Using HLS IP in Zynq Soc Design

Lab #A UG871

107012045 郭柏辰 2022 / 3 / 18

一、Lab 1 ----- MACC design

1. Introduction

```
#include "hls_macc.h"

void hls_macc(int a, int b, int *accum, bool accum_clr)
{
#pragma HLS INTERFACE s_axilite port=return bundle=HLS_MACC_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=a bundle=HLS_MACC_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=b bundle=HLS_MACC_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=accum bundle=HLS_MACC_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=accum_clr bundle=HLS_MACC_PERIPH_BUS

   static acc_reg = 0;
   if (accum_clr)
      acc_reg = 0;
   acc_reg += a * b;
   *accum = acc_reg;
}
```

Figure. hls_macc.c

本實驗所要完成之演算法為乘法累加器。共有四個 IO，包括，32 bits int A，32 bits int B，32 bits int accum，1 bit bool accum_clr。其中”a”和“b”為輸入要相乘的兩筆整數，”accum”為輸出累加後結果，”accum_clr”為選擇是否要先清空原本累加之內容。

另外其輸出接口為 AXI-Lite，並且將其 bundle 在 ”HLS_MACC_PERIPH_BUS”。

2. Design flow

We use vivado design flow to implement design. 首先透過 HLS 完成 kernel IP 的製作，第二步透過 vivado 的 Block design 完成可執行之 bitstream 和 handoff file。最後透過 Vitis IDE 完成 software 配置與 PL(programmable logic)端的控制，此步驟也可以使用 Overlay 進行代替。

a. HLS

建立一個 HLS project，並配置 pynq-z2 作為本次實驗用板子。

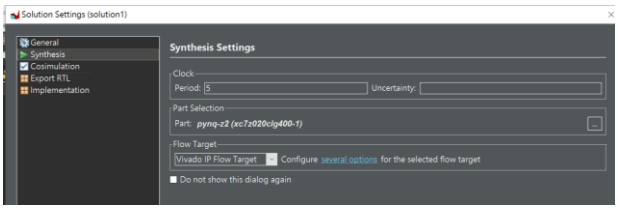

Figure. Set up parts

進入 Project 後打開 Explorer 匯入 source 與 testbench

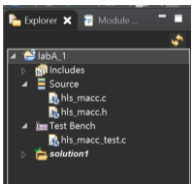

Figure. Add source and testbench

打開 project setting/synthesis，設定 Top function 要與合成目標一致。

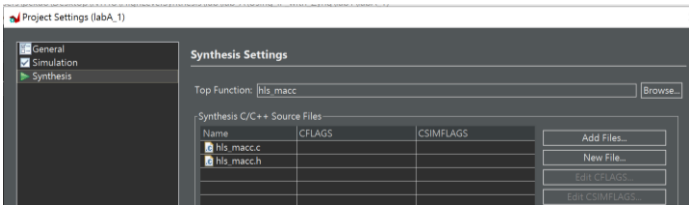

Figure. Set top function

配置好後，進行 C-simulation，確認 software behavior。

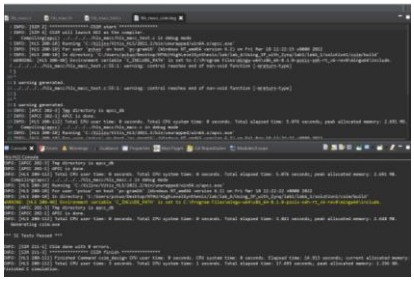

Figure. Run c-simulation

接著進行 Synthesis。

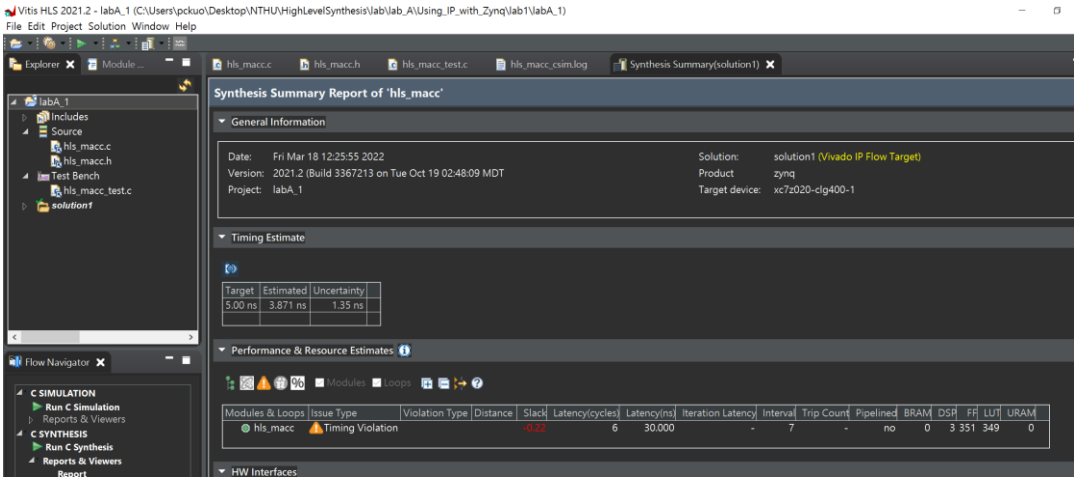

Figure. Synthesis summary

確認無誤之後，進行 Co-simulation，確認 RTL 模型與 C code 行為相符合。

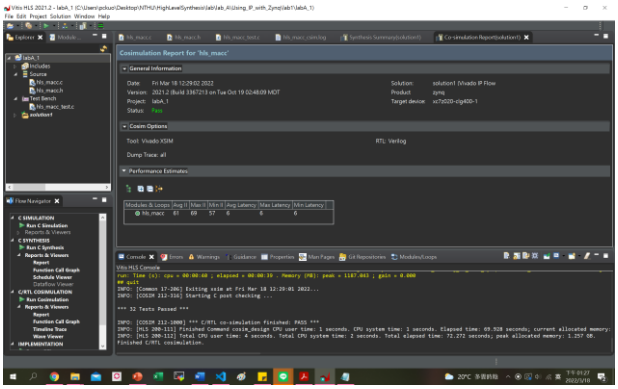

Figure. Co-sim

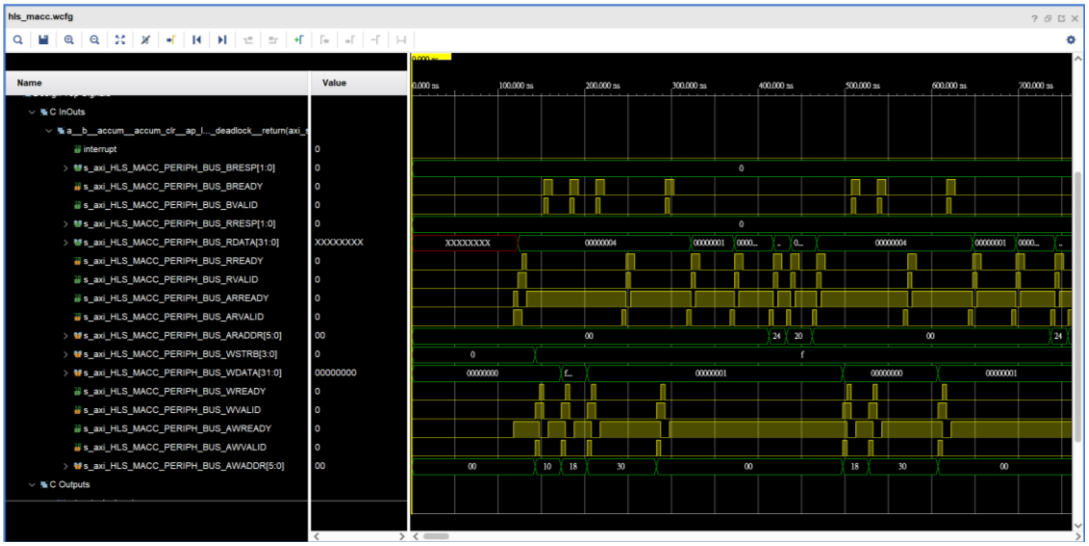

Figure. Co-sim waveform

最後進行 export RTL，選擇 vivado IP flow 進行輸出，記得配置相應版本提供後面 vivado IP 管理。

b. Vivado

打開 vivado，創建新的 RTL project，配置好板子設定選擇 pynq-z2。進入到 project 後，到 project Mamager/ setting/ IP / Repository/ Add，選擇匯入 HLS IP。之後創立 Block design。加入(add IP) zynq processor Block 和 HLS IP block。先進行 Run Block Automation，自動配置後再手動進行修改，雙擊 zynq processor block，MIO configuration/ Application processor unit/ 關閉 timer 0。到 interrupts/ fabric interrupts/ PS-PL interrupt ports/打開 IRQ_F2P。最後進行 Run Connection Automation 自動接線，與手動接線，連接 HLS interrupts 與 zynq IRQ_F2P。連接圖如下

