Implementing FFT Accelerators with SDSoC™ Using Open-Source Software and C-Callable IP

Introduction

The Fast Fourier Transform (FFT) is one of the fundamental building blocks of Digital Signal Processing (DSP) and Signal Analysis. Due to its frequent use, many device manufacturers offer code libraries and intellectual property (IP) optimized for their architecture in order to achieve the highest possible performance. Xilinx, for example, offers a fully customizable FFT IP core that is optimized for their FPGA, Zynq® SoC, and Zynq MPSoC programmable logic devices. When it comes to general purpose computers there are a few open source code libraries. One such open source library is the Fastest Fourier Transform in the West (FFTW) library which can be obtained from www.fftw.org.

This paper looks at the performance of two different single-precision floating-point FFT implementations using a Xilinx Zynq UltraScale+™ MPSoC device and the SDSoC Development Environment. The first implementation uses version 3.3.7 of the FFTW library compiled for the ARM® Cortex®-A53 processor within a Xilinx ZU3EG device. The second implementation is a FPGA accelerator using the Xilinx LogiCore™ IP FFT version 9.0 (XFFT) running in the programmable logic of the ZU3EG device.

Environment

The Avnet® UltraZed-EG™ Starter Kit was chosen as the test platform for evaluating FFT performance. The UltraZed features the Xilinx Zynq UltraScale+ MPSoC ZU3EG device. The ZU3EG device features a Processing System (PS) containing quad-core ARM Cortex-A53 application processors (APU), dual-core ARM Cortex-R5 real-time processors (RPU), and an ARM Mali™-400 MP2 GPU. Linux was chosen as the host operating system which provides symmetric multi-processing support. For an apples-to-apples comparison, this paper only looks at running the FFTW library on a single APU core running at 1.1GHz.

The ZU3EG device also features FPGA programmable logic (PL) with 360 dedicated DSP slices to support heavy computational workloads. The XFFT running in PL operates at a 300 MHz clock rate using the pipelined-streaming I/O architecture. During execution of the XFFT, the PS offloads processing to the PL and then waits for the results to be computed. This processing model is characteristic of a heterogeneous compute system with a host processor and co-processor accelerator. The Xilinx Zynq UltraScale+ MPSoC architecture combines the heterogeneous system into a single chip solution.

The SDSoC platform for the UltraZed-EG Starter Kit can be downloaded from the zedboard.org website using the link http://www.zedboard.org/content/ultrazed-3eg-starter-kit-sdsoc-platform-sdsoc-20172. The platform is built using the 2017.2 Xilinx tool suite, but can be used with the 2017.4 release without issue. For more details on the UltraZed-EG Starter Kit visit http://www.zedboard.org/product/ultrazed-eg-starter-kit.

Test Setup

Xilinx offers a feature-rich software defined development environment for implementing complex algorithms in embedded processors and FPGA fabric. One such tool is the SDSoC Development Environment. SDSoC stands for Software Defined System-on-Chip and allows for easy acceleration of software defined functions using FPGA programmable logic. SDSoC takes functions defined in C/C++ and moves them to FPGA fabric for acceleration. More details on SDSoC can be found on the Xilinx website at https://www.xilinx.com/sdsoc.

For the purpose of this paper, SDSoC is the perfect environment to run FFT implementations in both software and in FPGA fabric. Thus, a software based test bench was created as the main application calling both the FFTW and XFFT libraries. FFT execution time performance was measured using a built-in high-resolution timer operating at the 1.1GHz APU clock. Figure 1 depicts a block diagram of the SDSoC based test bench.

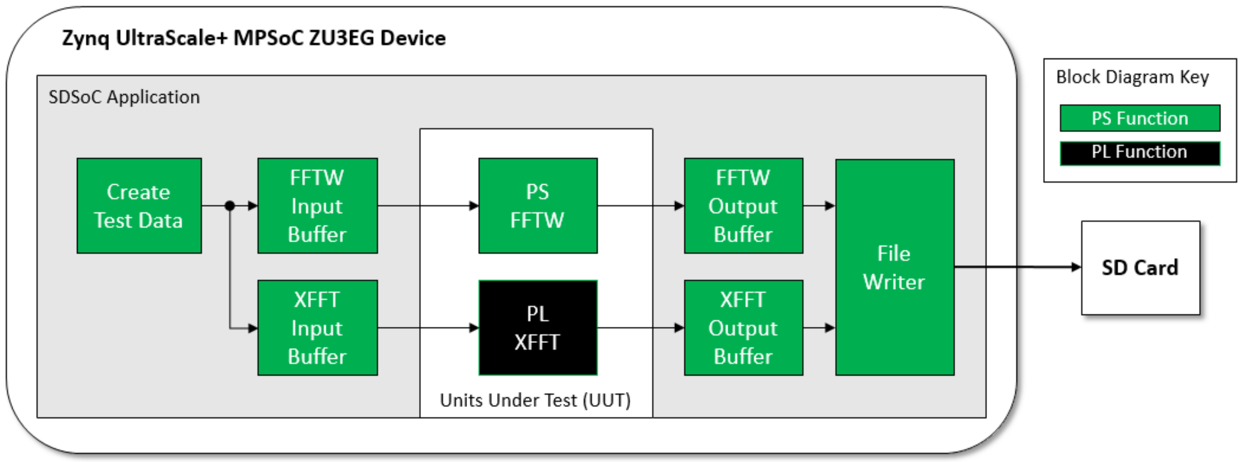


Figure 1 – SDSoC Test bench

Two methods of measuring execution time performance were used for XFFT analysis. The first method measures the average time of a blocking XFFT function call. The second method measures the average time of a non-blocking XFFT function call. Both methods perform 1000 function calls and compute the average execution time per call. With non-blocking function calls the SDSoC application continues execution after calling the PL accelerator without waiting for the accelerator to finish processing. A blocking function call waits for the PL accelerator to complete processing before proceeding. Performing non-blocking function calls helps overcome the overhead of transferring data between the PS and PL because the next set of data can be loaded while the current data is being processed. Figure 2 demonstrates the processing flow for the two measurement methods.

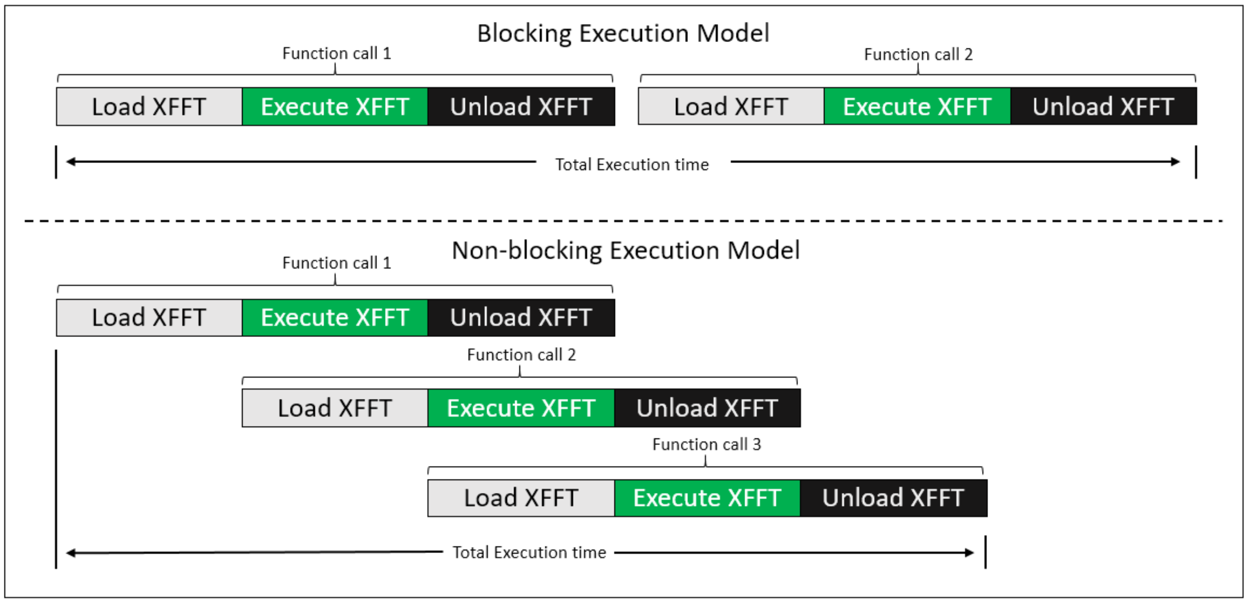


Figure 2 – PL Accelerator Execution Model

Results

Table 1 below gives the average round-trip processing times per function call for the FFTW and XFFT implementations for power-of-two FFT sizes ranging from 8 to 16384. Round-trip processing time includes any overhead associated with data movement between PS and PL.

FFT execution performance using FFTW and XFFT (FFTW in ARM Cortex-A53 @ 1.1GHz; XFFT in Xilinx PL @ 300MHz)			
FFT Size	**FFTW Execution Time (us)**	**Blocking XFFT Execution Time (us)**	**Non-blocking XFFT Execution Time (us)**
8	0.2	2.1	9.3
16	0.3	2.2	9.4
32	0.8	2.6	9.5
64	1.4	3.2	9.7
128	3.0	4.2	9.6
256	8.2	5.9	10.0
512	21.5	9.4	10.3
1024	58.9	16.2	12.9
2048	151.3	30.0	14.2
4096	442.8	57.3	25.2
8192	853.3	112.1	40.0
16384	2215.0	221.6	76.6

Table 1 – FFT Execution Time Performance

As seen in Table 1 above and Figure 3 below, the XFFT implementation has a longer execution time than the FFTW library for FFT sizes smaller than 256. For FFT sizes larger than 256 the XFFT implemented in the PL outperforms the FFTW library running on the ARM processor. Figure 3 also shows XFFT sizes 8 through 32 have approximately the same processing time. This indicates that the overhead of moving the data from the PS to the PL is overshadowing the FFT computation time. Thus, for small FFT sizes it does not make sense to offload a single FFT operation to a PL accelerator.

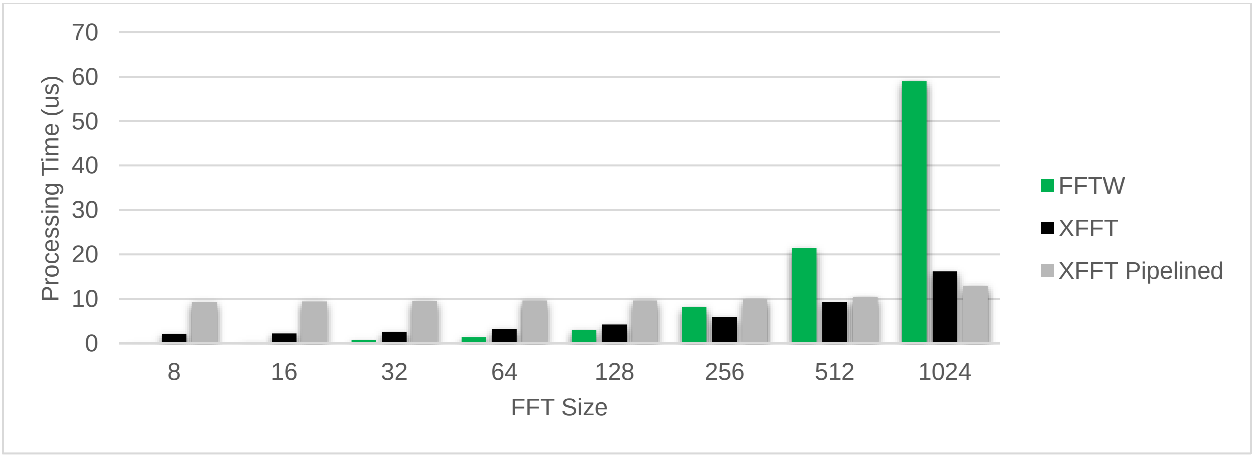


Figure 3 – FFT Processing Times (8 – 1024 point)

Figure 4 shows the processing times for FFT sizes larger than 1024 points. For the 16384 point FFT the processing time for FFTW is 2215 microseconds, the blocking XFFT is 222 microseconds, and the non-blocking XFFT implementation comes in at 77 microseconds. That corresponds to a speed up between 10X and 29X for executing FFT operations in the PL over the ARM processor.

