

Design and Analysis of an FPGA Based Low Tap Band-stop FIR Filter

by

Lucas Rosler

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science in Engineering

in the

Electrical Engineering

Program

YOUNGSTOWN STATE UNIVERSITY

May 2021

Design and Analysis of an FPGA Based Low Tap Band-stop FIR Filter

Lucas Rosler

I hereby release this thesis to the public. I understand that this thesis will be made available from the OhioLINK ETD Center and the Maag Library Circulation Desk for public access. I also authorize the University or other individuals to make copies of this thesis as needed for scholarly research.

Signature: _____

Date: _____

Lucas Rosler, Student

Approvals:

Date: _____

Frank X Li, Thesis Advisor

Date: _____

Faramarz Mossayebi, Committee Member

Date: _____

Edward Burden, Committee Member

Date: _____

Dr. Salvatore A. Sanders, Dean of Graduate Studies

Abstract

The use of Field Programmable Gate Arrays (FPGAs), although a newer technology, has paved its way to become a cornerstone of Digital Signal Processing (DSP). FPGAs have a wide variety of uses, offering the user the ability to create customized hardware at the gate level in the matter of minutes. Comprised of thousands of Logic Elements (LEs) and Adaptive Logic Modules (ALMs), the uses of FPGAs are endless. Through the continued research of simple circuitry, the speed and efficiency of more complicated designs will improve. In this thesis, the digital filter, a crucial aspect of DSP, is designed, analyzed, and implemented. Using MATLAB filter designer, ModelSim, and Quartus Prime a low tap band-stop filter was designed and synthesized using VHDL codes. The digital filter was simulated then analyzed with a focus on filtering performance, hardware usage depending on filter order, and slack reduction through state machine formatted code. The results of the experiment show a working filter along with a full analysis of situational performance results. These include higher order filters using more hardware than lower order filters, state machine formatted code using more ALMs and less registers than non-state machine formatted code, and state machine formatted code reducing slack compared to that of the non-state machine formatted code.

Table of Contents

Abstract	iii
List of Figures	v
List of Tables	vii
Acknowledgements.....	viii
CHAPTER 1 RESEARCH AREAS FOR FPGA BASED FIR FILTERS	
1.1 Project Objectives	1
1.2 Research Question	1
1.3 Thesis Overview	1
CHAPTER 2 DIGITAL FILTER AND FPGA LITERATURE REVIEW	
2.1 Introduction.....	3
2.2 Digital Filters	3
2.2.1 FIR vs IIR Filter.....	3
2.2.2 Sampling Rate.....	4
2.3 Hardware Elements.....	5
2.4 FPGA Programming	5
2.4.1 Fixed-Point vs Floating-Point.....	5
2.4.2 Data Pipelining.....	6
2.4.3 Edge Detection.....	7
2.5 Slack.....	8
CHAPTER 3 METHODOLOGY	
3.1 Hardware and Software tools.....	9
3.1.1 MATLAB.....	9
3.1.2 ModelSim.....	10
3.1.3 Quartus Prime and Cyclone V DE-10 Standard	10
3.2 Digital Filter Design	10
3.2.1 MATLAB Filter Design.....	10
3.2.2 VHDL Filter Design	14
3.3 Filter Simulation	18
3.4 Filter Implementation.....	25
3.4.1 ADC Control Module	26
3.4.2 Physical Implementation.....	27
3.5 State Machine Format	34
3.6 Higher Order Filter	35
3.7 Quartus Prime Analysis	38
3.7.1 Hardware Analysis.....	38
3.7.2 Slack Analysis.....	43
CHAPTER 4 CONCLUSION	
4.1 Conclusion	46
Appendix.....	48
Bibliography	63

List of Figures

Figure 1: FIR Structure.....	4
Figure 2: IIR Structure.....	4
Figure 3: Edge Detection Using Gates.....	7
Figure 4: MATLAB Filter Designer Default Screen.....	9
Figure 5: 13-Coefficient Filter Frequency Specifications.....	11
Figure 6: 13-Coefficient Filter Magnitude Response.....	12
Figure 7: 13-Coefficient Filter Impulse Response.....	13
Figure 8: 13-Coefficients for Filter.....	13
Figure 9: Stored Filter Coefficients.....	15
Figure 10: Data Pipelining.....	15
Figure 11: VHDL Clock Divider.....	16
Figure 12: Gate Edge Detect.....	16
Figure 13: Negative Value Solution.....	17
Figure 14: Test Data Code.....	18
Figure 15: Test Data Visualization.....	19
Figure 16: VHDL Test Data Extraction.....	20
Figure 17: 60Hz Filter Simulation.....	20
Figure 18: 60Hz Combined with 1KHz Filter Simulation.....	21
Figure 19: 5KHz Filter Simulation.....	21
Figure 20: 60Hz Combined with 5KHz Filter Simulation.....	22
Figure 21: 3.7KHz Filter Simulation.....	23
Figure 22: 60Hz Combined with 3.7KHz Filter Simulation.....	24
Figure 23: 6.3KHz Filter Simulation.....	24
Figure 24: 60Hz Combined with 6.3KHz Filter Simulation.....	25
Figure 25: ADC Control Module Simulation.....	27
Figure 26: Pin Assignments.....	28
Figure 27: Experimental Setup.....	28
Figure 28: 1KHz Sinusoidal Test Wave.....	29
Figure 29: 1KHz Test Wave LED Output.....	29
Figure 30: 3.7KHz Sinusoidal Test Wave.....	30
Figure 31: 3.7KHz Test Wave LED Output.....	30
Figure 32: 5KHz Sinusoidal Test Wave.....	31
Figure 33: 5KHz Test Wave LED Output.....	31
Figure 34: 6.3KHz Sinusoidal Test Wave.....	32
Figure 35: 6.3KHz Test Wave LED Output.....	32
Figure 36: 8KHz Sinusoidal Test Wave.....	33
Figure 37: 8KHz Test Wave LED Output.....	33
Figure 38: 25-Coefficient Filter Frequency Specifications.....	35
Figure 39: 25-Coefficient Filter Magnitude Response.....	36
Figure 40: 25-Coefficient Filter Impulse Response.....	36
Figure 41: 13-Coefficient Non-State Machine Hardware Usage.....	38

Figure 42: 25-Coefficient Non-State Machine Hardware Usage.....	39
Figure 43: 13-Coefficient State Machine Hardware Usage.....	40
Figure 44: 25-Coefficient State Machine Hardware Usage.....	41
Figure 45: Visual Hardware Summary.....	42
Figure 46: 50MHz Non-State Machine Slack Analysis.....	43
Figure 47: 50MHz State Machine Slack Analysis.....	43
Figure 48: 100MHz Non-State Machine Slack Analysis.....	44
Figure 49: 100MHz State Machine Slack Analysis.....	44

List of Tables

Table 1: Fixed-Point Notation Calculation.....	14
Table 2: Fixed-Point Notation Calculation for Higher Order Filter.....	37

Acknowledgement

I would like to first and foremost thank my family and friends for their support throughout my academic career. To my parents, Vance and Barb, I owe everything for giving me the opportunity to pursue my passions and making me the man I am today. To my siblings, Colin, Leah, Ben, and Isaac, thank you for all your support. Thanks to Mirella for listening to me complain and critiquing my poor writing.

I would also like to thank everyone at Youngstown State University more specifically, my master's advisor Dr. Li for your knowledge and guidance along the way. Also, thank you to my committee members Dr. M and Professor Burden for taking the time to help me achieve my academic goals.

Chapter 1 Research Areas for FPGA Based FIR Filters

1.1 Project Objectives

The main project objective was to design, code, simulate, analyze, and implement a Field Programmable Gate Array (FPGA) based low-tap band-stop Finite Impulse Filter (FIR) filter using VHDL. First, with MATLAB filter designer, a 13-coefficient band-stop FIR filter was designed. Using this design, an implementable filter was programmed using VHDL code followed by a successful functionality simulation and demonstration with a DE-10 Standard FPGA board through Quartus Prime. Upon completion, this same VHDL design was used to create a replica FIR filter with 25 filter coefficients in order to analyze the difference in hardware usage between the two. Next, the original filter design was altered into state machine format to analyze the slack reduction between the state machine format design and the non-state machine format design. The final step was to simulate the slack in Quartus Prime to demonstrate the slack reduction.

1.2 Research Question

Through the design, simulation, and implementation of a low-tap band-stop FIR filter, the analysis of hardware usage depending on filter order and the usefulness of state machine formatted code with a focus on the change in slack is the primary research goal of this thesis.

1.3 Thesis Overview

The thesis is split up into four chapters. The first chapter covers what the project is over and the main research question within the thesis. Chapter two is a literature review, going over background information that was used throughout the design and implementation of the project. The third chapter covers the actual design, simulation, and

implementation process along with giving any findings from the process. The final chapter combines the information and gives a conclusion based on the findings.

Chapter 2 Digital Filter and FPGA Literature Review

2.1 Introduction

Digital filtering is a diverse subject that can be accomplished in many different forms using many different methods. By using electronics, along with their computation powers, a digital filter can be used to cancel nearly any frequency range. There are multiple types of digital filters, infinite specifications, a wide array of electronics platforms where digital filtering can be performed, and varying methods of filtering within those platforms.

2.2 Digital Filters

2.2.1 FIR vs IIR Filter

Digital filters are generally classified as Finite Impulse Response (FIR) and Infinite Impulse Response (IIR). While there are advantages and disadvantages to using each type of filter both are valid methods of digital filtering.

The FIR filter is generally seen as the most commonly used type of filter for digital filtering. Major reasons behind this are that FIR filters are always stable and do not use a recursive method when performing computations. They are easier to program and more predictable, but this generally creates a need for higher filter orders to perform the same filtering as IIR filters which use a recursive method [12]. The non-recursive method increases the likelihood of being able to use fixed-point formatting because the main fractional numbers are the pre-known coefficients. FIR filters are created by running an input signal and delayed input signal through several multipliers and adders [11]. The basic structural makeup of a FIR filter is given in *Figure 1* with Z^{-1} representing a delay, and b_n representing the coefficient values.

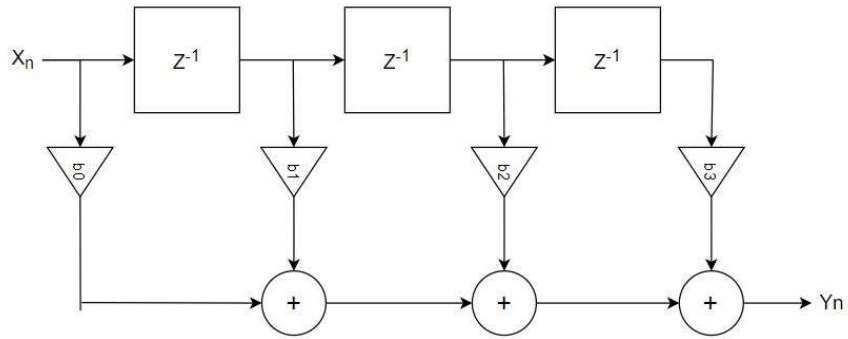


Figure 1: FIR Structure

IIR filters are used less in digital filtering because of their recursive behavior causing the programmer to need, most likely, to use floating point numbers. This makes fractional values more difficult to keep track of than fixed-point numbers. IIR filters are also less numerically stable than FIR filters. This is caused by using feedback paths [1].

A basic example of an IIR filter structure is given in Figure 2.

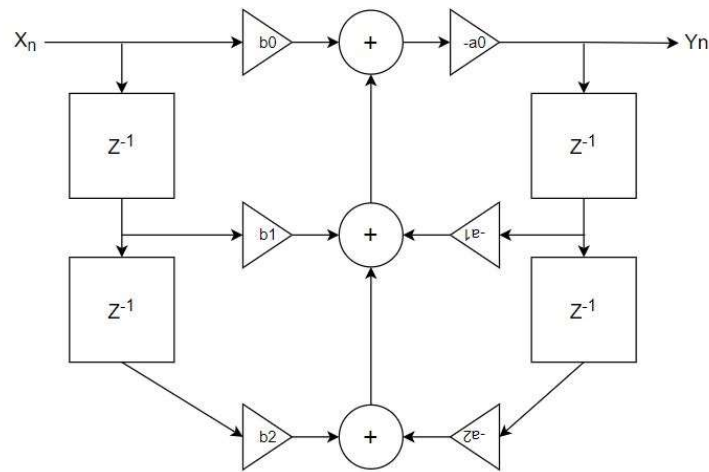


Figure 2: IIR Structure

2.2.2 Sampling Rate

The sampling rate, or sampling frequency, is a fundamental aspect of any digital filter. It is the rate at which samples from an analog input are taken and fed into the digital filter. Sampling rate is the samples taken per second. This rate alters every aspect of the filter including possible cutoff frequencies and filter coefficient values [7]. One

matter of importance when considering sampling rate is the Nyquist theorem. The Nyquist theorem states that the sampling frequency must be at least equal to or greater than twice the frequency of interest [8]. If this criterion is not met, then the signal being sampled can be misrepresented in digital form making it unusable. The formula is represented by *Equation 1*.

$$f_{nyquist} = \frac{f_{sample}}{2} \quad (1)$$

2.3 Hardware Elements

FPGAs have a variety of hardware components that are combined and interconnected to turn software into hardware. These basic building blocks are the foundation of an FPGAs functionality. The main hardware component within an FPGA is the Configurable Logic Block (CLB). A CLB consists of two different types of hardware, Flip-Flops, and Look-Up Tables (LUTS). These make up the base of an FPGA and are the blocks used most by the programmer. Another hardware aspect of FPGAs are DSP blocks. DSP blocks are prebuilt multiplier circuits that are used to decrease the usage of Flip-Flops and LUTs when a large number of multipliers are being used. This allows the Flip-Flops and LUTs to be utilized elsewhere in the software. Both RAM and Input/Output (I/O) blocks are other hardware components within an FPGA. RAM is used to store information, such as data sets, while I/O blocks connect the FPGA to the physical world. Both components allow FPGAs to be more practical to the real world making them more useful in industry [6].

2.4 FPGA Programming

2.4.1 Fixed-Point vs Floating-Point

There are two main methods for using fractional bits in binary form, fixed-point

notation, and floating-point notation. There are both advantages and disadvantages to using each.

Fixed-point notation uses a pre-set shift value to scale up and down fractional bits. It uses constant scaling as opposed to floating-point numbers which use dynamic scaling. This notation consists of a sign bit, decimal segment, and fractional segment. The programmer needs to keep track of this pre-set shift value so the changes can be counteracted to return the value to its original state once calculations using the value are completed. Fixed-point notation is easy to track but does not allow for the user to implement numbers of extreme magnitudes. This can be limiting to the user depending on the values that they need to store. This notation is very fast when working with base two numbers [3].