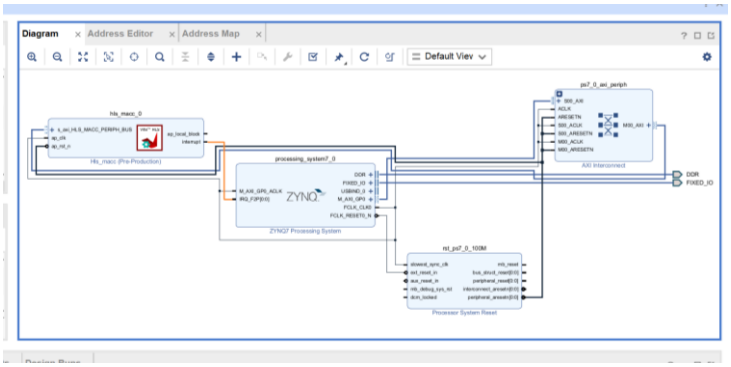

Figure. Block design connect interrupt and IRQ_F2P

確認 address map 無誤，即可進行後面步驟。

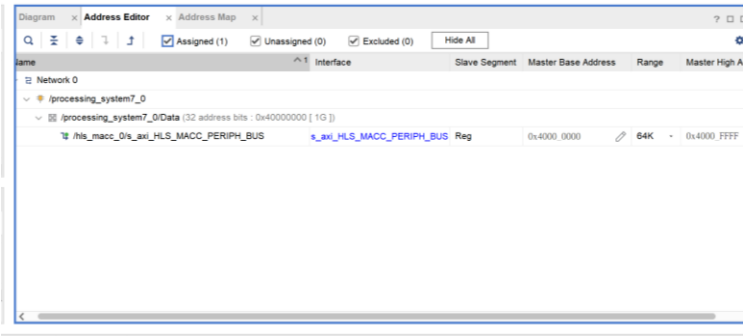

Figure. Check address map

打開 project manager/source/選擇 design，右鍵選單選 gererate output products… 完成後，再次右鍵選單選 Create HDL wrapper。最後創立好之後 Generate bitstream。

(optional) 生成 bitstream 之後使用 file/export/export hardware/輸出 XSA 檔案提供 vitis IDE 使用。打開 IDE 介面創立 Application project，並選定剛匯出的 XSA platform，選擇 hello world 模板方便配置。確認完成配置後進入選單 tools/program FPGA(此步驟需要 JTAG 連接 Host 端)，之後再以 serial port 連接。最後即可 run as/ launch on hardware (system debugger)。並以設置好的 c code 取代 helloworld.c 完成 RS 端對 PL 端的控制，並且進行 interrupt 之演示。

c. Overlay

由於本次實作上暫時法連接 Host 端 program FPGA，另外可以在遠端使用 juptyer notebooks 以 overlay 進行與 kernel 連接控制。這邊我們需要修改原本的 c code 變成 python。其中我們以 asyncio APIs 作為替代原本 ISR interrupt 之模塊，並分別以 CPU 與 FPGA 模塊計算之結果相互比較其正確性，並計時。

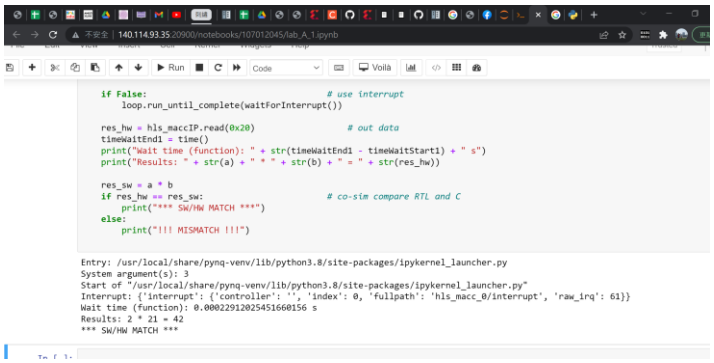

Figure. Overlay results

d. Vitis IDE

若手邊有 Pynq-Z2 板子則可針對在 Host 端進行編程 FPGA 並透過 IDE 對 PS 端進行控制來操作 PL 端編成好的 kernel。

先利用 vivado 匯出之.xsa file(記得要包含 Bitstream)建立一個 helloworld.c 模板的 application project。再來將 Pynq-z2 板子切換至 JTAG 模式，如下圖。

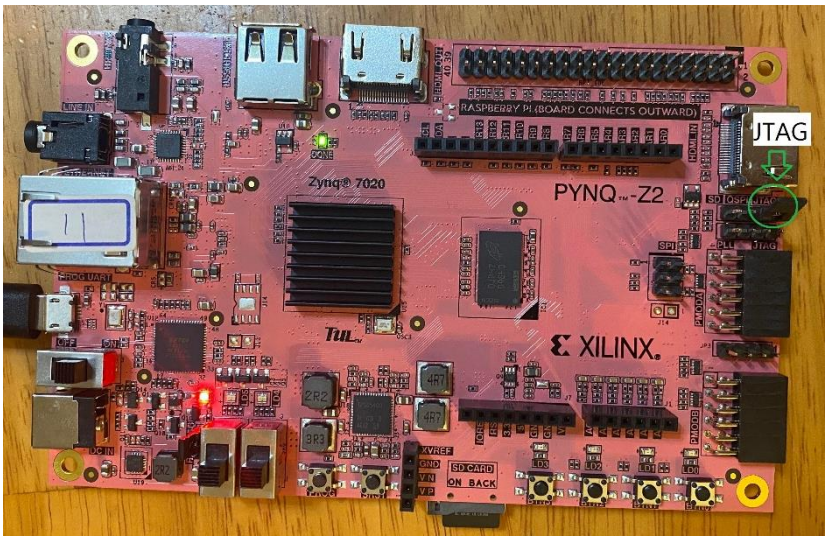

Figure. switch to jtag mode

切換完成後，選單 xilinx/program device，選擇好 device 後 program。再來切換到 debug mode，選擇 vitis serial terminal 點選 add，選擇好 port 並將 baud rate 設定為 115200，然後連接。連接完成後加入原先撰寫好的 C source，與 helloworld.c 代換。點選 build。注意這邊因為 vitis(2021.2)針對 xhls_macc.h 有部分 function 格式修改，如下表，因此要針對 source file 做些微修改。

Table. Xhls_macc.h some revisions

Vitis 2021.2	Older version
`XHls_macc_SetA();`	`XHls_macc_Set_a();`

`XHls_macc_SetB();`	`XHls_macc_Set_b();`
`XHls_macc_SetAccum_clr();`	`XHls_macc_Set_accum_clr();`
`XHls_macc_Get_accum();`	`XHls_macc_GetAccum();`

最終執行結果如下圖。

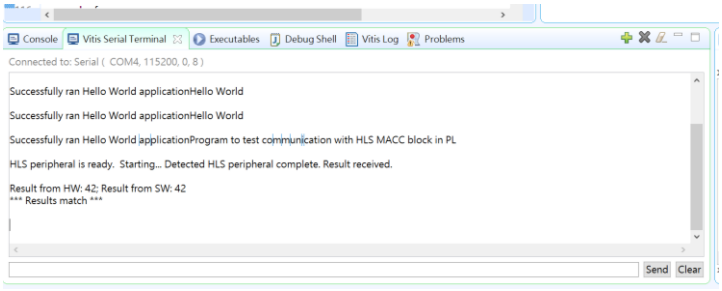

Figure. Vitis Serial terminal result

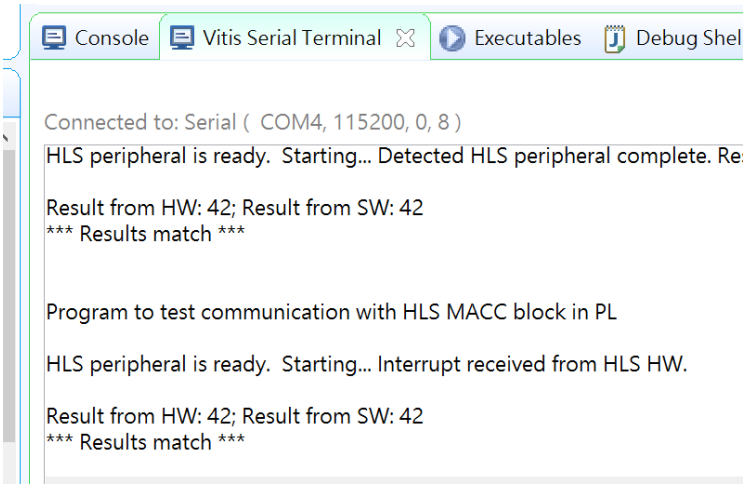

Figure. Vitis Serial terminal result (interrupt)

3. Optimization

HLS 已針對 IP 做相當程度的優化。整個 function 相當簡單，邏輯部分只有 acc_reg += a * b。需要乘法器與加法器。因此我們可以先分析其原本的 Timing Analysis 如下圖。可以看到讀入資料後所需要做最長的兩件事分別是 mul(乘法)與 add(加法)。

