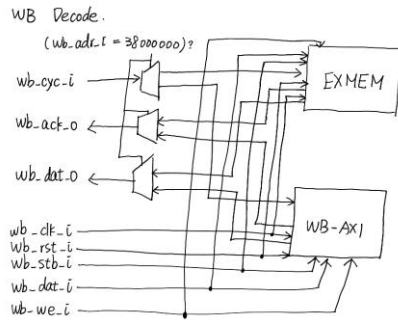


# SoC Lab4-2

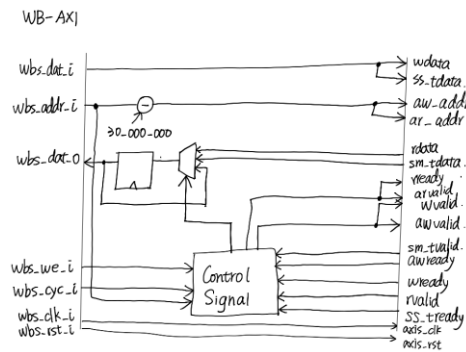
一、Design block diagram – datapath, control-path

## Datapath

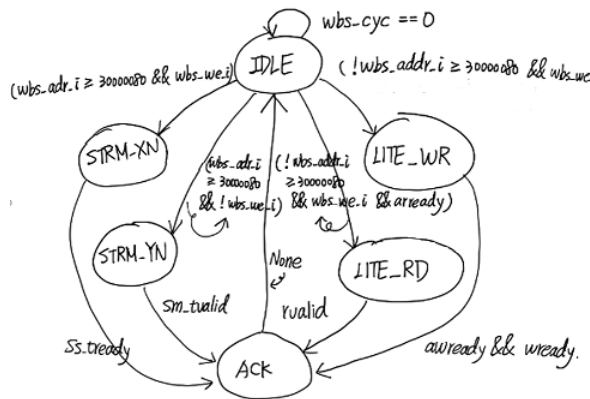
### WB Decoder



### WB-AXI



## Control path(WB-AXI)



在 WB-AXI 中，一共定義了六個狀態。

IDLE: 在這個狀態中，會去看 WB 的 ADDR、WE 來判斷是下個 cycle 要進入哪個狀態。

LITE\_WR: 由於 FIR 的設計是 AW、W 是各自獨立的通道且深度只有 1，因此如果兩個 ready 皆為 1 的話，就代表 AW、W 都有成功寫入，返回可接收的狀態。

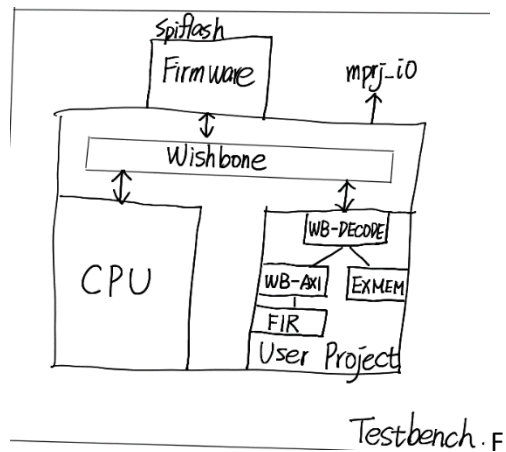
LITE\_RD: 只要判定 rvalid 是否為 1，就能知道是否能將 rdata 傳送回 WB。