Floating-point numbers are very similar to fixed-point numbers but differ in the fact that extra storage is needed to track the magnitude and magnitude change of the value. This generally makes using floating-point numbers more difficult for the programmer because they have an extra value to keep track of. They consist of a sign bit, exponent portion, and mantissa portion. Floating-point numbers, unlike fixed-point numbers, can be used with values of larger magnitude, making them more versatile than fixed-point numbers. This notation is slower than fixed-point numbers especially when working with base two numbers [3].

2.4.2 Data Pipelining

Data pipelining is the process of collecting, storing, filtering, utilizing, and deleting data in a system. There are many types of data pipelining that change in

complexity depending on the users and system's needs [13]. Generally, relevant data is collected and stored within the system. That data is then filtered through, removing any anomalies that could have a negative effect on the systems functionality. This filtered through data is then utilized within the system allowing it to learn and adapt [2]. From here the used data can either be stored, if still relevant, or removed from the system to allow for more storage space to become available for future data. This process keeps systems relevant and up to date.

2.4.3 Edge Detection

Edge detection using an FPGA is integral for the completion of meaningful code. In VHDL, a rising or falling clock edge function can be used to detect if the system clock has just risen or fallen. These rising and falling clock edge functions cannot be used with non-clock signals. Instead, a basic gate circuit can be implemented that informs the user if the edge is rising for falling. All that is needed for this circuit to work is the previous signal value and the current signal value. From here, those values are run through an inverter as well as an AND gate. The circuit used for edge detection is given in *Figure 3*.

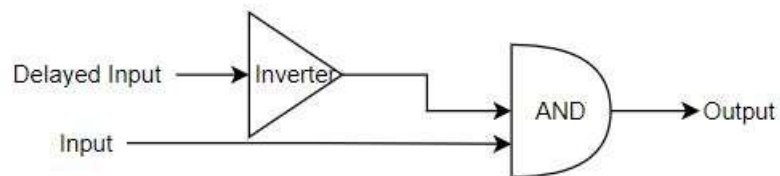


Figure 3: Edge Detection Using Gates

Figure 3 shows a basic rising edge detection circuit. If the previous signal was set to zero and the current signal was set to one, then the output from the AND gate would be equal to one. On the next clock cycle, after the edge is detected, both the previous and current signals will be set to one causing the output of the AND gate to fall to zero. To detect a

falling edge of the signal, the inverter would be placed on the current input rather than the delayed input.

2.5 Slack

Slack is the amount of time between when a signal leaves one register within an FPGA design, travels across a connect, and enters into the next register. A positive slack indicates that the signal had enough time to make it from one register to the next register before a second system clock period begins [10]. A more positive slack means that this signal had more time to spare from one register to the next. A negative slack indicates that the signal left the original register but did not have enough time between system clock edges to make it to the next register. This leaves the next register at some intermediate value that has the possibility of being unreadable. The main way to fix a negative slack is to add registers between existing registers. This means that less information will need to be processed after each clock edge [10]. By coding in a state machine format, the programmer can control the number of clock edges required to complete a process by placing each function in a separate state; therefore, giving them more control over the amount of slack within each clock period. Reducing negative slack is not the only goal in slack reduction, increasing positive slack is as well. With a more positive slack the system can be run at higher system clock speeds. This can make an entire system more efficient and effective in completing its designed task.

Chapter 3 Methodology

3.1 Hardware and Software Tools

3.1.1 MATLAB

MATLAB, more specifically MATLAB Filter Designer, was first used during the FIR filter design. This software allows the user to create completely customized filters followed by giving an analysis of the filter. For the purposes of this thesis, the Response Type, Design Method, Filter Order, and Frequency Specifications settings were all used to create the desired filter. The Response Type setting allows the user to select the type of filter that is being created. The Design Method allows the user to choose between an IIR and FIR filter as well as the design method. The Filter Order setting permits the user select the desired number of filter coefficients. Lastly the Frequency Specifications allows the user to set the characteristics of the filter [4].

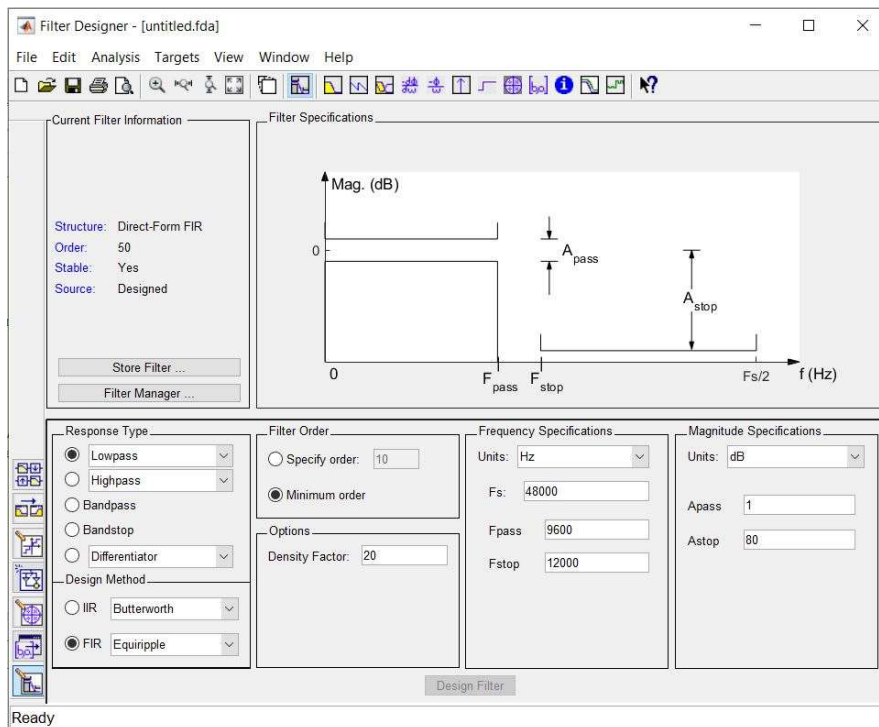


Figure 4: MATLAB Filter Designer Default Screen

Once designed, the filter can be analyzed by the software. The Frequency Response, Impulse Response, and Filter Coefficients were found using the software's analysis capabilities.

The simulation portion also used MATLAB to create a text file with test waves to simulate the programmed filter at different frequencies. The text file was created using the cosine function, recording values at a certain sampling frequency.

3.1.2 ModelSim

The ModelSim software gives the user a programming environment which then allows the user to simulate behavioral, RTL, and gate-level code [9]. ModelSim was used to create and debug the digital FIR filter in VHDL. Once created, a testbench was made in ModelSim that fed waveforms to the filter allowing the code to be simulated and analyzed at varying frequencies.

3.1.3 Quartus Prime and DE-10 Standard

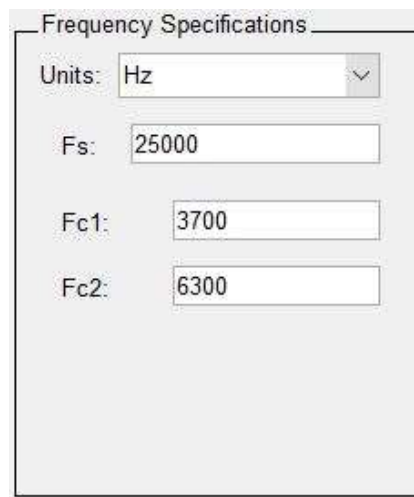
Quartus Prime is a software that provides everything necessary to design and implement FPGA logic [5]. For the purposes of this thesis, the Synthesis and Timing Analysis capabilities of Quartus Prime were used. When initially creating a project, a specific board is selected that the user's code can be uploaded to, allowing the software to determine the percentage of hardware used on the board. The Cyclone V DE-10 Standard was the selected board which includes a 12-bit ADC, ARM Cortex-A9 Dual-Core processor, 110,000 LEs, and 41509 ALMs [14].

3.2 Digital Filter Design

3.2.1 MATLAB Filter Design

The first step in the process was creating a band-stop filter using MATLAB Filter

Designer. The chosen filter type was a FIR filter as they are more code friendly than IIR filters due to IIR filters' recursive nature. The designed filter's goal was to filter out a signal with a frequency of 5KHz with a user selected sampling frequency of 25KHz, which is more than double the desired filtering frequency. One of the main points of focus was creating a filter that uses a limited amount of hardware, implying that it needed to be a low-tap filter. To accommodate this, the number of filter coefficients was limited to 13. A side-effect of limiting the number of coefficients was a decrease in filter performance compared to one that contained many filter coefficients. One aspect of the diminished performance is that the low-tap filter contained more of a gradual upper and lower transition band than that of a filter with a high coefficient count. These gradual transition bands increased the bandwidth of the filter. Another aspect that was taken into consideration when choosing the cutoff frequencies was having a low amount of passband ripple. Taking all of this into consideration and through trial-and-error, the lower cutoff frequency (f_L) was set to 3.7KHz and the upper cutoff frequency (f_H) was set to 6.3KHz.



Frequency Specifications	
Units:	Hz
Fs:	25000
Fc1:	3700
Fc2:	6300

Figure 5: 13-Coefficient Filter Frequency Specifications

The filter was created using the windowing method utilizing the frequency specifications shown in *Figure 5*. MATLAB filter designer allows the user to select the type of windowing for the design and the Kaiser method was chosen. With the Kaiser windowing method both the passband and stopband ripple sizes are only minorly effected by any alteration in the window length. Through the filter designer, a frequency response for the filter was obtained, as shown in *Figure 6*, giving a visual reference of the filter's performance.

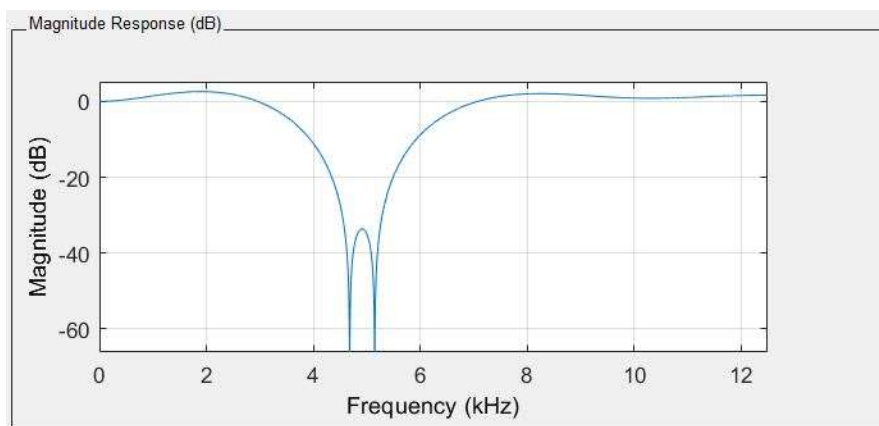


Figure 6: 13-Coefficient Filter Magnitude Response

As shown by the frequency response of the filter, in *Figure 6*, there is limited passband ripple while there is a more significant amount of stopband ripple. The stopband ripple should not diminish the filters performance as it has a worst-case a magnitude of about -35dB. The magnitude response also gives a visual indicator of the gradual slope of the transition band. Next, the impulse response and coefficients of the filter were analyzed, given in *Figure 7* and *Figure 8*.

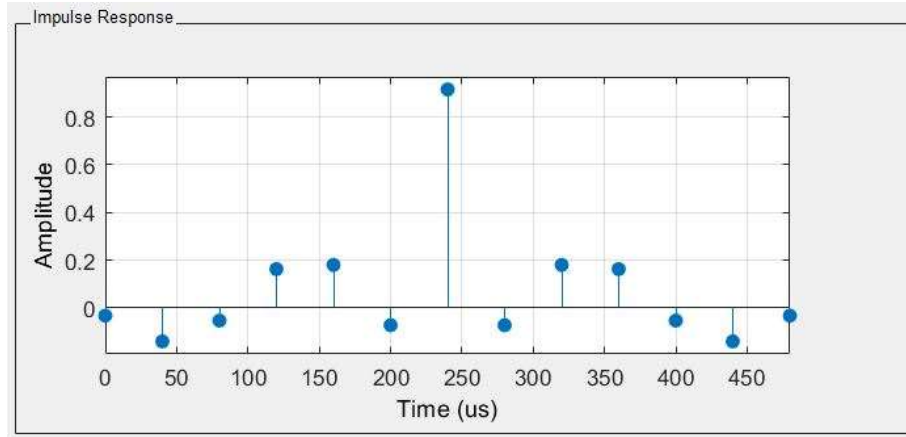


Figure 7: 13-Coefficient Filter Impulse Response

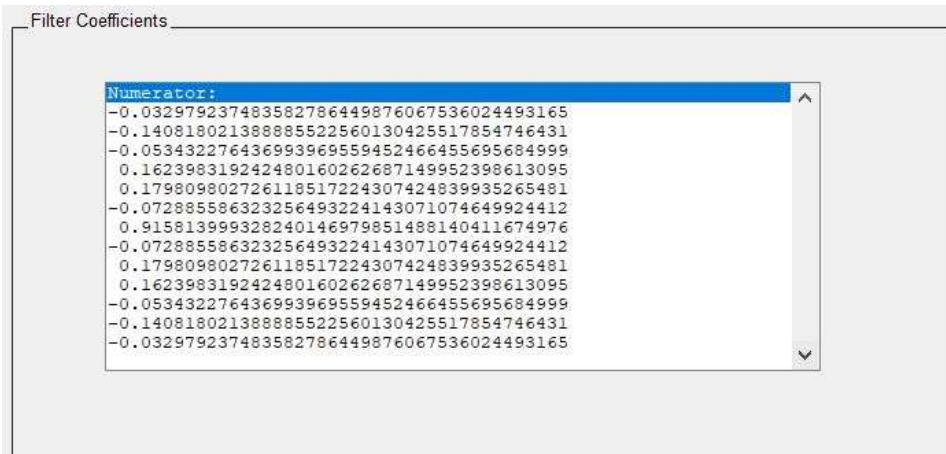


Figure 8: 13-Coefficients for Filter

The impulse response shown in *Figure 8* displays the graphical representation of the filter coefficients in *Figure 8*. One item of importance is that the magnitude of the filter coefficients are all within a range of 0.88283 of each other. This will be a main factor when deciding whether to use fixed-point or floating-point numbers in the VHDL design. Another important item is that the filter coefficients involve both positive and negative numbers. When coding the filter, signed numbers will need to be used throughout the VHDL design and thus need to be converted to unsigned logic vectors before being sent to a Digital-to-Analog Converter (DAC) for real world use.

