EGH449 FIR FILTER

Sophia Politylo (10489045), Nick Meurant (n10485571)

NOVEMBER 1, 2021

QUEENSLAND UNIVERSTY OF TECHNOLOGY

Contents

Overview of Design

RTL Schematic

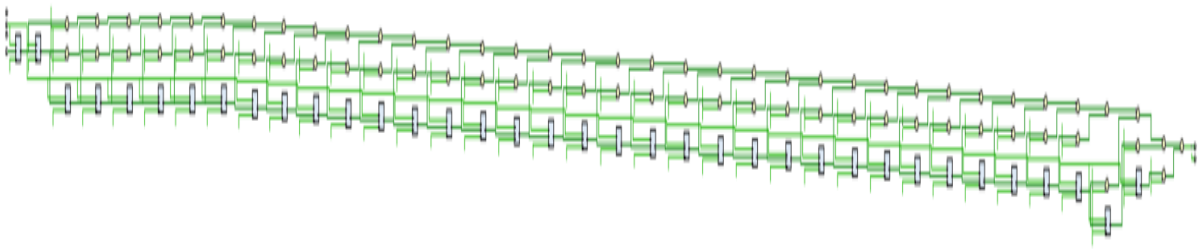


Figure 1 Overview of RTL Schematic

Figure 1 shows a very high-level overview of RTL schematic (Appendix 1-10 for more high-resolution images). The FIR filter is in the transpose form. There are a total of 36 coefficients being used to perform the filtering.

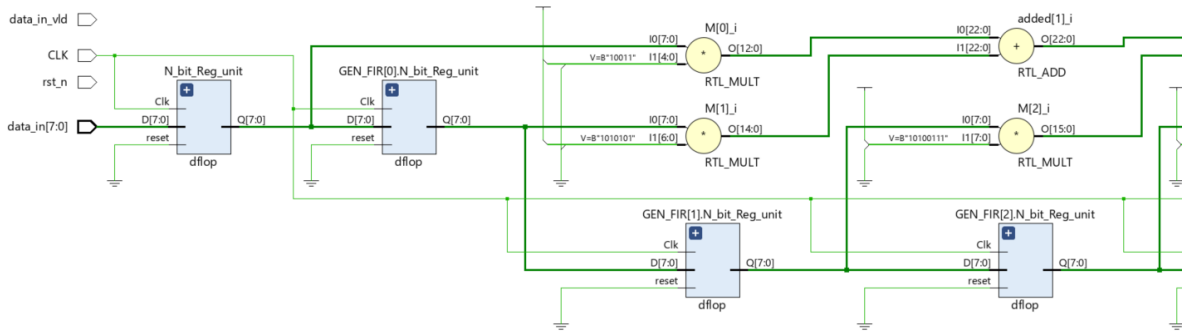


Figure 2 Input Stage and Transpose structure

The filter as mentioned previously is used in a transpose structure. Data_in[7:0] contains the input byte that is then to be filtered through the filter. The design uses a dflop to introduce a delay needed for the input values to multiply with the coefficients in the correct order. The output of the multiplication is then summed together.

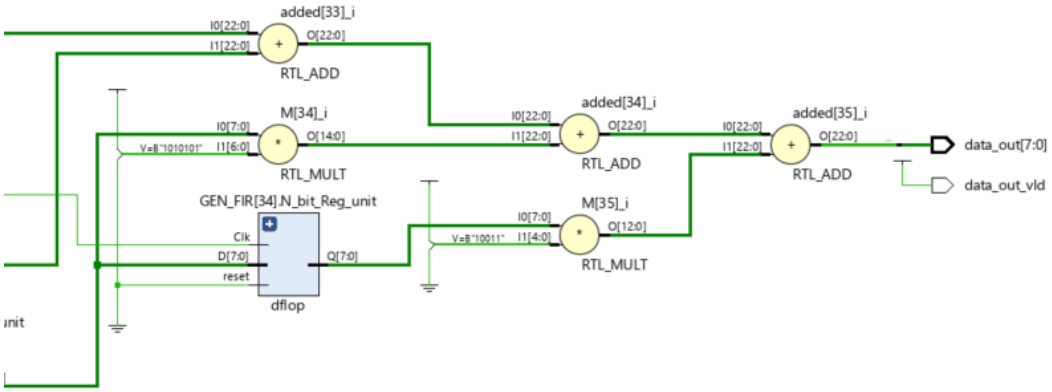


Figure 3 Output Stage of FIR Filter

Figure 3 shows the output stage of the filter. All of the products of the signal and the coefficients are then summed and added to data_out[7:0].

Floor Plan

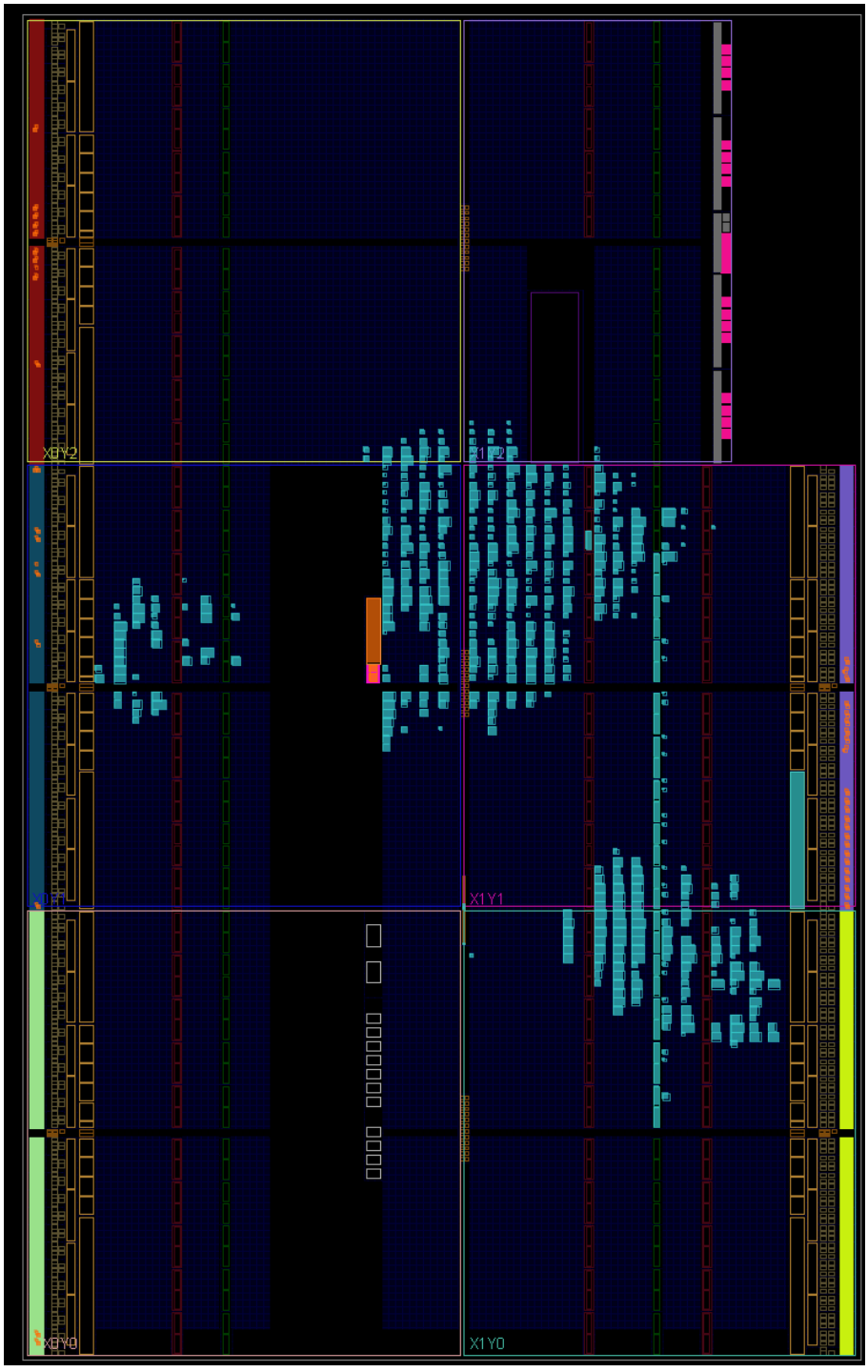


Figure 4 Floor Plan of FPGA (including EGH449 codebase)

Figure 4 refers to the floor plan used by the entire implementation. This includes the EGH449 code base as well as the FIR filter implementation created.

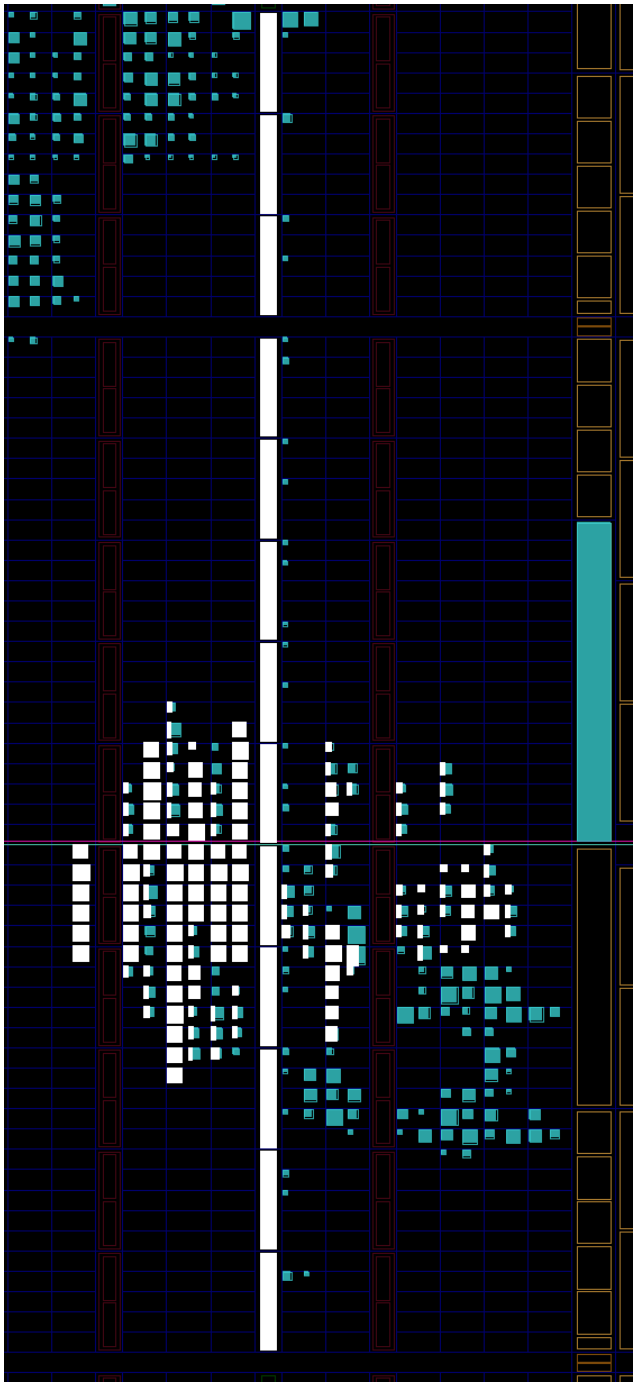


Figure 5 Floor Plan Only FIR Filter

Figure 5 refers to the floor plan of the FIR filter. The white highlighted sections are the blocks used by the FIR filter implementation.

