FPGA-Based DSP System

Final Report

Author
Aaron NGUYEN
Advisor
Dr. Wayne PILKINGTON

June 2018
# Table of Contents

List of Figures 3  
List of Tables 3  
Listings 3  
Abstract 4  

1 Introduction 5  
1.1 Overview 5  
1.2 Motivation 5  
1.3 Description of Potential Customer 5  
1.4 Customer Needs 5  
1.5 System Context 5  
1.6 Alternative/Competitor Solutions 5  

2 Background 7  
2.1 FPGA 7  
2.2 FIR Filter 7  
2.3 IP Cores 7  
2.3.1 Clock Wiz 7  
2.3.2 DDS Compiler 8  
2.4 IP2S Protocol 8  

3 Product Design Engineering Requirements 10  
3.1 Functional Requirements 10  
3.2 Minimum Performance Specifications 10  
3.3 User Interface 10  

4 System Design - Functional Decomposition (Level 1) 11  
4.1 Overall System Functional Block Diagram 12  

5 Technology Choices and Design Approach Alternatives Considered 13  
5.1 Main Processor 13  
5.2 Input and Output 13  

6 Project Design Description 14  
6.1 ADC 14  
6.1.1 Theory of Operation 14  
6.1.2 Specifications 14  
6.2 Filter 14  
6.2.1 Theory of Operation 14
List of Figures

1. P2S Protocol Timing Diagram ........................................... 8
2. Level 0 Block Diagram for FPGA DSP System .......................... 10
3. Level 1 Block Diagram for FPGA DSP System ......................... 12
4. Direct Form 1 FIR Filter Block Diagram .................................. 15
5. Complete Physical Design .................................................. 17
6. ADC Behavioral Simulation .................................................. 18
7. DAC Behavioral Simulation .................................................. 18
8. Filter Expected Response (9 Tap Moving Average) ..................... 19
10. Filter Behavioral Simulation — 6.5 kHz Input .......................... 20
11. System Post-Implementation Simulation .................................. 21
12. Clock Signals (ADC and DAC), ADC Input Grounded ................. 22
13. Output — ADC Connected, No ADC Input ............................. 22
14. Output — Frequency Response, ADC Connected No ADC Input .... 23
15. Output — 500 Hz Sinusoid Input, Filter Disabled ...................... 24
16. Output — 500 Hz Sinusoid Input, Filter Enabled ...................... 24
17. Output — Multiple Sinusoid Input, Filter Disabled ................... 25
18. Output — Multiple Sinusoid Input, Filter Enabled ................... 25
19. Time Estimate Schedule .................................................... 34
20. Actual Schedule ............................................................ 34
21. ADC Module Schematic .................................................... 35
22. Layout and Implementation of ADC PCB Module ..................... 36

List of Tables

1. ADC Functional Requirements ............................................. 11
2. FPGA Functional Requirements .......................................... 11
3. DAC Functional Requirements ............................................. 11
4. System Prototype BOM ..................................................... 33

Listings

1. System Wrapper Code ....................................................... 37
2. Clock Driver ............................................................... 40
3. ADC Driver ............................................................... 41
4. Filter Driver ............................................................... 42
5. DAC Driver ............................................................... 45
Abstract

The purpose of this project is to create a modular FPGA-based filtering system for audio in VHDL. The final implementation has a working input, output, and filtering system but the filtering system still must be fine-tuned.
1 Introduction

1.1 Overview
This project implements a filtering system on the Nexys 4 DDR board. It receives one input (stereo 3.5mm or RCA), filters it with the FPGA, and then outputs the result to a stereo 3.5mm jack.

1.2 Motivation
The purpose of this product is to allow for simple synthesis of an FIR filter in an FPGA rather than a DSP chip. Since this is a relatively niche product in its current state, the customer will likely use it to analyze and compare FPGA-based filters with DSP chip-based filters. The main problem it solves for the customer is the customer does not have to implement the entire filtering system from scratch; the input and output methods are already designed and a base filter driver is already given. The example logic is already within the filter driver and it can easily be changed from its current direct form I structure to another structure by following the example syntax and logic.

1.3 Description of Potential Customer
In its current state, the primary customer for this product would be students in an educational environment. The filter is completely programmable and can be easily converted to another filter form. Input and output logic are already taken care of and do not need to be worried about.

1.4 Customer Needs
The customer/market requirements are as follows:

- Quick filter disabling method for comparison of filtered signal vs unfiltered signal.
- Programmable VHDL-based filtering.
- Relatively simple to use.

1.5 System Context
Since this project is solely a DSP unit, it must sit in-between a line-level output device (e.g. pre-amplifier or computer) and the device to be outputted to. The system ideally outputs a signal with magnitude equal to the signal coming in.

1.6 Alternative/Competitor Solutions
The main competitor for this product are the devices created by MiniDSP. MiniDSP products utilize discrete DSP chips rather than FPGAs though, so their performance is limited. The
MiniDSP 2x4 HD, for example, can have a maximum of ten parametric filters on each input channel and ten parametric filters on each output channel, or they can utilize a maximum 1024 tap FIR filter.

This solution can easily exceed a 1024 tap filter but it can only be programmed via putting in filter coefficients line by line.
2 Background

2.1 FPGA

An FPGA, or a field-programmable gate array, is an integrated circuit with programmable transistor-level logic. It can be programmed with a hardware design language such as VHDL or Verilog. The primary benefit of FPGAs is that they can have multiple processes that run in parallel with sequential "code" running inside the processes.

2.2 FIR Filter

An FIR Filter is a filter whose response ends in a finite amount of time. In general, an FIR filter cannot have feedback or else it would have an infinite impulse response; it would be an IIR filter. The benefits of FIR filters are stability, predictability, and ease of design but one large drawback is that they require a longer filter length than IIR filters to achieve the same performance.

To achieve filter lengths with tens of thousands or millions of taps, either an FPGA or a powerful computer must be used. DSP chips may not suffice for filters with very long lengths since they compute sequentially rather than in parallel. As an example, if we had a symmetric FIR filter with a million taps, a DSP chip would take half a million clock cycles to compute the output. This computation would have to be done before the next sample arrives. In comparison, an FPGA can compute the product of each coefficient’s value multiplied by their respective delayed input samples in one cycle, and accumulate all of the outputs on the next cycle.

2.3 IP Cores

2.3.1 Clock Wiz

The Clock Wiz core is utilized to generate a clock that is difficult to create in code. Specifically, this project needs a 24.576 MHz system clock driving both the ADC and DAC module, and a 6.144 MHz bit clock. This is obtainable by choosing the following settings. Unlisted settings are left as their default value.

- Clocking Options
  - MMCM
  - Frequency Synthesis
  - Minimize Output Jitter

- Output Clocks
  - Clock 1 — 24.576 MHz
  - Clock 2 — 6.144 MHz

The actual clock value will be very slightly off but it should not matter as long as they are synced. The LRCK clock is generated in code since the Clock Wiz core cannot generate clocks below approximately 4 MHz.
2.3.2 DDS Compiler

The DDS Compiler is utilized in this project to simulate an analog waveform in Vivado. These input waveforms are utilized to test the characteristics of the filter driver.

The DDS compiler generates an output waveform according to the input clock frequency \( f_{clk} \), the phase bit width \( B_\theta \), and the phase increment \( \Delta \theta \). The IP Core’s settings were set as follows. Settings unlisted were left as their default value.

- Configuration
  - System Clock \( f_{clk} = 100 \) MHz
  - \( B(\theta) = 30 \)
  - Output Width = 24

- Implementation
  - Phase Increment Programmability — Streaming (allows for increasing phase increment i.e. frequency)

The frequency output (m_axis_data_tdata) is governed by the following equation:

\[
f_{out} = \frac{f_{clk} \times \Delta \theta}{2^{B_\theta}}
\]

Therefore, to get a sample frequency output of 20 kHz, the following phase increment must be utilized:

\[
\Delta \theta = 2^{B_\theta} \times \frac{f_{out}}{f_{clk}}
\]

\[
= 2^{30} \times \frac{20000}{100000000}
\]

\[
\approx 236223
\]

A value of 236223 would therefore need to be set as the input (s_axis_phase_tdata) to the DDS Compiler core. An example of its output can be found later in Section 8 — Integrated System Tests and Results.

