

FPGA Accelerated SDR

Custom IP Cores

Hao Wang

04-18-23

Abstract

A workflow for implementing custom IP cores for FPGAs, using a framework from Analog Devices Incorporated, was created and described. A proof-of-concept core and a core to interface with the Xilinx FFT core were implemented. Testbenches and test applications that run in Linux were also made to test the functionality of the implemented cores. Performance testing was done on the FFT core and shows a 60x speed up for a transform size of 8, and a 5.5x speed up for a transform size of 1024, when compared to the FFT implementation for SciPy in python.

Table of Contents

Introduction	3
Methodology.....	3
Development Environment.....	3
Validation	3
Building HDL and Boot File.....	4
AXI Compatible Core	5
AXI FFT Core	9
Testbenches.....	15
Performing an FFT	16
Performance Testing	17
Results.....	18
Conclusion.....	18
References.....	20
Appendix	20

Table of Figures

Figure 1. Message for a successful HDL build.	4
Figure 2. Make file for AXI Loopback Test.	5
Figure 3. TCL file for AXI Loopback Test.	6
Figure 4. Timing diagram for Microprocessor interface.....	7
Figure 5. Flowchart for the 3 processes in AXI Loopback Test.	8
Figure 6. System board with AXI Loopback Test instance.	8
Figure 7. Output of the test program for the AXI Loopback Test.	9
Figure 8. Timing diagram for AXIS transactions.	10
Figure 9. TDATA format for config channel.	10
Figure 10. TDATA format for input and output channels.	11
Figure 11. Flowchart for fft_config.....	11
Figure 12. Flowchart for fft_data_input.....	12
Figure 13. Flowchart for fft_data_output.....	12
Figure 14. Flowchart for read process of AXI FFT.....	13
Figure 15. Bus definitions for the AXI FFT core.....	14
Figure 16. Parameters for the AXI FFT core for a testbench.....	15
Figure 17. Successful testbench run.....	16
Figure 18. Output of the AXI FFT core test application.....	17

Table of Tables

Table 1. Register map for AXI FFT.	13
Table 2. Speed results comparing FPGA to Python.	18
Table 3. FPGA utilization of the AXI FFT and Xilinx FFT cores.....	18

Introduction

Field programmable gate arrays (FPGAs) have the potential to accelerate several common operations in SDR, such as filtering, upscaling, and performing Fast Fourier Transforms (FFT). Compared to software implementations, an FPGA is capable of massively parallelized computing, resulting in greater throughput. This advantage can be used for multi-stream filtering with many taps or performing FFTs with large transform sizes.

FPGAs are widely used for acceleration in a multitude of applications, including SDR. One such project is RFNoC, which aims to provide a framework to integrate intellectual property (IP) cores into the SDR workflow. It is maintained by Ettus Research and provided with their USRP devices [1]. Analog Devices Incorporated (ADI) is another group that uses FPGAs to interface with their RF transceivers. They support various configurations of FPGA boards and peripherals based on the reference design for each transceiver. The most common software configuration for ADI is Linux, but they also provide a barebones OS-less driver [2].

This document aims to detail the following processes:

- Setting up the development environment for the ADI reference designs hardware description language (HDL).
- Compiling the project for a ZedBoard, AD-FMCOMMS3-EBZ, and Linux configuration.
- Creating, integrating, and interfacing with a custom Advance Extensible Interface (AXI) compatible IP core.
- Creating an IP core to perform FFTs.
- Creating and running test benches.
- Interfacing with the AXI FFT core in a Linux application.
- Performance testing of the AXI FFT core.

Methodology

Development Environment

To set up the host environment to build the HDL, Vivado needs to be installed on a Linux-based host. The installer can be found here: <https://www.xilinx.com/products/design-tools/vivado.html>. Version 2022.2 was chosen because the current master branch on ADI's HDL repository depends on this version. The install location was set at the default directory: `"/tools/xilinx"`. Vitis was chosen as it contains components required to build the boot file. And the Zynq™ 7000 SoC Architecture was enabled so that the ZedBoard can be set as a build target. The `"make"` command is also necessary and can be installed by the `"build-essentials"` package. Lastly, the following script needs to be run to enable the environment: `"/tools/Xilinx/Vivado/2022.2/settings64.sh"`. This script needs to be run every time a new terminal is opened; adding `"source /tools/Xilinx/Vivado/2022.2/settings64.sh"` to the `"~/.bashrc"` file can automate this process.

Validation

The hardware needs to be validated first. Setup the ZedBoard as follows:

- Ensure the AD-FMCOMMS3-EBZ is connected and secured to the ZedBoard's FMC slot.
- Jumpers are installed for pins JP2 and JP6.

- Jumpers connect ground to pins JP11, JP8, and JP7.
- Jumpers connect 3.3V to pins JP10 and JP8.
- The dip switches, from left to right, are 11000011, where a 1 is the up position and a 0 is the down position.
- Make sure the power switch is off and then connect the DC power.
- Plug in an ethernet cable and HDMI cable.
- Plug in a micro-USB hub to the J13 USB port and plug in a mouse and keyboard into the hub.

Next the SD card needs to be prepared. Plug in the SD card provided with the AD-FMCOMMS3-EBZ to the host computer, and copy the following files from the “BOOT” partition to the root of the partition:

- “zynq-zed-adv7511-ad9361-fmcomms2-3-4/BOOT.BIN”
- “zynq-zed-adv7511-ad9361-fmcomms2-3-4/zynq-zed-adv7511-ad9361-fmcomms2-3/devicetree.dtb”
- “zynq-common/ulmage”

Finally, eject the SD card from the host computer and plug it into the ZedBoard. And then set the power switch to the ON position. The green power LED should turn on immediately, and the blue “done” LED should turn on after a few seconds. The connected screen should then turn on and show a Linux GUI.

For future steps, it can be helpful to get the IP address of the ZedBoard. When a Linux desktop is available, open a terminal and use the command “ifconfig”.

Building HDL and Boot File

Next, the unmodified HDL should be built and run to verify the build environment. First clone the HDL repo in appx. 1. The location for the repo was chosen to be “~/source/adi_hdl”. Then navigate to the ZedBoard project directory, “~/source/adi_hdl/projects/fmcomms2/zed”, in the terminal and run the “make” command. This will take about 15 minutes and it should complete with the message shown in Figure 1.

```
Building fmcomms2_zed project [/home/haow6/source/adi_hdl/projects/fmcomms2/zed/fmcomms2_zed_vivado.log] ... OK
haow6@Spire:~/source/adi_hdl/projects/fmcomms2/zed$
```

Figure 1. Message for a successful HDL build.