3.2.2 VHDL Filter Design

Once the design of the low tap band-stop FIR filter was completed, the next step was creating the filter in the ModelSim environment using VHDL. As noted previously, the filter coefficients were positive and negative decimal values all near the same magnitude, making fixed-point notation with two's complement the best option for implementing the coefficients within the code. Using Excel, the filter coefficients were transformed into usable binary numbers, the process being shown in *Table 1*.

Table 1: Fixed-Point Notation Calculation

Filter Coefficients	Coefficient * 2^N	Rounded	Signed 12-bit Coefficients
-0.03298	-16.88536959	-17	11111101111
-0.14082	-72.09882695	-72	111110111000
-0.05343	-27.35732554	-27	111111100101
0.16240	83.14793945	83	000001010011
0.17981	92.062619	92	000001011100
-0.07289	-37.3174202	-37	111111011011
0.91581	468.8967677	469	000111010101
-0.07289	-37.3174202	-37	111111011011
0.17981	92.062619	92	000001011100
0.16240	83.14793945	83	000001010011
-0.05343	-27.35732554	-27	111111100101
-0.14082	-72.09882695	-72	111110111000
-0.03298	-16.88536959	-17	111111101111

The first step in calculating the signed 12-bit filter coefficients from the decimal filter coefficients was entering the decimal filter coefficients into *Equation 2*.

$$\text{Scaled Filter Coefficient} = \text{Filter Coefficient} * 2^N \quad (2)$$

In *Equation 2*, N was chosen to be nine allowing there to be one signed bit as well as two whole number bits adding together to give the overall 12-bits. The two whole number bits were unused; however, they were installed for ease of use in the case of filter alterations.

Once calculated, the scaled filter coefficients then needed to be rounded to the nearest whole number. From there, the numbers were transformed into 12-bit signed form with the application of two's complement.

Following the conversion of the filter coefficients, these values were then entered into the filter's VHDL codes, in array format, under the name `coeffs` as performed in lines 1 and 4 of *Figure 9*.

```

type coeffs is array (0 to 12) of signed(11 downto 0);           --coefficient storage
type previous_data_storage is array (0 to 12) of signed(11 downto 0); --stores previous data
type product_storage is array (0 to 12) of signed(23 downto 0);  --coefficients * input signal
signal coefficient : coeffs := (0 => "111111101111", 1 => "111110111000", 2 => "111111100101", 3 => "000001010011"

```

Figure 9: Stored Filter Coefficients

Note that in *Figure 9*, the populating of the array coefficient, in line 4, is cut off as there are too many values to fit in a single image. Next, as 12-bit values were entered from the Analog-to-Digital Converter to the filter entity, data pipelining was carried out to store the necessary 13 input values in chronological order. The values were first converted into type signed before being stored for future use. At each rising edge of the sample clock, all the coefficient values were needed to calculate the output value of the filter at any given moment. The use of data pipelining was demonstrated in the code from *Figure 10*.

```

signal_memory <= signed(original_signal) & signal_memory(0 TO 11);

```

Figure 10: Data Pipelining

Since data was entered at the sampling frequency of 25KHz and the default clock speed of the DE-10 Standard board is 50MHz, the system clock frequency needed to be divided to ensure the timing of the VHDL filter was lined up with timing of the MATLAB designed filter. This was done using the clock dividing technique of counting several 50MHz clock cycles and creating a toggle after a certain period of time of a

sample clock variable. The technique is shown in *Figure 11*.

```
elseif rising_edge(clk) then
    frequency_sample <= frequency_sample + 1;

    if(frequency_sample = "001111101000" ) then
        frequency_sample <= "000000000000";
```

Figure 11: VHDL Clock Divider

In *Figure 11*, frequency sample is increased at every rising edge of the system clock. Once frequency sample reaches the desired value, frequency sample is reset which will be followed by the sample clock toggle. To find the desired value that frequency sample must equal, both the system clock frequency and the sampling frequency were used in *Equation 3*.

$$frequency_sample = \frac{\left(\frac{System\ Clock}{Sampling\ Frequency}\right)}{2} \quad (3)$$

It should be noted that division by two was used since the system was run on only rising clock edges and not any falling clock edges, meaning only half of the clock period was needed. While the use of `rising_edge()` is acceptable when referring to the system clock, it cannot be used with any clock created by clock division. Instead, the newly divided clock used registers and gates to calculate when a rising edge occurred. By storing both the previous (`sample_clk_delay`) and current (`sample_clk`) values of the newly divided clock into registers, then running them through a simple inverter and AND gate, the type of edge change can be determined. This process is produced in the code in *Figure 12*.

```
sample_clk_delay <= sample_clk;
sample_clk <= not sample_clk;
sample_clk_edge <= sample_clk and (not sample_clk_delay);
```

Figure 12: Gate Edge Detect

From here, based upon the sample clock, the multiplication and addition of the FIR filter were performed using the input signal and filter coefficients. One problem that occurred throughout the process was after the filter calculations, a very small negative number was output from the filter. Since the FPGA being used can only have positive voltages, a negative output will not be functional. Likely causes of this are passband ripple or errors due to filter coefficient rounding. The developed solution to this issue was a simple software fix. The process of this solution is first checking if the output value was going to be negative by investigating the signed bit and then placing the output value at zero if the value was negative. If this problem did not occur and the output was a positive value, then the output was turned from a signed value into a logic vector value. Next, to account for the fixed-point notation of the filter coefficients, the output value was shifted right logical by the previously determined value of 9. This negates the previous alterations made to the filter coefficients. Both the software fix for the output of small negative numbers as well as the fixed-point transformation are demonstrated in the code illustrated in *Figure 13*.

```
if sum(23)='1' then
    filtered_signal <= "000000000000000000000000";
else
    filtered_signal <= std_logic_vector(sum srl 9);
end if;
```

Figure 13: Negative Value Solution

The variable `filtered_signal` is the 24-bit digital output that will be fed into a DAC before being output from the FPGA. A 24-bit output was selected because that is the resolution of the DAC on the DE-10 Standard board while an input of 12-bits was chosen because that is the resolution of the ADC on the DE-10 Standard board.

3.3 Filter Simulation

To simulate the band-stop filter, MATLAB was used to create a text file populated with data that was then fed to the filter entity using a testbench. To create the data in MATLAB, the sampling frequency was applied to find the amount of time between when samples would be taken. From here, a time vector was created and utilized along with the cosine function and desired wave frequency to generate a test wave. The data was shifted to be positive, and its magnitude increased to ensure a more complete usage of the 0V to 5V range that the FPGA being used is capable of. The outcome was a vector populated with 125,000 test values. Each point along this test wave was then exported to a text file which was then saved in the FIR filter's project folder. This process is demonstrated in *Figure 14*, while a 5.6KHz test wave was plotted and is shown in *Figure 15*.

```
fs = 25000;  
dt = 1/fs;  
stoptime = 5;  
t = (0:dt:stoptime-dt)';  
fc = 5600;  
y = (cos(2*pi*fc*t) + 1) * 200;  
plot(y);  
fileID = fopen('test_data.txt','w');  
fprintf(fileID,'%f\n',y);  
fclose(fileID);
```

Figure 14: Test Data Code

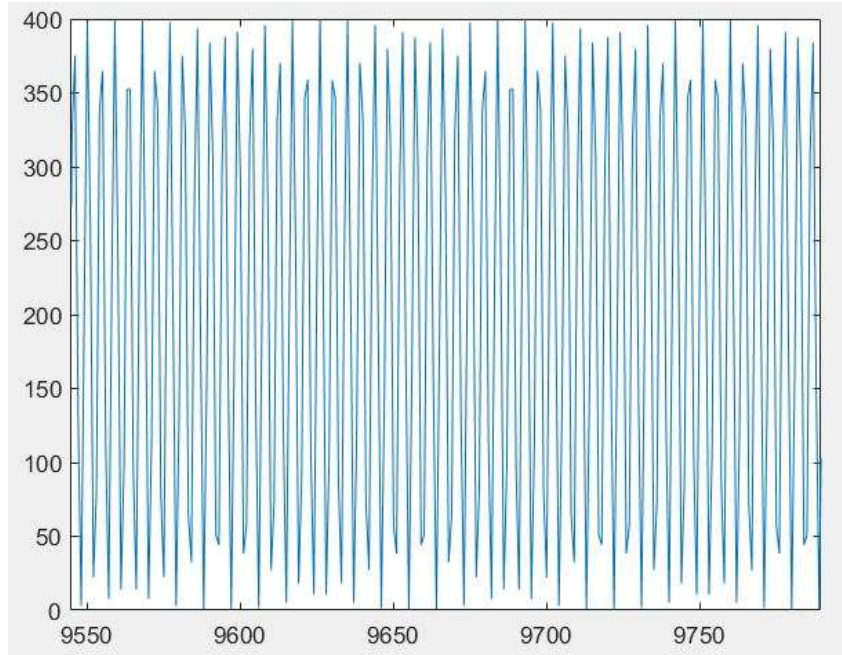


Figure 15: Test Data Visualization

The next step in the simulation process was importing the newly created test data into ModelSim followed by feeding it to the FIR filter entity. By creating a testbench and using the `readline` function along with the `read` function within ModelSim, the data was able to be extracted from the text document. Next, that data was turned into a 12-bit logic vector which was then sent to the filter as the unfiltered signal input. This entire extraction process needed to be timed up with the sampling frequency to ensure there would not be a misrepresentation of test data. This was done by executing the data extraction on the rising edge of a created 25KHz sample clock signal. The creation of the filter test input signal is given in *Figure 16*.

```

process(clk_sample)
  variable line_test_input : line;
  variable input: integer;
  begin
    if rising_edge(clk_sample) then
      readline(test_input, line_test_input);
      read(line_test_input, input);
      original_signal <= std_logic_vector(to_unsigned(input, 12));
    end if;
  end process;

```

Figure 16: VHDL Test Data Extraction

From here, using a 50Mhz system clock signal as well as clearing the filter reset bit, several sinusoidal waves were tested at frequencies of interest using ModelSim. These frequencies were selected using the desired frequency filtering values for the band-stop filter. The first of these frequencies being 60Hz. This frequency was selected to show that the filter allowed frequencies outside of the range of interest to pass through with minimal alteration. The resulting simulation is shown in *Figure 17*.

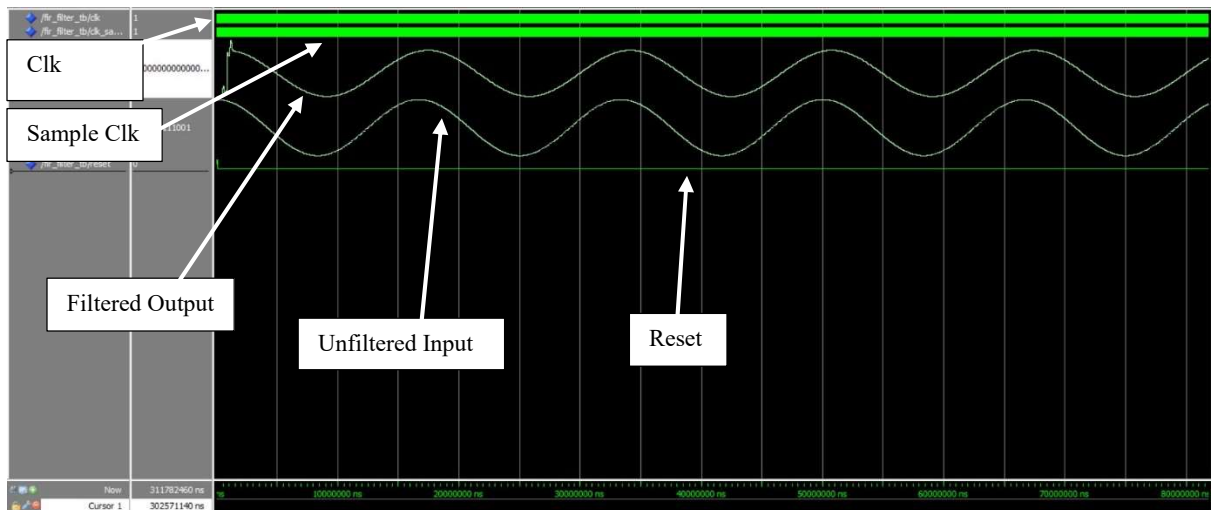


Figure 17: 60Hz Filter Simulation

In *Figure 17*, there is negligible modification to the wave due to the filtering excluding the several initial values after the reset bit is cleared. Once there was enough data, the filter entered into a steady state with only a slight delay. Another method of demonstrating the filter is by combining two waves, one with the frequency being tested

and another with a completely passable signal. The passible signal was selected to be 60Hz while the second signal was selected to be 1KHz. It was expected that for this scenario since both waves' frequencies are outside of the filters bandwidth the unfiltered input will be the exact same as the filtered output. The results are shown in *Figure 18*.

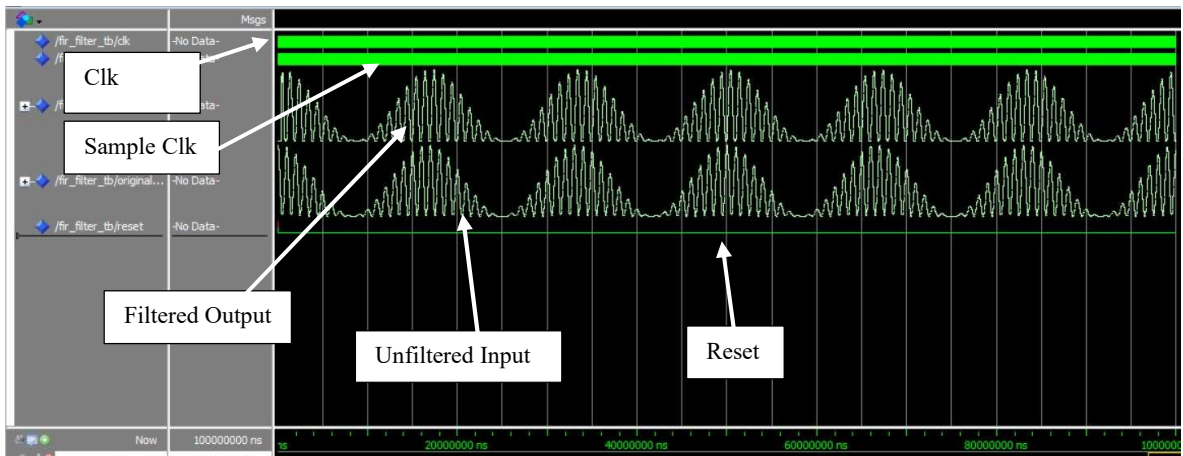


Figure 18: 60Hz Combined with 1KHz Filter Simulation

In *Figure 18*, as expected both the input and output waves were identical.

