

國立清華大學

系統晶片設計

SOC Design



國立清華大學
NATIONAL TSING HUA UNIVERSITY

Lab 4-2

組別：第 12 組

學號:111063548、111061624、112501538

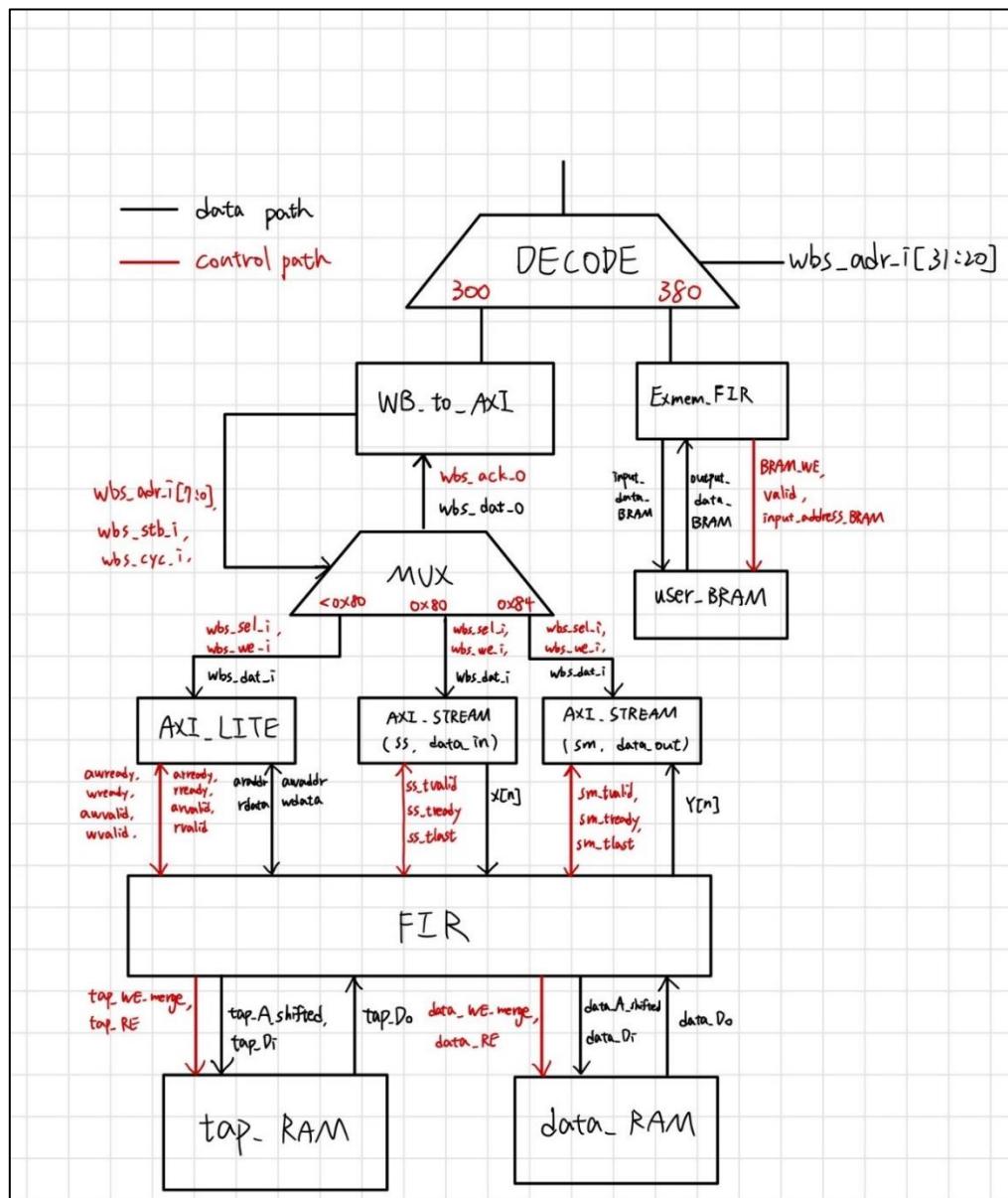
姓名:蕭方凱、尤弘瑋、葉承泓

指導老師:賴瑾教授

目錄

1.	Design block diagram – datapath, control-path.....	3
2.	The interface protocol between firmware, user project and testbench.....	4
3.	Waveform and analysis of the hardware/software behavior	6
4.	What is the FIR engine theoretical throughput, i.e. data rate? Actually measured throughput?	15
5.	What is latency for firmware to feed data?	19
6.	What techniques used to improve the throughput?.....	22
6-1.	Does bram12 give better performance, in what way?.....	24
6-2.	Can you suggest other method to improve the performance?.....	25
7.	Any other insights	26
7-1.	Synthesis report.....	26
(1)	User_proj_example_utilization_synth.rpt.....	26
(2)	In Vivado Utilization Summary	28
(3)	User_proj_example.vds	28
(4)	Timing_report_3_67ns.txt.....	30
7-2.	Latency Calculating in Testbench	32

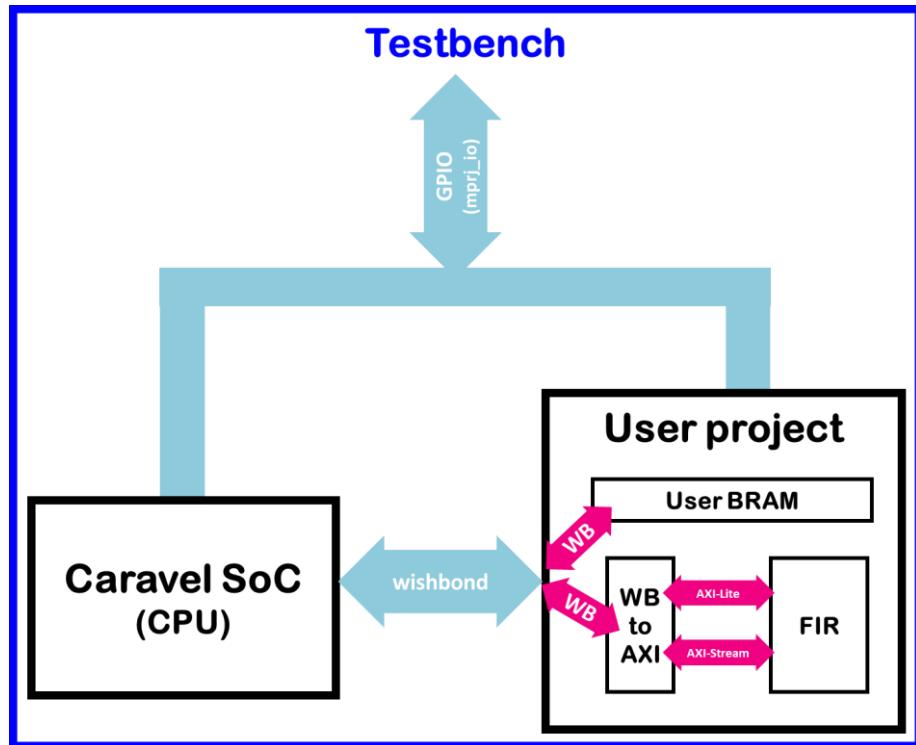
1. Design block diagram – datapath, control-path



2. The interface protocol between firmware, user project and testbench

在實作中，firmware code 會先透過 compiler 轉譯成 machine code（放在 counter_la_fir.hex 檔中），並在 testbench 中去讀這個檔案的資訊，再將這些 instruction 的內容存到相對應的 address 中(使用 global 的 configuration address map 來找到位址，以避免發生相同 local address 而 confusing 的情形)，例如存至 spiflash 或 user BRAM (在 lab4-1 與 lab4-2 皆有部分 code 放在這個 memory section，如此一來 access time 可以大幅縮短至 10 個 cycle 左右)中。

如下圖所示，在 Caravel SoC 中的 CPU (firmware) 與 user project 之間主要是透過 wishbone (WB)的 interface 做 communication，而 user project 內部才會再依據 hardware 所需的 protocol 自行轉換為 AXI-Lite 與 AIX-Stream，有如作為 lab3 中 FIR testbench 的功能，提供 tap parameter (AXI-Lite)、data input (AXI-Stream) 及接收 data output (AXI-Stream)。CPU 以及 user project 與外界（整個 lab4-2 的 testbench）之間則是使用 general purpose I/O(GPIO) 的 interface (pin 腳叫作 mprj_io[37:0]) 來互換資訊（在 lab4-2 中主要是由 CPU 來使用這個 interface 將 status 資訊（例如 start、end）和計算完的結果送出給 testbench，而 user project 只有用到 WB 與 CPU 做溝通）。



在 CPU 執行 firmware code 時，會先透過 wishbone (WB) 的 interface 至 address=0x3800 開頭的區段（位於 user project 中的 BRAM 裡）去拿取相對應的 instruction，接著執行 instruction，若此 instruction 有 read word、write byte 等 memory access 指令，例如在 C code 中使用

```
WB_address=(int*)(0x30000004);  
WB_return_data = *(WB_address);
```

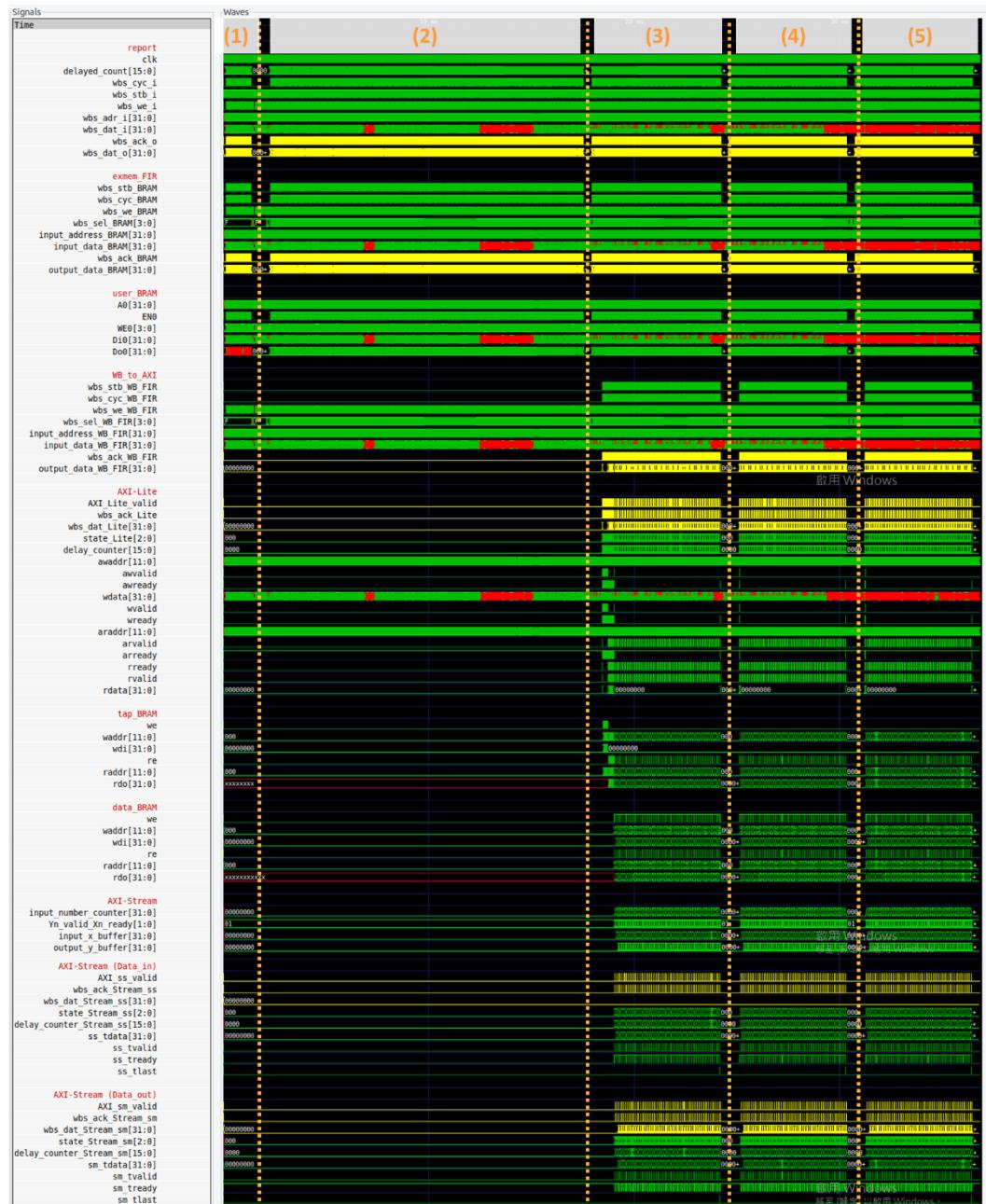
來讀取 global memory address 中 0x30000004 位址的 integer，此時 CPU 就會在 WB 中發起 request，並且在 WB 的 address 上寫明 0x30000004 這個位址，當對應到此位址的 slave 發現是在叫自己時，就要做出回應，例如在此例子中，address 為 0x30000004，是對應到 user project 中的 WB_to_AXI 底層的 memory (包含其中的 register 以及底下的 tap_RAM 等)，在處理完 data 後就要 return ACK 並(若有需要)將 WB data out 傳輸回 master (CPU)。若 CPU 的 instruction 有關於 mprj_io 的讀寫時，例如

```
reg_mprj_datal = 0x00A50000; // start mark on mprj[23:16]
```