2.4 I²S Protocol

A diagram of the I²S Protocol can be seen below.

![Figure 1: I²S Protocol Timing Diagram](image-url)
This protocol functions with two primary clocks and an input/output port. The LRCK clock is the channel select clock and is equivalent to the sampling rate. In this project, a sampling rate of 96 kHz is utilized. The bit clock allows for clocking in or clocking out bits to the device using the I²S protocol. This clock must be a multiple of the LRCK clock. In this project, the PCM1803 requires a bit clock of 6.144 MHz which is 64x LRCK.

Data is clocked in starting on the first falling edge of the bit clock after a channel switch (falling edge of LRCK clock). The protocol clocks in data from MSB to LSB order. After the output width or input width desired is achieved i.e. after the LSB is obtained, all subsequent bits are ignored until the next channel switch.
3 Product Design Engineering Requirements

3.1 Functional Requirements

To achieve the end goals for this product, the product’s DSP functionally must be fully programmable by the user. It must take a minimum of one stereo analog input and output to one stereo analog audio jack. The audio-signal must be allowed to pass-through and be filtered without being reprogrammed.

3.2 Minimum Performance Specifications

- Frequency Response: 20Hz – 20kHz
- Processing Capability: 24-bit/96 kHz Sampling Rate
- Programmable filtering.

3.3 User Interface

The user is capable of turning the filter on and off with a switch (switch 0). An LED indicator (LED 0) indicates whether or not the filter is on or off. When off, the FPGA passes the ADCs output to the DAC’s input. When on, the ADCs output is filtered before being sent to the DAC. The output can be analyzed with a spectrum analyzer, a scope, or with speakers or headphones.

The filter itself is programmable in the filter_driver.vhd file seen in the appendix. It must be coded in VHDL.

Figure 2: Level 0 Block Diagram for FPGA DSP System
4 System Design - Functional Decomposition (Level 1)

Table 1: ADC Functional Requirements

<table>
<thead>
<tr>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Outputs</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
</tr>
</tbody>
</table>

Table 2: FPGA Functional Requirements

<table>
<thead>
<tr>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Outputs</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
</tr>
</tbody>
</table>

Table 3: DAC Functional Requirements

<table>
<thead>
<tr>
<th>DAC Module</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Outputs</strong></td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
</tr>
</tbody>
</table>
4.1 Overall System Functional Block Diagram

Figure 3: Level 1 Block Diagram for FPGA DSP System
5 Technology Choices and Design Approach Alternatives Considered

5.1 Main Processor

The FPGA was chosen since it is capable of utilizing parallel processing to compute difference equations. The alternative solutions are DSP chips (cheap but less potential) and an ASIC solution (significantly more expensive). The Nexys 4 DDR development board was specifically chosen due to its relatively low price and due to the fact that its technical specifications are slightly better than the Basys 3.

DSP chips run an instruction at a time so their performance is limited by the processor’s speed. In an FPGA, a pipeline architecture can achieve much higher throughput since at every cycle, multiple operations are completed at once rather than just one due to its parallelized structure.

Designing a DSP system utilizing a custom-designed ASIC was not chosen due to the slow development process and large cost.

5.2 Input and Output

An external ADC is being utilized for the analog input since the Nexys 4 DDR’s internal XADC is capable of measuring a maximum differential voltage of 1 V (-0.5 to 0.5 V). This is too little since line-level outputs are rated at 2 $V_{peak}$. The device chosen was the PCM1803 since it utilizes the I2S protocol and it has stereo channel input and output, 24-bit data width, and an adjustable sampling rate.

An external PMOD module is being utilized since it implements 16 to 24-bit stereo conversion and has stereo output. The Pmod module is the Pmod I2S: Stereo Audio Output from Digilent. This is capable of utilizing some of the same clock rates that the PCM1803 utilizes. The clock rates chosen for the PCM1803 were specifically chosen such that they were also compatible with the Pmod I2S module.
6 Project Design Description

All the code for these systems can be found in Appendix E.

6.1 ADC

6.1.1 Theory of Operation

The ADC functions utilizing three clock signals: the system clock, the bit clock, and the channel select clock. All three must be precise ratios of each other as defined in the PCM1803 datasheet. It utilizes the I²S protocol described in Section 2.4. It clocks in data according to the protocol and outputs it directly to the FPGA. Each bit is clocked out according to the speed of the bit clock, and the channel is known by the status of the channel select clock. The system clock syncs the systems together.

To trigger on each channel switch, a clock with double the frequency of the channel select clock is utilized. Triggering on each falling edge on this clock is equivalent to triggering on the rising edge (right channel) and falling edge (left channel) of the channel select clock. This circumvents the issue of attempting to trigger on a rising edge and falling edge of a single clock in one process; it is not possible on the Nexys 4 DDR and most other FPGA boards.

6.1.2 Specifications

The ADC utilizes the following parameters:

- 24-bit/96kHz bit-depth/sampling rate.
  - System Clock — 24.576 MHz
  - Bit Clock — 6.144 MHz
  - LRCK Clock — 96 kHz
- 64x OSR

The bit-depth and sampling rates were chosen since 24-bit/96 kHz was easier to optimize for than CD quality (16-bit/44.1kHz). The 96 kHz sampling rate tells us that the maximum reproducible frequency is approximately 48 kHz. It also gives us approximately doubles the number of samples per cycle for 20 kHz (2 to4 samples) compared to 44.1 kHz, so this will give us slightly more true reproduction in the audible frequency band.

6.2 Filter

6.2.1 Theory of Operation

A direct form I filter was implemented in VHDL. It’s implementation can be seen below.
Figure 4: Direct Form 1 FIR Filter Block Diagram

The FIR filter is always assumed to be symmetric in the code.

When the ADC driver in the FPGA indicates that it is done reading in data, the filter begins execution. On the first rising edge of the system clock (24.576 MHz) after the ADC indicates it is ready, the filter driver delays all previous samples by one cycle and clocks in the new 24-bit sample given by the ADC driver. On the following cycle, it computes the output $y[n]$ by pre-adding the proper samples and multiplying them by their respective coefficient. It accumulates all of these values and outputs it to the DAC. The DAC does not need to know whether or not the filter’s output is ready since the process happens significantly faster than the rate at which the channel select changes and data is outputted.

6.3 DAC

6.3.1 Theory of Operation

The DAC performs exactly the same as the ADC. The only difference is that the roles are reversed; the DAC clocks in signals from the FPGA according to the $I^2$S protocol and outputs it to an external device.

6.3.2 Specifications

The DAC utilizes the following parameters:

- 24-bit/96kHz bit-depth/sampling rate.
  - System Clock — 24.576 MHz
  - Bit Clock — 6.144 MHz
  - LRCK Clock — 96 kHz

These are identical to the ADC’s parameters since the two devices must be synced.
6.4 System

6.4.1 Theory of Operation

The system receives its input through a stereo 3.5mm jack. This input is sampled by the ADC system and outputted to the FPGA one sample at a time. A filter enable switch determines whether or not the filter driver is enabled. If disabled, the signal passes through the FPGA unadulterated to the DAC. If enabled, the signal is filtered on every new sample and this filtered output is sent to the DAC.

The DAC has a channel select clock that is delayed half a clock cycle — $180^\circ$ — from the ADC’s channel select clock since the output to the DAC is delayed one cycle from the input. This allows for the respective channel outputs to remain in sync.
7 Physical Implementation & Packaging Concept

7.1 Hardware Configuration/Layout

On the ADC module, the components are laid out for maximum efficiency and readability on a single layer. All bypass capacitors are placed as close to their respective power pins as possible. The voltage regulators are placed at the top of the PCB near the power jack so they can easily receive the 9V input and pipe their 5V and 3.3V outputs directly to the ADC. Two LEDs are used to indicate the 5V and 3.3V voltage regulators are functional and are outputting their proper voltage.