STRM\_XN: 當 ss\_tready=1 時，此時就能將 wbs\_dat\_i 傳送到 AXI 中。

STRM\_YN: 只要判定 sm\_tvalid 是否為 1，就能知道是否能將 Y[n]傳送回 WB。

ACK: 當上述的狀態在完成動作時，就會统一到 ACK 狀態，返回 ACK 到 WB。

## 二、The interface protocol between firmware, user project and testbench



Firmware 會透過 WB 跟 CPU 做溝通，而當 CPU 獲取到指令後，就會將它寫入 EXMEM。

此時會先透過 WB BUS 傳送到 User Project 中，之後 WB Decoder 會將它轉傳到 EXMEM。

而當指令中需要使用到 FIR 硬體時，就會先透過 WB BUS 送到 User Project，之後 WB-AXI 會先將它轉成 AXI，再送到 FIR。

而在 Firmware 中，Caravel 會將運算結果傳送到 mprj\_io，此時 Testbench 就能從 mprj\_io 接收運算結果。

### 三、Waveform and analysis of the hardware/software behavior.

## Software:

```
void __attribute__ ( ( section ( ".mprjram" ) ) ) initfir() {  
    // initial your fir  
    int i;  
    uint32_t Mask, Status;  
  
    // send data length  
    reg_fir_data_len = fir_test_len;  
  
    // send taps  
    for(i=0; i<N; i++){  
        *(reg_fir_coeff+i) = taps[i];  
    }  
  
    // check ap_idle = 1  
    // fir_control[2] = ap_idle -> Mask  
    Mask = 0;  
    Mask |= (1 << 2);  
  
    Status = reg_fir_control & Mask;  
    while(Status != 4){  
        Status = reg_fir_control & Mask;  
    }  
  
    // send ap_start  
    // set fir_control[0] = 1 -> ap_start = 1  
    reg_fir_control = 1;  
}
```

在 Init 中，首先會先去傳送這次要測驗的長度(Data length)。

之後就會開始傳送 Taps，而因為 Taps 數量有點多，因此是採用 pointer 的方式去賦值，相對方便許多。

當參數都傳送完畢後，就會去檢查 ap\_idle 是否為 1，因為怕其他位元會去干擾到判斷，因此在判斷之前會去使用 Mask，將該位元單獨抓出來做判斷。

當 ap\_idle 為 1 時，就會跳出 while，傳送 ap\_start(reg\_fir\_control = 1)。

```
int* __attribute__ ( ( section ( ".mprjram" ) ) ) fir(){  
    initfir();  
    //write down your fir  
    int i;  
    uint32_t Mask, Status;  
  
    for(i=0; i<fir_test_len; i++){  
        // check X[n] = 1 is ready to accept input.  
        Status = reg_fir_control & Mask;  
        while(Status != 16){  
            Status = reg_fir_control & Mask;  
        }  
  
        // send X[n]  
        reg_fir_x = i+1;  
  
        // check when Y[n] is ready  
        Status = reg_fir_control & Mask;  
        while(Status != 32){  
            Status = reg_fir_control & Mask;  
        }  
  
        // receive Y[n]  
        outputsignal[i] = reg_fir_y;  
    }  
  
    // Read ap_done to set ap_idle  
    Status = reg_fir_control;  
  
    return outputsignal;  
}
```

在開始執行 fir 之前，會先去呼叫 Init()，初始化 FIR 硬體。

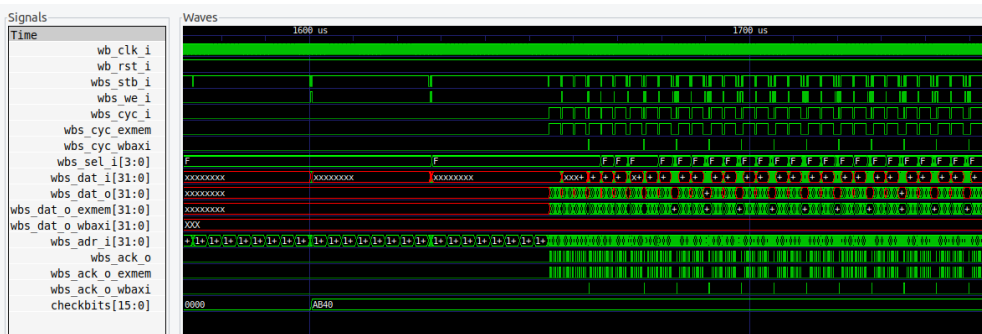
接下來，就可以開始傳送測資。

首先，會先去檢查 X[n] 是否處於 ready 的狀態，當檢查條件成立後，才會去執行傳送 X[n]。

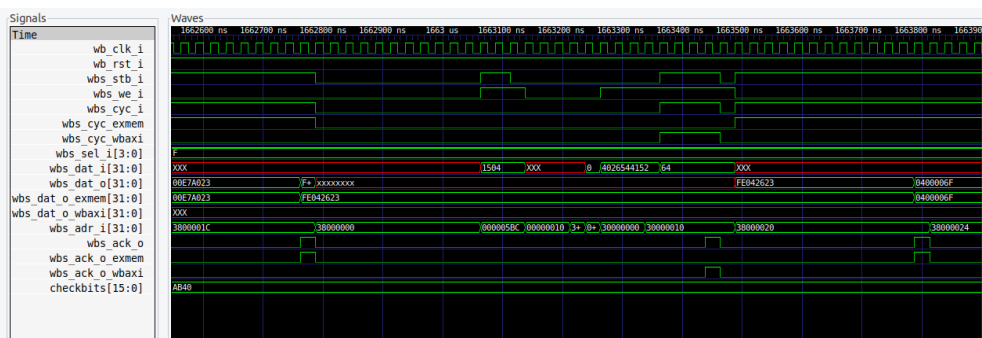
之後，就會則是去檢查 Y[n] 是否已經運算完畢，當檢查條件成立後，才會去執行接收 Y[n]。

