國立台灣大學電機資訊學院電子工程學研究所

系統晶片設計實驗
Soc Design Laboratory

Lab4-2 Report

Caravel User Project - FIR

學生： M11202109 蘇柏丞
M11202103 陳泓宇
M11202207 呂彥霖

老師： 賴 瑾

中華民國 112 年 11 月 9 日

一、 Introduction：

此次實驗使用 Caravel SoC 來完成，將我們在 Lab3 做完的 FIR 硬體放入此 SoC 的 user_project 的地方，來做實現 FIR 功能的加速，並且撰寫 firmware code，將其燒錄至 spi flash，並再寫入至 Bram 中，讓 RISC-V 讀取後來控制我們的加速電路。

此實驗中硬體的部份我們需要將 RISC-V 的控制訊號分給 FIR 以及 Bram 兩部分，因此撰寫了一個 WB_decoder.v 來做兩部分的控制，而 FIR 的部分，由於中間使用了 wishbone bus，因此需要做 wishbone bus 到 AXI lite 和 stream 的 interface 的轉換，而 wishbone bus 與 Bram 的溝通則是在 Lab4-1 完成並放入作業中。

Firmware 部分，使用了 MMIO 的方式來做硬體的控制、產生 FIR 要輸入的 data，以及控制 testbench 的 latency-timer 和與 CPU 之間的溝通。

二、 Block Diagram：

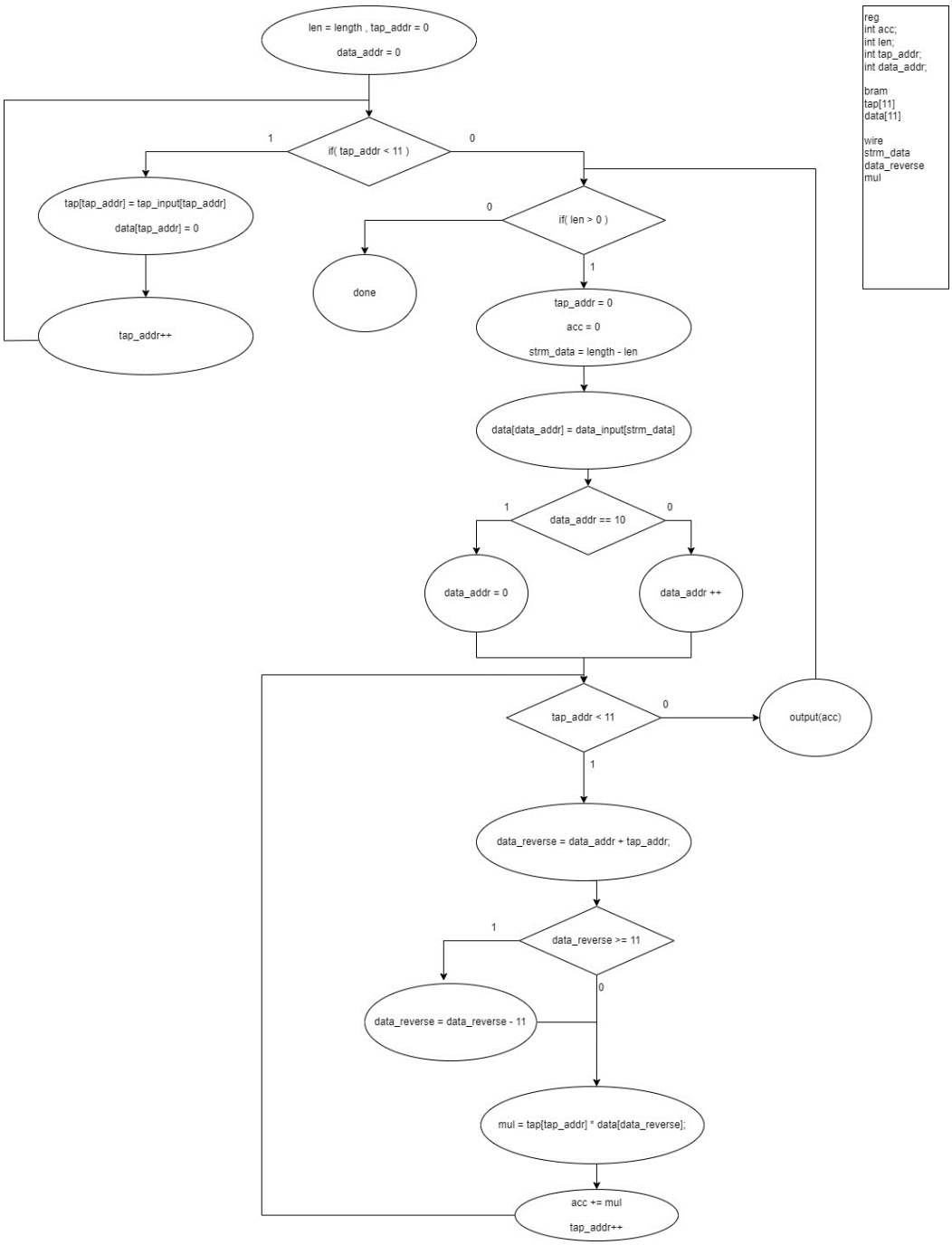
- Datapath

此次實驗我們先使用了 c 語言做功能的模擬，除了卷積功能的模擬以外還有做到 BRAM 的位址該如何選擇的部分，而不使用移位的方式來做值的讀取以及卷積，大致將功能描述出來後畫流程圖，如圖(一)以及圖(二)。

