ENGG-463 Lab 901 Report

Kirk A. Sigmon, kirk.a.sigmon.th@dartmouth.edu

March 11, 2025

Contents

1 Introduction

This report overviews a Digital Signal Processing ("DSP") circuit, implemented on the Zybo Z7-20 Field Programmable Gate Array ("FPGA") development board, that enables selective filtering of input audio data.

This paper provides an overview of the following:

- **The design process for the circuit**, including among other topics the development of AXI receivers/transmitters, a Serial Peripheral Interface ("SPI") controller, Inter-Integrated Circuit Sound ("I2S")-compliant audio transmitters and receivers, AXI-compatible First-In, First-Out ("FIFO")

transmitters, and more, including paper designs and Register Transfer Level ("RTL") schematics for the same;

- **Implementation details**, including discussions of VHSIC Hardware Description Language ("VHDL") code, high-level circuit designs, and C code implemented via the Vitis platform; and
- **Testing results**, including testbench simulation results, testing using an oscilloscope and waveform generator, and descriptions of real-world music filtering performance.

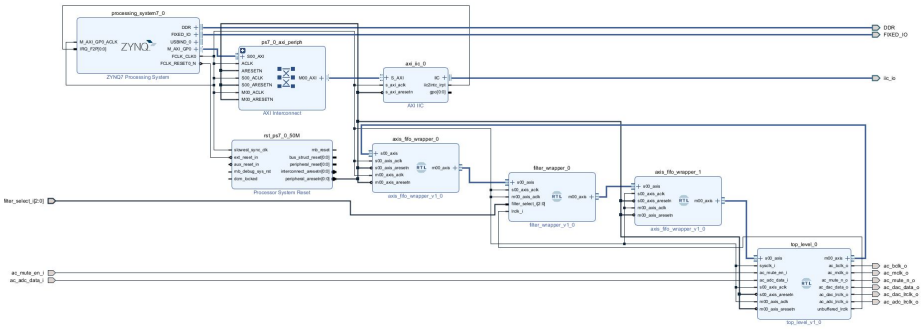
A YouTube video providing a brief summary of these concepts is available at https://youtu.be/DoEGDBczHY4

2 Design Process

2.1 Underlying Goals

The goal of this project is to receive input from a first 3.5mm audio jack on the Zybo Z7-20 FPGA development board and output filtered (or, if desired, unfiltered) audio on a second 3.5mm audio jack on the same development board. This process necessarily implicates DSP, which itself implies a variety of requirements: the conversion of audio data from an audio codec element (here, an onboard SSM2603 audio codec) into a series of bits while maintaining left/right audio channel separation, the processing of that data using various filters (eight total - four unqiue filters, multiplied by two for each channel), and the development of an AXI-implementing pipeline that respects various inherent parameters of the Zybo Z7-20 FPGA development board and the onboard SSM2603 audio codec (*e.g.,* a 125 MHz fabric clock, compliance with the audio input modes required by the SSM2603 datasheet, use of 24 bits of data stored in a total of 32 bits, etc.).

As a preliminary introduction to many of the components described below, the below RTL schematic provides a high-level idea of the overall circuit. In short, various standard elements (*i.e.,* the Zynq processor on the Zybo Z7-20 FPGA development board in conjunction with an AXI Interconnect, a Processor System Reset element, and an AXI IIC) are implemented alongside a series of elements comprising a "top level" element capable of receiving and transmitting audio coded data by interfacing with an audio codec using I2S, first AXI-compliant FIFO element configured to receive the audio data, an AXI-compliant filter element configured to selectively filter the audio data, and a second AXI-compliant FIFO element configured to receive the filtered audio data and forward it back to the "top level" element for output via an output audio jack on the Zybo Z7-20 FPGA development board.



Note that, in addition to the data transmitted and received via the audio codec (*i.e.,* the data received and transmitted via the 3.5mm audio jacks on the Zybo Z7-20 FPGA development board, such as `ac_adc_data_i`), this circuit is configured to receive a variety of different manual inputs, including a mute toggle driven by a rightmost switch on the development board (`ac_mute_en_i`, where a binary value of 1 permits audio transmission and a value of 0 causes muting) and filter selection input (`filter_select_i`. The filter selection input is driven by the three leftmost switches on the development board as follows:

Switch Values	Corresponding Filter
000	No Filter
001	Low Pass Filter
010	High Pass Filter
100	Band Stop Filter
011	Band Pass Filter
Any Other Value	No Filter

This process also necessarily requires four different clocks. A first clock, the fabric clock for the circuit, is configured via the Zynq processor to be 125 MHz. A second clock, referred to as `MCLK`, is 12.288 MHz and is derived using a clock divider IP provided in the Vivado software platform. Two other clocks are used: `BCLK`, a bit clock used to identify specific bits in transmission in compliance with the SSM2603 audio codec, and `LRCLK`, also referred to as `LR/WS CLK` or `PBLRC`, which defines audio channels (the audio data is for the left channel when the signal is "0" and for the right channel when the signal is "1"). The interrelation of these clocks is depicted in the figure below, taken from the SSM2603 audio codec datasheet.

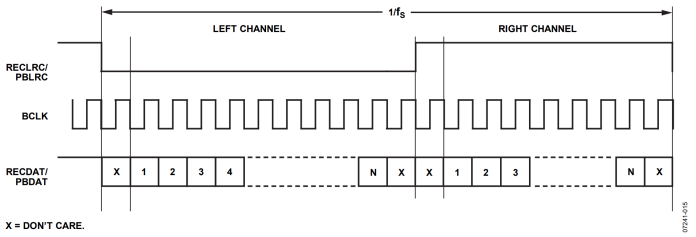
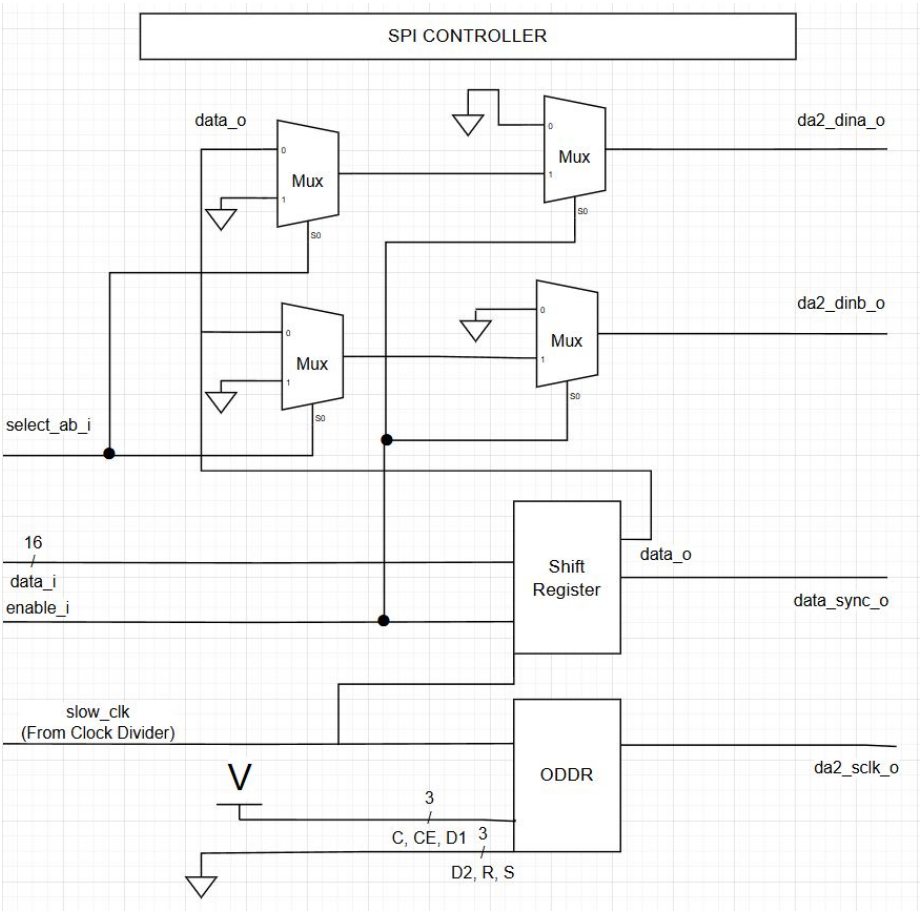


