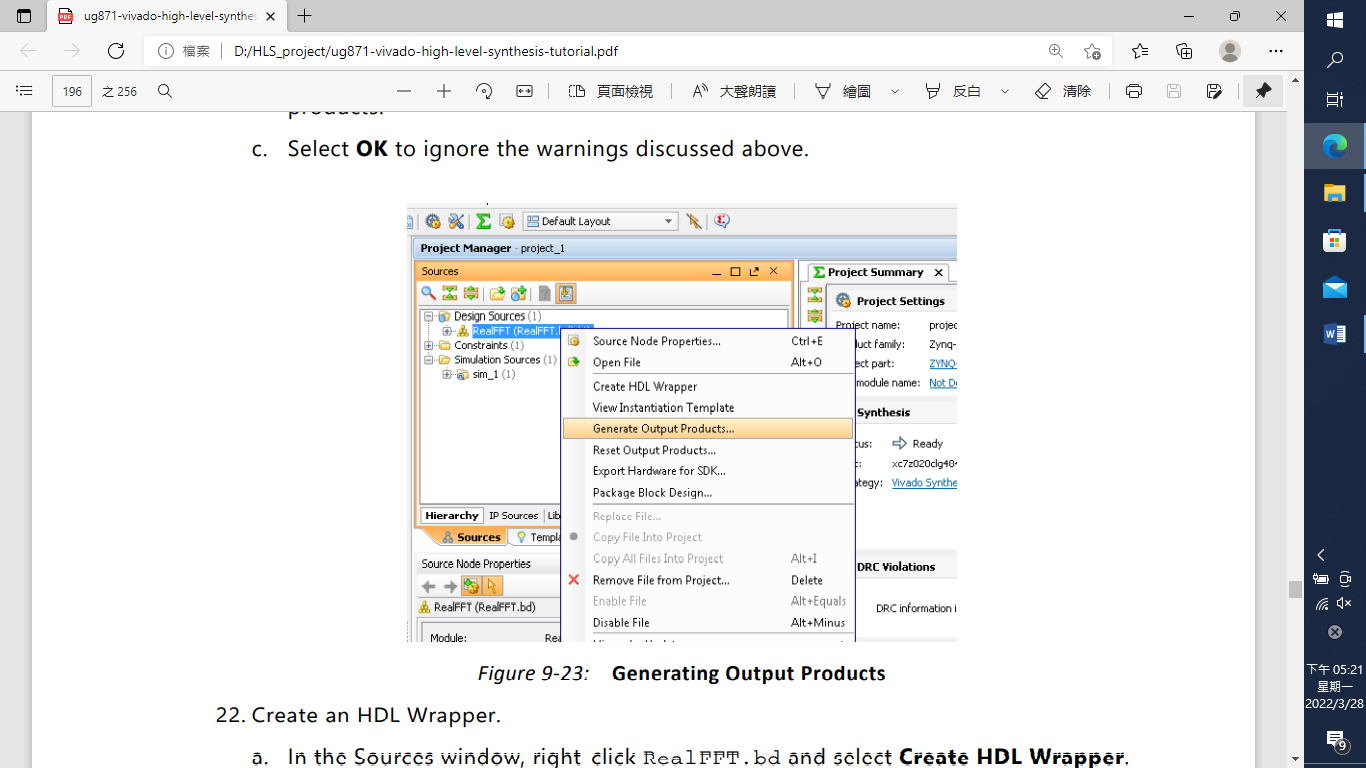
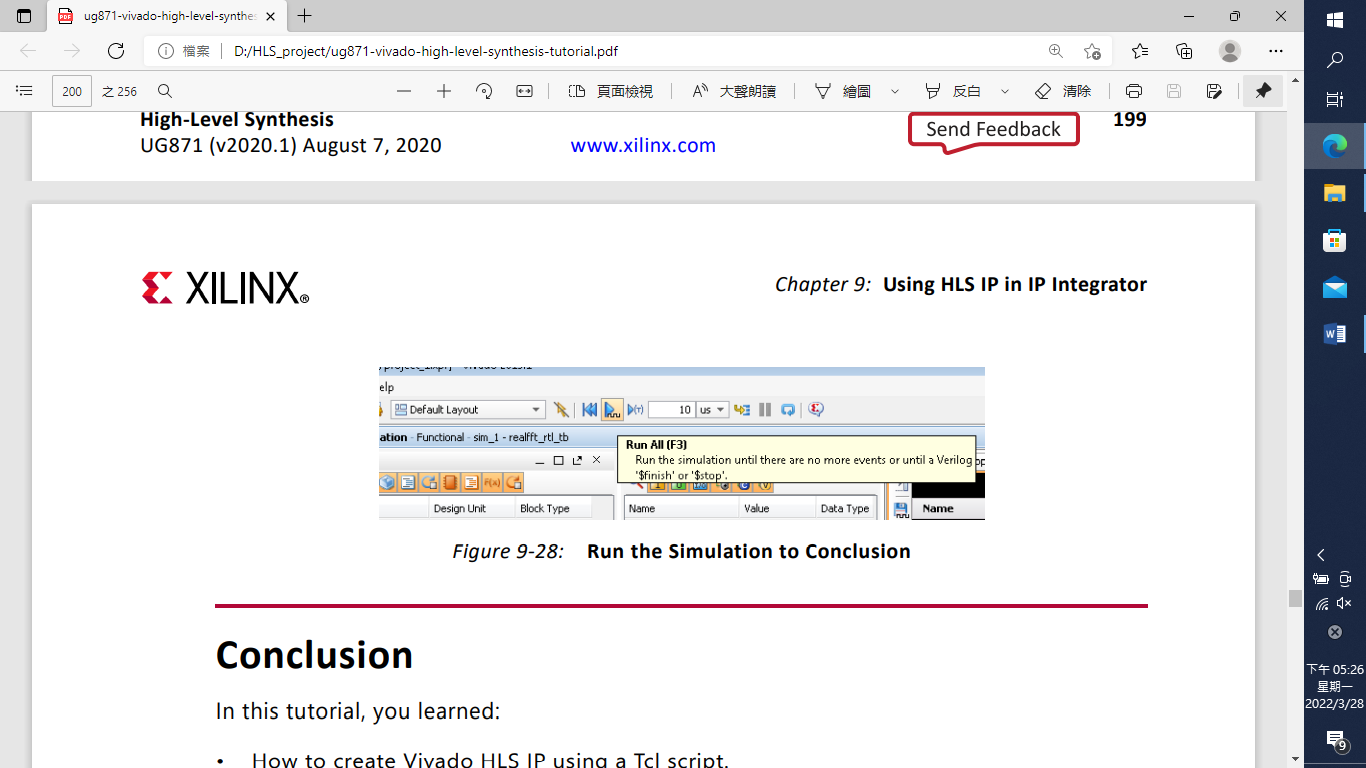
1. Lab操作流程與問題