At this stage it might be desirable to view the block diagram, address space, or other metrics in Vivado. This can be done in the terminal with the following command, “vivado fmcomms2_zed.xpr”

After building the HDL, the boot file needs to be built. First create a new directory for the build artifacts; “~/source/adi_linux” was chosen. In that directory, download the “build_boot_bin.sh” script from this article: <https://wiki.analog.com/resources/tools-software/linux-build/generic/zynq>. Or by using this direct link: “https://raw.githubusercontent.com/analogdevicesinc/wiki-scripts/master/zynq_boot_bin/build_boot_bin.sh”. Then make sure the script is marked as executable. Next turn off the ZedBoard and move the SD card to the host computer. In the “BOOT” partition of the SD card, navigate into the directory, “zynq-zed-adv7511-ad9361-fmcomms2-3-4/”, and open the “bootgen_sysfiles.tgz” archive. Copy the file, “u-boot-zed.elf”, from the archive into the “~/source/adi_linux” directory. Lastly, run the script file with the “fmcomms2_zed.sdk/system_top.xsa” file from the HDL build and “u-boot-zed.elf” file paths as arguments. The following is an example

invocation of the script, assuming that the current working directory is “~/source/adi_linux”, and all other example paths are used; “./build_boot_bin.sh
~/source/adi_hdl/projects/fmcomms2/zed/fmcomms2_zed.sdk/system_top.xsa ~/source/adi_linux/u-boot-zed.elf”.

Once the “BOOT.BIN” file is generated, it should be in the “output_boot_bin” directory. There are 2 ways to upload the boot file to the ZedBoard.

- Via SD card. On the host computer, replace the “BOOT.BIN” file in the root of the “BOOT” partition with the new “BOOT.BIN” file. Insert the SD card into the ZedBoard.
- Via SSH with the “scp” command. The following is a template invocation of the “scp” command assuming the current working directory is “~/source/adi_linux”, and “IP_ADDR” is replaced with the IP address of the ZedBoard; “scp output_boot_bin/BOOT.BIN root@<IP_ADDR>:/boot/BOOT.BIN”. The default password for the root user is “analog”.

After the boot file is replaced, restart the board, and verify that it boots correctly into Linux.

AXI Compatible Core

AXI is used as the communication technology between the microprocessor and peripherals. Each peripheral is mapped to its own address which can be accessed by applications via hardcoded memory addresses.

In ADI’s source code, most peripherals are implemented in its own library and are then parameterized and connected within individual projects. In appx. 1, the “axi_loopback_test” branch implements an example IP core that is compatible with the AXI protocol.

To make a new library, first create a new directory in the library directory. The directory should be named the same as the core, in this case “axi_loopback_test”. Then create a “Makefile” file in that directory. The make file details out the files that should be included when building. The first line of the make file is the library name, it should be the same as the name of the core. Then generic dependencies should include Verilog files that need to be built. Next, Xilinx dependencies include the TCL files for the core. Lastly, a script file is included as boilerplate. Figure 2 shows the entire make file for the AXI Loopback Test core.

```
1  LIBRARY_NAME := axi_loopback_test
2
3  GENERIC_DEPS += ../common/up_axi.v
4  GENERIC_DEPS += axi_loopback_test.v
5
6  XILINX_DEPS += axi_loopback_test_ip.tcl
7
8  include ../scripts/library.mk
```

Figure 2. Make file for AXI Loopback Test.

The next file created is the TCL file, it should be named with the following format: “<Core Name>_ip.tcl”, in this case “axi_loopback_test_ip.tcl”. The first few lines import script files that provide

utility functions. The “adi_ip_create” command is used to create a new core. The “adi_ip_files” command is used to list the Verilog files used in the core; the first argument is the name of the core while the second argument is a list containing file paths. The following 2 lines are boilerplate to finalize and save the core. Figure 3 shows the entire TCL file.

```
1  # ip
2
3  source ../../scripts/adi_env.tcl
4  source $ad_hdl_dir/library/scripts/adi_ip_xilinx.tcl
5
6  adi_ip_create axi_loopback_test
7  adi_ip_files axi_loopback_test [list \
8      "$ad_hdl_dir/library/common/up_axi.v" \
9      "axi_loopback_test.v"]
10
11 adi_ip_properties axi_loopback_test
12
13 ipx::save_core [ipx::current_core]
```

Figure 3. TCL file for AXI Loopback Test.

The last file to create is the Verilog file for the core; “axi_loopback_test.v”. The core should, at the minimum, include the AXI interface ports for a 16-bit address bus and a 32-bit data bus. To handle the AXI protocol, the Microprocessor interface core (up_axi) from ADI is used. This core has a simple interface based on requests and acknowledgements. For a write request, “up_wreq” is asserted on the positive edge of the clock. The core should then read the “up_waddr” and “up_wdata” buses while also asserting “up_wack”. The acknowledgement should be returned on the next clock cycle, but the core has a maximum of 32 clock cycles to record data. For a read request, “up_rreq” is asserted. Then again with

a buffer of 32 clock cycles, the core should read “up_raddr”, set “up_rdata”, and assert “up_rack”. Figure 4 shows the timing diagram for the Microprocessor interface with 1 write and 2 reads [3].

