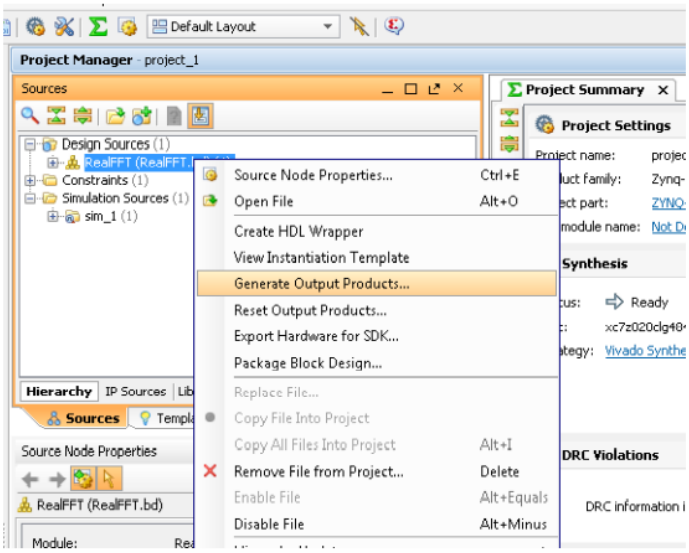
一、Lab 操作流程與問題

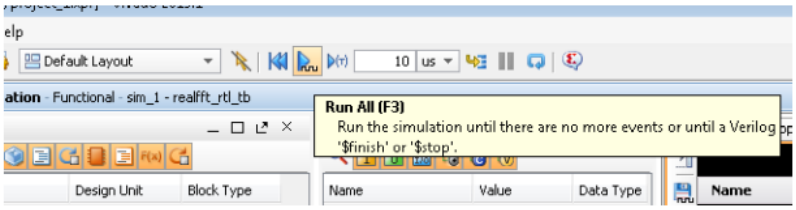
這一次 lab 的重點有兩個，一個是演算法本身的內容，另一個是學習如何在 Vivado 上合成自己在 HLS 上做出來的 IP 還有 Vivado 內建的 IP。以下列出個人認為這次 lab 所學到比上上次 lab1 和 lab2 多出來的流程，並且附上操作截圖。當儲存完設計好的 block design 之後，首先要做的第一件事情是 generate output products，操作截圖如下:



圖一

跑完 Generate Output Products 步驟之後，接著才是我們所熟悉的 HDL wrapper，不過我第一次跑的時候發現過了五十分鐘仍然跑不出來，最後才知道如果跑的時間太久可能是 Vivado 突然有問題，這時候需要重新開一個 project 重頭開始做。最後是我們利用 Vivado 上的 simulation 工具驗證電路合成之後的表現是否符合預期，因此我們同樣可以加進 testbench，並且 Run Simulation。最後要特別注意的是，因為我們的 testbench 秒數比較長，因此我們需要按下

Run ALL 指令，使得 testbench 可以不限時間地繼續往下跑，但是如果只是要挑其中一到三組資料驗證的話，其實可以只跑限定秒數 10us，或是設定成自己想要的秒數。



圖二

二、演算法與系統架構

這次 lab A 總共用到三個主要的演算法，分別是 STFT、FFT 和 descrambler。

(一) Short-Time Fourier Transform (STFT)

STFT 最常見的用途為分析連續長時間的時變頻譜資訊，像是聲音頻率分析或是生醫應用等等，在這一次 lab A 有用到 STFT。STFT 在輸入 STFT 的輸入訊號之前會將輸入訊號乘以 window function 的加權係數，也就是在時域上將輸入訊號和 window function 相乘。經過傅立葉轉換之後，相當於原訊號本身的頻譜和 window function 的頻譜做捲積，我們將上述敘述用數學算式表示:

$$X(\omega, m) = \sum_{n=-\infty}^{\infty} w[n]x[n+m]e^{-j\omega n} \ldots (1)$$

$$w[n]x[n] \leftrightarrow \frac{1}{2\pi} W(\omega) \otimes X(\omega) \ldots (2)$$

其中不同的 window function 有不同的頻譜，因此會產生出不同的頻譜結果，使得時間解析度或是頻率解析度不同，在這一次 lab A 所採用的 window function

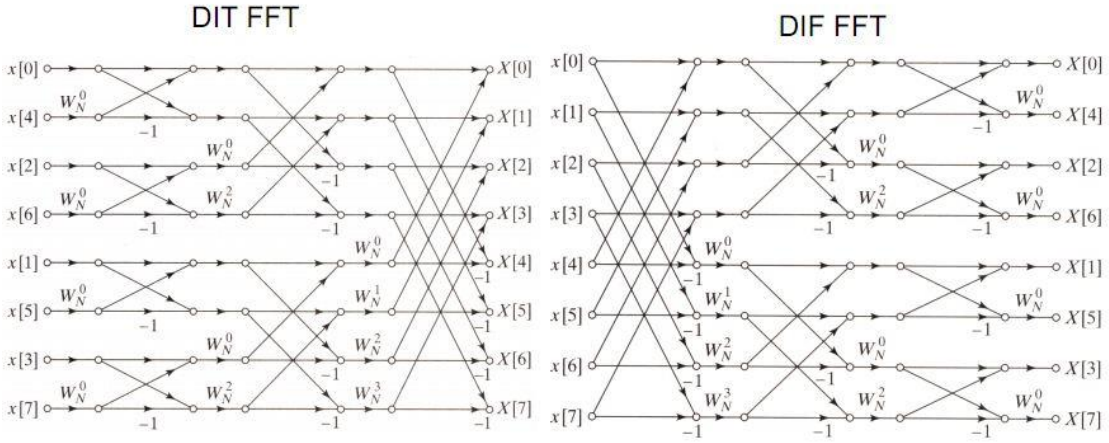
為相當常用的 Hamming window。

(二) Fast Fourier Transform (FFT)

FFT 就是計算複雜度比 DFT 更小卻能算出相同結果的演算法，常常應用在通訊系統硬體上。FFT 的原理為將 N-point 的 DFT 分成兩個$\frac{N}{2}$-point 的 DFT，其中有一個$\frac{N}{2}$-point 的 DFT 需要另外乘以矩陣元素為 twiddle factor 的向量，上述敘述的數學式子如下:

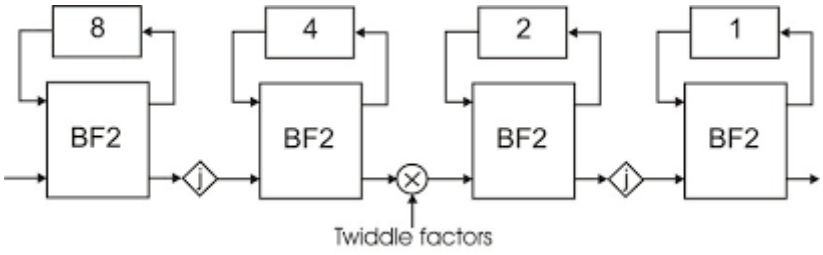
$$\begin{cases} \begin{bmatrix} X[0] \\ X[1] \\ \vdots \\ X[\frac{N}{2}-1] \end{bmatrix} = \begin{bmatrix} Y[0] \\ Y[1] \\ \vdots \\ Y[\frac{N}{2}-1] \end{bmatrix} + \begin{bmatrix} 1 & e^{-j\frac{2\pi}{N}\times 1} & \dots & e^{-j\frac{2\pi}{N}\times(\frac{N}{2}-1)} \end{bmatrix} \begin{bmatrix} Z[0] \\ Z[1] \\ \vdots \\ Z[\frac{N}{2}-1] \end{bmatrix} \\ \begin{bmatrix} X[\frac{N}{2}] \\ X[\frac{N}{2}+1] \\ \vdots \\ X[N-1] \end{bmatrix} = \begin{bmatrix} Y[0] \\ Y[1] \\ \vdots \\ Y[\frac{N}{2}-1] \end{bmatrix} - \begin{bmatrix} 1 & e^{-j\frac{2\pi}{N}\times 1} & \dots & e^{-j\frac{2\pi}{N}\times(\frac{N}{2}-1)} \end{bmatrix} \begin{bmatrix} Z[0] \\ Z[1] \\ \vdots \\ Z[\frac{N}{2}-1] \end{bmatrix} \end{cases} \dots (3)$$

事實上，任何 N 為 2 的冪次方的 DFT 都可以被無限分解直到矩陣的 size 為 1，而 FFT 即是利用將 N-point 的 DFT 經過無限分解而使得計算複雜度比原本的 DFT 少。FFT 的範例硬體架構如下:



圖三

圖三為展開 FFT 之後的架構圖，左邊 DIT 指的是 decimation in time，而右邊 DIF 則指的是 decimation in frequency。在這次 lab A 中 HLS 的 testbench 所採用的 FFT 為 decimation in frequency，以下都以 DIF 的架構討論。而從上圖一可以發現，FFT 的輸出其 index 的順序和輸入的時候不一樣，事實上其 index 順序為被 bit-reversed 後的 index 順序，因此在後面提到的 xfft2real 會做 bit-reverse 的處理。此外，最常見的 FFT 硬體架構如下:



圖四

圖四為 single-delay feedback 的 FFT 硬體架構，輸入和輸出訊號不像圖三同時進入架構內運算，而是像類似 stream 的方式資料會不停地按照順序進入，並且輸出訊號順序為 bit-reversed 的順序。也因為這樣，這次 lab A 的整個系統模組間的協定很適合採用 AXI Stream。

(三) Descrambler

在這次 lab A 因為 FFT 的輸入實部和虛部所乘以的 window function 具有時間差，因此需要在經過 FFT 之後的頻率訊號做 descramble 處理，使得誤差相位值被補償回來。Descrambler 演算法的數學算式如下:

$$y_1 \; is \; the \; i-th \; frequency \; component.$$
$$y_2 \; is \; the \; mirrored \; frequency \; component \; of \; y_1.$$

Then, we have

$$f = \frac{Re\{y_1\} + Re\{y_2^*\}}{2} + j\frac{Im\{y_1\} + Im\{y_2^*\}}{2} \ldots (4)$$

$$g = \frac{Im\{y_1\} - Im\{y_2^*\}}{2} + j\frac{Re\{y_2^*\} - Re\{y_1\}}{2} \ldots (5)$$

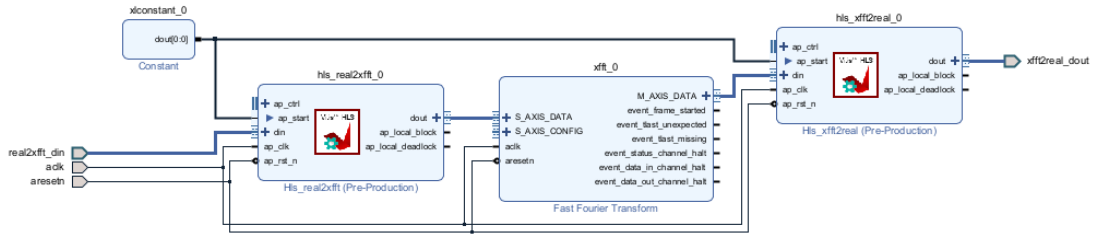
$$w = \cos\left(\frac{2\pi i}{1024}\right) - j\sin\left(\frac{2\pi i}{1024}\right) = e^{-j\frac{2\pi i}{1024}} \ldots (6)$$

$$cdata = f + wg \ldots (7)$$

其中，g 為被修正項，g 乘以 w 可以做相位補償，而最後可以得到消除不理想效應後的 cdata。

(四) 系統架構

下圖為在 Vivado 上接出來的 IP 電路。



圖三

下列表格為整個系統架構的 I/O ports。

name	I/O	Width
aclk	I	1
aresetn	I	1
ap_start	constant	1
real2xfft_din	I	16
xfft2real_dout	O	32

圖四

圖三中共有三個模組，hls_real2xfft_0、xfft_0 和 hls_xfft2real。因為資料為連續進入系統，所以三者的接口協定都採用 AXI Stream。aclk 就是時脈訊號，aresetn 則是 reset 控制訊號，ap_start 原本為協定中的控制訊號，作為開啟的用

途，在這次 lab 直接設定為常數 1，使得兩個需要 ap_start 的電路永遠處於可以運轉的狀態。real2xfft_din 為資料輸入，而 xfft2real_dout 則為資料輸出。

三、程式碼

	The core codes of the function real2xfft
1.	void hls_real2xfft(
2.	hls::stream<din_t>& din,
3.	hls::stream<xfft_axis_t>& dout) {
4.	#pragma HLS INTERFACE axis port=dout
5.	#pragma HLS INTERFACE axis port=din
6.	#pragma HLS ARRAY_PARTITION variable=data2window cyclic factor=2
7.	#pragma HLS ARRAY_PARTITION variable=windowed cyclic factor=2
8.	#pragma HLS ARRAY_STREAM variable=data2window,windowed depth=2
9.	#pragma HLS DATAFLOW
10.	din_t data2window[REAL_FFT_LEN], windowed[REAL_FFT_LEN];
11.	sliding_win_1in2out<din_t, REAL_FFT_LEN>(din, data2window);
12.	window_fn<din_t, din_t, coeff_t, REAL_FFT_LEN, WIN_FN_TYPE,
13.	2>(data2window, windowed);
14.	real2xfft_output:
15.	for (int i = 0; i < REAL_FFT_LEN; i += 2) {
16.	#pragma HLS PIPELINE rewind
17.	dout_t cdata(windowed[i], windowed[i + 1]);
18.	xfft_axis_t fft_axis_d;
19.	fft_axis_d.data = cdata;
20.	fft_axis_d.last = i == REAL_FFT_LEN - 2 ? 1 : 0;
21.	dout.write(fft_axis_d);
22.	}
23.	}