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

1. 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。

1. Short-Time Fourier Transform (STFT)

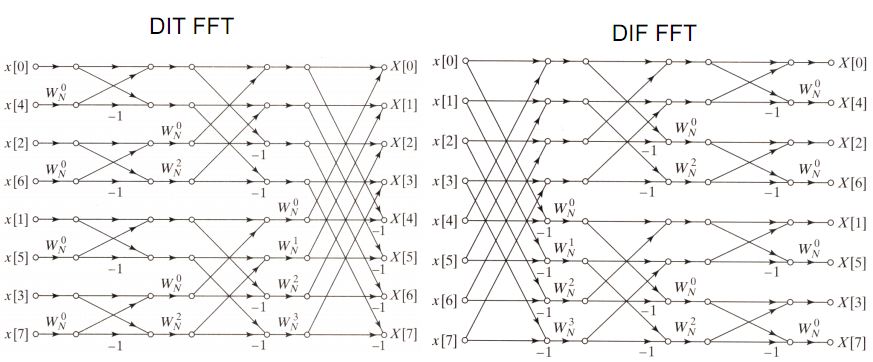
STFT最常見的用途為分析連續長時間的時變頻譜資訊，像是聲音頻率分析或是生醫應用等等，在這一次lab A有用到STFT。STFT在輸入STFT的輸入訊號之前會將輸入訊號乘以window function的加權係數，也就是在時域上將輸入訊號和window function相乘。經過傅立葉轉換之後，相當於原訊號本身的頻譜和window function的頻譜做捲積，我們將上述敘述用數學算式表示:

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。

1. Fast Fourier Transform (FFT)

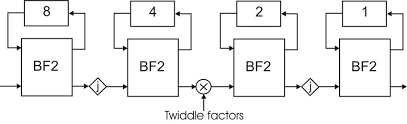
FFT就是計算複雜度比DFT更小卻能算出相同結果的演算法，常常應用在通訊系統硬體上。FFT的原理為將N-point的DFT分成兩個-point的DFT，其中有一個-point的DFT需要另外乘以矩陣元素為twiddle factor的向量，上述敘述的數學式子如下:

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。

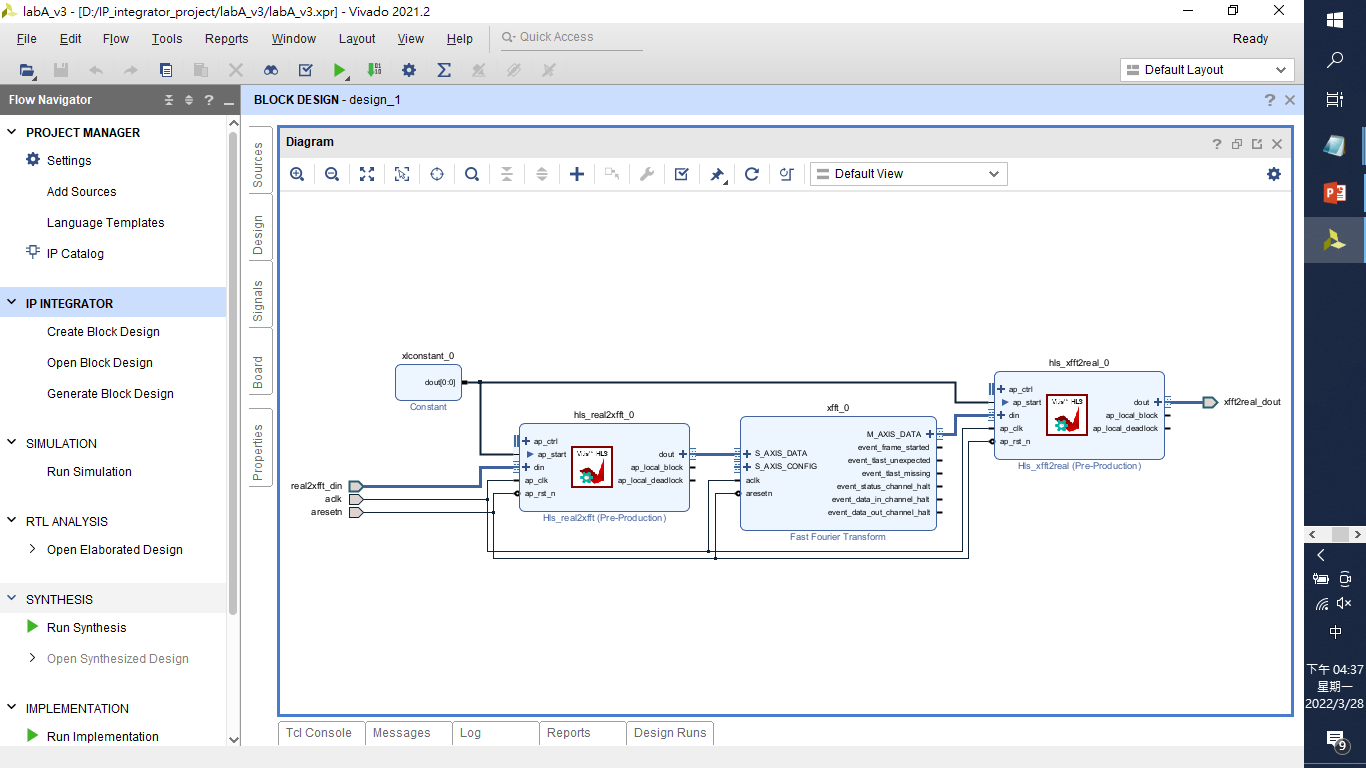
1. Descrambler

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:

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。

1. 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。

1. 程式碼

|  |  |
| --- | --- |
|  | The core codes of the function real2xfft |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23. | void hls\_real2xfft(  hls::stream<din\_t>& din,  hls::stream<xfft\_axis\_t>& dout) {  #pragma HLS INTERFACE axis port=dout  #pragma HLS INTERFACE axis port=din  #pragma HLS ARRAY\_PARTITION variable=data2window cyclic factor=2  #pragma HLS ARRAY\_PARTITION variable=windowed cyclic factor=2  #pragma HLS ARRAY\_STREAM variable=data2window,windowed depth=2  #pragma HLS DATAFLOW  din\_t data2window[REAL\_FFT\_LEN], windowed[REAL\_FFT\_LEN];  sliding\_win\_1in2out<din\_t, REAL\_FFT\_LEN>(din, data2window);  window\_fn<din\_t, din\_t, coeff\_t, REAL\_FFT\_LEN, WIN\_FN\_TYPE, 2>(data2window, windowed);  real2xfft\_output:  for (int i = 0; i < REAL\_FFT\_LEN; i += 2) {  #pragma HLS PIPELINE rewind  dout\_t cdata(windowed[i], windowed[i + 1]);  xfft\_axis\_t fft\_axis\_d;  fft\_axis\_d.data = cdata;  fft\_axis\_d.last = i == REAL\_FFT\_LEN - 2 ? 1 : 0;  dout.write(fft\_axis\_d);  }  } |