Under the ADC is a star ground point. This is utilized to keep the analog and digital grounds separate. The analog ground is attached to practically all components except for the buffer and for the 3.3V voltage regulator since this regulator’s output goes into the digital input of the ADC.

The header pins are placed exactly 300 mil part. This allows for the soldered headers to be directly placed into the FPGA’s I/O ports without modification.

The final implementation of the system can be seen below.

Figure 5: Complete Physical Design
8 Project (System-Level) Verification Test Plan

8.1 ADC Driver

Figure 6: ADC Behavioral Simulation

Figure 6 demonstrates how the ADC simulation performs. In summary, the behavior matches the I²S protocol exactly. In blue is the bit clock. When comparing the bit clock’s falling edge to the data in, it can be seen that the data ”inputted” from DIN is clocked in bit by bit as expected.

8.2 DAC Driver

Figure 7: DAC Behavioral Simulation

Figure 7 demonstrates how the VHDL simulation of the DAC performs. It again follows the I²S protocol exactly. At the channel switch, the data that is fed in (SDIN) is clocked out to
SDOUT. Comparing the two waveforms demonstrates that what goes in also exits the DAC driver exactly as expected.

### 8.3 Filter Driver

![Figure 8: Filter Expected Response (9 Tap Moving Average)](image)

Figure 8 demonstrates the frequency response of a 9 tap moving average filter. This filter was utilized in all subsequent testing procedures. The difference equation implemented before scaling can be seen below.

\[
\]

There are a few notable features. The filter is effectively a low pass filter with a cutoff frequency of approximately 5 kHz. Furthermore, there is one notch in the audible band that
resides at approximately 10.666 kHz. There are also three out-of-band notches located at multiples of 10.666 kHz.

The filter simulation can be seen in the next figure.

Figure 9: Filter Behavioral Simulation — 21.4 kHz Input

The DDS Compiler’s output is utilized in this simulation. The frequency of the waveform is approximately 21.4 kHz; note that the notch expected for the filter implementation is at 21.31 kHz. In this figure, it can be seen that the filter does perform attenuation as expected, but it does not completely notch out the signal. Furthermore, the signal is clipped significantly.

None-the-less, this does show that the filter is performing somewhat as expected and it shows how the input samples are propagating through the delay flip-flops.

Figure 10: Filter Behavioral Simulation — 6.5 kHz Input
In the simulation in Figure 10, the DDS Compiler’s output is set to 6.5 kHz. This corresponds to the approximate -6 dB frequency of the moving-average filter. This is the other problem with the filter; there is an unsolved issue that appears to be caused by underflow and overflow. Multiple fixes were attempted but none so far gave the response wanted.

### 8.4 System

![Figure 11: System Post-Implementation Simulation](image)

The post-implementation simulation seen in Figure 11 shows the expected behavior of the FPGA after combining the ADC, filter, and DAC drivers together. It is difficult to see or confirm but it has been confirmed that the output signal to the DAC (DAC_SDOUT) matches that of what is expected. The purple line in the figure indicates when the filter is turned on and this shows that the output pattern of the signal changes significantly despite the fact that the input is kept constant. Q is the filter’s output and D is its input.

Figure 12 shows a scope capture of the output of the programmed FPGA. The ADC’s input is grounded in this case. This figure confirms that the clocks are synced and that the signal from the ADC propagates to the DAC properly with nothing inputted.

Figure 13 demonstrates what occurs when the ADC is connected and powered on but no input is connected to the ADC. This was the largest problem of the project; the PCB is faulty but it could not be determined why. There is a significant amount of noise propagating through the system and this eliminates any possibility of a good SNR ratio.

Figure 14 shows the Fourier transform of the noise output of the ADC. This is after piping it through the FPGA unfiltered and outputted through the DAC. As can be seen, the primary noise source is located at approximately 3 kHz. The subsequent noisy frequencies appear to be specific odd harmonics of the primary noise source’s frequency.
Figure 12: Clock Signals (ADC and DAC), ADC Input Grounded

Figure 13: Output — ADC Connected, No ADC Input
Figures 15, 16, 17, and 18 were taken with a 50 kHz span and a 25 kHz center frequency for the FFT window. The system is theoretically capable of reproducing up to 48 kHz, so these settings are a fitting match to test its limits.

Figure 15 shows the Fourier transform of the unfiltered output signal of the system with a 500 Hz sine wave input. This demonstrates another wrap-around issue unrelated to the one described with Figure 10 since the filter is turned off. Yellow is the 500 Hz input signal and green is the 500 Hz output signal. The signal does propagate through as expected, but the upper and lower end of the waveforms become inverted. This is likely due to the inherent noise in the signal caused by the ADC’s circuit or due to the way data is being read in from the ADC. Regardless, our biggest spike in the Fourier transform is still 500 Hz, but its harmonics are heavily distorted.

Figure 16 shows the Fourier transform of the filtered output signal with the same 500 Hz sine wave input. The filter works almost exactly as expected. The magnitude of the signal is reduced by 6 dB and the resulting output follows a low-pass frequency reponse. The notable quality that does not seem to work is the 10.66 kHz notch; it is missing. The notch at approximately 21 kHz does exist, but the other notches appear to not match multiples of 10.66 kHz.

Figure 17 shows the Fourier transform of the unfiltered output signal of a system with a sum of sinusoid inputs generated with Audacity on a Windows computer. The signal’s frequencies are: 500 Hz, 2 kHz, 6 kHz, 10.66 kHz, 16 kHz, and 19.5 kHz, all with equal amplitudes. Frerquencies above 19.5 kHz were not chosen due to the limitations of the device being utilized.
Figure 15: Output — 500 Hz Sinusoid Input, Filter Disabled

Figure 16: Output — 500 Hz Sinusoid Input, Filter Enabled
Figure 17: Output — Multiple Sinusoid Input, Filter Disabled

Figure 18: Output — Multiple Sinusoid Input, Filter Enabled
As can be seen in 17, the input signals propagate through the system cleanly. A significant amount of noise generated by the ADC can still be seen and heard when connected, but this none-the-less demonstrates that the signals are capable of passing through the system relatively unperturbed.

Figure 18 shows the Fourier transform of the filtered output signal of the system with the same sum of sinusoids input. The filtering properties of the system are much more clear in this case. The waveform demonstrates that the amplitude of the output signal is overall reduced significantly. The magnitude still drops 6 dB for an unknown reason, so a flaw of the filtering system is that when enabled, the filter appears to always attenuate the overall magnitude of the input signal.

The 10.66 kHz notch is still absent but the 21 kHz notch is much more clear. The missing 31.5 kHz notch is also made more apparent, but there does appear to be a notch at around 40 kHz. The filter is therefore working somewhat properly since it does overall follow the frequency response curve expected aside from missing a few notches.

8.5 Summary

The overall system does work and filter mostly as expected but the results are not easily quantifiable due to their behavior. There is a significant amount of noise but the frequency response exceeds the 20Hz–20kHz requirements due to the sampling rates chosen. Furthermore, stereo audio input and output is fully functional, and the filter is easily disabled which allows for quick analysis or comparisons without reprogramming.
9 Conclusions

The specifications indicated in the Detailed Design Review were somewhat met. A DSP system was designed and it is modular and programmable but not with C nor Matlab nor with a .coe file. Furthermore, there is a significant amount of noise and some overflow/underflow errors that eliminate qualifying for any good THD or SNR metric. Despite these flaws, when measured, it still displays proper filtering characteristics of the filter designed and programmed.

Many improvements can be made to the system. The system is still a prototype since there was insufficient time to work on the project. The most significant design improvement would be to change the ADC module to one that can utilize the 3.3V power port from the FPGA. This will significantly reduce board complexity since the entire power layout can be removed and since this would allow for a complete redesign which may fix the noise issues encountered with the PCM1803.