其中 reg_mprj_datal 即對應到 mprj 的 wire，此時就會用到 GPIO 的 interface 與 testbench 做溝通。而 user project 與 testbench 之間的 GPIO interface (user_proj_example.counter.v 中的 io_in、io_out、io_oeb) 在這次 lab 中並沒有實作到。

3. Waveform and analysis of the hardware/software behavior

如下圖所示，我們將實作過程中的主要訊號全部顯示出來，並且 time scale 為整個 simulation 的長度：



可發現共可將整個 simulation 過程分為 5 個區段：

- (1) 區段 1：執行 counter_la_fir.c 中的前置作業，這些 instruction 是被放在 100 為開頭的 memory 區段裡；也穿插 fir.c 的 initialization（寫在 initfir()

函式中) 的過程，此部分則是被放在 380 為開頭的 memory 區段裡，即 user BRAM 中。將波形部分放大後可得



可看到 wbs_adr_i 這個訊號的 address 位址的確為 100 及 380 為開頭，表示 CPU 要從這些地方送 request 拿取 instruction。觀察上圖可發現 100 為開頭的 response time 比 380 為開頭的還要多了許多倍，由此可證實 access spiflash (對應到 memory address 即為 100 開頭) 所需的時間遠比 access user_BRAM 所需的時間 (對應到 memory address 為 380 開頭) 還要長！

- (2) 區段 2：執行 software 的 FIR 運算 (使用 lab4-1 的 fir.c 中的實作內容—fir()函式)，會發現這個區段佔了整個 simulation 中最大 part，由此可見使用 software 來執行 FIR 的運算遠比使用 hardware 來得慢。將此部分波形放大如下：



觀察 wbs_adr_i 這個訊號的 address 位址的確皆為 380 為開頭，且依序執行，此即為 CPU 在執行 assembly/firmware code 的類似 program counter 的行為。

- (3) 執行第一 round 的 hardware FIR implementation：底下會做詳細說明，介紹 software 與 hardware 之間的溝通方式及合作模式
(4) 執行第二 round 的 hardware FIR implementation：以確保有先回復初始設定才開始進行新一波的運算

(5) 執行第三 round 的 hardware FIR implementation：以確保有先回復初始設定才開始進行新一波的運算

在執行 hardware FIR implementation 的過程中，FIR engine 的部分是修改（因為 lab4-2 中的 bram11.v 與 lab3 中的行為不同；另外也將 handshake 的條件設置得更加 general）自 lab 3 實作的 hardware。由於它與外界之間是使用 AXI-Lite 與兩個 AXI-Stream 為 interface 做資料傳遞，與 Caravel SoC 中 CPU 與 user project 之間的 WB interface 傳遞方式不同，因此需要 WB_to_AXI 的轉換器；此外，由於 user project 中本來已有放 user BRAM，所以需要有 decoder 來分流輸入與輸出訊號，並且是透過不同的 address section 來區分這兩個分支(user_BRAM v.s. FIR)。

在 software 中（主要是 fir.c 中）是負責給予測試 data 並收取 output data 來 check FIR 回傳資料的正確性，我們主要是依照 lab 3 中的 testbench 的測試流程，將其轉換為軟體程式碼，而將 data input 給 user project 的過程則是透過呼叫此函式來達成：

```
void __attribute__((section(".mprjram"))) WB_write(int* WB_address, int write_data){  
    *(WB_address)=write_data;  
}
```

其中需要給予兩個 arguments：要寫入值的 address 以及欲寫入之值。

而若要讀取某個 address 的值以便檢查正確性，會透過呼叫此函式來達成：

```
int __attribute__((section(".mprjram"))) WB_read(int* WB_address){  
    return *(WB_address);  
}
```

其中只需要給予一個 argument：要讀值的 address。

透過這些指標運算，CPU 即會自動使用 WB interface 送出 request 並放上指定的 address，即可與相對應 address 得 WB slave 溝通及做資料傳輸。

User project 的 WB interface 只有一個，但當接收到 380 開頭/300 開頭這兩種 request 時，需要分別拿出底層 BRAM/底層 FIR 所運算吐出的資料來送回 WB 中，因此 WB decoder 就透過 address 來做 MUX，透過 WB address 的開頭來選取相對應的 module 執行，如下圖所示：



其中 input 訊號被分流至兩側 (_BRAM 與 _WB_FIR)，可看到關鍵的 input 訊號 (stb 與 cyc) 會被 gating 來確保一次只有一邊會執行運算，如圖中橘色圈圈處只有 user_BRAM 在運算、圖中紅色圈圈處只有 FIR 在運算。若 address 為 380 開頭，則會是 user_BRAM 做運算，此時 exmem_FIR(被一起 implement 在 user_proj_example.counter.v 裡面) 會 access user_BRAM，拿到值再傳給 decoder 輸出至 WB interface；若 address 為 300 開頭，則會是 FIR 做運算，此時這些 data 會傳給下層的 WB_to_AXI 做 interface protocol 轉換再傳給更下的 FIR，拿到 output 值再一直往上層傳至 decoder 輸出至 WB interface。最後的 ACK 與 WB 的 data_out 也都透過 address 來 MUX 決定何方為主要輸出者。

在 WB_to_AXI 中，由於向上為 WB interface，只有 1 個 channel，但向下有 1 組 AXI-Lite interface + 2 組 AXI-Stream interface，因此需要再透過更細緻的 address 來分流。我們透過題目中的 configuration address map 發現，雖然這些 address 都是定義在 300 開頭的 memory 區塊中，但可透過 LSB 的 byte 來分辨出來，像是 0x00~0x7F 就是屬於 AXI-Lite 的 interface、0x80 是屬於 AXI-Stream (ss, Data_in) 的 interface、0x84 是屬於 AXI-Stream (sm, Data_out) 的 interface。因此有如上面 decoder 的作法，我們透過 address 即可得知哪個 interface 要運作，其他的在此時就不運作（可將 valid 設為 0 即可防止 handshake 發生），實作上我們使用

```

always @* begin
    if(wbs_cyc_i && wbs_stb_i && (wbs_adr_i[7:0] <= 8'h7F)) begin
        AXI_Lite_valid=1;
    end
    else begin
        AXI_Lite_valid=0;
    end
end

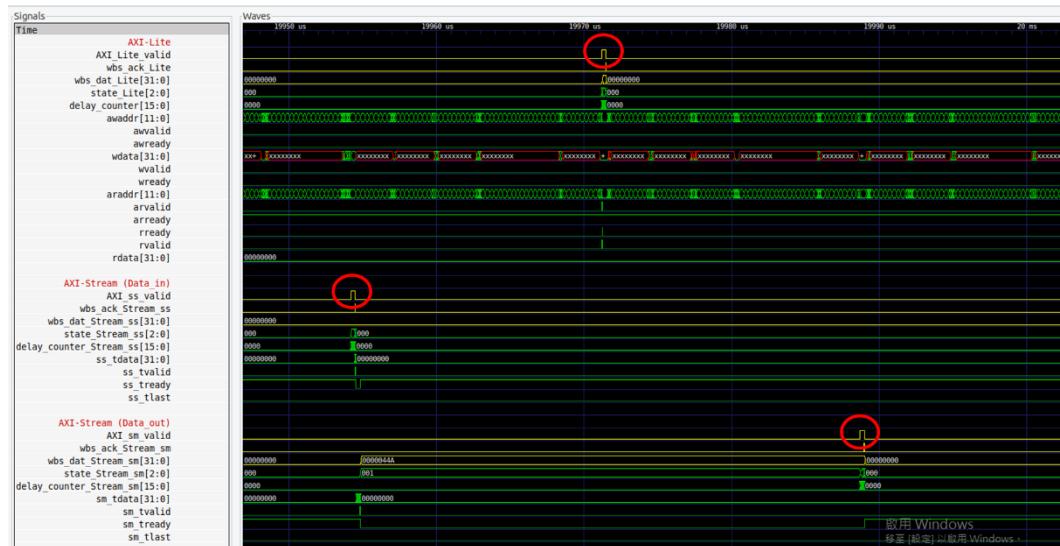
```

```

if(wbs_cyc_i && wbs_stb_i && (wbs_addr_i[7:0] == 8'h80)) begin
    AXI_ss_valid=1;
end
else begin
    AXI_ss_valid=0;
end
if(wbs_cyc_i && wbs_stb_i && (wbs_addr_i[7:0] == 8'h84)) begin
    AXI_sm_valid=1;
end
else begin
    AXI_sm_valid=0;
end

```

來達成上述目標。由於同一時間 address 只會來一個，因此只會有一個 interface 是 valid 的，它們之間為 mutually exclusive 的關係，如下圖所示：