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.  2.  3.  4.  5.  6.  7.  8.  9.  10. | #include <complex>  #include <ap\_fixed.h>  #define DIN\_W 16  #define DOUT\_W DIN\_W  typedef ap\_fixed<DIN\_W, 1> din\_t;  typedef complex<ap\_fixed<DOUT\_W, 1> > dout\_t;  typedef struct {  dout\_t data;  ap\_uint<1> last;  } 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.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35. | template<typename T, int LEN>  void sliding\_win\_1in2out(  hls::stream<T>& din,  T \*dout) {  enum {DELAY\_LEN = LEN / 2};  enum {DELAY\_FIFO\_DEPTH = DELAY\_LEN / 2};  T din\_val;  static ap\_shift\_reg<T, DELAY\_LEN> delay\_line;  T nodelay[LEN/2], delayed[LEN/2];  #pragma HLS ARRAY\_PARTITION variable=nodelay,delayed cyclic factor=2  #pragma HLS STREAM depth=DELAY\_LEN variable=nodelay  #pragma HLS STREAM depth=DELAY\_FIFO\_DEPTH variable=delayed  #pragma HLS INLINE  #pragma HLS DATAFLOW  sliding\_win\_delay:  for (int i = 0; i < LEN / 2; i++) {  #pragma HLS pipeline rewind  din\_val = din.read();  nodelay[i] = din\_val; // din  delayed[i] = delay\_line.shift(din\_val);  }  sliding\_win\_output:  for (int i = 0; i < LEN; i++) {  #pragma HLS UNROLL factor=2  #pragma HLS pipeline rewind  T dout\_val;  if (i < LEN / 2) {  dout\_val = delayed[i];  } else {  dout\_val = nodelay[i % (LEN / 2)];  }  dout[i] = dout\_val;  }  } |

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.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | template<class TI, class TO, class TC, int SZ, win\_fn\_t FT, int UF>  void window\_fn(TI \*indata, TO \*outdata) {  TC coeff\_tab[SZ];  init\_coef\_tab<TC,SZ,FT>(coeff\_tab);  #pragma HLS ARRAY\_PARTITION variable=coeff\_tab cyclic factor=UF  apply\_win\_fn:  for (unsigned i = 0; i < SZ; i++) {  #pragma HLS UNROLL factor=UF  #pragma HLS PIPELINE rewind  outdata[i] = coeff\_tab[i] \* indata[i];  }  } |

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.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.  37.  38.  39.  40.  41.  42.  43.  44.  45.  46.  47.  48.  49.  50.  51.  52.  53.  54.  55.  56.  57.  58.  59.  60.  61.  62.  63. | int main(void) {  int err\_cnt = 0;  short din\_val = 0;  din\_t \* const signal\_buf = new din\_t [REAL\_FFT\_LEN];  hls::stream<din\_t> frontend\_din("fe\_din");  hls::stream<xfft\_axis\_t> frontend\_dout("fe\_dout");  dout\_t \* const fft\_din = new dout\_t [REAL\_FFT\_LEN/2];  dout\_t \* const fft\_dout = new dout\_t [REAL\_FFT\_LEN/2];  hls::stream<xfft\_axis\_t> backend\_din("be\_din");  hls::stream<dout\_t> backend\_dout("be\_dout");  ofstream tvin\_ofs("realfft\_fe\_tvin.dat");  tvin\_ofs.fill('0');  ofstream tvout\_ofs("realfft\_be\_tvout.dat");  tvout\_ofs.fill('0');  for (int i = 0; i < NUM\_TESTS; i++) {  signal\_gen(signal\_buf, REAL\_FFT\_LEN / 2);  for (int j = 0; j < REAL\_FFT\_LEN / 2; j++) {  frontend\_din << signal\_buf[j];  tvin\_ofs.width(DIN\_W / 4);  tvin\_ofs<< hex << ap\_uint<DIN\_W>(signal\_buf[j].range(DIN\_W - 1, 0)) << endl;  }  hls\_real2xfft(frontend\_din, frontend\_dout);  for (int j = 0; j < REAL\_FFT\_LEN / 2; j++) {  xfft\_axis\_t windowed\_samples = frontend\_dout.read();  fft\_din[j].real(windowed\_samples.data.real());  fft\_din[j].imag(windowed\_samples.data.imag());  }  fft\_rad2\_dit\_nr<DOUT\_W, 16>(fft\_dout, fft\_din, REAL\_FFT\_LEN / 2, false);  for (int j = 0; j < REAL\_FFT\_LEN / 2; j++) {  xfft\_axis\_t fft\_axis\_out;  fft\_axis\_out.data.real(fft\_dout[j].real());  fft\_axis\_out.data.imag(fft\_dout[j].imag());  fft\_axis\_out.last = j == REAL\_FFT\_LEN / 2 - 1 ? 1 : 0;  backend\_din << fft\_axis\_out;  }  hls\_xfft2real(backend\_din, backend\_dout);  for (int j = 0; j < REAL\_FFT\_LEN / 2; j++) {  dout\_t dout = backend\_dout.read();  float re = dout.real().to\_float();  float im = dout.imag().to\_float();  real32\_t mag = sqrt(re \* re + im \* im);  ap\_uint<2\*DOUT\_W> tv\_dout = (dout.imag().range(DOUT\_W - 1, 0),  dout.real().range(DOUT\_W - 1, 0));  tvout\_ofs.width(2 \* DOUT\_W / 4);  tvout\_ofs << hex << tv\_dout << endl;  if (i == NUM\_TESTS - 1) {  printf("%4d:\t{ %9.6f, %9.6f }; mag = %8.6f\n", j, re, im, mag);  }  }  fflush(stdout);  cout << endl;  }  tvin\_ofs.close();  tvout\_ofs.close();  delete [] signal\_buf;  delete [] fft\_din;  delete [] fft\_dout;  cout << "\*\*\* TEST COMPLETE \*\*\*" << endl << endl;  return err\_cnt;  } |