Summary of System Features and Operation

FIR Filter Explanation/Justification

An FIR filter is a digital filter that stand for Finite Impulse Response filter. A FIR filter is finite due to there being no feedback from previous calculations. With the filter only outputting a finite number of outputs for each input value, with the number of responses corresponding to the number of coefficients of the filter. For example, a single input of all zeros with a 1 in the middle will output a total of non-zero values from that 1 value that equals the number of filter coefficients. The basic structure of an FIR filter implemented on an FPGA is shown below in figure 6. From this figure the basic flow of input data can be seen with the x value slowing moving along the coefficients from 0 to 4, with x moving to a new coefficient each clock cycle due to the delay introduced via the flip flops. Due to this design, it means that the final value out of the filter will be delayed by the length of the filter given it will take the number of coefficients – 1, due to the flip flops.

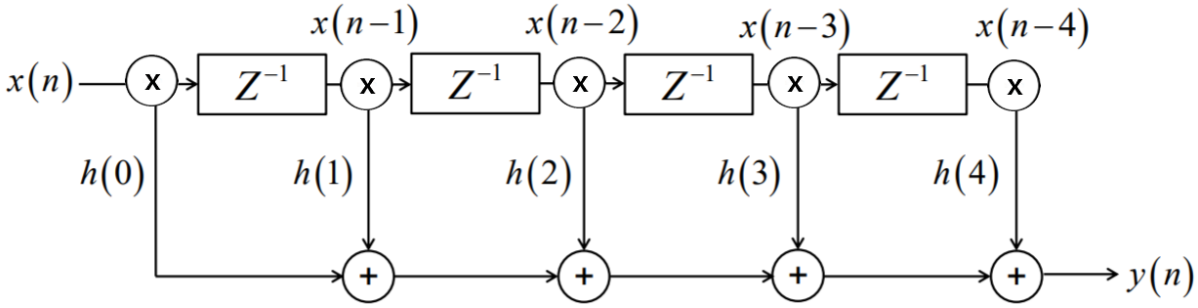


Figure 6 - normal FIR filter Structure

This original FIR design however is lacking in timing due to the number of computations that need to be performed each clock cycle. This is due to the simple fact that each addition in the filter structure needs to wait for the corresponding multiplication to complete before adding it to the current result. Which means that each clock cycle the FPGA needs to perform the equation shown in equation 1. Which with only 5 coefficients should not cause a timing problem but when increased to the needed 36, timing problem will become an issue.

$$[h(0) \times x(n)] + [h(1) \times x(n-1)] + [h(2) \times x(n-2)] + [h(3) \times x(n-3)] + [h(4) \times x(n-4)]$$

Equation 1 - equation preformed each clock cycle using a normal FIR filter structure

Timing becomes an issue in this design since each addition and multiplication is not instant and still requires a set amount of time to complete. Due to this when enough additions and multiplications are chained together the time taken for all these calculations will be added up and if it surpasses the clock cycle; it will cause a negative stack meaning that all computations could not be completed before the next clock cycle, causing a distorted output that is unable to correctly calculate the FIR filter. This was indeed the case when trying this filter type with the timing for the filter being seen in figure 7, where the total negative stack being –416ns.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-52.509 ns	Worst Hold Slack (WHS):	0.022 ns	Worst Pulse Width Slack (WPWS):	3.000 ns
Total Negative Slack (TNS):	-416.014 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	8	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	4239	Total Number of Endpoints:	4239	Total Number of Endpoints:	1663

Timing constraints are not met.

Figure 7 - timing on the FPGA for an implemented normal FIR filter using 36 coefficients

To fix this timing issue the normal FIR filter can be transposed, with the transposed structure being shown in figure 8. This structure aims to remove the number of computations that need to be performed each clock cycle through hiding the additions within the flip flops. Via doing this it ensures that the total number of additions that need to be performed each clock cycle is reduced through adding the time for each flip flop to the total time allowed for each computation. Along with this the transposed design allows for half the number of multiplications since the coefficients mirror each other on both sides of the halfway point and since all coefficients are being multiplied by the x_in, means the FPGA can re uses the first half of the calculations for the second mirrored coefficients value.

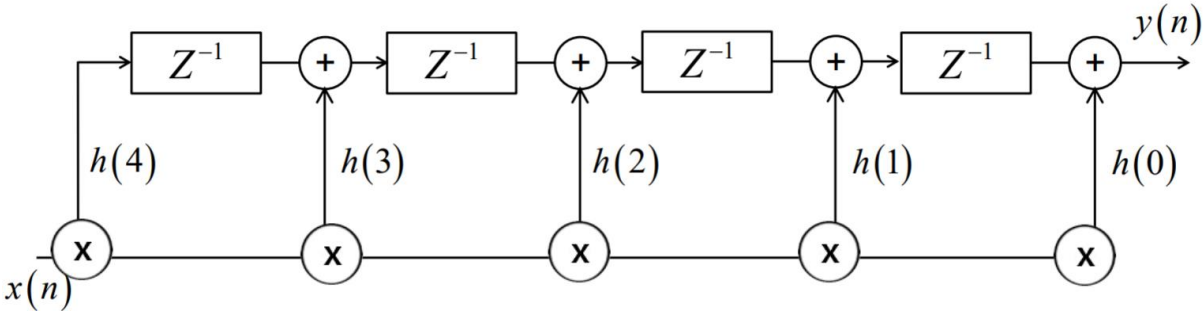


Figure 8 - Transposed FIR Filter Structure

With the transposed structure it means that each clock cycle, the calculation that needs to be completed is just the first coefficient value and the addition of that to the total sum so far, with the equation that needs to be competed looking like the equation in equation 2. This final added value is all the other multiplications up until that flip flop resales, meaning that an extra period is added for each flip flop between the final addition and first flip flop meaning that no longer will all additions need to be calculated at once with the added value slow being passed along the pipeline each clock cycle, meaning that in theory only one addition needs to be completed before the end of the current clock cycle.

$$h(0) \times x + added(n)$$

where:

$$n = number\ of \text{ coefficients}$$

Equation 2 - set of equation preformed each clock cycle at the output of the FIR filter

Using this new transposed model to implement the 36 coefficient FIR filter on an FPGA fixed the timing issues seen previously using the normal structure for a FIR filter, with the transposed filter having no negative timing stack and at worse taking 1.503ns well within the allowed clock cycle. With the output being shown in figure 9.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1.503 ns	Worst Hold Slack (WHS):	0.030 ns	Worst Pulse Width Slack (WPWS):	3.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	5732	Total Number of Endpoints:	5732	Total Number of Endpoints:	1826

All user specified timing constraints are met.

Figure 9 - timing for a transposed filter implemented on an FPGA using 36 coefficients

FIR Filter as an Application of Convolution

The FIR filter is a practical application of the convolution theorem.

$$z(t) = x(t) * y(t) = \int_{-\infty}^{\infty} x(\tau)y(t-\tau)d\tau.$$

$$Z(t) = Convolution\ Output$$

$$x(t) = Input\ Data$$

$$y(t) = FIR\ filter\ Response\ in\ Time\ Domain$$

$$X(\tau) = x(t)\ in\ frequency\ domain$$

$$y(t-\tau) = y(t)\ shifted\ by\ \tau\ in\ frequency\ domain$$

Since both x(t) and y(t) are both discrete signals, the integral between the two signals is just the sum of function at their respective discrete time. When convolving two signals together, one signal (y(t) in this case) is effectively "slid" across x(t). This can be simulated on an FPGA by multiplying y(t) and x(t) together and then adding that value to a large sum at the end of the FPGA.

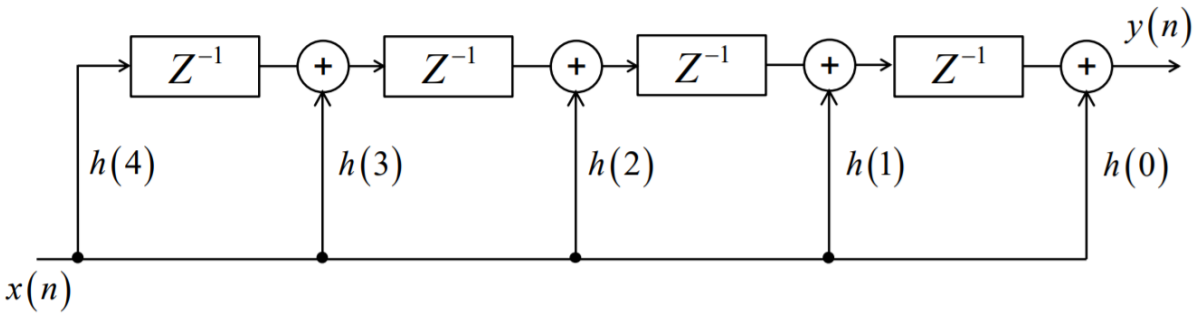


Figure 10 Transpose Structure (Week 9 Lecture)

Figure 10 shows how an FPGA can simulate the convolution of two signals in the time domain. Z^{-1} shows a finite time delay, this is to simulate x(t) being slid across y(t) (mentioned previously). These outputs are then summed up and shown at the output.