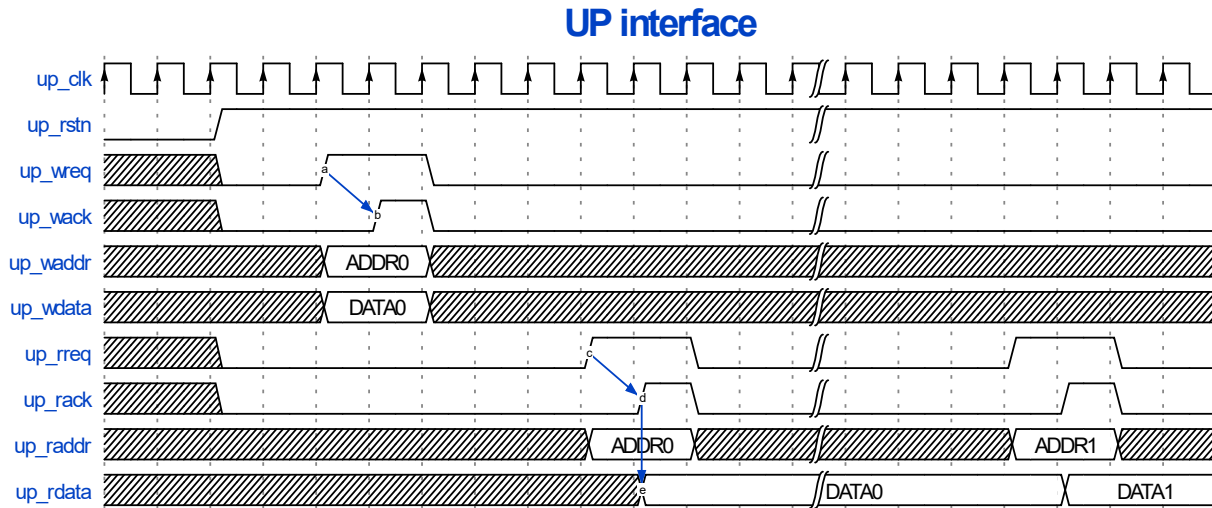


Figure 4. Timing diagram for Microprocessor interface.

Lastly, the library needs to be added to the “clean” and “lib” targets of the library make file. In “~/source/adi_hdl/library/Makefile”, “\$(MAKE) -C axi_loopback_test clean” was added below “clean:” and “\$(MAKE) -C axi_loopback_test” below “lib:”.

In the Verilog module for the core, the Microprocessor interface is instantiated with the AXI bus and relevant output signals. Local parameters are made to define constants for register addresses. Since the microprocessor interface emits DWORD addresses, the address definitions have a bit width of 14. Then 32-bit registers are defined for version, ID, and a scratch variable. The core also defines a controllable reset register that can be controlled by a write request.

For the read interface, it is contained within a positive edge clock process. When the controllable reset is asserted, the read acknowledgement and data signals are set to zero. Otherwise, it checks for a read request, and if present, asserts the read acknowledgement and assigns the register of the corresponding address to the data bus. If no register corresponds to the address, a zero is sent. The write interface is split into 2 processes, register writing, and reset handling. For register writing, when the controllable reset is asserted, registers are set to their default values, otherwise, it checks if there is a write request and if so, assigns the data bus to the corresponding register based on the address. In the other process, it checks if the AXI reset is asserted, and if so, de-asserts the write acknowledgement and asserts the controllable reset. Otherwise, it checks if there is a write request and asserts the write acknowledgement as necessary. If the address corresponds to the reset address, it sets the controllable reset to the value of the first bit of the data bus, otherwise the controllable reset is de-asserted. Figure 5 shows a flowchart of the 3 processes.

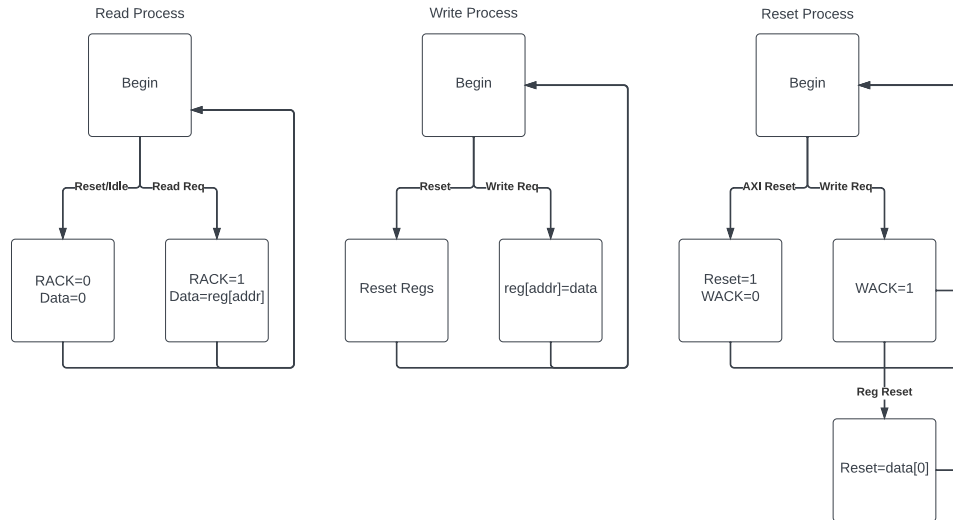


Figure 5. Flowchart for the 3 processes in AXI Loopback Test.

At this point, the custom IP core can be added to the project. In the ZedBoard project directory, edit the make file and add the core to the library dependencies. Then in the “system_bd.tcl” file instantiate and connect the core. The first command used is “ad_ip_instance” which creates the core; the first argument is the name of the core, and the second argument is the name of the core instance. The “ad_connect” command is used to connect wires and buses; the first argument is the source while the second argument is the destination. For the AXI Loopback Test core, the clock and reset buses need to be connected to the system. The “ad_cpu_interconnect” is then used to connect the AXI bus of the core to the interconnect; the first argument is the starting address of the memory space the core should be mapped to, and the second argument is the name of the core instance. The memory address should be chosen to avoid overlapping address, in this case the address 0x44000000 was chosen. And since the address bus is 16-bits wide, the high address is 0x4400ffff. Figure 6 shows the entire “system_bd.tcl” file with the core instantiated and connected.

```

You, 33 minutes ago | 9 authors (You and others)
1 source $ad_hdl_dir/projects/common/zed/zed_system_bd.tcl
2 source ../common/fmcomms2_bd.tcl
3 source $ad_hdl_dir/projects/scripts/adi_pd.tcl
4
5 set mem_init_sys_path [get_env_param ADI_PROJECT_DIR "" ]mem_init_sys.txt;
6
7 # lookback_test
8 ad_ip_instance axi_loopback_test axi_loopback_test_0
9 ad_connect sys_cpu_clk axi_loopback_test_0/s_axi_aclk
10 ad_connect sys_cpu_resets axi_loopback_test_0/s_axi_aresetn
11 ad_cpu_interconnect 0x44000000 axi_loopback_test_0
12
13 #system ID
14 ad_ip_parameter axi_sysid_0 CONFIG.ROM_ADDR_BITS 9
15 ad_ip_parameter rom_sys_0 CONFIG.PATH_TO_FILE "[pwd]/$mem_init_sys_path"
16 ad_ip_parameter rom_sys_0 CONFIG.ROM_ADDR_BITS 9
17
18 sysid_gen_sys_init_file
19
20 ad_ip_parameter axi_ad9361 CONFIG.ADC_INIT_DELAY 23
21
22 ad_ip_parameter axi_ad9361 CONFIG.TDD_DISABLE 1
23
24

```

Figure 6. System board with AXI Loopback Test instance.