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.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25. | void signal\_gen(din\_t \*signal, int num\_samples) {  enum {NUM\_FREQ = 5};  struct freq\_comp\_data {  double cycles\_per\_win;  double phase;  double amplitude;  } freq\_set[NUM\_FREQ] = {  {497.0, 0.7, 0.8}, {235.0, 1.6, 1.0}, {100.0, 0.0, 0.6},  {35.0, 0.0, 0.8}, {5.0, 0.0, 0.9}  };  static uint64\_t t = 0;  for(int i = 0; i < num\_samples; i++) {  double sum\_freq = 0.0, sum\_ampl = 0.0;  for (int j = 0; j < NUM\_FREQ; j++) {  sum\_freq += freq\_set[j].amplitude \*  cos(2.0 \* M\_PI \* freq\_set[j].cycles\_per\_win \* t / (2 \* num\_samples));  sum\_ampl += freq\_set[j].amplitude;  }  din\_t sample = ap\_fixed<DIN\_W, 1, AP\_TRN, AP\_SAT>(sum\_freq / sum\_ampl);  signal[i] = sample;  t++;  }  } |

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:

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.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.  37.  38.  39.  40.  41.  42.  43. | template<int IOW, int TW>  void fft\_rad2\_dit\_nr(  std::complex<ap\_fixed<IOW, 1> > \*y\_out,  std::complex<ap\_fixed<IOW, 1> > \*x\_in,  const uint32\_t n\_pts, bool ifft) {  std::complex<ap\_fixed<TW, 1, AP\_TRN, AP\_SAT> > \* const wtmp =  new std::complex<ap\_fixed<TW, 1, AP\_TRN, AP\_SAT> > [n\_pts / 2];  gen\_twiddles<ap\_fixed<TW, 1, AP\_TRN, AP\_SAT> >(wtmp, n\_pts);  std::complex<ap\_fixed<TW, 1, AP\_TRN, AP\_SAT> > \* const w =  new std::complex<ap\_fixed<TW, 1, AP\_TRN, AP\_SAT> > [n\_pts / 2];  rad2\_bitrev\_sort<std::complex<ap\_fixed<TW, 1, AP\_TRN, AP\_SAT> > >(w,  wtmp, n\_pts / 2);  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;  } |

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:

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。

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.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.  37.  38. | template<typename TI, typename TO, int LOG2\_REAL\_SZ, bool BITREV>  void xfft2real(  hls::stream<xfft\_axis\_t>& din,  hls::stream<TO>& dout) {  enum {REAL\_SZ = (1 << LOG2\_REAL\_SZ)};  TI descramble\_buf[REAL\_SZ/2];  #pragma HLS ARRAY\_PARTITION block factor=2 variable=descramble\_buf  #pragma HLS INLINE  const complex<coeff\_t> twid\_rom[REAL\_SZ/2] = {  #include “w\_rom\_1k\_init.txt”  };  realfft\_be\_buffer:  for (int i = 0; i < REAL\_SZ / 2; i++) {  #pragma HLS PIPELINE rewind  xfft\_axis\_t tmp = din.read();  ap\_uint<LOG2\_REAL\_SZ-1> dst\_addr = i;  if (BITREV)  dst\_addr = dst\_addr.range(0, LOG2\_REAL\_SZ – 2);  descramble\_buf[dst\_addr] = tmp.data;  }  realfft\_be\_descramble:  for (int i = 0; i < REAL\_SZ / 2; i++) {  #pragma HLS PIPELINE  TI y1 = descramble\_buf[i];  TO cdata;  if (i == 0) {  cdata = TO((y1.real() + y1.imag()), (y1.real() – y1.imag()));  } else {  TI y2 = conj(descramble\_buf[(REAL\_SZ / 2) – i]);  TI f((( y1.real() + y2.real()) / 2), ((y1.imag() + y2.imag()) / 2));  TI g((( y1.imag() – y2.imag()) / 2), ((y2.real() – y1.real()) / 2));  TO wg = TO(twid\_rom[i]) \* TO(g);  cdata = f + wg;  }  dout << cdata;  }  } |