Simulation Results

How To Run Simulation

To test out the FIR filter a simulation Vhdl file was created within the project that enables the analysis of the created FIR filter within Vivado. To run the simulation all that needs to be done is to open the demo_all project file with the user clicking on the Run simulation button. From here Vivado will simulate the created FIR filter for a given input. To change the given input the user can use the simulation file fir_test.vhdl. With this file the user can add, change, or remove the lines between 72 – 144, with the user being able to change the 8 bits into the data_in port to any combination of 0 and 1, up to 8 bits. For the timing each new input into the filter is every 20ns. An example input can be seen within figure 11, with the data_in needing to stay with the string of bits being able be changed into any combinations of 0 and 1 along with the wait being able to be changed to a multiple of 20ns if a given input is wanted to be kept for more than one data input cycle.

```
data_in <= "00000000"; wait for 20 ns;
```

Figure 11 - sample code that can be used to create a simulated data_in signal for the FIR filter

The generated simulation wave forms can be seen within figure 12. Within these wave forms there are three main values that are important to the FIR filter. The first one is the clock called CLK, which shows how often each new data input and output should toggle, with it being used to ensure that the output changes each cycle and count the total number of outputs if double ups are found over two clock cycles. The second signal is the data_in which is 8n bits wide. This signal shows the data that was inputted into the FIR filter. The last important signal is the data_out signal which is 8 bits wide again. This signal shows the values that the FIR filter outputs, with the given value being able to be compared to the MATLAB computed version for the input value.

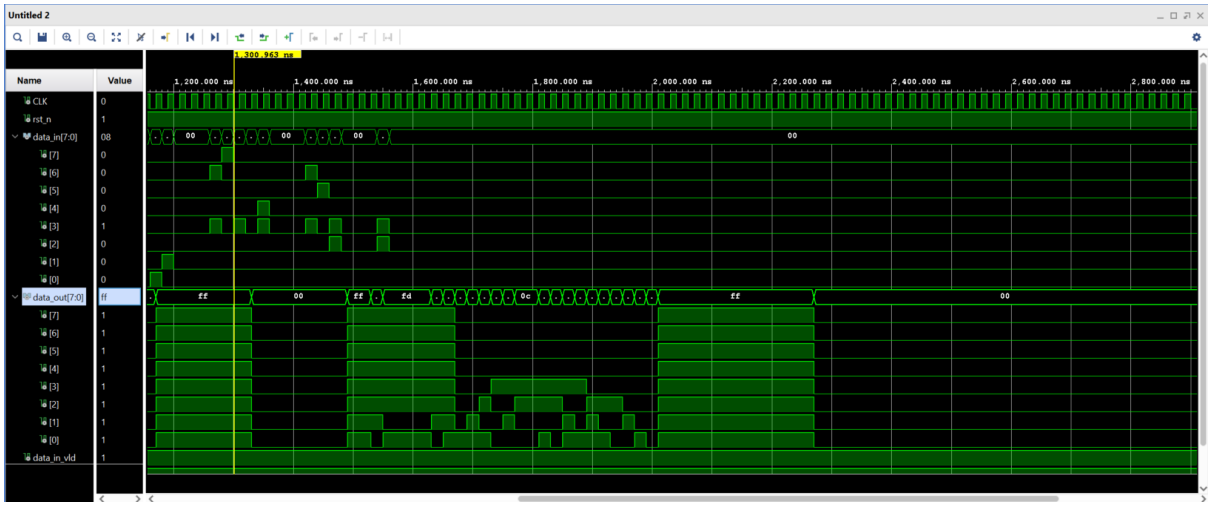


Figure 12 - sample simulation view for the created FIR_test.vhdl simulation file for the FIR filter

Vivado Results Compared to MATLAB

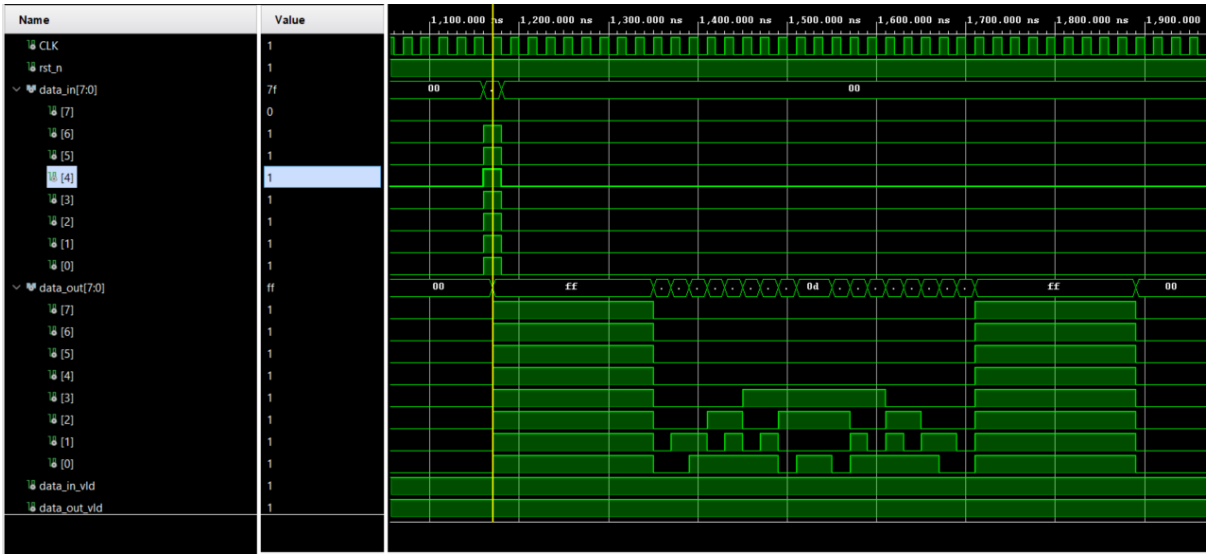


Figure 13 Vivado Simulation Results (input = 127)

Figure 13 shows the output of Vivado with an input impulse of 127. The input and output are both signed integers.

-1	-1	-1	-1	-1	-1	-1	-1	-1	0	2	3	5
	7	9	11	12	13	13	12	11	9	7	5	3
	2	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	

Table 1 MATLAB Output, 127 Input

Table 1 refers to the results generated by MATLAB with an input impulse of 127. As you can see the results match up perfectly with the MATLAB results.

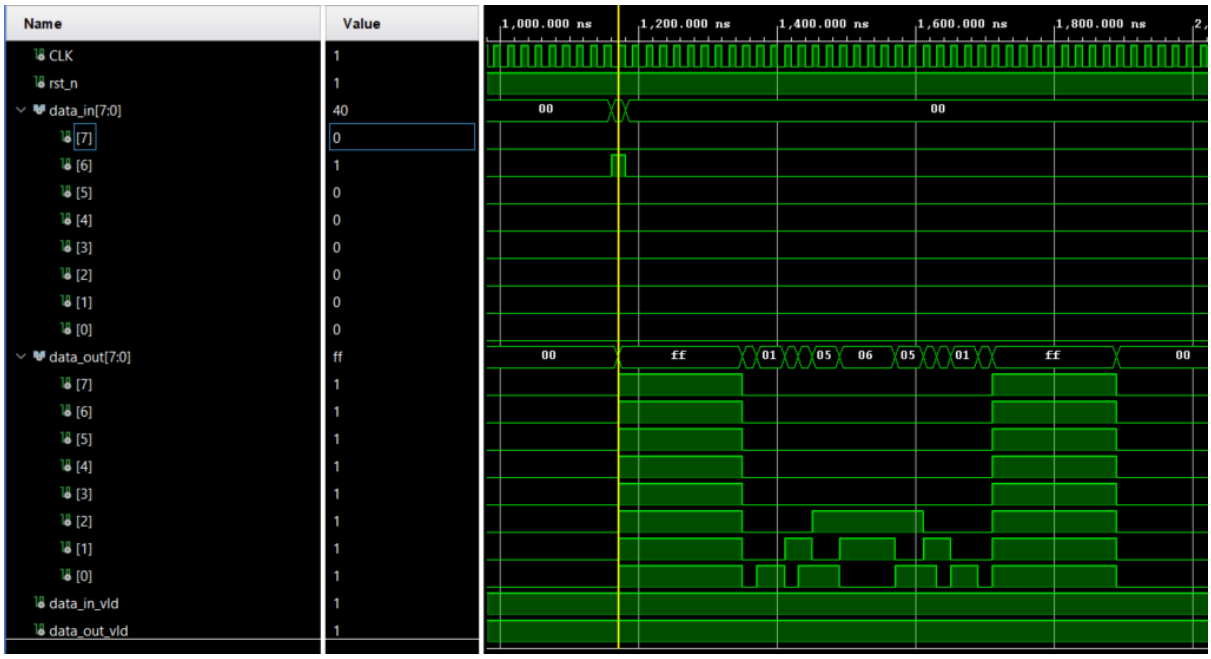


Figure 14 Vivado Simulation Results (input = 64)

Figure 14 refers to the output of the FIR filter with an input impulse of 64 (2^6). The input and output are both signed integers with their values representing whole numbers. As you can see, a new output is calculated every new rising edge of the clock. The filter coefficients consist of 36 different numbers. The values of the coefficients are mirrored upon the centre. This is evident in the fact that the outputs are mirrored about the centre as well.

-1	-1	-1	-1	-1	-1	-1	-1	-1	0	1	1	2
	3	5	5	6	6	6	6	5	5	3	2	1
	1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	

Table 2 MATLAB simulation Results (input = 64)

Table 2 shows the output of the filter coefficients generated in MATLAB. As you can see the results are the same between MATLAB and Vivado. The MATLAB has 36 different outputs, this is because of the number of coefficients is 36. Like the Vivado results, the output is mirrored upon the centre.