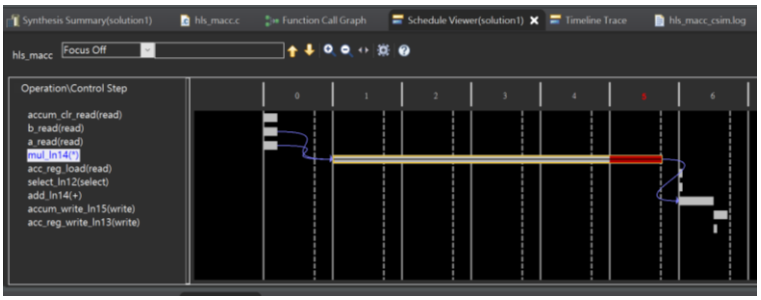

Figure. Optimized timing analysis

已知最長時間是 mul，那我們試著找方法減少 mul 的 operation time(latency)。這邊我們嘗試用#pragma HLS bind_op 來指定修改 operation 要使用何種方式完成。

```
15  #pragma HLS bind_op variable=acc_reg op=mul impl=fabric
16  #pragma HLS bind_op variable=acc_reg op=add impl=fabric
```

Figure. #pragma HLS bind_op example

其指令為#pragma HLS bind_op variable=<variable> op=<type>\ impl=<value> latency=<int>[1]。其中 variable 為指定變數，op 為選擇要設定的運算符(add、mul)，impl 為選擇要用何種硬體完成(dsp、fabric)，latency 選擇指定完成的 latency。其中下表有列出 combinational logic 支援的合成方法與 latency 估計。

Table. Supported combinations of finctional operations, implementation…

Table: Supported Combinations of Functional Operations, Implementation, and Latency

Operation	Implementation	Min Latency	Max Latency
add	fabric	0	4
add	dsp	0	4
mul	fabric	0	4
mul	dsp	0	4
sub	fabric	0	4
sub	dsp	0	0

根據上表，我們分別針對 mul 以 dsp 方式或以 fabric 方式合成出來並進行比較，比較如下表。

Table. Information of different implement method on “mul” operation

	DSP	fabric
Estimated timing (ns)	3.871	12.592
slacks	-0.22	-8.94
Latency(cycle)	6	2
Latency(ns)	30	25.184
Usage of dsp	3	0
Usage of FF	351	296
Usage of LUT	349	1369

可以觀察到，使用 DSP 方法會有較大的 latency，而用 fabric 方法只需要 2 cycle 的 latency，但 timing 會上升，經過整體比較之後發現使用 fabric 會有較小的 delay time。但會使用到較多的 LUT 資源。
因此我們同樣針對 fabric 方案進行合成硬體並實測。

[1] https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/pragma-HLS-bind_op

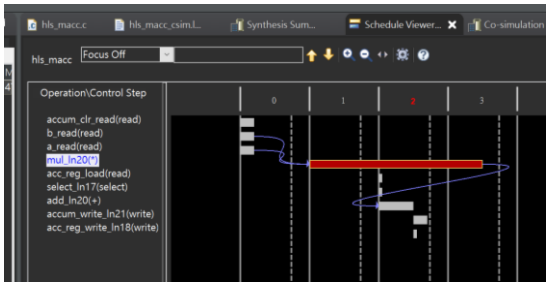

Figure. Timing analysis of fabric method “mul”

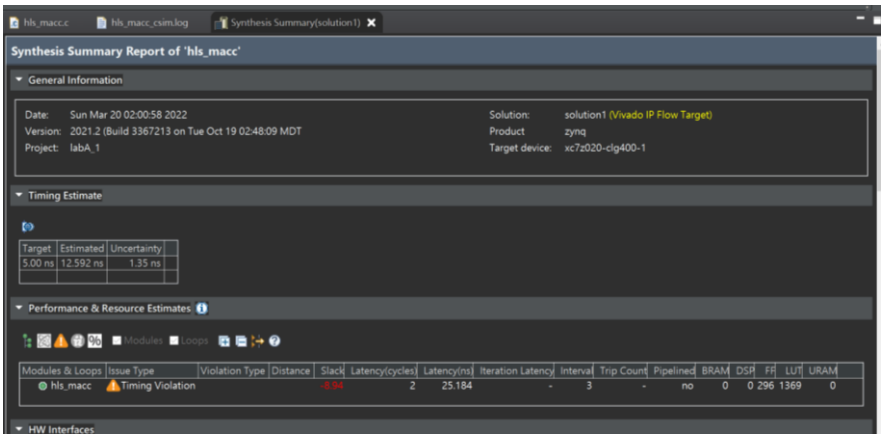

Figure. Synthesis summary of fabric method “mul”

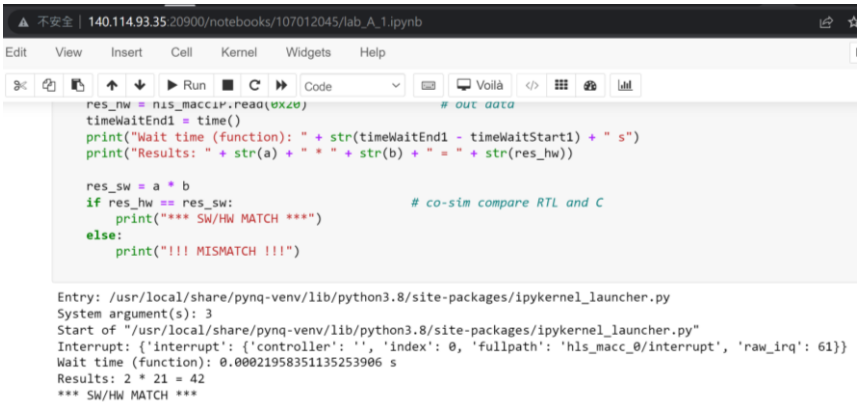

Figure. Overlay result of fabric method “mul”

經過兩者實際運算比較在 overlay 上季時的結果可以發現使用 fabric method 合成出來的 latency 較小，運算速度較快一些。

Table. Time compare between dsp method and fabric method “mul”

	dsp method	fabric method
Wait time (ms)	0.22912	0.21958

另外一種優化為若改成使用 axi master 方式進行，但實做出來 latency 大過於 axi lite，並且需要使用到 BRAM…等記憶體，相權衡之後對於此簡單的運算可能 pwer 會過大，並且需要修改 testbench，不利於實作與比較。

4. Performance Analysis

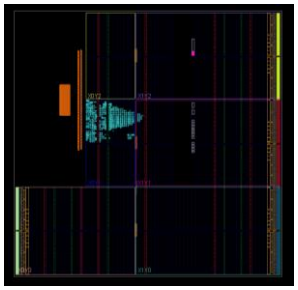

Figure. Vivado device diagram

最後合成結果如上。我們先分析 vivado 合成後最後出來的 timing 結果。以下皆有達到其 timing 要求。

Table. Timing report

Worst negative slack(setup time)	0.480 ns
Worst hold slack (hold time)	0.052 ns
Worst pulse width slack (WPWS)	4.020 ns

接下來分析其 power

Total on-chip power	1.403 W
Dynamic power	1.269 W
Static power	0.134 W

分析其 design utilization

LUT	1177
FF	748
BRAM	0
URAM	0
DSP	0

二、Lab 2 ----- real2xfft

1. Introduction

此 lab 主要要完成的在於結合多個 HLS IP 與 vivado IP，並且實作到 pynq-z2 板子上，透過 PS 端進行控制。其中兩個 HLS IP 分別為 Front-end “real2xfft.cpp”與 back-end “xfft2real.cpp”，主要是分別在做創建 sliding windows 和處理 output data 要到 stream data 的部分。FFT 的部分由 vivado 內建 IP 完成。Real FFT frontend processing takes in a stream of real data samples, creates a sliding window (1/2 old + 1/2 new data) and applies a Hamming windowing function to it before packing pairs of samples into a complex format to stream to an XFFT IP Core of N/2 points. This tutorial version takes 16-bit data assumed to be in a normalize fixed point format (1 whole bit and 15 fractional bits) and outputs 16-bit scaled (i.e. also normalized)

complex spectral data. RealFFT backend processing reorders the incoming stream (din) into bit-reversed address order then applies the algorithm to extract the spectral data for an FFT of N real points from data received from an N/2 complex FFT. The first output location contains the 0th and N/2th bin values, which are strictly real; outputs 1 to (N-1) are complex.

2. Design flow
   (1) HLS
      a. Front-end HLS design
      Create a new project and select “pynq-z2” board, then setting up the clock to 5 ns. Open explorer and add source files and testbench file. The file organization is shown below.

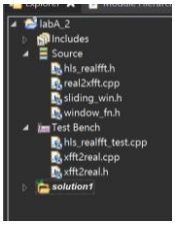

Figure. frond-end HLS file management

Run c-simulation and check no error message. Run synthesis.