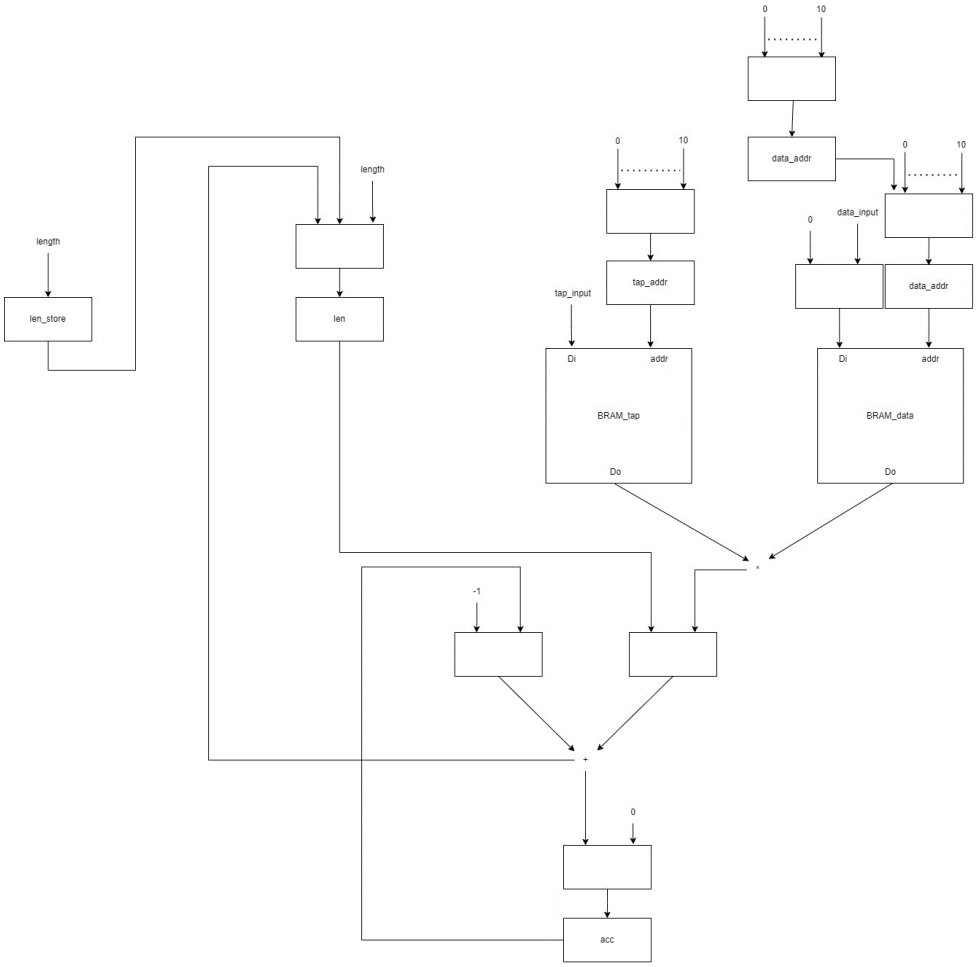
此次實驗我們設計成全部只使用一個加法以及一個乘法，並用 resource sharing 的方式共用加法器的部分，而會用到加法器的有兩個部分，第一個就是卷積的部分，需要做加法，另一個是紀錄目前是第幾筆資料的暫存器，而我們使用減法的方式來更新目前筆數的資料，因此我們的加法是丟入-1 也就是 32’hFFFF_FFFF 的方式來做到減一的操作。

暫存器的部分我們使用了五組個暫存器，其中 tap_addr 和 data_addr 是使用了 4bits，紀錄目前的位址，以及一個 4bits 的暫存器用於紀錄這次的 data_addr 該從哪個位址開始，剩下三個 32bits 的暫存器用於紀錄目前筆數、此次輸入的 length 值、以及用於卷積計算的累加器。

除了紀錄輸入的 length 值的暫存器外，都使用 mux 來做選擇輸入的動作，因此控制訊號就是圍繞這些 mux，以及我們使用了 enable 來控制是否寫入此暫存器。


len = length , tap_addr = 0
data_addr = 0
if(tap_addr < 11)
1
0
tap[tap_addr] = tap_input[tap_addr]
data[tap_addr] = 0
tap_addr++
if(len > 0)
0
1
done
tap_addr = 0
acc = 0
strm_data = length - len
data[data_addr] = data_input[strm_data]
data_addr == 10
1
0
data_addr = 0
data_addr ++
tap_addr < 11
0
output(acc)
1
data_reverse = data_addr + tap_addr;
data_reverse >= 11
1
0
data_reverse = data_reverse - 11
mul = tap[tap_addr] * data[data_reverse];
acc += mul
tap_addr++
reg
int acc;
int len;
int tap_addr;
int data_addr;
bram
tap[11]
data[11]
wire
strm_data
data_reverse
mul

圖(一)



圖(二)

- FSM

1. IDLE：

   在 IDLE 中會等待 awaddr 以及 wdata 來做功能的選擇，能夠進到 RD_LEN、RD_TAP、RD_Xn，此狀態會將 ap_idle 拉起，讓 tb 知道可以進行其他操作。