Another significant improvement would be to improve the filter driver. In its current state, its frequency response looks almost exactly as expected, but the waveforms are distorted due to underflow and overflow issues. Furthermore, there appears to be a passive 6 dB attenuation on the output whenever the filter is enabled.
Bibliography


A Analysis of Senior Project Design

A.1 Summary of Functional Requirements

The FPGA-based DSP system is capable of filtering an analog input signal and sending it as an analog output. It takes in a stereo 3.5mm or RCA analog input and outputs to a stereo 3.5mm jack. The clock generation for the FPGA is done internally so no external clock modules are necessary. Theoretically, the filter length can easily exceed a thousand taps but this is untested since half the number of taps would need to be added as lines of code, assuming the filter wanted is symmetric.

A.2 Primary Constraints

There were two primary constraints when designing this project. The first primary constraint was having to code in VHDL. The two intro classes to VHDL (CPE133/233) only touch the basics of hardware design. Specifically, the first problem with coding in VHDL is that we learned how to solve problems in behavioral simulation but not how our code can affect behavior post-synthesis and post-implementation. The latter two simulations replicate how the hardware code will actually behave on the board.

The issues faced in both post-synthesis and post-implementation simulation are timing constraints. Since we’re physically configuring the transistor level logic of the FPGA and we’re actually routing the signals around in code, timing constraints must be considered. If a signal is moving too quickly or too slowly, it can render another signal useless since its data would be invalid or it would just not output what is expected. Since a behavior simulation that does not run post-synthesis or post-implementation does not take timing constraints into account, it only shows how the code logic functions. The behavioral simulation therefore may not match the post-synthesis or post-implementation simulation, and this will cause the programmed FPGA to also not behave as expected.

The other constraint for VHDL primarily deals with how VHDL treats math. The filter driver is close to being fully completed, but it isn’t since the only thing that needs to be fixed is overflow and underflow. In any other language, this could easily be fixed by casting or checking / comparing to limits. In VHDL, casting is fairly convoluted since we also have to keep track of the number of bits we want or expect.

The second primary constraint for this project was the fact that it was not easy to prototype my ADC design before I had to spend money to fabricate it. In fact, I didn’t. The module requires two different power inputs, multiple control signals, three clocks, too many capacitors, signal inputs, and signal outputs. This is likely even more difficult to prototype on a breadboard than it to prototype by fabricating the PCB and assembling it. A carrier board was also not available due to how old the chip is. Another ADC was not chosen though since it was difficult to find one that had stereo 24-bit resolution with a minimum of 96 kHz sampling rate and that utilized the I²S protocol rather than the SPI protocol.

A.3 Economic

- Original estimated cost of component parts
$355.00

- Actual final cost of component parts
  - $222.23
  - See attached BoM (or Appendix M)

- No additional equipment costs for development were incurred.
- Original estimated development time as of start of project.
  - 100 hours.

- Actual development time at end of project.
  - 160+ hours.
  - Estimated at 4 hours a weekday for the past 6 weeks plus less the 4 weeks before that, and plus time spent during the first quarter.

A.4 If manufactured on a commercial basis

With the system in its current state, it is likely that less than a hundred would be purchased per year. It is a niche product that would fit more in an educational environment than a consumerist environment.

The manufacturing cost if implemented with the Artix-7 rather than the Nexys 4 DDR Development board would be still close to $200 per system. This is primarily due to the FPGA chip being $155 each.

If sold commercially, this product could probably be sold for a max of $300 since it would be bought for experimental purposes rather than be utilized as an actual filtering system.

The profit at these rates would therefore be a maximum of $10000 per year.

The system itself utilizes approximately 0.210 W according to Vivado. This is 50 times less than a standard LED lightbulb. Running this 24/7 equates to less than 2 kWh, or $0.80 USD in a worst-case scenario, per year.

A.5 Environmental

The environmental impact of this device is minimal; it does not use any special materials or require special manufacturing procedures. Furthermore, it also utilizes a minimal amount of power. The biggest concern is that if this system is handsoldered, leaded solder would likely be used since it is easier and cheaper to work with compared to unleaded equivalents.

A.6 Manufacturability

The biggest manufacturing challenge for this device, if implemented without the development board, is soldering the FPGA chip. The FPGA uses a BGA package which cannot be handsoldered. Otherwise, the device is fairly simple to manufacture.
A.7 Sustainability

Since the system currently has no chassis, the biggest challenge associated with maintaining the completed system is to keep it away from environmental harm. It is highly susceptible to both water and ESD without the protection of a chassis.

The project primarily impacts the sustainable use of plastics since it does still use organic material for the PCB. The metals utilized, on the other hand, are sustainable due to their recyclability and abundance. All parts chosen are also RoHS compliant.

Three major upgrades that can be made for this project are:

1. Fix and improve filtering system.
2. Change ADCs to one that utilizes a single supply and that has a carrier board available (enables simpler prototyping).
3. Implement a chassis.

The primary challenge with implementing 1) is the fact that it still has to be done in VHDL. With 2), the entire PCB would have to be redesigned. The benefit though is that if a replacement is found and if it can be driven with a 3.3 V rail, it can be powered straight from the FPGA. With 3), the designer would have to delve into a CAD program like SolidWorks which is an entirely different skillset.

A.8 Ethical

A major ethical concern for any filtering system is the ability to weaponize it. If a device can produce subsonic or supersonic frequencies, a filtering system can easily increase the power of those frequencies or decrease the audible frequencies that leak through. In the case of supersonic frequencies, the audible band can be reduced and the supersonic power increased such that a person can be physically damaged without knowing what’s happening or where it’s coming from. Supersonic frequencies are also highly directional, so they can be utilized to single out targets.

Another concern is that it gives someone the ability to damage another person’s equipment. The filter can be modified such that it outputs a DC level value or such that it clips which can damage devices like speakers or microphones.

A.9 Health and Safety

The largest safety concern when manufacturing this project deals with burns. If handsoldered, one can easily get burned by the hot tip.

Likewise, health concerns also have to do with soldering. Since flux is utilized to ensure good connection between the solder and the pads, ventilation is needed.

A.10 Social and Political

Major social and political considerations associated with the use of this product can be seen in the ethical section. There are no other notable concerns otherwise.
A.11 Development

During the development of this project, I primarily learned how to code more advanced VHDL. By extension, I learned how to create drivers that interface with external devices in VHDL. I also learned how filters can be implemented and improved on VHDL. This also taught me limitations of VHDL for filtering since floating point numbers cannot easily be used like in C with a DSP chip.

Independently learned tools utilized for this project were spectrum analyzers and logic analyzers. The logic analyzers proved invaluable for creating the ADC and DAC drivers since they allowed me to easily see all the clock and data signals (8) at once.
## B Parts List and Costs