Figure 25. I²S Audio Input Mode

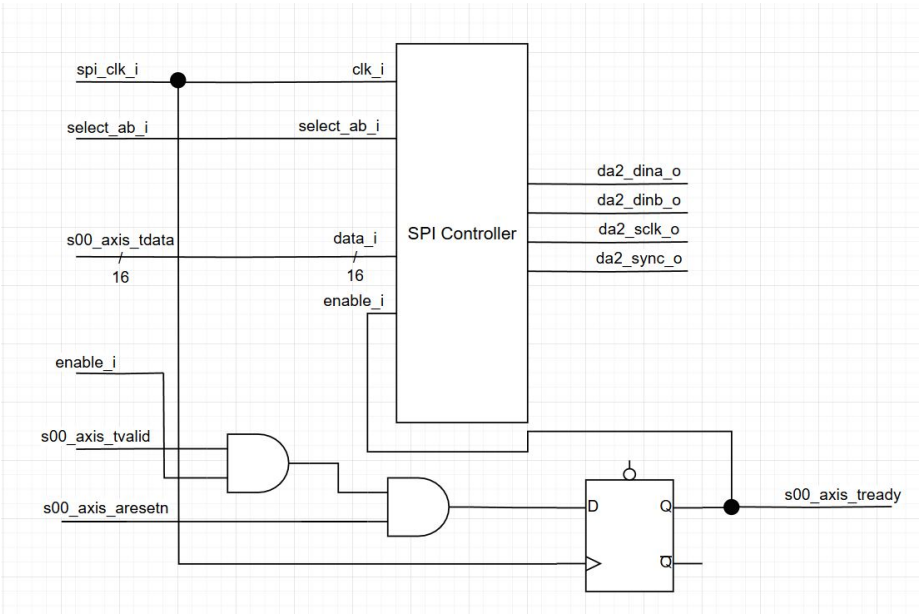
Individual paper designs and RTL schematics for these components are provided in significant detail below.

2.2 SPI Controller and AXI Stream Receiver

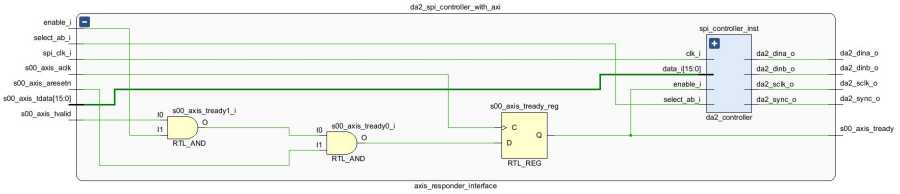
The circuit described above uses a SPI controller to receive data (as input `data_i`) and output that data using one of two outputs (`da2_dina_o` and `da2_dinb_o`) using a shift register. That SPI controller was prototyped using the following paper diagram, which prototyped the circuit at a high level:



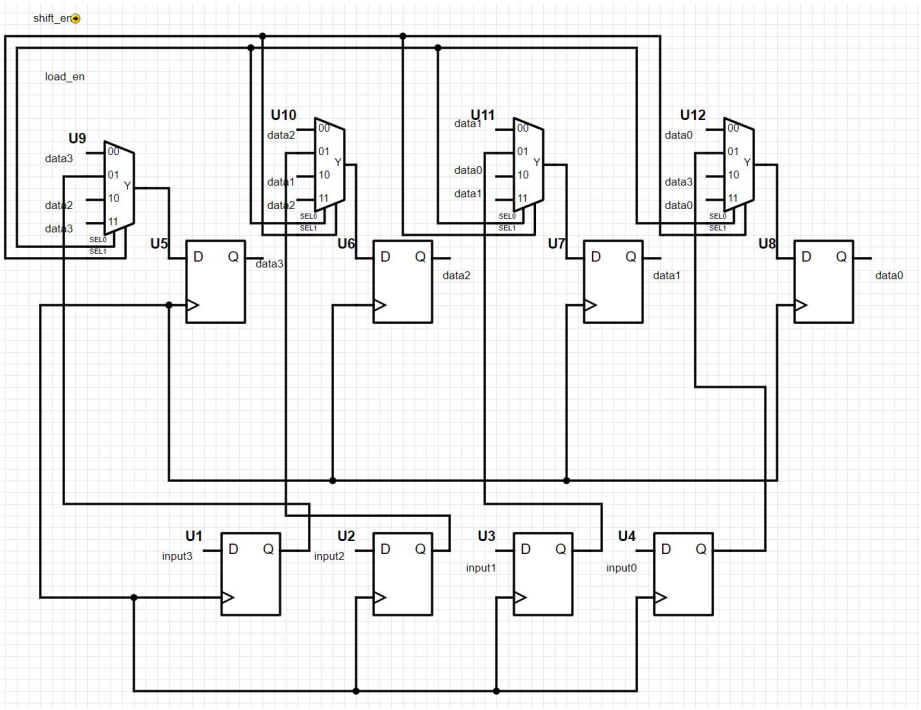
This SPI controller was then wrapped in an AXI stream receiver such that the SPI controller could be controlled using an AXI interface. A prototype paper design of this wrapping is as follows:



The corresponding RTL diagram generated by Vivado, inclusive of such AXI-compatible wrapping, is as follows:



Note that this SPI controller uses a shift register. At a very high level, a shift register (here depicted as shifting four bits) can be represented as follows:

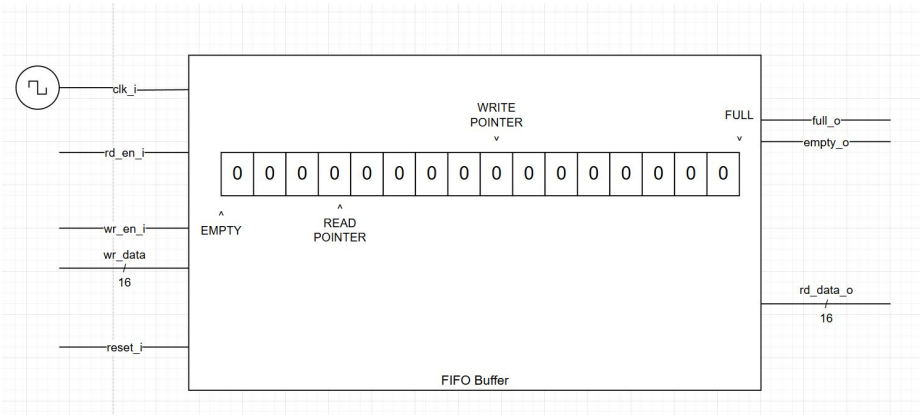


Practically speaking, the shift register is capable of shifting a larger number of bits (*e.g.,* the full 32-bit phrase if necessary, or the 24 bits storing the audio data). As such, this paper design is merely illustrative.

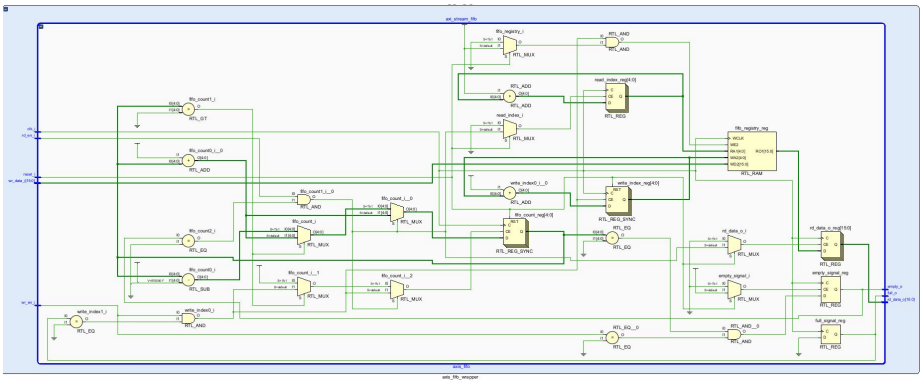
2.3 AXI Stream FIFO

As indicated by the above high-level diagram, two FIFOs were used in this circuit.