2. RD_LEN：

   讀取 length 的值，保存至暫存器，完成後回到 IDLE。

3. RD_TAP：

   讀取 11 個 tap 的值到 tap BRAM，並同時將 data BRAM 清零，完成後進到 CK_TAP。

4. CK_TAP：

   將讀取到 tap BRAM 的數值傳至 tb，在經由 tb 做 tap 的核對，完成後進到 IDLE 等待新的指令。

5. RD_Xn：

   由 stream-in 讀取 data 後將 data 寫入至 data BRAM 中正確的位置，讀到資料後進到 MUL_ADD，進行卷積運算。

6. MUL_ADD：

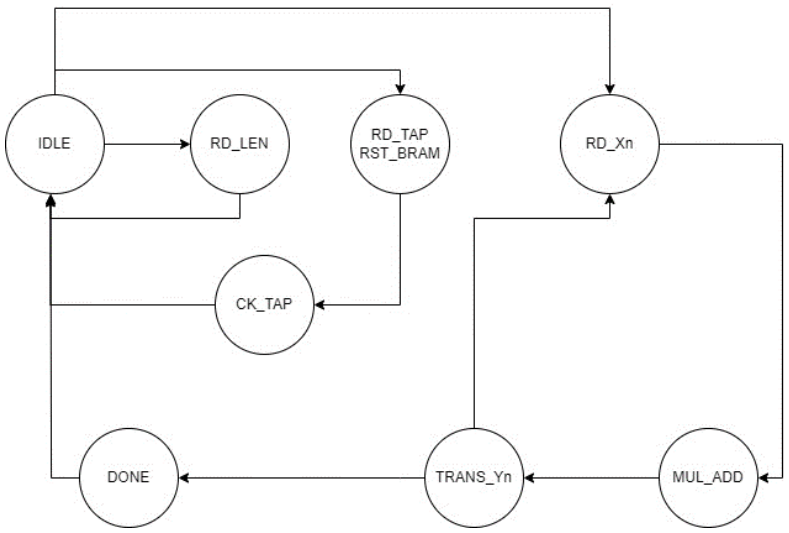
   開始執行 11 個週期的卷積運算，每個週期會讀出對應的 data 和 tap 做相乘，最後在將乘法的結果和累加器的結果相加後放入累加器暫存器中，完成後進到 TRANS_Yn。

7. TRANS_Yn：

   將計算的結果經由 stream-out 送到 tb 端，並等到 tb 端回傳 ready 訊號後，若全部筆數已完成，則進入 DONE，否則就是回到 RD_Xn 繼續下一個讀取、做卷積的循環，此狀態在將要進入 DONE 狀態時會將 ap_done 拉起。

8. DONE：

   此狀態會將 data BRAM 的值在次清空，這樣到 IDLE 時若要直接進到 RD_Xn 做讀 data，卷積的動作時結果才會正確。



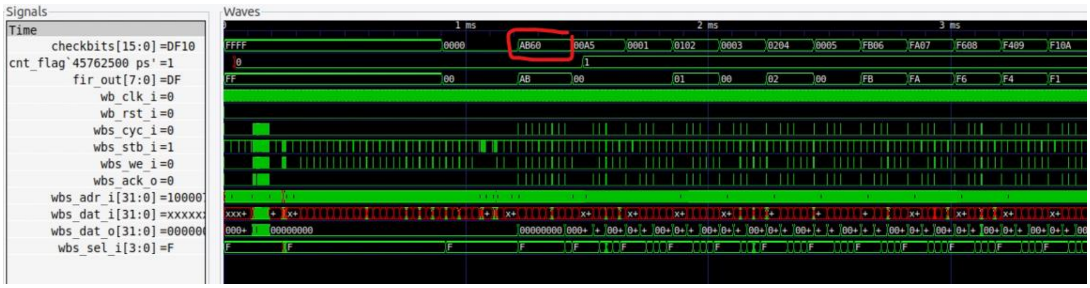
圖(三)

三、 Waveform and analysis of the hardware/software behavior

由圖(四)的 firmware code 中可以看到，在打出 0xAB60 後會先確定一次 FIR 是否 IDLE 如圖(五)。

```
int fir_times;
int counter;
int data;
//int taps[11] = {0,-10,-9,23,56,63,56,23,-9,-10,0};
int taps[11] = {0,1,-2,3,-4,-3,4,-3,2,-1,0}; // tap trasnfer
for(fir_times = 0; fir_times < 3 ; fir_times++){
        reg_mprj_datal = 0xAB600000;//start

        while((reg_fir_control & 0x00000004) == 0x00000004); // check idle

        reg_fir_length = N;         // data length trasnfer
        reg_fir_coeff0 = taps[10];
        reg_fir_coeff1 = taps[9];
        reg_fir_coeff2 = taps[8];
        reg_fir_coeff3 = taps[7];
        reg_fir_coeff4 = taps[6];
        reg_fir_coeff5 = taps[5];
        reg_fir_coeff6 = taps[4];
        reg_fir_coeff7 = taps[3];
        reg_fir_coeff8 = taps[2];
        reg_fir_coeff9 = taps[1];
        reg_fir_coeff10 = taps[0];

        while((reg_fir_control & 0x00000004) == 0x00000004); // check idle

        reg_fir_control = 0x00000001; // ap_start assert

        reg_mprj_datal = 0x00A50000;  // let latency-timer (in testbench) start counting
```