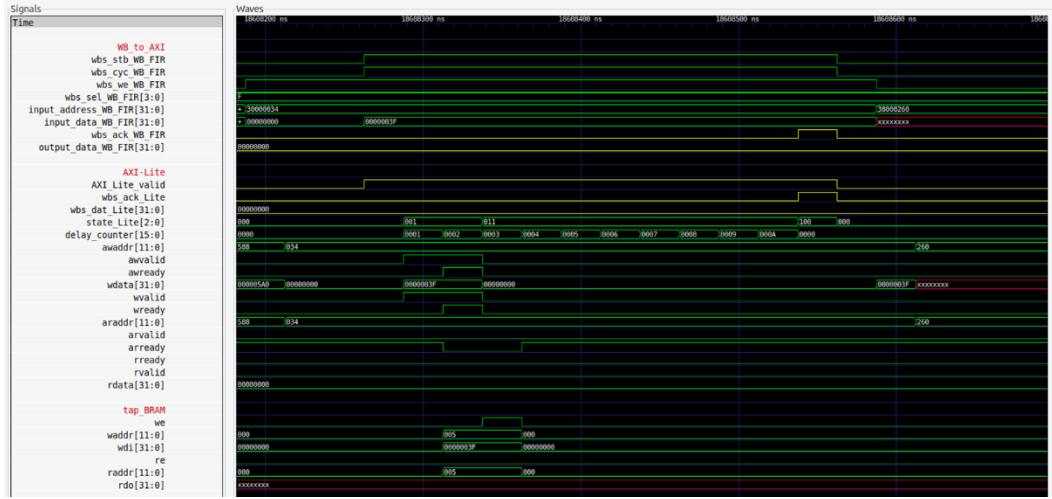


由此圖可知各 interface 的 valid 不會同時發生，這是合理的，因為 WB 一次只會 access 一個 global memory，因此理應最多只會碰到其中一個 channel。

● WB to AXI-Lite write :

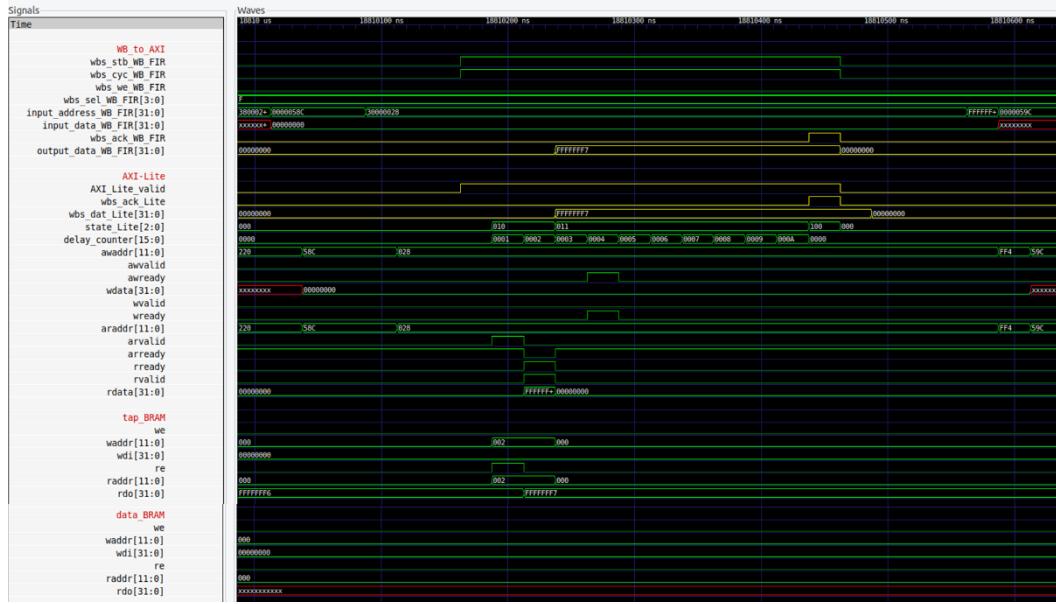
當 WB 的 address 開頭為 300 (表示要存取 user project 中關於 FIR 的此區塊)，且 address 結尾為 0x00~0x7F 時，表示要透過 AXI-Lite 讀寫資訊，此時 WB_to_AXI 轉換器就會將 AXI_Lite_valid 拉為 1 (其他 valid 降為 0)，並且 address 只取 LSB 的 3 bytes 傳到 AXI-Lite 的 address channel 上，由於 WB_WE 的值為 1，表示要 write，故 address 放到 write address channel，並將 WB 的 Data_in 的值連線給 wdata，接著宣告 awvalid 與 wvalid，即可等待 FIR 的 awready 與 wready，完成 handshake 後，WB_to_AXI 的 module 再將

ACK 拉為 1，上層就會將此資訊(在 delay 滿 10 個 cycle 後(利用 delay_counter 來計數)) forward 給 WB interface，讓 CPU 得知已完成 write 的動作。過程波形圖如下圖所示：



● WB to AXI-Lite read :

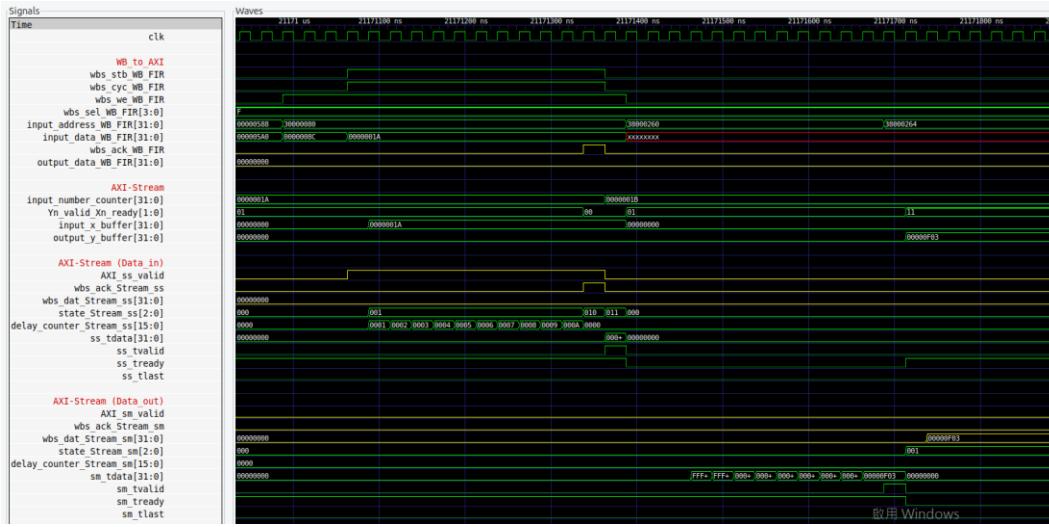
當 WB 的 address 開頭為 300 (表示要存取 user project 中關於 FIR 的此區塊)，且 address 結尾為 0x00~0x7F 時，表示要透過 AXI-Lite 讀寫資訊，此時 WB_to_AXI 轉換器就會將 AXI_Lite_valid 拉為 1 (其他 valid 降為 0)，並且 address 只取 LSB 的 3 bytes 傳到 AXI-Lite 的 address channel 上，由於 WB_WE 的值為 0，表示要 read，故 address 放到 read address channel，接著宣告 arvalid，即可等待 FIR 的 already，完成 handshake 後，等待 FIR 傳送回 rdata，完成 rdata channel 的 handshake 後，WB_to_AXI 的 module 再將 ACK 拉為 1，並將 rdata (若為讀取 address 0x00 的位址，則會再加(merge)上 Xn_ready 與 Yn_valid 這兩個 bit 的資訊(由於 FIR 內部並無實作到 Xn_ready 與 Yn_valid 這兩個 bit，且這兩個資訊與 input buffer/output buffer 有關，放在 WB_to_AXI module 中去使用它較為合理，因此在 WB_to_AXI module 中加上 2-bit 的 register 來儲存此資訊)) 接線至 WB Data_out，上層就會將此資訊 (在 delay 滿 10 個 cycle 後(利用 delay_counter 來計數)) forward 給 WB interface，讓 CPU 得知已完成 read 的動作。過程波形圖如下圖所示：



● WB to AXI-Stream Data_in (ss) :

(註：在 CPU 透過 WB 來讀取 0x80 這個位址之前，會先到 0x00 的位址確認 bit 4 的值為 1 (表示 input buffer is ready to accept the next input data x[n])，才會做此 request)

當 WB 的 address 開頭為 300 (表示要存取 user project 中關於 FIR 的此區塊)，且 address 結尾恰好為 0x80 時，表示要透過 AXI-Stream 寫入新的一筆 input data，由於我們有在此 module 中實作 input buffer 以增進效能，故這筆 data 會被寫入 input buffer 中，並在 10 個 cycle 的 delay 後將 ACK 拉為 1，上層就會將此資訊 forward 給 WB interface，讓 CPU 得知已完成 write 的動作。如此一來 input buffer 中就有新的值可以給 FIR 使用了，因此 AXI-Stream_ss 所對應的 FSM 會進入下一個 state，將 ss_tvalid 宣告為 1 (若 WB_to_AXI 內部計數發現為最後一筆，則會再將 ss_tlast 也拉到 1)，即可等待 FIR 的 ss_tready，完成 handshake 後，FSM 又會再回到第一個 state，並將 0x00 的 bit 4 改為 0，表示需要等著 CPU 再給一筆新的 input data。過程波形圖如下圖所示：

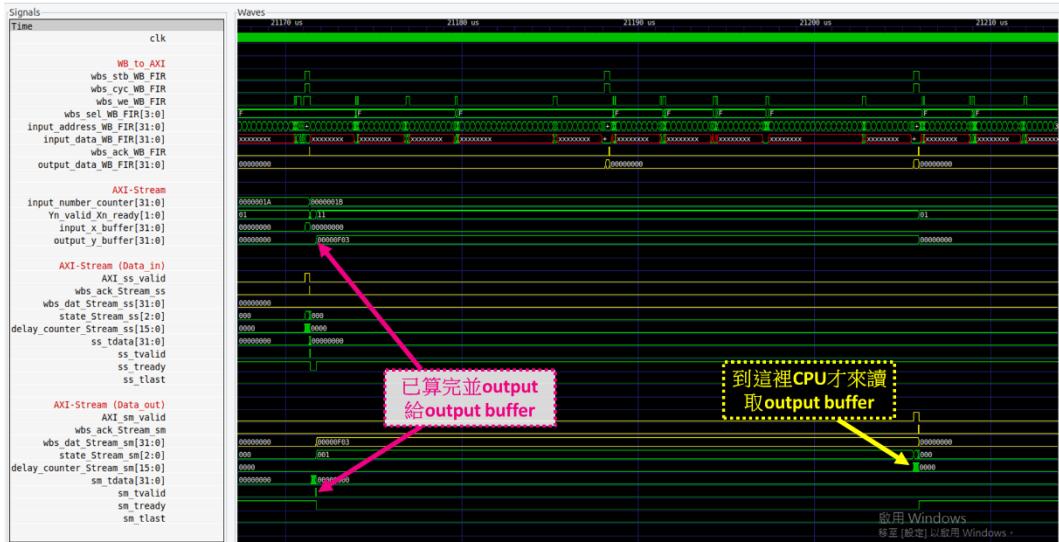


圖中可看出的確有等到 WB 的 ACK 發布後，才進入下一個 state 而將 ss_tvalid 拉起來，正好此時 FIR 正在 idle (因為 firmware 較慢而使得 FIR 早已做完等很久了) 而有 ss_tready 為 1，故可立即 handshake 傳遞 data 給 FIR。在圖中的左下角，可看到 FIR 已完成運算並將 output data 透過 AXI-Stream 傳給上層的 WB_to_AXI，且回到 idle 狀態將 ss_tready 拉起來等待新的 input data，但由於 CPU 還沒給 input buffer 新的值，在我們的 design 中使用 FSM 的好處之一為此情況下，因為使用 AXI-Stream 的 state 還沒到，所以會自動將這個 channel 關掉 (透過將 ss_tvalid 關為 0)。