The next frequency of interest was 5KHz. Any value at this frequency should have been filtered out. The outcome of the simulation is given in *Figure 19*.

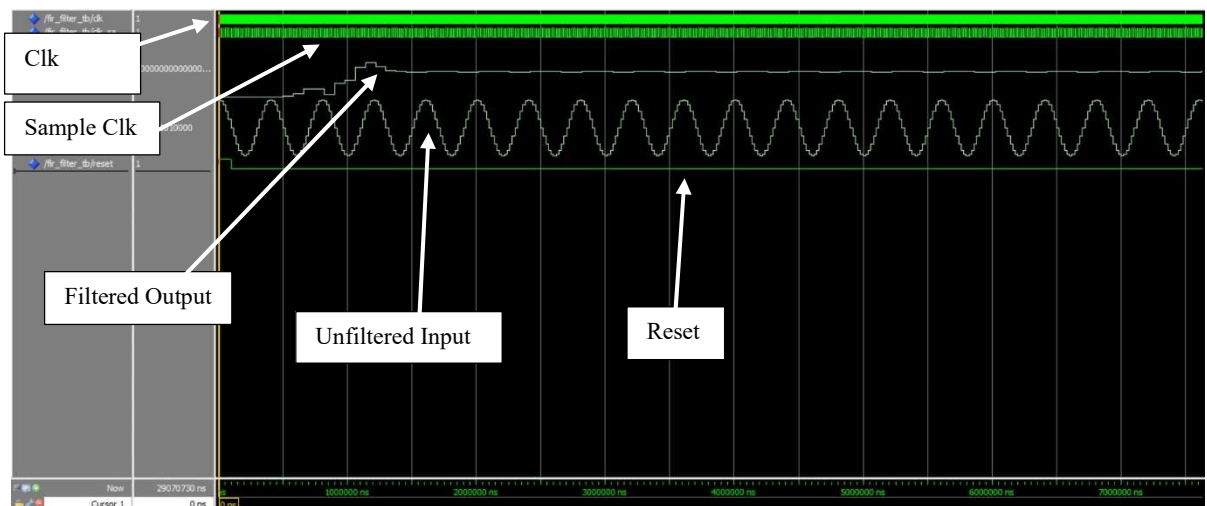


Figure 19: 5KHz Filter Simulation

As shown in *Figure 19*, there is an extended initial delay compared to that of the

non-filtered frequency; however, the 5KHz wave was successfully filtered. Once in steady state, there was a slight oscillation between three bits meaning that there was not complete filtering. This is due to real world factors, one of which is rounding. Next, the filter was tested by combing two separate signals. The passible signal was selected to be 60Hz while the second signal was selected to be 5KHz. It was expected that for this scenario that the 5KHz signal would be filtered out leaving only the 60Hz. The results are shown in *Figure 20*.

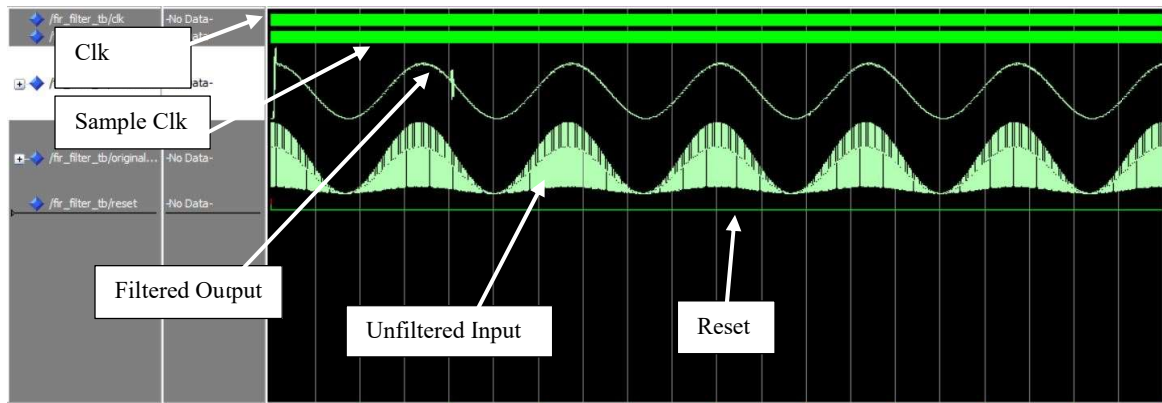


Figure 20: 60Hz Combined with 5KHz Filter Simulation

As expected, *Figure 20* shows the 5KHz portion from the wave being filtered out.

The next values that were examined were that of the upper and lower cutoff frequencies, being 3.7KHz and 6.3KHz, respectively. These are points of interest because there was expected to be some alteration to the wave, while there was not expected to be complete filtering. The simulation of the 3.7KHz wave is provide in *Figure 21*.

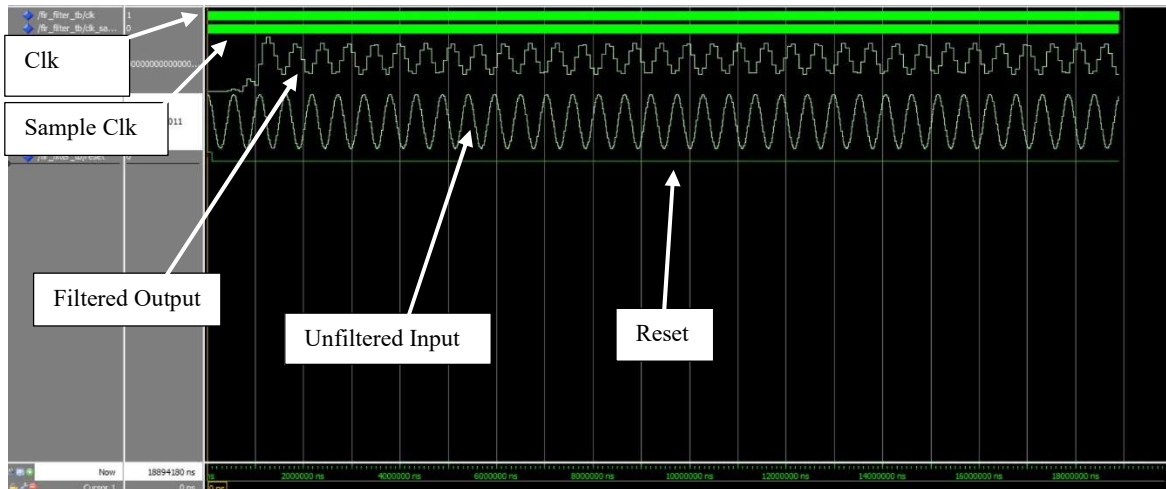


Figure 21: 3.7KHz Filter Simulation

As expected, the 3.7KHz input signal did not run through the filter without alteration. While the signal is recognizably a sinusoidal shape, it clearly differs by a range of magnitude from 60%-80% of the original signal. Not only was the peak-to-peak value reduced, but the entire signal was compromised. It should be noted that the output signal does begin to repeat itself after every five output signal periods. Next, the filter was tested by combining two separate signals. The passible signal was selected to be 60Hz while the second signal was selected to be 3.7KHz. It was expected that for this scenario that the 3.7KHz signal would be slightly filtered out leaving both the 60Hz and 3.7KHz wave with the 3.7KHz wave being at a 60%-80% decreased magnitude. The results are shown in *Figure 22*.

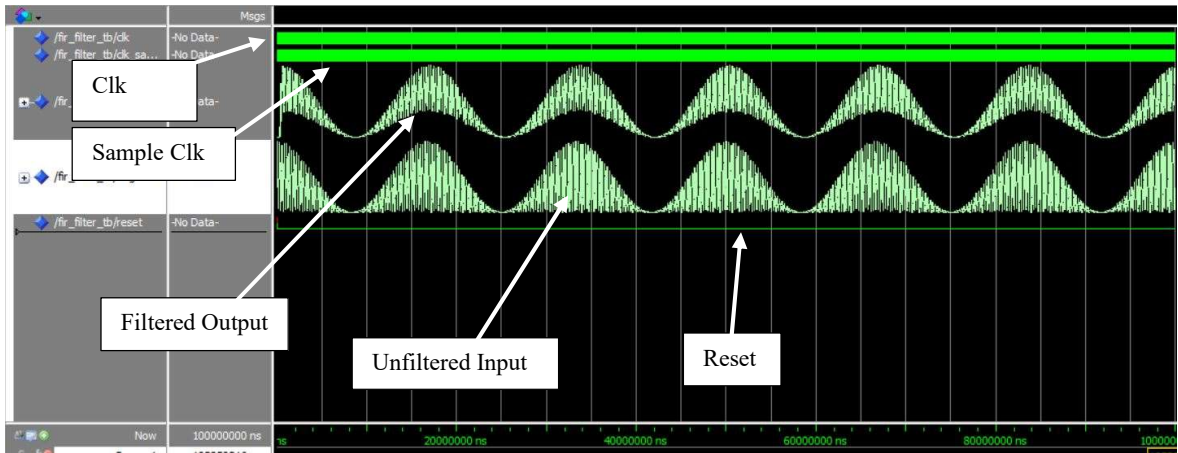


Figure 22: 60Hz Combined with 3.7KHz Filter Simulation

The results shown in *Figure 22* were as expected with the 3.7KHz waves magnitude being slightly altered at 60%-80% the original magnitude and the 60Hz wave going unchanged.

This same result was expected for the 6.3KHz wave as the 3.7KHz wave, but the results differed. The simulation outcome is shown in *Figure 23*.

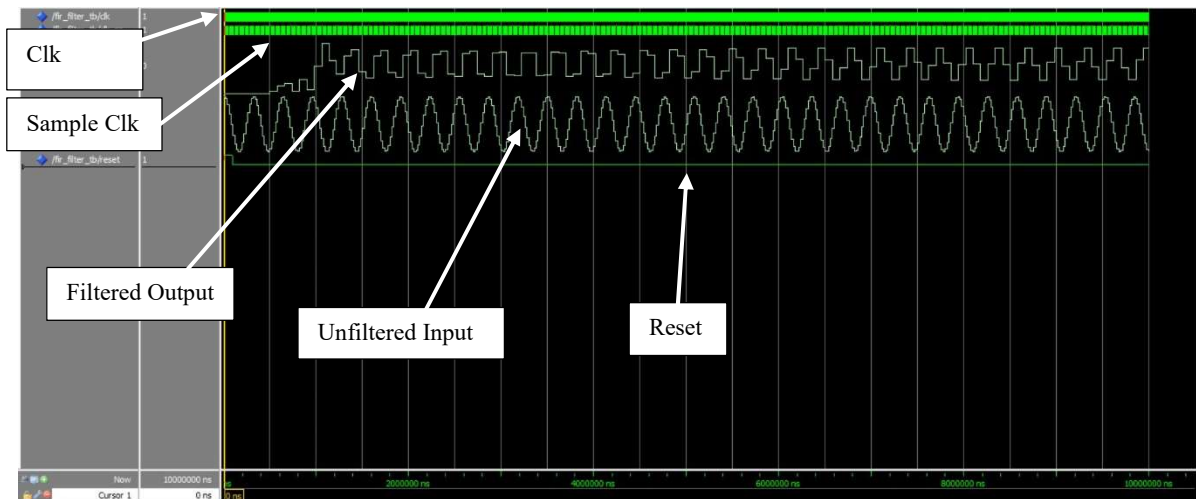


Figure 23: 6.3KHz Filter Simulation

From *Figure 23*, there is a recognizable output oscillation; however, it is plain to

see that it is severely distorted. With the amount of distortion shown, any information that the signal carried would most likely be lost. The signal also continues to change indicating that the signals filtering is not completed. Next, the filter was tested by combing two separate signals. The passible signal was selected to be 60Hz while the second signal was selected to be 6.3KHz. It was expected that for this scenario that the 6.3KHz signal would be slightly filtered out leaving both the 60Hz and 6.3KHz wave with the 6.3KHz wave being at a 60%-80% decreased magnitude. The results are shown in *Figure 24*.

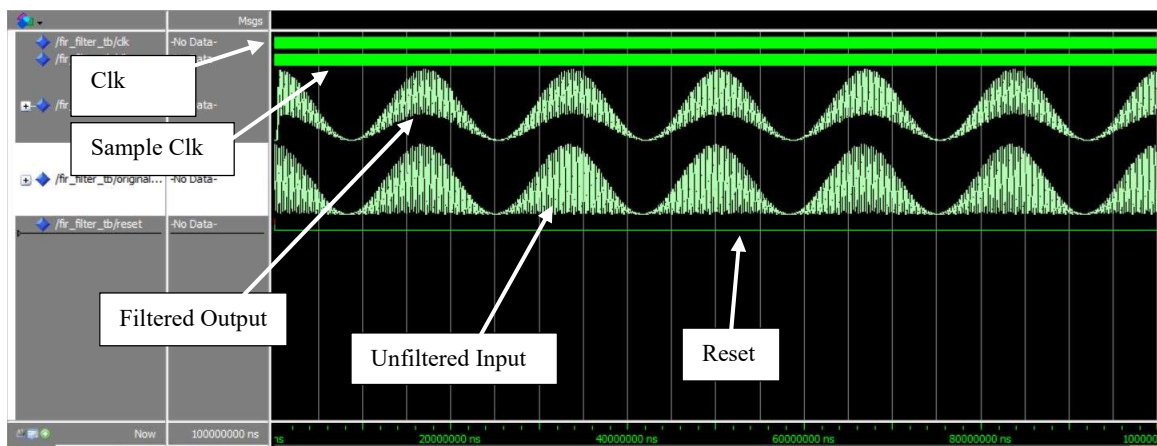


Figure 24: 60Hz Combined with 6.3KHz Filter Simulation

The results shown in *Figure 24*, were as expected.

Through visual comparison, the filters performance was better at the lower cutoff frequency than at the upper cutoff frequency. Overall, it did not perform well around the boundaries of the desired frequency cutoff range. Outside and well within the filtering frequency range, the designed filter performed sufficiently.

3.4 Filter Implementation

Implementing the filter onto an FPGA was done through Quartus Prime using the DE-10 Standard board. The components used in the implementation of the FIR filter

consisted of an analog to digital converter (ADC), switch, light emitting diode (LED), and function generator. The DE-10 Standard contained each of these elements except for the function generator. By setting a threshold in the VHDL code and turning on an LED every time the input signal was above this value, the LED was able to indicate the strength of the signal. When the frequency of the input signal was outside of the filters cancelling bandwidth, the LED was at its brightest. When the frequency of the input signal was within the filters cancelling bandwidth, the LED would dim or turn completely off depending on the magnitude of the signal that the filter cancelled. Summarized shortly, by using Pulse Width Modulation (PWM) and the characteristics of a sinusoidal input wave, the LED would indicate the input signals strength. This in turn showed the filtering of the input signal.