圖五

這個函數對應到真實電路中 real2xfft 模組的 top module，輸入為整個系統的輸入，輸出訊號會送到 fft 模組。第二行和第三行為整個模組的 arguments，有點類似真實電路的 ports，並且其資料型態需要配合訊號所使用的協定，像是

這次 project 採用 AXI Stream 協定，因此兩個分別代表輸入和輸出的 arguments 資料型態為 hls::stream<A>&。只要訊號使用 stream 協定，則必須使用 hls::stream<>宣告。使用此資料型態宣告的最大特色為每次資料只能讀一次，再讀一次則會自動讀到下一筆資料。<>內的 A 代表資料的大小，像是 din_t 代表 16 位有正負號並且其數值大小在[-1,1)之間(此規格宣告在其他標頭檔，在下個部分會解釋)。xfft_axis_t 則為複數，並且不管是實數還是虛數，其規格皆和 din_t 相同。

	Some parts of the header file hls_realfft.h
1.	#include <complex>
2.	#include <ap_fixed.h>
3.	#define DIN_W 16
4.	#define DOUT_W DIN_W
5.	typedef ap_fixed<DIN_W, 1> din_t;
6.	typedef complex<ap_fixed<DOUT_W, 1> > dout_t;
7.	typedef struct {
8.	dout_t data;
9.	ap_uint<1> last;
10.	} xfft_axis_t;

圖六

上一個部分所提到的標頭檔即 hls_realfft.h，din_t 其實就是用 HLS 內建標頭檔 ap_fixed.h 人為製造出來的資料型態。ap_fixed 的 template 最多有五個，第一個代表資料寬度(width)，第二個則代表 integer word length，像是如果是 1 的話，以二進位制來說，就是只有一位數在小數點左邊，因此如果數字有正負號的話，數字範圍就是[-1,1)。至於 dou_t 的話，比 din_t 多出 complex<>，而<>內的資料型態和 din_t 相同，因此 dou_t 其實就是 din_t 的複數版本。最

後是第七行到第十行的 xfft_axis_t，可以發現這個 struct 內有兩個資料，一個是代表訊號本身，另一個則是 last 訊號，這個 last 訊號對應到採用 stream 協定的真實電路中 last 控制訊號，當 last 為 1 的時候，代表這筆資料是這一組連續資料中的最後一筆資料。

接著回到圖五的第五行到第十一行，這部分主要都是 pragma 指令。第一個使用到的指令為:

#pragma HLS INTERFACE axis port=xxx

其指令的意義為使得 xxx 變數在之後被合成出電路中的訊號時，設定成 AXI STREAM 協定。第二個用到的指令為:

#pragma HLS ARRAY_PARTITION variable=xxx cyclic factor=K

其指令的意義為使得 xxx 變數在之後被合成出電路中的訊號時，可以將一整組向量的資料分成兩組對應的電路結構，像是如果後面接上 cyclic factor=K 的話，對應到真實電路可以類比為將從上級輸入的訊號輪流分堆到 K 組 FIFO 或是 K 條訊號線，並且用多工器實踐電路。第三個用到的指令為:

#pragma HLS ARRAY_STREAM variable=xxx depth=K

其指令的意義為使得 xxx 變數在之後被合成出電路中的訊號線上加上 FIFO，depth 的設定值即為 FIFO 的長度，需要加上 ARRAY_的原因是因為此訊號在這之前有被分割矩陣，因此需要加上 ARRAY 才不會讓 HLS 判斷這個訊號整組向量只需要一組深度為 2 的 FIFO。第四個用到的指令為:

#pragma HLS DATAFLOW

其指令的意義為使得在和此指令同一個層級中的所有後級的 task 不需要等待

前級的 task 做完所有運算就可以開始進行運算，也就是說只要一拿到可以拿來運算的上一級輸出訊號，就立刻開始運算並且傳送輸出訊號給下一輪。因為此電路的輸入輸出協定接為 stream，因此可以預期電路的輸入輸出都像是流水線一樣不停地輸入訊號和輸出訊號，因此在這個電路中間如果沒有設定 dataflow 的話，中間會因為大量資料卡頓使得 latency 大幅增加。

圖五中第十二行到第十四行對應到這個電路的兩個 submodules，sliding_win_1in2out 和 window_fn，並且兩者的輸出訊號分別為 data2window 和 windowed，其資料型態都是 din_t，也就是寬度 16 位元，並且數值範圍在[-1,1)之間的訊號。以下圖七和圖八分別為上述兩個 submodule 內部的 codes。

	The core codes of the function sliding_win_1in2out
1.	template<typename T, int LEN>
2.	void sliding_win_1in2out(
3.	hls::stream<T>& din,
4.	T *dout) {
5.	enum {DELAY_LEN = LEN / 2};
6.	enum {DELAY_FIFO_DEPTH = DELAY_LEN / 2};
7.	T din_val;
8.	static ap_shift_reg<T, DELAY_LEN> delay_line;
9.	T nodelay[LEN/2], delayed[LEN/2];
10.	#pragma HLS ARRAY_PARTITION variable=nodelay,delayed cyclic
11.	factor=2
12.	#pragma HLS STREAM depth=DELAY_LEN variable=nodelay
13.	#pragma HLS STREAM depth=DELAY_FIFO_DEPTH variable=delayed
14.	#pragma HLS INLINE
15.	#pragma HLS DATAFLOW
16.	sliding_win_delay:
17.	for (int i = 0; i < LEN / 2; i++) {

```
18. #pragma HLS pipeline rewind
19.         din_val = din.read();
20.         nodelay[i] = din_val; // din
21.         delayed[i] = delay_line.shift(din_val);
22.     }
23. sliding_win_output:
24.     for (int i = 0; i < LEN; i++) {
25. #pragma HLS UNROLL factor=2
26. #pragma HLS pipeline rewind
27.         T dout_val;
28.         if (i < LEN / 2) {
29.             dout_val = delayed[i];
30.         } else {
31.             dout_val = nodelay[i % (LEN / 2)];
32.         }
33.         dout[i] = dout_val;
34.     }
35. }
```

圖七

這個函數的功能為將輸入訊號進行分流的處理，並且對應到真實的電路會有 shift registers 和 MUXs。第一行顯示函數 sliding_win_1in2out 有兩個 template，分別為 T 和參數 LEN，在上一級的函數分別被設定為 din_t 和 1024，也就是說，這個函數的 codes 以下用 T 宣告的變數，其資料型態都和此函數的輸入訊號相同。接者首先從第十六行到第二十二行開始解釋，第十九行的指令如下:

```
din_val = din.read();
```

在更前面的第七行有用 T 宣告 din_val，因此 din_val 的資料型態和 din 相同，因此 din_val 可以順利接收這個函數的輸入訊號。此外，因為這個函數的輸入訊號協定採用 AXI Stream，所以每讀一次，其內部被讀出來的資料就會消失，下