Lastly, follow the same procedure above to build the HDL and boot file and to upload it to the ZedBoard.

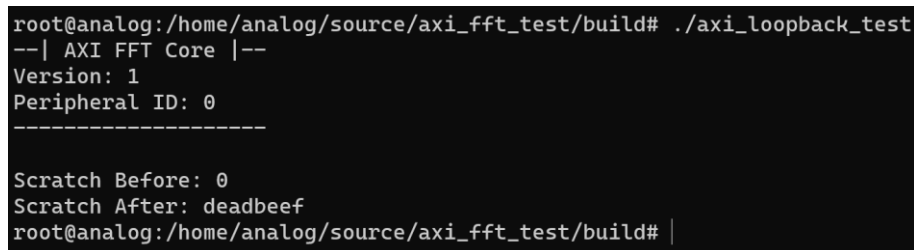
To access the core in Linux, the memory file is mapped into the application and is accessed directly with read and writes. This method has several disadvantages, such as no interrupt signaling, no multi-access protection, and worst of all, requires the application to be run as the root user. This method is only used due to how easy and fast it is to set up, in other words, a kernel driver was not needed.

For the test applications in appx. 2, C++ 20 was selected as the language, and building is organized using CMake. To build them, first clone the repo in appx. 2. Then download the toolchain archive, https://developer.arm.com/-/media/Files/downloads/gnu-a/8.3-2019.03/binrel/gcc-arm-8.3-2019.03-x86_64-arm-linux-gnueabi.tar.xz, and extract it to a directory named “toolchain” inside the repo directory. Then, assuming the current directory is in the repo, run the following command to build; “cmake -B build && cd build && make”.

The test application for the AXI Loopback Test core works as follows:

1. Define statements are used to define the address of the IP core and register DWORD offsets.
2. The file “/dev/mem” is opened with read and write permissions. If it fails to open the memory file, it prints an error message and aborts.
3. The device memory is mapped into the application memory using mmap. The mapping has read and write permissions with the “MAP_SHARED” flag enabled.
4. The core is reset, and the version and ID registers are read. Then the scratch register is read from, written to, and read from again.
5. The memory map is unmapped, and the memory file is closed.

Figure 7 shows the output of the test program.

A terminal window showing the output of the AXI Loopback Test program. The prompt is root@analog: /home/analog/source/axi_fft_test/build#. The program output is: --| AXI FFT Core |--, Version: 1, Peripheral ID: 0, followed by a separator line. Then it shows Scratch Before: 0 and Scratch After: deadbeef. The prompt returns to root@analog: /home/analog/source/axi_fft_test/build#.

```
root@analog: /home/analog/source/axi_fft_test/build# ./axi_loopback_test
--| AXI FFT Core |--
Version: 1
Peripheral ID: 0
-----
Scratch Before: 0
Scratch After: deadbeef
root@analog: /home/analog/source/axi_fft_test/build#
```

Figure 7. Output of the test program for the AXI Loopback Test.

AXI FFT Core

In addition to the AXI Loopback Test core, the AXI FFT core was made, which is available on the “axi_fft” branch of appx. 1. This core provides an AXI compatible interface to the Xilinx LogiCORE IP FFT core to compute the FFT. Like the AXI Loopback Test core, a new library was set up. The new core is split into one top IP with 3 sub-IPs. The top IP interfaces with the AXI bus for register access, while the sub-IPs contain block RAM for data storage and control the AXI4Stream (AXIS) protocol to the Xilinx FFT core.

The interface for the Xilinx FFT core contains 2 slave AXIS channels for input data and configuration data and 1 master AXIS channel for output data. In addition, the core was configured to use a transform size of either 8 or 1024, IEEE 32-bit floating point samples, and the inclusion of a reset port.

Compared to the AXI protocol the AXIS protocol is simpler and unidirectional. There are 4 signals, tvalid, tready, tlast, and tdata. At the start of the transaction, the master asserts tvalid and sets tdata. Then the slave asserts tready to complete a transaction, and the whole process is repeated. At the last transaction, the master should assert tlast. If the master and slave are both able to write and record transactions within a clock cycle, transactions can occur on every clock cycle. Figure 8 shows the timing diagram for the AXIS protocol.

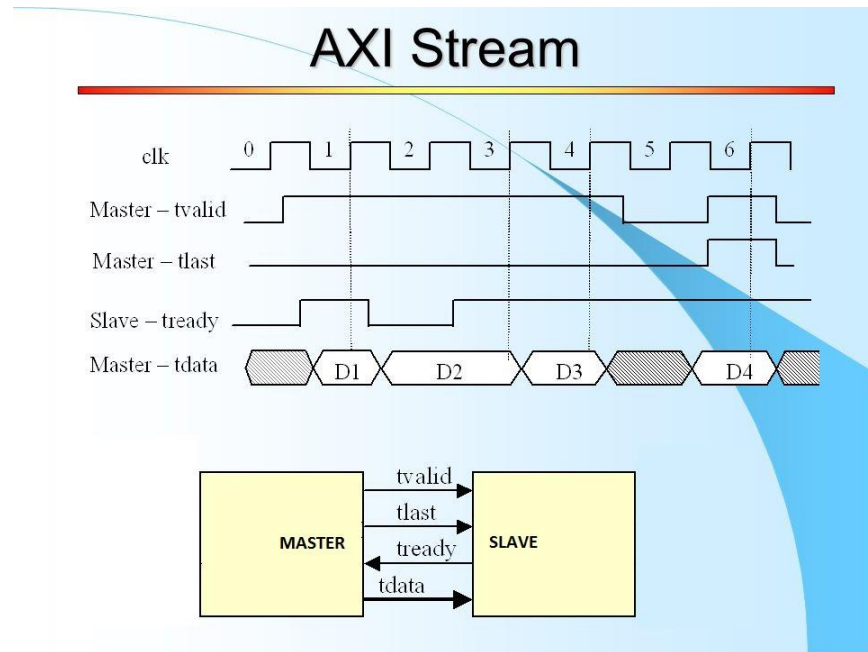


Figure 8. Timing diagram for AXIS transactions.