3.4.1 ADC Control Module

First, the ADC needed to be created to turn the input analog signal to a digital signal. The ADC needed to be controlled using an ADC module which was created in VHDL and added to the project. This control module consisted of several signals including signals by the name of CS, DIN, DOUT, and SCLK. The signal CS was given by the FPGA indicating the start of the operation, DIN was sent by the FPGA giving the pin address that the analog signal would be received at, DOUT was sent from the ADC and contained the digital signal that was being sent to the FPGA, and SCLK was the rate at which samples were taken. The ADC took 16 clock edges to perform one round of operations and send the FPGA a 12-bit number. The simulated operation of the ADC control module is given in *Figure 25*.

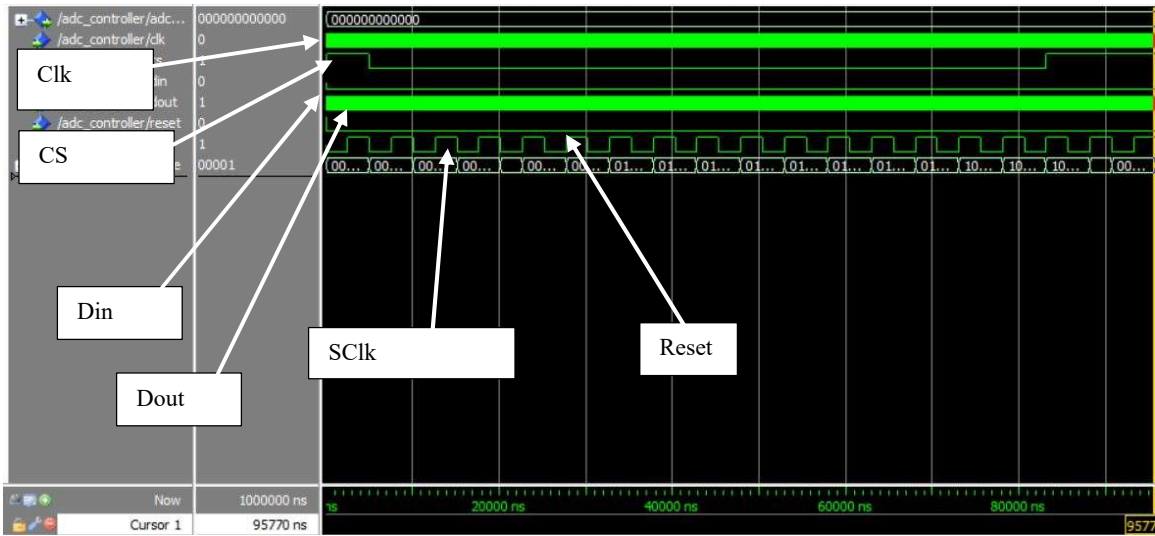


Figure 25: ADC Control Module Simulation

In Figure 25, CS indicates the start of an operation. The next three bits sent are the DOUT bits and are sent on the falling edge of SCLK to prevent any race conditions between DOUT and DIN. The remaining 12 rising clock edges of SCLK are used for DOUT values to be sent to the FPGA for processing.

3.4.2 Physical Implementation

After the ADC control module was created and simulated, it was combined with the FIR filter VHDL codes making the overall system inputs and output clear. The inputs and outputs of the system were then assigned to physical components using Quartus Prime and the DE-10 Standard manual. These connections are shown in Figure 26 with the column labeled “To” being the signal and the column labeled “Value” being the physical pin that the signal was attached to.

	tatu	From	To	Assignment Name	Value	Enabled
1	✓		in clk	Location	PIN_AF14	Yes
2	✓		in reset	Location	PIN_AB30	Yes
3	✓		in dout1	Location	PIN_V23	Yes
4	✓		out din1	Location	PIN_W22	Yes
5	✓		out sclk1	Location	PIN_W24	Yes
6	✓		out cs1	Location	PIN_Y21	Yes
7	✓		out filt...out1	Location	PIN_AA24	Yes

Figure 26: Pin Assignments

A function generator was used to provide an input signal for testing. The test setup is given in *Figure 27*.

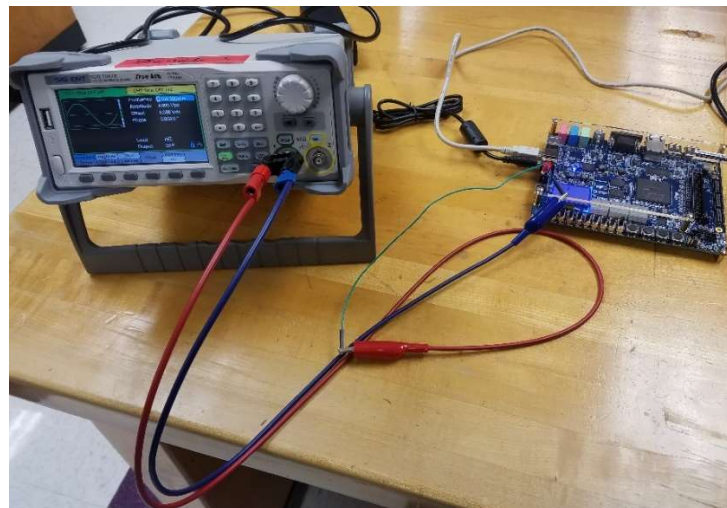


Figure 27: Experimental Setup

The first test signal was a 1KHz, 4-volt peak-to-peak sinusoidal wave. It was expected that the LED would be completely on as there was no filtering occurring at this frequency. The outcome ended up as expected. The input wave and output LED values are shown in *Figure 28* and *Figure 29*, respectively.

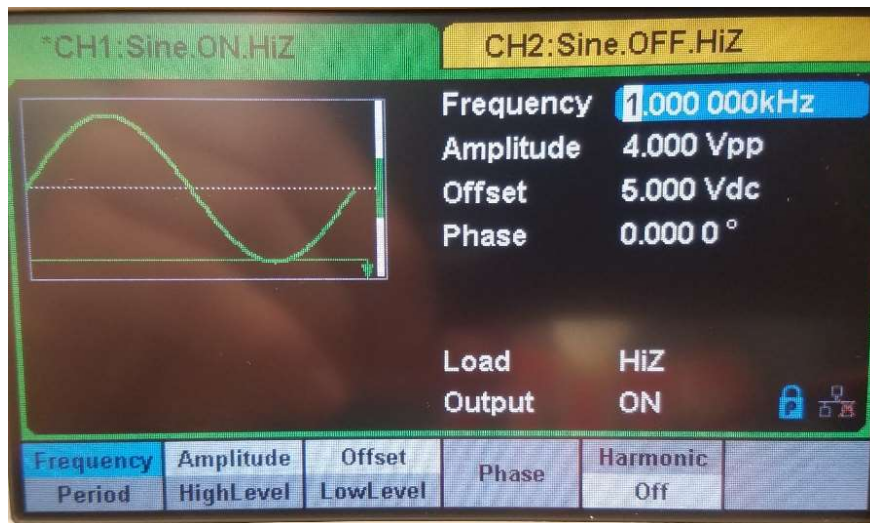


Figure 28: 1KHz Sinusoidal Test Wave



Figure 29: 1KHz Test Wave LED Output

The next test signal was a 3.7KHz, 4-volt peak-to-peak sinusoidal wave. It was expected that the LED intensity would be slightly dimmed compared to that of the intensity of the previous LED. The outcome ended up as expected with a slight amount of filtering, resulting in a less intense LED light. The input wave and output LED values are shown in *Figure 30* and *Figure 31*, respectively.



Figure 30: 3.7KHz Simusoidal Test Wave



Figure 31: 3.7KHz Test Wave LED Output

Next, the test signal was a 5KHz, 4-volt peak-to-peak sinusoidal wave. It was expected that the LED would be completely off because the signal would be filtered, causing it to entirely fall below the threshold for the LED to turn on. The outcome ended up as expected with the LED turning off. The input wave and output LED values are shown in *Figure 32* and *Figure 33*, respectively.

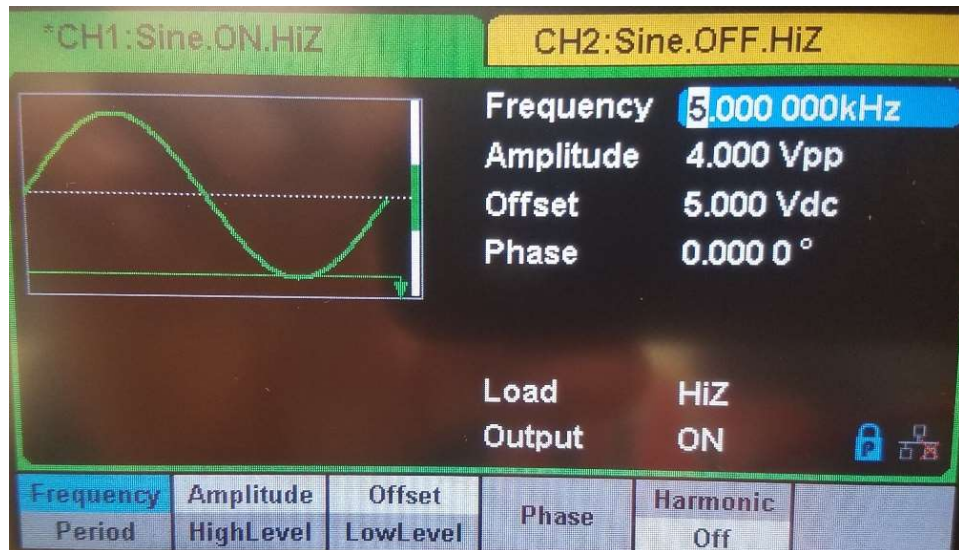


Figure 32: 5KHz Sinusoidal Test Wave



Figure 33: 5KHz Test Wave LED Output

The next test signal was a 6.3KHz, 4-volt peak-to-peak sinusoidal wave. It was expected that the LED intensity would be slightly dimmed compared to that of the intensity of the previous LED. The outcome ended up as expected with a slight amount of filtering, resulting in a less intense LED light. The input wave and output LED values are shown in *Figure 34* and *Figure 35*, respectively.

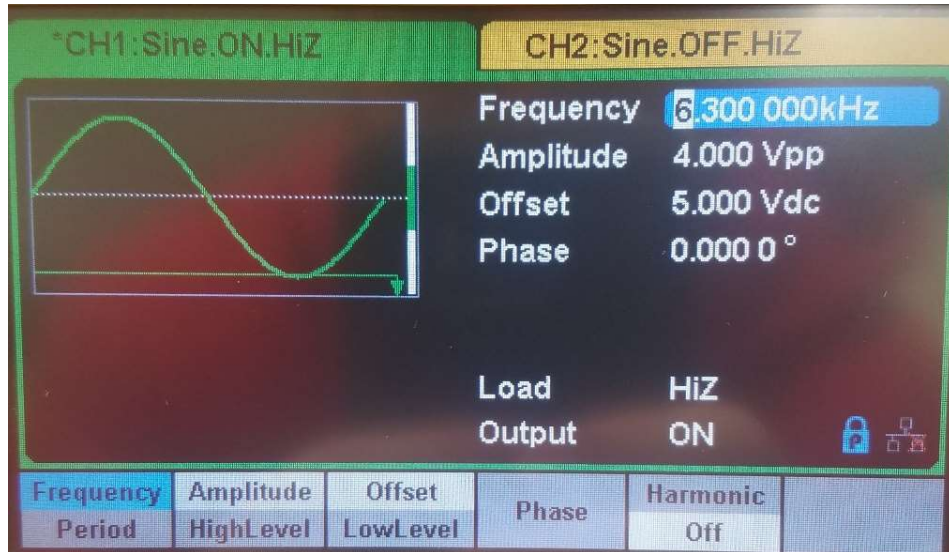


Figure 34: 6.3KHz Sinusoidal Test Wave



Figure 35: 6.3KHz Test Wave LED Output

Finally, the test signal was an 8KHz, 4-volt peak-to-peak sinusoidal wave. It was expected that the LED would be completely on because the signal would not be filtered. The outcome ended up as expected with the LED turning on. The input wave and output LED values are shown in *Figure 36* and *Figure 37*, respectively.

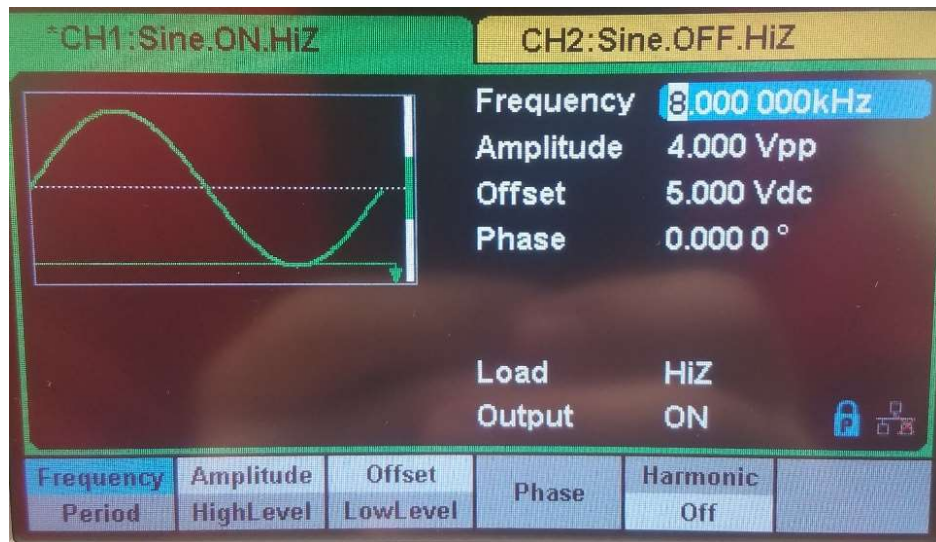


Figure 36: 8KHz Sinusoidal Test Wave



Figure 37: 8KHz Test Wave LED Output

The implementation of the band-stop FIR filter was successful. The results from this physical display were in line with the results from the filter simulation performed in ModelSim.