圖(四)

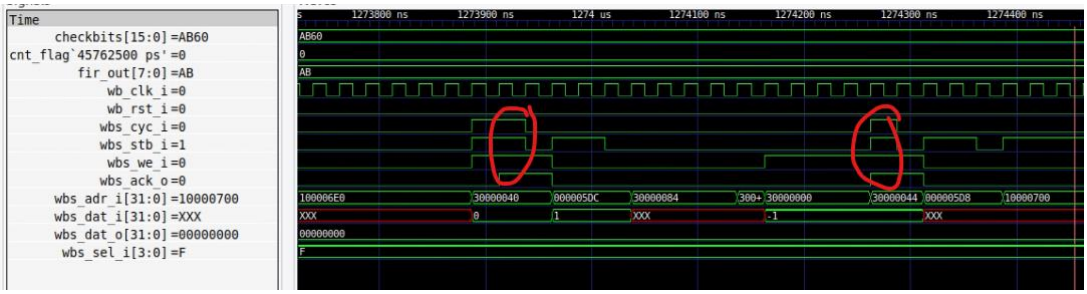


圖(五)

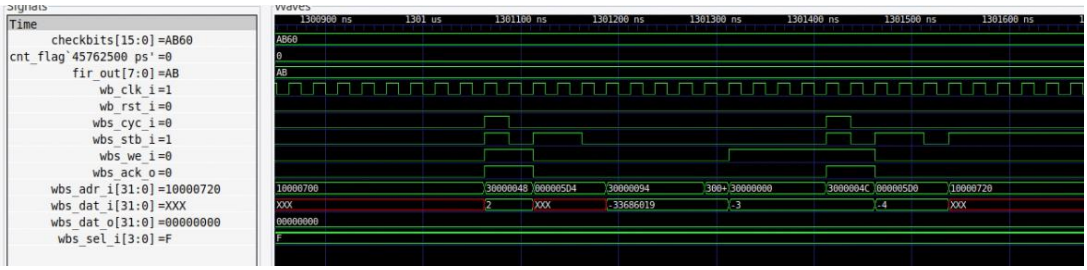
讀到 FIR 的狀態且為 IDLE 後就開始傳送 length，如圖(六)波型。

圖(六)

以及傳送 data 的部分，如圖(七)和圖(八)波型。

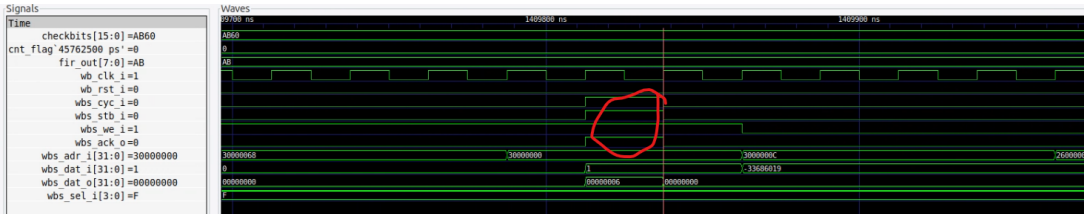


圖(七)

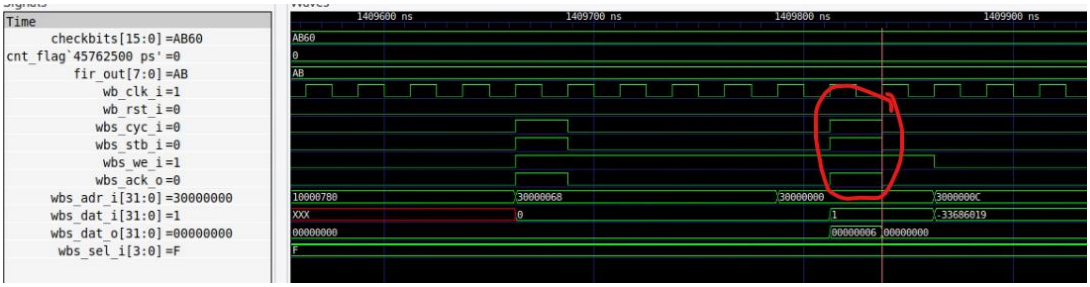


圖(八)

在傳送完這些東西之後會再次檢查是否回到 IDLE 後，打出 ap_start 並開始計數如圖(九)和圖(十)波型。



圖(九)



圖(十)

接著會打出 0xA500 讓 TB 端開始做 latency 的計數，TB 端為圖(十一)和其對應波型圖(十二)。

```
/////////////////////// latency-timer counting part ///////////////////////
initial begin
        latercy_cnt = 0;
        cnt_flag = 0;
        wait(checkbits == 16'h00A5);
        $display("latency-timer start counting");
        cnt_flag = 1;
        wait((checkbits & 16'h005A) == 16'h005A);
        $display("latency-timer : %d (cycles)" , latercy_cnt);
        cnt_flag = 0;
        #200;
        latercy_cnt = 0;
        cnt_flag = 0;
        wait(checkbits == 16'h00A5);
        $display("latency-timer start counting");
        cnt_flag = 1;
        wait((checkbits & 16'h005A) == 16'h005A);
        $display("latency-timer : %d (cycles)" , latercy_cnt);
        cnt_flag = 0;
        #200;
        latercy_cnt = 0;
        cnt_flag = 0;
        wait(checkbits == 16'h00A5);
        $display("latency-timer start counting");
        cnt_flag = 1;
        wait((checkbits & 16'h005A) == 16'h005A);
        $display("latency-timer : %d (cycles)" , latercy_cnt);
        cnt_flag = 0;
        #200;
        $finish;
end
```