一次在讀的時候就會直接讀到下一筆資料，因此這個迴圈的 iteration 意義在於每一個 initiation interval 接收一次從輸入 port 進來的輸入訊號，也就是說輸入訊號連續進來。接著 din_val 被指派到 no_delay 向量。此外，delay_line.shift() 有點類似真實電路中的 shift registers，並且長度為 512，也就是說 din_val 在進入 shift register 之後過了 512 個 II 之後才會傳給 delayed 向量。

第十行和第十一行所使用的指令如下:

#pragma HLS ARRAY_PARTITION variable=xxx cyclic factor=2

這個指令的意義在上面已經說明，不再贅述。在這裡需要使用這個指令的原因是因為這個函數內部有平行化，並且展開 2 倍的硬體資源，所以我們必須要將變數向量設定成可以分開兩塊，使得後續 HLS 可以順利合成。

第二十三行到第三十四行的程式主要目的為將訊號進行分流，其中第二十五行的指令如下:

#pragma HLS UNROLL factor=2

此指令呼應上述的 ARRAY_PARTITION 指令，在這個 for 迴圈我們平行化兩倍，硬體資源使用會多出將近兩倍，但是同時 latency 也會減少將近兩倍。而 unrolled 之後的 for 迴圈訊號被執行的順序仍然為遞增，只是每次增加的值改為 2，而且 for 迴圈會同時執行兩組資料。觀察第二十八行到第三十二行可以發現，這部分的迴圈是先把 delayed 向量的資料輸出，再將 nodelay 向量的資料輸出，由此我們可以推論出這個函數會先消耗 256 個 II，每個 II 輸出 delayed[2i]和 delayed[2i+1]，並且 i 從 0 到 255，再來同樣消耗 256 個 II，每個 II 輸出 nodelay[2i]

和 nodelay[2i+1]。

因此，我們可以推論出為什麼第十二行和第十三行設定數值所代表的意義。兩個指令如下:

```
#pragma HLS STREAM depth=512 variable=nodelay
#pragma HLS STREAM depth=256 variable=delayed
```

其中我把常數變數改成對應數值方便討論。訊號 nodelay 需要設定 FIFO 深度為 512 的原因是因為訊號 nodelay 會比 delayed 晚 512 個 for 迴圈 iterations 被輸出，因此我們需要為 nodelay 加裝深度為 512 的 FIFO。而 delayed 的 FIFO 深度設定為 256 的原因是因為這個函數第一個迴圈求出整組 delayed 向量所需要的 iteration 數為 512，第二個迴圈所需要的 iteration 數同樣也是 512，但是後者比前者多出 unroll，因此實際上在真實電路中只需要 256 個 II 就可以算出整組訊號向量，也就是說在最糟的情況下，後者會比前者快 256 個 iteration，因此中間需要安插長度為 256 的 FIFO。

第十四行、第十五行和第二十六行分別有三個優化用的指令，三個指令如下:

```
#pragma HLS INLINE
#pragma HLS DATAFLOW
#pragma HLS pipeline rewind
```

INLINE 的用途為解除程式的分層效應，使得硬體優化少更多限制。DATAFLOW 前面已經提過，pipeline rewind 的用途則是盡量增加 pipeline 運算效率。

	The core codes of the function window_fn
1.	template<class TI, class TO, class TC, int SZ, win_fn_t FT, int UF>

```
2.  void window_fn(TI *indata, TO *outdata) {
3.      TC coeff_tab[SZ];
4.      init_coef_tab<TC,SZ,FT>(coeff_tab);
5.  #pragma HLS ARRAY_PARTITION variable=coeff_tab cyclic factor=UF
6.  apply_win_fn:
7.      for (unsigned i = 0; i < SZ; i++) {
8.  #pragma HLS UNROLL factor=UF
9.  #pragma HLS PIPELINE rewind
10.         outdata[i] = coeff_tab[i] * indata[i];
11.     }
12. }
```

圖八

函數 window_fn 的功能為將輸入訊號乘以窗函數的係數。第四行為初始化窗函數的係數，雖然將固定指派固定參數的指令寫在會被重複執行的函數裡面在軟體模擬會消耗更多時間，可是因為這個函數最後會被生成出對應的電路，所以一定要把參數寫在函數內。第五行則因為第八行的 unroll，將參數的 array 做 partition，使得最後能合成出平行化的電路。第七行到第十一行就是乘法迴圈，第八行的指令使得這部分的電路被平行化兩倍，也就是說這裡會有兩組乘法器同時運作，提高電路吞吐量。

最後回到圖五的第十六行到第二十五行，這部分將從 window_fn 輸出的訊號輪流分堆到 FFT 輸入訊號的實部和虛部。第二十二行則呼應 AXI Stream 協定的 last 訊號，當最後一對資料被輸出時會跳為 1，告訴下一級電路已經輸出完畢這一輪所有訊號。

```
    The main function of the testbench hls_realfft_test.cpp
1.  int main(void) {
```

```
2.      int err_cnt = 0;
3.      short din_val = 0;
4.      din_t * const signal_buf = new din_t [REAL_FFT_LEN];
5.      hls::stream<din_t> frontend_din("fe_din");
6.      hls::stream<xfft_axis_t> frontend_dout("fe_dout");
7.      dout_t * const fft_din = new dout_t [REAL_FFT_LEN/2];
8.      dout_t * const fft_dout = new dout_t [REAL_FFT_LEN/2];
9.      hls::stream<xfft_axis_t> backend_din("be_din");
10.     hls::stream<dout_t> backend_dout("be_dout");
11.     ofstream tvin_ofs("realfft_fe_tvin.dat");
12.     tvin_ofs.fill('0');
13.     ofstream tvout_ofs("realfft_be_tvout.dat");
14.     tvout_ofs.fill('0');
15.     for (int i = 0; i < NUM_TESTS; i++) {
16.         signal_gen(signal_buf, REAL_FFT_LEN / 2);
17.         for (int j = 0; j < REAL_FFT_LEN / 2; j++) {
18.             frontend_din << signal_buf[j];
19.             tvin_ofs.width(DIN_W / 4);
20.             tvin_ofs<< hex << ap_uint<DIN_W>(signal_buf[j].range(DIN_W -
21. 1, 0)) << endl;
22.         }
23.         hls_real2xfft(frontend_din, frontend_dout);
24.         for (int j = 0; j < REAL_FFT_LEN / 2; j++) {
25.             xfft_axis_t windowed_samples = frontend_dout.read();
26.             fft_din[j].real(windowed_samples.data.real());
27.             fft_din[j].imag(windowed_samples.data.imag());
28.         }
29.         fft_rad2_dit_nr<DOUT_W, 16>(fft_dout, fft_din, REAL_FFT_LEN /
30. 2, false);
31.         for (int j = 0; j < REAL_FFT_LEN / 2; j++) {
32.             xfft_axis_t fft_axis_out;
33.             fft_axis_out.data.real(fft_dout[j].real());
34.             fft_axis_out.data.imag(fft_dout[j].imag());
35.             fft_axis_out.last = j == REAL_FFT_LEN / 2 - 1 ? 1 : 0;
36.             backend_din << fft_axis_out;
37.         }
38.         hls_xfft2real(backend_din, backend_dout);
39.         for (int j = 0; j < REAL_FFT_LEN / 2; j++) {
```

```
40.                 dout_t dout = backend_dout.read();
41.                 float re = dout.real().to_float();
42.                 float im = dout.imag().to_float();
43.                 real32_t mag = sqrt(re * re + im * im);
44.                 ap_uint<2*DOUT_W> tv_dout = (dout.imag().range(DOUT_W -
45. 1, 0),
46.                         dout.real().range(DOUT_W - 1, 0));
47.                 tvout_ofs.width(2 * DOUT_W / 4);
48.                 tvout_ofs << hex << tv_dout << endl;
49.                 if (i == NUM_TESTS - 1) {
50.                     printf("%4d:\t{ %9.6f, %9.6f }; mag = %8.6f\n", j, re, im, mag);
51.                 }
52.             }
53.             fflush(stdout);
54.             cout << endl;
55.         }
56.         tvin_ofs.close();
57.         tvout_ofs.close();
58.         delete [] signal_buf;
59.         delete [] fft_din;
60.         delete [] fft_dout;
61.         cout << "*** TEST COMPLETE ***" << endl << endl;
62.         return err_cnt;
63. }
```