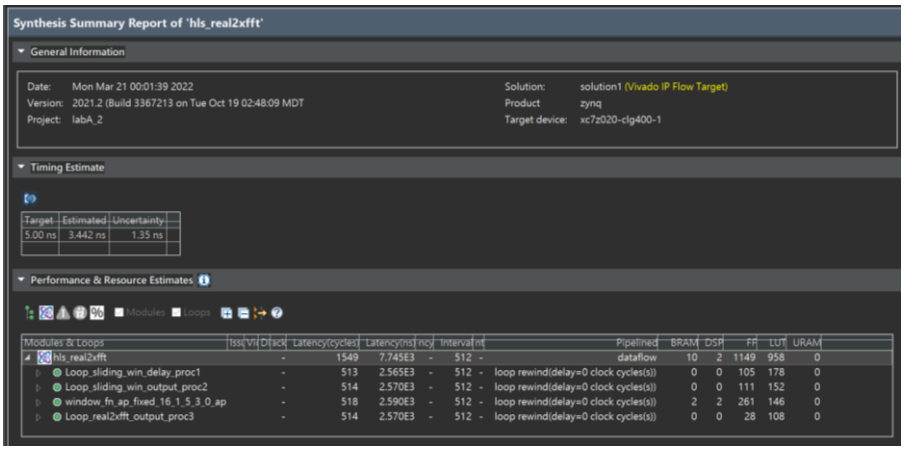

Figure. frond-end synthesis summary

完成合成後，進行 co-simulation

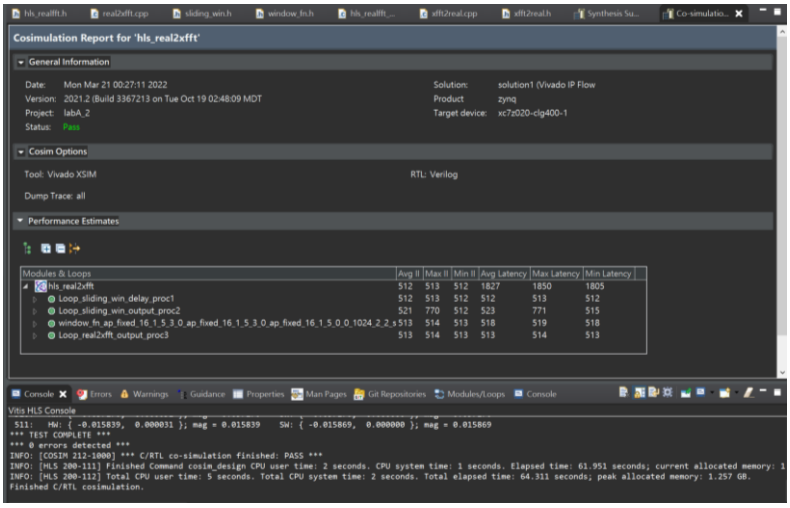

Figure. frond-end co-simulation

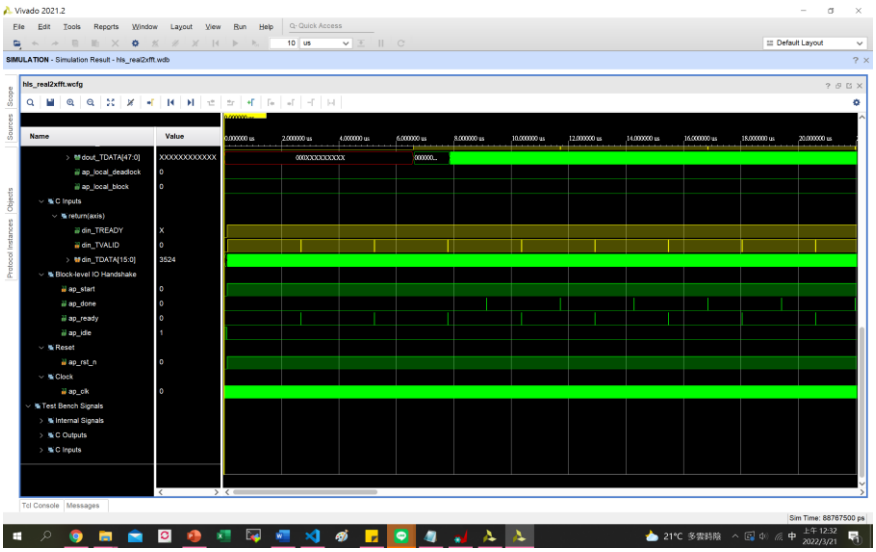

Figure. frond-end co-sim waveform

最後匯出 IP。

b. Back-end HLS design

另外創建一個新 project，同樣進行設定，並且匯入 source 和 testbench，其檔案管理如下。

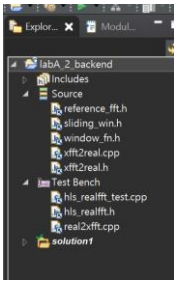

Figure. back-end file management

設定好之後同樣進行 c-sim，確定 software behavior 正常。

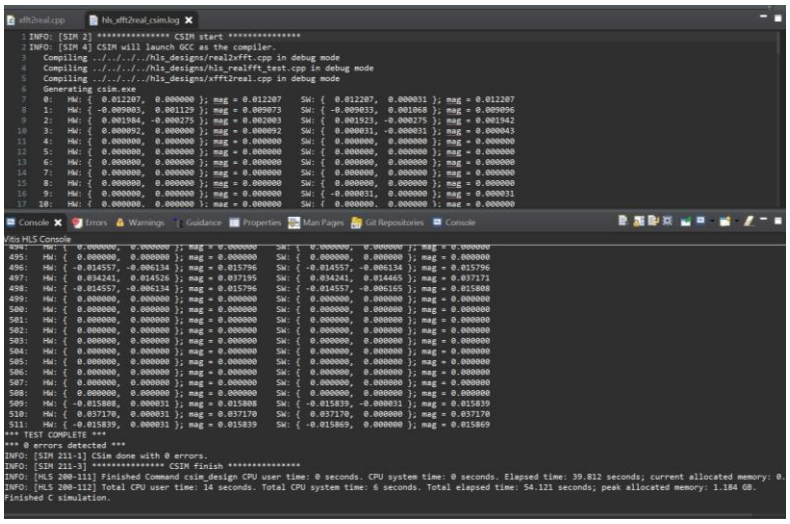

Figure. back-end c-simulaiton

設定好 top-function 之後進行合成。

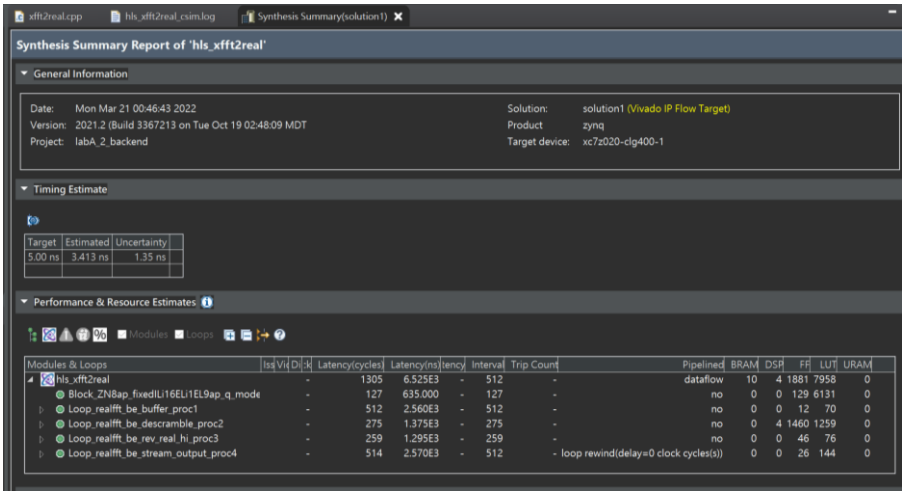

Figure. back-end synthesis summary

之後進行 co-simulation，比較合成之後的 RTL 結果和 C 是否符合。

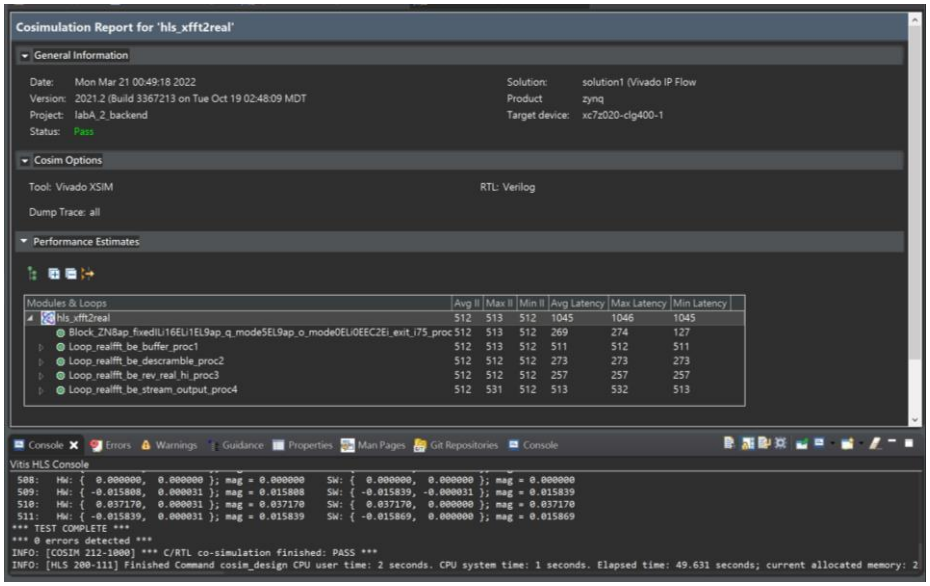

Figure. back-end co-simulation

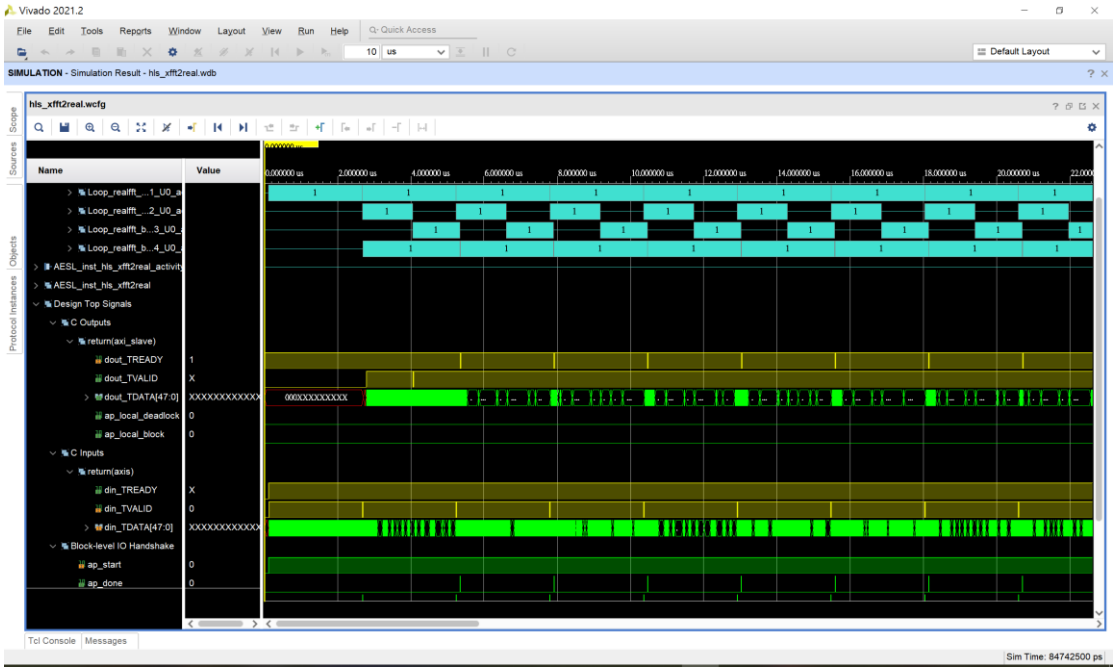

Figure. back-end co-sim waveform

確認無誤之後匯出 HLS IP。

(2) Vivado Block design

打開 vivado，創建新的 RTL project，配置好板子設定選擇 pynq-z2。進入到 project 後，到 project Mamager/ setting/ IP / Repository/ Add，選擇匯入 HLS IP。之後創立 Block design。先加入(add IP) fast fourier transform IP，並進行 re-customize，在 configuration 修改 transform length、target clock frequency、architecture(pipelined, streamIO)，然後在 implementation 修改 control signal 選擇 ARESETn