最後，執行完所有測資後，就會去讀取 ap\_done，這樣 ap\_idle 就會被拉為 1，進入到 IDLE 狀態，接著就會返回一個指標，讓 counter\_la\_example.c 能夠將結果丟到 mprj\_io 上，之後再由 Testbench 去檢查結果是否正確。

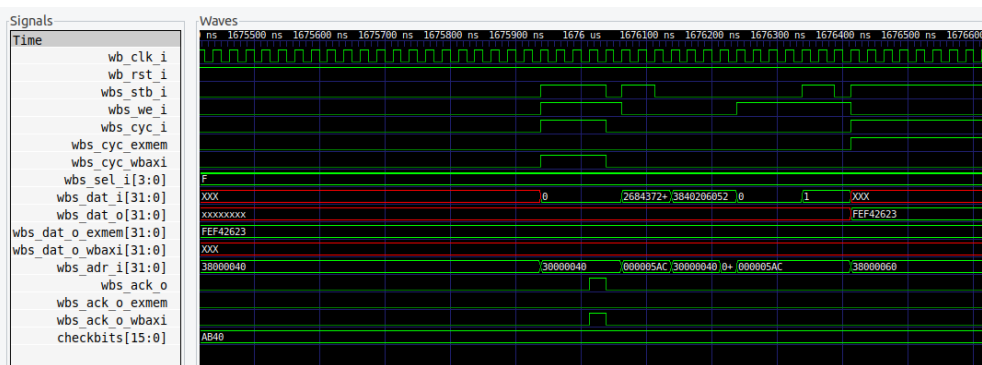
## Hardware:



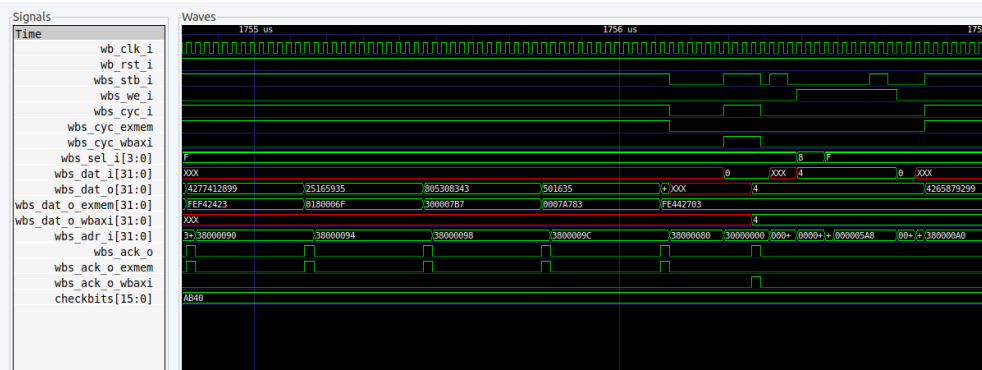
當 Checkbits 接收到 AB40 時，就代表 FIR 運算開始。



接下來，會去送測試長度(64)到 0x30000010，表示這次的測試共有 64 筆測資。

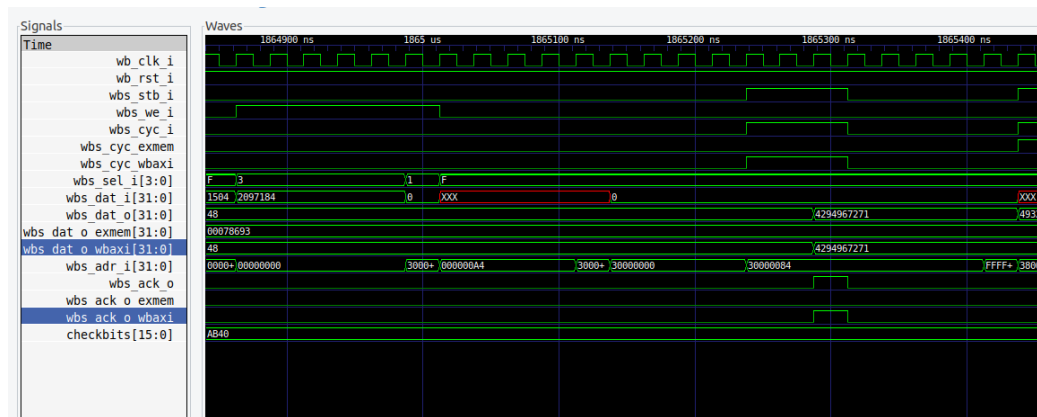


送完測試長度後，就要開始傳送 Taps(0x30000040)。

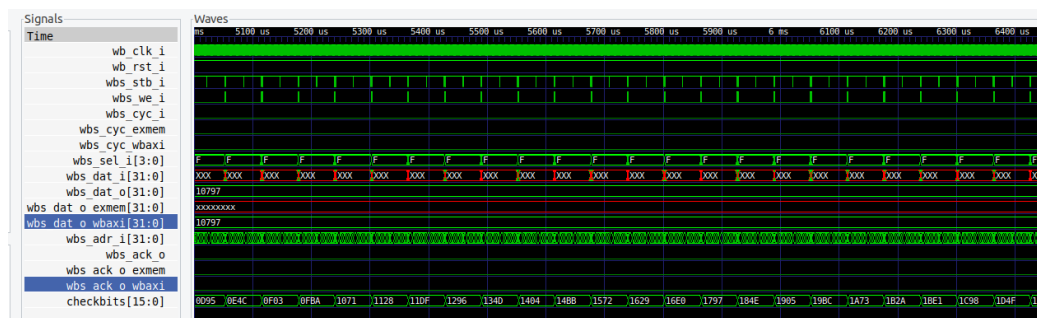


之後就要去檢查 `ap_idle(0x3000000 [2])` 是否為 1，如果為 1 就能開始 FIR 運算。

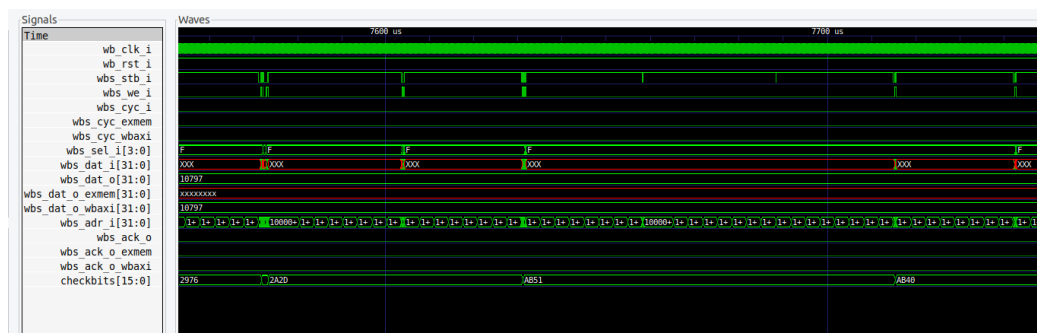




檢查完畢後，就能傳送 Y[n] (0x3000084)到 FIR。



當測資都傳送完畢後，接下來就要開始丟到 Checkbits 去與 Testbench 做溝通，並檢查運算的結果是否正確。

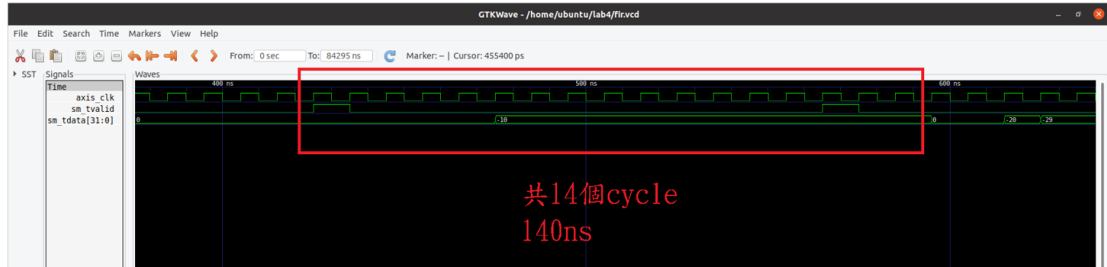


當結果都檢查完畢後，最後就會傳送 AB51，代表結束這一輪的測試。

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

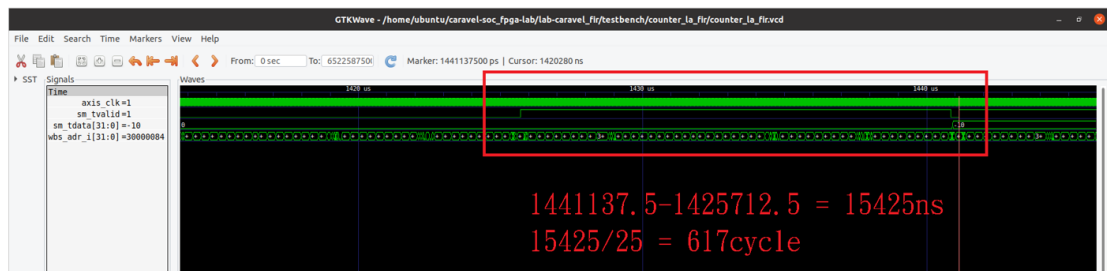
**Theoretical throughput: 7.14M output / sec**

**Theoretical data rate: 7.14M output per second**



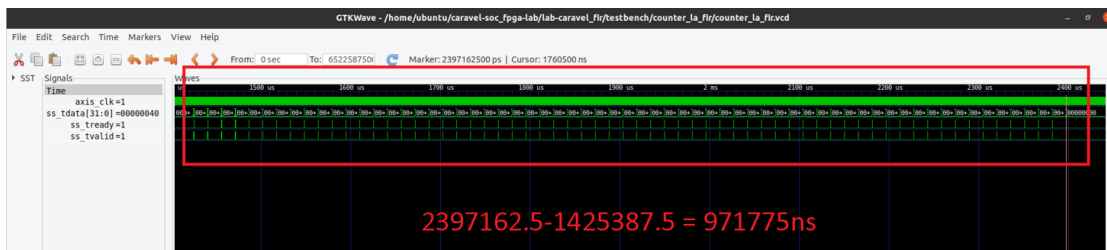
FIR 運算的時間 從輸入  $X[n]$  開始算起，大約需要 14 個 cycle 就能完成  $Y[n]$  的計算。

**Actually measured throughput: 64.8K output / sec**



實際上測量到的 Throughput 是大約 64.8K output / sec，這部分與 Theoretical 有著很大的差異，而之所以造成 Theoretical 跟 Actually 會有這麼大的差異，從觀察到的結果來看，很大的原因是因為 CPU 在丟入  $X[n]$  之後便會去執行其他指令，而 FIR 其實已經早就算好  $Y[n]$  等待著 CPU 接收，結果 CPU 過了很長的時間才去執行接收  $Y[n]$  的指令，最後就導致它的 Throughput 變得非常低。

五、What is latency for firmware to feed data?



CPU 從第一筆  $X[n]$  送到最後一筆  $X[n]$  的 Latency 是 971775ns(38817cycles)。  
而每筆  $X[n]$  的 Latency 則是 15425ns(617cycles)。