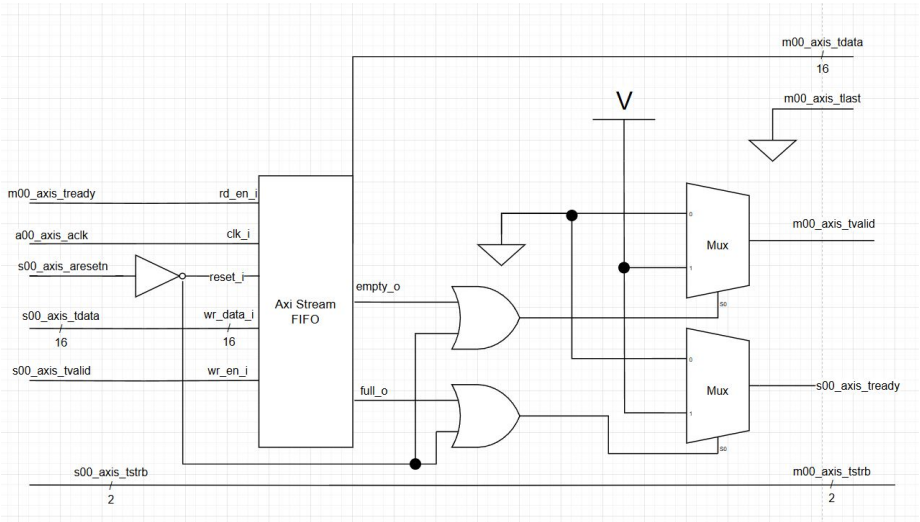
Initially, a standard (non-AXI) FIFO with two separate pointers (that is, the ability to simultaneously read and write) was prototyped on paper. That prototype is depicted in the following paper design, showing enable controls (`wr_en_i` and `rd_en_i`) for writing and reading, a clock input, a reset input, and a data input (`wr_data`) along with output signals including a full signal (`full_o`), an empty signal (`empty_o`), and a data output (`rd_data_o`).



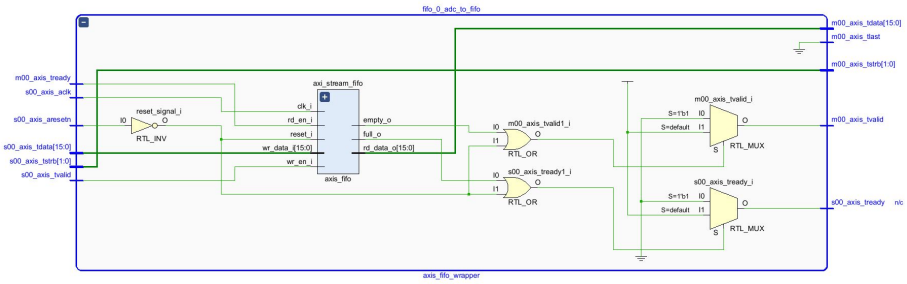
This can be translated into a fairly complex RTL diagram in the Vivado software:



This FIFO can be wrapped and made AXI-compliant fairly simply, with a focus on using the empty and full signals for the purposes of driving valid and ready signals. A paper design along those lines was used to prototype the wrapping process:



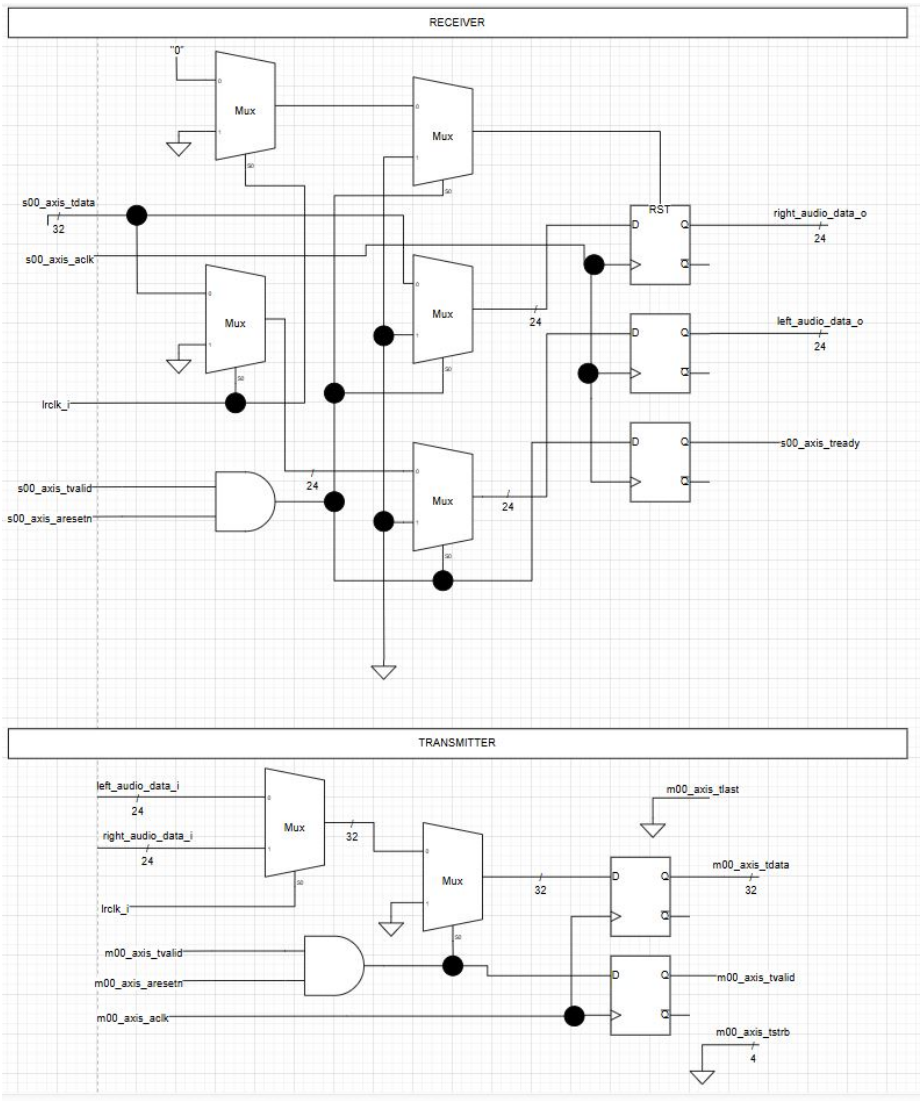
The corresponding RTL schematic generated in Vivado is nearly identical:



2.4 AXI Stream Receiver and Transmitter

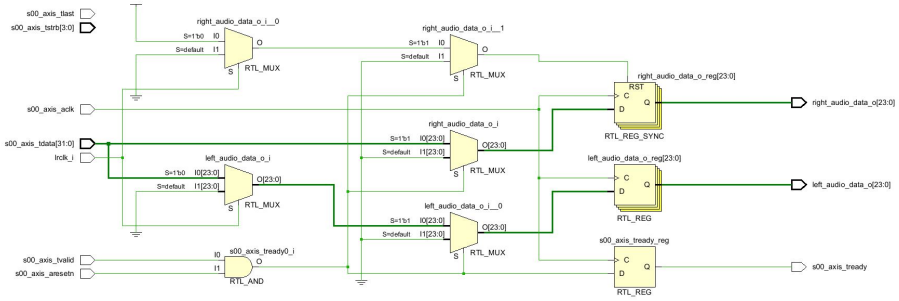
To facilitate input and output of data (*e.g.,* in the filter component), an AXI-implementing stream receiver and an AXI-implementing stream transmitter were programmed. These circuits are, in short, responsible for handling AXI communications. In the case of the AXI stream receiver, the job is (in plain English) to take AXI-compliant instructions and identify left and/or right audio data from the signal `s00_axis_tdata` based on the left-right clock `lrclk_i`. Conversely, in the case of the AXI stream transmitter, left/right audio data is taken along with AXI inputs to output an AXI-compliant data stream (including the data as `m00_axis_tdata`) based on the left-right clock `lrclk_i`. Both circuits also operate in view of the bits available to the circuit, retrieving 24 bits of audio data from a 32 bit "phrase" and/or re-transmitting 24 bits of audio data back into a 32 bit "phrase" as needed.