For the configuration channel of the Xilinx FFT core, there is only one transaction with the width of tdata depending on the configuration. Tdata first contains a forward or inverse bit to configure if the forward or inverse transformation should be done. Following that are bits for the scale schedule, the width of which is determined by Equation 1, where NFFT is the base 2 logarithm of the transform size.

$$2 \times \text{ceil}\left(\frac{NFFT}{2}\right)$$

Equation 1. Scale Schedule width.

Each pair of bits in the scaling schedule determines how the data is scaled between each stage of the core. The default scaling schedule is 1/n, which corresponds to [10, 10, ..., 10]. When doing the forward transformation, it is common to not have normalization, for which the scaling schedule should be set to all zeros. Figure 9 shows the tdata format, with dotted lines being optional.

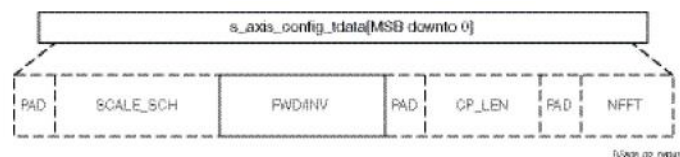


Figure 9. TDATA format for config channel.

For the input channel, there are 2^{NFFT} transactions. For each transaction, the first 32-bits correspond to the real component of the data sample, and the next 32-bits correspond to the imaginary component. As a result, the width of tdata is 64 bits. The output channel also follows the same data format. Figure 10 shows the TDATA format. The padding can be ignored as data samples are byte aligned, and only one processing channel is in use.

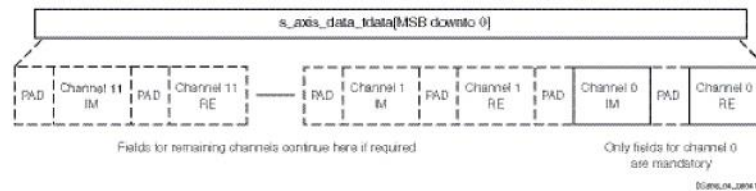


Figure 10. TDATA format for input and output channels.

Further details regarding the 3 channels and their optional fields are available on the product guide document from Xilinx [4].

The sub-IP core, “fft_config”, is used to control the configuration channel. The interface to the core contains the master AXIS port, scale schedule signal, forward/reverse signal, and a one clock cycle commit trigger signal. The core itself is a simple 2-state state machine. It transitions from the IDLE state to the TRANSMIT state when the commit signal is high, and transitions from the TRANSMIT state to the IDLE state when the last transfer is completed. In the IDLE state, it de-asserts the AXIS bus. In the TRANSMIT state, it asserts the AXIS bus and sets the data bus with the scale schedule and forward/reverse signals. Figure 11 shows a flowchart of the state machine.

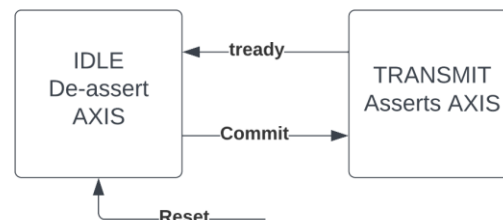


Figure 11. Flowchart for fft_config.

The sub-IP core, “fft_data_input”, is used to control the input channel. The interface to the core contains a RAM write port, master AXIS port, a one clock cycle trigger signal, and a currently streaming signal. The core instantiates 2 block RAMs that was provided by ADI (ad_mem.v). 2 separate RAMs, one for real and one for imaginary samples, are needed to support reading 2 values within the same clock cycle. The write interface selects which RAM block to use based on if the address is even, even for real samples and odds for imaginary samples. The core is a 3-state state machine. It transitions from the IDLE state to the STALL state when the trigger is asserted. It transitions from the STALL state to the STREAMING state in 2 clock cycles. And it transitions from the STREAMING state to the IDLE state when the last transfer is completed. In the IDLE state, the AXIS bus and streaming signal are de-asserted and internal counters are reset. In the first clock cycle of the STALL state, it preloads the AXIS data bus with the first sample pair and increments the RAM address. In the second clock cycle it asserts the tvalid and

streaming signals. This state is needed because both the address and output of the RAM is buffered, causing a 2-clock cycle delay. In the STREAMING state, on every clock cycle, the next sample is loaded, and the RAM address is incremented. If it is the last transaction, the tvalid and streaming signals are de-asserted. Figure 12 shows a flowchart of the state machine.

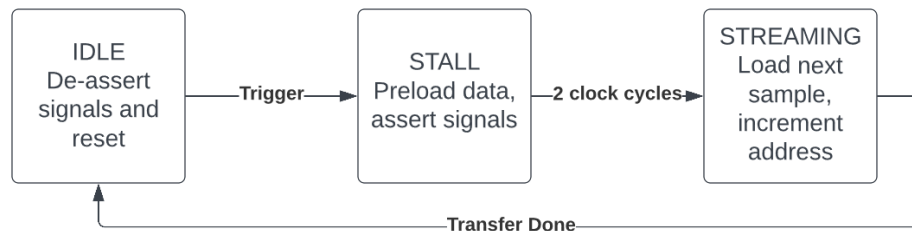


Figure 12. Flowchart for `fft_data_input`.

The sub-IP core, “`fft_data_output`”, is used to consume output data samples and to store them. The interface to the core contains the slave AXIS port, a RAM read port, and a one clock cycle received signal. Like `fft_data_input`, 2 RAM blocks are instantiated to support writing 2 values at the same time. The core is a 4-state state machine. It transitions from the IDLE state to the BEGIN state when tvalid is asserted. It transitions from the BEGIN state to the RECEIVING state on the next clock cycle. It transitions from the RECEIVING state to the done state when all transfers are completed. And it transitions from the DONE state to the IDLE state on the next clock cycle. In the IDLE state, it de-asserts the tready and received signals. In the BEGIN state it asserts the tready signal. In the RECEIVING state, it stores the samples from the data bus into RAM and increments the counters. In the DONE state, it asserts the received signal, which is only asserted for one clock cycle because it immediately transitions back to the IDLE state. Figure 13 shows a flowchart of the state machine.