(active low)，確認 output ordering(bit/digit reversed order)和 throttle scheme(non-real time)。

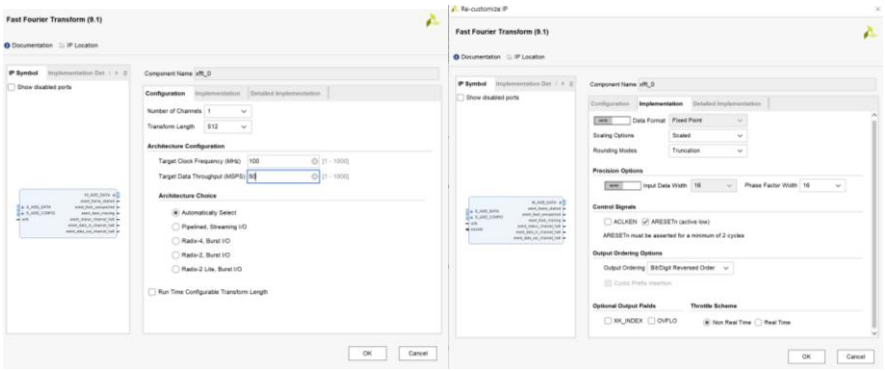

Figure. re-customized FFT IP

然後加入兩個 HLS IP，包括 hls_real2xfft_0 和 hls_xfft2xreal_0，並且以以下方式與 FFT IP 連接。連結完成後，將三者選擇起來點擊右鍵，選擇 Create Hierarchy，並命名 RealFFT。

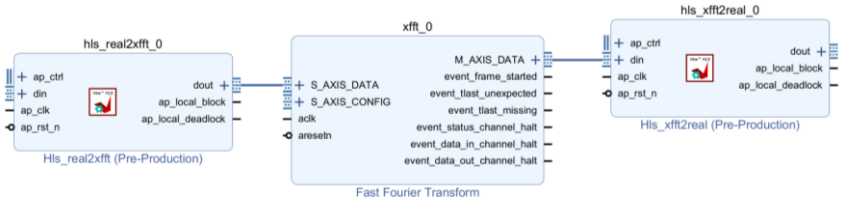

Figure. a FFT hierarchy block

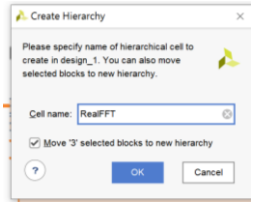

Figure. create hierarchy

包裝完成後重新打開 RealFFT，在 Hls_real2xfft 中找到 din，並針對該 pin 右鍵選單選 create interface pin，這邊要記得注意不是 port(不一樣)，有錯誤要重來，並且命名 realfft_s_axis_din。另外針對 ap_clk 點選 create pin，命名 aclk，這邊要記得注意不是 port(不一樣)，有錯誤要重來，針對 ap_rst_n 命名 aresetn。

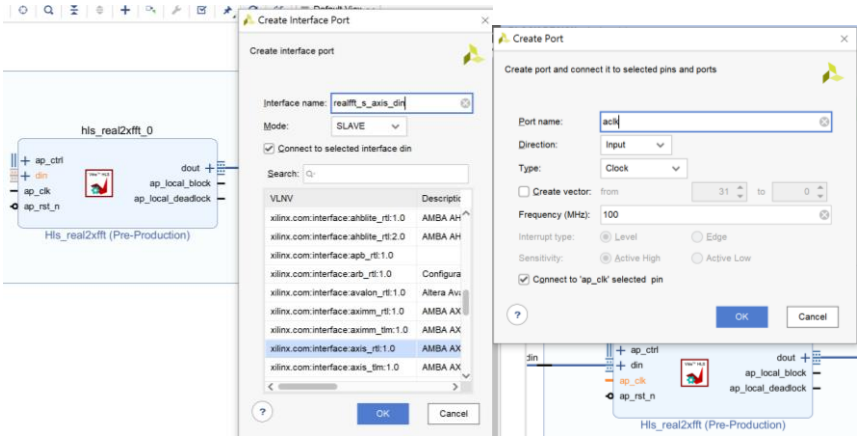

Figure. create interface port of din and ap_clk

同理，針對 Hls_real2xfft 中找到 dout，並針對該 pin 右鍵選單選 create interface pin，這邊要記得注意不是 port(不一樣)，有錯誤要重來，並且命名 realfft_m_axis_dout。另外針對 ap_clk 點選 create pin，命名 aclk 這邊要記得注意不是 port(不一樣)，有錯誤要重來。

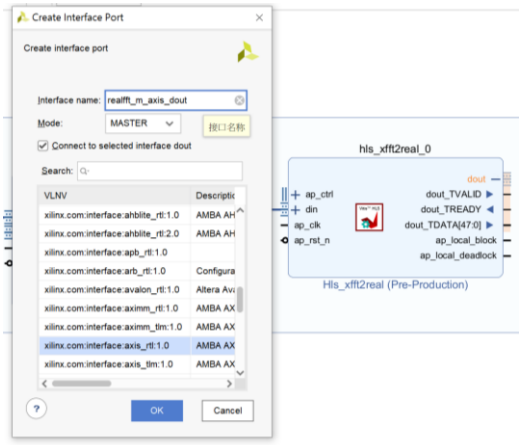

Figure. create interface pin

最後，將所有的 aresetn 連結起來，將所有的 aclk 連接起來。然後 add IP 創立一個 constant，並設定 1(high)，然後將其連接到 ap_start。然後 add IP 創立一個 constant，並設定 0(high)，為 16bits 寬，然後將其連接到 FFT IP 的 s_axis_config_tdata 與 s_axis_config_tvalid。