圖九

圖九的程式為 HLS 模擬 CSim 和 CoSim 時候所使用的 testbench 檔案，整體來說類似於原本我們在軟體開發程式的 main function，所以 HLS 在編譯的時候會判定這個檔案為主函式，進而開始執行其他子函式。首先一開始產生測資的函式為 signal_gen()，signal_gen 的內部程式如下。

	The core codes of the function signal_gen
1.	void signal_gen(din_t *signal, int num_samples) {
2.	enum {NUM_FREQ = 5};

```
3.      struct freq_comp_data {
4.          double cycles_per_win;
5.          double phase;
6.          double amplitude;
7.      } freq_set[NUM_FREQ] = {
8.              {497.0, 0.7, 0.8}, {235.0, 1.6, 1.0}, {100.0, 0.0, 0.6},
9.              {35.0, 0.0, 0.8}, {5.0, 0.0, 0.9}
10.     };
11.     static uint64_t t = 0;
12.     for(int i = 0; i < num_samples; i++) {
13.         double sum_freq = 0.0, sum_ampl = 0.0;
14.         for (int j = 0; j < NUM_FREQ; j++) {
15.             sum_freq += freq_set[j].amplitude *
16.                 cos(2.0 * M_PI * freq_set[j].cycles_per_win * t / (2 *
17. num_samples));
18.             sum_ampl += freq_set[j].amplitude;
19.         }
20.         din_t sample = ap_fixed<DIN_W, 1, AP_TRN, AP_SAT>(sum_freq /
21. sum_ampl);
22.         signal[i] = sample;
23.         t++;
24.     }
25. }
```

圖十

圖六的程式為 signal_gen()函式內部程式，給定要產生的資料個數，會回傳儲存於 signal 向量的輸入訊號測資。第三行到第十一行是名為 freq_comp_data 的 struct，並且將資料儲存於 freq_set 向量。向量 freq_set 有三個子資料型態，分別代表訊號的頻率、振幅和初始相位。第十三行到第二十五行為產生訊號資料的迴圈，其對應的公式如下:

$$\text{signal}[t] = \frac{\text{sum\_freq}[t]}{\text{sum\_amp}}, \text{ where}$$

$$\text{sum\_freq}[t] = \sum_{i=0}^{4} A[i] \times \cos\left(\frac{2\pi f[i]t}{2 \times 512}\right) \text{ and}$$

$$\text{sum\_amp} = \sum_{i=0}^{4} A[i]$$

也就是說，最後的 signal 向量其實就是上述 freq_set 五種頻率的訊號所組合出來的綜合訊號，並且每一種頻率的訊號成分有不同且固定的振幅，而分母的 sum_amp 功用為正規化訊號值，因為往後輸入訊號的資料型態為 din_t，其數值範圍在[-1,1)之間，所以正規化之後的訊號值可以確保資料數值不會違反後續變數資料型態的規定。此外，f[i]不是真的訊號的頻率，精確來說，f[i]定義為每經過一個 window length(1024)，此訊號的 cos 波經過共多少週期，因此在圖十的第四行變數名稱為 cycles_per_win。另外，整個函數似乎沒有使用到初始相位，如果要使用可以在 cos()的()內加上初始相位。

回到圖九，第二十三行呼叫函數 hls_real2xfft()，並且()內有兩個 arguments，對應到 real2xfft 的輸入，因此在第五行宣告的時候使用 din_t 資料型態，至於則對應到 real2xfft 的輸出，同樣地，第六行宣告的時候使用 xfft_axis_t 資料型態，也就是複數版的 din_t 加上 last 控制訊號。

執行完 hls_real2xfft()之後，接著是第二十四行到第二十八行的 for 迴圈，因為 fft 的輸入不需要使用到控制訊號 last，所以我們在這裡將 frontend_dout 輪流分配到 fft_din 向量的實部和虛部，fft_din 的資料型態為 dou_t，因此可以使用複數資料型態可以使用的函數 real()和 imag()來指派數值給 fft_din。

第二十九行到第三十行則為 fft 的函數，其呼叫指令如下:

```
fft_rad2_dit_nr<DOUT_W, 16>(fft_dout, fft_din, REAL_FFT_LEN / 2, false);
```

第三個 argument 為 FFT 的 size，而第四個 argument 則為設定是否為 IFFT，這次 lab 採用 FFT 所以設定 false。函數 fft_rad2_dit_nr 的內部程式如下圖十一。

