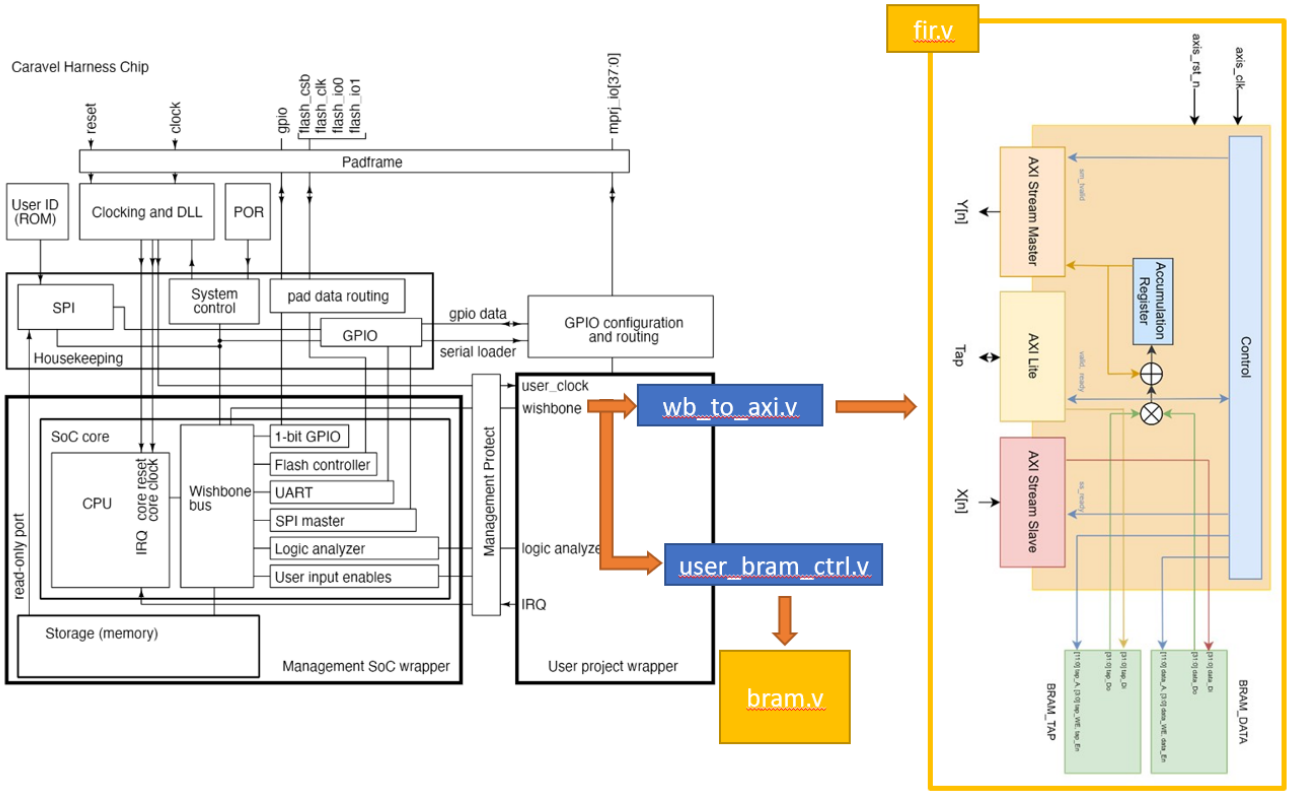
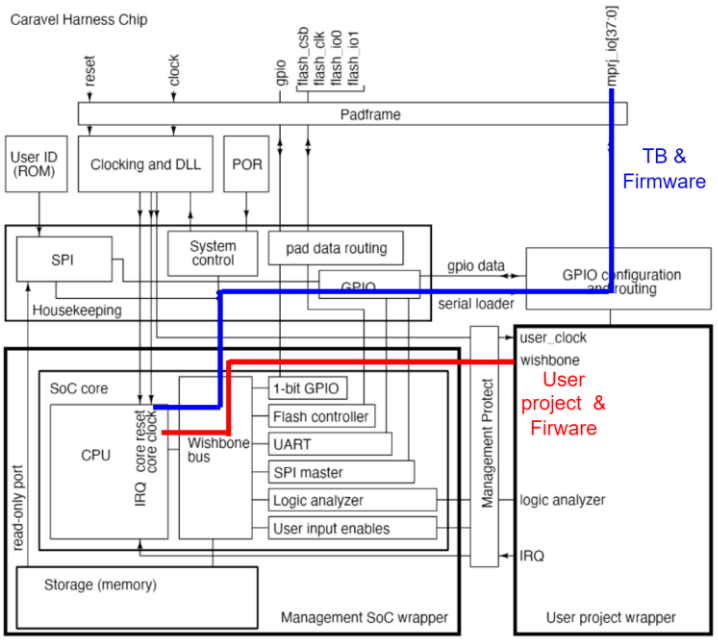
SOC Lab 4-2
Group 10