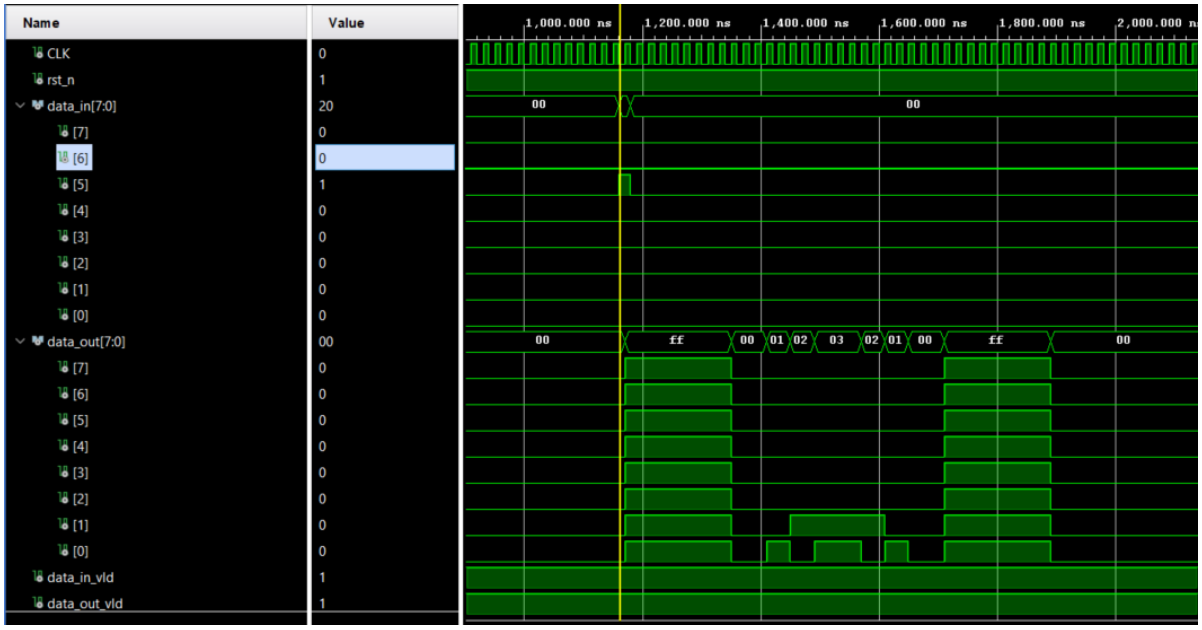


Figure 15 Vivado Simulation Results (input = 32)

-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	0	1
	1	2	2	3	3	3	3	2	2	1	1	0
	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	

Table 3 MATLAB output (input = 32)

Verification of Results Using Testbenches (Email Service)

This section will compare the sound wave of 'HillTopHoods.wav' in its unfiltered form, its MATLAB filtered form and the result generated by the Vivado Email Service (testbench) all in the frequency domain.

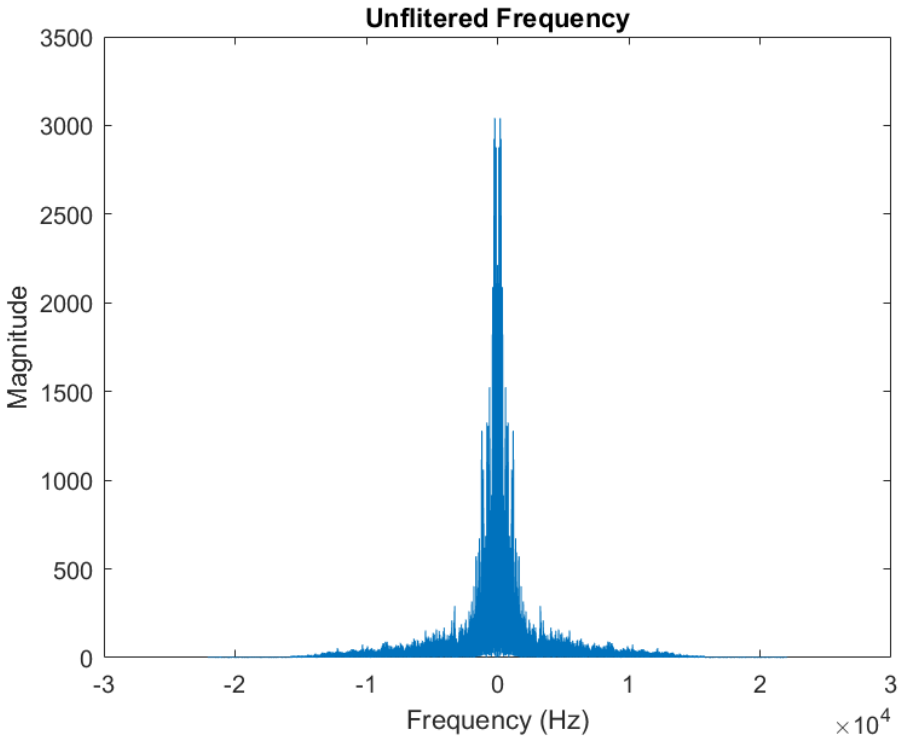


Figure 16 Unfiltered HillTopHoods Song Frequency Domain

Figure 16 shows the unfiltered Hilltop Hoods song plotted in the frequency domain. It has quite a large frequency spectrum spanning out to approximately 15,000Hz. The filter to be applied to this signal is a low pass. Ideally, after applying a low pass filter, all the higher frequencies are then removed and only the lower frequencies are left.

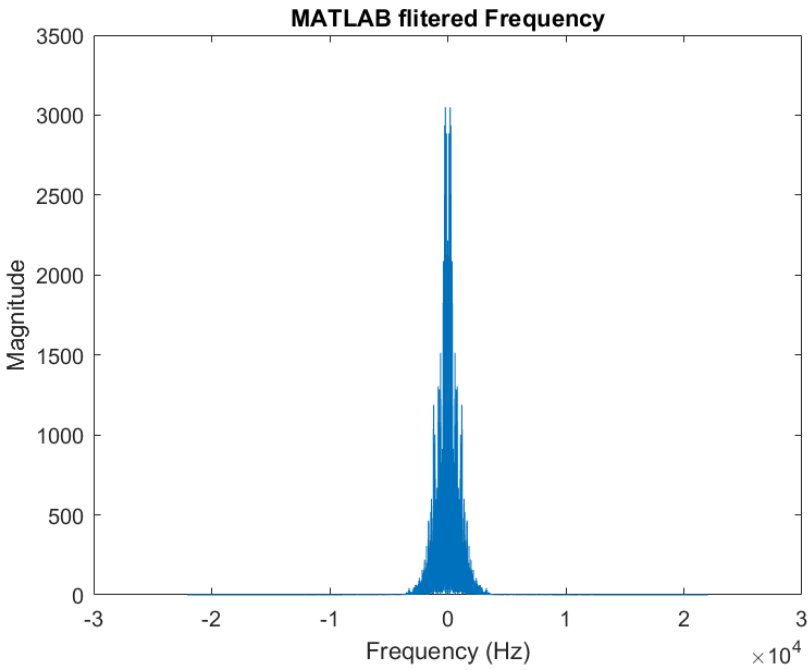


Figure 17 MATLAB Filtered HillTopHoods Song Frequency Domain

Figure 17 refers to the filtered output of the Hilltop Hoods song after a low pass filter has been applied in MATLAB. The "filter" function was used with the h values generated in previous sections of the code. As you can see, the sound wave no longer has frequencies higher than 5000Hz. This means that a low pass filter has been applied to this signal.

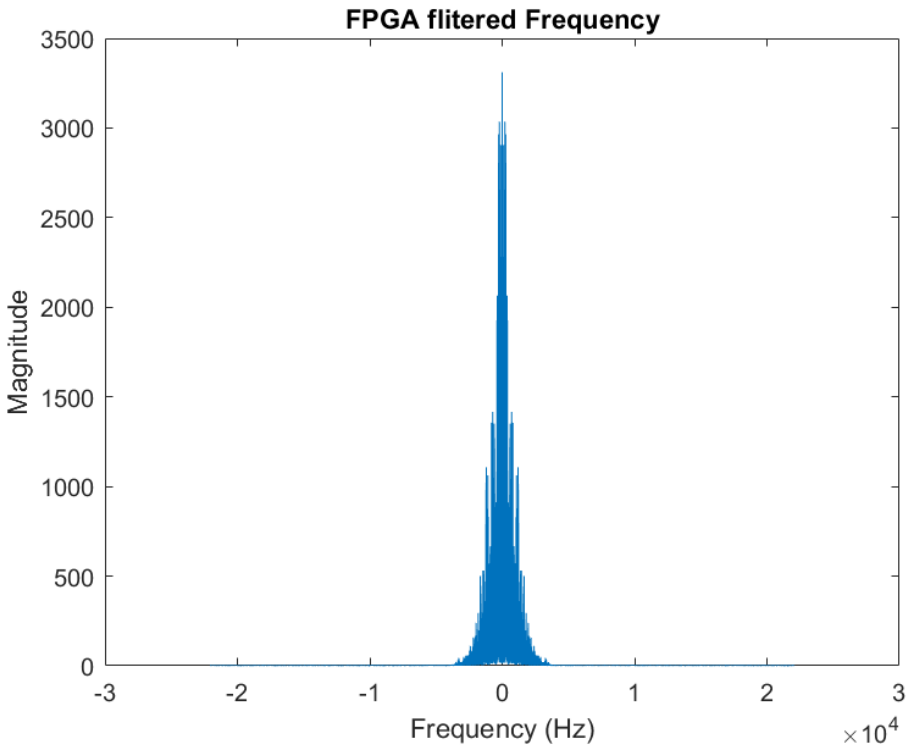


Figure 18 FPGA Filtered HillTopHood Song Frequency Domain

Figure 18 refers to the sound wave of the Hilltop Hoods song after being inputted through the Vivado Testbench. Like the MATLAB filtered sound, no frequencies higher than 5,000Hz are now present in the sound file in the frequency domain. One difference, however, is that that there now appears to a spike situated almost exactly at 1 Hz. This spike was not present in the unfiltered and MATLAB filtered sound waves. One potential explanation for this is due to how the email testbench is setup. There is a large amount of converting that needs to be done to move the sound file through email, to the UART, through the filter, back to email and then displayed on MATLAB. There is a possibility that this conversion between many types of data has caused there to be incorrect information.