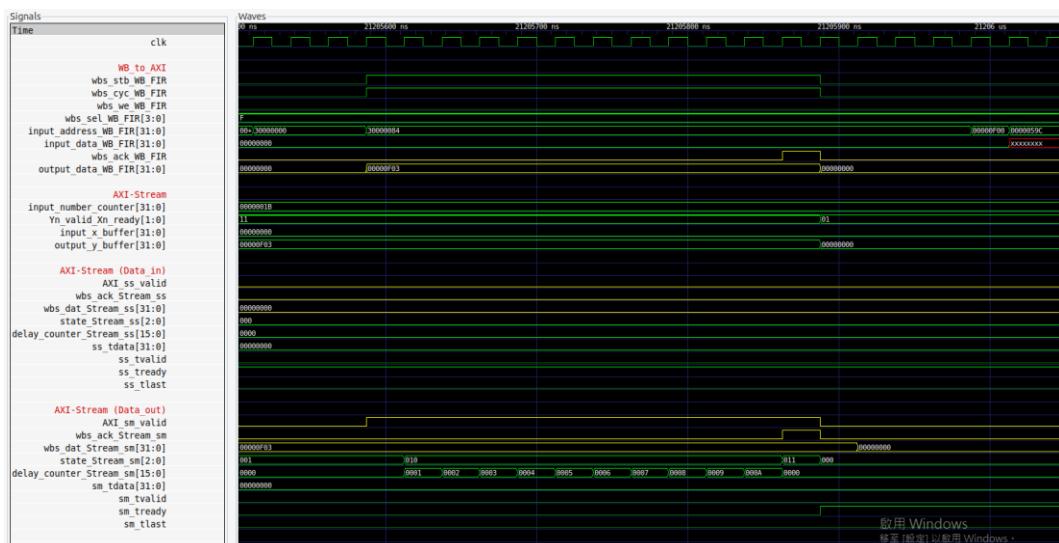
● WB to AXI-Stream Data_out (sm) :

(註：在 CPU 透過 WB 來讀取 0x84 這個位址之前，會先到 0x00 的位址確認 bit 5 的值為 1 (表示 output buffer is valid to give the next output data y[n])，才會做此 request)

在上圖中的左下角，可看到 FIR 已完成運算並將 output data 透過 AXI-Stream 傳給上層的 WB_to_AXI，WB_to_AXI 會在 handshake 時將此值記錄在 output buffer 中，並進入(專屬於 AXI-Stream_sm 所對應的 FSM 的)下個 state，並將 0x00 的 bit 5 改為 1，表示正等著 CPU 來拿新的 output data。但由於 CPU 正在忙其他事情，且需要先存取 0x00 的位置確認 bit 5 為 1，才會來存取 0x84 的位置，故由下圖可看到需要過很久之後 CPU 才會收走這筆 data：



接著說明由 output buffer 傳輸給 CPU 的過程：由於 CPU 發現 0x00 的 bit 5 為 1，而得知 output buffer 中的東西是可以拿取的新 data，因此將 WB 的 address 開頭設為 300（表示要存取 user project 中關於 FIR 的此區塊），且 address 結尾設為 0x84 時，表示要透過 AXI-Stream 讀出新的一筆 output data，由於我們有在此 module 中實作 output buffer 以增進效能，故是由 output buffer 提供此值給 WB Data_out，並在 10 個 cycle 的 delay 後將 ACK 拉為 1，上層就會將此資訊 forward 給 WB interface，讓 CPU 拿到運算結果。output data 傳至 WB interface 過程的波形圖如下圖所示：



由圖中可看到在 ACK 拉起來（表示成功完成 data 傳輸至 WB）後，AXI-Stream_sm 所對應的 FSM 的 state 又回到 000，也就是等著收取 FIR 計算的下一筆 output，故此時將 sm_tready 拉為 1 以等著接收 data！

4. What is the FIR engine theoretical throughput, i.e. data rate? Actually measured throughput?

FIR engine 理論上的 throughput 為處理一筆 data 所需的時間，也就是連續兩筆 output 之間的時間差，在 lab3 中由於有特別設計 tap_RAM 與 data_RAM，以及 data in/out 之間的時序，所以能夠非常有效率的 access 這些 BRAM，因此能夠在 7204 個 cycle 中就完成 600 比 data 的輸出（如下圖所示，此截圖來自 Github 上 lab3 當時的模擬結果/SOC_design/lab3/xsim.log）

```

000  [PASS] [Pattern      599] Golden answer:      -1090, Your answer:      -1090
661  -----End the data input(AXI-Stream)-----
662  [PASS] [Pattern      599] Golden answer:      -915, Your answer:      -915
663  OK: exp =          2, rdata =          6
664  OK: exp =          4, rdata =          4
665  -----
666  -----Congratulations! Pass-----
667  FIR engine latency =    7204 clock cycles
668  $finish called at time : 72935 ns : File "/home/ubuntu/lab_3/fir/tb/fir_tb.v" Line 247

```

平均約 12 個 cycle 就可完成一筆 data。而當時合成出來的結果為最快速度

對應到的 cycle time=5.9ns，因此 data rate= $\frac{1}{12 \times 5.9 \text{ ns}} = 14.125 \text{ MHz}$ 。但由於

lab4-2 中 bram11.v (tap_RAM 及 data_RAM) 的存取方式有變動，因此重新 design 以符合新的 BRAM 存取模式（無法及時完成 write 完馬上 read 的動作），這導致原本設計的 cycle 流程需要 delay 才能 read，因此平均每個 data 需再多出兩個 cycle 來處理，新的設計執行 testbench 測試後能在 simulation 中完成 3 個 cycle 的 600 筆 data，且結果正確，如下圖所示（此檔案位於 SOC-design/lab4-2/lab4-2/lab3--modification/fir/xsim.log）

```

1881  OK: exp =          0, rdata =          0
1882  [PASS] [Pattern      598] Golden answer:      -1098, Your answer:      -1098
1883  -----End the data input(AXI-Stream)-----
1884  [PASS] [Pattern      599] Golden answer:      -915, Your answer:      -915
1885  OK: exp =          2, rdata =          6
1886  OK: exp =          4, rdata =          4
1887  -----
1888  -----Congratulations! Pass-----
1889  FIR engine latency =    8403 clock cycles
1890  $finish called at time : 253455 ns : File "/home/ubuntu/lab4-2/lab3--modification/fi

```

共需要 8403 個 cycle 完成 600 筆 data，平均每筆 data 需 14 個 cycle，因此

data rate= $\frac{1}{14 \times 5.9 \text{ ns}} = 12.107 \text{ MHz}$ 。故 theoretical throughput 約為 12.107 MHz

（在 data_in 能及時順利給出、data_out 能順利及時輸出的情況下）。

實際上的 throughput 則會受到 CPU 透過 WB 給 data 而造成的 delay、

WB_to_AXI module 為了滿足兩邊的 protocol (WB v.s. AXI) 進行轉換與 buffer data 所需的 hardware 而導致更長的 critical path 而影響 timing。其中花上最多 cycle 的是 CPU 要透過 WB 從 user BRAM 中取得 firmware code/instruction，這個部份會 delay 至少 10 個 cycle，接著將 data 傳給 WB_to_AXI 的 module (以傳到更下層的 FIR engine 中) 又會需要再 delay 至少 10 個 cycle。故每個 data 的計算都至少需要跑過下列的 phase：data_in 前需要不斷存取 0x00 的位址以確認 x[n] ready 會需要至少 10 個 cycle → data input 至少需要 10 個 cycle → data output 前需要不斷存取 0x00 的位址以確認 y[n] valid 會需要至少 10 個 cycle → data output 至少需要 10 個 cycle。因此每筆 data 的運算會至少需要 40 個 cycle，還不包括 CPU 向 user BRAM 索取 instruction 的部分。由上述可知導致 cycle 數下不來的 bottleneck 在於 firmware 的相關存取與執行，再加上 CPU 與 user project 之間的資料傳遞實在是需要花太多 cycle 了。有底下兩種方式來計算平均 data rate：

➤ 方法 1：使用 latency timer 的記錄值

Latency timer 會在 FIR start 時 (已寫好 data_length 與 tap_RAM 等 status，正式開始 ap_start) 開始計算 (也就是 Start Mark ('hA5) on mprj[23:16] 被 testbench 偵測到時)，直到 counter_la_fir.c 輸出最後 11 筆 data output 至 mprj 上 (也就是 EndMark ('h5A – mprj[23:16]) 被 testbench 偵測到時) 才記錄下 cycle 數。這個方法會稍微有點不準確，因為不只有計算到 FIR 在運作的時間，還包括了 counter_la_fir.c 在輸出 data 過程的 cycle count (況且這部分是由 CPU 執行 firmware code 來達到，需要花上很多時間處理)。

實作結果如 Github 上/SOC-design/lab4-2/Simulation log/simulation.log 檔中所示：

```
++++++ Step 1: Use firmware (lab4-1) ++++++
LA Test 1 started
(Firmware) Detect start mark (8'hA5) on mprj[23:16], start latency timer.
(Firmware) Detect end mark (8'h5A) on mprj[23:16], record latency timer.
(Firmware) Success: Final Y[7:0] output to mprj[31:24] is 0x76, the same as the golden value.
Info: FIR "firmware" latency = 1510415 clock cycles

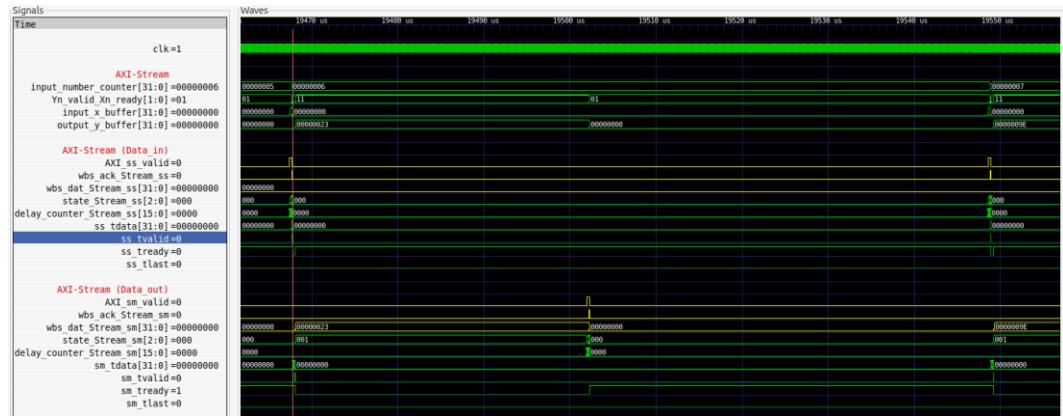
++++++ Step 2: Use hardware (FIR engine)(lab3) ++++++
(Round 1) Detect start mark (8'hA5) on mprj[23:16], start latency timer.
(Round 1) Detect end mark (8'h5A) on mprj[23:16], record latency timer.
(Round 1) Success: Final Y[7:0] output to mprj[31:24] is 0x76, the same as the golden value.
Info: FIR engine (hardware) latency in round 1 = 551675 clock cycles
(Round 2) Detect start mark (8'hA5) on mprj[23:16], start latency timer.
(Round 2) Detect end mark (8'h5A) on mprj[23:16], record latency timer.
(Round 2) Success: Final Y[7:0] output to mprj[31:24] is 0x76, the same as the golden value.
Info: FIR engine (hardware) latency in round 2 = 551675 clock cycles
(Round 3) Detect start mark (8'hA5) on mprj[23:16], start latency timer.
(Round 3) Detect end mark (8'h5A) on mprj[23:16], record latency timer.
(Round 3) Success: Final Y[7:0] output to mprj[31:24] is 0x76, the same as the golden value.
Info: FIR engine (hardware) latency in round 3 = 551675 clock cycles
Info: Total FIR engine (hardware) latency = 1655025 clock cycles
LA Test 2 passed
-----Congratulations! Pass-----
```