	The core codes of the function fft_rad2_dit_nr
1.	template<int IOW, int TW>
2.	void fft_rad2_dit_nr(
3.	std::complex<ap_fixed<IOW, 1> > *y_out,
4.	std::complex<ap_fixed<IOW, 1> > *x_in,
5.	const uint32_t n_pts, bool ifft) {
6.	std::complex<ap_fixed<TW, 1, AP_TRN, AP_SAT> > * const wtmp =
7.	new std::complex<ap_fixed<TW, 1, AP_TRN, AP_SAT> > [n_pts / 2];
8.	gen_twiddles<ap_fixed<TW, 1, AP_TRN, AP_SAT> >(wtmp, n_pts);
9.	std::complex<ap_fixed<TW, 1, AP_TRN, AP_SAT> > * const w =
10.	new std::complex<ap_fixed<TW, 1, AP_TRN, AP_SAT> > [n_pts / 2];
11.	rad2_bitrev_sort<std::complex<ap_fixed<TW, 1, AP_TRN, AP_SAT> > >(w,
12.	wtmp, n_pts / 2);
13.	delete [] wtmp;
	uint32_t groups = 1;
	uint32_t dist = n_pts / 2;
	while (groups < n_pts) {
	for (uint32_t k = 0; k < groups; k++) {
	std::complex<ap_fixed<TW, 1> > wk = w[k];
	if (ifft)
	wk.imag(-w[k].imag());
	for(uint32_t j = 2 * k * dist; j < (2 * k + 1) * dist; j++) {
	std::complex<ap_fixed<IOW, 1> > y_r = groups == 1 ?
	x_in[j] : y_out[j];
	std::complex<ap_fixed<IOW, 1> > z_r = groups == 1 ?

```
                        x_in[j + dist] : y_out[j + dist];
                ap_fixed<IOW + 1, 2> a = wk.real() * z_r.real() -
                        wk.imag() * z_r.imag();
                ap_fixed<IOW + 1, 2> b = wk.real() * z_r.imag() +
                        wk.imag() * z_r.real();
                y_out[j + dist].real((y_r.real() - a) / 2);
                y_out[j + dist].imag((y_r.imag() - b) / 2);
                y_out[j].real((y_r.real() + a) / 2);
                y_out[j].imag((y_r.imag() + b) / 2);
            }
        }
        groups *= 2;
        dist /= 2;
    }
    delete [] w;
}
```

33. 34. 35. 36. 37. 38. 39. 40. 41. 42. 43.	

圖十一

圖十一的程式為 fft 演算法的程式。第六行到第十六行所有指令的目的為創造出向量 w，並且儲存 fft 的 twiddle factors，因為要配合下面程式的使用，因此 w 其實為 w_tmp 向量經過 bit-reverse 之後的向量，而 w_tmp 向量內數值產生的公式如下:

$$\text{w\_tmp} = \cos\left(\frac{2\pi}{1024} \times \text{i}\right) - \text{j}\sin\left(\frac{2\pi}{1024} \times \text{i}\right) = \text{e}^{-\text{j}\frac{2\pi}{1024}\times\text{i}} \quad \forall \ 0 \le \text{i} < 512$$

並且此公式寫在 gen_twiddles()函數內，用來產生 w_tmp 的數值。第十七行到第四十一行的程式為 FFT 演算法的核心部分，它的作法採用 butterfly 由 FFT

size 的一半到最後只有 1，並且 y_r 為每一組 butterfly 的上層輸入，或是說不需要進行旋轉的輸入，z_r 則為需要進行旋轉的輸入。訊號 z_r 經過旋轉之後，其實部為 a，虛數為 b，而 z_r 和 a+jb 之間的關係是可以用下列公式表示。

$$\begin{gathered} a + jb = [Re\{wk\} + jIm\{wk\}][Re\{z\} + jIm\{z\}], \\ \text{where } a = Re\{wk\} \times Re\{z\} - Im\{wk\} \times Im\{z\} \; and \\ b = \; Re\{wk\} \times Im\{z\} - Im\{wk\} \times Re\{z\}. \end{gathered}$$

上述公式呼應圖十一的程式第二十九行到第三十二行，得到 a 和 b 值。接著，第三十三行到第三十六行就是 FFT 的 butterfly，其中 dist 變數代表 butterfly 兩組輸入訊號的相對距離，y_out 為 butterfly 之後的輸出，index 比較小者對應到相加後的輸出，反之則為相減後的輸出。也因此，第二十五行到第二十八行需要設定 if-else 判斷，因為第一次執行的時候，輸入來源為上一級的函數輸出，而第二輪執行的時候，butterfly 的輸入為上一級的 butterfly，因此使用 y_out 作為輸入訊號。另外，在計算出 y_out 之前還需要除以 2，原因是因為 FFT 的傅立葉轉換和反轉換需要多除以 FFT size，才能使得兩個轉換矩陣相乘為 identity matrix。在這次 lab A，就是把 N 分之一分數放在 Fourier 轉換這邊而非過往我們所熟悉的反轉換那邊，我推論原因)是因為如果不這麼調整，則數值會超過 din_t 和 dou_t 所規範的數值範圍[-1,1)，因此如此設定很合理。

解釋完 FFT 內部程式之後，回到圖五，第三十一行到第三十七行就是將資料型態為 dou_t 的 FFT 輸出轉為符合下一級輸入的資料型態，因為下一級的函數為 xfft2real，它會被合成為符合 AXI Stream 的電路，因此我們需要將資料型態轉為 xfft_axis_t，也就是將原本的複數多新增 last 控制訊號，以符合下一級函數的

輸入訊號協定。

第三十八行則為 hls_xfft2real()函數，其內部程式如下圖十二。

	The core codes of the function hls_xfft2real
1.	template<typename TI, typename TO, int LOG2_REAL_SZ, bool BITREV>
2.	void xfft2real(
3.	hls::stream<xfft_axis_t>& din,
4.	hls::stream<TO>& dout) {
5.	enum {REAL_SZ = (1 << LOG2_REAL_SZ)};
6.	TI descramble_buf[REAL_SZ/2];
7.	#pragma HLS ARRAY_PARTITION block factor=2
8.	variable=descramble_buf
9.	#pragma HLS INLINE
10.	const complex<coeff_t> twid_rom[REAL_SZ/2] = {
11.	#include “w_rom_1k_init.txt”
12.	};
13.	realfft_be_buffer:
14.	for (int i = 0; i < REAL_SZ / 2; i++) {
15.	#pragma HLS PIPELINE rewind
16.	xfft_axis_t tmp = din.read();
17.	ap_uint<LOG2_REAL_SZ-1> dst_addr = i;
18.	if (BITREV)
19.	dst_addr = dst_addr.range(0, LOG2_REAL_SZ – 2);
20.	descramble_buf[dst_addr] = tmp.data;
21.	}
22.	realfft_be_descramble:
23.	for (int i = 0; i < REAL_SZ / 2; i++) {
24.	#pragma HLS PIPELINE
25.	TI y1 = descramble_buf[i];
26.	TO cdata;
27.	if (i == 0) {
28.	cdata = TO((y1.real() + y1.imag()), (y1.real() – y1.imag()));
29.	} else {
30.	TI y2 = conj(descramble_buf[(REAL_SZ / 2) – i]);
31.	TI f(((y1.real() + y2.real()) / 2), ((y1.imag() + y2.imag()) / 2));
32.	TI g(((y1.imag() – y2.imag()) / 2), ((y2.real() – y1.real()) / 2));
33.	TO wg = TO(twid_rom[i]) * TO(g);

```
34.                     cdata = f + wg;
35.                 }
36.                 dout << cdata;
37.         }
38. }
```