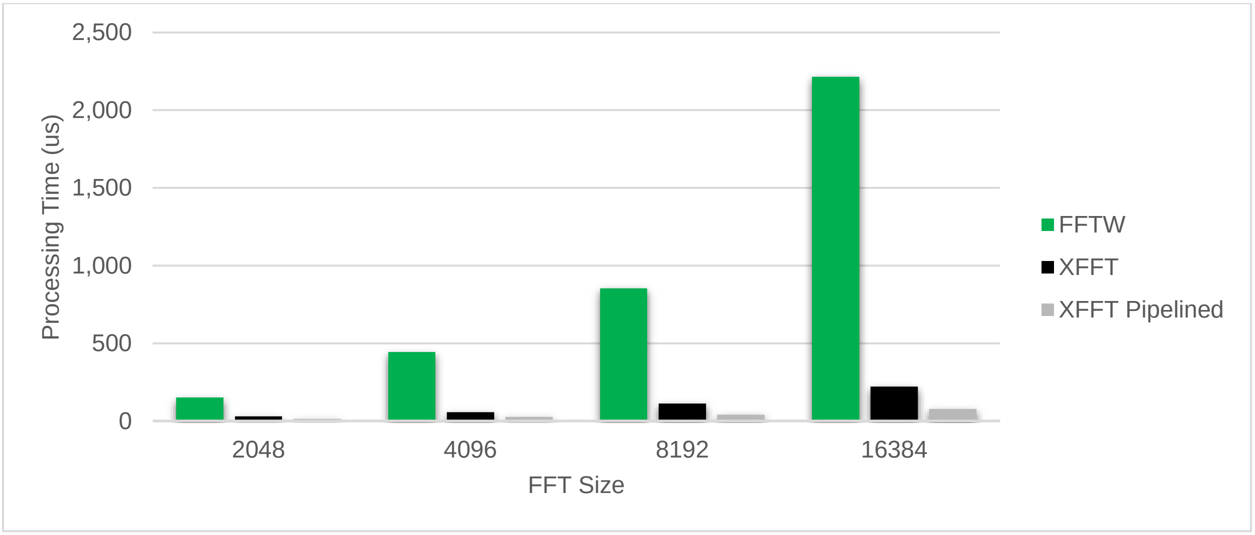


Figure 4 – FFT Processing Times (2048 – 16384 point)

As mentioned previously, smaller FFT sizes have a difficult time overcoming the data movement overhead. One way to get around this is to group multiple small transfers into one large transfer, i.e. group 100 16-point FFT data sets into one 1,600 word data transfer. Figure 5 below shows the results of doing this with 1,024 16-point FFTs. As shown in Figure 5, the average execution time for a 16-point FFT running in programmable logic is drastically reduced and now out performs the FFTW implementation.

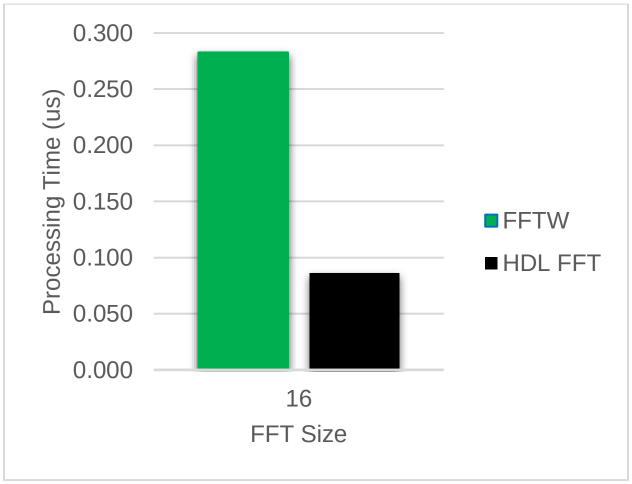


Figure 5 – 16-point FFT Processing Times with Large Data Transfer
(Note: HDL FFT was hand-written in C and compiled into programmable-logic using SDSoC)

Table 2 below summarizes the implementation accuracy using the mean-square-error (MSE) when compared to a double-precision MATLAB® model. The MSE statistic provides the average of the square difference between the single-precision floating-point FFT implementation and the double-precision floating-point MATLAB model. A small MSE indicates a good overall match between the model and the implementation. As indicated in Table 2, the FFTW and XFFT implementations closely match the MATLAB model.

FFT accuracy compared to double-precision MATLAB model		
FFT Size	FFTW Mean Square	XFFT Mean Square Error
8	1.7e-12	1.7e-12
16	3.0e-12	2.9e-12
32	4.9e-12	5.1e-12
64	1.2e-11	1.2e-11
128	2.3e-11	2.1e-11
256	4.3e-11	4.4e-11
512	8.9e-11	9.2e-11
1024	1.9e-10	1.8e-10
2048	3.7e-10	3.6e-10
4096	7.4e-10	7.1e-10
8192	1.5e-09	1.5e-09
16384	3.1e-09	2.9e-09

Table 2 – FFT Implementation Accuracy

Conclusion

This paper presented two options for implementing the Fast Fourier Transform in the Zynq UltraScale+ MPSoC device family from Xilinx. The first option was a software only implementation using the FFTW open-source code library which runs on the ARM Cortex-A53 processor. The second option used the Xilinx LogiCore IP to implement the FFT in programmable logic as an accelerator function. Significant gains were achieved with the accelerator which boosted execution time performance by a factor of 29 for the 16384-point FFT size. Smaller FFT sizes have difficulty overcoming the data movement overhead, but optimization can be achieved through strategic planning and control of data movement between the processing system and programmable logic. For the 16-point FFT size, execution speedup of 2.6X is possible with an accelerator when compared to the FFTW library running on the A53 processor.

Appendix A: Compiling FFTW for the ARM Cortex-A53 processor

This section describes the steps necessary to compile the FFTW library for use with the Zynq UltraScale+ MPSoC application processor. The following steps assume that appropriate Xilinx tools have been installed and the OS being targeted is Linux. The procedure below can be performed on a Windows machine, but it is recommended that a Linux machine is used (Note: the steps outlined below have not been tested in the Windows environment). Anywhere the <> brackets are used requires a replacement with a path for your specific environment – the text has also been changed to orange for your convenience.

1. Download the FFTW 3.3.7 source code from www.fftw.org/download.html and extract to your desired location, i.e. `~/fft_lib/fftw/`.

   > Note: The rest of these instructions assume that `~/fft_lib/fftw` is the location you extracted the source files to. If you used a different location then adjust accordingly.

2. Create a build directory in `~/fft_lib/fftw`, i.e. `~/fft_lib/fftw/build`

3. Change directory to `~/fft_lib/fftw/build`

4. Create an install directory in `~/fft_lib/fftw/build` i.e. `~/fft_lib/fftw/build/install`

5. Setup your environment by executing the following on the Linux command line (assumes bash shell is being used)
   a. `export CROSS_COMPILE=aarch64-linux-gnu-`
   b. `source <Xilinx-install-path>/SDK/2017.4/settings64.sh`

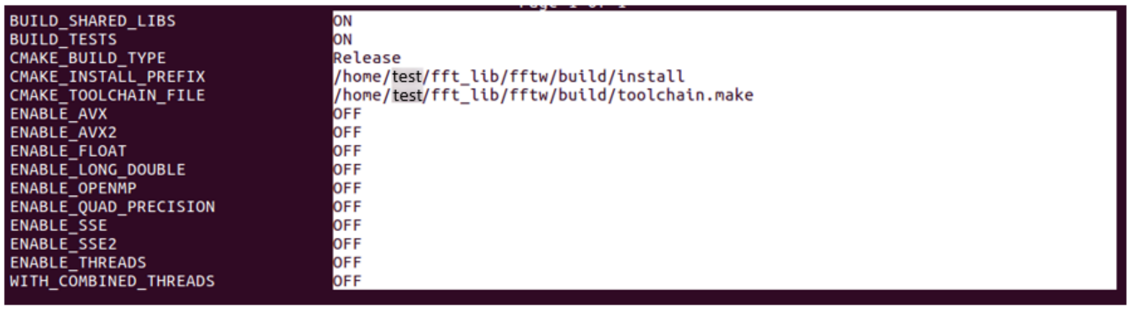
6. Create a `toolchain.make` file with the following contents (this is for the A53 architecture, the A9 architecture uses a different toolchain):

```
set( CMAKE_SYSTEM_NAME Linux )
set( CMAKE_SYSTEM_PROCESSOR arm )
set( CMAKE_C_COMPILER aarch64-linux-gnu-gcc )
set( CMAKE_CXX_COMPILER aarch64-linux-gnu-g++ )
set( CMAKE_INSTALL_PREFIX <full-path-to-your-fftw-directory>/build/install )
set( CMAKE_FIND_ROOT_PATH <Xilinx-install-
path>/SDK/2017.4/gnu/aarch64/lin/aarch64-linux/bin )
```

7. At the Linux command prompt execute:
```
cmake -D CMAKE_TOOLCHAIN_FILE=toolchain.make -D CMAKE_INSTALL_PREFIX=<full-
path-to-your-fftw-directory>/build/install <full-path-to-your-fftw-directory>
```

8. When cmake completes execute:
```
ccmake .
```

9. A menu will open that looks like:



```
BUILD_SHARED_LIBS                ON
BUILD_TESTS                      ON
CMAKE_BUILD_TYPE                 Release
CMAKE_INSTALL_PREFIX             /home/test/fft_lib/fftw/build/install
CMAKE_TOOLCHAIN_FILE             /home/test/fft_lib/fftw/build/toolchain.make
ENABLE_AVX                       OFF
ENABLE_AVX2                      OFF
ENABLE_FLOAT                     OFF
ENABLE_LONG_DOUBLE               OFF
ENABLE_OPENMP                    OFF
ENABLE_QUAD_PRECISION            OFF
ENABLE_SSE                       OFF
ENABLE_SSE2                      OFF
ENABLE_THREADS                   OFF
WITH_COMBINED_THREADS            OFF
```

Navigation within the cmake menu is performed using the arrow keys on your keyboard. To change/edit an option, navigate to the appropriate line and then press enter on the keyboard.

a. Change `BUILD_SHARED_LIBS` to `OFF` by pressing enter on your keyboard

b. Move the cursor to the `ENABLE_FLOAT` line and change it to `ON` by pressing enter on your keyboard

c. Type `c` to configure

d. Type `g` after configuration completes to generate the Makefile and exit the cmake menu