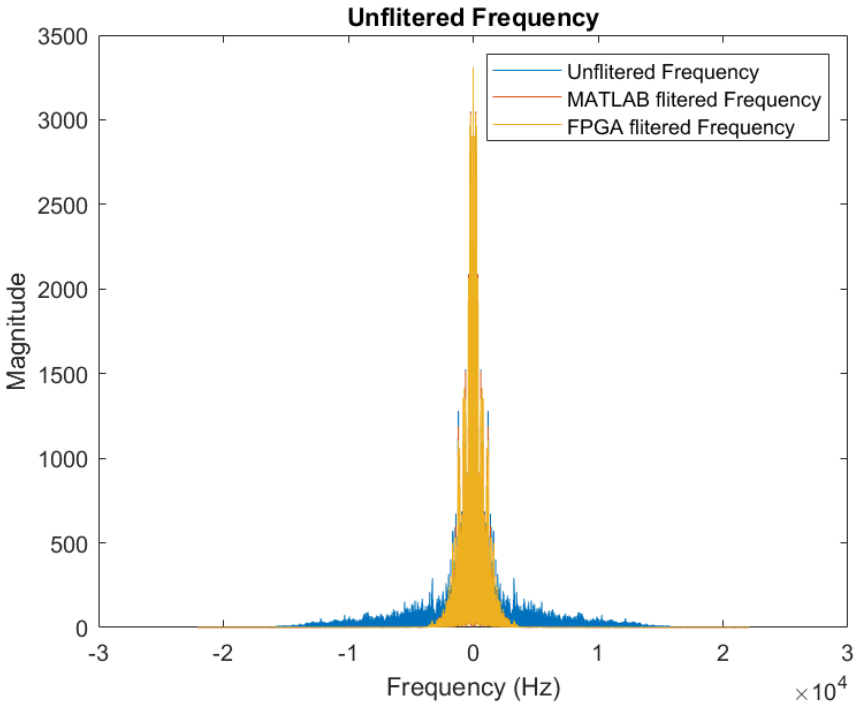


Figure 19 Unfiltered, MATLAB Filtered and FPGA Filtered HillTopHood Song in Frequency Domain

Figure 19 contains a plot of the unfiltered, MATLAB filtered, and FPGA filtered versions of the Hilltop Hoods song. It appears that the FPGA filter can mimic the MATLAB filter extremely accurately. Both the MATLAB and FPGA filter appears to have the same band stop. One issue however, the FPGA filter appears to not be able to completely remove the sound in the higher frequencies. This is due to the filter outputting whole numbers. As mentioned previously, the relative error introduced by multiplying x(t) (sound file) by the outer limits of the coefficients (h(36) or h(1)) is extremely high. The reasoning for this is because the FPGA filter is only able to output whole numbers. When the FIR filter attempts to multiply the signal by a very small negative h value, the smallest negative number it can use to represent that number is -1. This introduces a problem since now all the outputted values have a magnitude of 1. This is what the very small yellow signals are and can faintly be sometimes heard in the form of static.

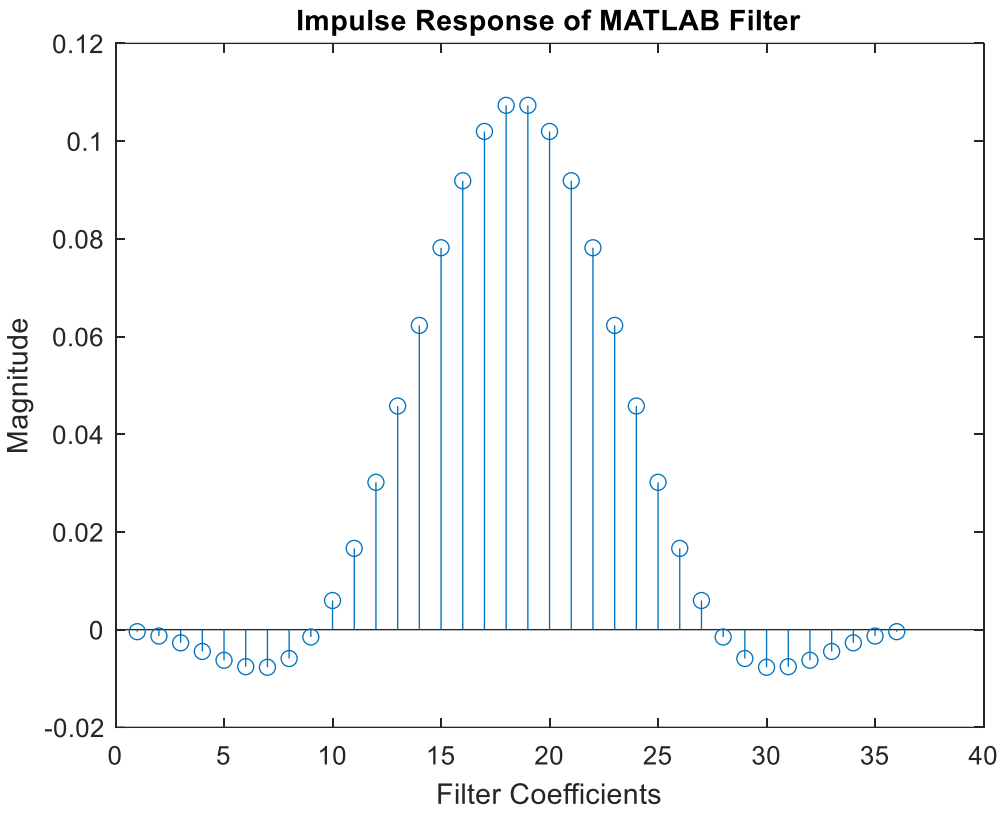


Figure 20 Coefficient Values of MATLAB Filter

Figure 20 shows the values of the coefficients from the MATLAB filter. As mentioned previously, the outer parts of the filter contain very small coefficients. When the convolution is performed of the sound file and this filter, the small negative coefficients product is rounded to negative 1 instead of 0. The static present in the FPGA filtered song is fundamentally impossible to remove without allowing the output to be decimal binary numbers.

Analysis of Design

Speed and Throughput

Definition of Throughput

For the analysis of this section this definition of throughput will be used, **"Throughput** is the rate at which the system can process inputs. It is an amount of measurements per a given time." (NI, 2021).

Definition of Latency

For the analysis of the filter, this definition of latency will be used "**Latency** is the amount of time it takes to complete an operation. It is measured in units of time, and is often found in units of milliseconds, microseconds, and nanoseconds." (NI, 2021).

Therefore, this section will measure how quick each input can be fed through each of the coefficients of the FIR filter.

```
76      data_in <= "00000000"; wait for 20 ns;  --0
77      data_in <= "00000000"; wait for 20 ns;  --0
78            data_in <= "00000000"; wait for 20 ns;  --0
79      data_in <= "00000000"; wait for 20 ns;  --0
80      data_in <= "00000000"; wait for 20 ns;  --0
81      data_in <= "00000000"; wait for 20 ns;  --0
82      data_in <= "00000000"; wait for 20 ns;  --0
83            data_in <= "00000000"; wait for 20 ns;  --0
84      data_in <= "00000000"; wait for 20 ns;  --0
85      data_in <= "00000000"; wait for 20 ns;  --0
86      data_in <= "00000000"; wait for 20 ns;  --0
87      data_in <= "00000000"; wait for 20 ns;  --0
88            data_in <= "00000000"; wait for 20 ns;  --0
89      data_in <= "00000000"; wait for 20 ns;  --0
90      data_in <= "00000000"; wait for 20 ns;  --0
91      data_in <= "00000000"; wait for 20 ns;  --0
92      data_in <= "00000000"; wait for 20 ns;  --0
93      data_in <= "00000000"; wait for 20 ns;  --0
94     data_in <=  "01111111"; wait for 20 ns;  --0
95      data_in <= "00000000"; wait for 20 ns;  --0
96      data_in <= "00000000"; wait for 20 ns;  --0
97      data_in <= "00000000"; wait for 20 ns;  --0
98      data_in <= "00000000"; wait for 20 ns;  --0
99      data_in <= "00000000"; wait for 20 ns;  --0
100           data_in <= "00000000"; wait for 20 ns;  --0
101     data_in <= "00000000"; wait for 20 ns;  --0
102     data_in <= "00000000"; wait for 20 ns;  --0
103     data_in <= "00000000"; wait for 20 ns;  --0
104     data_in <= "00000000"; wait for 20 ns;  --0
105           data_in <= "00000000"; wait for 20 ns;  --0
106     data_in <= "00000000"; wait for 20 ns;  --0
107     data_in <= "00000000"; wait for 20 ns;  --0
108     data_in <= "00000000"; wait for 20 ns;  --0
```

Figure 21 Testing Filter with input = 127

Figure 21 shows the file that was used to test the filter in previous sections. As you can see there is a wait of 20ns between every input. This, therefore, means that the clock is operating at a speed of 50MHz. Since the FIR filter can filter an input through each of the coefficients per clock cycle (shown before), this filter has speed value of 50,000,000MHz or 20ns per calculation.

Since the input is an 8-bit integer (1 byte), 50,000,000 can be passed through a second. This means that the filter has a throughput of 50MB/s.

When considering the full output of an impulse, it is going to take 36 clock cycles for the full output to be produced. This can be calculated by $36 * 20ns = 720\ ns$, this is effectively the round-trip response time for an impulse.

Filter Accuracy

When creating the filter there are obvious memory constraints that are present in the implementation of an FIR filter. For example, the filter is limited to only using 8-bit input, 16-bit coefficient and then an 8-bit whole number output. This means that a significant amount of accuracy is lost when using the filter. This section will analyse this loss of clarity with an input impulse of 127.

Since the MATLAB and the Vivado results are the same (proved previously), the MATLAB results will be used to compare the accuracy lost in the filter.

```
96 -    values = ones(1,36);
97 -    difference = ones(1,36);
98 -    for i=1:36
99 -        values(i) = h(i)*127;
100 -       difference(i) = abs(values(i)-y_out2(i+36));
101 -   end
```

Figure 22 Code Used to Calculate Difference

Figure 22 shows the code used to calculate the absolute error incurred by rounding the numbers down to their nearest whole number. h() contains all the values of the filter coefficients, y_out2() contains the output of the filter. There are 36 leading zeros added to the input to simulate an impulse response, that is the reason for 37 being used as the starting index.

0.947232398231361	0.834167302152154	0.657020780794057	0.432830826171582
0.203819837978475	0.0375464493323030	0.0211153606278015	
0.249389904271524	0.808556321729563	0.757689195637204	
0.111836213704326	0.830320163784789	0.813330589636476	
0.908512064924530	0.927394931758506	0.669534498403193	
0.950553810701850	0.629303644041169	0.629303644041169	
0.950553810701850	0.669534498403193	0.927394931758506	
0.908512064924530	0.813330589636476	0.830320163784789	
0.111836213704326	0.757689195637204	0.808556321729563	
0.249389904271524	0.0211153606278015	0.0375464493323030	
0.203819837978475	0.432830826171582	0.657020780794057	
0.834167302152154	0.947232398231361		

Table 4 Output of Difference() in MATLAB code

Table 4 refers to the output of the difference vector created in figure 22. Since the output bits are rounded down to their nearest whole number, the absolute error is never higher than one. A more accurate show of error is to represent the percent error introduced by the filter.

$$Percent\ Error = \left| \frac{FIR_{Output} - Expected}{Expected} \right| * 100\%$$

$$FIR_{output} = Vivado\ Results$$

$$Expected = MATLAB\ Calculations\ using\ decimal\ numbers\ output$$

Equation 3 - precent error equation

This is the formula that is going to be used to calculate the percent error. This formula can represent the error as a percent error rather than an absolute number. This is especially useful since some outputs were only -1 but had an absolute error of 0.85 or higher.