### B.1 Prototype

Table 4: System Prototype BOM

<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Manufacturer Part Number</th>
<th>Description</th>
<th>Quantity</th>
<th>Unit Price</th>
<th>Extended Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panasonic Electronic Components</td>
<td>ECH-U1H103GX5</td>
<td>CAP FILM 10000PF 2% 50V DC 1206</td>
<td>2</td>
<td>0.53</td>
<td>$1.06</td>
</tr>
<tr>
<td>EPCOS (TDK)</td>
<td>B32921C3103K289</td>
<td>CAP FILM 10000PF 10% 30VAC RAD</td>
<td>2</td>
<td>0.5</td>
<td>$1.00</td>
</tr>
<tr>
<td>Vishay Vitramon</td>
<td>VJ1206Y104KXAAT</td>
<td>CAP CER 0.1UF 50V X7R 1206</td>
<td>4</td>
<td>0.39</td>
<td>$1.56</td>
</tr>
<tr>
<td>Nichicon</td>
<td>UWX10C100MCL2GB</td>
<td>CAP ALUM 10UF 20% 16V SMD</td>
<td>4</td>
<td>0.28</td>
<td>$1.12</td>
</tr>
<tr>
<td>Samsung Electro-Mechanics</td>
<td>CL31A106KACLNNC</td>
<td>CAP CER 10UF 25V X5R 1296</td>
<td>2</td>
<td>0.29</td>
<td>$0.58</td>
</tr>
<tr>
<td>Nichicon</td>
<td>UUD1H010MCLIGS</td>
<td>CAP ALUM 1UF 20% 50V SMD</td>
<td>2</td>
<td>0.44</td>
<td>$0.88</td>
</tr>
<tr>
<td>Murata Electronics North America</td>
<td>GRM31MR71C225KA35L</td>
<td>CAP CER 2.2UF 16V X7R 1296</td>
<td>2</td>
<td>0.34</td>
<td>$0.68</td>
</tr>
<tr>
<td>Visual Communications Company - VCC</td>
<td>CMD17-21VRD/TR8</td>
<td>LED RED DIFFUSED 0805 SMD</td>
<td>2</td>
<td>0.54</td>
<td>$1.08</td>
</tr>
<tr>
<td>Switchcraft Inc.</td>
<td>35RASMT4BH4TRX</td>
<td>CONN JACK STEREO 3.5MM SMD R/A</td>
<td>1</td>
<td>0.98</td>
<td>$0.98</td>
</tr>
<tr>
<td>CUI Inc.</td>
<td>PJ-059A</td>
<td>CONN PWR JACK 2X5.5MM SOLIDER</td>
<td>1</td>
<td>0.77</td>
<td>$0.77</td>
</tr>
<tr>
<td>CUI Inc.</td>
<td>RCJ-2123</td>
<td>CONN RCA JACK 3.2MM R/A</td>
<td>1</td>
<td>1.67</td>
<td>$1.67</td>
</tr>
<tr>
<td>TE Connectivity AMP Connectors</td>
<td>9-103795-0-25</td>
<td>BRKWAY .100 50POS R/A</td>
<td>1</td>
<td>2.9</td>
<td>$2.90</td>
</tr>
<tr>
<td>Vishay Dale</td>
<td>CRCW0805100RFKEAC</td>
<td>RES SMD 100 OHM 1% 1/8W 0805</td>
<td>2</td>
<td>0.1</td>
<td>$0.20</td>
</tr>
<tr>
<td>Vishay Dale</td>
<td>CRCW0805150RJNEA</td>
<td>RES SMD 150 OHM 5% 1/8W 0805</td>
<td>1</td>
<td>0.1</td>
<td>$0.10</td>
</tr>
<tr>
<td>Vishay Dale</td>
<td>CRCW080520K0FKEAC</td>
<td>RES SMD 20K OHM 1% 1/8W 0805</td>
<td>3</td>
<td>0.1</td>
<td>$0.30</td>
</tr>
<tr>
<td>Vishay Dale</td>
<td>CRCW080564R9FKEA</td>
<td>RES SMD 64.9 OHM 1% 1/8W 0805</td>
<td>1</td>
<td>0.1</td>
<td>$0.10</td>
</tr>
<tr>
<td>Texas Instruments</td>
<td>SN74LVC1G34DBVR</td>
<td>IC BUF NON-INVERT 5.5V SOT23-5</td>
<td>1</td>
<td>0.37</td>
<td>$0.37</td>
</tr>
<tr>
<td>Texas Instruments</td>
<td>LP2992A1M5X-3.3/NOPB</td>
<td>IC REG LINEAR 3.3V 250MA SOT23-5</td>
<td>1</td>
<td>1.14</td>
<td>$1.14</td>
</tr>
<tr>
<td>Texas Instruments</td>
<td>LP2992A1M5X-5.0/NOPB</td>
<td>IC REG LINEAR 5V 250MA SOT23-5</td>
<td>1</td>
<td>1.14</td>
<td>$1.14</td>
</tr>
<tr>
<td>Texas Instruments</td>
<td>PCM1803ADB</td>
<td>IC ADC 24BIT STEREO 20-SSOP RES SMD 10K OHM 1% 1/8W 0805</td>
<td>11</td>
<td>0.055</td>
<td>$0.60</td>
</tr>
<tr>
<td>Vishay Dale</td>
<td>CRCW080510K0FKEA</td>
<td>Artix-7 FPGA</td>
<td>1</td>
<td>179</td>
<td>$179.00</td>
</tr>
<tr>
<td>Bay Area Electronics</td>
<td>PCB</td>
<td>PCB for ADC</td>
<td>1</td>
<td>30</td>
<td>$30</td>
</tr>
</tbody>
</table>
## C Project Schedule

<table>
<thead>
<tr>
<th>Design Phase</th>
<th>Research</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preliminary Project Specification</td>
<td>Choose FPGA for implementation</td>
</tr>
<tr>
<td>Choose &amp; simulate/test amplifier topology</td>
<td>Decide on gain and switch mechanisms and implement</td>
</tr>
<tr>
<td>Concept Design and Subsystem Specification Review (Preliminary Design Review)</td>
<td>Learn about FPGA capabilities (internal DAC conversion, possible DSP methods, etc.)</td>
</tr>
<tr>
<td>Implement FPGA DAC algorithm if applicable; if not get DAC</td>
<td>Detailed Design and Subsystem Specification Review</td>
</tr>
</tbody>
</table>

**FINALS & SPRING BREAK**

### Implementation Phase

<table>
<thead>
<tr>
<th>Order parts, design PCB</th>
<th>Send PCB for fabrication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Chassis &amp; Print</td>
<td>Receive PCB &amp; assemble board</td>
</tr>
<tr>
<td>Create filtering method for FPGA</td>
<td>First System Demonstration</td>
</tr>
<tr>
<td>Project Verification Testing</td>
<td>First Draft Project Report Complete</td>
</tr>
<tr>
<td>Demonstration at Senior Project Expo</td>
<td>Project Demonstration &amp; Oral Presentation to Advisor</td>
</tr>
</tbody>
</table>

**Project Report Final Version**

---

**Figure 19: Time Estimate Schedule**

---

**Figure 20: Actual Schedule**
D  PCB Design

D.1  Schematic

Figure 21: ADC Module Schematic
D.2 Layout

(a) Top Layer

(b) Bottom Layer

(c) Combined Top and Bottom Layers

(d) 3D Model (Top)

(e) 3D Model (Bottom)

(f) Physical Implementation

Figure 22: Layout and Implementation of ADC PCB Module


E VHDL Source Code

Listing 1: System Wrapper Code

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity wrapper_filter_system is
  Port (  
    -- Main Inputs
    CLK_FPGA : in STD_LOGIC;
    CPU_RESETN : in STD_LOGIC;
    
    -- Debugging Output
    DB_CLK_MASTER_ADC : out STD_LOGIC;  -- JB1, System Clock for ADC (24.576 MHz)
    DB_CLK_BCK_ADC : out STD_LOGIC;  -- JB2, Channel Select for ADC (96 kHz)
    DB_ADC_DOUT : out STD_LOGIC;  -- JB3, Bit Clock for ADC (6.144 MHz)
    DB_CLK_MASTER_DAC : out STD_LOGIC;  -- JB7, System Clock for DAC (24.576 MHz)
    DB_CLK_LRCK_DAC : out STD_LOGIC;  -- JB8, Channel Select for DAC (96 kHz)
    DB_CLK_BCK_DAC : out STD_LOGIC;  -- JB9, Bit Clock for DAC (6.144 MHz)
    DB_ADC_DOUT : out STD_LOGIC;  -- JB10, Data Output to DAC
    
    -- ADC — TI PCM1803
    ADC_DIN : in STD_LOGIC;  -- JC7, labeled as DOUT in datasheet
    ADC_BCK : out STD_LOGIC;  -- JC8
    ADC_LRCK : out STD_LOGIC;  -- JC9
    ADC_SCK : out STD_LOGIC;  -- JC10
    ADC_FMT : out STD_LOGIC_VECTOR(1 downto 0);  -- JD2 <FMT[1]>, JD1 <FMT[0]>
    ADC_MD : out STD_LOGIC_VECTOR(1 downto 0);  -- JD4 <MD[1]>, JD3 <MD[0]>
    ADC_AL_PDWN : out STD_LOGIC;  -- JD8
    ADC_BPAS : out STD_LOGIC;  -- JD9
    ADC_OSR : out STD_LOGIC;  -- JD10
    