最後配置完成的設定如下圖

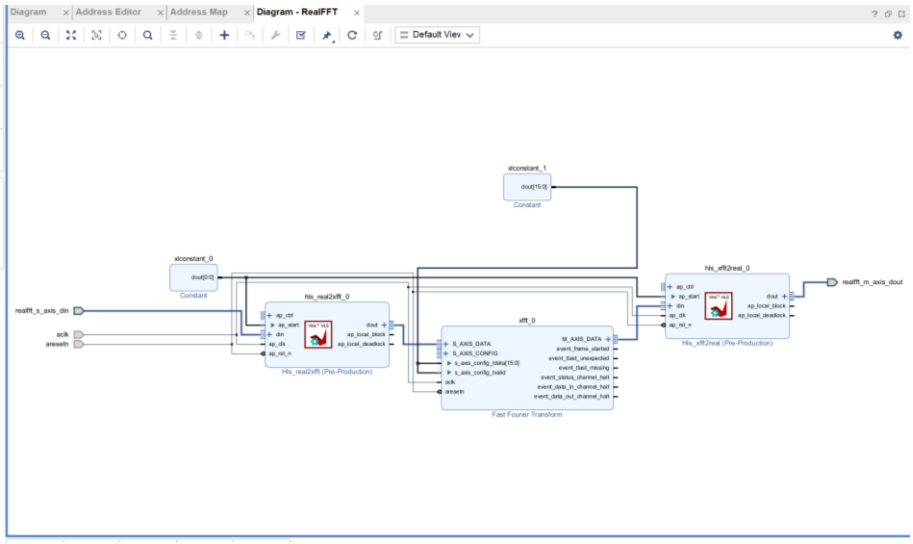

Figure. RealFFT block diagram

保存後回到主 diagram。添加 ZYNQ7-processing processor，這時會有 designer assistance 出現，點選 run block automation。完成後，雙擊 ZYNQ block，在 PS-PL Configuration/HP Slave AXI 點選 S_AXI_HP0 開啟，然後 Data width 為 64。再來點選 Clock Configuration/PL Fabric Clocks/選擇 FCLK_CLK0，frequency 為 100 MHz，完成後進行 apply customization。

再來，先將 realfft_s_axis_din(RealFFT)與 S_AXI_HP0(ZYNQ)連接，會自動生成一些 Block，如 AXI DMA，interconnection…等。完成後，再將 realfft_m_axis_dout(RealFFT)與 S_AXI_HP0(ZYNQ)連接，雙重連接後會顯示 Make connection dialog，重新布局 internnection 與讓"S2MM"連接。這時會有 designer assistance 出現，點選 run Connection automation，將 aclk 和 aresetn…等連線，最後系統如下圖。

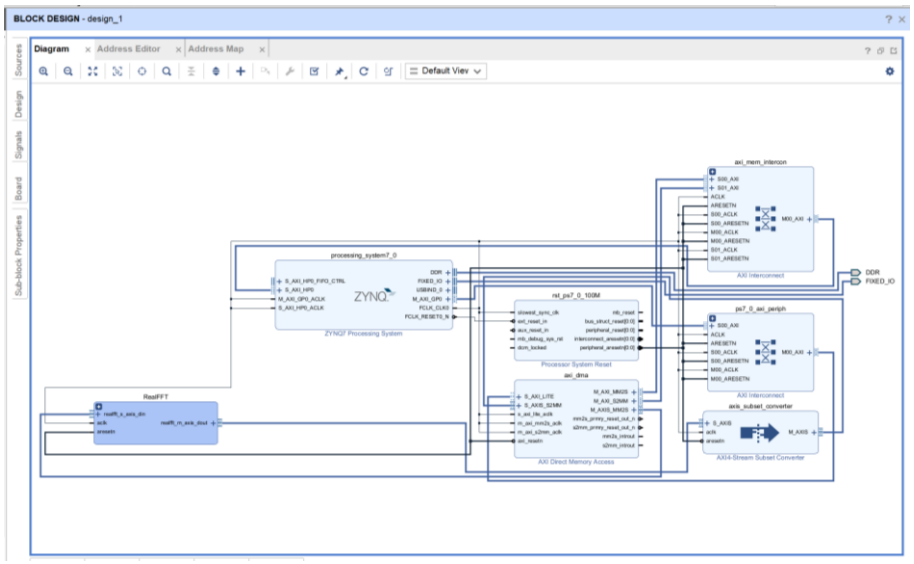

Figure. overall block design

完成後點選 validate design，只要沒有 error 即可。之後回到 explorer，點選 source/design 中的 block design，右鍵選單點 generate output products。完成後，在點選 create HDL wrapper。完成後點選 generate bitstream。

(3) Vitis IDE

生成 bitstream 之後，選單 file/export/export hardware，選擇 include bitstream。再來透過 Pynq-Z2 板子針對在 Host 端進行編程 FPGA 並透過 IDE 對 PS 端進行控制來操作 PL 端編成好的 kernel。

先利用 vivado 匯出之.xsa file(記得要包含 Bitstream)建立一個 helloworld.c 模板的 application project。再來將 Pynq-z2 板子切換至 JTAG 模式，如下圖。

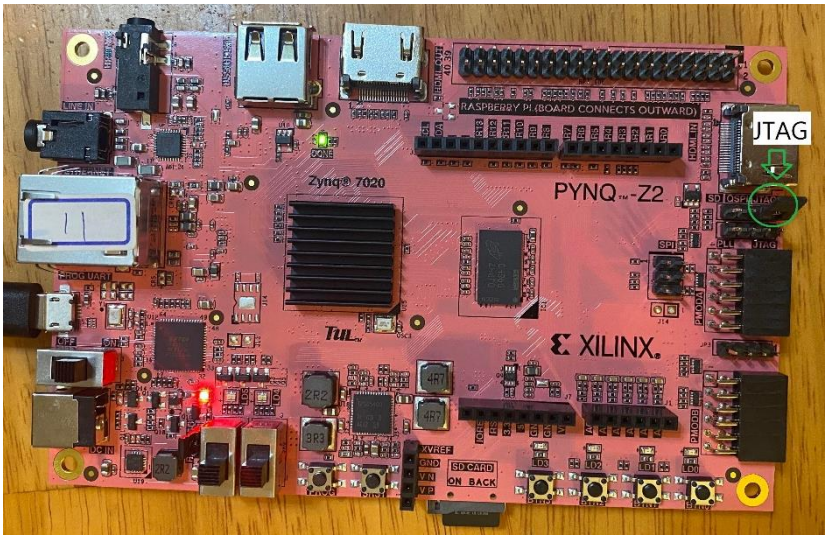

Figure. switch to jtag mode

切換完成後。切換到 debug mode，選擇 vitis serial terminal 點選 add，選擇好 port 並將 baud rate 設定為 115200，然後連接。連接完成先回到 design mode，點選 helloworld.c 進行 build。然後再回到 debug mode，再 explore 選單針對 PS 端的 cpu 點選 run as/launch hardware(single application debug)，確定 terminal 中有顯示資料，確定 PS 端與 CPU 溝通正常。然後替換 helloworld.c 為本次實驗要測試的 source，重複以上 build 與 run，注意這邊要先編輯 c/c++ build setting，點選 arm gcc linker/libraries，點選添加，輸入 m，保存，m 代表 libm(math)資料庫。

