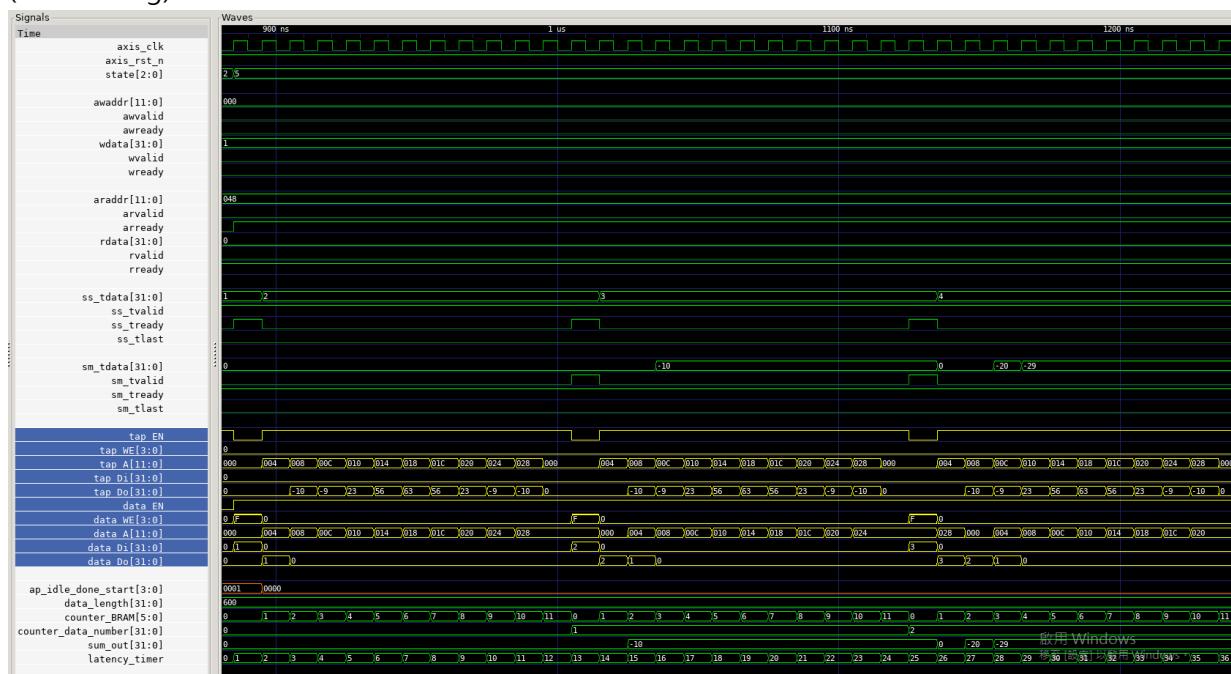
(data_RAM reset+tap_RAM initialization)



(read back tap_RAM)



(FIR working)

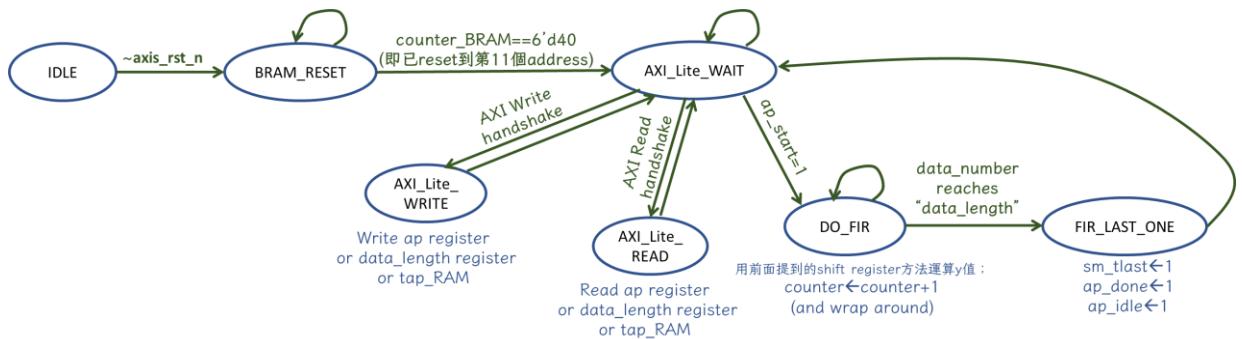


5. FSM

前面的波形圖中的 state[2:0]已有 indicate 出各 operation 下的 state 為何，其 number 與報告中所稱的名字對應關係如下表：