R11943022 電子所碩二 范詠為
R11943025 電子所碩二 謝郁楷
R11943124 電子所碩二 曾郁瑄

1. Design block diagram -datapath, control-path



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



firmware 與 user project:

firware 與 user project 之間是透過 wishbone 的 protocol 來連接，如上圖的紅色路徑。下面的 code 是 firware 中控制兩者傳輸的部分，reg_fir_x 與 reg_fir_y 分別是指到 0x3000_0080，與 0x3000_0084 的指標，因為這個 address 的區域是屬於 user project，所以會生成對應的讀或寫的 wishbone cycle 到 user project，而我們的 design 即可以抓到這些 cycle 並做出對應的 output 送到 0x3000_0084 並回應 firware 發到 0x3000_0084 這個位置的讀取 wishbone cycle。

```
int* __attribute__ ( ( section ( ".mprjram" ) ) ) fir(){
    initfir();
    for(int i = 0; i < L; i++)  {
        *reg_fir_x = i;                 // send x[n] to FIR
        outputsignal[i] = *reg_fir_y;   // receive y[n] from FIR
    }
```

firmware 與 testbench:

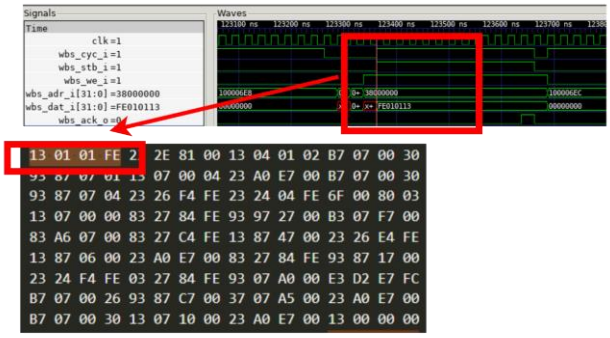
firware 是透過 wishbone 與 GPIO 連接，如上圖藍色路線所示。一開始 firware code 會 configure GPIO，讓他的[31:16]bits 為 management 的 output，之後 firmware 會寫透過寫入 0X2600_0000C (GPIO 31 to 0) 即可將 data 傳到 testbench。

3. Waveform and analysis of the hardware/software behavior.

Startup code:

系統一開始會執行左邊的 init code，這個 code 會產生 0x3800_0000 的 write cycle 到 wishbone 上面，user bram 即可以將 data 抓出來並存入。

```
sram_init:
  la a0, _fsram
  la a1, _esram
  la a2, _esram_rom
sram_loop:
  beq a0,a1,sram_done
  lw a3,0(a2)
  sw a3,0(a0)
  add a0,a0,4
  add a2,a2,4
  j sram_loop
sram_done:
```



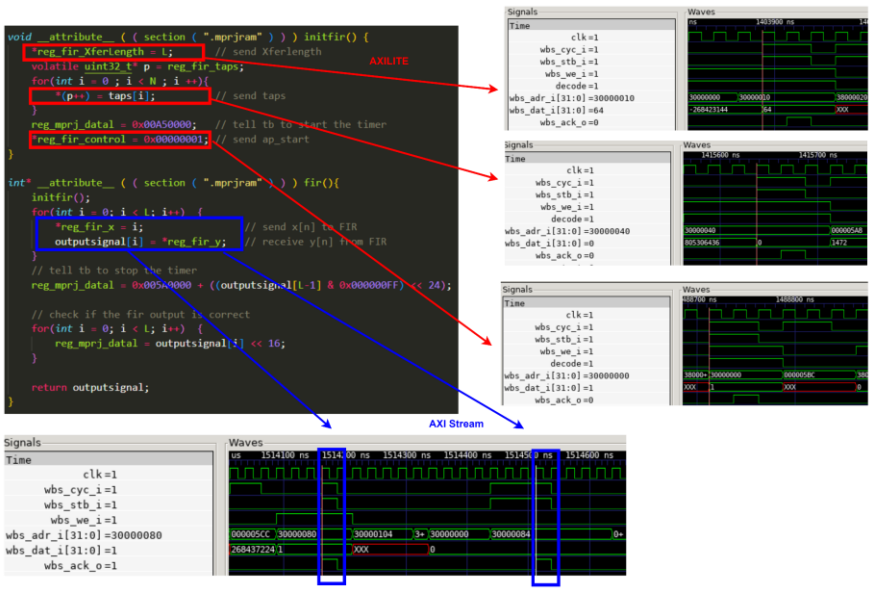
Fir calculation:

AXILite:

1. 將 data length 傳入。
2. 傳入 tap 的 value。
3. 傳入 ap_start。

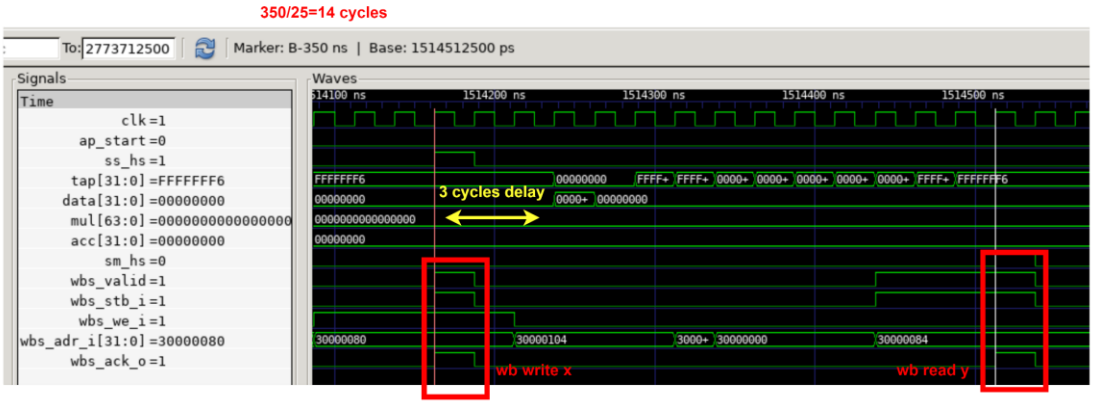
AXI Stream:

4. 交替的產生讀寫的 cycle 將 x 寫入 fir 並讀出 y。

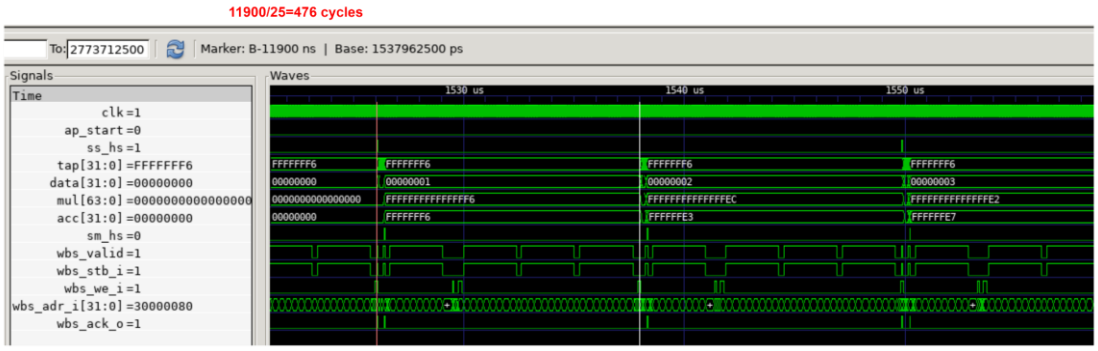


4. What is the FIR engine theoretical throughput, i.e. data rate? Actually measured throughput?

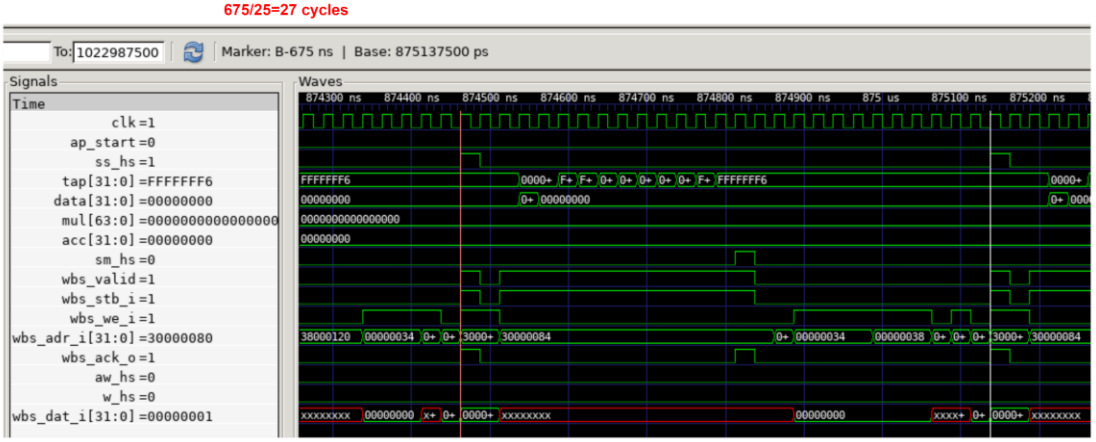
在 lab 3 中我們設計的 fir 在 fully pipelined 的情況下可以做到 11 個 cycles 一個 output。在 lab 4 中我們將 design 接上 wishbone，因為系統發出 wishbone cycle 到 user project 有一定的間隔，換句話說無法作到 fully pipelined 的設計，造成理論上的 through put 變為 1 output/ 14 cycles。波形圖如下所示，因為讀 data bram 有固定的 3 cycles 的 delay，原本可以 pipeline 來避免但現在不行。



實際上的 throughput 如下圖所示，為 476 cycles per output。

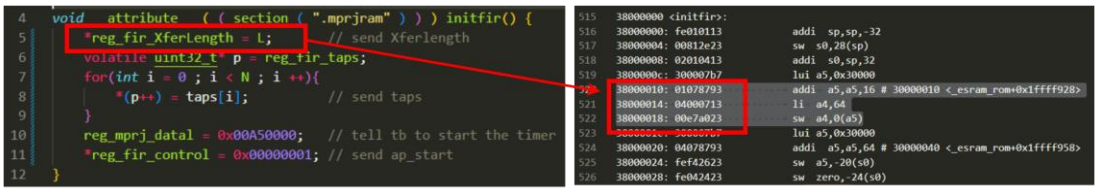


參考 report 後面優化的部分，我們最後做到 27 cycles per output



5. What is latency for firmware to feed data?

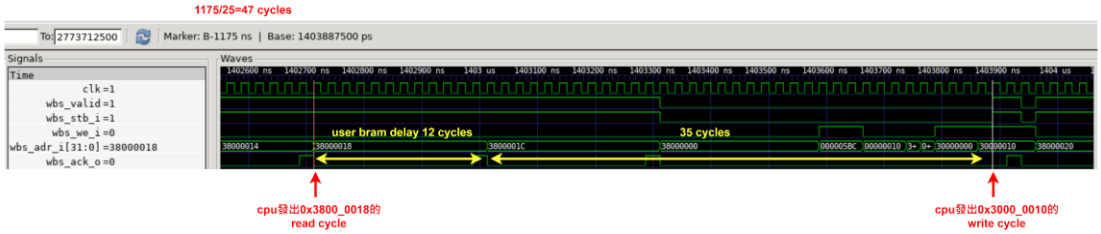
要測量這個 latency 我們要找到 cpu 讀取組語中對應的 sw 指令到 cpu 發出帶著資料的 wishbone cycle 間的 latency。我們以傳送 data length 這件事情為例，我們可以發現對應的 sw 是在位置 0x3800_0018 的地方。



而我們的 data length 則是定義在 0x3000_0010 的位置。

```
#define reg_fir_control      ((volatile uint32_t*) 0x30000000)
#define reg_fir_XferLength   ((volatile uint32_t*) 0x30000010)
#define reg_fir_taps         ((volatile uint32_t*) 0x30000040)
#define reg_fir_x            ((volatile uint32_t*) 0x30000080)
#define reg_fir_y            ((volatile uint32_t*) 0x30000084)
```

以下波型可以看到 cpu 先發出 read cycle，讀取在 0x3800_0018 這個位置的指令的，也就是讀 sw 那條指令，經過 user bram 的 12 個 cycle 的 delay cpu 收到，再經過 35 個 cycle 看到 cpu 送出到 0x3800_0010 的 write cycle。