圖(十一)

圖(十二)

接著進到 FIR 主要的功能部份，firmware 的部分使用一層迴圈來完成，N=64，傳送 64 筆由 CPU 產生的 data 到 FIR，每次會先讀取 Xn 是否可以傳送，直到 FIR 回送準備好的信號後傳送 Xn。

再來會去讀取 Yn 是否運算完成，讀到後會收取 FIR 運算完的 data，並將其左移 24bit，此舉是為了讓 data 對齊 mprj，使其由 mprj[31:24]傳送回 TB 端，接著判斷是否為最後一筆資料，若為最後一筆資料則在 mprj[23:16]輸出 0x5A；若不是最後一筆資料則在 mprj[23:16]輸出目前的計數數字，讓 TB 端可以將 data 和在 TB 端產生的 golden data 做比對並輸出結果是否正確。

最後將 FIR 的狀態清除並重複以上動作直到 data 傳送完。如圖(十三)。

```
for(counter = 1 ; counter < N ; counter++){

        while((reg_fir_control & 0x00000010) == 0x00000010); // X[n]_ready to accept input

        reg_fir_x = counter;

        while(reg_fir_control == 0x00000020); // Y[n] is ready to read

        data = reg_fir_y << 24;

        if(counter == N - 1){
                reg_mprj_datal = data + 0x005A0000; // let latency-timer (in testbench) stop counting
        }else{
                reg_mprj_datal = data + (counter << 16);
        }


        reg_fir_control = 0x00000000;

}
```

圖(十三)

而在 TB 端的部分會去做 latency timer 的控制，以及 TB 會去比對資料的正確性後，將結果顯示出來，如以下四張圖。

```
/////////////////////// latency-timer counting part ////////////
initial begin
        latercy_cnt = 0;
        cnt_flag = 0;
        wait(checkbits == 16'h00A5);
        $display("latency-timer start counting");
        cnt_flag = 1;
        wait((checkbits & 16'h005A) == 16'h005A);
        $display("latency-timer : %d (cycles)" , latercy_cnt);
        cnt_flag = 0;
        #200;
        latercy_cnt = 0;
        cnt_flag = 0;
        wait(checkbits == 16'h00A5);
        $display("latency-timer start counting");
        cnt_flag = 1;
        wait((checkbits & 16'h005A) == 16'h005A);
        $display("latency-timer : %d (cycles)" , latercy_cnt);
        cnt_flag = 0;
        #200;
        latercy_cnt = 0;
        cnt_flag = 0;
        wait(checkbits == 16'h00A5);
        $display("latency-timer start counting");
        cnt_flag = 1;
        wait((checkbits & 16'h005A) == 16'h005A);
        $display("latency-timer : %d (cycles)" , latercy_cnt);
        cnt_flag = 0;
        #200;
        $finish;
end
```

圖(十四)

```
initial begin
        #20;
        for(iter = 1 ; iter <= N ; iter = iter + 1)begin
                acc = 0;
                for(tap_addr = 0 ; tap_addr < 11 ; tap_addr = tap_addr + 1)begin
                        data[tap_addr] = tap_addr == 10 ? iter : data[tap_addr + 1];
                end
                for(tap_addr = 0 ; tap_addr < 11 ; tap_addr = tap_addr + 1)begin
                        temp = data[tap_addr] * coef[10 - tap_addr];
                        acc = acc + temp;
                end
                golden[iter] = acc;
        end

end


initial begin
        wait(checkbits == 16'h00A5);
        @(posedge clock);
        for(ans_cnt = 1; ans_cnt < N-1; ans_cnt = ans_cnt + 1)begin
                wait((checkbits & ans_cnt) == ans_cnt);
                checkbits_cmp = checkbits;
                if(fir_out == golden[ans_cnt])
                        $display("%2dth correct : fir_out = %h expect = %h" , ans_cnt , fir_out ,golden[ans_cnt]);
                else
                        $display("%2dth error   : fir_out = %h expect = %h" , ans_cnt , fir_out ,golden[ans_cnt]);

                wait(checkbits_cmp != checkbits);
                @(posedge clock);
        end


        wait((checkbits & 16'h005A) == 16'h005A);
        if(fir_out == golden[ans_cnt])
                $display("%2dth correct : fir_out = %h expect = %h" , ans_cnt , fir_out ,golden[ans_cnt]);
        else
                $display("%2dth error   : fir_out = %h expect = %h" , ans_cnt , fir_out ,golden[ans_cnt]);
        wait(checkbits_cmp != checkbits);
        @(posedge clock);

end
```

圖(十五)