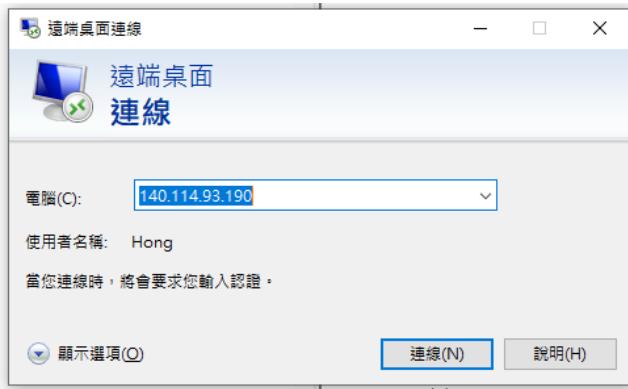
state number	state name
0	IDLE
1	BRAM_RESET
2	AXI_Lite_WAIT
3	AXI_Lite_WRITE
4	AXI_Lite_READ
5	DO_FIR
6	FIR_LAST_ONE

其 FSM 為：

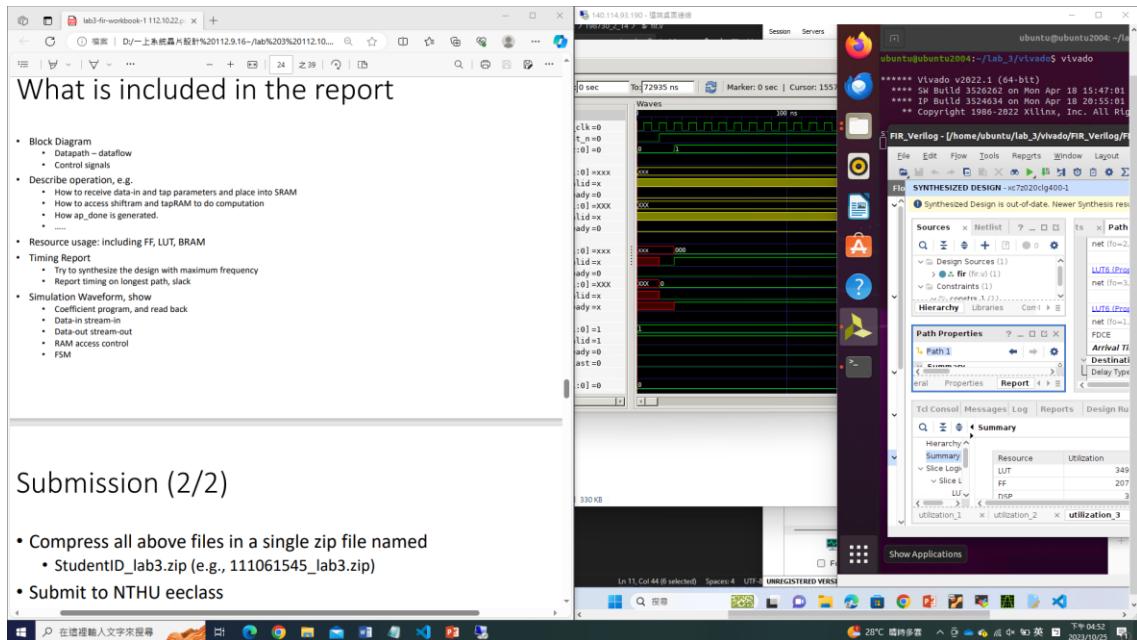


● 說明受到清大這波停電的影響情形

在 lab0 安裝 Ubuntu VM 時，我發現壓縮檔就有 100 多 GB，解壓縮後的檔案甚至需要 200 多 GB 的硬碟空間才放得下，然而我的筆電（作業系統為 Windows 10）當時能用的空間只有 200 多 GB，若安裝下去會剩沒多少空間，而沒辦法放其他資料，因此我將 VM 及相關軟體安裝在學校實驗室的電腦中（作業系統為 Windows 11），並且透過 Windows 內建軟體「遠端桌面連線」來遠端連線到實驗室的電腦，如下圖所示。



實際操作時的情形：



(Windows 10 的筆電(左半)遠端連線到 Windows 11 的實驗室 PC (右半) , 並且在 PC 中開啟 VM 及 MobaXterm 等軟體)

每當清大跳電(無論有無事先預告) , 實驗室的 PC 就會關機 , 而無法遠端連線過去 , 除了來不及儲存的 current state 無法復原外 , 在復電時也無法遠端開機 , 只能等到隔天到實驗室去手動開機。反覆多次的跳電使得每次停電的當天無法繼續寫這份作業 , 需要等到復電後去手動開機才能繼續 , 而導致進度落後。