我們總共讓 hardware 執行了 3 個 round 的 simulation，以便觀察是否會在多次執行時沒有成功 reset 回初始狀態。並且將其與 lab4-1 的軟體比較，可發現每個 round (64 筆) 需要花上 551675 個 clock cycle (如圖中紅色框框) 才能完成，平均每筆 data 花上 8620 個 cycle 才完成，而合成出來的最大速度所對應到的 cycle time 為 10.4ns (由於有乘法器和加法器，再加上這次 lab 加

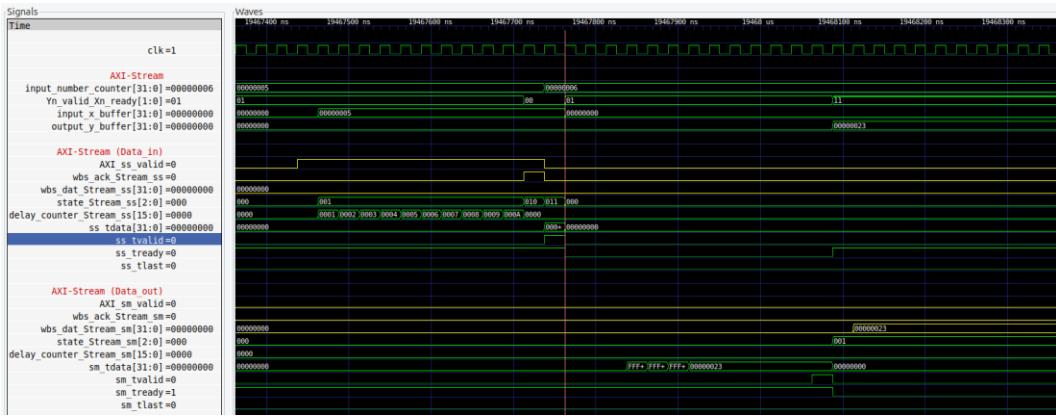
入的 protocol 轉換的 hardware 以及 user BRAM，因此 critical path 變長)，故 data rate = $\frac{1}{8620 \times 10.4 \text{ ns}} = 0.0112 \text{ MHz}$ per data。故 actually measured throughput 約為 0.0112MHz (因為 data_in 無法及時順利給出、data_out 無法順利及時輸出而導致)。雖然這比 theoretical throughput 來得慢得多了，但它還是有運用到 hardware 與 software/firmware/CPU 之間的平行運算(parallelism)，因此還是比上圖中藍色框框表示的純軟體/firmware 的 sequential 實作方式 (處理 64 筆 data 總共需要 1510415 個 cycle) 還快上許多。

➤ 方法 2：觀察 waveform 找出相鄰兩次 data_in 的時間差

Data rate 的倒數為 data period，指的是平均而言運算每個 data 所需的時間，在每次運算為對稱 (此指消耗相同 cycle 數) 的情形下，亦可以視為連續兩次相鄰 input data 之間所過的時間，或是連續兩次相鄰 output data 之間所過的時間。在此我們量的是連續兩次相鄰 input data，為了能精確計算出時間差，我們直接測量波形圖中處理 1 個 data 需要幾個 cycle，再乘以合成時得到的 cycle time 即可有 data period。如下圖所示，為了避免邊緣效應 (邊緣處可能與中間的行為稍有差異的現象)，我們從 data 數目中找了稍偏中間的位置，在此為第 6 與第 7 個 data_in，接著測量相鄰兩次 input handshakes 達成的時間點之間差了多少 clock cycle：



上圖記錄了連續兩次 (由其中的 input_number_counter 可知為第 6 筆 data 至第 7 筆 data) 的 data_in 的波形圖，波形中的 ss_tvalid (底色為藍色) 可以拿來判斷是否 handshake 發生 (傳入 data_in 給 FIR engine)，因為它的 falling edge 即表示 handshake 發生時。我們測量連續兩次 handshake 發生的時間點，放大上圖可得：



由此可知第 6 筆 data_in 的 handshake 發生的時間點為 19467762.5ns。



由此可知第 7 筆 data_in 的 handshake 發生的時間點為 19548887.5ns。

兩者之間時間差為 $19548887.5\text{ns} - 19467762.5\text{ns} = 81125\text{ns}$ 。由上圖亦可得知 clock period 為 25ns (只是模擬波形圖中所顯示的時間，並非合成結果)，因此可得共 $81125\text{ns} \div 25\text{ns} = 3245$ (個 cycle)。由合成結果我們有最大速度所對應到的 cycle time 為 10.4ns，故此方式所計算出來的 data rate (更加精確)

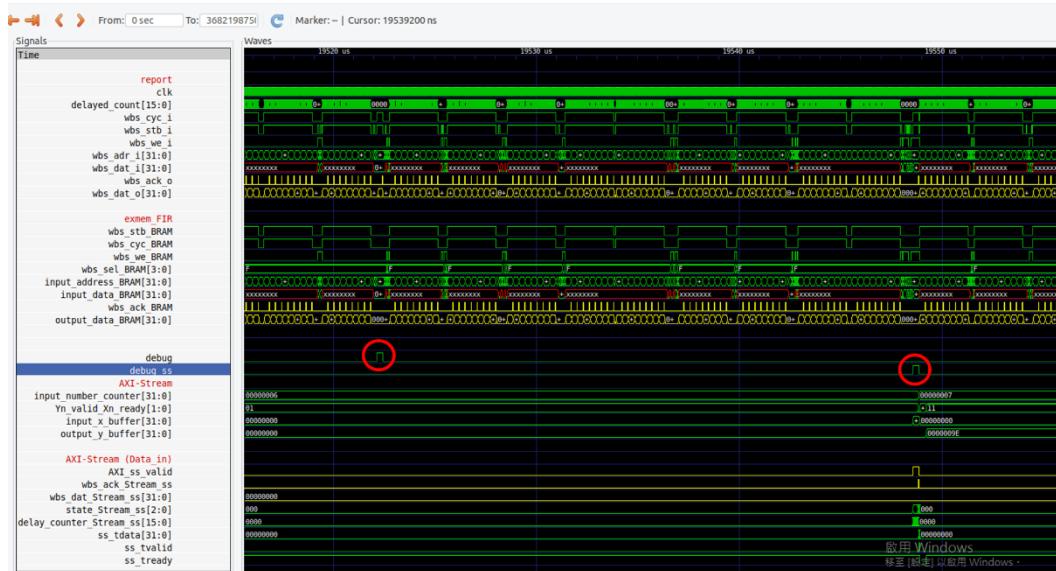
為 $\frac{1}{3245 \times 10.4\text{ ns}} = 0.0297\text{ MHz}$ ，比方法 1 所計算出來的高約一倍，由此可見

software 處理計算後的 data 也花上了大量時間！

5. What is latency for firmware to feed data?

為了讓 CPU 在 FIR 不需要接收新的 input data ($x[n]$) 時能夠先去做其他事情，並且過一段時間再回來檢查 FIR 是否需要新的 input data，因此在 configuration address map 0x00 位址中的 bit 4 存了一個值，若此值為 0，表示 WB_to_AXI module (位於/SOC-design/lab4-2/lab4-2/lab-caravel_fir/rtl/user/user_proj_example.counter.v 中，在實作時誤將此 module 的名稱定為”AXI_to_Stream”，但其內部的確是進行 WB to AXI (含 AXI-Lite + AXI-Stream) 的轉換) 中的 input buffer 仍有未使用到的值，CPU 即可知道可以先去做其他事情，晚一點再回來 check 是否需要再輸入新 input data 進來；若此 bit 值為 1，則表示 input buffer 中本來存的值已經丟進 FIR engine 中，故 CPU 就需要再輸入一筆 input data 給 input buffer 暫存。因此我們認為 firmware feed input data 的過程會從「送 request 給 0x00 位址」就開始計算，直到成功完成 input data 送至 input buffer 並 handshake 達成時才停止計算，另外底下也會說明用另一個觀點來計算，也就是純粹看 CPU 透過 WB 單純傳 input data 的這次 handshake 所耗費的時間，所量測出來的結果。

首先先說明第一個觀點 (從 0x00 的 access 就開始算)，其相關波形圖如下，同樣也是為了避免邊緣效應而取第 6 筆 input data 附近的行為：

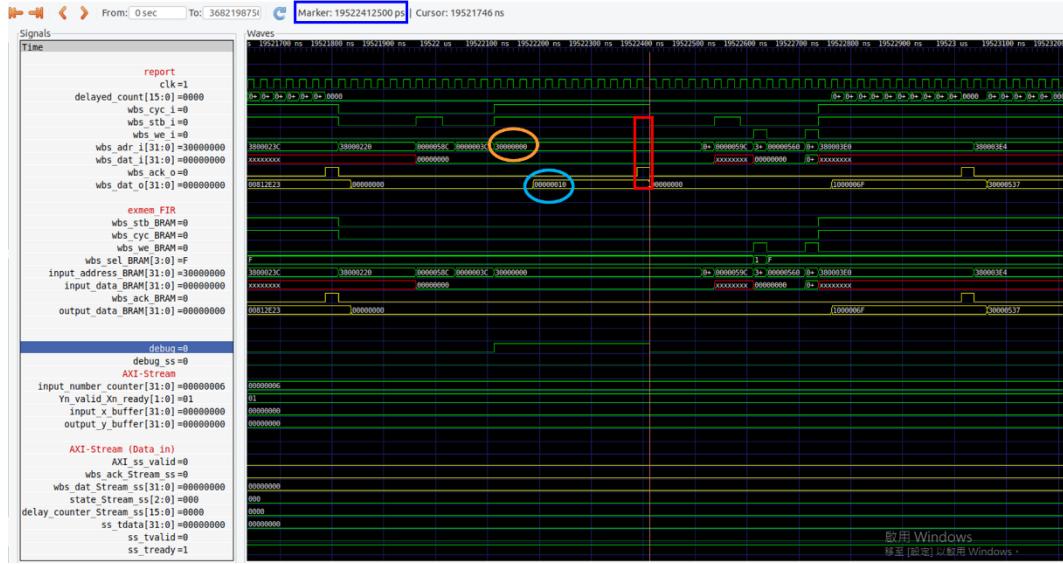


為了方便觀察我們所關心的行為的波形位置，我們 define 了一些 debug 使用的訊號：

```
assign debug = wbs_cyc_i && wbs_stb_i && (wbs_adr_i[31:20] == 12'h300) && (wbs_adr_i[7:0] == 8'h00);  
assign debug_ss = wbs_cyc_i && wbs_stb_i && (wbs_adr_i[31:20] == 12'h300) && (wbs_adr_i[7:0] == 8'h80);
```

也就是”debug”訊號會在 WB 的 stb 與 cyc 為 1，且 address 為 300 開頭 (表示為 configuration address map 中所提到的那些 data 所存放的區域)，且 00

結尾，也就是對應到 0x00 的位址（因為 300 開頭我們只定義了 30000000 這個區塊，因此只有這個地方會對 WB 的 request 有 response），此時才會為 1；而”debug_ss”訊號會在 WB 的 stb 與 cyc 為 1，且 address 為 300 開頭，且 80 結尾，也就是對應到 0x80 的位址，才會為 1。可藉由此兩個訊號快速辨別出 0x00 與 0x80 (input buffer 的位址) 這兩個 address 何時有被 access 到。我們接下來要測量出此兩個 access 時間點之間的時間差，即為所求。下圖為將 access 0x00 位址的波形放大檢視：