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:

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。

1. 分析電路表現
2. 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

1. Cycle數

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

圖表三、real2xfft的latency

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Latency(cycles) | | Latency(absolute) | | Interval | | Pipeline  Type |
| min | max | min | Max | min | max |
| 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  Type |
| min | max | min | Max | min | max |
| 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.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  **17.**  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34. | void sliding\_win\_1in2out(  hls::stream<T>& din,  T \*dout) {  enum {DELAY\_LEN = LEN / 2};  enum {DELAY\_FIFO\_DEPTH = DELAY\_LEN / 2};  T din\_val;  static ap\_shift\_reg<T, DELAY\_LEN> delay\_line;  T nodelay[LEN/2], delayed[LEN/2];  #pragma HLS ARRAY\_PARTITION variable=nodelay,delayed cyclic factor=2  #pragma HLS STREAM depth=DELAY\_LEN variable=nodelay  #pragma HLS STREAM depth=DELAY\_FIFO\_DEPTH variable=delayed  #pragma HLS INLINE  #pragma HLS DATAFLOW  sliding\_win\_delay:  for (int i = 0; i < LEN / 2; i++) {  **#pragma HLS pipeline rewind**  din\_val = din.read();  nodelay[i] = din\_val;  delayed[i] = delay\_line.shift(din\_val);  }  sliding\_win\_output:  for (int i = 0; i < LEN; i++) {  #pragma HLS UNROLL factor=2  **#pragma HLS pipeline rewind**  T dout\_val;  if (i < LEN / 2) {  dout\_val = delayed[i];  } else {  dout\_val = nodelay[i % (LEN / 2)];  }  dout[i] = dout\_val;  }  } |

圖十三

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Latency(cycles) | | Latency(absolute) | | Interval | | Pipeline  Type |
| min | max | min | Max | min | max |
| 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。

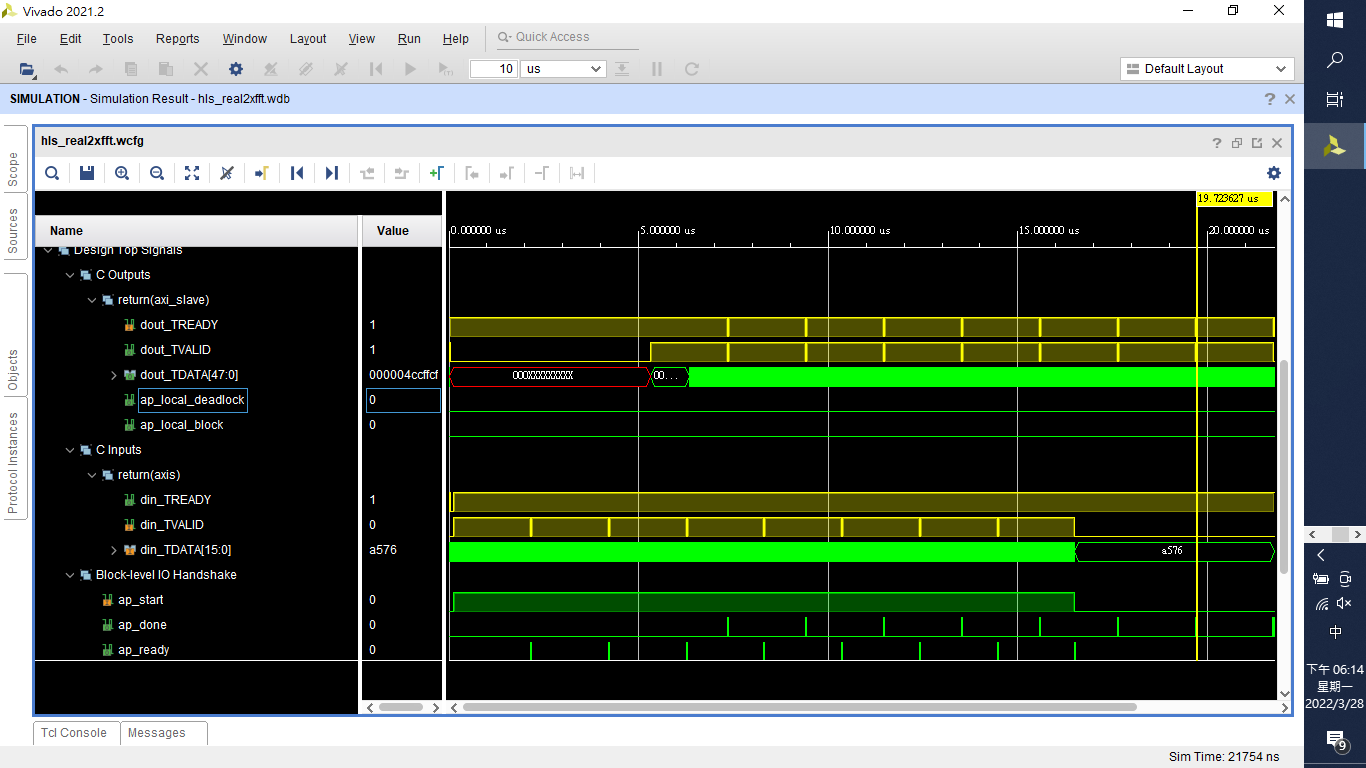
|  |  |
| --- | --- |
|  | Some parts of hls\_real2xfft\_csim.log |
| …  18.  19.  20.  48.  49.  50.  …  113.  114.  115.  …  248.  249.  250.  …  510.  511.  512.  … | …  4: { -0.050415, 0.000000 }; mag = 0.050415  5: { 0.118408, 0.000000 }; mag = 0.118408  6: { -0.050415, 0.000000 }; mag = 0.050415  …  34: { -0.044769, -0.000031 }; mag = 0.044769  35: { 0.105225, -0.000031 }; mag = 0.105225  36: { -0.044800, 0.000000 }; mag = 0.044800  …  99: { -0.033600, 0.000000 }; mag = 0.033600  100: { 0.078918, 0.000000 }; mag = 0.078918  101: { -0.033569, -0.000031 }; mag = 0.033569  …  234: { -0.056000, -0.000061 }; mag = 0.056000  235: { 0.131592, 0.000031 }; mag = 0.131592  236: { -0.056000, -0.000031 }; mag = 0.056000  …  496: { -0.044769, 0.000031 }; mag = 0.044769  497: { 0.105225, 0.000031 }; mag = 0.105225  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。