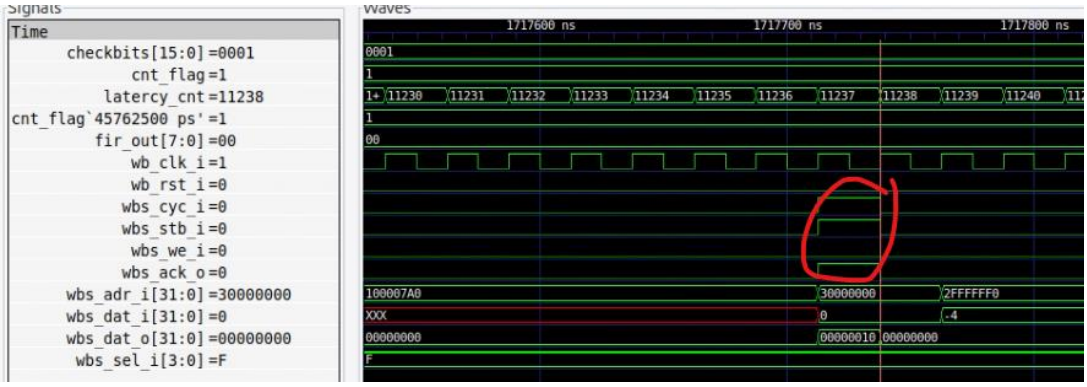
```
ubuntu@ubuntu2004:~/Lab_4_2/me_1110_ver1/lab-caravel_fir_
Reading counter_la_fir.hex
counter_la_fir.hex loaded into memory
Memory 5 bytes = 0x6f 0x00 0x00 0x0b 0x13
VCD info: dumpfile counter_la_fir.vcd opened for output.
Monitor:MPRJ-Logic WB Started
latency-timer start counting
 1th correct : fir_out = 00 expect = 00
 2th correct : fir_out = 01 expect = 01
 3th correct : fir_out = 00 expect = 00
 4th correct : fir_out = 02 expect = 02
 5th correct : fir_out = 00 expect = 00
 6th correct : fir_out = fb expect = fb
 7th correct : fir_out = fa expect = fa
 8th correct : fir_out = f6 expect = f6
 9th correct : fir_out = f4 expect = f4
10th correct : fir_out = f1 expect = f1
11th correct : fir_out = ee expect = ee
12th correct : fir_out = eb expect = eb
13th correct : fir_out = e8 expect = e8
14th correct : fir_out = e5 expect = e5
15th correct : fir_out = e2 expect = e2
16th correct : fir_out = df expect = df
17th correct : fir_out = dc expect = dc
18th correct : fir_out = d9 expect = d9
19th correct : fir_out = d6 expect = d6
20th correct : fir_out = d3 expect = d3
21th correct : fir_out = d0 expect = d0
22th correct : fir_out = cd expect = cd
23th correct : fir_out = ca expect = ca
24th correct : fir_out = c7 expect = c7
25th correct : fir_out = c4 expect = c4
26th correct : fir_out = c1 expect = c1
27th correct : fir_out = be expect = be
28th correct : fir_out = bb expect = bb
29th correct : fir_out = b8 expect = b8
30th correct : fir_out = b5 expect = b5
31th correct : fir_out = b2 expect = b2
32th correct : fir_out = af expect = af
33th correct : fir_out = ac expect = ac
34th correct : fir_out = a9 expect = a9
35th correct : fir_out = a6 expect = a6
36th correct : fir_out = a3 expect = a3
37th correct : fir_out = a0 expect = a0
38th correct : fir_out = 9d expect = 9d
39th correct : fir_out = 9a expect = 9a
40th correct : fir_out = 97 expect = 97
41th correct : fir_out = 94 expect = 94
42th correct : fir_out = 91 expect = 91
43th correct : fir_out = 8e expect = 8e
```

圖(十六)

```
44th correct : fir_out = 8b expect = 8b
45th correct : fir_out = 88 expect = 88
46th correct : fir_out = 85 expect = 85
47th correct : fir_out = 82 expect = 82
48th correct : fir_out = 7f expect = 7f
49th correct : fir_out = 7c expect = 7c
50th correct : fir_out = 79 expect = 79
51th correct : fir_out = 76 expect = 76
52th correct : fir_out = 73 expect = 73
53th correct : fir_out = 70 expect = 70
54th correct : fir_out = 6d expect = 6d
55th correct : fir_out = 6a expect = 6a
56th correct : fir_out = 67 expect = 67
57th correct : fir_out = 64 expect = 64
58th correct : fir_out = 61 expect = 61
59th correct : fir_out = 5e expect = 5e
60th correct : fir_out = 5b expect = 5b
61th correct : fir_out = 58 expect = 58
62th correct : fir_out = 55 expect = 55
latency-timer :      487919 (cycles)
63th correct : fir_out = 52 expect = 52
```

圖(十七)

下面的部分是抽出 FIR 的執行過程的波型以及解釋，圖(十八)是讀取 FIR 是否準備收取 data。



圖(十八)

接著是傳送 data 到 FIR 中的部分，如圖(十九)。

圖(十九)