1795.10223410288	503.017386183705	191.562853958084	76.3142367646595
25.5997131931761	3.90111806499704	2.15708366221210	
33.2249600279474	422.346838001818	100	5.29568595228113
21.6775655370892	13.9907850946309	11.4877749122230	
9.34177534119951	5.73745678111504	7.33986997464424	
4.61728390882468	4.61728390882468	7.33986997464424	
5.73745678111504	9.34177534119951	11.4877749122230	
13.9907850946309	21.6775655370892	5.29568595228113	100
422.346838001818	33.2249600279474	2.15708366221210	
3.90111806499704	25.5997131931761	76.3142367646595	
191.562853958084	503.017386183705	1795.10223410288	

Table 5 Relative_Erorr Values

Table 5 refers to the percent error generated by the FIR filter. As you can see there are huge percent errors being generated at the ends of the filter. This is because the h coefficients are extremely small, (h(1) = -4.154929273121222e-04). When the filter tries to represent that small of a negative number, the smallest number representation is therefore negative 1. When higher h values are used (in the middle of the filter), the relative error get quite small to around the 4.6% error mark.

```
103 -    relative_error = ones(1,36);
104 -  for i=1:36
105 -      relative_error(i) = abs((y_out2(i+36)-values(i))./values(i)).*100;
106 -  end
```

Figure 23 MATLAB Code Used to Calculate Percent Error

Figure 23 refers to the code used to calculate the percent error introduced by the FPGA implementation of the FIR filter. Y_out2 is the output of the FIR filter using 8-bit whole number representation, values() is the expected output using large bit values to represent decimal numbers.

Meeting the Spec

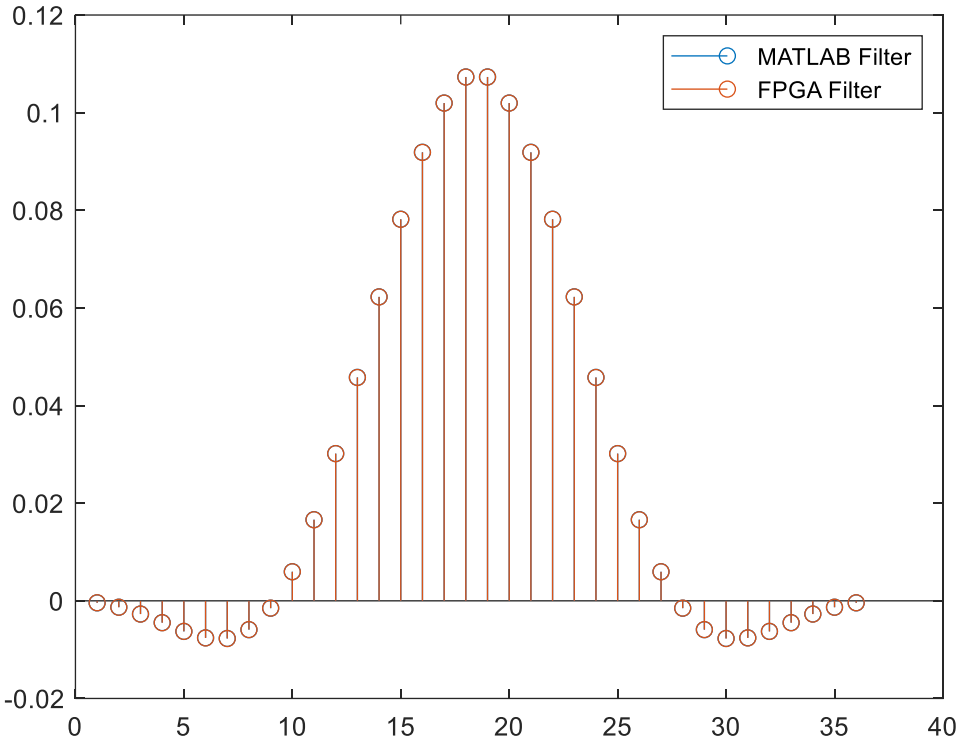


Figure 24 MATLAB Filter Coefficients and 16-bit Coefficients (FPGA)

Figure 24 is showing the difference in values between the MATLAB filter coefficients and the 16-bit coefficients used in the filter. This is a representation of the absolute error. As you can see, there is very little difference between the MATLAB and the 16-bit coefficients used in the FPGA design. This was expected through given the diminishing importance of bits as they move to the right. With most right bit currently only repressing 0.0000305. Most of the error within the design is introduced when the output is transformed into an 8bit representation from the 24bit multiplication value. Due to this a small error will be introduced as the 22 to 14 will be taken preserving most of the whole bits but removing all fraction bits from the output data.

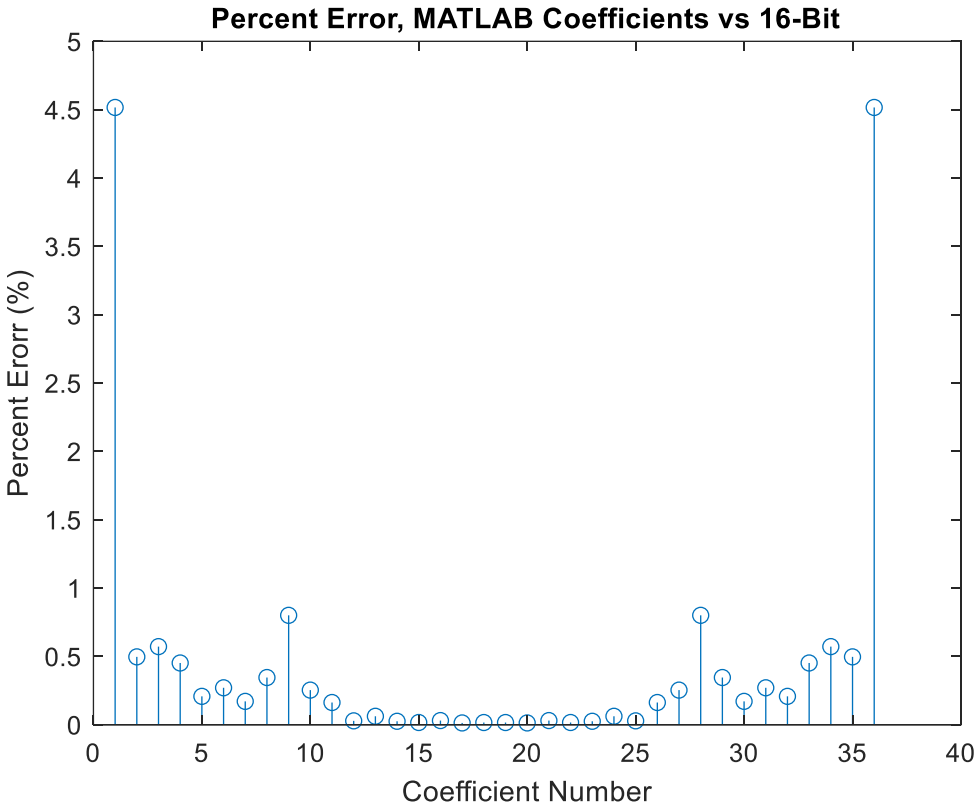


Figure 25 Percent Error, MATLAB Coefficients vs 16-Bit

Figure 25 shows the percent error introduced by approximating the MATLAB coefficients with 16-bit numbers. As mentioned previously, the error introduced by this stage is quite small with most of the error being around ~0.5%. One exception to this is smallest negative h values at 1 and 36, were 16-bit is still too small represent this small of a signed fractional number, thus causing an error of 4.5%.

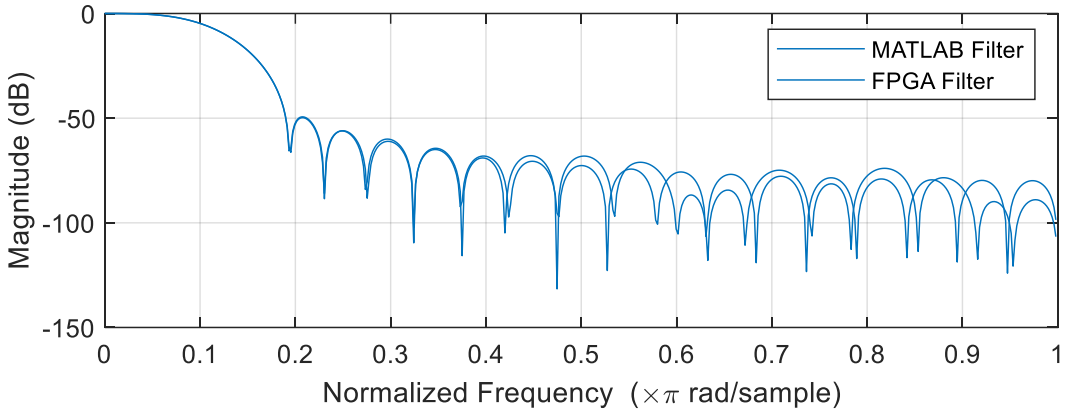


Figure 26 Magnitude Response FPGA vs MATLAB Filter

Figure 26 shows the magnitude response of the FPGA Filter coefficients (16-bit) vs MATLAB coefficients. As you can see the FPGA filter is able to accurately track the magnitude response of the MATLAB filter until about 0.4 pi radians per sample. After 0.4, They begin to separate from one another. In the opinion of the report, the FPGA filter has done a very good job of replicating the magnitude response of the MATLAB filter. One reason for this is because, when the filters begin to not replicate one another, the magnitude is already in the ~-75 dB range. At this low range of dB, the slight differences won't make an audible difference to the person listening to the filtered song.