    -- DAC — PMOD I2S from Digilent
    DAC_SCK : out STD_LOGIC;  -- JA7, labeled as mclk in datasheet
    DAC_LRCK : out STD_LOGIC;  -- JA8
    DAC_BCK : out STD_LOGIC;  -- JA9, labeled as SCK in data sheet
    DAC_SDOUT : out STD_LOGIC;  -- JA10, labeled as SDIN in data sheet
    FILTER_ENABLE : in STD_LOGIC;  -- SW[0], LED[0]
    FILTER_ENABLE_LED : out STD_LOGIC
  );
end wrapper_filter_system;

architecture Behavioral of wrapper_filter_system is
  constant C_DATA_WIDTH : integer := 24;
  constant C_NUM_COEFFS : integer := 9;
  constant C_SCALAR : integer := 512;
  component clk_wiz_0
    port (  
      -- Clock in ports
      CLK_IN : in STD_LOGIC;
      -- Clock out ports
      MCLK : out STD_LOGIC;
      CLK_BCK : out STD_LOGIC;
      -- Status and control signals
      reset : in STD_LOGIC;
      locked : out STD_LOGIC
    );
end component;

component clocks
  generic (  
    G_LRCK_COUNT : integer := 64/2);  -- 6.144 MHz / 64 = 96000
  Port (  
    RESET : in STD_LOGIC := '0';
  );
end component;
```
CLK_IN : in STD_LOGIC := '0';  --- 24.576 MHz
CLK_BCK_IN : in STD_LOGIC := '0';  --- 6.144 MHz
UNLOCKED : in STD_LOGIC := '0';  --- active high
OUT_CLK_LRCK_DOUBLE : out STD_LOGIC := '0';  --- 192 kHz
OUT_CLK_LRCK_ADC : out STD_LOGIC := '0';  --- 96 kHz
OUT_CLK_LRCK_DAC : out STD_LOGIC := '0');  --- 96 kHz
end component;

component adc_driver
  Port (  
    UNLOCKED: in STD_LOGIC;
    RESET : in STD_LOGIC;
    SCK_IN : in STD_LOGIC;  --- 24.576 MHz
    LRCK_IN : in STD_LOGIC;  --- 96kHz
    BCK_IN : in STD_LOGIC;  --- 6.144MHz
    MD : out STD_LOGIC_VECTOR(1 downto 0);  --- 0,0
    FMT : out STD_LOGIC_VECTOR(1 downto 0);  --- 0,1
    OSR : out STD_LOGIC;  --- low — 64x osr delta sigma
    BPAS : out STD_LOGIC;  --- low — bypass so no dc in dout
    AL_PDWN : out STD_LOGIC;  --- high
    GOOD : out STD_LOGIC;  --- to processor
    DOUT : out STD_LOGIC_VECTOR(23 downto 0)
  );
end component;

component filter_driver
  generic (  
    G_DATA_WIDTH : integer := 24;
    G_NUM_COEFFS : integer := 9;
    G_SCALAR : integer := 512
  );
  port (  
    MCLK : in STD_LOGIC;  --- MCLK (24.576MHz). Must be synced
    ENABLE : in STD_LOGIC;
    ENABLE_LED : out STD_LOGIC;
    GOOD : in STD_LOGIC;
    DIN : in STD_LOGIC_VECTOR (G_DATA_WIDTH-1 downto 0);
    DOUT : out STD_LOGIC_VECTOR (G_DATA_WIDTH-1 downto 0)
  );
end component;

component dac_driver
  Port (  
    RESET : in STD_LOGIC;
    UNLOCKED : in STD_LOGIC;
    SCK_IN : in STD_LOGIC;
    LRCK_IN : in STD_LOGIC;
    BCK_IN : in STD_LOGIC;
    SDIN : in STD_LOGIC_VECTOR(23 downto 0);
    SDOUT : out STD_LOGIC
  );
end component;

signal s_reset : STD_LOGIC := '0';
signal s_mclk : STD_LOGIC := '0';  --- 24.576 MHz
signal s_clk_bck : STD_LOGIC := '0';  --- 6.144 MHz
signal s_unlocked : STD_LOGIC := '0';
signal s_clk_lrck_double : STD_LOGIC := '0';
signal s_clk_lrck_adc : STD_LOGIC := '0';
signal s_clk_lrck_dac : STD_LOGIC := '0';
signal s_good : STD_LOGIC := '0';
signal s_adc_dout : STD_LOGIC_VECTOR(C_DATA_WIDTH-1 downto 0) := (others => '0');
signal s_filter_dout : STD_LOGIC_VECTOR(C_DATA_WIDTH-1 downto 0) := (others => '0');
signal s_dac_dout : STD_LOGIC := '0';
begin
MAIN_CLOCK: clk_wiz_0
Port map (
  CLK_IN => CLK_FPGA,
  MCLK => s_mclk,
  CLK_BCK => s_clk_bck,
  reset => s_reset,
  locked => s_unlocked
);

CLOCK_SYSTEM: clocks
  generic map (G_LRCK_COUNT => 64/2)
  port map (RESET => s_reset,
             CLK_IN => s_mclk,
             CLK_BCK_IN => s_clk_bck,
             UNLOCKED => s_unlocked,
             OUT_CLK_LRCK_DOUBLE => s_clk_lrck_double,
             OUT_CLK_LRCK_ADC => s_clk_lrck_adc,
             OUT_CLK_LRCK_DAC => s_clk_lrck_dac
);

ADC_SYSTEM: adc_driver
  port map (UNLOCKED => s_unlocked,
            RESET => s_reset,
            SCK_IN => s_mclk,
            LRCK_IN => s_clk_lrck_double,
            BCK_IN => s_clk_bck,
            DIN => ADC_DIN,
            MD => ADC_MD,
            FMT => ADC_FMT,
            OSR => ADC_OSR,
            BPAS => ADC_BPAS,
            AL_PDWN => ADC_AL_PDWN,
            GOOD => s_good,
            DOUT => s_adc_dout
);

FILTER_SYSTEM: filter_driver
  generic map (G_DATA_WIDTH => C_DATA_WIDTH,
               G_NUM_COEFFS => C_NUM_COEFFS,
               G_SCALAR => C_SCALAR)
  port map (MCLK => s_mclk, -- MCLK (24.576MHz). Must be synced
            ENABLE => FILTER_ENABLE,
            ENABLE_LED => FILTER_ENABLE_LED,
            GOOD => s_good,
            DIN => s_adc_dout,
            DOUT => s_filter_dout
);

DAC_SYSTEM: dac_driver
  port map (RESET => s_reset,
            UNLOCKED => s_unlocked,
            SCK_IN => s_mclk,
            LRCK_IN => s_clk_lrck_double,
            BCK_IN => s_clk_bck,
            SDIN => s_filter_dout, -- VECTOR OUTPUT TO DAC
            SDOUT => DAC_SDOUT — ACTUAL OUTPUT TO DAC
);

s_reset <= not CPU_RESETN; — RESET is inverted on board
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity clocks is
  generic ( G_LRCK_COUNT : integer := 64/2);  -- 6.144 MHz / 64 = 96000
  Port ( RESET : in STD_LOGIC := '0';
         CLK_IN : in STD_LOGIC := '0';  -- 24.576 MHz
         CLK_BCK_IN : in STD_LOGIC := '0';
         UNLOCKED : in STD_LOGIC := '0';  -- active high
         OUT_CLK_LRCK_DOUBLE : out STD_LOGIC := '0';  -- 192 kHz
         OUT_CLK_LRCK_ADC : out STD_LOGIC := '0';  -- 96 kHz
         OUT_CLK_LRCK_DAC : out STD_LOGIC := '0');  -- 96 kHz
end clocks;

architecture Behavioral of clocks is
  signal s_clk_lrck_adc : STD_LOGIC := '0';
  signal s_clk_lrck_dac : STD_LOGIC := '0';
  signal s_clk_bck_adc : STD_LOGIC := '0';
  signal s_clk_bck_dac : STD_LOGIC := '0';
  signal s_clk_lrck_double : STD_LOGIC := '0';
begin
  ADC_LRCK: process(RESET, CLK_BCK_IN)
  begin
    variable v_adc_lrck_count : integer range 0 to G_LRCK_COUNT := 0;
    if (RESET = '1') then
      s_clk_lrck_adc <= '0';
      v_adc_lrck_count := 0;
    elsif (falling_edge(CLK_BCK_IN)) then
      if (v_adc_lrck_count = G_LRCK_COUNT) then
        s_clk_lrck_adc <= not s_clk_lrck_adc;
        v_adc_lrck_count := 0;
      end if;
      v_adc_lrck_count := v_adc_lrck_count + 1;
    end if;
  end process ADC_LRCK;

  DAC_LRCK: process(RESET, CLK_BCK_IN)
  begin
    variable v_dac_lrck_count : integer range 0 to G_LRCK_COUNT := 0;
    if (RESET = '1') then
      s_clk_lrck_dac <= '0';
      v_dac_lrck_count := 0;
    elsif (falling_edge(CLK_BCK_IN)) then
      if (v_dac_lrck_count = G_LRCK_COUNT) then

Listing 2: Clock Driver
The clock is used to trigger on each falling edge of a channel switch.