3.5 State Machine Format

Positive slack is necessary for any type of FPGA implementation. Fast clock speeds and overprocessing within a clock period can result in negative slack. This makes slack reduction tremendously important in any FPGA design. By reducing the amount of information processed in a single clock cycle, the systems slack can be improved. This can be done through the implementation of state machine formatted code which uses Look-Up Tables (LUTs) on FPGAs in place of registers.

Due to the fact that the FIR filter was run at a sampling frequency of 25KHz, there were enough clock cycles between sample clock trigger edges to perform all the FIR calculations on separate clock periods. This action reduced the number of computations within a single system clock period which in turn reduces the negative slack of the system. One downside to implementing the FIR filter in state machine format is if there are too many calculations within a single sample clock period, then the filter will not be prepared to output a correct value in time. Another downside is that the VHDL code will be much longer and less visually appealing than non-state machine formatted code but can be easier to follow. State machine format may also use different hardware which could limit the user depending on the resources at the user's disposal.

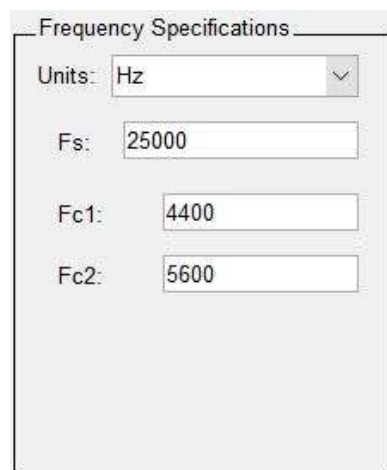
With all the possible implications in mind, the previously designed band-stop FIR filter was coded into state machine format. This permitted an investigation into the amount of slack reduction and a comparison of FPGA hardware use between state machine formatted code and non-state machine formatted code. The 12th order non-state machine formatted code was 66 lines in length while the newly created 12th order state machine formatted code was 154 lines in length. This confirmed the statement that state

machine formatted code is generally longer in length than non-state machine formatted code based around the same design.

3.6 Higher Order Filter

Filter order is one of the most important factors in filter performance. With a higher number of coefficients, the transition bands can be steeper and allow for a narrower filter. The cost of having more filter coefficients in a digital filter is both a decrease in initial filter speed and an increase in FPGA hardware use. To examine the relationship between FPGA hardware use and filter order, a second filter was designed using MATLAB Filter Designer and implemented using the same format of code as used for the 13-coefficient filter.

The newly designed band-stop filter was focused on cancelling out the same frequency as before, 5KHz, while still using the previous sampling frequency of 25KHz. It was designed using 25 filter coefficients as opposed to the previous 13 filter coefficients. One change to the filter design was how narrow the band-stop filter could be. This range was found using the same method as the 13-coefficient filter and is shown in *Figure 38*.



The image shows a dialog box titled "Frequency Specifications" with the following fields:

Parameter	Value
Units	Hz
Fs	25000
Fc1	4400
Fc2	5600

Figure 38: 25-Coefficient Filter Frequency Specifications

The frequency response of the newly designed filter had slightly more passband ripple with no stopband ripple. The value for f_L was increased by 700Hz while the value for f_H was decreased by 700Hz. This is given in *Figure 39*.

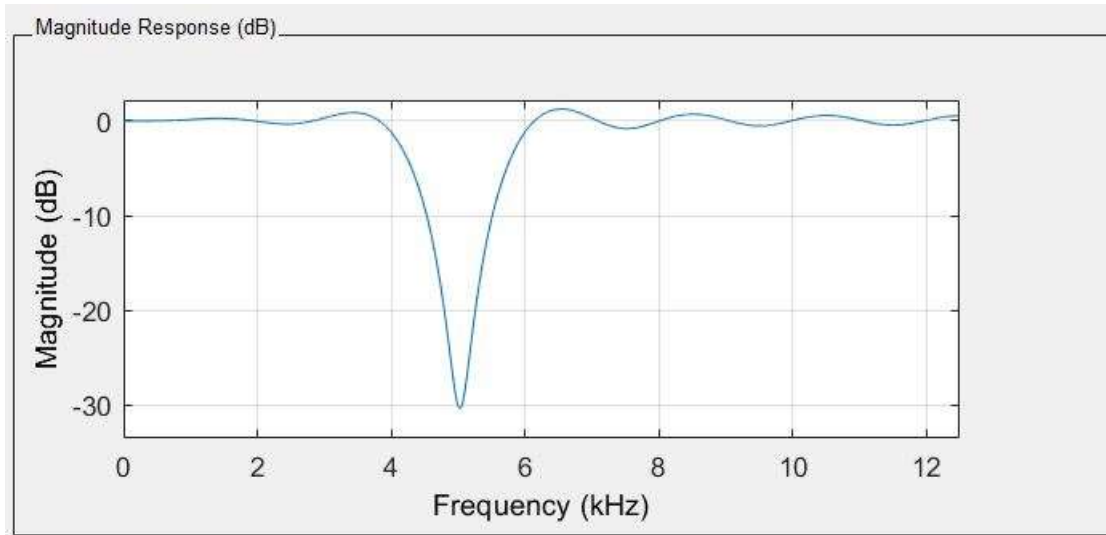


Figure 39: 25-Coefficient Filter Magnitude Response

The impulse response of the filter visually demonstrates one of the main focuses of the new filter design. This focus being an increase in filter coefficient count. Shown in *Figure 40* is the impulse response of the 24-tap band-stop filter.

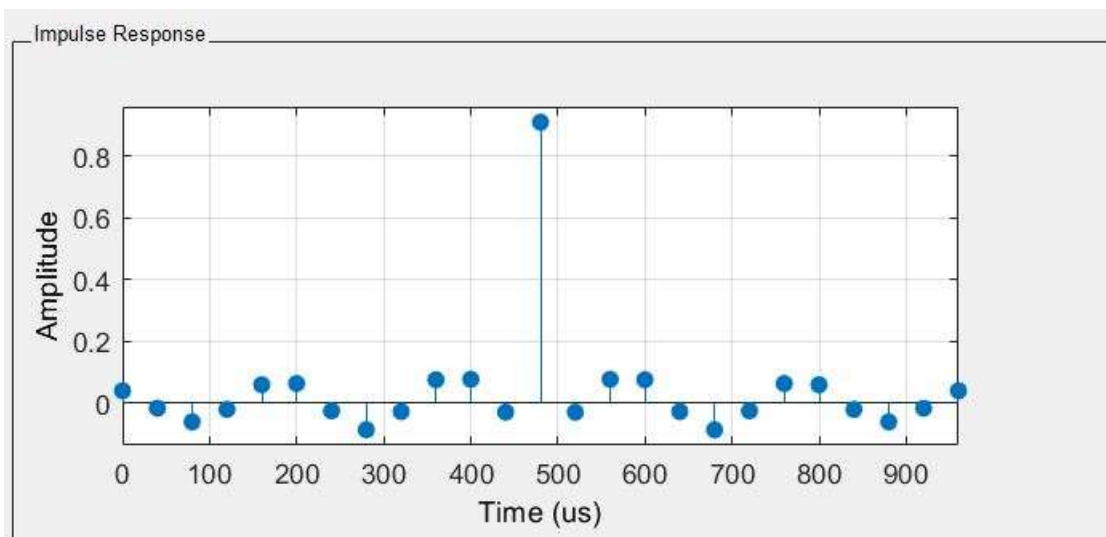


Figure 40: 25-Coefficient Filter Impulse Response

From here, the coefficient values retrieved from MATLAB Filter Designer were transformed into binary numbers. These numbers were then stored in VHDL code in an array to create the digital filter. As before, this was done using fixed-point notation as well as two's complement. The entire conversion process is given in *Table 2*.

Table 2: Fixed-Point Notation Calculation for Higher Order Filter

Filter Coefficients	Coefficient * 2^N	Rounded	Signed 12-bit Coefficients
0.03951	20.22851344	20	000000010100
-0.01705	-8.728061243	-9	11111110111
-0.06135	-31.41106098	-31	11111100001
-0.02080	-10.64717768	-11	11111110101
0.05898	30.19980186	30	00000011110
0.06318	32.350596	32	000000100000
-0.02558	-13.0985349	-13	11111110011
-0.08690	-44.49047705	-44	111111010100
-0.02792	-14.29544491	-14	11111110010
0.07532	38.56580036	39	000000100111
0.07694	39.393976	39	000000100111
-0.02976	-15.23912402	-15	11111110001
0.91082	466.3423862	466	000111010010
-0.02976	-15.23912402	-15	11111110001
0.07694	39.393976	39	000000100111
0.07532	38.56580036	39	000000100111
-0.02792	-14.29544491	-14	11111110010
-0.08690	-44.49047705	-44	111111010100
-0.02558	-13.0985349	-13	11111110011
0.06318	32.350596	32	000000100000
0.05898	30.19980186	30	00000011110
-0.02080	-10.64717768	-11	11111110101
-0.06135	-31.41106098	-31	11111100001
-0.01705	-8.728061243	-9	11111110111
0.03951	20.22851344	20	000000010100

3.7 Quartus Prime Analysis

3.7.1 Hardware Analysis

With both a 13-coefficient and a 25-coefficient band-stop FIR filter, each written using a state machine format, the hardware can be analyzed to see which design is superior. This was done through the comparison of register and ALM usage as well as an inspection of the correlation between hardware usage and filter order. The necessary information for this analysis was retrieved using the compilation report in Quartus Prime. The results, from Quartus Prime, for the 13-coefficient filter in non-state machine format is given in *Figure 41*.

Flow Status	Successful - Mon Feb 08 12:06:19 2021
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	FIR_filter
Top-level Entity Name	FIR_filter
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	156 / 41,910 (< 1 %)
Total registers	374
Total pins	38 / 499 (8 %)
Total virtual pins	0
Total block memory bits	0 / 5,662,720 (0 %)
Total DSP Blocks	9 / 112 (8 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 15 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 41: 13-Coefficient Non-State Machine Hardware Usage

In *Figure 41*, the Logic utilization and Total registers sections are the items of interest. The 13-coefficient non-state machine FIR filter utilizes 374 registers while only using 156 ALMs.

Next, the 25-coefficient non-state machine formatted filter was run through Quartus Prime. The output from the analysis is given in *Figure 42*.

Flow Status	Successful - Mon Feb 15 12:40:26 2021
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	FIR_filter
Top-level Entity Name	FIR_filter
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	630 / 41,910 (2 %)
Total registers	734
Total pins	38 / 499 (8 %)
Total virtual pins	0
Total block memory bits	0 / 5,662,720 (0 %)
Total DSP Blocks	11 / 112 (10 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 15 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 42: 25-Coefficient Non-State Machine Hardware Usage

The 25-coefficient filter in non-state machine format used 630 ALMs and a total of 734 registers. From the 13-coefficient filter, there was over a 300% increase in ALM usage while just under a 100% increase in register usage.

The 13-coefficient state machine formatted filter was the next component that was analyzed using Quartus Prime. The ALM and register values are compiled and presented in *Figure 43*.

Flow Status	Successful - Mon Feb 08 12:22:58 2021
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	FIR_filter
Top-level Entity Name	FIR_filter
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	336 / 41,910 (< 1 %)
Total registers	311
Total pins	38 / 499 (8 %)
Total virtual pins	0
Total block memory bits	0 / 5,662,720 (0 %)
Total DSP Blocks	9 / 112 (8 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 15 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 43: 13-Coefficient State Machine Hardware Usage

As shown in *Figure 43* there were 336 ALMs used in the 13-coefficient state machine formatted filter along with 311 registers. This illustrates over a 100% increase in ALM usage between the 13-coefficient state machine filter and the 13-coefficient non-state machine filter. *Figure 43* also shows just above a 16% decrease in register usage from the state machine formatted filter compared to the non-state machine formatted filter.

The final analysis was run on the 25-coefficient state machine formatted filter.

The results are given in *Figure 44*.

Flow Status	Successful - Mon Feb 15 13:19:42 2021
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	FIR_filter
Top-level Entity Name	FIR_filter
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	672 / 41,910 (2 %)
Total registers	608
Total pins	38 / 499 (8 %)
Total virtual pins	0
Total block memory bits	0 / 5,662,720 (0 %)
Total DSP Blocks	11 / 112 (10 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 15 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 44: 25-Coefficient State Machine Hardware Usage

There were 672 ALMs utilized in the filter's creation while there were only 608 registers used. Compared to the 25-coefficient non-state machine filter, there was just above a 6% increase in ALM usage and a 17% decrease in register usage.

Data visualization is a key component of any type of data analysis. The resulting ALM utilization, register utilization, and total hardware amounts were collected and visually represented using Python. The graphical outcome is displayed in *Figure 45*.

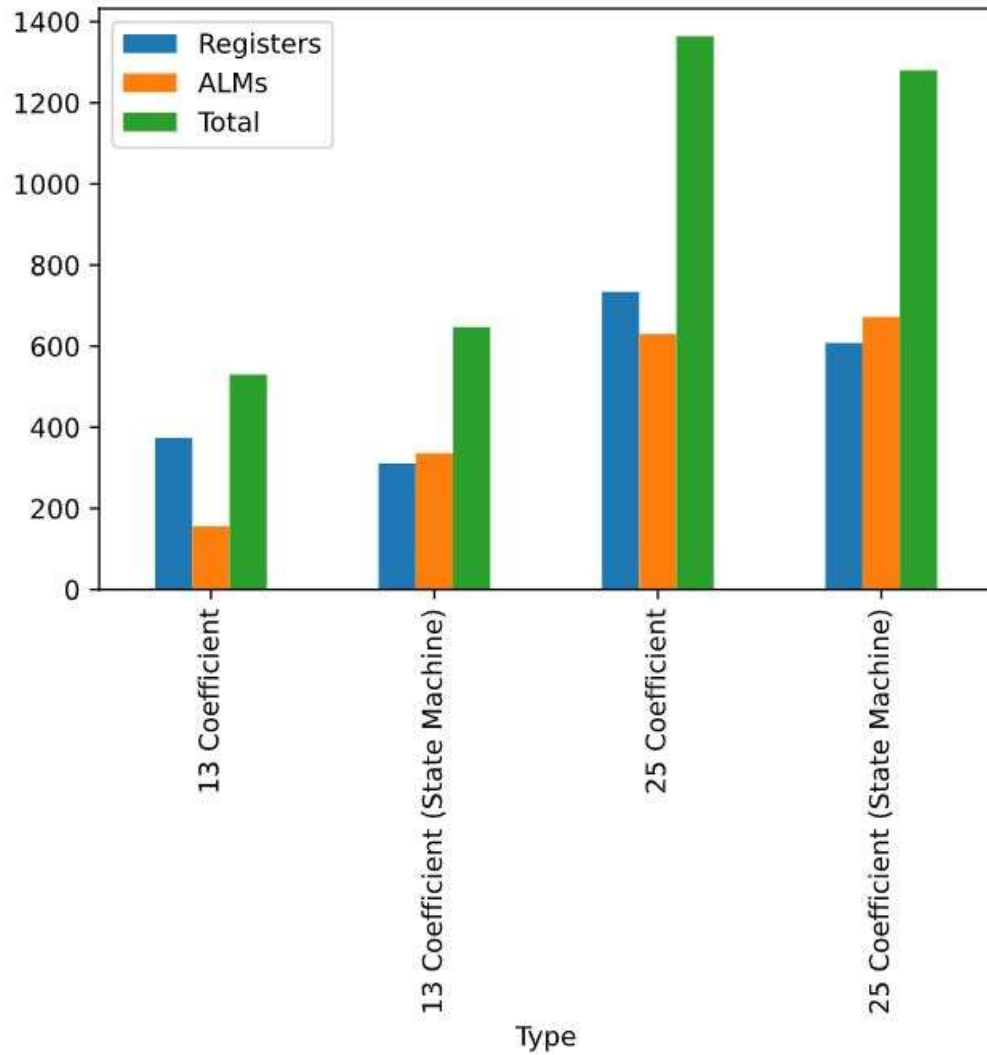


Figure 45: Visual Hardware Summary

The results showed an increase in ALM usage with a decrease in register usage in the state machine formatted code with respect to the non-state machine formatted code. The 13-coefficient state machine filter had more overall hardware used than that of the 13-coefficient non-state machine filter. This outcome was reversed for the 25-coefficient filters. Both 13-coefficient filters used less than 50% of the total amount of hardware when compared to their 25-coefficient counterparts. The 13-coefficient filter used 9 DSP blocks while the 25-coefficient filter used 11 DSP blocks.