圖十二

圖十二的程式為作為 descrambler 用之 xfft2real 函數的內部程式。第七行到第八行為宣告變數向量 descramble_buf，作為後面第十八行到第二十一行的 bit-reverse 後的儲存變數向量。第十行到第十二行為 twi_rom 其中在用 include 指令把已經寫好的數值.txt 檔案引用到 project 之內，而此固定參數用途在後面 descrambler 演算法會解釋。第十八行到第二十一行對應到 FFT 的 bit reverse，因為在上述的 FFT 演算法並沒有對最終的輸出訊號做 bit reverse，因此我們需要在 xfft2real 函數做 bit-reverse 的處理，使得輸出訊號的 index 順序能按照頻率大小排列，此外，index 為 0 代表 DC frequency component，而 1、2…對應到在 DC frequency component 右邊的 frequency components。頻譜都是左右對稱，DC frequency component 左邊的 frequency components 則從 511 開始往下數，直到 256 為最大頻率的 component。第二十二行到第三十七行則為 descrambler 的核心演算法，其目的為消除因為 FFT 實部和虛數乘以有時間差的 window function coefficient，而必須對頻譜上的 component 做補償相位差的處理，首先相關公式如下:

已知在 real2xfft 函數內，window_fn 內部訊號分成兩條路線，一條對應到實數，另一條則對應到虛數。基於原本的 code 設定，如果實數乘以 time index 為

k 的 window function coefficient，則與之同一 pair 的虛數會乘以 time index 為 k+1 的 window function coefficient，因此實數和虛數兩者所乘上的 window function 存在時間差，我們可以列式如下:

$$\mathrm{Re}\{\text{FFT input}\} = \mathrm{Re}\{\text{nodelay or delayed}\} \times \mathrm{w}[k]$$
$$\mathrm{Im}\{\text{FFT input}\} = \mathrm{Im}\{\text{nodelay or delayed}\} \times \mathrm{w}[k+1]$$

時域上兩者具有時間差，會導致經過 FFT 轉換之後的兩者在頻域上會有相位差，而 descrambler 的用途為補償此現象所造成的誤差，公式如第一部分。

因此，HLS 大部分的程式已經解釋完畢。

四、分析電路表現

(一) 硬體資源使用情形

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	42	-
FIFO	-	-	-	-	-
Instance	-	4	1638	11909	-
Memory	12	-	192	0	0
Multiplexer	-	-	-	72	-
Register	-	-	8	-	-
Total	12	4	1838	12023	0
Available	280	220	106400	53200	0

Utilization(%)	4	1	1	22	0

圖表一、xfft2real 的硬體資源使用情形

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	26	-
FIFO	4	-	640	312	-
Instance	2	2	881	724	-
Memory	4	-	64	0	0
Multiplexer	-	-	-	36	-
Register	-	-	4	-	-
Total	10	2	1589	1098	0
Available	280	220	106400	53200	0
Utilization(%)	3	~0	1	2	0

圖表二、real2xfft 的硬體資源使用情形

(二) Cycle 數

Latency(cycles)		Latency(absolute)		Interval		Pipeline
min	max	min	Max	min	Max	Type
1549	1552	6.196	6.208	512	512	dataflow

圖表三、real2xfft 的 latency

Latency(cycles)		Latency(absolute)		Interval		Pipeline
min	max	min	Max	min	max	Type
1041	1042	4.164	4.168	530	530	dataflow

圖表四、xfft2real 的 latency

圖表三和圖表四的 latency 符合下面 CoSim 的模擬波形圖，不過在這裡計算方式為完整把一整組向量訊號計算完所需要的 cycle 數，因此要加上一個 interval，這個表格中的 interval 就是 initiation interval。

Latency(cycles)		Latency(absolute)		Interval		Pipeline
min	max	min	Max	min	max	Type
1051	1054	6.204	6.216	**516**	**516**	dataflow

圖表五、xfft2real 的 latency(二)

如果將下列 sliding_win_1in2out 函數內的被紅色粗體字標記的#pragma HLS pipeline rewind 拿掉的話，這個 for 迴圈會多出來空閒的 cycles，使得一組變數向量的 interval 總合多出 4 個 cycle，使得最後電路的吞吐量稍微降低，因此以吞路量來說，這個指令有讓電路吞吐量更大。

	The core codes of the function sliding_win_1in2out
1.	void sliding_win_1in2out(
2.	hls::stream<T>& din,
3.	T *dout) {
4.	enum {DELAY_LEN = LEN / 2};
5.	enum {DELAY_FIFO_DEPTH = DELAY_LEN / 2};
6.	T din_val;
7.	static ap_shift_reg<T, DELAY_LEN> delay_line;
8.	T nodelay[LEN/2], delayed[LEN/2];
9.	#pragma HLS ARRAY_PARTITION variable=nodelay,delayed cyclic
10.	factor=2
11.	#pragma HLS STREAM depth=DELAY_LEN variable=nodelay
12.	#pragma HLS STREAM depth=DELAY_FIFO_DEPTH variable=delayed
13.	#pragma HLS INLINE
14.	#pragma HLS DATAFLOW
15.	sliding_win_delay:
16.	for (int i = 0; i < LEN / 2; i++) {

```
17.  #pragma HLS pipeline rewind
18.          din_val = din.read();
19.          nodelay[i] = din_val;
20.          delayed[i] = delay_line.shift(din_val);
21.      }
22.  sliding_win_output:
23.      for (int i = 0; i < LEN; i++) {
24.  #pragma HLS UNROLL factor=2
25.  #pragma HLS pipeline rewind
26.          T dout_val;
27.          if (i < LEN / 2) {
28.              dout_val = delayed[i];
29.          } else {
30.              dout_val = nodelay[i % (LEN / 2)];
31.          }
32.          dout[i] = dout_val;
33.      }
34.  }
```

圖十三

Latency(cycles)		Latency(absolute)		Interval		Pipeline
min	max	min	Max	min	max	Type
1051	1053	6.204	6.212	**517**	**517**	dataflow

圖表六、xfft2real 的 latency(三)

如果改將綠色粗體標記的的#pragma HLS pipeline rewind 拿掉的話，則 interval 會上升到 517，其效應和上一個 rewind 差不多，都是讓 interval 的 cycle 數稍微提升。

(三)模擬結果

(1) CSim 模擬結果

參考上述圖十，freq_set 共有五個不同頻率的訊號，其五個 cycle_per_window 的值分別為 5、35、47、100、235。因此，我們可以觀察 CSim 印出來的結果