此時 CPU access 到 address 為 30000000 (即 configuration address map 中的 0x00)，並發現此時 WB 的 data_out 回傳(00000010)₁₆=(00010000)₂，可知 0x00 的 bit 4 位置（也就是 Xn_ready，存放在 Yn_valid_Xn_ready 這個 register 的 bit 0）的值為 1，代表 input buffer 需要接收新 data，因此 CPU 就執行下列指令：

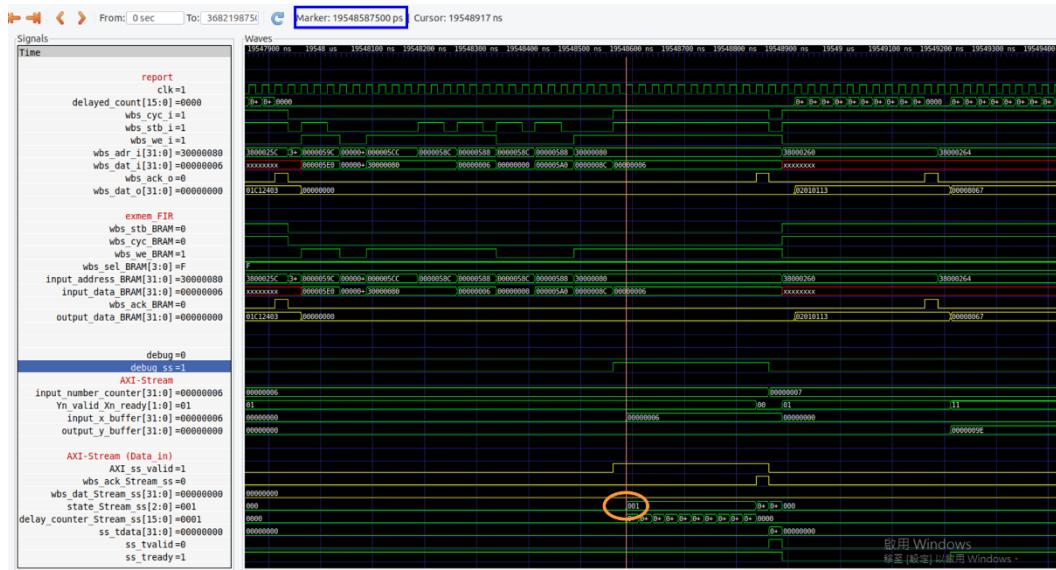
`WB_write((int*)(FIR_BASE_ADDRESS+0x80), x[input_data_count]);`

其中 WB_write()為一自定義的 function，定義為

```
void __attribute__((section(".mprjram"))) WB_write(int* WB_address, int write_data){
    *(WB_address)=write_data;
}
```

也就是將新的 data 值寫入到 0x80 (input buffer 的 address) 的位置中。由於這個過程是使用 CPU 執行 firmware code，故會花上許多 cycle 來完成。此時的時間點為 19522412.5 ns。

接著觀察實際傳遞 input data 至 input buffer 的波形，也就是上上張圖中的右邊紅色圈圈所指出的時間點：



由圖中可看到此時的 `debug_ss` 被拉為 1，表示 CPU 正透過 WB 來 access 0x80 的位址，並且 `write enable` 值為 1，表示要寫入 `input buffer` 中，而欲寫入之 `data_in` 為 `0006`，此時的確將 `input buffer` 的值成功寫入為 `0006`，故 latency 計算到此處結束，此時的時間點為 `19548587.5 ns`。

因此計算出來的 latency for firmware to feed data 為 $19548587.5 \text{ ns} - 19522412.5 \text{ ns} = 26175 \text{ ns}$ 。由於波形圖中的 clock period 為 25ns (只是模擬波形圖中所顯示的時間，並非合成結果)，因此可得為 $26175\text{ns} \div 25\text{ns} = 1047$ (個 cycle)。若使用合成結果 10.4 ns，可得 latency 為 $1047 \times 10.4 \text{ ns} = 10888.8 \text{ ns}$ 。

第二種觀點則是純粹看 CPU 透過 WB 單純傳 `input data` 的這次 handshake 所耗費的時間，由上圖可知其實就是經過一次的 WB request (`write enable=1`、`address=30000800`，即對應到 `0x80` 位址) 即可將 `data_in` 確實寫入至 `input buffer` 中，故計算出來的 latency for firmware to feed data 即為 WB 的 delay，在此次 lab 中設定為 10 個 clock cycles，但加上 `data` 送至 `WB_to_AXI module` 的時間(1 cycle)以及 ACK 回覆的時間(1 cycle)，總共需要 12 個 cycles，若使用合成結果 10.4 ns，可得 latency 為 $12 \times 10.4 \text{ ns} = 124.8 \text{ ns}$ 。

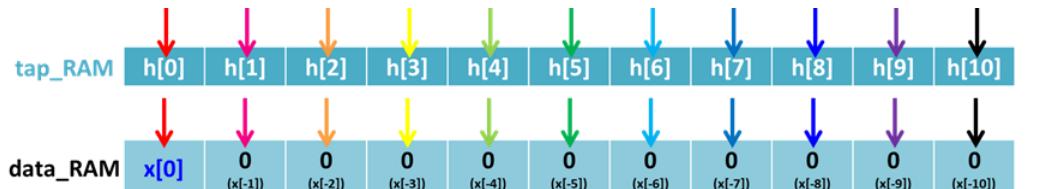
上方兩種觀點的計算結果並不相同，是因為計算的是不同層級概念的 latency，但都與 `feed data_in` 相關。

6. What techniques used to improve the throughput?

為了提升 data rate，也就是希望在越少 cycle 內（更準確地說，是越短的時間內，因此也要考慮 Maximum frequency 的影響）能夠完成一筆 data 的處理，我們使用了許多種方法來達到此目標，主要有下列 2 種方法：

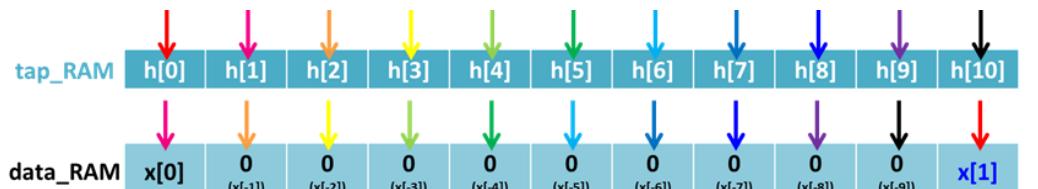
方法 1：我們延續利用了 lab3 中想到的方法：由於 tap_BRAM 與 data_BRAM 的 access 實在是耗費太多 cycle 了，況且每筆 data 都要 access 11 次左右，再加上每筆運算都需要花上這些 cycle，實在是非常浪費時間資源，因此我們要盡量在每一個 cycle 都直接 access 到想要的 address，才不會浪費。使用上述的想法，我們透過移動 address pointer 的方式來 access 到我們想要的位置，以最佳效率運用 BRAM。如同 lab 3 的 report 所述，主要是透過下列流程達到這份任務：

For 第一筆 output data $y[0]$ ：



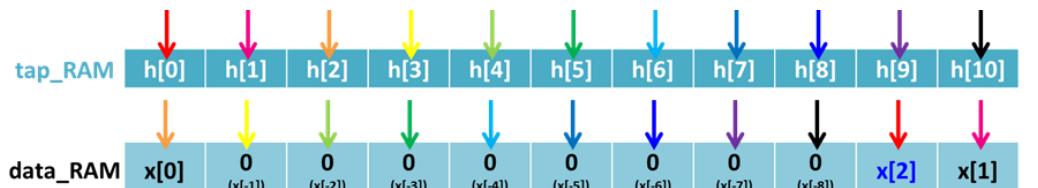
$$y[0]=h[0]x[0]+h[1]x[-1]+h[2]x[-2]+h[3]x[-3]+h[4]x[-4]+h[5]x[-5]+h[6]x[-6]+h[7]x[-7]+h[8]x[-8]+h[9]x[-9]+h[10]x[-10]$$

For 第二筆 output data $y[1]$ ：



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

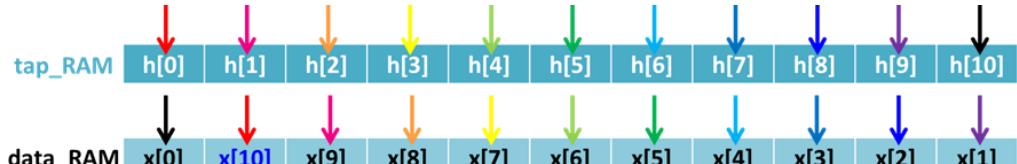
For 第三筆 output data $y[2]$ ：



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

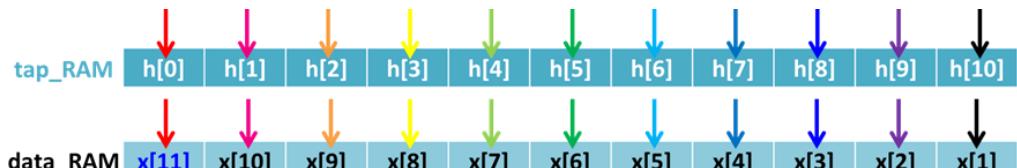
.....(以此類推).....

For data_RAM 繞了一圈寫回來時：



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

For 讀入 x[11]後（將 x[0] 覆蓋掉）：



$$y[11] = h[0]x[11] + h[1]x[10] + h[2]x[9] + h[3]x[8] + h[4]x[7] + h[5]x[6] + h[6]x[5] + h[7]x[4] + h[8]x[3] + h[9]x[2] + h[10]x[1]$$

在上方這些圖中，每當要計算一筆新的 data 時，先 input 新 data 進來，並將其儲存在 data_RAM 的上次黑色箭頭最後所指的位址 → 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（前次 data 最後的黑色箭頭處）→ 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 output sm_tdata 的值。

方法 2：在此次 lab 中，我們在 WB_to_AXI module 中加入了一組 input buffer 與 output buffer，其中 input buffer 可以預先從 CPU 拿一組 input data 暫存起來，待 FIR engine 需要時 (ss_tready 拉起來後) 可以直接透過 AXI-Stream 的 interface 紿予；而 output buffer 則是透過 AXI-Stream (sm)的方式在 FIR engine 運算完畢後將 output 值暫存起來，待 CPU 有空時再來拿取。使用這份 input buffer 及 output buffer，我們可以更加有效率地利用 hardware (FIR engine) 與 firmware (CPU) 之間的平行運算，使兩邊同時在進行處理，藉此提升 throughput。

若沒有 input buffer 的協助，則當 FIR 正在 idle，可以趕緊拿下一筆 input data 來進行運算時，卻需要等待直到 CPU 有空給 input data，而導致此種情況下反而很接近 sequential 運算（執行順序為 CPU → FIR），平行度幾乎為 0，導致平行運算的效果大打折扣。故在兩者的 interface 處，也就是 WB_to_AXI module 中，加入了 input buffer，如次一來當 CPU 有空但 FIR 仍未處理好上筆 data 時，就可以先把下筆 input data 先拿到了，之後 FIR 執行完上筆 data，