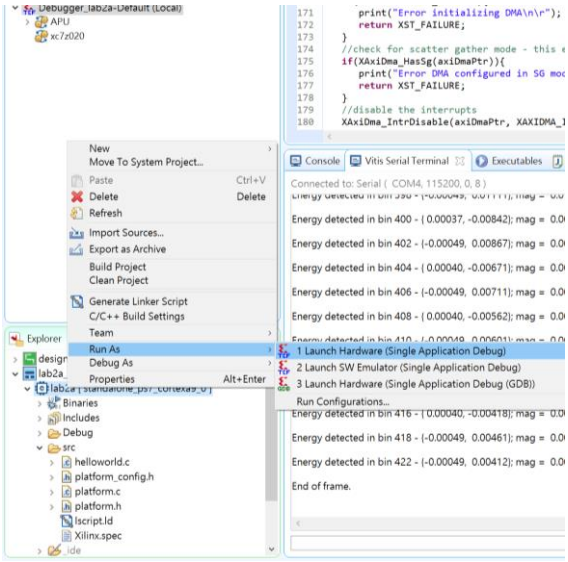

Figure. run as

最後執行完結果如下圖。

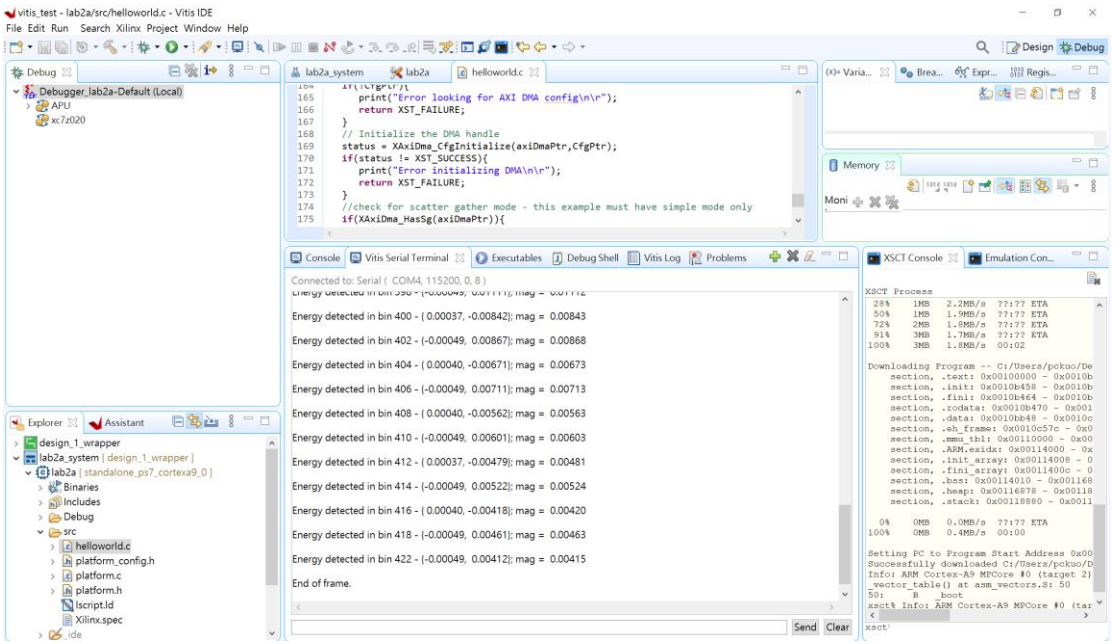

Figure. pynq-z2 run result

3. Optimization

比較，在關閉所有優化#pragma 與 pipeline 和 dataflow，其執行結果如下。

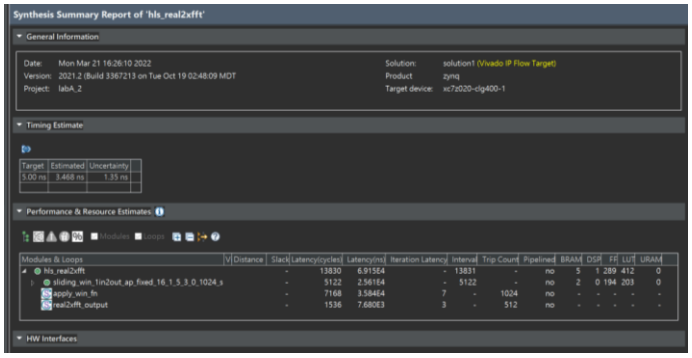

Figure. no optimization synthesis summary

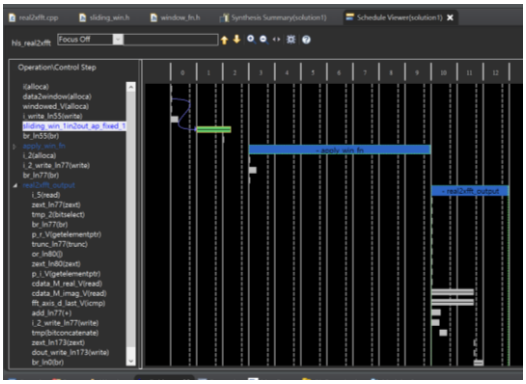

Figure. no optimization timing view

我們可以看到整體的 latency 來到 13830 cycle，6.915E4 ns，相當大，並且 dataflow 流程很亂。

那再來我們進行單純有 pipeline 和 dataflow 之模型做比較。

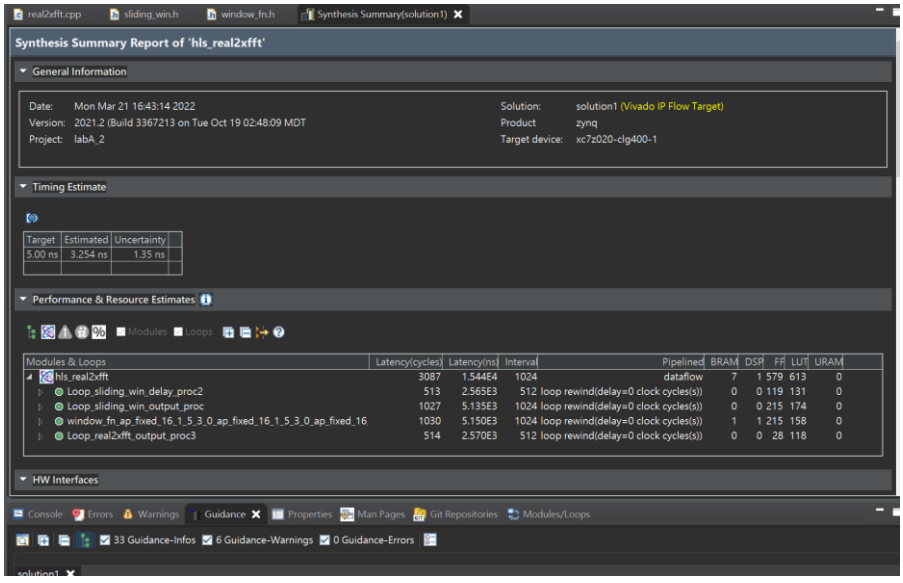

Figure. Only pipeline and dataflow synthesis summary

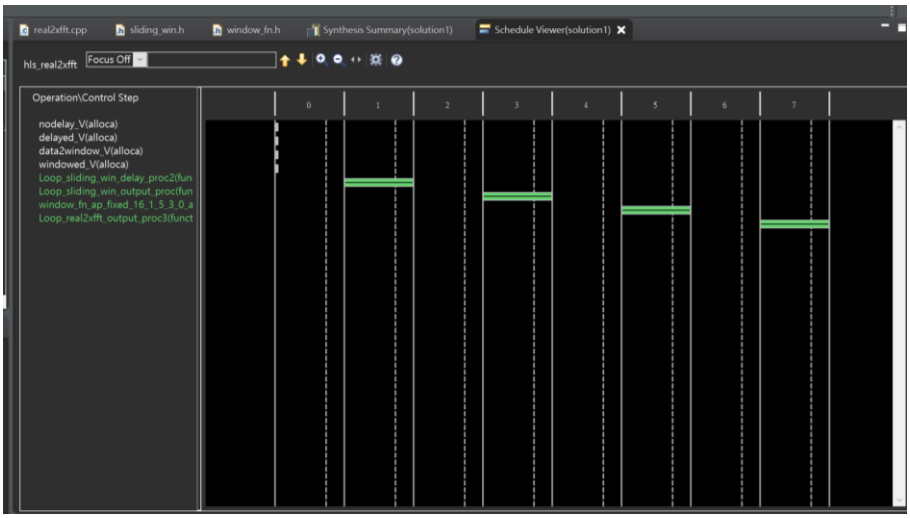

Figure. Only pipeline and dataflow timing summary

我們可以看到整體的 latency 來到 3087 cycle，1.544E4 ns，相較於沒有優化已有大幅度的改善了，並且 dataflow 變好。

另外我們在適度加上一些優化，看效能是否有變好

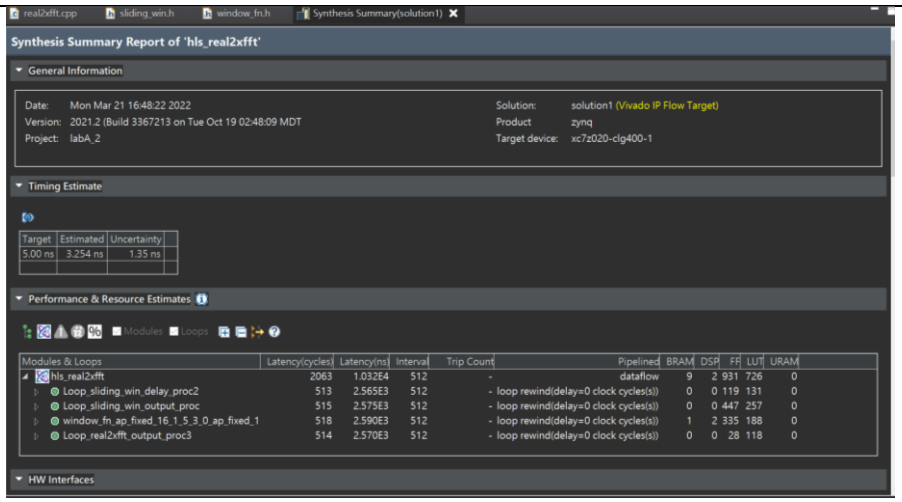

Figure. add unroll synthesis summary

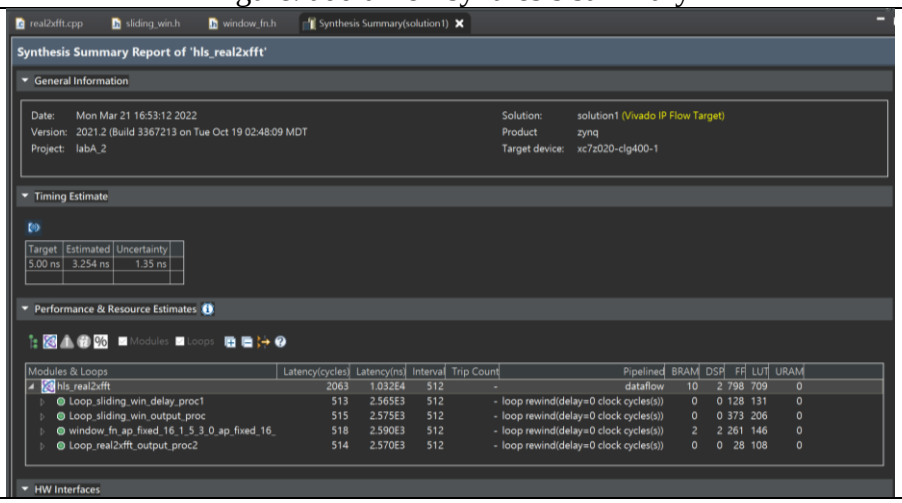

Figure. add stream synthesis summary

這邊我們發現有一個#pragma HLS ARRAY_STREAM 現在已經不支援，因此我們著手修改成#pragma HLS STREAM 的格式嘗試優化

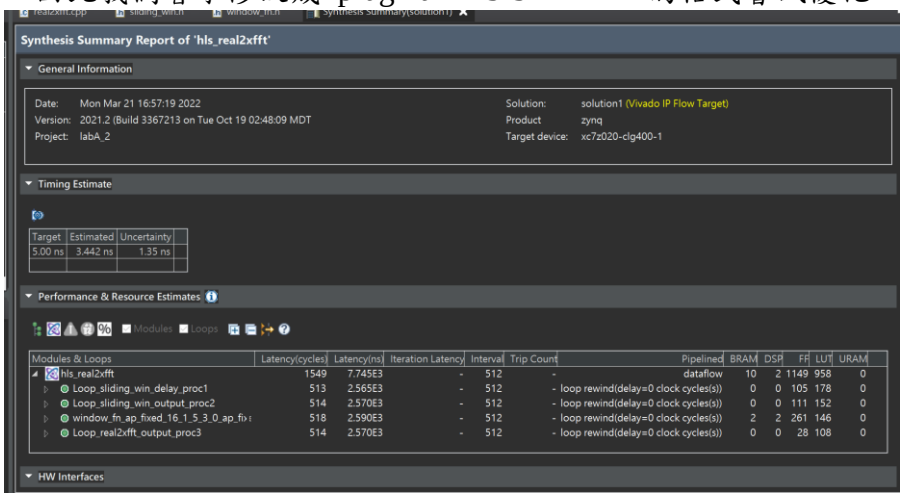

Figure. add all stream synthesis summary

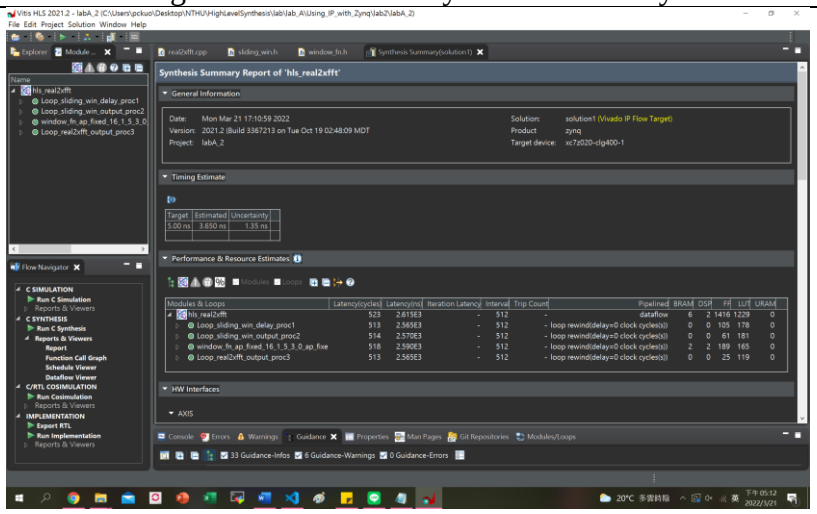

Figure. add array_partition synthesis summary

最後我們嘗試可以優化到的結果如下並且與完全沒有優化做比較，如下表。可以看到我們多消耗了硬體資源，但大幅減少了 latency。而這邊硬體資源的增加主要來自於 array_partition。

Table. optimization compare

	optimized	Non-optimized
Estimated timing (ns)	3.650	3.468
Latency (cycles)	523	13830
Latency (ns)	2.615 E3	6.915 E4
BRAM	6	5
DSP	2	1
FF	1416	289
LUT	1229	412

最後我們嘗試不同情況的 array partition 之情況，比較其優化結果。

	Array partition 2	Array partition 4
Estimated timing (ns)	3.650	3.532
Latency (cycles)	523	524
Latency (ns)	2.615 E3	2.620 E3
BRAM	6	10
DSP	2	4
FF	1416	2924
LUT	1229	2207

可以觀察到，雖然其 timing 有減少，而 cycle 只增加 1，但是 latency 卻沒有明顯進步，並且硬體消耗甚大。

4. Performance analysis

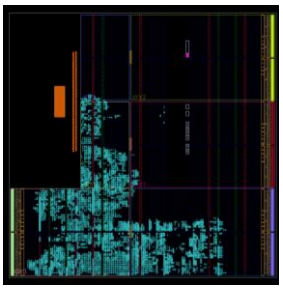

Figure. device diagram

最後合成結果如上。我們先分析 vivado 合成後最後出來的 timing 結果。以下皆有達到其 timing 要求。

Table. Timing report

Worst negative slack(setup time)	0.980 ns
Worst hold slack (hold time)	0.021 ns
Worst pulse width slack (WPWS)	3.750 ns

接下來分析其 power

Total on-chip power	1.509 W
Dynamic power	1.370 W
Static power	0.139 W

分析其 design utilization

LUT	7894
FF	9094
BRAM	18.5
URAM	0
DSP	18

三、Observation