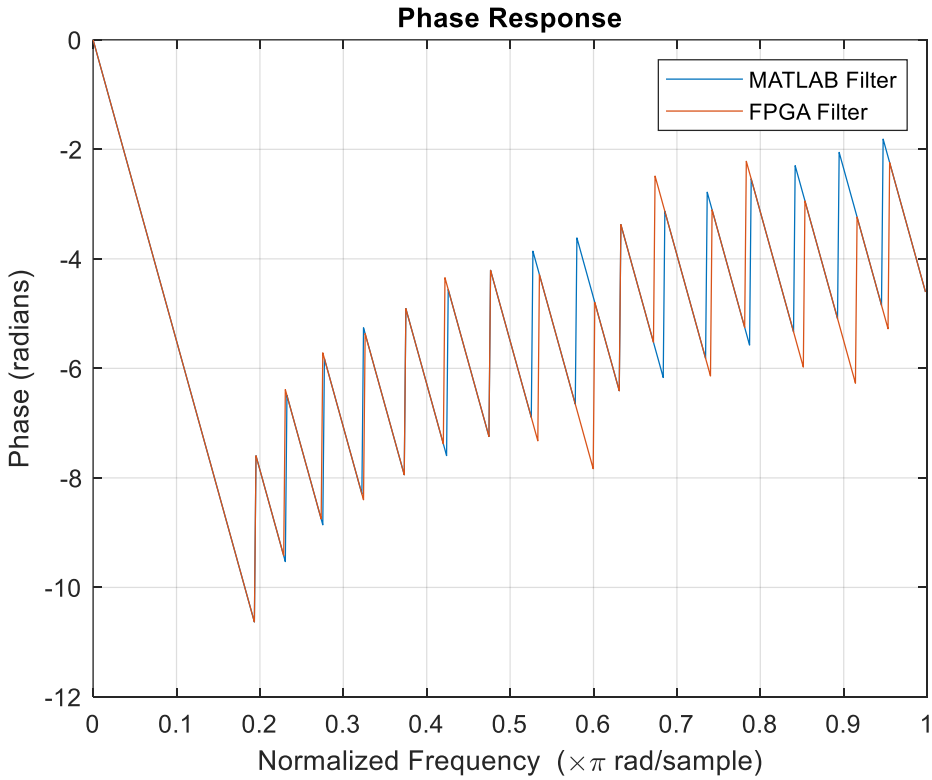


Figure 27 Phase Response FPGA vs MATLAB Filter

Figure 27 refers to the filter response of the FPGA filter (16-bit) compared to the MATLAB filter. Like the magnitude response, the FPGA filter begins to separate from the MATLAB filter at around 0.4 pi radians per sample. In the opinion of the report, this FPGA filter approximation is acceptable.

FPGA Resources Used

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	1483	20800	7.13
LUTRAM	18	9600	0.19
FF	1774	41600	4.26
BRAM	0.50	50	1.00
DSP	26	90	28.89
IO	40	210	19.05
BUFG	4	32	12.50
MMCM	1	5	20.00

Figure 28 Resources Used

Figure 28 refers to the resources used for the implementation of the filter. This implementation of an FIR filter uses a significant amount of look up tables, Digital Signal Process, Input Output and Mixed Mode Clock Manager. The high use of DSP is to be expected since an FIR filter is a digital signal process. The look up tables are added in by Vivado to improve performance of the filter.

Timing Issues

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1.441 ns	Worst Hold Slack (WHS):	0.022 ns	Worst Pulse Width Slack (WPWS):	3.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	5733	Total Number of Endpoints:	5733	Total Number of Endpoints:	1827

All user specified timing constraints are met.

Figure 29 Timing of FPGA

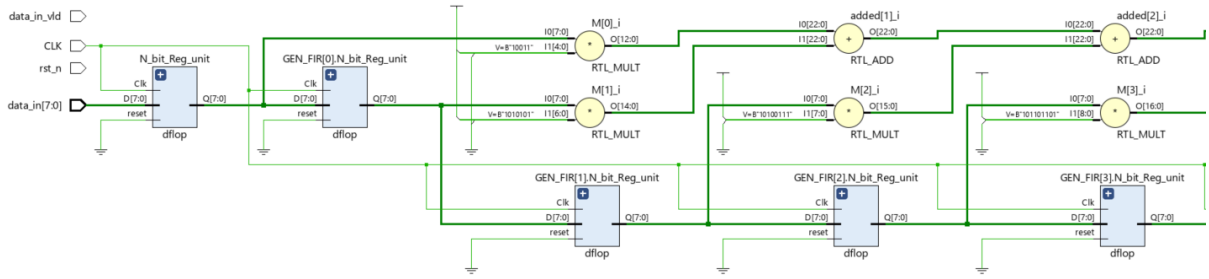
The timing of the filter allows for a slack of 1.441ns. This slack value means that there is time left over for each clock cycle and therefore means that the circuit is operating as expected. This negative slack was able to be achieved after switching the standard design of an FIR filter to the transposed structure of the filter. As mentioned previously in the report, changing the FIR filter to the transposed structure helped fix all the timing issues present in the filter implementation.

Bibliography

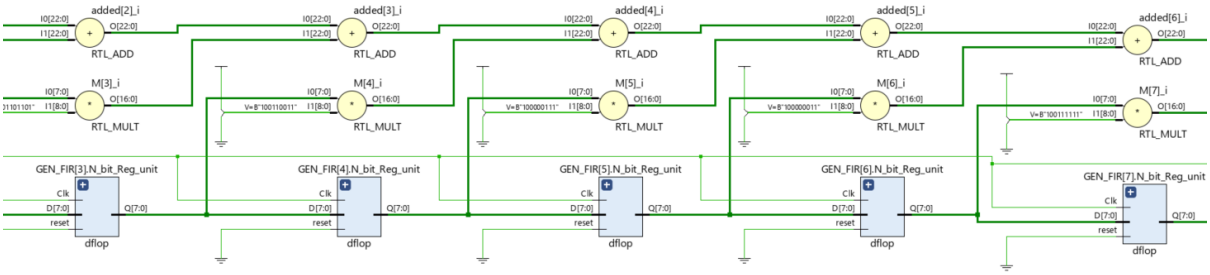
Amoako, G. (2012, July 16). *Floating number conversion to binary and vice-versa.* Retrieved from MathWorks: https://au.mathworks.com/matlabcentral/fileexchange/13899-floating-number-conversion-to-binary-and-vice-versa

NI. (2021, August 28). *Make it Faster: More Throughput or Less Latency?* Retrieved from NI: https://www.ni.com/en-au/innovations/white-papers/13/make-it-faster--more-throughput-or-less-latency-.html