First, a paper design was prototyped for both circuits:

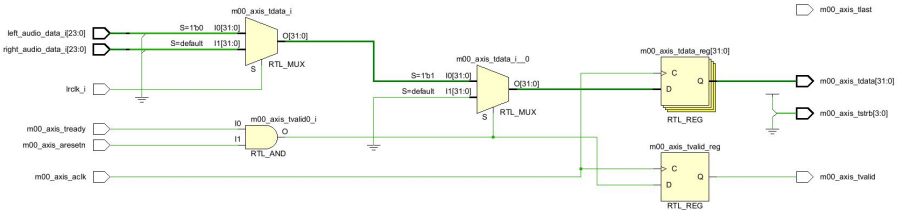


The circuit was then programmed in VHDL. The corresponding RTL schematics generated in Vivado are nearly identical.

I2S Receiver:

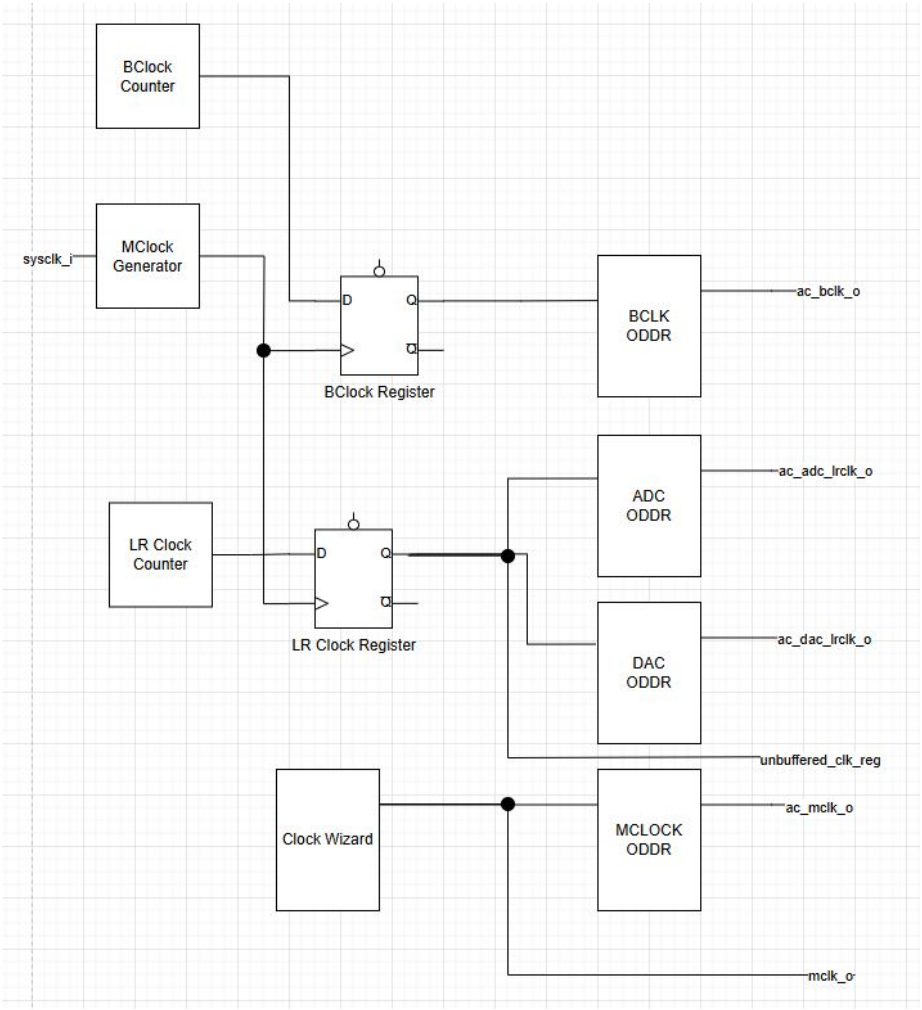


I2S Transmitter:

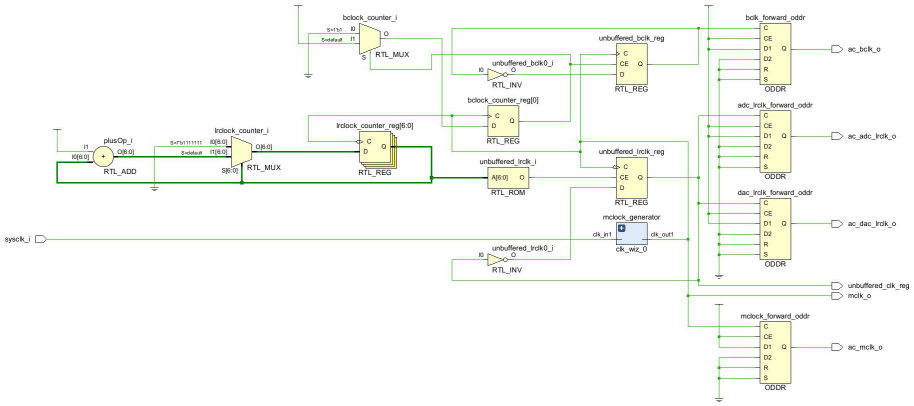


2.5 System Timing

System timing was achieved using the creation of four clocks: the system fabric clock (125 MHz), `MCLK` (12.288 MHz), `BCLK` (which is `MCLK`/4), and `LRCLK` (which is `MCLK`/256). To achieve this goal, clocks were generated using a clock generation circuit. A paper design was first used to prototype the circuit:



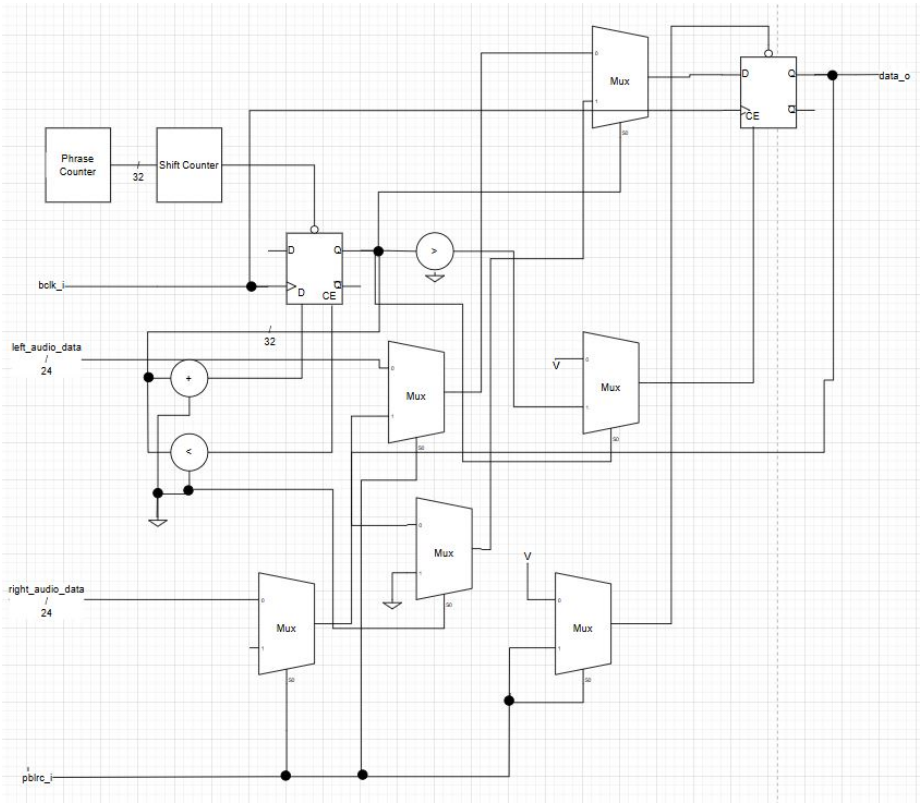
This system was then programmed in VHDL and using IP cores (*e.g.,* the clock generator) in Vivado. The RTL schematic is generally similar, albeit with more detail to reflect how counters are used to generate `BCLK` and `LRCLK`:



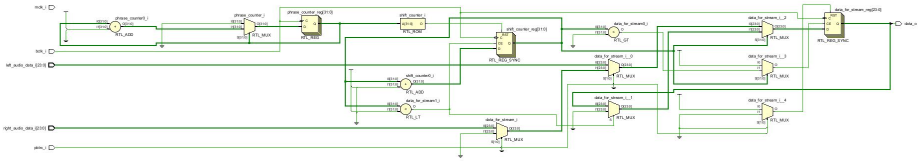
As will be detailed in the testing results section below, these clocks were validated in simulation.

2.6 I2S Transmitter/Receiver

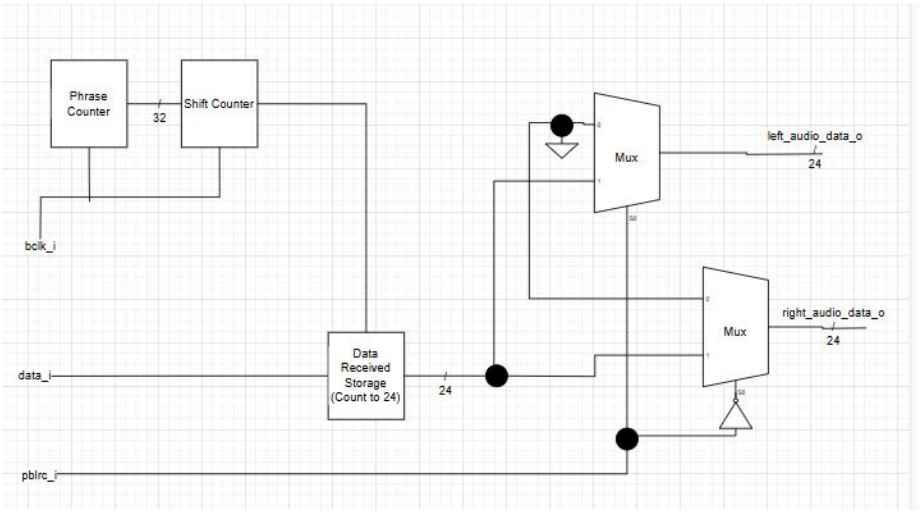
An I2S Transmitter and I2S Receiver were created for the purposes of communicating with the audio codec. A paper design for the I2S transmitter was used to prototype the circuit:



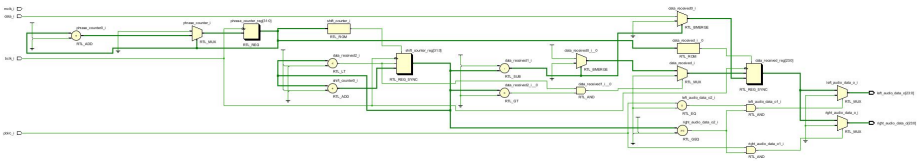
The circuit was then programmed in VHDL. The corresponding RTL diagram for the I2S transmitter provides a bit more detail on use of the shift counter, but is otherwise identical from the purposes of input/output:



The I2S Receiver was significantly simpler. A paper design was first prototyped:

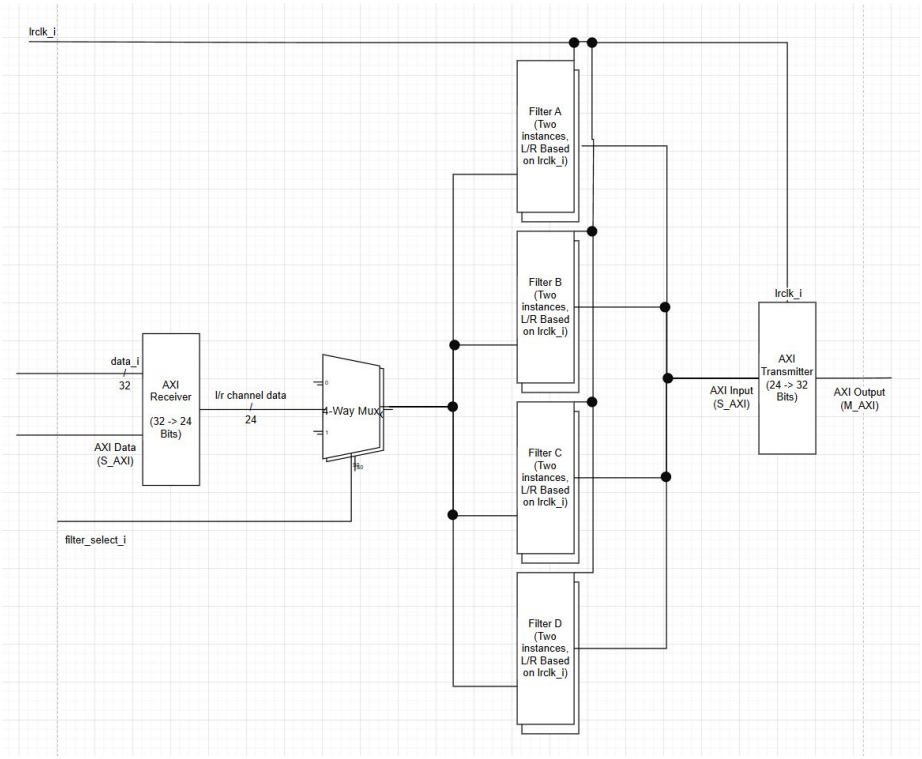


The circuit was then programmed in VHDL. The corresponding Vivado RTL schematic for the I2S receiver is much more complicated, being more precise about the circuitry used to serially collect data from `data_i`, but is otherwise identical from the purposes of input/output:



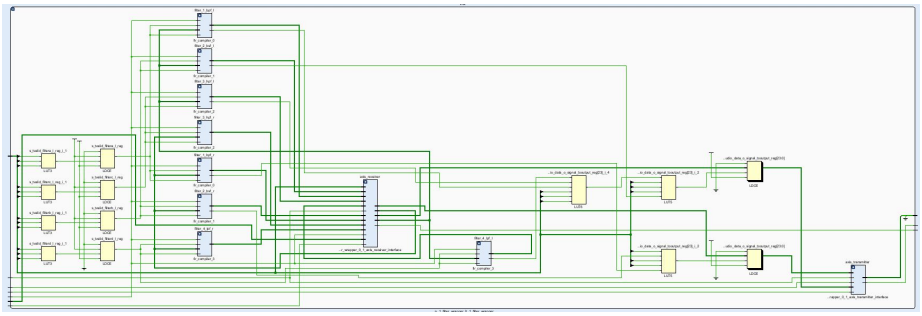
2.7 Filter Wrapper

An AXI-compliant wrapper for the four different filters used in this system was prototyped on paper. This paper design recognizes that the wrapper needs to have an AXI stream receiver and transmitter (discussed above) as well as a multiplexer that routes data to one of four possible filters (for a total of eight, recognizing the left/right channels indicated by `lrclk_i`) based on `filter_select_i`:



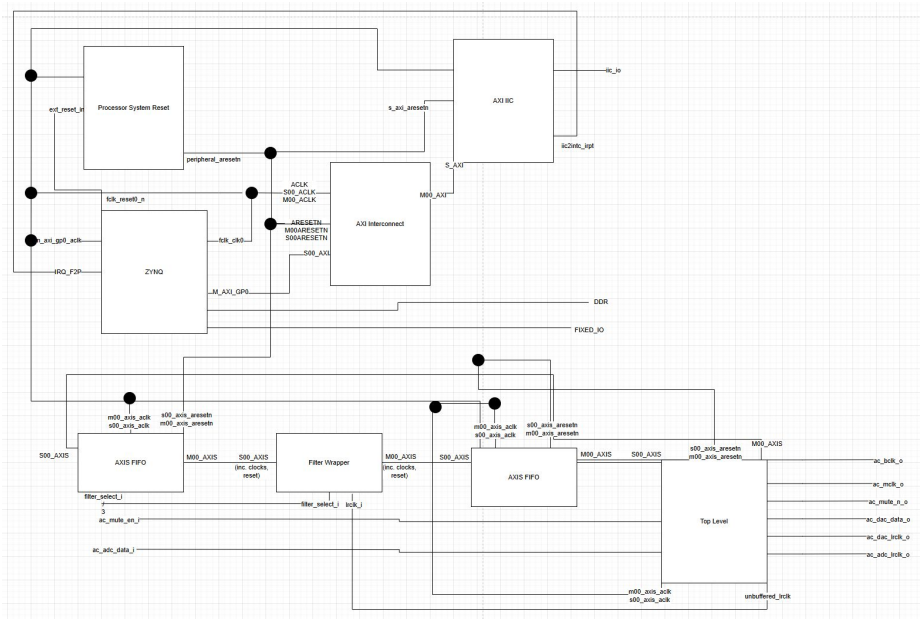
Note that this paper design, for the purposes of readability, combines all AXI signals (*e.g.,* signals like `s00_axis_tready`, `m00_axis_tvalid`, and so forth) into a single input for the purposes of simplicity because the primary focus is on how the input data is received by the AXI receiver, transformed into left or right channel data (and converted from 32 bits to 24 bits), then ultimately passed to one of four filters (eight total, recognizing the left/right channel distinction) for filtering and output. Those filters, by virtue of implementing Vivado IP cores, are also AXI-compliant, although we nonetheless implement an AXI receiver/transmitter to handle the particularities of the audio signal (*e.g.,* converting from 32 bits to 24, identifying different left/right audio channels). The filtered data is then passed as AXI input to the AXI transmitter, which converts the signal from 24 bits back to 32 bits and outputs it via AXI.

The corresponding RTL schematic generated by Vivado is somewhat different in that it relies heavily on lookup tables and structures the circuit in a much more complex way (after all, there are eight total filters, so the wiring becomes quite complex):

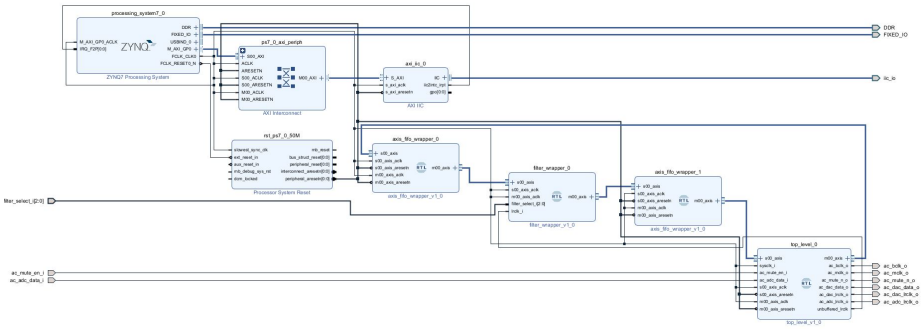


2.8 Total Block Diagram

The total block diagram for the system was designed on paper during a prototyping phase, as illustrated below.



This block design was then developed in the Vivado software. The corresponding RTL schematic generated by Vivado is essentially identical, although it is in some cases more descriptive as to particular AXI signals:



3 Implementation and Programming Details

The circuits detailed above were programmed using VHDL code that was subsequently compiled into a bitstream using the AMD Vivado software platform. This process is detailed below.

As a first step, individual constituent parts of the aforementioned circuit design were programmed using VHDL. For example, individual portions of the "top level" circuit, including the AXI stream receiver, AXI stream transmitter, I2S audio receiver, I2S audio transmitter, and the clock generator were programmed in VHDL. These discrete elements were then connected together using a "top level" VHDL wrapper. Separately, a FIFO VHDL script was programmed and wrapped using an AXI-compliant wrapper. Later, an AXI-complaint filter element was programmed in VHDL by connecting an AXI stream receiver and AXI stream transmitter to four different filters (one for band pass, one for band stop, one for high pass, and one for low pass). The filters themselves were developed using the AMD Vivado FIR Compiler core in Vivado and using pre-established .coe files

Next, the constituent parts of the circuit were connected together using a block design in Vivado. Specifically, the programmed scripts were connected as RTL elements to the Zynq processor, an AXI Interconnect, an AXI IIC, and a Processor System Reset element. This generated the top-level block design included at the beginning of this section.

During various portions of the above two processes, various testbenches were used to validate performance. For example, one testbench was developed and used to confirm that the AXI stream receiver and AXI stream transmitter were compliant (and could, for instance, communicate with one another). Similarly, a different testbench was used to confirm that the I2S audio receiver and I2S audio transmitter received and transmitted audio data in a manner that complied with the SSM2603 audio codec. These testbenches are discussed later in Section 4.

Once thoroughly constructed and vetted, the AMD Vivado tool was used to perform synthesis, generate an implemented design, and ultimately generate hardware comprising a bit stream of the completed circuit. This hardware was then transferred to the AMD Vitis software for further development.

The AMD Vitis software was used to implement pre-established (*i.e.,* not developed by the author) C code to perform various audio processing tasks. This C code includes, among other things, I2C drivers for the IP core (in iic.c and iic.h), interrupt controllers (in intc.c and intc.h, and standard drivers for startup and configuration of the Zynq processor (including, for example, platform_config.h, platform.c, and platform.h).

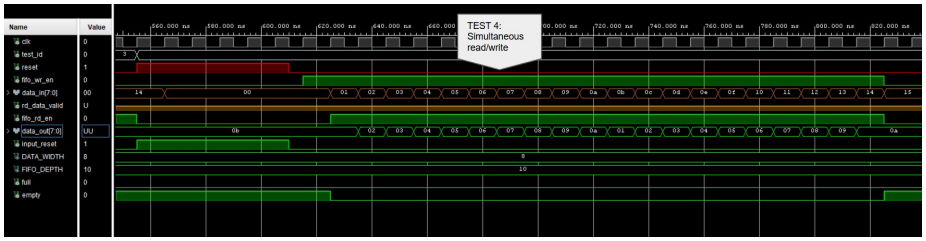
The C code in Vitis was used to compile both a representation of the Vivado-developed hardware as well as the C-based software to generate a package deployed and executed on the Zybo Z7-20 FPGA development board. This deployment was then tested using the Analog Discovery 3 waveform generator and oscilloscope tool. Once further validated, music input/output performance was testing using an input (the headphone jack of a laptop) and an output (Universal Serial Bus-powered speakers). A photograph of the latter setup is provided below.