## 六、What techniques used to improve the throughput?

從波形上看，總體 Throughput 會去被 CPU 主導，是因為在執行的過程中，CPU 會再去執行其他程式，而 CPU 在分配給 Firmware 的時間 window 並不長，因此 Firmware Code 如果能在越短的程式內完成，就能加快得到運算結果的時間，進而增加 Throughput。

### 1. 取消 Check:

無 Check : 5101275ns

VS

有 Check:5970850ns

```
ubuntu@ubuntu2004:~/caravel_soc_fpga-lab$ cd /caravel_soc_fpga-lab/caravel_fir/testbench/counter_la_fir5
source run_sir
counter_la_fir5 has loaded into memory
Memory 1 bytes = 1024 KiB
INFO: dump/la_counter_la_fir5.vcd opened for output.
la_test
1: start
la_test
1: passed
latency = 5101275
-----
```

```
ubuntu@ubuntu2004:~/SoC/caravel_soc_fpga-lab-main$
PASS: LOOP = 2, TEST_IDX = 49, DATA = 8235
PASS: LOOP = 2, TEST_IDX = 50, DATA = 8488
PASS: LOOP = 2, TEST_IDX = 51, DATA = 8661
PASS: LOOP = 2, TEST_IDX = 52, DATA = 8784
PASS: LOOP = 2, TEST_IDX = 53, DATA = 8967
PASS: LOOP = 2, TEST_IDX = 54, DATA = 9150
PASS: LOOP = 2, TEST_IDX = 55, DATA = 9333
PASS: LOOP = 2, TEST_IDX = 56, DATA = 9516
PASS: LOOP = 2, TEST_IDX = 57, DATA = 9699
PASS: LOOP = 2, TEST_IDX = 58, DATA = 9882
PASS: LOOP = 2, TEST_IDX = 59, DATA = 10065
PASS: LOOP = 2, TEST_IDX = 60, DATA = 10248
PASS: LOOP = 2, TEST_IDX = 61, DATA = 10431
PASS: LOOP = 2, TEST_IDX = 62, DATA = 10614
PASS: LOOP = 2, TEST_IDX = 63, DATA = 10797
-----
-----Congratulations! Pass-----
-----
-----LOOP 0 Latency = 5970850-----
```