是否符合我們所設定的頻率值。以下為節錄 CSim 模擬之後的.log 檔案。

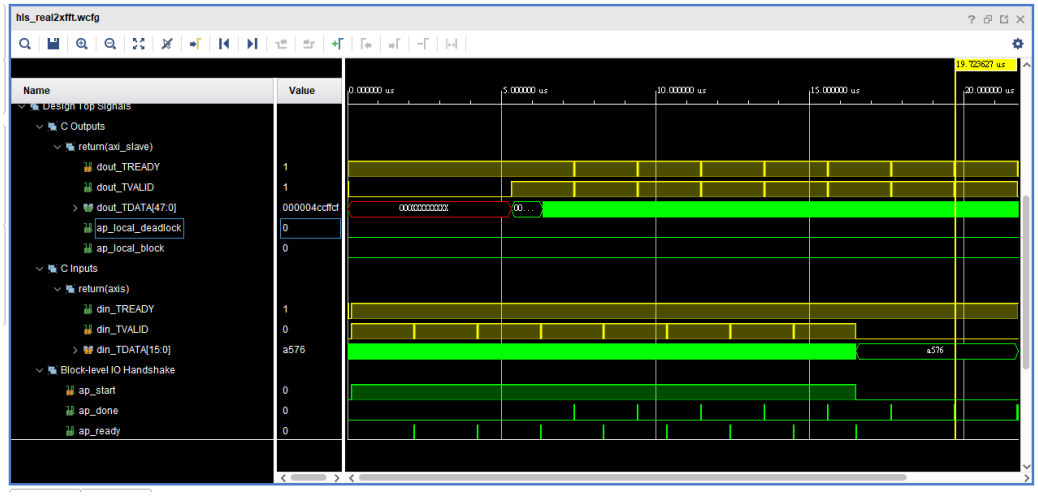
	Some parts of hls_real2xfft_csim.log
…	…
18.	4: { -0.050415, 0.000000 }; mag = 0.050415
19.	5: { 0.118408, 0.000000 }; mag = 0.118408
20.	6: { -0.050415, 0.000000 }; mag = 0.050415
	…
48.	34: { -0.044769, -0.000031 }; mag = 0.044769
49.	35: { 0.105225, -0.000031 }; mag = 0.105225
50.	36: { -0.044800, 0.000000 }; mag = 0.044800
…	…
113.	99: { -0.033600, 0.000000 }; mag = 0.033600
114.	100: { 0.078918, 0.000000 }; mag = 0.078918
115.	101: { -0.033569, -0.000031 }; mag = 0.033569
…	…
248.	234: { -0.056000, -0.000061 }; mag = 0.056000
249.	235: { 0.131592, 0.000031 }; mag = 0.131592
250.	236: { -0.056000, -0.000031 }; mag = 0.056000
…	…
510.	496: { -0.044769, 0.000031 }; mag = 0.044769
511.	497: { 0.105225, 0.000031 }; mag = 0.105225
512.	498: { -0.044800, 0.000031 }; mag = 0.044800
…	…

圖十三

因為除了上述所提到的 frequency 以外，其餘的 mag 值幾乎為 0，因此驗證結果正確。而因為不管是 front-end 還是 back-end，CSim 的模擬結果理論上應該會相同，而實際上兩者.log 檔確實相同，因此只需要放上來其中一個.log 檔的結果即可。

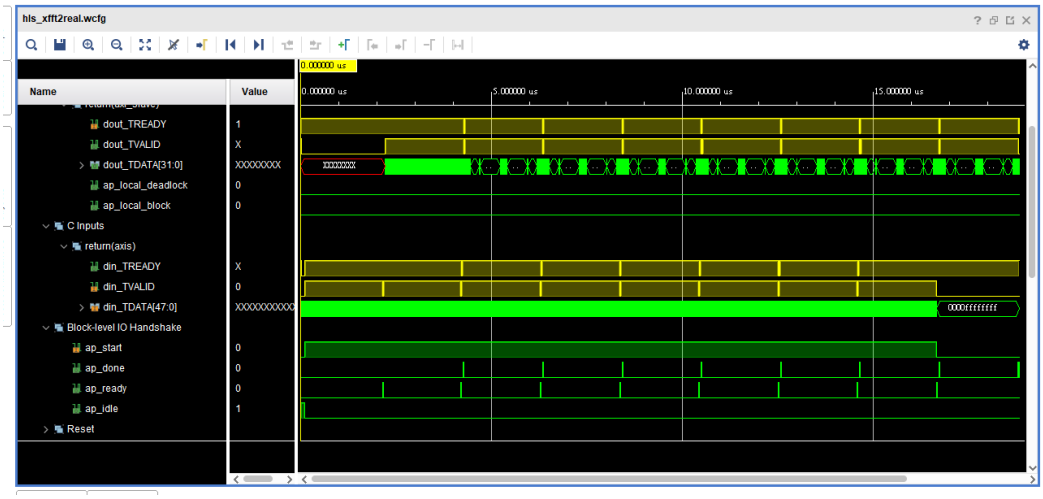
(2) CoSim 模擬結果

首先是 front-end 的 CoSim 模擬波型圖，波型圖如下:



圖十四

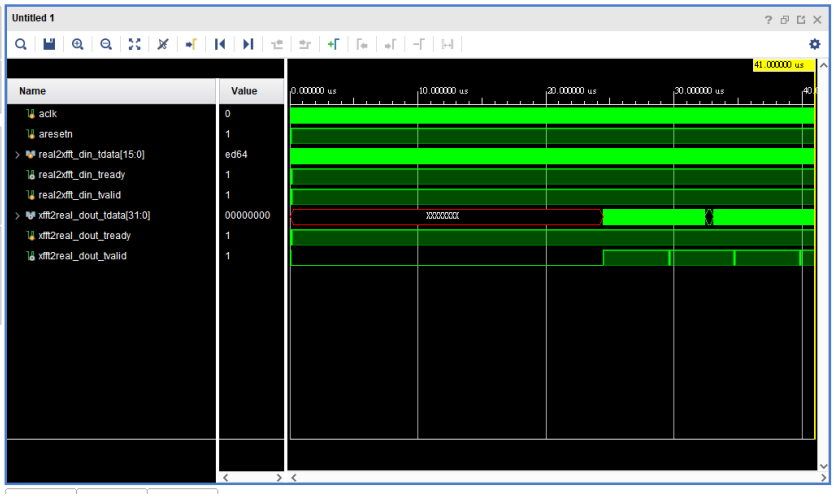
由圖十四的模擬波型可以發現 real2xfft 的 latency 為 1280 cycles，也就是說，當第三組輸入訊號輸入到一半的時候，第一組輸出訊號才開始輸出。



圖十五

圖十五為 xfft2real 的 CoSim 模擬波形圖，latency 為 512 cycles，第二組輸入訊號在輸入的時候，同時第一組輸出訊號也正在輸出。

(3)Vivado testbench 模擬波形圖



圖十六

在圖十六的 Vivado 模擬結果中，可以發現整個系統的輸出在輸出的 valid 跳為 1 的時候開始輸出，而很明顯地可以發現每隔一段時間 valid 就有一瞬間會跳到 0，而兩個 valid 為 0 的間隔之間的輸出對應到一組完整的 FFT 的輸出。此外整個系統從開機到開始有輸出訊號的時間間格約為 35us。

五、參考資料與附件

(一) 參考資料

(1) 老師的線上講義

(2) Lab A 的 workbook

(二)Github 連結: https://github.com/chou111064529/HLS_LABA