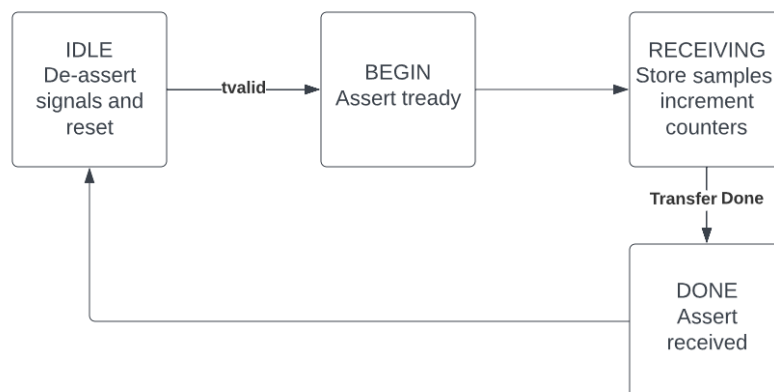


Figure 13. Flowchart for `fft_data_output`.

The top-IP core, “`axi_fft`”, instantiates the Microprocessor interface and the 3 sub-IP cores. Its interface includes the AXI slave port, controllable reset for the xilinx FFT core, master AXIS ports for the input and configuration channels, and a slave AXIS port for the output channel. Table 1 shows the register mapping for the core.

Table 1. Register map for AXI FFT.

Register	DWORD Address (hex)	Default	Access
Version	00	1	RO
Peripheral ID	01	0	RO
Scratch register	02	0	RW
Identity	03	0x46465443 (FFTC)	RO
NFFT	04	3	RO
FFT Config	10	1	RW
Status	11	0	RO
Reset	20	NA	WO
Input trigger	21	NA	WO
Config trigger	22	NA	WO
Input data start	40	NA	WO
Output data start	Input data start + number of samples	NA	RO

The first bit of the status register signals when a transfer from the Xilinx FFT core to the `fft_data_output` core was completed. The input trigger, when written to with any value, asserts the trigger signal for the `fft_data_input` core, which causes stored samples to be sent the Xilinx FFT core. The configuration trigger, when written to with any value, asserts the trigger signal for the `fft_config` core, which causes the current configuration to be sent to the Xilinx FFT core. Input data is sequenced and accessed in interwoven IQ samples, i.e., $\{Re_0, Im_0, Re_1, Im_1, Re_2, Im_2, \dots\}$. Immediately after the input samples are the output samples with the same sequencing as the input data.

Register read requests are handled in one process. If the controllable reset is asserted, the read acknowledgement and data bus are de-asserted. If there is a read request and the address corresponds to the output data RAM region, the first clock cycle stalls the request to allow the buffered RAM to update, and the second clock cycle asserts the acknowledgement and data bus with the requested data. If there is a read request and the address does not correspond to the output data RAM region and instead to a register, the read acknowledgement is asserted, and the corresponding register is assigned to the data bus. When there is not a read request, the read acknowledgement and data bus are de-asserted. Figure 14 shows the flowchart for the read process.

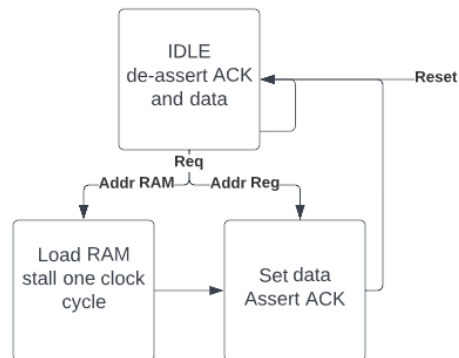


Figure 14. Flowchart for read process of AXI FFT.

Register write requests are split into 5 processes. For normal registers like scratch and FFT config, during controlled reset, they are set to their default values, otherwise if there is a write request, the corresponding register is set to the data bus. For the input RAM, if there is a write request to the RAM address region, the address, data, and enable lines for the RAM port in the `fft_data_input` core are asserted. Otherwise, they are all de-asserted. For triggers, if there is a write request and the address corresponds to a trigger, that trigger is asserted. Otherwise, all triggers are de-asserted. This applies to the input trigger and configuration trigger. For the status register, if controllable reset is asserted, the register is set to the default value. If the received signal is asserted by the `fft_data_output` core, the first bit of the register is asserted. If there is a read request and the address corresponds to the data input trigger, the first bit of the register is de-asserted. Lastly, for the controllable reset, during AXI reset, the write acknowledgement is de-asserted, and the controllable reset is asserted. Otherwise, if there is a write request the write acknowledgement is asserted. If the write address corresponds to the reset address, the first bit of the data bus is assigned to the controllable reset register.

Additionally, in the `"axi_fft_ip.tcl"` file, AXI buses need to be defined to allow the `"ad_connect"` command to work. This can be done via the `"adi_add_bus"` command. The first argument is the prefix of the bus name, second is whether the bus port is a slave or master, third is the abstraction type of the bus, fourth is the bus type, and fifth is a mapping of individual port names to their type. Furthermore, a clock needs to be associated to each of the buses, which can be done via the `"adi_add_bus_clock"` command. The first argument is the name of the clock source, second is a colon separated list of buses to associate, and third is an optional reset signal. Figure 15 shows the bus definitions and clock associations.

```

17  ## Interface definitions
18  adi_add_bus "m_axis_i" "master" \
19      "xilinx.com:interface:axis_rtl:1.0" \
20      "xilinx.com:interface:axis:1.0" \
21      {
22          {"m_axis_i_tready" "TREADY"} \
23          {"m_axis_i_tvalid" "TVALID"} \
24          {"m_axis_i_tlast" "TLAST"} \
25          {"m_axis_i_tdata" "TDATA"} \
26      }
27
28  adi_add_bus "s_axis_o" "slave" \
29      "xilinx.com:interface:axis_rtl:1.0" \
30      "xilinx.com:interface:axis:1.0" \
31      {
32          {"s_axis_o_tready" "TREADY"} \
33          {"s_axis_o_tvalid" "TVALID"} \
34          {"s_axis_o_tlast" "TLAST"} \
35          {"s_axis_o_tdata" "TDATA"} \
36      }
37
38  adi_add_bus "m_axis_c" "master" \
39      "xilinx.com:interface:axis_rtl:1.0" \
40      "xilinx.com:interface:axis:1.0" \
41      {
42          {"m_axis_c_tready" "TREADY"} \
43          {"m_axis_c_tvalid" "TVALID"} \
44          {"m_axis_c_tlast" "TLAST"} \
45          {"m_axis_c_tdata" "TDATA"} \
46      }
47
48  adi_add_bus_clock s_axi_aclk s_axi:m_axis_i:s_axis_o:m_axis_c s_axi_aresetn

```

Figure 15. Bus definitions for the AXI FFT core.