```vhdl
-- 6.144 MHz / 32 = 192000
variable v_double_lrck_count : integer range 0 to G_LRCK_COUNT/2 := 0;
begin
  if (RESET = '1') then
    s_clk_lrck_double <= '0';
    v_double_lrck_count := 0;
  elsif (falling_edge(CLK_BCK_IN)) then
    if (v_double_lrck_count = G_LRCK_COUNT/2) then
      s_clk_lrck_double <= not s_clk_lrck_double;
      v_double_lrck_count := 0;
    end if;
    v_double_lrck_count := v_double_lrck_count + 1;
  end if;
end process DOUBLE_LRCK;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity adc_driver is
  Port (  
    UNLOCKED: in STD_LOGIC; -- from clockwiz, unused
    RESET : in STD_LOGIC;
    SCK_IN : in STD_LOGIC; -- 24.576 MHz
    LRCK_IN : in STD_LOGIC; -- 192kHz
    BCK_IN : in STD_LOGIC; -- 6.144MHz
    DIN : in STD_LOGIC;
    MD : out STD_LOGIC_VECTOR(1 downto 0); -- 0,0
    FMT : out STD_LOGIC_VECTOR(1 downto 0); -- 0,1
    OSR : out STD_LOGIC; -- low -- 64x osr delta sigma
    BPAS : out STD_LOGIC; -- low -- bypass so no dc in dout
    AL_PDWN : out STD_LOGIC; -- high
    GOOD : out STD_LOGIC; -- to processor
    DOUT : out STD_LOGIC_VECTOR(23 downto 0)
  );
end adc_driver;