1. 在 lab1 中，由於一開始還沒有拿到板子，無法直接對 host 端進行編程，因此另外撰寫 python code 嘗試在 jupyter_notebook 上執行。最後發現原來其實在 vivado 完成後的 bitstream 檔案可以有兩種路徑(design flow)，overlay 或是 vitis。
2. 了解如何使用 block design、IP integrator，以及如使用 interrupt。
3. 了解如何使用 AXI-lite 與 AXI-stream。

四、Github hierarchy

Link: https://github.com/pckuo95/HLS_labA

- lab1/
  - HLS_macc/
    - IP/
      - export.zip
      - xilinx_com_hls_hls_macc_1_0.zip
      - readme.txt // 兩個皆是 HLS IP
    - report/
      - csynth.rpt
      - hls_macc_csim.log
      - hls_macc_csynth.rpt
      - readme.txt
  - HLS_source/ // HLS code and testbench
    - hls_macc.c
    - hls_macc.h
    - hls_macc_test.c // testbench
    - run_hls.tcl // run file
    - readme
  - Vivado/
    - report/
      - design_1_wrapper_power_routed.rpt
      - design_1_wrapper_timing_summary_routed.rpt
      - design_1_wrapper_utilization_placed.rpt
      - readme.txt
    - design_1_wrapper.bit // bitstream
    - design_1_wrapper.hwh // handoff file
    - design_wrapper.xsa // export hardware
  - arm_source/
    - overlay/
      - pynq_overlay_test.py // overlay control PS
      - readme.txt
    - vitis/
      - zynq_design_test.c // PS 端 code
      - readme.txt
- lab2/
  - HLS_real2xfft/ // front-end
    - report/
      - csynth.rpt
      - hls_real2xfft_cosim.rpt
      - hls_real2xfft_csim.rpt
      - hls_real2xfft_csynth.rpt
      - readme.txt
    - xilinx_com_hls_hls_real2xfft_2_0.zip // HLS frond-end IP
  - HLS_source/ // HLS code and testbench
    - hls_realfft.h
    - hls_realfft_test.cpp // testbench
    - real2xfft.cpp // front-end source
    - reference_fft.h
    - run_hls.tcl // run file
    - sliding_win.h
    - w_rom_1k_1quad_init.txt
    - xfft2real.cpp // back-end source
    - xfft2real.h // back-end header
    - readme.txt
  - HLS_xfft2real/ //back-end
    - report/
      - csynth.rpt
      - hls_xfft2real_csim.log
      - hls_xfft2real_csynth.rpt
      - readme.txt
    - xilinx_com_hls_hls_xfft2real_1_0.zip // HLS back-end IP
  - Vivado/
    - design_1.hwh // handoff file
    - design_1_wrapper.bit // bitstream
    - design_1_wrapper.xsa // export hardware

- readme.txt
- arm_source/
  - helloworld.c // PS 端 code
  - readme.txt
- report/
  - labA_no5_107012045.pdf // this report
  - labA_no5_107012045.pptx // slide
- README.md