The AXI FFT core can now be added to the ZedBoard project. First the AXI FFT core was defined with an NFFT of 10, and a Xilinx FFT core was defined with a transform length of 1024. Then the clock and reset signals were connected to both cores, and the AXI FFT core was assigned the address 0x44000000. Lastly, the AXIS ports were connected between the AXI FFT core to the Xilinx FFT core.

Testbenches

Testbenches are an integral part of designing and testing a core before running it on hardware. The testbenches made can be found in appx. 3, in the “axi_fft” branch. To begin, clone the testbench repo in appx. 3 as a submodule to the directory, “testbenches”. If the current directory is the HDL repo, the following git command can be used; “git submodule add https://github.com/arandomdev/adi_testbenches ./testbenches”. Then create a new directory in the testbenches directory for a new testbench, in this case it will be called “axi_fft2”.

There are 6 files needed to create a testbench. The make file was copied from another testbench (axi_tdd) and customized. The “SV_DEPS” variable contains all the SystemVerilog files that need to be compiled but excludes the individual test programs in the “tests” subdirectory. The “ENV_DEPS” variable contains the TCL files needed for the testbench. And the “LIB_DEPS” variable includes the library IP cores that are needed for the testbench, in this case only AXI FFT is needed.

The “system_project.tcl” file defines the Vivado project parameters. It was also copied from axi_tdd and customized. The part number in the “adi_sim_project_xilinx” command was changed to “xc7z020clg484-1” to match that of the ZedBoard.

The “system_bd.tcl” file defines the board setup and connections. First the “adi_env.tcl” file was included to provide utilities and commands. Then global variables for core parameters are declared using the “global” keyword, in this case “axi_fft” and “xfft”. The “ad_ip_instance” command is used to create an instance of the AXI FFT core and the Xilinx FFT core. And lastly, the clock, reset, and AXIS ports are connected, while an address was assigned to the AXI FFT core.

The “system_tb.sv” file contains boilerplate to launch the test programs. While also copied from axi_tdd, it only contains the test program instantiation, and removes the port arguments from the test harness instantiation.

The “cfigs” directory contains any number of potential configurations to test. For each configuration file, a global variable should be set to define the parameters for the instantiated cores. Figure 16 shows the definition of the parameters for the AXI FFT core.

```
3  set axi_fft [list \  
4      NFFT 3 \  
5      PERI_ID 0 \  
6      IDENT 0x46465443 \  
7      SCALE_SCH_WIDTH 4 \  
8      CONFIG_WIDTH 8 \  
9  ]
```

Figure 16. Parameters for the AXI FFT core for a testbench.

The “tests” directory contains any number of test programs to be run. The test program for the AXI FFT core works as follows:

1. Define a test vector, which is an 8-point, 1 period cosine wave.
2. Instantiate and start the test harness.
3. Bring up the core by writing to the reset register. The “RegWrite32” function of the AXI management object in the test harness is used.
4. Load the test vector to the core by writing to the input RAM addresses.
5. Trigger a calculation by writing to the input trigger and poll for the status register for the done signal.
6. Read and print out the values from the core. The “RegRead32” function was used.
7. Stop the test harness and testbench.

To run the testbench, navigate to the directory of the testbench and run the “make” command. Figure 17 shows the output of the “make” command on a successful run.

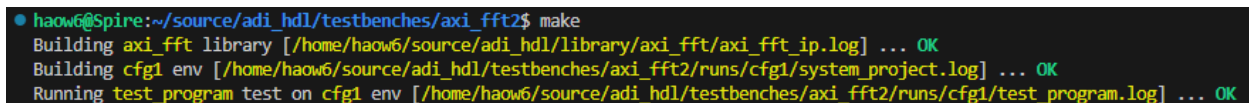
A terminal window showing the output of the 'make' command. The prompt is 'haow6@Spire:~/source/adi_hdl/testbenches/axi_fft2\$'. The output shows three steps: 'Building axi_fft library' with path '/home/haow6/source/adi_hdl/library/axi_fft/axi_fft_ip.log' and status '... OK'; 'Building cfg1 env' with path '/home/haow6/source/adi_hdl/testbenches/axi_fft2/runs/cfg1/system_project.log' and status '... OK'; and 'Running test_program test on cfg1 env' with path '/home/haow6/source/adi_hdl/testbenches/axi_fft2/runs/cfg1/test_program.log' and status '... OK'.

Figure 17. Successful testbench run.

The “test_program.log” file can be opened to view and verify the printed values. To instead open the Vivado interface to access the waveform, use the command, “make MODE=gui”.

Performing an FFT

Another test application was implemented in appx. 2 to perform FFT with a test vector for input samples. Called “axi_fft_test”, the program works as follows:

1. Opens and maps the AXI FFT core into memory and resets it using the reset register.
2. By reading the NFFT register, it calculates the transform size and the address of the output RAM.
3. Prints out information regarding the core and loads a test vector that matches the transform size.
4. Sets the configuration to a scaling schedule of zero and to calculate the forward transform and commits it.
5. Loads the test vector into the core and prints out how long it took.
6. Triggers a calculation and prints out how long it took.
7. Offloads the output data into a buffer and prints out how long it took.
8. Prints out each output sample and whether it passed a test. If the output sample is within 1×10^{-4} of the precomputed sample, it is considered a pass. The noise in the results is introduced by the finite bit width numbers used in the core. As such, even a real valued input will result in miniscule exponential components [4].