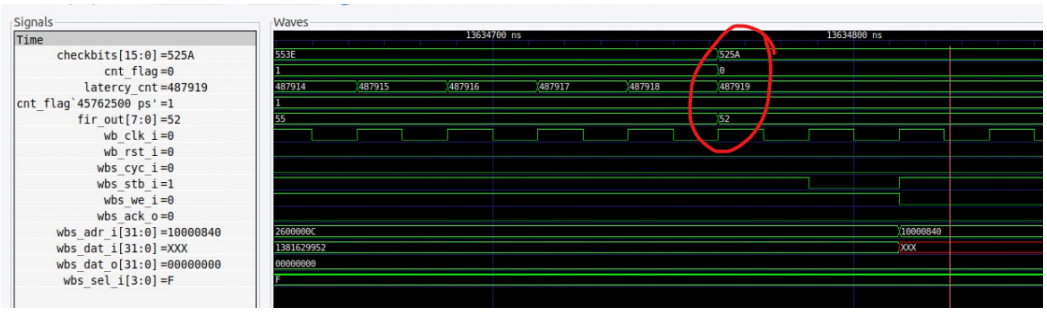
圖(二十)為收到 FIR 傳回 Yn 準備好的信號。

圖(二十)

然後收取 FIR 運算完的結果，如圖(二十一)。

圖(二十一)

最後一筆資料加上 0x005A 的結果如圖(二十二)。



圖(二十二)

圖(二十三)是將 FIR 的狀態回復至原先狀態。

圖(二十三)

圖(二十四)是最後的部分，firmware code 的部分，會先讀取到 done flag，最後到迴圈外面輸出 0xAB61 後結束模擬。

```
while((reg_fir_control & 0x00000002) == 0x00000002); // done flag assert

}
reg_mprj_datal = 0xAB610000;
```

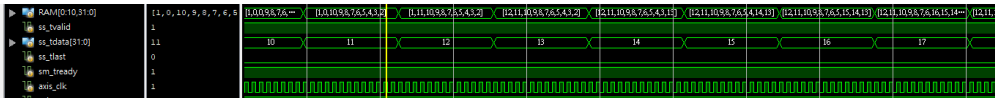
圖(二十四)

四、 The interface protocol between firmware, user project and testbench

Firmware 和 user project 和 Testbench 之間的溝通是透過 Wishbone bus，以及透過 Wishbone bus to AXI 的部分轉換成 FIR 看的懂的 AXI interface。在 FIR 計算開始前 firmware 會打出 0x00A5 到 mprj 的第 23 到第 16 隻讓 latency timer 開始計數，並根據 Configuration Register Address map 傳送適當的資料到 FIR，而 FIR 的輸出結果會被我們的 firmware 端存起來，而我們的 firmware 端會把 FIR 的資料結果放到 mprj 的第 31 到第 24 隻，而 mprj 的第 23 到第 16 隻 firmware 會傳出目前是第幾筆資料被送出，且在最後一筆資料時會送出 0x5A。藉由此操作就能夠讓 TB 端能夠核對送回來的答案。

五、 FIR engine theoretical throughput & Actually measured throughput

原先的 FIR Module 設計為收到值後立即展開運算，並在運算結束後，輸出結果，因此一筆資料的輸入-運算-輸出總共消耗 13 個 cycle 數，如圖(二十五)，但這屬於理論上的資料處理速度，當放上 SoC 時，因為 CPU 會執行 Firmware 要求執行的其它指令，而不是永遠卡在該處等待 FIR 的輸出結果，所以無法達成該理論上的速度。



圖(二十五)

由圖(二十六)可以看到輸入到輸出總共需要花費 27075ns，也就是 1083 個 cycle 才會結束，且因為 RISCV-CPU 在執行指令時，每條指令所需要的 cycle 數不同，才會需要等待中間 0x10000xxx 類型的指令，完成後才會再次輸出 0x380000084 Address 並允許 Y 值輸出。

圖(二十六)

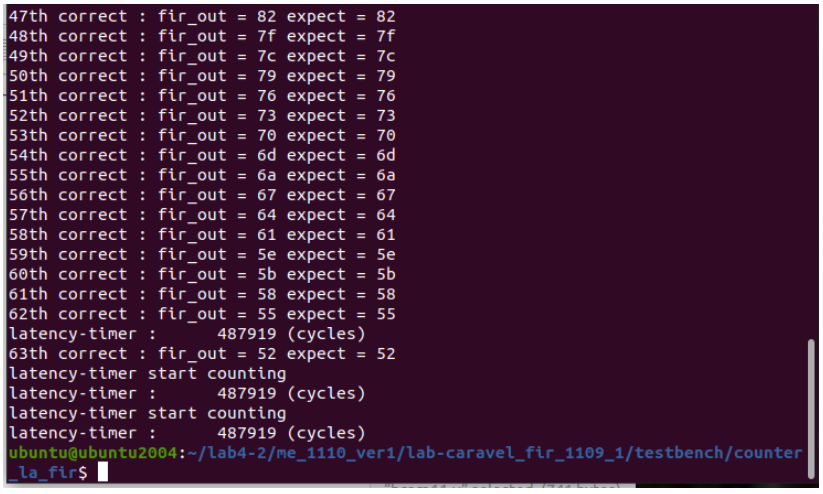
六、 Latency for firmware to feed data

Latency-timer : 487919 cycle

Gate-resource : LUT+FF = 12598

The Longest Path Delay : 5.584ns

Metrics = 34.32



圖(二十七)

七、 Techniques to improve throughput

- Does bram12 give better performance, in what way?