所以 firmware feed data 的 latency 為 47 cycles。

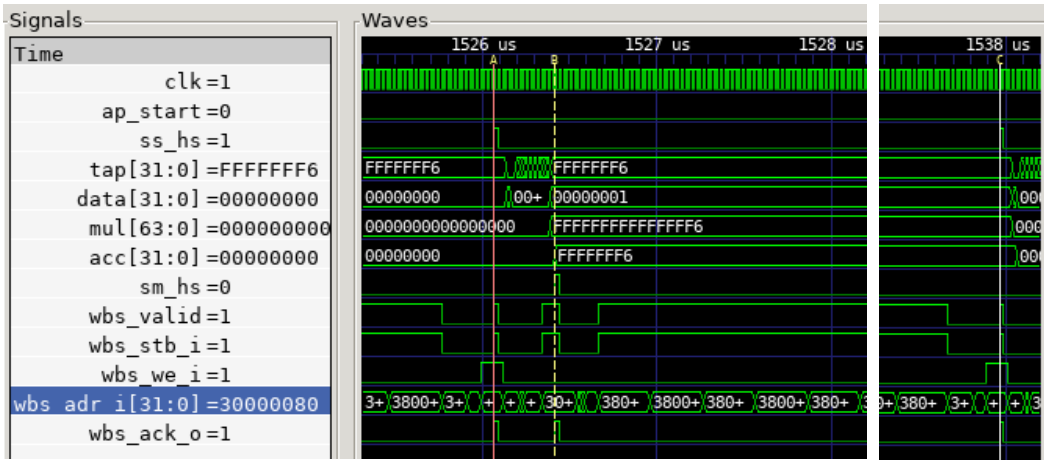
6. What techniques used to improve the throughput?

- Does bram12 give better performance, in what way?

我們在 Lab3 的設計沒有使用 bram12。

- Can you suggest other method to improve the performance?

分析整個系統的波型圖可以發現，**兩筆資料間需要 476 clock cycles(下圖 A 到 C，中間部分有省略)，而實際計算 FIR 的 cycle 數只有 14(下圖 A 到 B)，佔比不高**，其他時間看波型得知有許多 wishbone cycle，其 wbs_adr_i[31:0]大部分是 0x38 開頭，也就是在抓 firmware code。所以我們認為做 FIR 本身優化幫助並不大，考慮到整個系統才是最重要的。



我們想到**三個可以提升 throughput 的方式**。首先是整體的 clock period，在假設系統 clock 的 bottleneck 在 FIR 的前提下(畢竟我們只有針對 User Project 做合成)，想辦法優化 FIR 的 critical path 對整個系統會有非常大的幫助。我們將原先沒有切 pipeline 的版本做合成，clock cycle 最低為 16.451ns。而在**運算單元前後擋好 register 後，clock period 可以壓到 10.547ns**，前後如下面兩張圖。

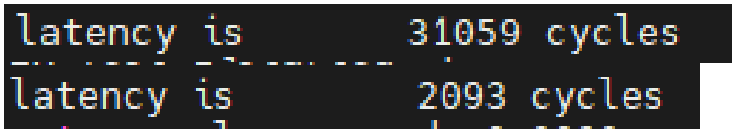
Name	Waveform	Period (ns)	Frequency (MHz)
wb_clk_i	{0.000 8.226}	16.451	60.787

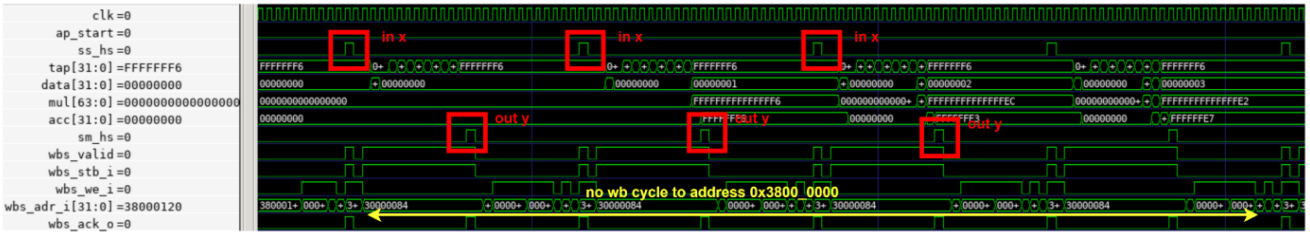
Name	Waveform	Period (ns)	Frequency (MHz)
wb_clk_i	{0.000 5.274}	10.547	94.814