在傳送 X[n]、Y[n]前，如果先去檢查 X[n] ready(0x3000000 [4])、Y[n] valid(0x3000000 [5])的話，CPU 勢必就需要多花一些指令去做檢查。

然而我們從波型上看的話，其實在 FIR 運算的過程中，CPU 是會跑去執行其他程式的指令，而那個間隔是非常久的，久到超過 FIR 的運算時間，因此在 CPU 傳送 X[N]、接收 Y[N]之前，有沒有去檢查 ready、valid 都不會影響運算的結果。

於是我們就拿有無 Check 來做比較，左邊是都不去檢查 直接傳送 X[N]、接收 Y[N]，而右邊則是在每次傳送前，都會去檢查 FIR 是否處於可傳送狀態。

結果顯示，無 Check 的總體 Latency 有明顯較有 Check 的 Latency 低很多，因此在這個 case 中，如果想要 Throughput 增加，可以選擇不要去檢查 X[n] ready(0x3000000 [4])、Y[n] valid(0x3000000 [5])。

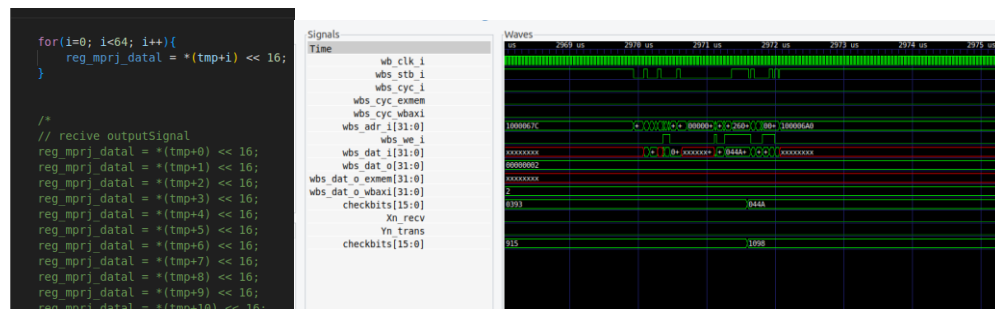


## 2. 將 for loop 展開:

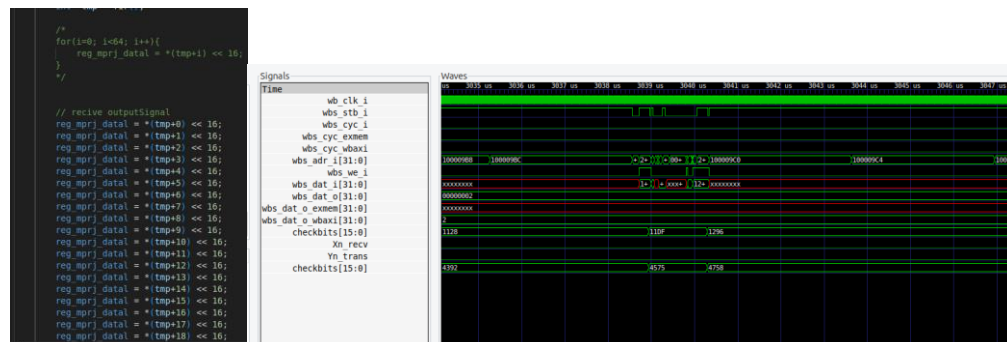
如前面所說，總體的 Throughput 是由於 CPU 在執行指令時，會再去服務其他程式的關係，因而導致總體 Throughput 變少。