3.7.2 Slack Analysis

For any FPGA design to be functional on a board the design must have a positive setup slack. Setup slack is the difference between the amount of time that is required for data to reach the next register and how long the data actually takes to reach the next register. The path of interest when analyzing slack is the worst-case path out of all the paths available on the FPGA. This is because if this path is a positive value than in the worst-case scenario there will still be enough time for the data's arrival. Quartus Prime performs slack analysis and allows the user to alter the analysis settings, in this case clock speed. For the 13-coefficient non-state machine FIR filter a test clock of 50MHz was used giving the results in *Figure 46*.

	Clock	Setup	Hold	Recovery	Removal	Minimum Pulse Width
1	▼ Worst-case Slack	8.242	0.072	N/A	N/A	9.322
1	clk	8.242	0.072	N/A	N/A	9.322
2	▼ Design-wide TNS	0.0	0.0	0.0	0.0	0.0
1	clk	0.000	0.000	N/A	N/A	0.000

Figure 46: 50MHz Non-State Machine Slack Analysis

In *Figure 46*, there is a setup slack of 8.242, this means that in the worst-case scenario data will have enough time to travel from one register to another before the next system clock period.

The 13-coefficient state machine FIR filter was then put under analysis with a 50MHz simulation clock. The results are shown in *Figure 47*.

	Clock	Setup	Hold	Recovery	Removal	Minimum Pulse Width
1	▼ Worst-case Slack	12.220	0.065	N/A	N/A	9.324
1	clk	12.220	0.065	N/A	N/A	9.324
2	▼ Design-wide TNS	0.0	0.0	0.0	0.0	0.0
1	clk	0.000	0.000	N/A	N/A	0.000

Figure 47: 50MHz State Machine Slack Analysis

The worst-case setup slack for the 13-coefficient state machine filter was 12.22. This shows that the state machine format has a better slack performance than the non-state machine formatted code. While both are implementable at a 50MHz system clock the state machine format requires less time for data to arrive between registers.

Next, the 13-coefficient non-state machine formatted code was tested using a 100MHz clock. Figure 48 shows the outcome.

	Clock	Setup	Hold	Recovery	Removal	Minimum Pulse Width
1	▼ Worst-case Slack	-0.435	0.082	N/A	N/A	4.322
1	clk	-0.435	0.082	N/A	N/A	4.322
2	▼ Design-wide TNS	-2.591	0.0	0.0	0.0	0.0
1	clk	-2.591	0.000	N/A	N/A	0.000

Figure 48: 100MHz Non-State Machine Slack Analysis

When run at a 100MHz clock, the non-state machine formatted code has a worst-case slack of -0.435. Since this value is negative, the design is not implementable on a board because somewhere in the design there is not enough time for information to be passed from one register to the next during a single system clock period.

The 13-coefficient state machine formatted code was then run using a 100MHz system clock. The results are shown in Figure 49.

	Clock	Setup	Hold	Recovery	Removal	Minimum Pulse Width
1	▼ Worst-case Slack	2.717	0.083	N/A	N/A	4.321
1	clk	2.717	0.083	N/A	N/A	4.321
2	▼ Design-wide TNS	0.0	0.0	0.0	0.0	0.0
1	clk	0.000	0.000	N/A	N/A	0.000

Figure 49: 100MHz State Machine Slack Analysis

The state machine formatted code had a worst-case slack of 2.717. This shows that the

slack will not be an issue when it comes to implementing the design onto a board.

While both designs successfully ran when a 50MHz system clock was applied, only the state machine design successfully ran when a 100MHz system clock was applied. State machine formatting allows the user to break up large amounts of data, that need to be processed, into smaller more manageable amounts. By breaking up the amount of data that was processed in a single clock period, the system needs to process less data in the same amount of time, increasing positive slack and decreasing negative slack. In this case, this positive slack increase and negative slack decrease allows the design to be implemented with a high frequency system clock.

Chapter 4 Conclusion

4.1 Conclusion

Using MATLAB Filter Designer, a 5KHz low-tap band-stop FIR filter with a sampling frequency of 25KHz was successfully designed. While the performance of the filter was negatively affected by the low number of coefficients, the filter was successfully programmed and simulated using fixed-point numbers and data pipelining. The filter was then successfully implemented onto a DE-10 Standard board with the experimental values correlating well with the simulation values retrieved using ModelSim.

After the development of the filter, a higher order filter was created that focused about the same 5KHz frequency using the previously selected 25KHz sampling frequency. Both the low order and higher order filters were then programmed using a state machine format. The results yielded 4 unique band-stop FIR filters with different pros and cons for each. The higher order filters had better performance but utilized more hardware than the lower order filters. The state machine formatted code used more ALMs compared to the non-state machine formatted code which used more registers. There were varying results for total amount of hardware usage between the state machine formatted code and the non-state machine formatted code. The state machine formatted code had better results from the slack analysis but was more tedious to program and less organized than that of the non-state machine formatted code.

As the use of FPGAs in industry and research expands, the basic designs act as a building block for more complex solutions. Through continual improvement of

fundamental designs, the effects of this improvement will have a larger and larger impact on the FPGA field.

Appendix

Label	Code	Language
A.1	13-Coefficient Band-Stop FIR Filter Non-State Machine Format	VHDL
A.2	13-Coefficient Band-Stop FIR Filter State Machine Format	VHDL
A.3	25-Coefficient Band-Stop FIR Filter Non-State Machine Format	VHDL
A.4	25-Coefficient Band-Stop FIR Filter State Machine Format	VHDL
A.5	Filter Testbench	VHDL
A.6	Test Data Development	MATLAB
A.7	Data Analysis	Python

A.1

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity FIR_filter is
    port(clk : in std_logic;
         reset : in std_logic;
         original_signal : in std_logic_vector(11 downto 0);
         filtered_signal : out std_logic_vector(23 downto 0));
end FIR_filter;

architecture behavior of FIR_filter is
    type coeffs is array (0 to 12) of signed(11 downto 0);
    type previous_data_storage is array (0 to 12) of signed(11 downto 0);
    type product_storage is array (0 to 12) of signed(23 downto 0);

    signal coefficient : coeffs := (0 => "111111101111", 1 => "111110111000", 2 =>
"111111100101", 3 => "000001010011", 4 => "000001011100", 5 => "111111011011",

```



```

6 => "000111010101", 7 => "111111011011", 8 => "000001011100", 9 =>
"000001010011", 10 => "11111100101", 11 => "111110111000", 12 =>
"11111101111");
    signal signal_memory : previous_data_storage;
    signal prod : product_storage;
    signal frequency_sample : std_logic_vector(11 downto 0);
    signal sample_clk : std_logic; --sampling clock
    signal sample_clk_delay : std_logic; --necessary to detect rising edge of non
clock
    signal sample_clk_edge : std_logic; --used to signal rising edge in sample clock
    signal sum : signed(23 downto 0);

begin
process(clk, reset)
begin
if reset = '1' then --resets code
    frequency_sample <= (others => '0');
    sample_clk <= '0';
    sample_clk_delay <= '0';
    sample_clk_edge <= '0';
    sum <= (others => '0');
    filtered_signal <= (others => '0'); --filtered output signal is cleared
    signal_memory <= (others => ("000000000000"));
    prod <= (others => ("000000000000000000000000"));
elsif rising_edge(clk) then
    frequency_sample <= frequency_sample + 1;

    if(frequency_sample = "001111101000" ) then
        frequency_sample <= "000000000000"; --value reset
        sample_clk_delay <= sample_clk;
        sample_clk <= not sample_clk; --toggle sampling clk
        sample_clk_edge <= sample_clk and (not sample_clk_delay);
        if sample_clk_edge = '1' then
            signal_memory <= signed(original_signal) &
signal_memory(0 TO 11);
            sum <= (others => '0'); --ensure sum is cleared

            for i in 0 to 12 loop
                prod(i) <= signal_memory(i) * coefficient(i);
            end loop;

            sum <=
prod(0)+prod(1)+prod(2)+prod(3)+prod(4)+prod(5)+prod(6)+prod(7)+prod(8)+prod(9)+
prod(10)+prod(11)+prod(12);

            if sum(23)='1' then

```

```

                filtered_signal <= "000000000000000000000000";
            else
                filtered_signal <= std_logic_vector(sum srl 9);
            end if;
        end if;
    end if;
end process;
end behavior;

```

A.2

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity FIR_filter is
    port(clk : in std_logic;
         reset : in std_logic;
         original_signal : in std_logic_vector(11 downto 0);
         filtered_signal : out std_logic_vector(23 downto 0));
end FIR_filter;

architecture behavior of FIR_filter is
    type coeffs is array (0 to 12) of signed(11 downto 0);           --coefficient
    storage
    type previous_data_storage is array (0 to 12) of signed(11 downto 0);   --stores
    previous data
    type product_storage is array (0 to 12) of signed(23 downto 0);

    signal coefficient : coeffs := (0 => "11111101111", 1 => "111110111000", 2 =>
    "11111100101", 3 => "000001010011", 4 => "000001011100", 5 => "111111011011",
    6 => "000111010101", 7 => "111111011011", 8 => "000001011100", 9 =>
    "000001010011", 10 => "11111100101", 11 => "111110111000", 12 =>
    "11111101111");
    signal signal_memory : previous_data_storage;
    signal prod : product_storage;
    signal frequency_sample : std_logic_vector(11 downto 0);
    signal sample_clk : std_logic;           --sampling clock
    signal sample_clk_delay : std_logic;
    signal sample_clk_edge : std_logic;
    signal sum : signed(23 downto 0);
    signal start : std_logic;
    signal state : std_logic_vector(4 downto 0);

```

```

begin
process(clk, reset)
begin
if reset = '1' then
--resets code
frequency_sample <= (others => '0');
sample_clk <= '0';
sample_clk_delay <= '0';
sample_clk_edge <= '0';
start <= '0';
sum <= (others => '0');
filtered_signal <= (others => '0'); --filtered output signal is cleared
signal_memory <= (others => ("000000000000"));
prod <= (others => ("000000000000000000000000"));
state <= "00000";
elsif rising_edge(clk) then
frequency_sample <= frequency_sample + 1;

if(frequency_sample = "001111101000" ) then
frequency_sample <= "000000000000"; --value reset
sample_clk_delay <= sample_clk;
sample_clk <= not sample_clk; --toggle sampling clk
sample_clk_edge <= sample_clk and (not sample_clk_delay);
if sample_clk_edge = '1' then
start <= '1';
state <= "00001";
end if;
elsif start = '1' then
if state = "00001" then
signal_memory <= signed(original_signal) &
signal_memory(0 TO 11);
state <= "00010";
elsif state = "00010" then
sum <= (others => '0');
state <= "00011";
elsif state = "00011" then
prod(0) <= signal_memory(0) * coefficient(0);
state <= "00100";
elsif state = "00100" then
prod(1) <= signal_memory(1) * coefficient(1);
state <= "00101";
elsif state = "00101" then
prod(2) <= signal_memory(2) * coefficient(2);
state <= "00110";
elsif state = "00110" then
prod(3) <= signal_memory(3) * coefficient(3);
state <= "00111";

```

```

elsif state = "00111" then
    prod(4) <= signal_memory(4) * coefficient(4);
    state <= "01000";
elsif state = "01000" then
    prod(5) <= signal_memory(5) * coefficient(5);
    state <= "01001";
elsif state = "01001" then
    prod(6) <= signal_memory(6) * coefficient(6);
    state <= "01010";
elsif state = "01010" then
    prod(7) <= signal_memory(7) * coefficient(7);
    state <= "01011";
elsif state = "01011" then
    prod(8) <= signal_memory(8) * coefficient(8);
    state <= "01100";
elsif state = "01100" then
    prod(9) <= signal_memory(9) * coefficient(9);
    state <= "01101";
elsif state = "01101" then
    prod(10) <= signal_memory(10) * coefficient(10);
    state <= "01110";
elsif state = "01110" then
    prod(11) <= signal_memory(11) * coefficient(11);
    state <= "01111";
elsif state = "01111" then
    prod(12) <= signal_memory(12) * coefficient(12);
    state <= "10000";
elsif state = "10000" then
    sum <= prod(0);
    state <= "10001";
elsif state = "10001" then
    sum <= sum + prod(1);
    state <= "10010";
elsif state = "10010" then
    sum <= sum + prod(2);
    state <= "10011";
elsif state = "10011" then
    sum <= sum + prod(3);
    state <= "10100";
elsif state = "10100" then
    sum <= sum + prod(4);
    state <= "10101";
elsif state = "10101" then
    sum <= sum + prod(5);
    state <= "10110";
elsif state = "10110" then