architecture Behavioral of adc_driver is
  signal s_good : STD_LOGIC := '0';
  signal s_dout : STD_LOGIC_VECTOR(23 downto 0) := (others => '0');
  signal s_ready_adc : boolean := FALSE;
  signal s_used_lrck : boolean := FALSE;
  signal s_dout_index : integer range -1 to 23 := -1;
  signal s_din : STD_LOGIC;
begin
  -- 192 kHz rate, fs is at 96kHz => triggers on every channel switch’s
  -- falling edge
  LRCK_TRIGGER: process(RESET, LRCK_IN, s_used_lrck)
  begin
    if (RESET = '1') then
      s_ready_adc <= FALSE;
    else
      -- This clock is used to trigger on each falling edge of a channel switch.
      v_dac_lrck_count := v_dac_lrck_count + 1;
    end if;
  end process DAC_LRCK;
```

Listing 3: ADC Driver
if (falling_edge(LRCK_IN)) then
  s_ready_adc <= TRUE;
end if;
if (s_used_lrck) then
  s_ready_adc <= FALSE;
end if;
end process;

-- output of ADC (input to FPGA)
CLOCK_OUTPUT: process(RESET, BCK_IN, s_ready_adc)
variable v_dout_index : integer range -1 to 23 := -1;
begin
  if (RESET = '1') then
    s_good <= '0';
    v_dout_index := -1;
    s_dout <= (others => '0');
    elsif (falling_edge(BCK_IN)) then
      if (s_ready_adc) then
        s_good <= '0';
        v_dout_index := 23;
        s_dout(22 downto 0) <= (others => '0');
        s_used_lrck <= TRUE;
      end if;
      if (v_dout_index > -1) then
        s_dout(v_dout_index) <= DIN;
        v_dout_index := v_dout_index - 1;
      else
        s_good <= '1';
        s_used_lrck <= FALSE;
      end if;
    end if;
  end process;

MD <= "00" when (RESET = '0') else "ZZ";
FMT <= "01" when (RESET = '0') else "ZZ";
OSR <= '0' when (RESET = '0') else 'Z';
BPAS <= '0' when (RESET = '0') else 'Z';
AL_PDWN <= '1' when (RESET = '0') else '0';
GOOD <= s_good when (RESET = '0') else '0';
DOUT <= s_dout;
end Behavioral;

Listing 4: Filter Driver

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity filter_driver is
  generic
  (G_DATA_WIDTH : integer := 24;
   G_NUM_COEFFS : integer := 9;
   G_SCALAR : integer := 512);
  Port (MCLK : in STD_LOGIC; — MCLK (24.576MHz). Must be synced
     ENABLE : in STD_LOGIC;
     ENABLE_LED : out STD_LOGIC;
     DIN : in STD_LOGIC_VECTOR (G_DATA_WIDTH-1 downto 0);
     GOOD : out STD_LOGIC;
     DOUT : out STD_LOGIC_VECTOR (G_DATA_WIDTH-1 downto 0));
end filter_driver;

architecture Behavioral of filter_driver is
  function source: https://groups.google.com/forum/#!topic/comp.lang.vhdl/sG3bWafr41c
function f_log2(input:integer) return integer is
  variable v_temp : integer;
  variable v_log : integer;
begin
  v_temp := input;
  v_log := 0;
  while (v_temp /= 0) loop
    v_temp := v_temp / 2;
    v_log := v_log + 1;
  end loop;
  return v_log;
end function f_log2;

type xn_array is array (0 to 2**(f_log2(G_NUM_COEFFS))-1) of signed(G_DATA_WIDTH downto 0);
type n_array is array (0 to 2**(f_log2(G_NUM_COEFFS))-1) of signed(G_DATA_WIDTH downto 0);
type bk_coeffs is array (0 to 2**(f_log2(G_NUM_COEFFS))-1) of signed(17 downto 0);
constant C_LIMIT_LOW : signed(G_DATA_WIDTH-1 downto 0) := to_signed(-( 2**( G_DATA_WIDTH-1 ) ), G_DATA_WIDTH);
constant C_LIMIT_HIGH : signed(G_DATA_WIDTH-1 downto 0) := to_signed(2**( G_DATA_WIDTH-1 )-1, G_DATA_WIDTH);
constant C_LOG_SCALAR : integer := f_log2(G_SCALAR);

-- multiply all coeffs by G_SCALAR and put into rom
constant ROM_COEFFS : bk_coeffs :=
  (9 tap moving average filter
   y[n] = 512 * 0.111(x[n] + x[n-1] + ... + x[n-8])
   0 => to_signed(56, 18),
   1 => to_signed(56, 18),
   2 => to_signed(56, 18),
   3 => to_signed(56, 18),
   4 => to_signed(56, 18),
   5 => to_signed(56, 18),
   6 => to_signed(56, 18),
   7 => to_signed(56, 18),
   8 => to_signed(56, 18),
   others => (others => '0')));

signal xn : xn_array := (others => (others => '0'));
signal yn : signed(48 downto 0) := (others => '0');
signal s_processed : STD_LOGIC := '0';
signal s_wait_next_cycle : boolean := FALSE;
signal s_negative : STD_LOGIC := '0';
attribute use_dsp : string;
attribute use_dsp of yn : signal is "yes";
begin
  -- This one is slower than FILTER2 since it computes each coefficient output at every clock edge
  -- FILTER: process(MCLK, GOOD, DIN, s_processed, s_wait_next_cycle)
  --      variable v_i : integer range 0 to G_NUM_COEFFS-1;
  --      constant v_i_mid : integer := (G_NUM_COEFFS-1)/2;
  -- begin
  --      if (rising_edge(MCLK)) then
  --        if (GOOD = '1') then
  --          if (s_wait_next_cycle = TRUE) then
  --            for i in G_NUM_COEFFS-1 downto 1 loop
  --              xn(i) <= xn(i-1);
  --            end loop;
  --            xn(0) <= signed('0' & DIN);
  --            yn <= (others => '0');
  --            v_i := 0;
  --            s_processed <= '0';
  --            s_wait_next_cycle <= FALSE;
  --            report "Rising Edge GOOD Processed";
  --          elseif (v_i < v_i_mid) then -- goes up to one before middle if odd num coeffs
  --            yn <= yn + resize(ROM_COEFFS(v_i) * ( xn(v_i) + xn(G_NUM_COEFFS-1-v_i) ), 48);
  --            v_i := v_i + 1;
  --          elseif (v_i = v_i_mid) then -- goes up to one before middle if even num coeffs
  --            for i in G_NUM_COEFFS-1 downto 1 loop
  --              yn <= yn + resize(ROM_COEFFS(v_i) * ( xn(v_i) + xn(G_NUM_COEFFS-1-v_i) ), 48);
  --              v_i := v_i + 1;
  --          end loop;
  --          end if;
  --        elseif (GOOD = '0') then
  --          if (s_wait_next_cycle = TRUE) then
  --            for i in G_NUM_COEFFS-1 downto 1 loop
  --              xn(i) <= xn(i-1);
  --            end loop;
  --            yn <= (others => '0');
  --            v_i := 0;
  --            s_processed <= '0';
  --            s_wait_next_cycle <= FALSE;
  --            report "Rising Edge BAD Processed";
  --          elseif (v_i < v_i_mid) then -- goes up to one before middle if odd num coeffs
  --            yn <= yn + resize(ROM_COEFFS(v_i) * ( xn(v_i) + xn(G_NUM_COEFFS-1-v_i) ), 48);
  --            v_i := v_i + 1;
  --          end if;
  --        end if;
  --      end if;
  --    end process;
  --  end if;
  --end process;
  -- This code may be too slow for some applications. It is slower than FILTER2 output
  -- because it computes each coefficient output at every clock edge.
elsif (G_NUM_COEFFS mod 2 = 1 and v_i = v_i_mid) then — odd case
  yn <= yn + resize(ROM_COEFFS(v_i) * xn(v_i),48);
  v_i := v_i + 1;
elsif (v_i > v_i_mid) then — logic delays output one clock cycle
  s_processed <= '1';
end if;
else — GOOD = '0'
  s_wait_next_cycle <= TRUE;
end if;
end if;
end process;

— filters data in 2 cycles
FILTER2: process(MCLK, GOOD, DIN, s_processed, s_wait_next_cycle)
  constant v_i_mid : integer := (G_NUM_COEFFS-1)/2;
  variable v_yn : signed(48 downto 0) := (others => '0');
begin
  if (rising_edge(MCLK)) then
    if (GOOD = '1') then
      if (s_wait_next_cycle = TRUE) then
        for i in G_NUM_COEFFS-1 downto 1 loop
          xn(i) <= xn(i-1);
        end loop;
        xn(0) <= signed('0' & DIN);
        v_yn := (others => '0');
        s_processed <= '0';
        s_wait_next_cycle <= FALSE;
        report "Rising Edge GOOD Processed";
      elsif (s_processed = '0') then
        for i in 0 to v_i_mid-1 loop
          v_yn := v_yn + resize(ROM_COEFFS(i) * (xn(i) + xn(G_NUM_COEFFS-1-i)), 48);
        end loop;
        if (G_NUM_COEFFS mod 2 = 1) then
          v_yn := v_yn + resize(ROM_COEFFS(v_i_mid) * xn(v_i_mid), 48);
        end if;
      else
        yn <= v_yn;
        s_processed <= '1';
      end if;
    else — GOOD = '0'
      s_wait_next_cycle <= TRUE;
    end if;
  end if
end process;

end Behavioral;

— Attempted an overflow fix via a process but didn’t work. Kept for reference.
TRUNCATE: process(s_processed)
  variable yn_shifted : signed(48 downto 0) := (others => '0');
begin
  if (rising_edge(s_processed)) then
    yn_shifted := shift_right(yn, C_LOG_SCALAR);
    report "yn_shifted: " & integer'image(to_integer(yn_shifted));
    if (yn_shifted <= C_LIMIT_LOW) then
      report "C_LIMIT_LOW: " & integer'image(to_integer(C_LIMIT_LOW));
      yn_truncated <= std_logic_vector(C_LIMIT_LOW);
    elsif (yn_shifted >= C_LIMIT_HIGH) then
      report "C_LIMIT_HIGH: " & integer'image(to_integer(C_LIMIT_HIGH));
      yn_truncated <= std_logic_vector(C_LIMIT_HIGH);
    else
      yn_truncated <= yn_shifted(48)&std_logic_vector(yn_shifted(G_DATA_WIDTH-2 downto 0));
    end if;
  end if;
end process;

DOUT <= std_logic_vector(yn(G_DATA_WIDTH-1+C_LOG_SCALAR downto C_LOG_SCALAR)) when (ENABLE = '1' and s_processed = '1') else DIN when (ENABLE = '0');
ENABLE_LED <= ENABLE;
end Behavioral;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity dac_driver is
  Port ( RESET : in STD_LOGIC;
          UNLOCKED : in STD_LOGIC;
          SCK_IN : in STD_LOGIC;
          LRCK_IN : in STD_LOGIC;
          BCK_IN : in STD_LOGIC;
          SDIN : in STD_LOGIC_VECTOR(23 downto 0);
          SDOUT : out STD_LOGIC);
end dac_driver;

architecture Behavioral of dac_driver is
  signal s_sdout : STD_LOGIC := '0';
  signal s_used_lrck : boolean := FALSE;
  signal s_ready : boolean := FALSE;
begin
  LRCK_TRIGGER: process(RESET, LRCK_IN, s_used_lrck)
  begin
    if (RESET = '1') then
      s_ready <= FALSE;
    else
      if (falling_edge(LRCK_IN)) then
        s_ready <= TRUE;
      end if;
      if (s_used_lrck) then
        s_ready <= FALSE;
      end if;
    end if;
  end process;

  CLOCK_OUT_SAMPLES: process(RESET, BCK_IN, s_ready)
  variable v_out_index : integer range -1 to 23 := -1;
  variable v_sdin : STD_LOGIC_VECTOR(23 downto 0);
  begin
    if (RESET = '1') then
      v_sdin := (others => '0');
      s_sdout <= '0';
      v_out_index := -1;
    elsif (falling_edge(BCK_IN)) then
      if (s_ready) then
        v_out_index := 23;
        v_sdin := SDIN;
        s_used_lrck <= TRUE;
      end if;
      if (v_out_index > -1) then
        s_sdout <= v_sdin(v_out_index);
        v_out_index := v_out_index - 1;
      else
        v_sdin := (others => '0');
        s_sdout <= '0';
        s_used_lrck <= FALSE;
      end if;
    end if;
  end process;

  SDOUT <= s_sdout;
end Behavioral;