於是我去觀察了波形，結果發現到每次 CPU 所分配給 FIR Firmware 的時間 window，大約能夠執行兩行的程式碼，如下圖所示

\*\*採用 for loop 去傳 reg\_mprj\_data1



\*\*將 for loop 展開 去傳 reg\_mprj\_data1



使用 for 傳送 reg\_mprj\_data1

VS

將 for 展開傳送 reg\_mprj\_data1

```
ubuntu@ubuntu2004: ~/SoC/caravel-soc_fpga-lab-main/lab-caravel
PASS: LOOP = 2, TEST_IDX = 49, DATA = 8235
PASS: LOOP = 2, TEST_IDX = 50, DATA = 8418
PASS: LOOP = 2, TEST_IDX = 51, DATA = 8601
PASS: LOOP = 2, TEST_IDX = 52, DATA = 8784
PASS: LOOP = 2, TEST_IDX = 53, DATA = 8967
PASS: LOOP = 2, TEST_IDX = 54, DATA = 9150
PASS: LOOP = 2, TEST_IDX = 55, DATA = 9333
PASS: LOOP = 2, TEST_IDX = 56, DATA = 9516
PASS: LOOP = 2, TEST_IDX = 57, DATA = 9699
PASS: LOOP = 2, TEST_IDX = 58, DATA = 9882
PASS: LOOP = 2, TEST_IDX = 59, DATA = 10065
PASS: LOOP = 2, TEST_IDX = 60, DATA = 10248
PASS: LOOP = 2, TEST_IDX = 61, DATA = 10431
PASS: LOOP = 2, TEST_IDX = 62, DATA = 10614
PASS: LOOP = 2, TEST_IDX = 63, DATA = 10797
-----Congratulations! Pass-----
-----LOOP 0 Latency = 4829350-----
-----LOOP 1 Latency = 4825725-----
-----LOOP 2 Latency = 4825725-----
ubuntu@ubuntu2004: ~/SoC/caravel-soc_fpga-lab-main/lab-caravel_fir/testben
```

總體 latency: 4825725ns

```
ubuntu@ubuntu2004: ~/SoC/caravel-soc_fpga-lab-main/lab-caravel
PASS: LOOP = 2, TEST_IDX = 49, DATA = 8235
PASS: LOOP = 2, TEST_IDX = 50, DATA = 8418
PASS: LOOP = 2, TEST_IDX = 51, DATA = 8601
PASS: LOOP = 2, TEST_IDX = 52, DATA = 8784
PASS: LOOP = 2, TEST_IDX = 53, DATA = 8967
PASS: LOOP = 2, TEST_IDX = 54, DATA = 9150
PASS: LOOP = 2, TEST_IDX = 55, DATA = 9333
PASS: LOOP = 2, TEST_IDX = 56, DATA = 9516
PASS: LOOP = 2, TEST_IDX = 57, DATA = 9699
PASS: LOOP = 2, TEST_IDX = 58, DATA = 9882
PASS: LOOP = 2, TEST_IDX = 59, DATA = 10065
PASS: LOOP = 2, TEST_IDX = 60, DATA = 10248
PASS: LOOP = 2, TEST_IDX = 61, DATA = 10431
PASS: LOOP = 2, TEST_IDX = 62, DATA = 10614
PASS: LOOP = 2, TEST_IDX = 63, DATA = 10797
-----Congratulations! Pass-----
-----LOOP 0 Latency = 2344850-----
-----LOOP 1 Latency = 2344850-----
-----LOOP 2 Latency = 2344850-----
ubuntu@ubuntu2004: ~/SoC/caravel-soc_fpga-lab-main/lab-caravel_fir/testben
```

總體 latency: 2344850ns

結果發現，使用 for loop 迴圈的 firmware，在每個分配的時間 window 內就只能傳出一筆的 reg\_mprj\_data1，這是因為一個用於執行 for 的判斷，另一個則是用於執行傳送 reg\_mprj\_data1。

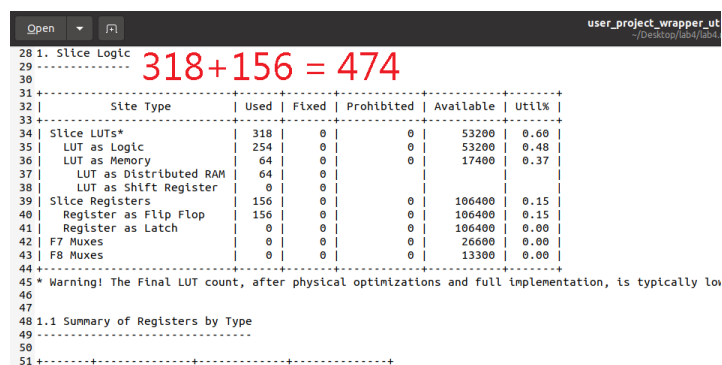
而將 for loop 展開的 firmware，由於它不需要去判斷 for 的關係，因此在每個分配的時間 window 內去執行兩行程式碼，就能傳送兩筆的 reg\_mprj\_data1。