即可透過 AXI-Stream (ss)迅速拿到下筆 input data 即可開始處理，而提升 hardware 的使用率，改善 throughput！

同理，若沒有 output buffer 的協助，則當 FIR 運算完結果後，要將 output 值透過 WB_to_AXI 傳給 CPU，但 CPU 可能正在忙別的事，此時由於沒有 buffer 能將運算結果先暫存起來，導致 FIR 只能繼續等待直到 CPU 來拿 output 值，這也導致此種情況下反而很接近 sequential 運算（執行順序為 FIR→CPU），平行度幾乎為 0，導致平行運算的效果大打折扣。故在兩者的 interface 處，也就是 WB_to_AXI module 中，加入了 output buffer，如次一來當 FIR 運算完上筆 data 時但 CPU 還在忙時，就可以先透過 AXI-Stream (sm)把此筆 output data 暫存在 output buffer 中，並可立即開始進行下筆 data 的運算，之後 CPU 有空時再來向 output buffer 領取 data 即可，因而可提升 hardware 的使用率，改善 throughput！

此方法的實作過程會需要在 0x00 的位址中運用 2 個 bits 來記錄 input buffer 與 output buffer 的當下使用情形，才能讓 CPU 得知 input buffer (對應到位址為 0x80) 的值是否已被拿過(若已被 FIR 拿過則可透過 WB 的 write 功能釋出新的 request 覆寫上新的 input data，並同時將此 bit flip 為 0，表示此筆新 data 仍未被 FIR 拿過)，以及(另一 bit)讓 CPU 得知 output data 的值是否已經計算完畢而可以領取(若有新的 output data 尚未領取，則值為 1，此時 CPU 就可以透過 WB 的 read 功能到 0x84 的位址拿取 output data，並同時將此 bit flip 為 0，表示此筆新 output data 已被領取過)。

6-1. Does bram12 give better performance, in what way?

在這次的 lab 實作上，我們延續 lab3 的設計方法，使用 bram11.v，而沒有使用到 bram12.v。而若使用 bram12.v 的好處有：

	Bram11	Bram12
儲存空間	$2^{11}=2048$ bits	$2^{12}=4096$ bits
面積	小	大
功耗	小	大
成本	小	大
設計複雜程度	簡單	複雜

Summary:若使用 bram12 可以讓儲存的空間變大，在 data RAM 的使用上就可以多存一筆 data，而在這次 lab 的電路中若使用 bram12 我們認為會讓計算更快，因為 CPU 的運算實在是太慢了，硬體的 FIR 每次做完都要等很久才有下一筆 input data，所以如果可以多一個位置暫存 input data 感覺就像是擴充 input buffer，CPU 可以先一口氣傳兩筆 input 這樣之後 CPU 在忙其他事的時候 FIR 至少可以做兩筆，減少 latency。但缺點就是如同上表所述，運

用 Bram12 會讓面積功耗等等變的更大，所以在電路設計時也要考量到是要追求甚麼目標，避免使用了一個不會帶來益處的方式。

6-2. Can you suggest other method to improve the performance?

FoM (performance metrics) 包含下列三項：

- Number of clock cycles (the latency-timer in testbench) : 551675 cycles
- Clock period (worst-case timing) : 10.4 ns
- Gate resources(number of LUT+FF) : 798 (LUT) + 520 (FF) = 1318 pieces

因此，FoM (performance metrics)

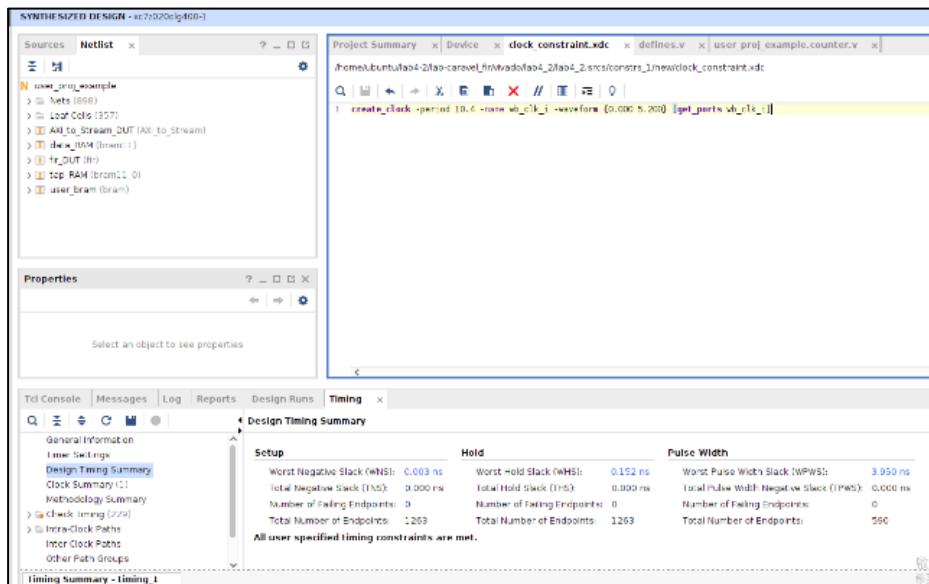
$$= \text{Number of clock cycles} \times \text{Clock period} \times \text{Gate resources}$$

$$= 7561919560 \text{ (cycles}\cdot\text{ns)}$$

7. Any other insights

7-1. Synthesis report

我們將 design (包括 fir.c、fir.h、user_proj_example.counter.v、bram.v，以及修改 testbench 以偵測 output signal 並驗證其正確性) 加入 Vivado 軟體中 (simulation 是透過 run_sim 的 script 執行的，並無使用 Vivado)，並加上 timing constraint 後，執行 synthesis，可得到如下圖的結果，可看到最高速度大約是 clock period ≈ 10.4ns，對應到速度約為 96.154MHz。此時的 slack 為正值。



(1) User_proj_example_utilization_synth.rpt

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

FF & LUT:

1. Slice Logic						
Site Type	Used	Fixed	Prohibited	Available	Util%	
Slice LUTs*	798	0	0	53200	1.50	
LUT as Logic	734	0	0	53200	1.38	
LUT as Memory	64	0	0	17400	0.37	
LUT as Distributed RAM	64	0				
LUT as Shift Register	0	0				
Slice Registers	520	0	0	106400	0.49	
Register as Flip Flop	517	0	0	106400	0.49	
Register as Latch	3	0	0	106400	<0.01	
F7 Muxes	0	0	0	26600	0.00	
F8 Muxes	0	0	0	13300	0.00	

BRAM:

2. Memory						
Site Type	Used	Fixed	Prohibited	Available	Util%	
Block RAM Tile	4	0	0	140	2.86	
RAMB36/FIFO*	4	0	0	140	2.86	
RAMB36E1 only	4					
RAMB18	0	0	0	280	0.00	

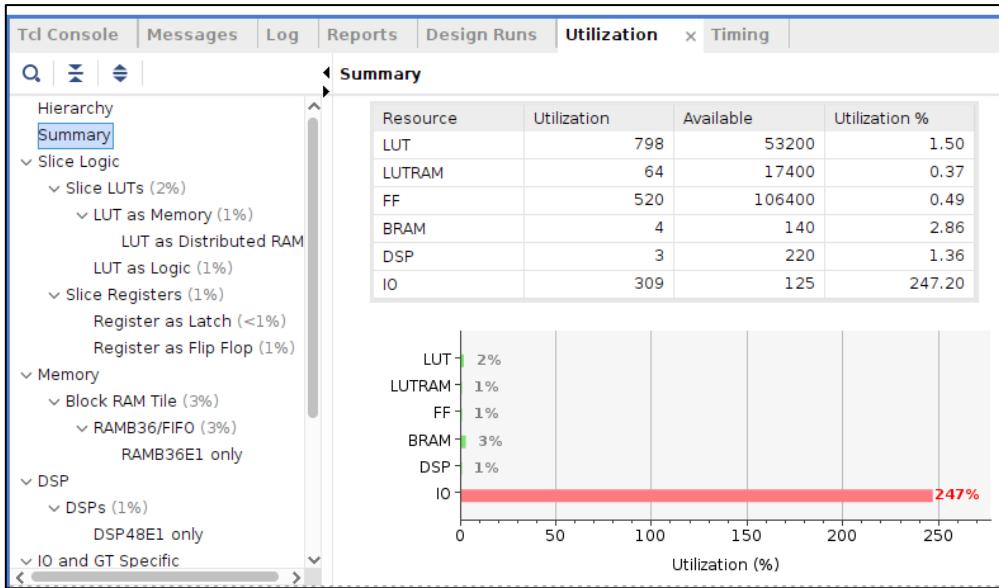
DSP:

3. DSP						
Site Type	Used	Fixed	Prohibited	Available	Util%	
DSPs	3	0	0	220	1.36	
DSP48E1 only	3					

IO:

4. IO and GT Specific						
Site Type	Used	Fixed	Prohibited	Available	Util%	
Bonded IOB	309	0	0	125	247.20	
Bonded IPADs	0	0	0	2	0.00	
Bonded IOPADs	0	0	0	130	0.00	
PHY_CONTROL	0	0	0	4	0.00	
PHASER_REF	0	0	0	4	0.00	
OUT_FIFO	0	0	0	16	0.00	
IN_FIFO	0	0	0	16	0.00	
IDELAYCTRL	0	0	0	4	0.00	
IBUFDS	0	0	0	121	0.00	
PHASER_OUT/PHASER_OUT_PHY	0	0	0	16	0.00	
PHASER_IN/PHASER_IN_PHY	0	0	0	16	0.00	
IDELAYE2/IDELAYE2_FINEDELAY	0	0	0	200	0.00	
ILOGIC	0	0	0	125	0.00	
OLOGIC	0	0	0	125	0.00	

(2) In Vivado Utilization Summary



由(1)、(2)點的 report 結果可知，此 design 合成後所使用的

1. FF 個數為 520 個，而此 FPGA 板中共有 106400 個，故 utilization 為 0.49%
2. LUT 個數為 798 個，而此 FPGA 板中共有 53200 個，故 utilization 為 1.50%
3. BRAM 使用 4 個，而此 FPGA 板中共有 140 個，故 utilization 為 2.86%

(3) User_proj_example.vds

```
Detailed RTL Component Info :
+---Adders :
    2 Input   32 Bit      Adders := 4
    2 Input   16 Bit      Adders := 4
    2 Input   12 Bit      Adders := 4
    2 Input   6 Bit       Adders := 2
+---Registers :
    32 Bit     Registers := 16
    16 Bit     Registers := 4
    12 Bit     Registers := 3
    6 Bit      Registers := 1
    4 Bit      Registers := 2
    1 Bit      Registers := 19
+---Multipliers :
    32x32  Multipliers := 1
+---RAMs :
    128K Bit   (4096 X 32 bit)      RAMs := 1
    352 Bit    (11 X 32 bit)        RAMs := 2
```

```
+---Muxes :  
 2 Input 32 Bit      Muxes := 32  
 4 Input 32 Bit      Muxes := 4  
 5 Input 32 Bit      Muxes := 3  
 3 Input 32 Bit      Muxes := 2  
 8 Input 32 Bit      Muxes := 5  
 2 Input 16 Bit      Muxes := 9  
 4 Input 16 Bit      Muxes := 2  
 5 Input 16 Bit      Muxes := 1  
 5 Input 12 Bit      Muxes := 1  
 2 Input 12 Bit      Muxes := 7  
 3 Input 12 Bit      Muxes := 3  
 4 Input 12 Bit      Muxes := 1  
 8 Input 12 Bit      Muxes := 2  
 2 Input 8 Bit       Muxes := 1  
 2 Input 6 Bit       Muxes := 3  
 3 Input 6 Bit       Muxes := 1  
 8 Input 6 Bit       Muxes := 1  
 5 Input 5 Bit       Muxes := 1  
 2 Input 5 Bit       Muxes := 8  
 2 Input 4 Bit       Muxes := 6  
 4 Input 4 Bit       Muxes := 1  
 8 Input 4 Bit       Muxes := 4  
 2 Input 3 Bit       Muxes := 1  
 8 Input 3 Bit       Muxes := 1  
 24 Input 3 Bit      Muxes := 1  
 4 Input 2 Bit       Muxes := 2  
 2 Input 2 Bit       Muxes := 7  
 3 Input 2 Bit       Muxes := 1  
 4 Input 1 Bit       Muxes := 12  
 2 Input 1 Bit       Muxes := 35  
 5 Input 1 Bit       Muxes := 7  
 3 Input 1 Bit       Muxes := 3  
 6 Input 1 Bit       Muxes := 1  
 8 Input 1 Bit       Muxes := 14
```