10. At the Linux command prompt execute:
```
make
```

11. When make completes execute the following at the Linux command prompt:
```
make install
```

12. The following files will be populated in your `~/fft_lib/fftw/build/install` directory

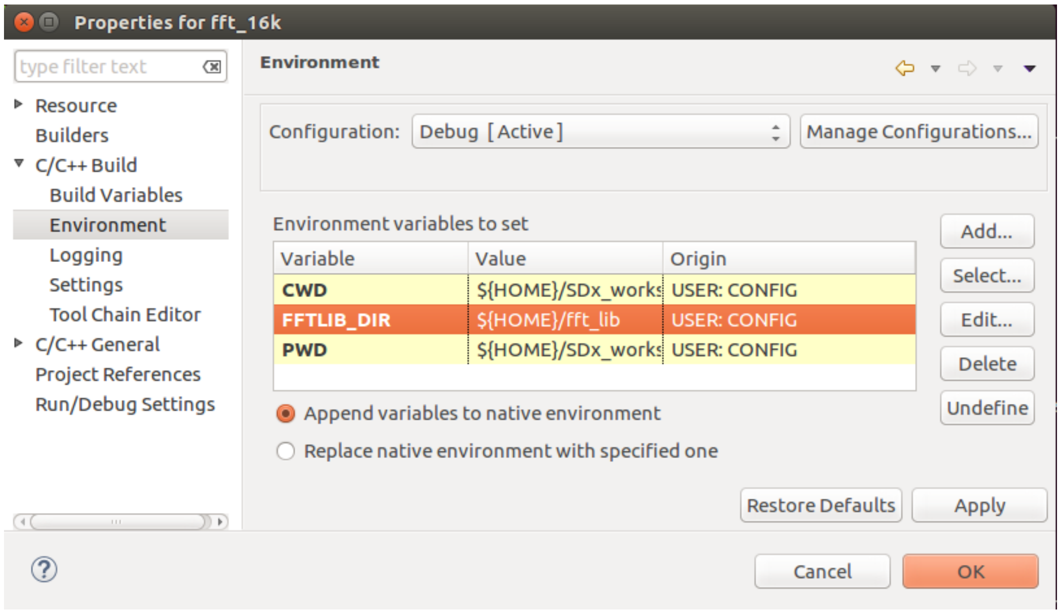
```
lib/libfftw3f.a
include/fftw3.h
```

These two files are needed for your SDSoC environment to correctly include and link the FFTW library

13. In your SDSoC project (assuming you have a project already set up that you want to include FFTW in, if not create one)

a. Right-click on the project you want to add FFTW to and select `C/C++ Build Settings`

b. Under the `C/C++ Build` → `Environment` menu set a build variable named `FFTLIB_DIR` that points to your `fft_lib` directory



Configuration: Debug [Active]

Environment variables to set

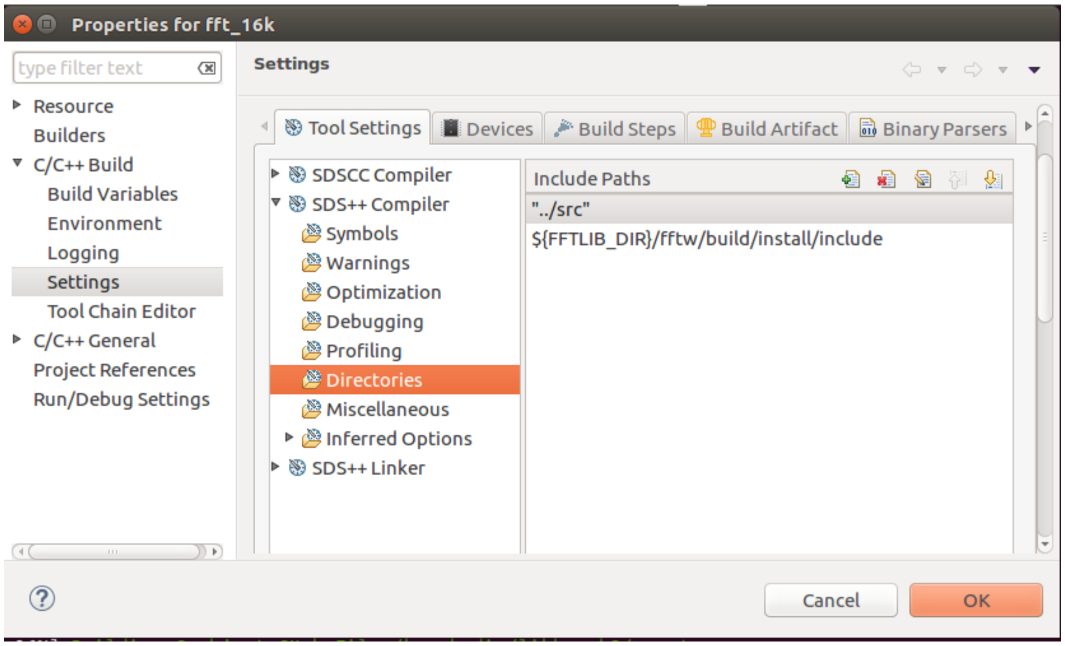
Variable	Value	Origin
CWD	${HOME}/SDx_works	USER: CONFIG
FFTLIB_DIR	${HOME}/fft_lib	USER: CONFIG
PWD	${HOME}/SDx_works	USER: CONFIG

[✓ Append variables to native environment]
Replace native environment with specified one

c. Click on `Settings` under `C/C++ Build`

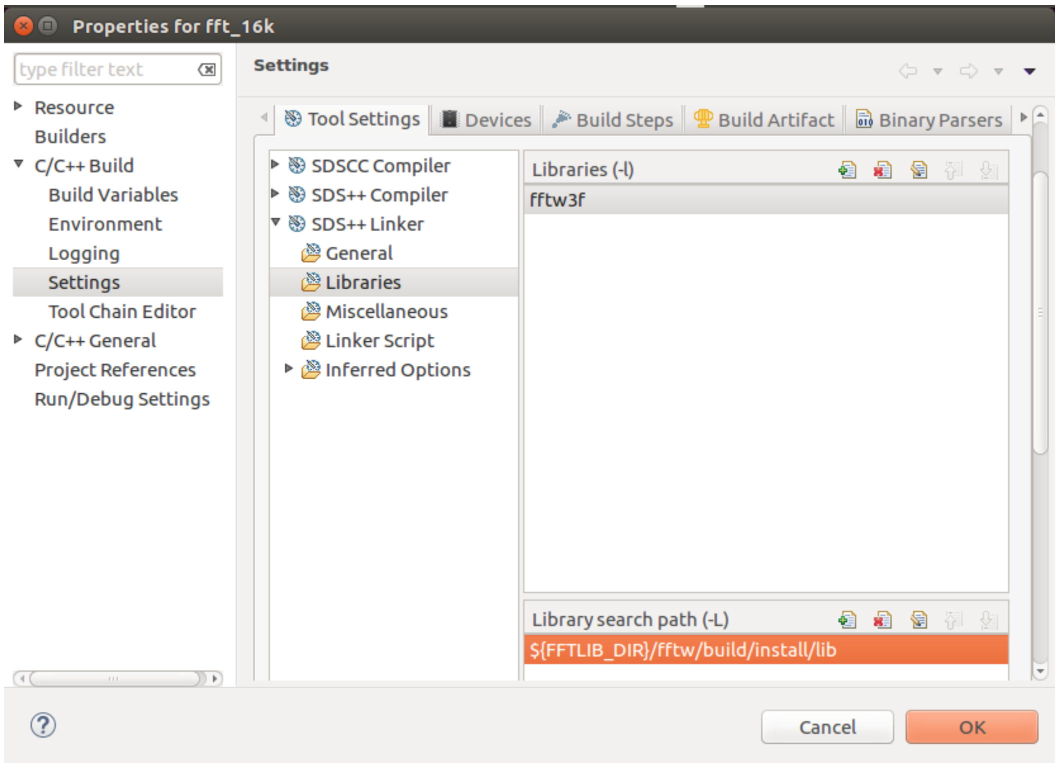
d. Assuming you are using C++, expand the `SDS++ Compiler` → `Tool Settings`. If you are using C then expand the `SDSCC Compiler` → `Tool Settings`.

e. Under `SDS++ Compiler` → `Directories` add the path location of your include directory, i.e. `${FFTLIB_DIR}/fftw/build/install/include` directory



Include Paths
"../src"
${FFTLIB_DIR}/fftw/build/install/include

f. Under `SDS++ Linker` → `Libraries` add `fftw3f` to the `Libraries` dialog and the path location of the library (i.e. `${FFTLIB_DIR}/fftw/build/install/lib`) to the `Library search path` dialog



Libraries (-l)
fftw3f

Library search path (-L)
${FFTLIB_DIR}/fftw/build/install/lib

14. Click OK and exit the project settings menu

15. You have successfully included the FFTW library for your ARM Cortex-A53 processor, see the fftw3.h header file for function prototypes and check out http://www.fftw.org/index.html#documentation for appropriate usage of the library functions

Appendix B: Creating a C-callable library for the Xilinx LogiCore IP FFT (v9.0)

Two different methods were employed when creating the C-callable XFFT library. This section covers both methods and why they were used. Anywhere the <> brackets are used requires a replacement with a path for your specific environment – the text has also been changed to orange for your convenience.

Method 1: Direct packaging of the XFFT IP

This method packages the XFFT IP directly without opening Vivado® and allows for full customization of the XFFT IP. There are some issues with using the run-time configurable transform length option, which will be explained in more detail for method 2. The following steps outline packaging of a C-callable library using a static FFT size and using the Xilinx LogiCore XFFT (v9.0) IP. These steps follow the method defined in Appendix E of Xilinx UG1027 (v2017.4).

1. Create a folder to put your files, i.e. `~/fft_lib/xfft`

   Create a header file containing constants and function prototypes for the XFFT functions being compiled. For this example there are two functions

```
/* xfft.h */

#ifndef XFFT_H_
#define XFFT_H_

#include "stdint.h"

#define N_FFT 16384

void xfft( uint64_t x[N_FFT], uint64_t y[N_FFT] );
void xfft_config( uint8_t config[1] );

#endif
```

   The `xfft()` function is used to call the accelerator while `xfft_config()` is used to configure the accelerator (i.e. forward versus inverse, etc.). Each argument must map to a physical AXI Interface on the underlying accelerator. For detailed port information for the XFFT IP please see PG109.

   Note: The data type for `x` & `y` is given as `uint64_t`. The XFFT core has a 64-bit AXI input interface that expects the imaginary component in the upper 32-bits and the real component in the lower 32-bits. We could create a `struct` or use the `complex<float>` type instead of `uint64_t` and rely on the SDSoC compiler to pack the data into a 64-bit container before sending it to the XFFT accelerator. The important thing is making sure the size of the data being transferred is aligned with the size expected by the accelerator.

2. Create function definition files which have empty function bodies

```
/* xfft.cpp */

#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include "xfft.h"

void xfft( uint64_t x[N_FFT], uint64_t y[N_FFT] ) { }
```

```
/* xfft_config.cpp */

#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include "xfft.h"

void xff_config( uint8_t config[1] ) {  }
```

Note: UG1127 states that `stdlib.h` and `stdio.h` must be included in the function definition file.