接著，我們觀察到因為在做 FIR 運算時，CPU 是 IDLE 的，也就是要等到我們 14 個 cycles 算完回 wbs_ack_o，系統才會繼續讀 firmware code。於是我們想說**利用存在 data_Bram 的資料，做到 CPU 和 FIR 的平行**。做法是先算完前十筆已存在 data_Bram 的資料，再拉起 ss_ready 收最後一筆 data，運算完後即拉起 sm_valid 傳出結果，同時回 wbs_ack_o 讓 CPU 繼續執行下一筆 firmware code，這個間隔估計會從 3+11=14 cycles 減少到 3+1=4 cycles。但因為兩筆資料間

隔了許多 firmware code(476 cycles)，所以效果不會太大(僅省了 10/476=2.1%)。而這部分的優化要改動 fir.v，但因為時間的關係並沒有實作，且在最後一個優化方式之後因為 data 之間沒有 firmware code 的 wishbone cycle 的空檔(見 7. Any other insights?)，所以沒辦法做這個優化。

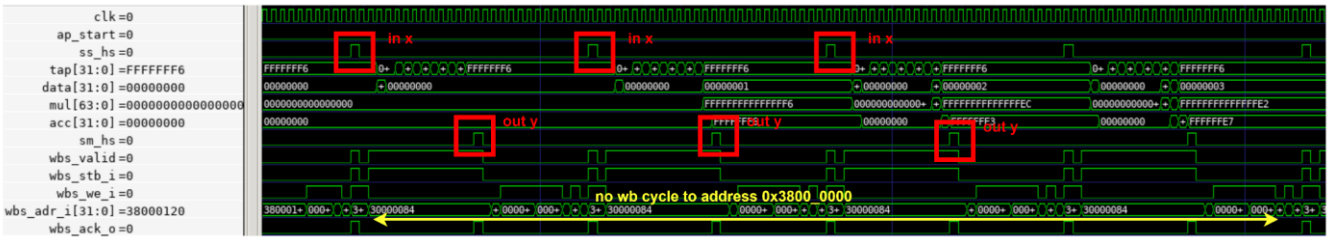
最後，也是優化最多的部分，我們針對開頭提到的，兩筆資料間佔多數時間的是 wishbone cycle 執行數條 firmware code，而非 FIR 運算。儘管我們的 fir.c 已經寫得很精簡，我們覺得展開 for loop 應該會有不錯的效果，我們嘗試在 **run_sim 中修改 riscv32-unknown-elf-gcc 的參數，加上-O3，讓 compiler 幫助我們做 assembly code 的簡化**。最後效果非常明顯兩筆 data 間 CPU 甚至沒有讀 0x38 firmware code 而是不斷接續的傳資料，**從 476 個 cycles 降低到 27 個 cycles，總 cycle 數從 31059 降低到 2093，如圖**。





7. Any other insights ?

我們開-O3 後看波型圖遇到一個問題，在兩筆資料間 CPU 甚至沒有去拿 firmware code，可以看到下圖沒有 0x38 開頭的 wbs_adr_i。我們原本想說至少會有一個 12 cycle delay 的時間去拿 firmware code 但看起來是沒有。



於是我們去檢查生出來的 assembly code，發現的確有相較前一版更簡潔，也有 bne 指令判斷 for 迴圈的跳出條件(data 傳 64 個之後會跳出 for loop)，但在波型上只有第一筆資料有執行 114->118->11c->120->128，之後直接離開 for loop 去到 12c，直到所有 data 算完才繼續拿剩下的 firmware code(0x38)，我們不太清楚為什麼第二筆開始 CPU 不用去 0x38 讀 114 而可以直接發 0x30000080 去送資料，推測是第一次抓 code 後將他存到其他地方像是 cache。

```
38000114: 08e62023          sw  a4,128(a2) # 30000080 <_esram_rom+0x1ffffd48>
38000118: 08462583          lw  a1,132(a2)
3800011c: 00170713          addi  a4,a4,1
38000120: 00468693          addi  a3,a3,4 # a50004 <_fstack+0xa4fa04>
38000124: feb6ae23          sw  a1,-4(a3)
38000128: ff0716e3          bne a4,a6,38000114 <fir+0x94>
```

8. Final Score

Metrics : #-of-clock (latency-timer) * clock_period * gate-resource

- #-of-clock: 2093
- clock_period: 10.547 ns
- gate-resource (LUT+FF): 374+396=770

Summary

Resource	Utilization	Available	Utilization %
LUT	374	53200	0.70
LUTRAM	65	17400	0.37
FF	396	106400	0.37
BRAM	16	140	11.43
DSP	3	220	1.36
IO	297	125	237.60

- Score = 2093*10.547*1e-9*770=16997650.67*1e-9=0.01699765067