對於 latency 而言，將 for 展開來的 firmware 也比使用 for firmware 快上不少。

總結來說，Firmware 所執行的程式碼數量相當重要，這會大幅影響到得到 FIR 運算結果的時間，因此對於 for 迴圈、if-else 之類的判斷式，如非必要就盡量不要使用，這樣才能盡可能的去提升 Throughput。

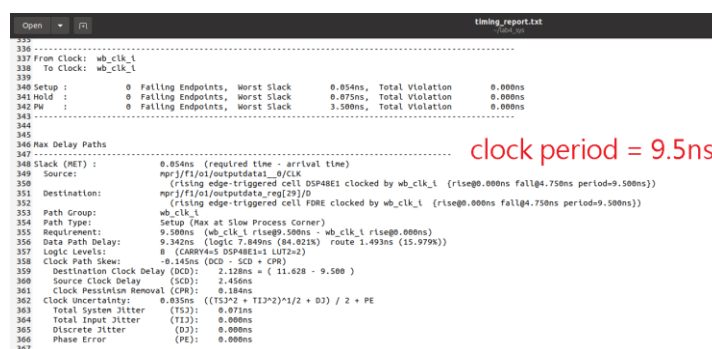
## 七、Resource

LUT+FF:



Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	318	0	0	53200	0.60
LUT as Logic	254	0	0	53200	0.48
LUT as Memory	64	0	0	17400	0.37
LUT as Distributed RAM	0	0	0	0	0.00
LUT as Shift Register	0	0	0	0	0.00
Slice Registers	156	0	0	106400	0.15
Register as Flip Flop	156	0	0	106400	0.15
Register as Latch	0	0	0	106400	0.00
F7 Muxes	0	0	0	26600	0.00
F8 Muxes	0	0	0	13300	0.00

clock period:



337 From Clock: wb_clk_i	
338 To Clock: wb_clk_i	
339	
340 Setup :	0 Falling Endpoints, Worst Slack 0.854ns, Total Violation 0.000ns
341 Hold :	0 Falling Endpoints, Worst Slack 0.875ns, Total Violation 0.000ns
342 PW :	0 Falling Endpoints, Worst Slack 3.380ns, Total Violation 0.000ns
343	
344	
345	
346 Max Delay Paths	
347	
348 Slack (NET) :	0.054ns (required time - arrival time)
349 Source:	mpj/fi/oi/outputdata1_0/clk (rising edge-triggered cell DSP48E1 clocked by wb_clk_i {rise@0.000ns fall@4.750ns period=9.500ns})
350	
351 Destination:	mpj/fi/oi/outputdata_reg[29]/0 (rising edge-triggered cell FDR clocked by wb_clk_i {rise@0.000ns fall@4.750ns period=9.500ns})
352	
353 Path Group:	Setup (Max at Slow Process Corner)
354 Path Type:	9.500ns (wb_clk_i rise@0.500ns - wb_clk_i rise@0.000ns)
355 Requirement:	9.342ns (Logic 7.849ns (84.821%) route 1.493ns (15.979%))
356 Data Path Delay:	0 (CARVEOUT DSP48E1s LUTs=2)
357 Logic Levels:	-0.145ns (DCD - SCD + CPR)
358 Clock Path Skew:	2.128ns = ( 11.628 - 9.500 )
359 Destination Clock Delay (DCD):	2.128ns
360 Source Clock Delay (SCD):	0.184ns
361 Clock Pessimism Removal (CPR):	0.184ns
362 Clock Uncertainty:	0.893ns ((T13*2 + T13*2)*1/2 + DJ) / 2 + PE
363 Total System Jitter (T13):	0.872ns
364 Total Input Jitter (T13):	0.000ns
365 Discrete Jitter (DJ):	0.000ns
366 Phase Error (PE):	0.000ns
367	

## 八、Github Link

<https://github.com/PatriChou/lab-caravel-fir.git>