4 Testing Results

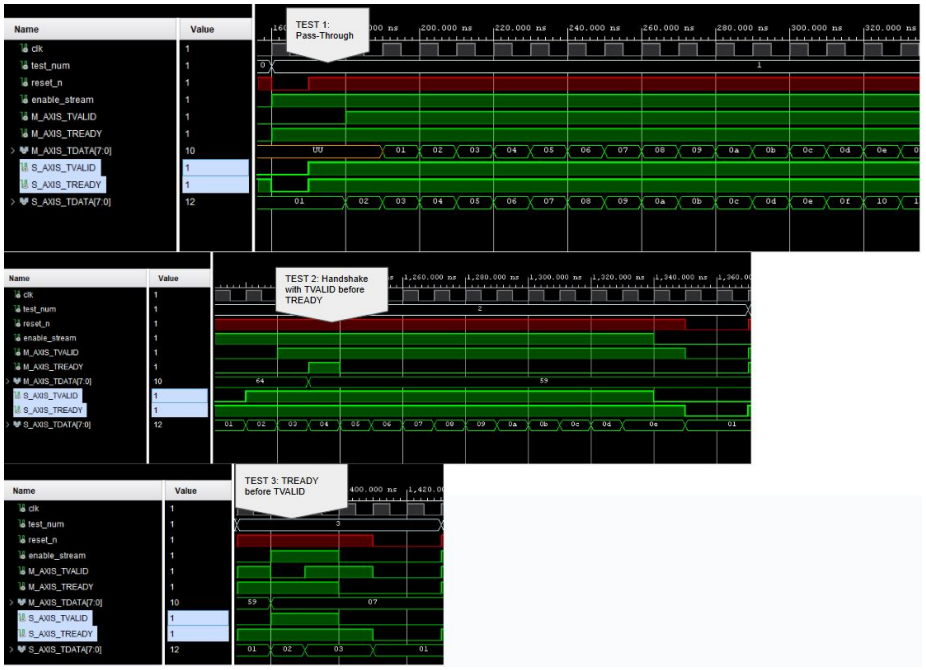
4.1 Testbench Simulation

A large variety of test benches were used to validate performance of various portions of the circuit described above through simulation. Various illustrative results from simulations using those testbenches are provided below; however, this is merely a fraction of the testbenches used during development.

One example of a testbench used during development was to test the performance of the FIFO. The FIFO performance was first tested on its own to validate, for example, the ability of the circuit to perform simultaneous read/write functionality (Test 4 in the testbench). Example results of such testing is included below.



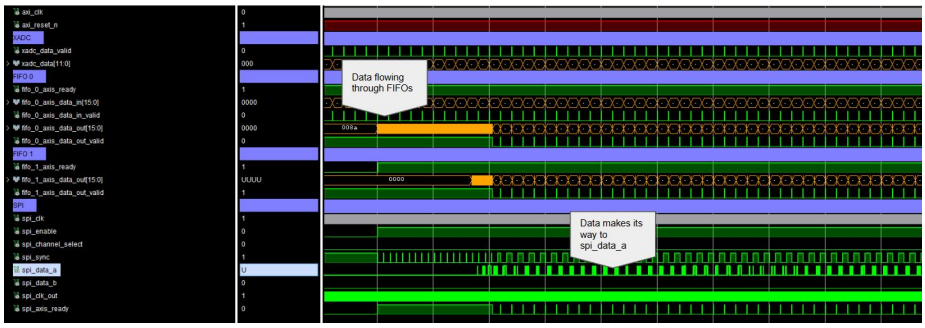
Then, when made AXI-compliant, the AXI-relevant performance of the FIFO was tested to confirm, for example, appropriate behavior with respect to various AXI signals. Examples of results of such tests are below.



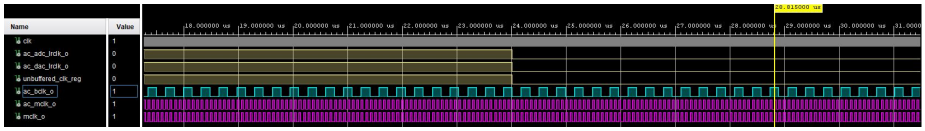
A FIFO-to-FIFO pass-through testbench was then implemented to ensure that output from one FIFO element could be used as input to the other FIFO circuit, with both FIFO communicating using AXI. An example of such testing is illustrated below, showing tests ensuring that complaint AXI-related signaling (*e.g.,* asserting both `TREADY` and `TVALID` simultaneously) performed as expected.

A testbench was also developed to evaluate performance of the AXI stream DAC. This testing was successful, as it showed output of the data received as expected. An example result of this part of the testing is included below.

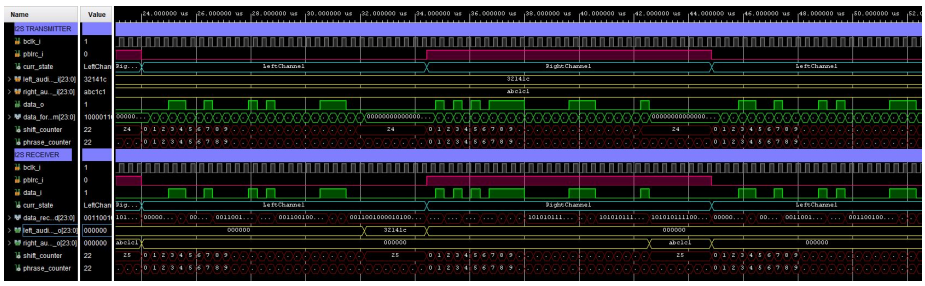
Then, yet another testbench was used to test performance of the XADC, the two FIFOs, and the SPI controller-including the AXI stream wrappers-in a single test. This testbench was also successful, showing data properly flowing through the FIFOs to the SPI controller. Example results of this testing are included below.



Around the same time, the different clock domains were prototyped and tested. An example of such testing is illustrated below, showing successful timing of signals such as `MCLK`, `BCLK`, and `LRCLK`:

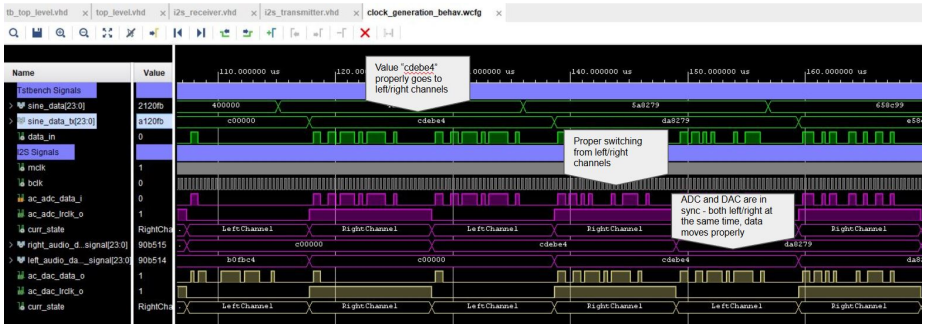


When developing the I2S transmitter and receiver, various testbenches were used. The testbench simulation below shows both the I2S transmitter properly transmitting left/right channel data in accordance with the protocol expected via the audio codec as well as proper receipt and decoding of the signal by the I2s receiver.