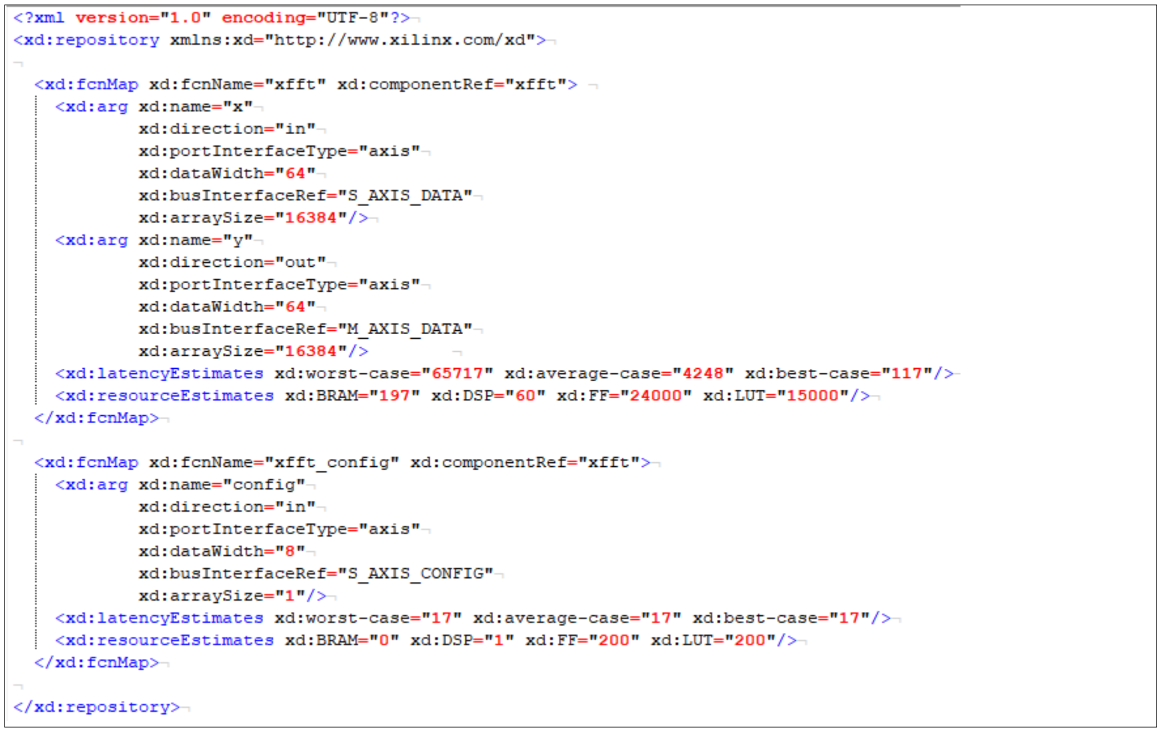
3. Create the IP configuration file

The IP configuration file sets parameters for the IP which would typically correspond to a HDL generic parameter. For example, the XFFT IP has a parameter called `data_format` which is set to `floating_point` below. Chapter 4 of PG109 lists all of the available User Parameters for the XFFT IP as well as acceptable values.

```
<?xml version="1.0" encoding="UTF-8"?>
<xd:component xmlns:xd="http://www.xilinx.com/xd" xd:name="xfft">
  <xd:parameter xd:name="transform_length" xd:value="16384"/>
  <xd:parameter xd:name="target_clock_frequency" xd:value="300"/>
  <xd:parameter xd:name="implementation_options" xd:value="pipelined_streaming_io"/>
  <xd:parameter xd:name="data_format" xd:value="floating_point"/>
  <xd:parameter xd:name="input_width" xd:value="32"/>
  <xd:parameter xd:name="phase_factor_width" xd:value="24"/>
  <xd:parameter xd:name="output_ordering" xd:value="natural_order"/>
  <xd:parameter xd:name="number_of_stages_using_block_ram_for_data_and_phase_factors" xd:value="7"/>
  <xd:parameter xd:name="complex_mult_type" xd:value="use_mults_performance"/>
</xd:component>
```

4. Create the function argument map

The purpose of the function argument map is to map software arguments to physical interfaces on the accelerator.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<xd:repository xmlns:xd="http://www.xilinx.com/xd">

  <xd:fcnMap xd:fcnName="xfft" xd:componentRef="xfft">
    <xd:arg xd:name="x"
            xd:direction="in"
            xd:portInterfaceType="axis"
            xd:dataWidth="64"
            xd:busInterfaceRef="S_AXIS_DATA"
            xd:arraySize="16384"/>
    <xd:arg xd:name="y"
            xd:direction="out"
            xd:portInterfaceType="axis"
            xd:dataWidth="64"
            xd:busInterfaceRef="M_AXIS_DATA"
            xd:arraySize="16384"/>
    <xd:latencyEstimates xd:worst-case="65717" xd:average-case="4248" xd:best-case="117"/>
    <xd:resourceEstimates xd:BRAM="197" xd:DSP="60" xd:FF="24000" xd:LUT="15000"/>
  </xd:fcnMap>

  <xd:fcnMap xd:fcnName="xfft_config" xd:componentRef="xfft">
    <xd:arg xd:name="config"
            xd:direction="in"
            xd:portInterfaceType="axis"
            xd:dataWidth="8"
            xd:busInterfaceRef="S_AXIS_CONFIG"
            xd:arraySize="1"/>
    <xd:latencyEstimates xd:worst-case="17" xd:average-case="17" xd:best-case="17"/>
    <xd:resourceEstimates xd:BRAM="0" xd:DSP="1" xd:FF="200" xd:LUT="200"/>
  </xd:fcnMap>

</xd:repository>
```

UG 1027 provides a description of each parameter within the function map file. Additional information is provided below.

- The `fcnName` argument corresponds to the software function being mapped and should match the function name from the `xfft.cpp` file exactly
- The `componentRef` argument corresponds to the name of the underlying accelerator IP which is taken from the IP-XACT VNLV identifier and can typically be found in the `component.xml` file created when the IP is packaged with Vivado
- `xd:arg xd:name` is the name of the software argument
- `xd:arg xd:busInterfaceRef` is the prefix for the interface to which the argument maps

  For example, if the accelerator has an AXI interface named `S_AXI_DATA` which has signals `S_AXI_DATA_tdata`, `S_AXI_DATA_tvalid`, etc. then the prefix is `S_AXI_DATA`

- `xd:arg xd:arraySize` is the size of the array being mapped to hardware
  - For the axis interface type this must be at least 1, which is why the `config` argument for function `xfft_config()` is an array of size 1

There needs to be a separate function map for each software function, but they can be co-located as shown above. You can see that the `xfft_config()` function maps to the `S_AXI_CONFIG` (configuration) interface of the XFFT IP core. It is possible to get rid of the `xff_config()` function and add a new argument to `xfft()`. However, it's best to separate configuration from processing so the accelerator does not get configured each time it is called. This in turn reduces the processing overhead. The accelerator can be configured once at the beginning of the application and then called multiple times before reconfiguring.

5. Packaging the IP
   a. There is a great Makefile example from the FIR filter sample project (`<install directory>/SDx/2017.4/samples/fir_lib/build/Makefile`) that can be modified for use with the XFFT. You will need to make the following modifications:
      1. Replace all instances of `libfir` with `libxfft`
      2. Remove lines containing `fir_reload`
      3. Replace all instances of `fir_compiler` with `xfft`
      4. Replace all instances of `fir` with `xfft`
      5. Replace all instances of `.c` with `.cpp`
      6. Change lines containing `-vlnv xilinx.com:ip:fir_compiler:7.2` to `-vlnv xilinx.com:ip:xfft:9.0`
      7. Add the target CPU option `-target-cpu cortex-a53`
      8. add the `-ip-repo` option to point to the Xilinx IP directory located in `<install directory>/Vivado/2017.4/data/ip/Xilinx`

   b. The updated make file command should be

```
libxfft.a: xfft.cpp xfft.h xfft.fcnmap.xml xfft.params.xml
       sdslib -lib libxfft.a \
         xfft xfft.cpp \
         xfft_config xfft_config.cpp \
         -vlnv xilinx.com:ip:xfft:9.0 \
         -ip-map xfft.fcnmap.xml \
         -ip-params xfft.params.xml \
         -target-cpu cortex-a53 \
         -ip-repo <install directory>/Vivado/2017.4/data/ip/xilinx
```

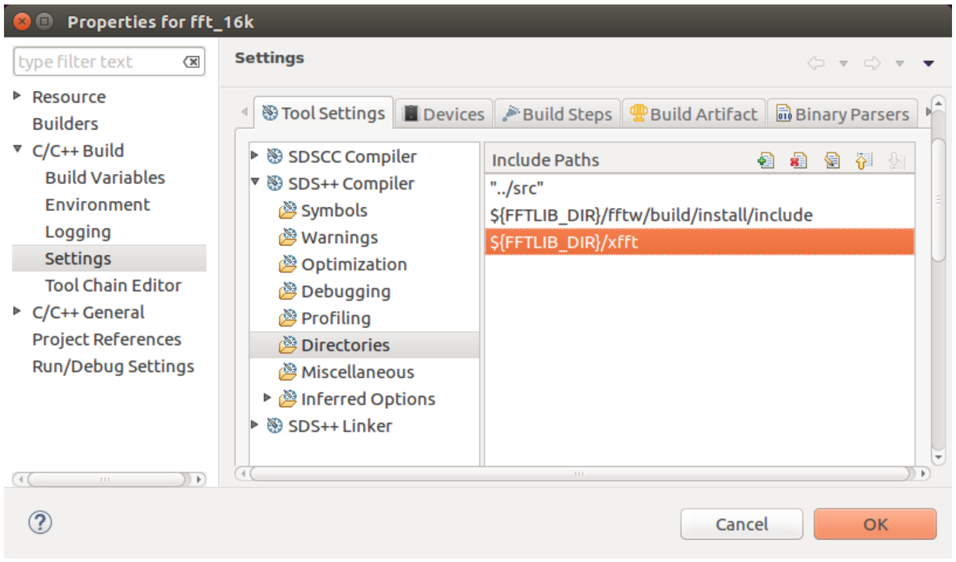
6. Open the Linux terminal if not already open and execute
   `source <install directory>/SDx/2017.4/settings64.sh`

7. From the terminal navigate to the location where you saved your header file, empty function definition file, parameters XML file, function map XML file, and Makefile – if they are not all in the same directory move them there now (i.e. `~/fft_lib/xfft/`)

8. At the command line type `make libxfft.a` and press `enter`

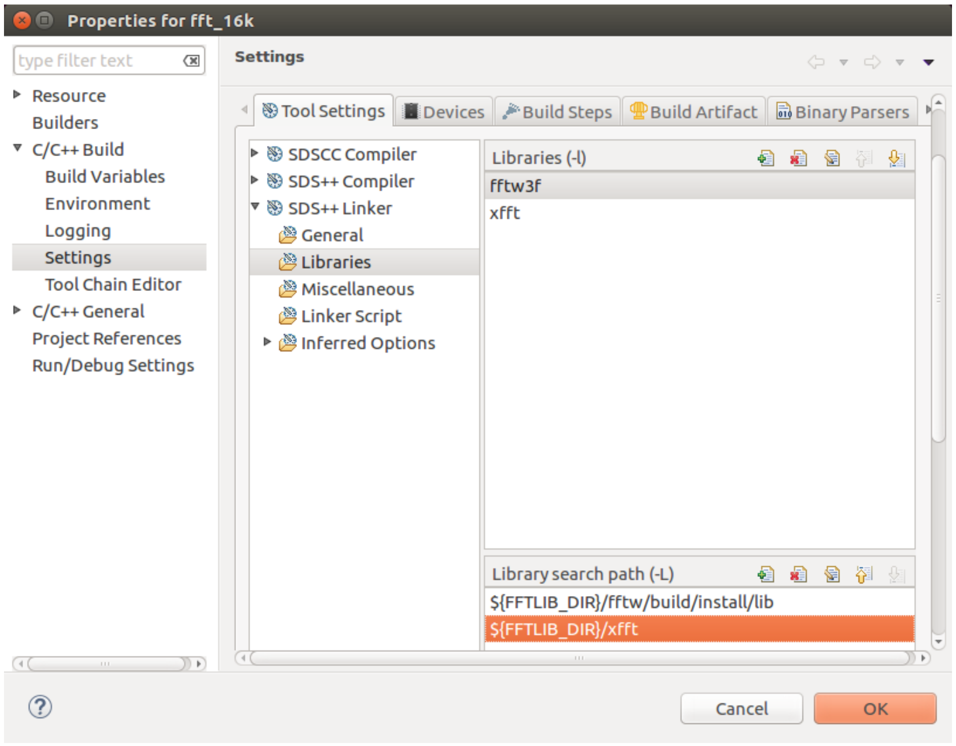
9. After the build completes you should see a `libxfft.a` file in the directory. This file is used by the SDS++ linker to include the accelerator in your SDSoC project.

10. To include your C-callable IP in SDSoC
    a. Right-click on the project you want to add XFFT to and select `C/C++ Build Settings`
    b. Assuming you are using C++, expand the `SDS++ Compiler` under the `Tool Settings` tab. If you are using C then expand the `SDSCC Compiler`.
    c. Under `SDS++ Compiler` → `Directories` add the path location of your directory containing the `xfft.h` file (i.e. `/home/<user>/fft_lib/xfft` – Note: figure below uses environment variable `FFTLIB_DIR` defined in Appendix A as `/home/<user>/fft_lib`)