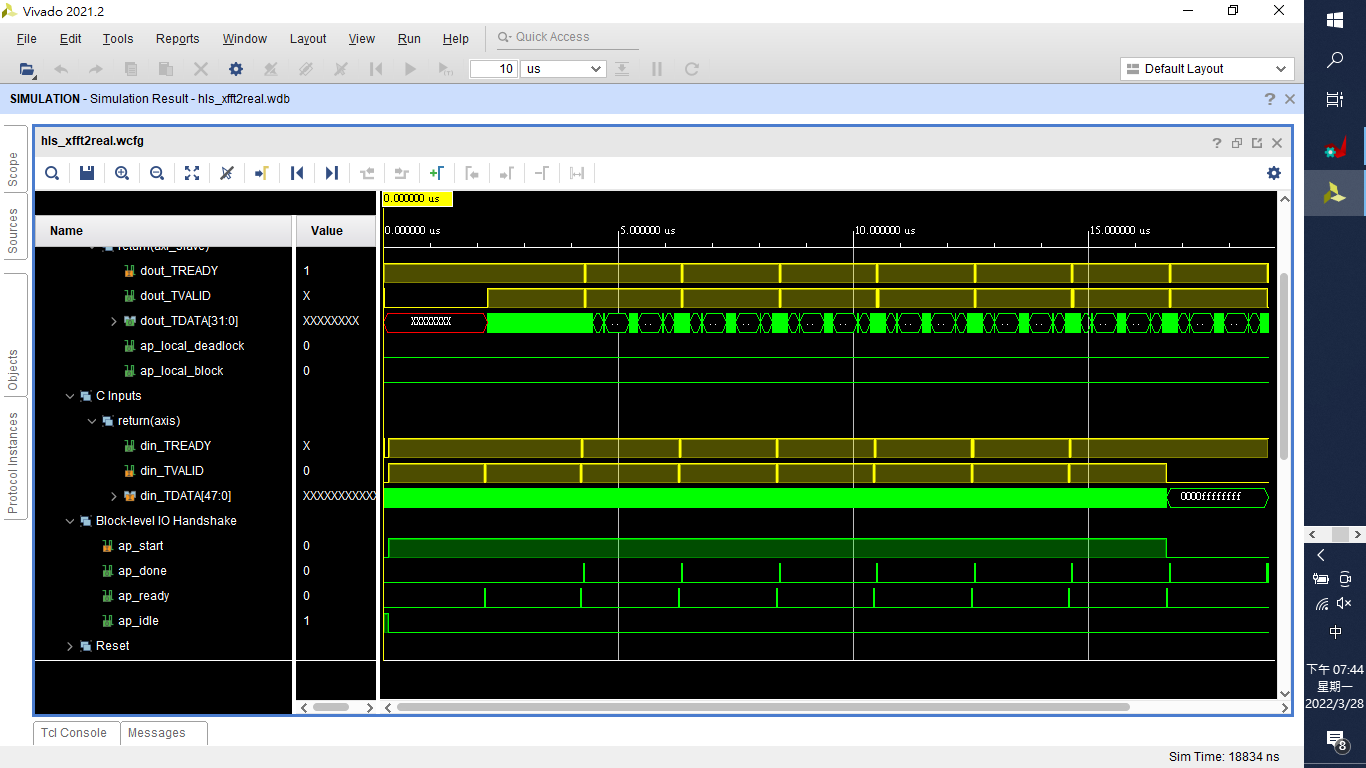
1. CoSim模擬結果

首先是front-end的CoSim模擬波型圖，波型圖如下:



圖十四

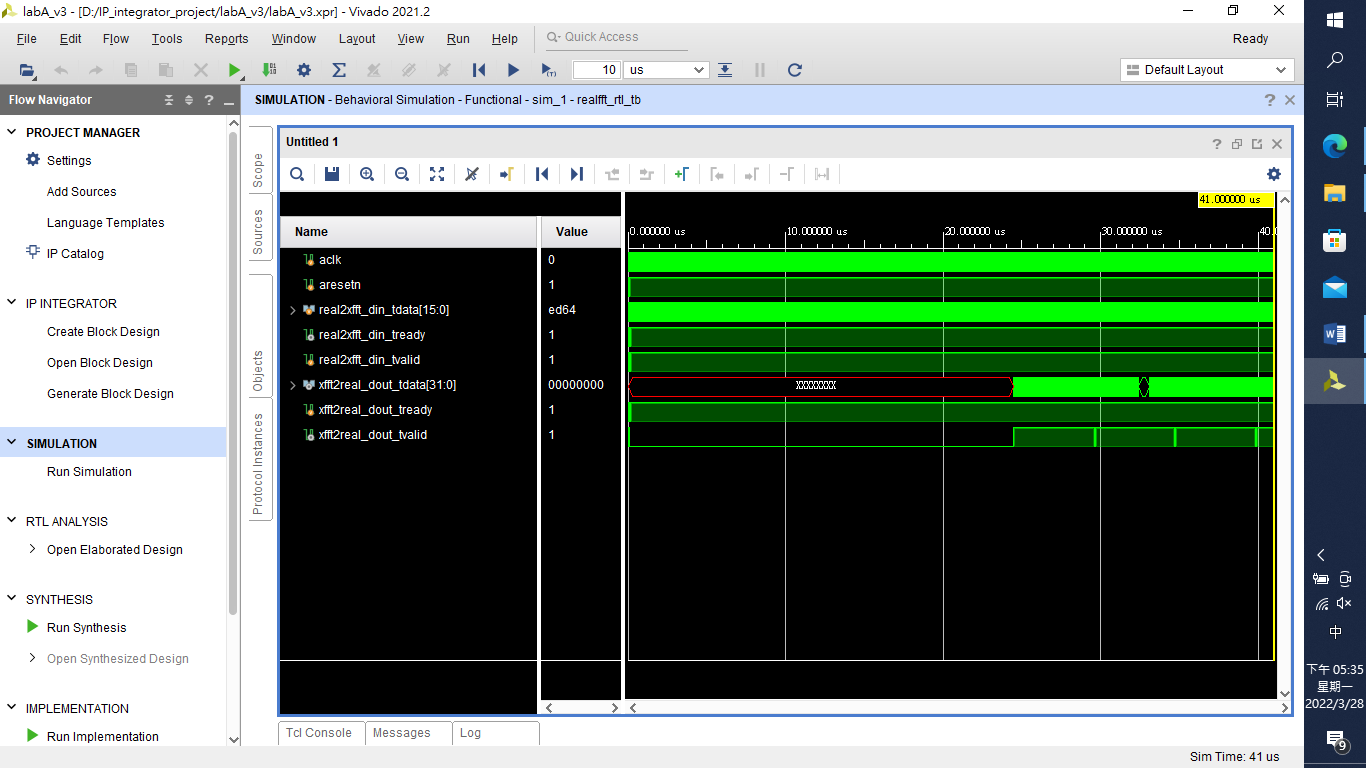
由圖十四的模擬波型可以發現real2xfft的latency為1280 cycles，也就是說，當第三組輸入訊號輸入到一半的時候，第一組輸出訊號才開始輸出。



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。

1. 參考資料與附件
2. 參考資料
3. 老師的線上講義
4. Lab A的workbook

(二)Github連結: https://github.com/chou111064529/HLS\_LABA