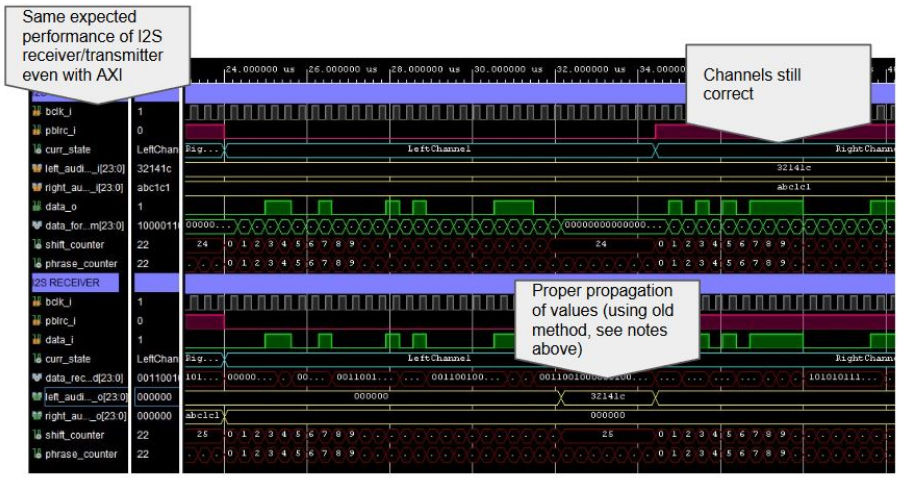


Note that this figure depicts a slightly older version of the testbench and I2S circuit, where the I2S receiver was configured to only hold the received value during the current state (*i.e.,* the value for the left/right output is cleared with every state change, meaning that there are only a few shift counters where the output is anything other than hex value "000000"). This was later changed to preserve the value until rewritten with a new value, as testing indicated better performance using this approach.

The I2S transmitter/receiver were later tested in conjunction with the clock generator to validate performance of the circuit in a more simplified manner:



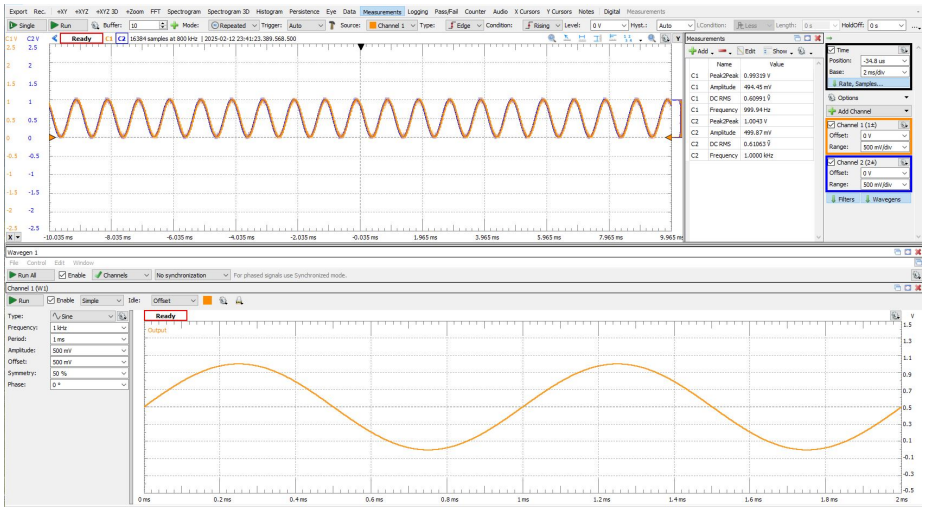
A similar testing process was performed to validate identical performance when wrapped using the AXI protocol:



4.2 AD3 Testing

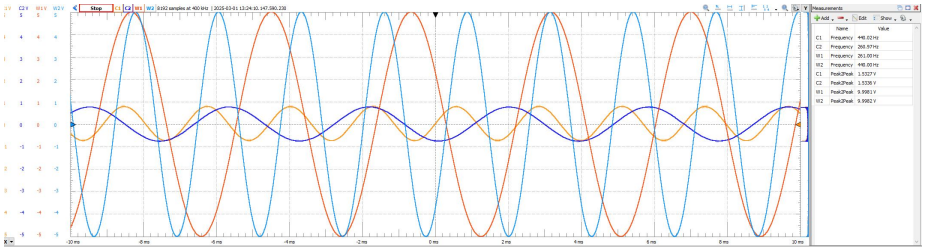
Once testbench simulations indicated satisfactory results, testing of the circuit using the Analog Discovery 3 ("AD3") waveform generator and oscilloscope was conducted in two steps: a first step largely focusing on validating FIFO performance, and a second step validating DSP performance without filtering.

As a preliminary step, the performance of the FIFO circuit alone was tested using the AD3 waveform generator and oscilloscope. That is, the circuit *without* any additional filtering/DSP and *without* use of the SSM2603 audio codec was tested using the AD3. This testing worked extremely well, indicating ideal performance of the AXI Stream interface and FIFO to FIFO pass-through without any loss. Specifically, a 1 kHz sine wave with an amplitude of 500 mV was shown as successfully passing through the circuit without any issues:



Note that the input (orange) and output (blue) waveforms here are nearly identical, confirming proper pass-through performance despite the perfunctory intermediary processing of the signal.

As a second step, the circuit *including* DSP functionality but without filtering functionality was tested using an Analog Discovery 3 wave generator and oscilloscope. The wave generator was configured to provide two input waves (261 Hz and 440 Hz with 50% symmetry and an amplitude of 5 V) as input along with a 2 V positive supply (V+) and a -2 V negative supply (V-). The results were as expected, showing identical frequency outputs from the circuit (albeit shifted in terms of time and in terms of amplitude due to the inherent features of the audio codec as well as the circuit as a whole):



Note again that the two waveforms are identical for the intended purposes (music), although they appear different. The input waveforms (orange/light blue) have a higher amplitude and are slightly shifted relative to the output waveforms (light orange/dark blue), but the frequencies remain the same. Pragmatically speaking, this means that the volumes might be different (an issue that could be remedied through modifying parameters of the audio codec), but the output is exactly as desired.

4.3 Music Testing

Testing of music was conducted using the Zybo Z7 board, USB-powered speakers, and a laptop configured to output music through a headphone jack. A depiction of this test setup is below.

Testing performance was generally excellent, but depended greatly on the music selected. In general, the high-pass and low-pass filters worked; however, it was greatly dependent on the music involved. A high-pass filter, for example, did not have significant effect on music that did not have enough low frequencies (*e.g.,* bass). Conversely, the low-pass filter did not seem to have a significant effect on music that did not have many high frequencies. The band-pass and band-stop filters had similar results, albeit in a more nuanced manner. Unfortunately, these properties were somewhat hard to pick up when recording video using a smartphone, likely due to the noise cancelling algorithms running on the smartphone.

One unexpected result was associated with human speech. Though tested multiple times (*e.g.,* to ensure that the .COE files were correct), it appeared that the low pass filter allowed speech better than the high pass filter. This is somewhat antithetical to conventional wisdom, where high-pass filters are used to help "protect" speech from low vibrations/rumble. This may be the result of a few things - the particular tenor of speech, particularities in the .COE file, or the like. This was the only unusual result seen in testing.

5 Conclusion

Given the stellar testing results indicated above, the circuit described in this paper performed as intended. Audio data was received via the audio codec, received via I2S, passed via AXI to a first FIFO and then to a filter wrapper implementing one of four different filters (or none at all), and then passed via AXI through a second FIFO wrapper and to an I2S transmitter for outputting via speaker. Performance along these lines was successfully validated using simulation testbenches, a digital oscilloscope and waveform generator, and through testing using real input music and speakers. All results were optimal (*e.g.,* the low pass filter did indeed filter low frequencies, the high pass filter did indeed filter high frequencies) with the minor exception of potentially unexpected behavior with respect to speech.

At a high level, this project helped underscore numerous important aspects of the FPGA design process. For example, the iterative construction of this circuit using smaller elements to generate a larger circuit helped emphasize the value of slowly constructing a circuit using individual pieces, rather than (for example) attempting to design the circuit in one singular programming effort. As another example, and following on the first point, by taking an iterative design approach using paper designs and testbenches, the proper functioning of smaller circuit elements (*e.g.,* the aforementioned AXI transmitter/receiver) could be validated before further design efforts were implemented. As yet another example, use of multiple testing methodologies (*e.g.,* using the AD3 oscilloscope, using testbenches and Vivado-based simulation, using real-world testing involving music) all helped identify errors and better illustrate the performance of a circuit before subsequent design tasks were performed.