```
"../src"
${FFTLIB_DIR}/fftw/build/install/include
${FFTLIB_DIR}/xfft
```

    d. Under `SDS++ Linker` → `Libraries` add `xfft` to the `Libraries (-l)` dialog and the path location of the `libxfft.a` library (i.e. `${FFTLIB_DIR}/xfft`) to the `Library search path (-L)` dialog (figure below)



Libraries (-l)

```
fftw3f
xfft
```

Library search path (-L)

```
${FFTLIB_DIR}/fftw/build/install/lib
${FFTLIB_DIR}/xfft
```

11. Click OK and exit the project settings menu

12. You have successfully included the XFFT library
    a. To use XFFT make sure to call `xfft_config()` with the correct configuration parameters before calling `xfft()`
    b. You do not need to specify the `xfft()` function for hardware acceleration within the SDSoC GUI – C-callable IP are hardware accelerated by definition
    c. Configuration parameters are dependent on your implementation and will vary based on how you parameterized the XFFT IP in your params.xml file – for more details on the XFFT configuration interface see PG109

Method 2: Packaging Using a HDL Wrapper

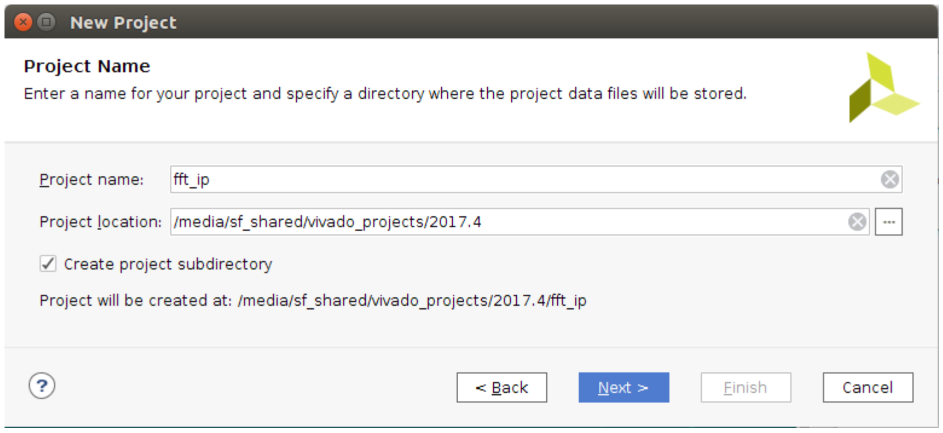
There are some issues with packaging the XFFT IP directly with the run-time configurable transform length configuration option. In order to transfer a variable amount of data using SDSoC from the PS to the PL, the data copy (or zero_copy) pragma must be used to specify the amount of data. The copy pragma can use a run-time determinable parameter to decide how much data to move, but the parameter used to compute this information must be part of the function call. The following example illustrates this point.

Example – using copy pragma to specify variable length transfer between PS and PL:

```
#pragma SDS data copy(data_in[0:size], data_out[0:size])
xfft( int size, uint64_t *data_in, uint64_t *data_out )
```

In the above example the size parameter is needed in the function prototype to determine the amount of data to move from PS to PL. For C-callable IP, there must be a one-to-one mapping between the function arguments and the ports on the IP. Thus, there must be an interface port on the IP that we can map the size argument to. This is where the HDL wrapper comes in. The HDL wrapper provides an AXI-Stream interface for the size argument that is left unconnected internally. That way SDSoC requirements are satisfied and we can use a run-time configurable FFT size. The following steps work through creating and packaging the XFFT IP using Vivado.

1. Create a Vivado project named fft_ip and click Next



New Project

Project Name

Enter a name for your project and specify a directory where the project data files will be stored.

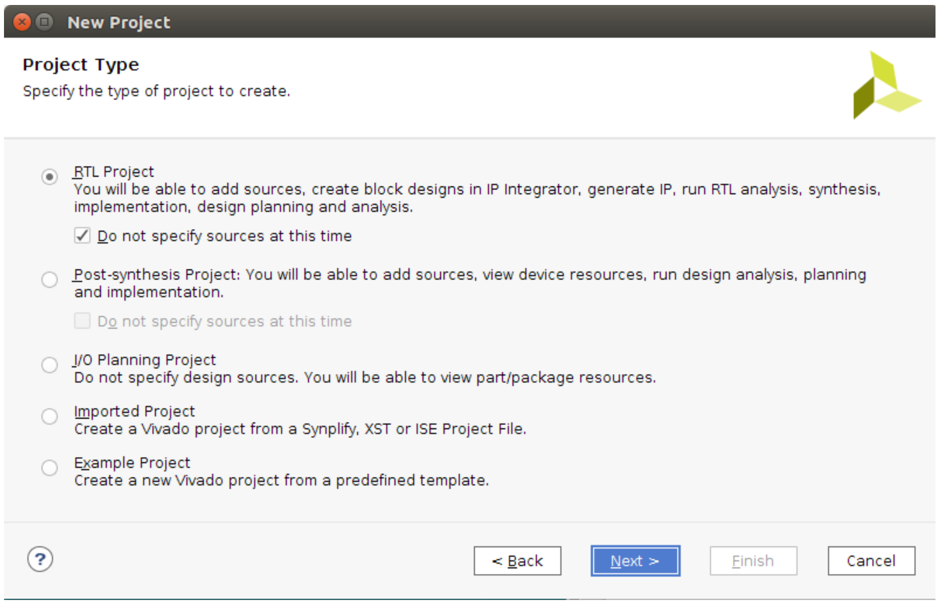
Project name: fft_ip

Project location: /media/sf_shared/vivado_projects/2017.4

☑ Create project subdirectory

Project will be created at: /media/sf_shared/vivado_projects/2017.4/fft_ip

2. For the project type select RTL Project and check the “Do not specify sources at this time” box then click Next



Project Type

Specify the type of project to create.

(●) RTL Project
You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.

[✓] Do not specify sources at this time

( ) Post-synthesis Project: You will be able to add sources, view device resources, run design analysis, planning and implementation.

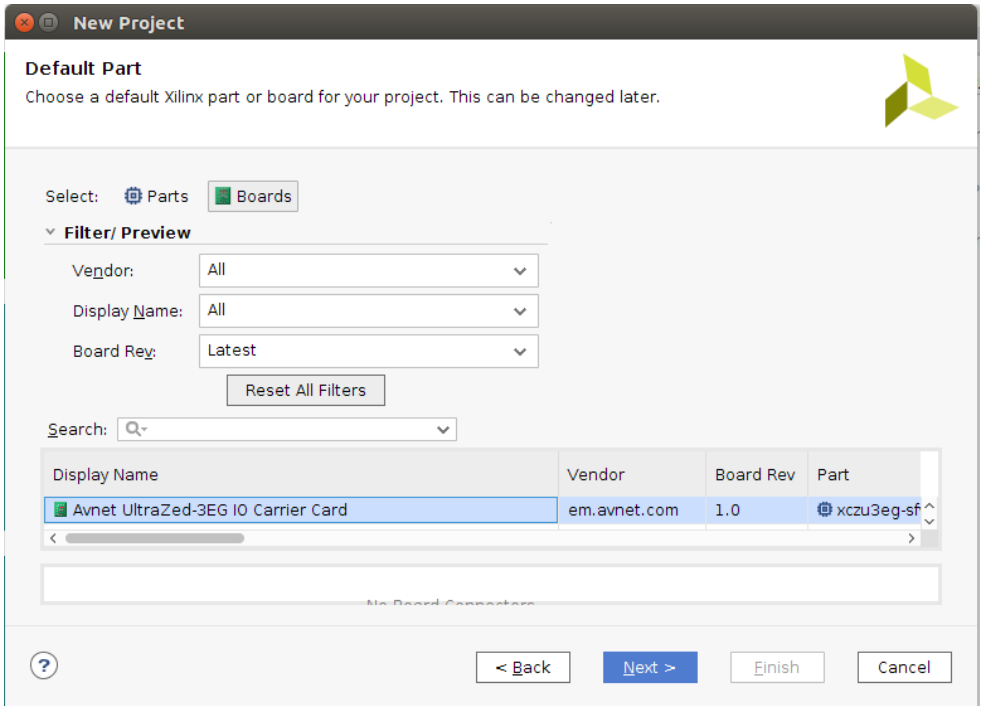
[ ] Do not specify sources at this time

( ) I/O Planning Project
Do not specify design sources. You will be able to view part/package resources.

( ) Imported Project
Create a Vivado project from a Synplify, XST or ISE Project File.

( ) Example Project
Create a new Vivado project from a predefined template.

3. Select the Avnet UltraZed-3EG IO Carrier Card board and click Next
Note: this step assumes that you have already installed the UltraZed IO Carrier Card board definition files which can be found at http://zedboard.org/sites/default/files/documentations/UltraZed_Board_Definition_Files_v2017_2_Release_All_CC_5_0.zip)



Default Part

Choose a default Xilinx part or board for your project. This can be changed later.

Select: Parts [✓ Boards]

Filter/ Preview

Vendor: All

Display Name: All

Board Rev: Latest

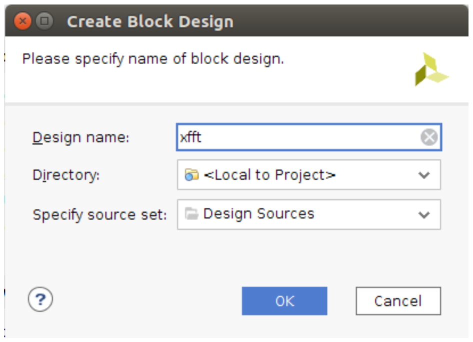
Search:

Display Name	Vendor	Board Rev	Part
Avnet UltraZed-3EG IO Carrier Card	em.avnet.com	1.0	xczu3eg-sf

No Board Connectors

4. Click Finish to create the project

5. Create a new block design by clicking on “Create Block Design” under the Flow Navigator window
    a. Name the block design xfft and click OK



Create Block Design

Please specify name of block design.

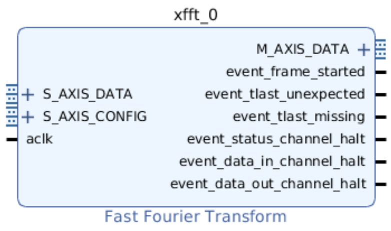
Design name: xfft

Directory: <Local to Project>

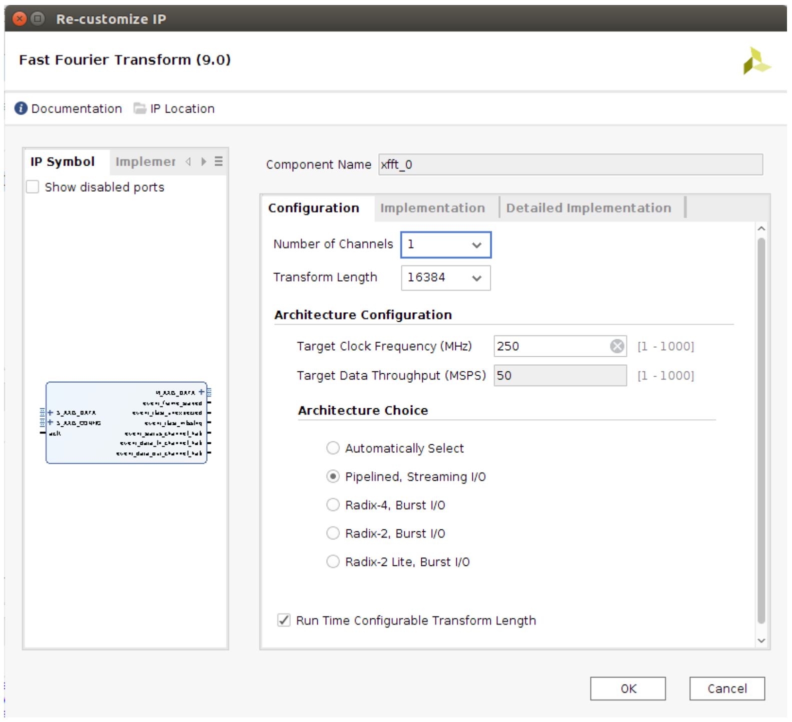
Specify source set: Design Sources

OK Cancel

6. Add the Fast Fourier Transform IP to the block design canvas



7. Double-click the IP to configure
    a. On the Configuration tab
        i. Set the Transform Length to the desired maximum. For this study the maximum was set to 16384
        ii. Set the Architecture to “Pipelined, Streaming I/O”
        iii. Check the “Run Time Configurable Transform Length” box



Re-customize IP

Fast Fourier Transform (9.0)

Documentation IP Location

IP Symbol Implemen

Show disabled ports

Component Name xfft_0

Configuration | Implementation | Detailed Implementation

Number of Channels: 1

Transform Length: 16384

Architecture Configuration

Target Clock Frequency (MHz): 250 [1 - 1000]

Target Data Throughput (MSPS): 50 [1 - 1000]

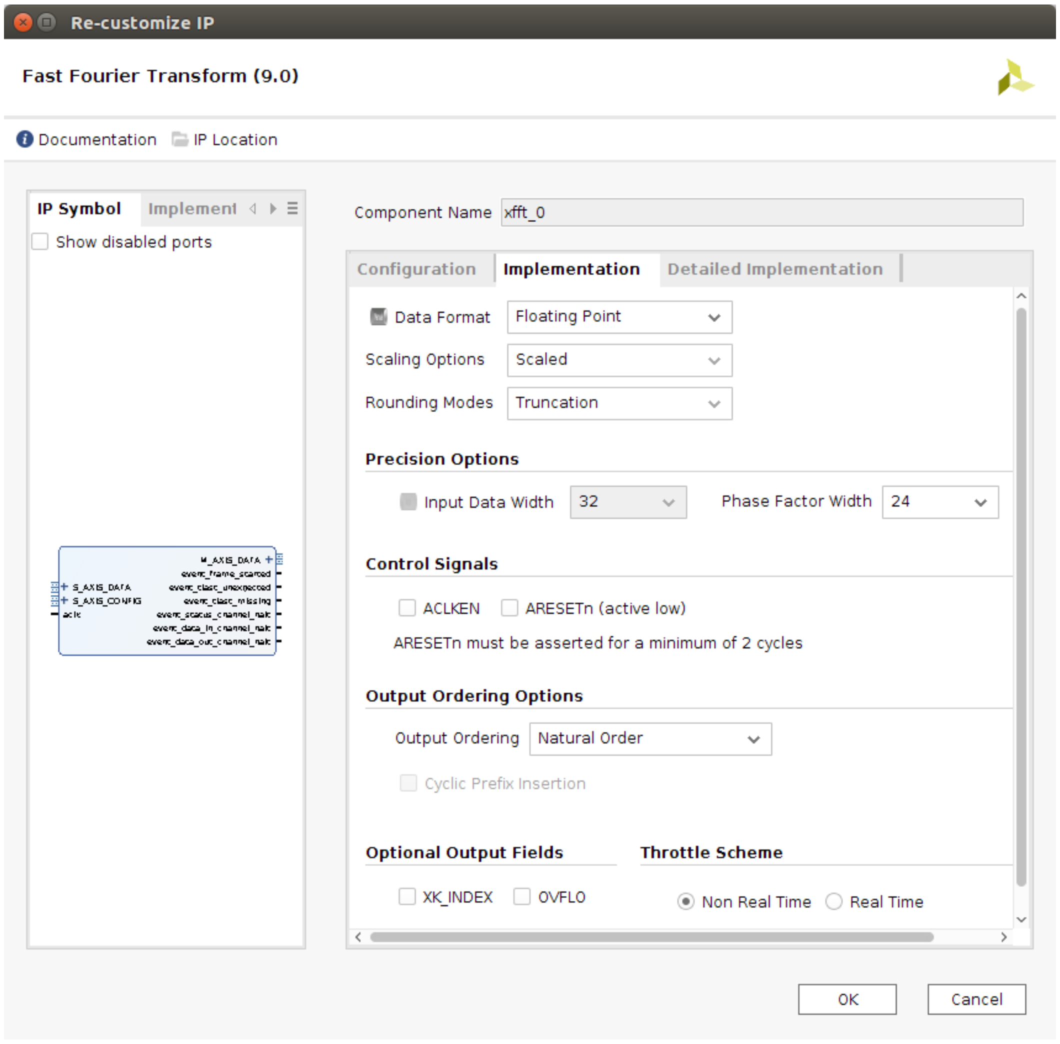
Architecture Choice

- Automatically Select
- [✓ Pipelined, Streaming I/O]
- Radix-4, Burst I/O
- Radix-2, Burst I/O
- Radix-2 Lite, Burst I/O

[✓] Run Time Configurable Transform Length

OK Cancel

b. On the Implementation tab
   i. Set Data Format to “Floating-Point”
   ii. Set Output Ordering to “Natural Order”
   iii. Leave other fields with the default values



Fast Fourier Transform (9.0)

Component Name: xfft_0

Data Format: Floating Point
Scaling Options: Scaled
Rounding Modes: Truncation

Precision Options
Input Data Width: 32
Phase Factor Width: 24

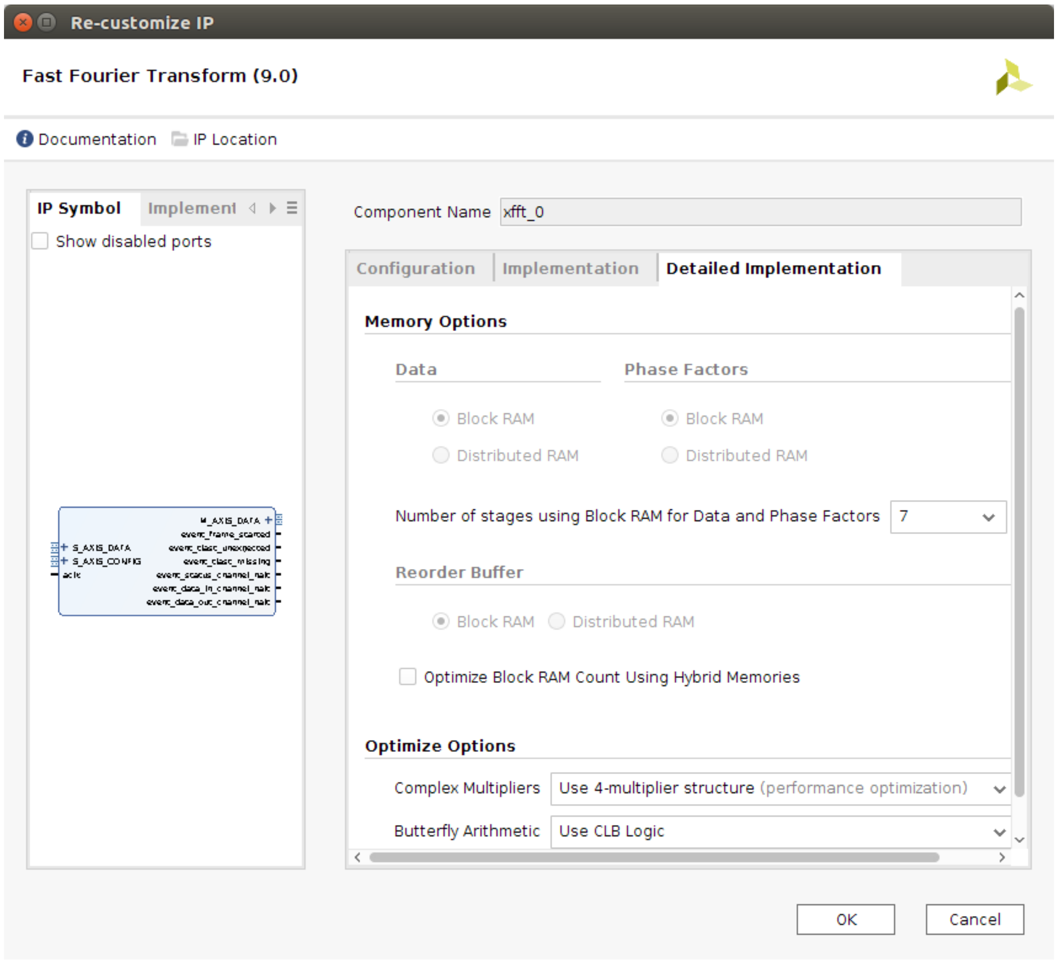
Control Signals
[ ] ACLKEN [ ] ARESETn (active low)
ARESETn must be asserted for a minimum of 2 cycles

Output Ordering Options
Output Ordering: Natural Order
[ ] Cyclic Prefix Insertion

Optional Output Fields
[ ] XK_INDEX [ ] OVFLO

Throttle Scheme
[✓ Non Real Time] Real Time

c. On the Detailed Implementation tab
   i. Under Optimize Options set the Complex Multipliers option to “Use 4-multiplier structure”



Re-customize IP

Fast Fourier Transform (9.0)

Documentation | IP Location

IP Symbol | Implement

Show disabled ports

Component Name: xfft_0

Configuration | Implementation | **Detailed Implementation**

Memory Options

Data: [✓ Block RAM] Distributed RAM

Phase Factors: [✓ Block RAM] Distributed RAM

Number of stages using Block RAM for Data and Phase Factors: 7

Reorder Buffer: [✓ Block RAM] Distributed RAM

Optimize Block RAM Count Using Hybrid Memories ☐

Optimize Options

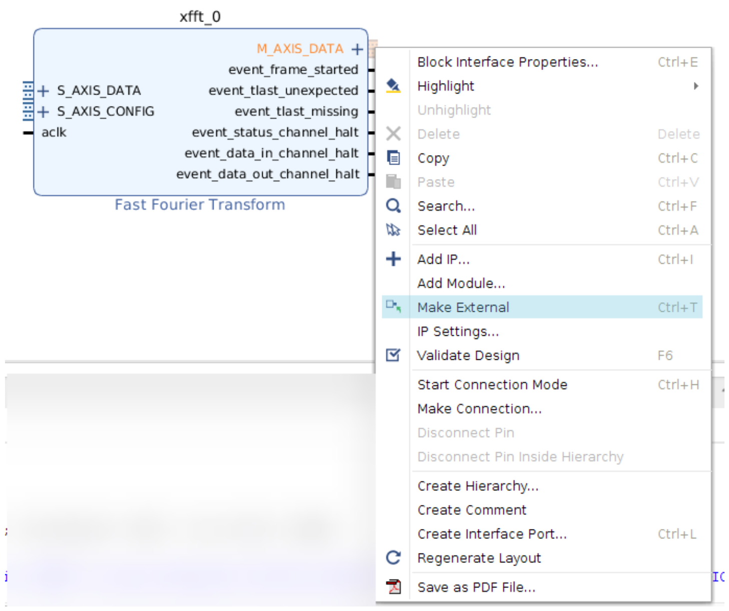
Complex Multipliers: Use 4-multiplier structure (performance optimization)

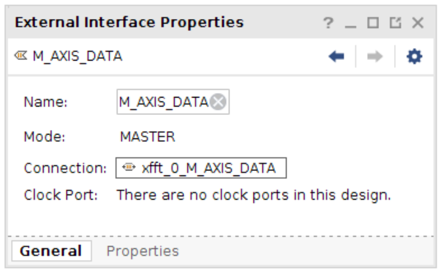
Butterfly Arithmetic: Use CLB Logic

OK | Cancel

d. Click OK to finish customizing the IP

8. Make the XFFT IP ports external
   a. Right-click on M_AXIS_DATA and select “Make External”. Vivado will add “_0” to the end of the port name, so modify the port to remove the “_0”




External Interface Properties

M_AXIS_DATA

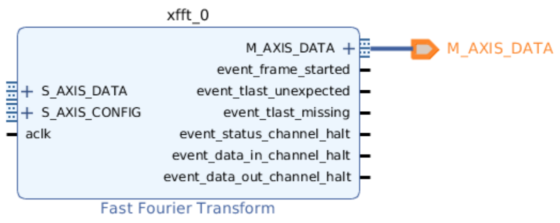
Name: M_AXIS_DATA

Mode: MASTER

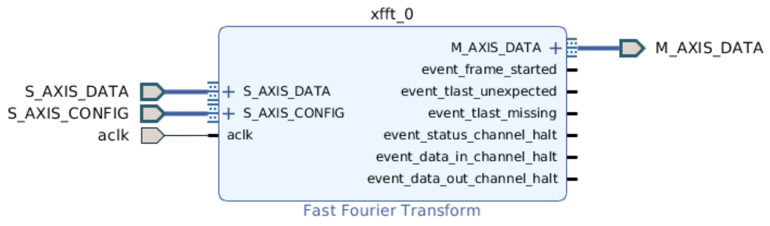
Connection: xfft_0_M_AXIS_DATA

Clock Port: There are no clock ports in this design.

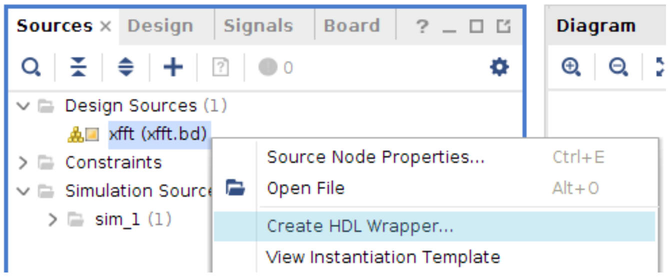
General Properties



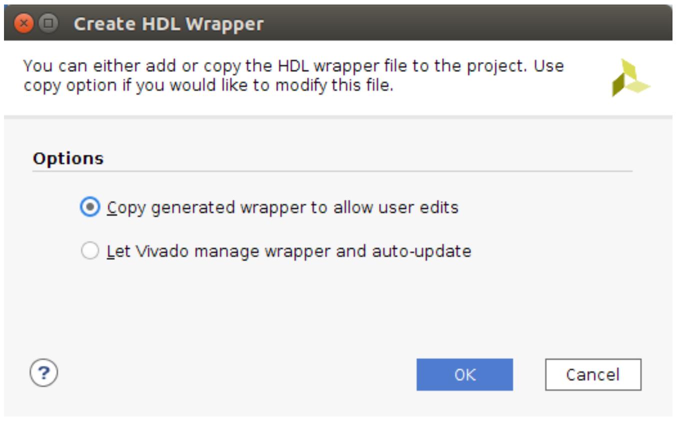
   b. Repeat step (a) for the aclk, S_AXIS_DATA, and S_AXIS_CONFIG ports



9. Create a HDL wrapper file
   a. In the Sources window right click on xfft and select “Create HDL Wrapper”



   b. Choose “Copy generated wrapper to allow user edits” option in the pop-up dialog and click OK


Create HDL Wrapper

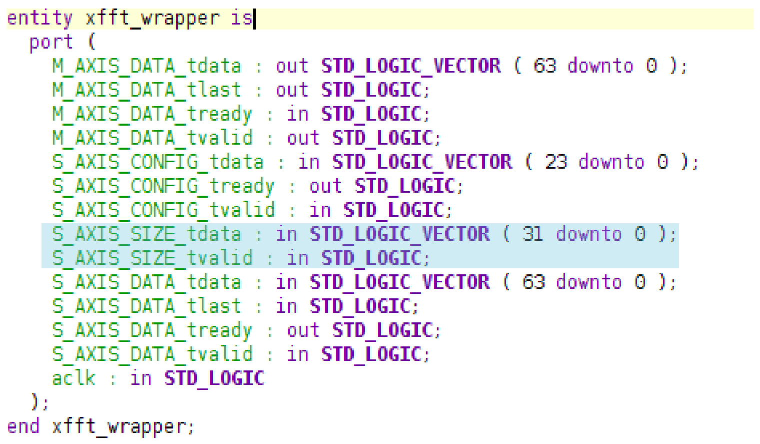
You can either add or copy the HDL wrapper file to the project. Use copy option if you would like to modify this file.

Options

[✓ Copy generated wrapper to allow user edits]

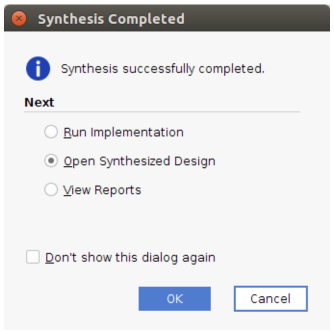
Let Vivado manage wrapper and auto-update

10. Add an AXI-Stream interface to the HDL wrapper for the “size” argument to map to