The test vector was generated using Python and corresponds to a cosine wave with a single period. Figure 18 shows the output of the test application using a transform size of 8.

```

root@analog:~# /home/analog/source/axi_fft_test/build/axi_fft_test
--| AXI FFT Core |--
Version: 1
Peripheral ID: 0
Identity: 46465443
NFFT: 3
Point Size: 8
-----

Setting config (No norm, forward)

Loading test vector (Cosine, 8 Point)
Took: 0.006348ms

Trigger and wait for compute
Took: 0.00315ms

Offloading data
Took: 0.014634ms


```

Index	Output	Key	Difference	Test
0	-0.0000002086	-0.0000002106	0.0000000020	Pass
1	0.0000000000	0.0000000000	0.0000000000	Pass
2	4.0000000000	4.0000000000	0.0000000000	Pass
3	-0.0000001192	-0.0000001130	-0.0000000063	Pass
4	0.0000000298	0.0000000318	-0.0000000020	Pass
5	-0.0000000596	-0.0000000596	0.0000000000	Pass
6	-0.0000000596	0.0000001192	-0.0000001788	Pass
7	-0.0000001788	-0.0000002242	0.0000000454	Pass
8	0.0000001490	0.0000001470	0.0000000020	Pass
9	0.0000000000	0.0000000000	0.0000000000	Pass
10	-0.0000000298	0.0000001192	-0.0000001490	Pass
11	0.0000001788	0.0000002242	-0.0000000454	Pass
12	0.0000000298	0.0000000318	-0.0000000020	Pass
13	0.0000000596	0.0000000596	0.0000000000	Pass
14	4.0000000000	4.0000000000	0.0000000000	Pass
15	0.0000001192	0.0000001130	0.0000000063	Pass

```

root@analog:~#

```

Figure 18. Output of the AXI FFT core test application.

Performance Testing

To test the performance of the AXI FFT core, another test application was implemented. Called “speedTest”, the program works as follows:

1. Opens and maps the core into memory and resets it.
2. Reads the configured NFFT value and calculates the transform size and output data address.
3. Prints out information about the core.
4. Sets the scale schedule to zero and to calculate the forward transformation.
5. Logs the current time.
6. Loads the test vector onto the core.
7. Repeatedly triggers a calculation for a set number of iterations.
8. Offloads the output data onto an unused buffer.
9. Log the current time and print out the difference.

The time to repeatedly load and offload data onto the core is not considered because the overhead from the up_axi IP is extreme and would provide accurate results for the performance of the core. And while not considering the time to load and offload data is also incorrect, it is more representative of the real-world performance.

Another test application was implemented to gather results for a FFT performed in software. Using the “scipy.fftpack.fft” function in the SciPy library for python, the FFT was repeatedly calculated with an identical test vector. 4 tests were done in both applications, with NFFTs of 3 and 8, and iterations of 16384 and 65536.

Results

Table 2 shows the results of the 4 tests. There does not seem to be a noticeable difference in speed up times when performing 16384 iterations versus 65536 iterations, suggesting that the compute time increases linearly with the number of iterations. But when NFFT increases from 3 to 10, the speed up decreases from about 60 times to 5.5 times, which is expected as the time of complexity of performing an FFT is $O(N\log(N))$.

Table 2. Speed results comparing FPGA to Python.

Iterations	NFFT	Time (ms)		Speed up
		Python	FPGA	
16384	3	1322.28	22.27	59.38
65536	3	5356.83	88.65	60.43
16384	10	3852.80	704.47	5.47
65536	10	15518.26	2821.15	5.50

Regarding the FPGA utilization of the cores, Table 3 shows the utilization for 3 main resources, LUTs, RAM, and DSP slices, when a 1024-point FFT core is configured. LUT and DSP usage is very low or nonexistent for the AXI FFT core as it just acts as an interface the Xilinx FFT core. Block RAM is heavily used however as it requires storage for the samples. The Xilinx FFT core uses much more resources, most importantly it uses 15% of the DSP slices for a moderate transform size.

Table 3. FPGA utilization of the AXI FFT and Xilinx FFT cores.

Core	LUTs (%)	Block RAM (%)	DSP (%)
AXI FFT	0.38	2.86	0
Xilinx FFT	8.81	6.07	15.45

Conclusion

A custom FFT core was successfully implemented and tested. It shows a speed up of about 5.5x when compared to the traditional software implementation. Its FPGA utilization is relatively small and provides room for even higher transform sizes. Additionally, a workflow was created to aid in the development of custom IP cores.

Future improvements to the AXI FFT core would most definitely include using direct memory access (DMA) for the loading and offloading of data. ADI even provides a versatile DMA core that could theoretically directly connect to the Xilinx FFT core for this purpose. Alternatively, the configuration for the Xilinx FFT core can be optimized. The current configuration has the highest throughput but also has the highest resource utilization. One such change is switching to fixed point samples with lower precision, this will reduce FPGA utilization and allow for higher transform sizes. Additionally, a Linux driver can be developed so that applications do not need root permissions to access the IP cores and its functions.

References

- [1] Ettus Research, "RFNoC™ (RF Network on Chip)," Ettus Research An NI Branch, [Online]. Available: <https://www.ettus.com/sdr-software/rfnoc/>. [Accessed 18 April 2023].
- [2] P. Pop, "ADI™ Reference Designs HDL User Guide," Analog Devices, 17 February 2023. [Online]. Available: <https://wiki.analog.com/resources/fpga/docs/hdl>. [Accessed 18 April 2023].
- [3] I. Csomortani, "Microprocessor interface," Analog Devices, 27 June 2017. [Online]. Available: https://wiki.analog.com/resources/fpga/docs/up_if. [Accessed 22 April 2023].
- [4] AMD, "Fast Fourier Transform v9.1 LogiCORE IP Product Guide," 4 May 2022. [Online]. Available: <https://docs.xilinx.com/r/en-US/pg109-xfft/Fast-Fourier-Transform-v9.1-LogiCORE-IP-Product-Guide>. [Accessed 23 April 2023].

Appendix

- 1. https://github.com/arandomdev/adi_hdl. Link to the repo with modified HDL code, relevant branches are "axi_loopback_test" and "axi_fft". Both branches were forked from the source repo, <https://github.com/analogdevicesinc/hdl>, at the commit 890569d53fc08e52a7a084e864b4fd268ba2fffe.
- 2. https://github.com/arandomdev/axi_fft_test. Link to the repo with test applications for the AXI Loopback Test core and AXI FFT core.
- 3. https://github.com/arandomdev/adi_testbenches. Link to the repo with testbenches, the relevant branch is "axi_fft". The branch was forked from the source repo, <https://github.com/analogdevicesinc/testbenches>, at the commit 40e1598d7e866dde8281ef254335715c37c7642d.