  可以，我們的 Data BRAM 的前 11 個位址用來儲存 Convolution 所需的資料，剩下的 1 個位址用來當作 Convolution 結果的 temp，如此以來算完 1 次 Convolution 後 Data BRAM 就可以馬上接下一筆資料，讓 Convolution 的運算可以壓在 11 個 cycle 內。

- Can you suggest other method to improve the performance?

  1. 平行化運算：

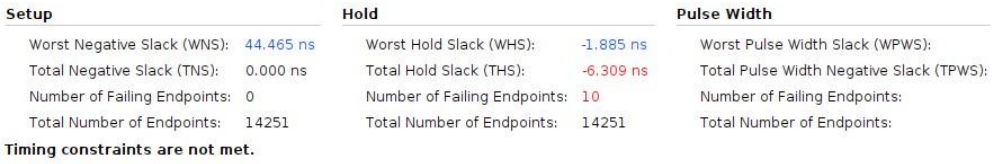
     對於多個 FIR 運算，考慮平行化處理。這可以通過將數據分成多個部分，分別進行 FIR 運算，然後合併結果。

  2. 流水線化：

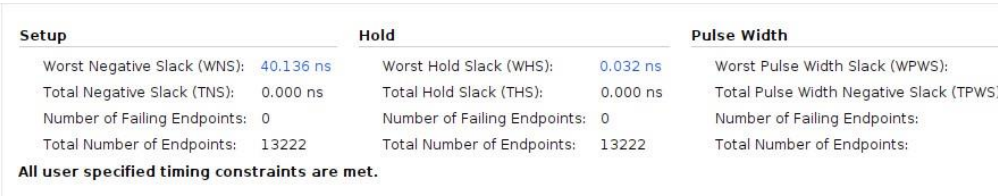
     在硬體和軟體層面，使用流水線化技術，將 FIR 運算劃分為多個階段，以實現連續的數據處理。這有助於提高效能並減少延遲。

八、 Observed & Learned

我們將整個 Cavarel SoC 使用 Vivado 進行 Synthesis 後發現我們的 Timing Report 中有 Hold time Violation，如圖(二八)，是因為時鐘繞的太遠，到達時間太晚。 而 Synthesis 之後給出的時序報告都是估計值，因此 Synthesis 之後可以不考慮 Hold Time，只考慮 Setup Time；即便此時 Hold Time Violation，我們也不需要去理會。在 Place Design 之後再去看 Hold Time，如果此時 Hold Time 的 Violation 比較小（比如-0.05ns），還是不需要理會的，因為工具在佈線時會修復 Hold，但如果 Slack 太大了，無法修復了，就會犧牲 setup 來彌補 hold。在圖(二九)中可以發現我們 implementation 後 Hold time Violation 被修掉了



圖(二八)



圖(二九)

九、 Resource Usage：

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Block RAM Tile (140)	DSPs (220)	Bonded IOPADs (130)
design_1_wrapper	5973	6625	169	47	7	3	130

圖(三十)

Resource	Utilization	Available	Utilization %
LUT	5973	53200	11.23
LUTRAM	351	17400	2.02
FF	6625	106400	6.23
BRAM	7	140	5.00
DSP	3	220	1.36

圖(三十一)

十、 Timing Report：

1. Design timing summary

Setup

Worst Negative Slack (WNS): 44.465 ns
Total Negative Slack (TNS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 14251

Hold

Worst Hold Slack (WHS): -1.885 ns
Total Hold Slack (THS): -6.309 ns
Number of Failing Endpoints: 10
Total Number of Endpoints: 14251

Pulse Width

Worst Pulse Width Slack (WPWS):
Total Pulse Width Negative Slack (TPWS):
Number of Failing Endpoints:
Total Number of Endpoints:

Timing constraints are not met.

圖(三十二)

2. Synthesize the design with maximum frequency

```
------------------------------------------------------------------
| Clock Summary
| -------------
------------------------------------------------------------------

Clock       Waveform(ns)         Period(ns)      Frequency(MHz)
-----       ------------         ----------      --------------
clk_fpga_0  {0.000 50.000}       100.000         10.000
```

圖(三十三)

3. Report timing on longest path, slack

```
Max Delay Paths
--------------------------------------------------------------------------------
Slack (MET) :             44.465ns  (required time - arrival time)
  Source:                 design_1_i/processing_system7_0/inst/PS7_i/FCLKCLK[0]
                            (clock source 'clk_fpga_0'  {rise@0.000ns fall@50.000ns period=100.000ns})
  Destination:            design_1_i/caravel_0/inst/housekeeping/wb_dat_o_reg[23]/D
                            (rising edge-triggered cell FDSE clocked by clk_fpga_0  {rise@0.000ns fall@50.000ns period=100.(
  Path Group:             clk_fpga_0
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            50.000ns  (clk_fpga_0 rise@100.000ns - clk_fpga_0 fall@50.000ns)
  Data Path Delay:        5.584ns  (logic 1.266ns (22.670%)  route 4.318ns (77.330%))
  Logic Levels:           8  (BUFG=1 LUT3=1 LUT5=1 LUT6=4 MUXF7=1)
  Clock Path Skew:        1.505ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    1.505ns = ( 101.505 - 100.000 )
    Source Clock Delay      (SCD):    0.000ns = ( 50.000 - 50.000 )
    Clock Pessimism Removal (CPR):    0.000ns
  Clock Uncertainty:      1.500ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    3.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

圖(三十四)