```
entity xfft_wrapper is
  port (
    M_AXIS_DATA_tdata : out STD_LOGIC_VECTOR ( 63 downto 0 );
    M_AXIS_DATA_tlast : out STD_LOGIC;
    M_AXIS_DATA_tready : in STD_LOGIC;
    M_AXIS_DATA_tvalid : out STD_LOGIC;
    S_AXIS_CONFIG_tdata : in STD_LOGIC_VECTOR ( 23 downto 0 );
    S_AXIS_CONFIG_tready : out STD_LOGIC;
    S_AXIS_CONFIG_tvalid : in STD_LOGIC;
    S_AXIS_SIZE_tdata : in STD_LOGIC_VECTOR ( 31 downto 0 );
    S_AXIS_SIZE_tvalid : in STD_LOGIC;
    S_AXIS_DATA_tdata : in STD_LOGIC_VECTOR ( 63 downto 0 );
    S_AXIS_DATA_tlast : in STD_LOGIC;
    S_AXIS_DATA_tready : out STD_LOGIC;
    S_AXIS_DATA_tvalid : in STD_LOGIC;
    aclk : in STD_LOGIC
  );
end xfft_wrapper;
```

11. Synthesize the design. When synthesis completes click on Cancel to close the pop-up



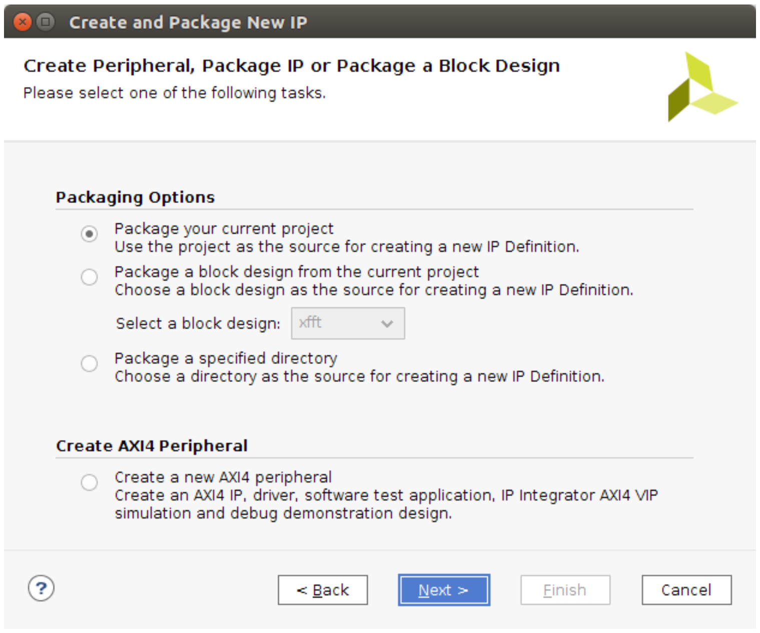
Synthesis successfully completed.

Next

- ( ) Run Implementation
- (●) Open Synthesized Design
- ( ) View Reports

[ ] Don't show this dialog again

12. Package the project as IP
    a. Go to Tools → Create and Package New IP
    b. Click Next
    c. Choose "Package your current project" and click Next



Create Peripheral, Package IP or Package a Block Design

Please select one of the following tasks.

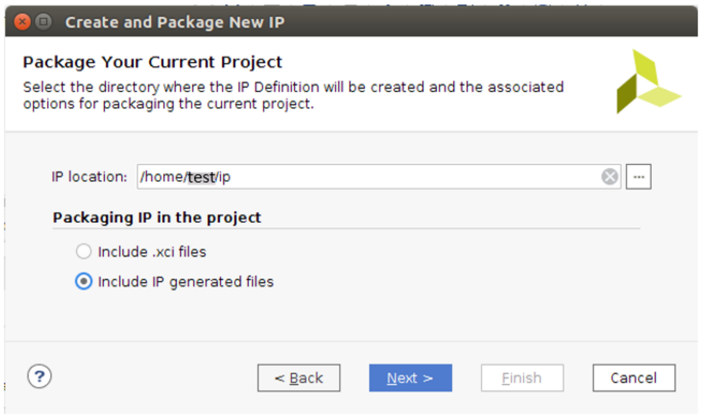
Packaging Options

- (●) Package your current project
  Use the project as the source for creating a new IP Definition.
- ( ) Package a block design from the current project
  Choose a block design as the source for creating a new IP Definition.
  Select a block design: xfft
- ( ) Package a specified directory
  Choose a directory as the source for creating a new IP Definition.

Create AXI4 Peripheral

- ( ) Create a new AXI4 peripheral
  Create an AXI4 IP, driver, software test application, IP Integrator AXI4 VIP simulation and debug demonstration design.

    d. Set the IP Location to your desired output path (i.e. ~/ip) and select "Include IP generated files" then click Next



Package Your Current Project

Select the directory where the IP Definition will be created and the associated options for packaging the current project.

IP location: /home/test/ip

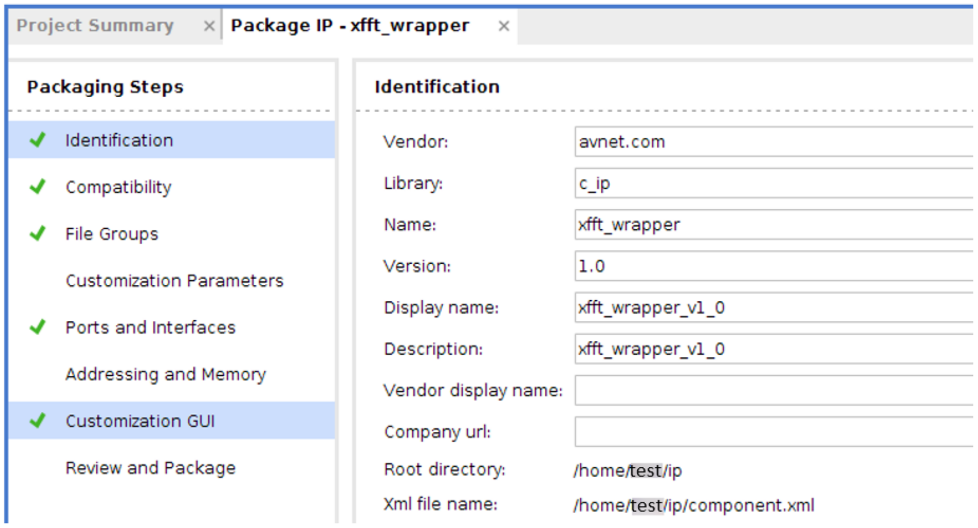
Packaging IP in the project

- ( ) Include .xci files
- (●) Include IP generated files

e. Click Finish and OK if you get a pop-up message about copying IP to your output location

f. On the “New IP Creation” window click Finish

g. A new Vivado project will open
    i. On the “Package IP” window under “Packaging Steps > Identification” fill out the information for your IP. This will be used later during the C-Callable IP packaging steps in SDSoC



Project Summary | Package IP - xfft_wrapper

Packaging Steps

- ✓ Identification
- ✓ Compatibility
- ✓ File Groups
- Customization Parameters
- ✓ Ports and Interfaces
- Addressing and Memory
- ✓ Customization GUI
- Review and Package

Identification

Vendor:	avnet.com
Library:	c_ip
Name:	xfft_wrapper
Version:	1.0
Display name:	xfft_wrapper_v1_0
Description:	xfft_wrapper_v1_0
Vendor display name:	
Company url:	
Root directory:	/home/test/ip
Xml file name:	/home/test/ip/component.xml

    ii. Under “Packaging Steps > Review and Package” click Package IP

h. Close the Vivado project when the IP packaging has completed

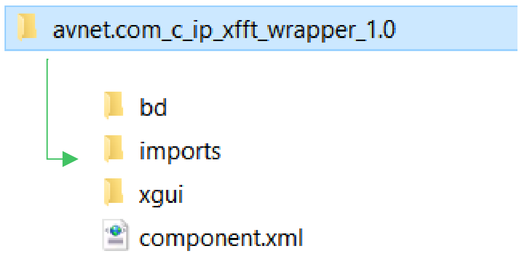
13. Change directory to the IP output location (i.e. ~/ip)
    a. Create a new directory using the VLNV format (Vendor, library, name, and version). This information was used in step 11.g.i.

    For example, if your VLNV was
    - Vendor = avnet.com
    - Library = c_ip
    - Name = xfft_wrapper
    - Version = 1.0

    Then you would create a directory named avnet.com_c_ip_xfft_wrapper_1.0.

    Copy the contents of the IP output to your newly created directory.

    Your directory structure should look like:



```
avnet.com_c_ip_xfft_wrapper_1.0
    bd
    imports
    xgui
    component.xml
```

14. The remainder of this procedure follows all steps outlined in Method 1 with a few minor differences noted below

    a. Update xfft.h, xfft.cpp, and xfft_config.cpp to account for port changes and to add data copy pragmas

```
/* xfft.h */

#ifndef XFFT_H_
#define XFFT_H_

#include "stdint.h"

#define N_FFT 16384

#pragma SDS data copy(size[0:0], x[0:*size], y[0:*size])
#pragma SDS data access_pattern(x:SEQUENTIAL, y:SEQUENTIAL)
void xfft_wrapper( int *size, uint64_t *x, uint64_t *y );

void xfft_config( uint16_t config[1] );

#endif
```

```
/* xfft.cpp */

#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include "xfft.h"

void xfft_wrapper( int *size, uint64_t *x, uint64_t *y ) { }
```

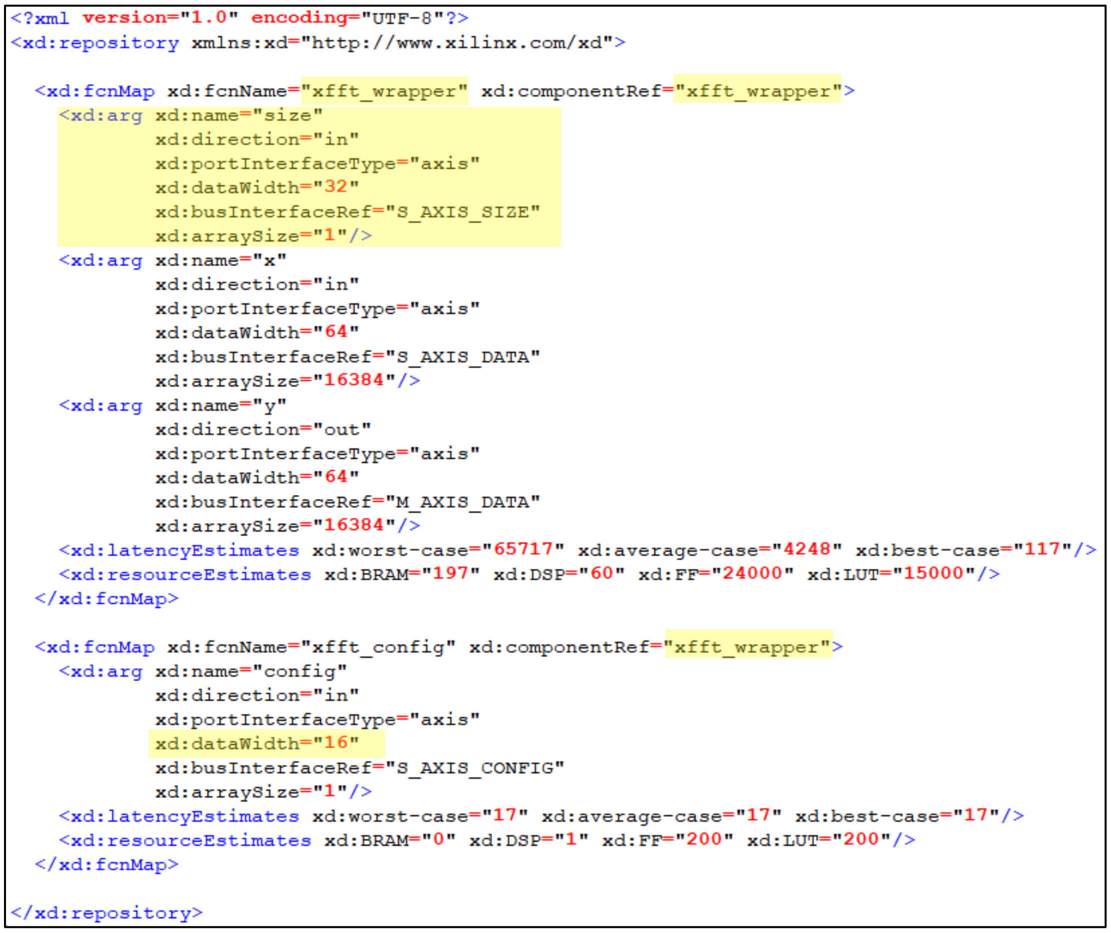
```
/* xfft_config.cpp */

#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include "xfft.h"

void xfft_config( uint16_t config[1] ) { }
```

Note: argument `config` of function `xfft_config` changed from an 8-bit to 16-bit data type to allow for run-time selection of the FFT size

b. Update the fncmap.xml file to incorporate the size argument, change the componentRef (and possibly the fcnName if your C-code function name changed) to the name of the IP that was packaged, i.e. xfft_wrapper, and update the configuration data width to 16. An updated function map is shown in the figure below.



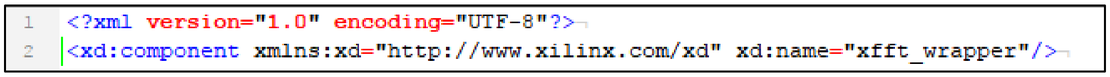
```xml
<?xml version="1.0" encoding="UTF-8"?>
<xd:repository xmlns:xd="http://www.xilinx.com/xd">

  <xd:fcnMap xd:fcnName="xfft_wrapper" xd:componentRef="xfft_wrapper">
    <xd:arg xd:name="size"
            xd:direction="in"
            xd:portInterfaceType="axis"
            xd:dataWidth="32"
            xd:busInterfaceRef="S_AXIS_SIZE"
            xd:arraySize="1"/>
    <xd:arg xd:name="x"
            xd:direction="in"
            xd:portInterfaceType="axis"
            xd:dataWidth="64"
            xd:busInterfaceRef="S_AXIS_DATA"
            xd:arraySize="16384"/>
    <xd:arg xd:name="y"
            xd:direction="out"
            xd:portInterfaceType="axis"
            xd:dataWidth="64"
            xd:busInterfaceRef="M_AXIS_DATA"
            xd:arraySize="16384"/>
    <xd:latencyEstimates xd:worst-case="65717" xd:average-case="4248" xd:best-case="117"/>
    <xd:resourceEstimates xd:BRAM="197" xd:DSP="60" xd:FF="24000" xd:LUT="15000"/>
  </xd:fcnMap>

  <xd:fcnMap xd:fcnName="xfft_config" xd:componentRef="xfft_wrapper">
    <xd:arg xd:name="config"
            xd:direction="in"
            xd:portInterfaceType="axis"
            xd:dataWidth="16"
            xd:busInterfaceRef="S_AXIS_CONFIG"
            xd:arraySize="1"/>
    <xd:latencyEstimates xd:worst-case="17" xd:average-case="17" xd:best-case="17"/>
    <xd:resourceEstimates xd:BRAM="0" xd:DSP="1" xd:FF="200" xd:LUT="200"/>
  </xd:fcnMap>

</xd:repository>
```

c. Update params.xml file to remove IP customization parameters



```xml
<?xml version="1.0" encoding="UTF-8"?>
<xd:component xmlns:xd="http://www.xilinx.com/xd" xd:name="xfft_wrapper"/>
```

d. Update `-ip-repo` flag in your Makefile to point to the directory where your IP resides, i.e. ~/ip (not ~/ip/avnet.com_c_ip_xfft_wrapper_1.0)

e. Complete IP packaging as defined in Method 1 starting with step 6.