```

```

        sum <= sum + prod(6);
        state <= "10111";
    elsif state = "10111" then
        sum <= sum + prod(7);
        state <= "11000";
    elsif state = "11000" then
        sum <= sum + prod(8);
        state <= "11001";
    elsif state = "11001" then
        sum <= sum + prod(9);
        state <= "11010";
    elsif state = "11010" then
        sum <= sum + prod(10);
        state <= "11011";
    elsif state = "11011" then
        sum <= sum + prod(11);
        state <= "11100";
    elsif state = "11100" then
        sum <= sum + prod(12);
        state <= "11101";
    elsif state = "11101" then
        if sum(23)='1' then
            filtered_signal <= "000000000000000000000000";
        else
            filtered_signal <= std_logic_vector(sum srl 9);
        end if;
        state <= "11111";
    else
        state <= "00000";
        start <= '0';
    end if;
end if;
end if;
end process;
end behavior;

```

A.3

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

```

```

entity FIR_filter is
    port(clk : in std_logic;
         reset : in std_logic;

```



```

        if sample_clk_edge = '1' then
            signal_memory <= signed(original_signal) &
signal_memory(0 TO 23);
            sum <= (others => '0'); --ensure sum is cleared

            for i in 0 to 24 loop
                prod(i) <= signal_memory(i) * coefficient(i);
            end loop;

            sum <=
prod(0)+prod(1)+prod(2)+prod(3)+prod(4)+prod(5)+prod(6)+prod(7)+prod(8)+prod(9)+
prod(10)+prod(11)+prod(12)+prod(13)+prod(14)+prod(15)+prod(16)+prod(17)+prod(18
)+prod(19)+prod(20)+prod(21)+prod(22)+prod(23)+prod(24);

            if sum(23)='1' then
                filtered_signal <= "000000000000000000000000";
            else
                filtered_signal <= std_logic_vector(sum srl 9);
            end if;
        end if;
    end if;
end if;
end process;
end behavior;

```

A.4

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity FIR_filter is
    port(clk : in std_logic;
         reset : in std_logic;
         original_signal : in std_logic_vector(11 downto 0);
         filtered_signal : out std_logic_vector(23 downto 0));
end FIR_filter;

architecture behavior of FIR_filter is
    type coeffs is array (0 to 24) of signed(11 downto 0);
    type previous_data_storage is array (0 to 24) of signed(11 downto 0);
    type product_storage is array (0 to 24) of signed(23 downto 0);

    signal coefficient : coeffs := (0 => "000000010100", 1 => "111111110111", 2 =>
"1111111100001", 3 => "111111110101", 4 => "000000011110", 5 => "000000100000",

```

```

6 => "111111110011", 7 => "111111010100", 8 => "111111110010", 9 =>
"000000100111", 10 => "000000100111", 11 => "111111110001", 12 =>
"000111010010", 13 => "111111110001", 14 => "000000100111", 15 =>
"000000100111", 16 => "111111110010", 17 => "111111010100", 18 =>
"111111110011", 19 => "000000100000", 20 => "000000011110", 21 =>
"111111110101", 22 => "111111100001", 23 => "111111110111", 24 =>
"000000010100");
    signal signal_memory : previous_data_storage;           --stores ADC values
for use in addition and multiplication
    signal prod : product_storage;                         --stores products from filter
arithmetic
    signal frequency_sample : std_logic_vector(11 downto 0);--used to adjust rate
that ADC sample is grabbed
    signal sample_clk : std_logic;                         --sampling clock
    signal sample_clk_delay : std_logic;                  --necessary to detect rising
edge of non clock
    signal sample_clk_edge : std_logic;                   --used to signal rising edge in
sample clock
    signal sum : signed(23 downto 0);
    signal start : std_logic;
    signal state : std_logic_vector(5 downto 0);

begin
process(clk, reset)
begin
if reset = '1' then                                     --resets code
    frequency_sample <= (others => '0');
    sample_clk <= '0';
    sample_clk_delay <= '0';
    sample_clk_edge <= '0';
    start <= '0';
    sum <= (others => '0');
    filtered_signal <= (others => '0');    --filtered output signal is cleared
    signal_memory <= (others => ("000000000000"));
    prod <= (others => ("000000000000000000000000"));
    state <= "000000";
elsif rising_edge(clk) then
    frequency_sample <= frequency_sample + 1;

    if(frequency_sample = "001111101000" ) then
        frequency_sample <= "000000000000";    --value reset
        sample_clk_delay <= sample_clk;--sample clk delayed clk value
        sample_clk <= not sample_clk;           --toggle sampling clk
        sample_clk_edge <= sample_clk and (not sample_clk_delay);
        if sample_clk_edge = '1' then
            start <= '1';

```



```

        state <= "000001";
    end if;
    elsif start = '1' then
        if state = "000001" then
            signal_memory <= signed(original_signal) &
signal_memory(0 TO 23);
            state <= "000010";
        elsif state = "000010" then
            sum <= (others => '0');
            state <= "000011";
        elsif state = "000011" then
            prod(0) <= signal_memory(0) * coefficient(0);
            state <= "000100";
        elsif state = "000100" then
            prod(1) <= signal_memory(1) * coefficient(1);
            state <= "000101";
        elsif state = "000101" then
            prod(2) <= signal_memory(2) * coefficient(2);
            state <= "000110";
        elsif state = "000110" then
            prod(3) <= signal_memory(3) * coefficient(3);
            state <= "000111";
        elsif state = "000111" then
            prod(4) <= signal_memory(4) * coefficient(4);
            state <= "001000";
        elsif state = "001000" then
            prod(5) <= signal_memory(5) * coefficient(5);
            state <= "001001";
        elsif state = "001001" then
            prod(6) <= signal_memory(6) * coefficient(6);
            state <= "001010";
        elsif state = "001010" then
            prod(7) <= signal_memory(7) * coefficient(7);
            state <= "001011";
        elsif state = "001011" then
            prod(8) <= signal_memory(8) * coefficient(8);
            state <= "001100";
        elsif state = "001100" then
            prod(9) <= signal_memory(9) * coefficient(9);
            state <= "001101";
        elsif state = "001101" then
            prod(10) <= signal_memory(10) * coefficient(10);
            state <= "001110";
        elsif state = "001110" then
            prod(11) <= signal_memory(11) * coefficient(11);
            state <= "001111";

```

```

elsif state = "001111" then
    prod(12) <= signal_memory(12) * coefficient(12);
    state <= "010000";
elsif state = "010000" then
    prod(13) <= signal_memory(13) * coefficient(13);
    state <= "010001";
elsif state = "010001" then
    prod(14) <= signal_memory(14) * coefficient(14);
    state <= "010010";
elsif state = "010010" then
    prod(15) <= signal_memory(15) * coefficient(15);
    state <= "010011";
elsif state = "010011" then
    prod(16) <= signal_memory(16) * coefficient(16);
    state <= "010100";
elsif state = "010100" then
    prod(17) <= signal_memory(17) * coefficient(17);
    state <= "010101";
elsif state = "010101" then
    prod(18) <= signal_memory(18) * coefficient(18);
    state <= "010110";
elsif state = "010110" then
    prod(19) <= signal_memory(19) * coefficient(19);
    state <= "010111";
elsif state = "010111" then
    prod(20) <= signal_memory(20) * coefficient(20);
    state <= "011000";
elsif state = "011000" then
    prod(21) <= signal_memory(21) * coefficient(21);
    state <= "011001";
elsif state = "011001" then
    prod(22) <= signal_memory(22) * coefficient(22);
    state <= "011010";
elsif state = "011010" then
    prod(23) <= signal_memory(23) * coefficient(23);
    state <= "011011";
elsif state = "011011" then
    prod(24) <= signal_memory(24) * coefficient(24);
    state <= "011100";
elsif state = "011100" then
    sum <= prod(0);
    state <= "011101";
elsif state = "011101" then
    sum <= sum + prod(1);
    state <= "011110";
elsif state = "011110" then

```

```

        sum <= sum + prod(2);
        state <= "011111";
    elsif state = "011111" then
        sum <= sum + prod(3);
        state <= "100000";
    elsif state = "100000" then
        sum <= sum + prod(4);
        state <= "100001";
    elsif state = "100001" then
        sum <= sum + prod(5);
        state <= "100010";
    elsif state = "100010" then
        sum <= sum + prod(6);
        state <= "100011";
    elsif state = "100011" then
        sum <= sum + prod(7);
        state <= "100100";
    elsif state = "100100" then
        sum <= sum + prod(8);
        state <= "100101";
    elsif state = "100101" then
        sum <= sum + prod(9);
        state <= "100110";
    elsif state = "100110" then
        sum <= sum + prod(10);
        state <= "100111";
    elsif state = "100111" then
        sum <= sum + prod(11);
        state <= "101000";
    elsif state = "101000" then
        sum <= sum + prod(12);
        state <= "101001";
    elsif state = "101001" then
        sum <= sum + prod(13);
        state <= "101010";
    elsif state = "101010" then
        sum <= sum + prod(14);
        state <= "101011";
    elsif state = "101011" then
        sum <= sum + prod(15);
        state <= "101100";
    elsif state = "101100" then
        sum <= sum + prod(16);
        state <= "101101";
    elsif state = "101101" then
        sum <= sum + prod(17);

```

```

        state <= "101110";
    elsif state = "101110" then
        sum <= sum + prod(18);
        state <= "101111";
    elsif state = "101111" then
        sum <= sum + prod(19);
        state <= "110000";
    elsif state = "110000" then
        sum <= sum + prod(20);
        state <= "110001";
    elsif state = "110001" then
        sum <= sum + prod(21);
        state <= "110010";
    elsif state = "110010" then
        sum <= sum + prod(22);
        state <= "110011";
    elsif state = "110011" then
        sum <= sum + prod(23);
        state <= "110100";
    elsif state = "110100" then
        sum <= sum + prod(24);
        state <= "110101";
    elsif state = "110101" then
        if sum(23)='1' then
            filtered_signal <= "00000000000000000000000000000000";
        else
            filtered_signal <= std_logic_vector(sum srl 9);
        end if;
        state <= "111111";
    else
        state <= "000000";
        start <= '0';
    end if;
end if;
end if;
end process;
end behavior;

```

A.5

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;
use STD.TEXTIO.all;

```

```

entity FIR_filter_tb is
end FIR_filter_tb;

architecture testbench of FIR_filter_tb is
    signal clk : std_logic := '0';
    signal reset : std_logic; --circuit input for reset
    signal original_signal : std_logic_vector(11 downto 0); --circuit input for original
signal
    signal filtered_signal : std_logic_vector(23 downto 0);
    signal clk_sample : std_logic := '0';
    file test_input : text open READ_MODE is "test_data.txt";

    begin

        circuit_A : entity work.FIR_filter port map(clk => clk, reset => reset,
original_signal => original_signal, filtered_signal => filtered_signal);

        reset <= '1','0' after 0.1 ms;

        process(clk_sample)
            variable line_test_input : line;
            variable input: integer;
            begin
                if rising_edge(clk_sample) then
                    readline(test_input, line_test_input);
                    read(line_test_input,input);
                    original_signal <= std_logic_vector(to_unsigned(input,
12));
                end if;
            end process;
        end testbench;

```

A.6

```

%% Wave creation
clear all;
clc;

fs = 25000;                %samples per second
dt = 1/fs;                %inverse samples per second
stoptime = 5;             %stoptime in seconds
t = (0:dt:stoptime-dt)';
fc = 5600;                %frequency
y = (cos(2*pi*fc*t) + 1) * 200;
plot(y);

```

```
fileID = fopen('test_data.txt','w');    %opens file ; opens in read only but 'w' makes it
write
fprintf(fileID,'%f\n',y);              %writes value then on the next line prints the next line
fclose(fileID);
```

A.7

```
import pandas as pd
import numpy as np
import seaborn
import matplotlib.pyplot as plt
```

```
hardware_count = pd.DataFrame({'Type':['13 Coefficient', '13 Coefficient (State Machine)', '25 Coefficient', '25 Coefficient (State Machine)'], 'Registers':[374, 311, 734, 608], 'ALMs':[156, 336, 630, 672]})
plt.figure()
hardware_count.plot.bar(x='Type')
hardware_count["Total"] = hardware_count.sum(axis=1)
plt.close()
plt.figure()
hardware_count.plot.bar(x='Type')
```

Bibliography

- [1] Admin. (2020, September 14). Difference between IIR and FIR filters: A practical design guide. Retrieved March 17, 2021, from <https://www.advsolned.com/difference-between-iir-and-fir-filters-a-practical-design-guide/>
- [2] Chen Sun, Abhinav Shrivastava, Saurabh Singh, Abhinav Gupta; in Proceedings of the IEEE International Conference on Computer Vision (ICCV), 2017, pg. 843-852
- [3] D. L. N. Hettiarachchi, V. S. P. Davuluru and E. J. Balster, "Integer vs. Floating-Point Processing on Modern FPGA Technology," 2020 10th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, 2020, pp. 0606-0612, doi: 10.1109/CCWC47524.2020.9031118.
- [4] Filter designer. (n.d.). Retrieved March 17, 2021, from <https://www.mathworks.com/help/signal/ug/introduction-to-filter-designer.html>
- [5] FPGA design software - Intel® Quartus® Prime. (n.d.). Retrieved March 17, 2021, from <https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html>
- [6] Fpga fundamentals. (n.d.). Retrieved March 17, 2021, from <https://www.ni.com/en-us/innovations/white-papers/08/fpga-fundamentals.html>
- [7] H. J. Landau, "Sampling, data transmission, and the Nyquist rate," in Proceedings of the IEEE, vol. 55, no. 10, pp. 1701-1706, Oct. 1967, doi: 10.1109/PROC.1967.5962.
- [8] Levesque, Luc. (2014). Nyquist sampling theorem: Understanding the illusion of a spinning wheel captured with a video camera. *Physics Education*. 49. 697. 10.1088/0031-9120/49/6/697.
- [9] Modelsim. (n.d.). Retrieved March 17, 2021, from <https://eda.sw.siemens.com/en-US/ic/modelsim/>
- [10] Morris, K., & Says:, K. (2013, May 28). Timing is everything. Retrieved March 17, 2021, from <https://www.eejournal.com/article/20130528-timing/>
- [11] N. (2015, July 10). What is FIR FILTER? - FIR filters for digital signal processing. Retrieved March 17, 2021, from <https://www.elprocus.com/fir-filter-for-digital-signal-processing/>

- [12] Recursive digital filter design. (n.d.). Retrieved March 17, 2021, from https://www.dsprelated.com/freebooks/filters/Recursive_Digital_Filter_Design.htm
1
- [13] Scheier, R. L. (n.d.). Fueling Decisions and Actions With Data Pipelining.
- [14] Technologies, T. (n.d.). SoC platform - cyclone - DE10-Standard. Retrieved March 17, 2021, from <https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=1081>