(4) Timing_report_10_4ns.txt

使用 Vivado 匯入 design 後，執行 synthesis，並盡量設定最高速度，此 design 所能達到的最快速度約為 clock period=10.4ns，換算下來 maximum frequency 約為 96.154MHz)，再按照 lab3 作業區的 SYN_Workflow.pptx 的操作流程，可得到合成後的 timing report，即 timing_report.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）

Clock period: 10.4ns

Max Delay Paths				
Location	Delay type	Incr(ns)	Path(ns)	Netlist Resource(s)
	(clock wb_clk_i rise edge)			
		0.000	0.000	r
		0.000	0.000	r wb_clk_i (IN)
	net (fo=0)	0.000	0.000	wb_clk_i
				r wb_clk_i_IBUF_inst/I
	IBUF (Prop_ibuf_I_0)	0.972	0.972	r wb_clk_i_IBUF_inst/0
	net (fo=1, unplaced)	0.800	1.771	wb_clk_i_IBUF
				r clk_BUFG_inst_i_1/I0
	LUT3 (Prop_lut3_I0_0)	0.124	1.895	r clk_BUFG_inst_i_1/0
	net (fo=1, unplaced)	0.800	2.695	clk
				r clk_BUFG_inst/I
	BUFG (Prop_bufg_I_0)	0.101	2.796	r clk_BUFG_inst/0
	net (fo=592, unplaced)	0.584	3.380	fir_DUT/U0_MAC/clk_BUFG
	DSP48E1			r fir_DUT/U0_MAC/product_0/CLK

DSP48E1 (Prop_DSP48E1_CLK_PCOUT[47])	4.206	7.586	r	fir_DUT/U0_MAC/product_0/PCOUT[47]
net (fo=1, unplaced)	0.055	7.641	r	fir_DUT/U0_MAC/product_0_n_106
			r	fir_DUT/U0_MAC/product_1/PCIN[47]
DSP48E1 (Prop_DSP48E1_PCIN[47]_P[0])	1.518	9.159	r	fir_DUT/U0_MAC/product_1/P[0]
net (fo=2, unplaced)	0.800	9.959	r	fir_DUT/U0_MAC/product_1_n_105
LUT2 (Prop_lut2_I0_0)	0.124	10.083	r	fir_DUT/U0_MAC/product_carry_i_3/I0
net (fo=1, unplaced)	0.000	10.083	r	fir_DUT/U0_MAC/product_carry_i_3_n_0
			r	fir_DUT/U0_MAC/product_carry_S[1]
CARRY4 (Prop_carry4_S[1]_CO[3])	0.533	10.616	r	fir_DUT/U0_MAC/product_carry_CO[3]
net (fo=1, unplaced)	0.009	10.625	r	fir_DUT/U0_MAC/product_carry_n_0
			r	fir_DUT/U0_MAC/product_carry_0/CI
CARRY4 (Prop_carry4_CI_CO[3])	0.117	10.742	r	fir_DUT/U0_MAC/product_carry_0_CO[3]
net (fo=1, unplaced)	0.000	10.742	r	fir_DUT/U0_MAC/product_carry_0_n_0
			r	fir_DUT/U0_MAC/product_carry_1/CI
CARRY4 (Prop_carry4_CI_0[3])	0.331	11.073	r	fir_DUT/U0_MAC/product_carry_1_0[3]
net (fo=2, unplaced)	0.629	11.702	r	fir_DUT/U0_MAC/product_3[27]
			r	fir_DUT/U0_MAC/sum_out_carry_5_i_1/I0
LUT2 (Prop_lut2_I0_0)	0.307	12.009	r	fir_DUT/U0_MAC/sum_out_carry_5_i_1/0
net (fo=1, unplaced)	0.000	12.009	r	fir_DUT/U0_MAC/sum_out_carry_5_i_1_n_0
			r	fir_DUT/U0_MAC/sum_out_carry_5_S[3]
CARRY4 (Prop_carry4_S[3]_CO[3])	0.376	12.385	r	fir_DUT/U0_MAC/sum_out_carry_5_CO[3]
net (fo=1, unplaced)	0.000	12.385	r	fir_DUT/U0_MAC/sum_out_carry_5_n_0
			r	fir_DUT/U0_MAC/sum_out_carry_6/CI
CARRY4 (Prop_carry4_CI_0[3])	0.331	12.716	r	fir_DUT/U0_MAC/sum_out_carry_6_0[3]
net (fo=1, unplaced)	0.618	13.334	r	fir_DUT/U0_MAC/MAC_output[31]
			r	fir_DUT/U0_MAC/sm_tdata_reg[31]_i_2/I1
LUT4 (Prop_lut4_I1_0)	0.307	13.641	r	fir_DUT/U0_MAC/sm_tdata_reg[31]_i_2/0
net (fo=1, unplaced)	0.000	13.641	r	fir_DUT/sm_tdata_before_FF[31]
FDCE			r	fir_DUT/sm_tdata_reg[31]/D

(clock wb_clk_i rise edge)	10.400	10.400	r	
	0.000	10.400	r	wb_clk_i (IN)
net (fo=0)	0.000	10.400	r	wb_clk_i_IBUF_inst/I
IBUF (Prop_ibuf_I_0)	0.838	11.238	r	wb_clk_i_IBUF_inst/0
net (fo=1, unplaced)	0.760	11.998	r	wb_clk_i_IBUF
			r	clk_BUFG_inst_i_1/I0
LUT3 (Prop_lut3_I0_0)	0.100	12.098	r	clk_BUFG_inst_i_1/0
net (fo=1, unplaced)	0.760	12.858	r	clk
			r	clk_BUFG_inst/I
BUFG (Prop_bufg_I_0)	0.091	12.949	r	clk_BUFG_inst/0
net (fo=592, unplaced)	0.439	13.388	r	fir_DUT/clk_BUFG
FDCE			r	fir_DUT/sm_tdata_reg[31]/C
clock pessimism	0.248	13.635		
clock uncertainty	-0.035	13.600		
FDCE (Setup_fdce_C_D)	0.044	13.644		fir_DUT/sm_tdata_reg[31]
				slack
		13.644		
		-13.641		
				0.003

7-2. Latency Calculating in Testbench

在這次 lab 中，我們有按照題目的指示，共跑 3 個 round，每個 round 執行 64 筆 data 的 input 及 output，並 check 最終的結果是否正確。

並且為了 debug 方便，我們有在 fir.c 中寫入判別式，比較 AXI-Lite 讀取出來的 ap 值、tap 值等結果的正確性，若過程中有錯誤就會馬上回傳”-2”並結束模擬，當 testbench 接收到 checkbit 上的值為”-2”時，就會在螢幕上 print 出 Error 相關資訊，如下圖所示為過程中曾經出現的錯誤截圖。

```
ubuntu@ubuntu2004:~/lab4-2/lab-caravel_fir/testbench/counter_la_fir$ source run_sim
Reading counter_la_fir.hex
counter_la_fir.hex loaded into memory
Memory 5 bytes = 0x6f 0x00 0x00 0x0b 0x13
VCD info: dumpfile counter_la_fir.vcd opened for output.
LA Test 1 started
(Round 1) Detect start mark (8'hA5) on mprj[23:16], start latency timer.
Error: Something wrong, so fir.c returned "-2" !!!
-----Simulation Failed-----
```

當通過層層難關（fir.c 的 check、testbench 的 check）後，若過程中的 return 結果都正確，最後就會 print 出成功訊息，並給出題目所需要的 latency timer 值。最終成功的 simulation 結果如下（當執行”source run_sim”的指令後在螢幕上 print 出的資訊）：

```
ubuntu@ubuntu2004:~/lab4-2/lab-caravel_fir/testbench/counter_la_fir$ source run_sim
Reading counter_la_fir.hex
counter_la_fir.hex loaded into memory
Memory 5 bytes = 0x6f 0x00 0x00 0x0b 0x13
VCD info: dumpfile counter_la_fir.vcd opened for output.

++++++ Step 1: Use firmware (lab4-1) ++++++
LA Test 1 started
(Firmware) Detect start mark (8'hA5) on mprj[23:16], start latency timer.
(Firmware) Detect end mark (8'h5A) on mprj[23:16], record latency timer.
(Firmware) Success: Final Y[7:0] output to mprj[31:24] is 0x76, the same as the golden value.
Info: FIR "firmware" latency = 1510415 clock cycles

++++++ Step 2: Use hardware (FIR engine)(lab3) ++++++
(Round 1) Detect start mark (8'hA5) on mprj[23:16], start latency timer.
(Round 1) Detect end mark (8'h5A) on mprj[23:16], record latency timer.
(Round 1) Success: Final Y[7:0] output to mprj[31:24] is 0x76, the same as the golden value.
Info: FIR engine (hardware) latency in round 1 = 551675 clock cycles
(Round 2) Detect start mark (8'hA5) on mprj[23:16], start latency timer.
(Round 2) Detect end mark (8'h5A) on mprj[23:16], record latency timer.
(Round 2) Success: Final Y[7:0] output to mprj[31:24] is 0x76, the same as the golden value.
Info: FIR engine (hardware) latency in round 2 = 551675 clock cycles
(Round 3) Detect start mark (8'hA5) on mprj[23:16], start latency timer.
(Round 3) Detect end mark (8'h5A) on mprj[23:16], record latency timer.
(Round 3) Success: Final Y[7:0] output to mprj[31:24] is 0x76, the same as the golden value.
Info: FIR engine (hardware) latency in round 3 = 551675 clock cycles
Info: Total FIR engine (hardware) latency = 1655025 clock cycles
LA Test 2 passed
-----Congratulations! Pass-----
```

其中 Step 1 為我們按照 Github 討論區中的說明，先用 lab4-1 的軟體所轉成的 firmware code 使用 CPU 執行一遍，並確保最終的 output 為正確的。接著進入 Step 2 才是使用 lab 3 的 FIR hardware 為基底稍作修改（以符合新版的 bram11.v），並透過 firmware 當作 lab 3 中的 testbench 功能給予 input data 及比較 output data，在螢幕上 print 出的最終結果。而由圖中可知每個 round 都有執行成功，且結果皆正確無誤，每個 round 皆使用了 551675 個 cycles，總共花費 1655025(=551675+551675+551675)個 cycles。相較於 Step 1 的軟體（一個 round 就需要耗費 1510415 個 cycles）執行速度快了很多。