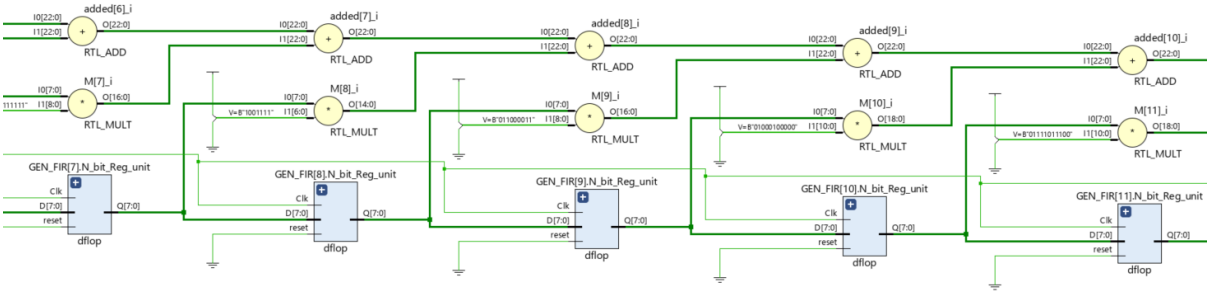
Appendix



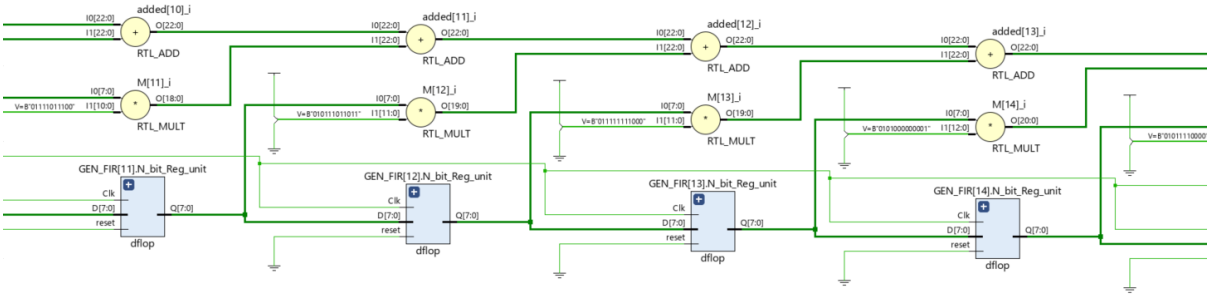
Appendix 1 RTL Schematic



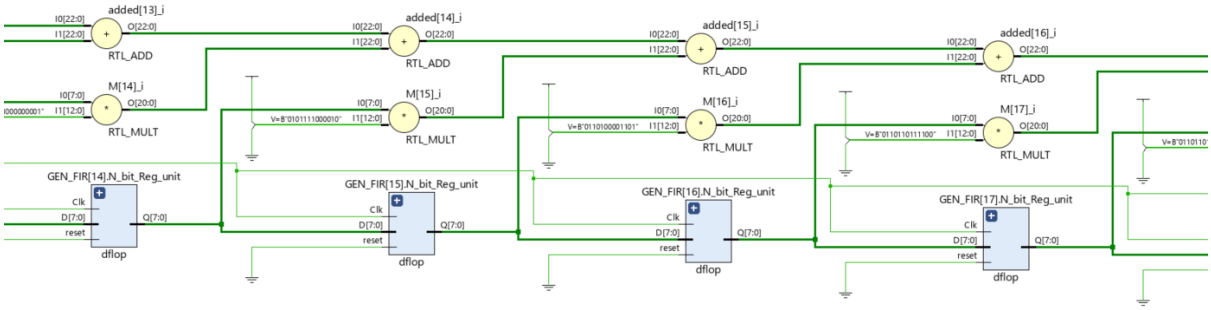
Appendix 2 RTL Schematic



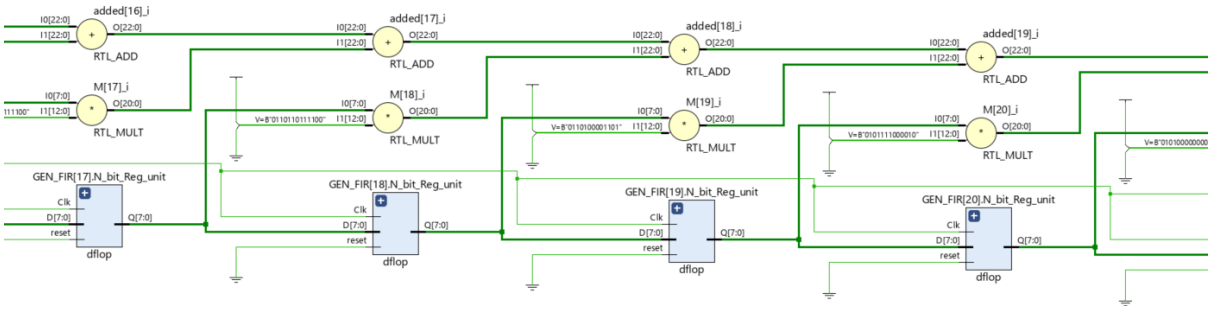
Appendix 3 RTL Schematic



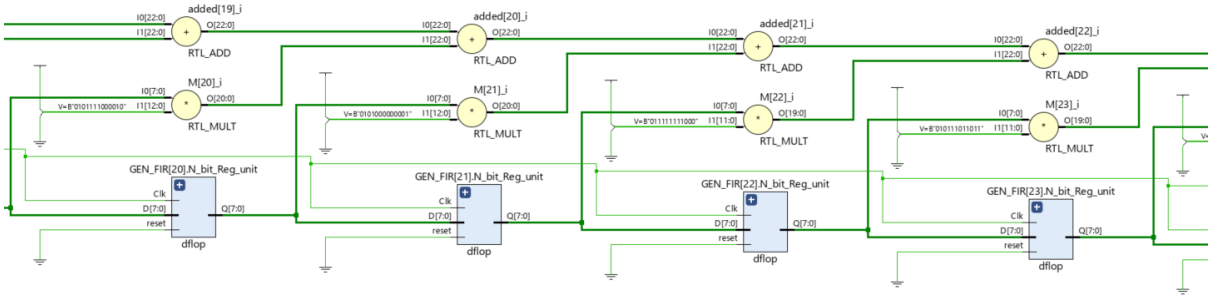
Appendix 4 RTL Schematic



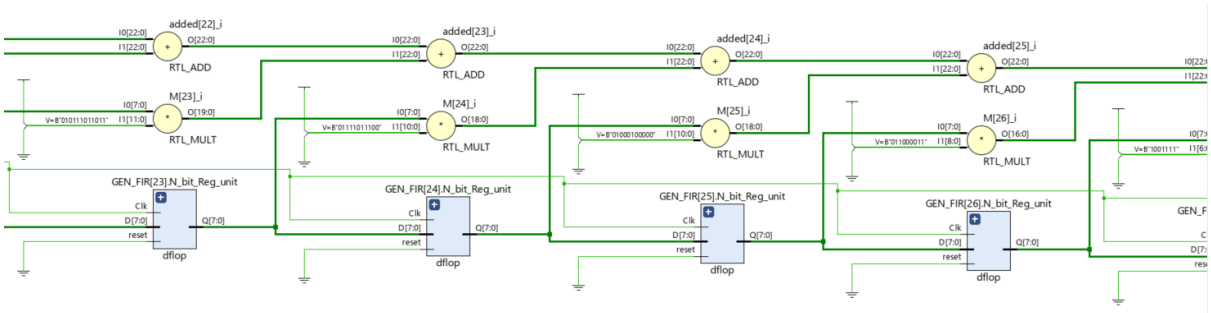
Appendix 5 RTL Schematic

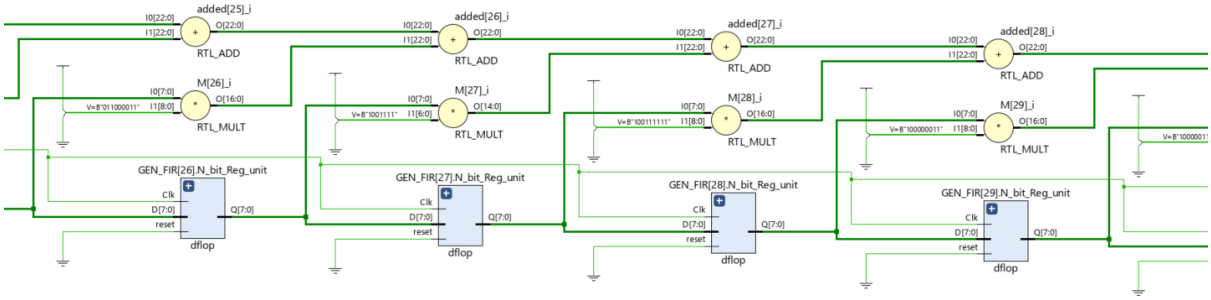


Appendix 6 RTL Schematic

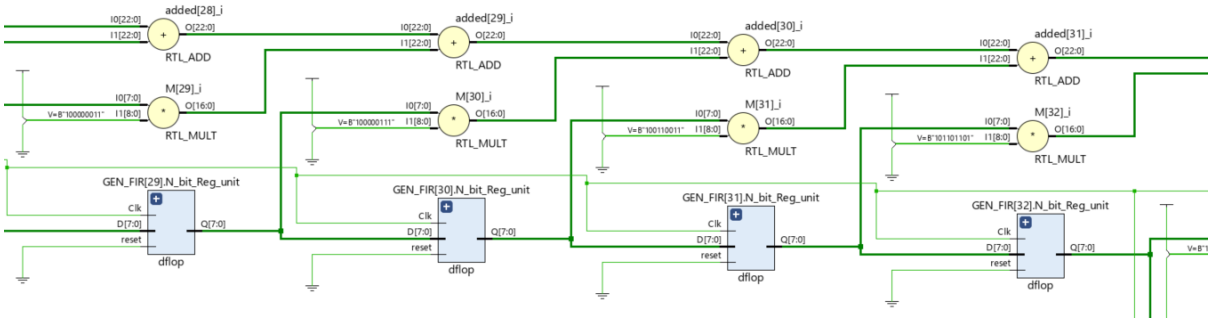


Appendix 7RTL Schematic

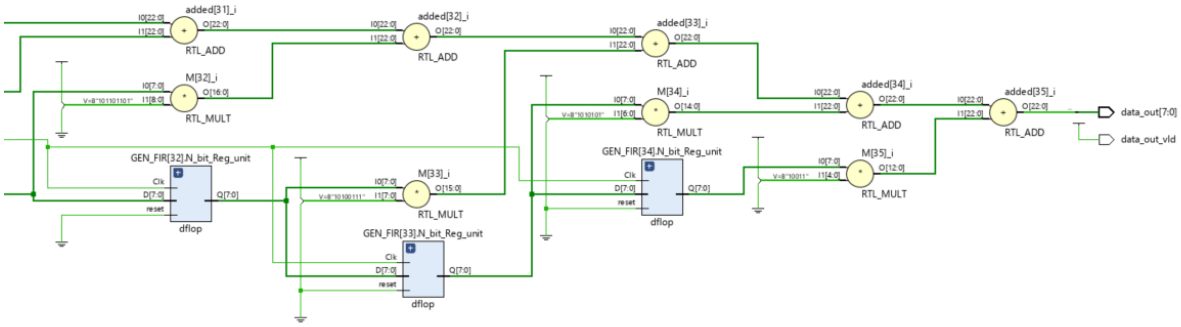




Appendix 8 RTL Schematic



Appendix 9 RTL Schematic



Appendix 10 RTL Schematic

```
numsamplesz1 = length(z1);
numsamplesz2 = length(z2);
numsamplesz3 = length(z3);

z1fft = abs(fft(z1));
z2fft = abs(fft(z2));
z3fft = abs(fft(z3));

z1fft = z1fft(1:numsamplesz1/2);
z2fft = z2fft(1:numsamplesz2/2);
z3fft = z3fft(1:numsamplesz3/2);

fz1 = fsamp*(0:numsamplesz1/2-1)/numsamplesz1;
fz2 = fsamp*(0:numsamplesz2/2-1)/numsamplesz2;
fz3 = fsamp*(0:numsamplesz3/2-1)/numsamplesz3;
fz3 = [fliplr(-fz3),fz3];
fz2 = [fliplr(-fz2),fz2];
fz1 = [fliplr(-fz1),fz1];
 t = [flipud(z3fft);z3fft];
 z2fft = [flipud(z2fft);z2fft];
 z1fft = [flipud(z1fft);z1fft];
figure(6)
plot(fz1,z1fft);title("Unflitered Frequency");xlabel("Frequency (Hz)");ylabel("Magnitude");
figure(7)
plot(fz2,z2fft);title("MATLAB flitered Frequency");xlabel("Frequency (Hz)");ylabel("Magnitude");
figure(8)
plot(fz3,t);title("FPGA flitered Frequency");xlabel("Frequency (Hz)");ylabel("Magnitude");
figure(9)
plot(fz1,z1fft);title("Unflitered Frequency");xlabel("Frequency (Hz)");ylabel("Magnitude");
hold on
plot(fz2,z2fft);
hold on
plot(fz3,t)
legend("Unflitered Frequency","MATLAB flitered Frequency","FPGA flitered Frequency")
hold off
```

Appendix 11 MATLAB Code Used to Plot HillTopHoods Song in Frequency Domain

```
[z1,fsamp] = audioread('audio/hilltop_hoods1_test.wav');
%[z1,fsamp] = audioread('audio/regurgitator_test.wav');
%[z1,fsamp] = audioread('audio/methyl_ethyl_test.wav');
[z3,fsamp] = audioread('audio\mus.wav');
```

Appendix 12 MATLAB Code Used to Generate HillTopHoods Song

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY my_FIR_filter IS
	GENERIC (
		Filter_count : INTEGER := 36; --number of h coef
		coef_bits : INTEGER := 16; -- number of bits in each coef`
		G_DATA_WIDTH : INTEGER := 8; --length of data in and out
		bits_out : INTEGER := 24 --length of multiplied values
	);
	PORT (
		CLK : IN std_logic; --clock signal
		rst_n : IN std_logic; --reset signal
		data_in : IN std_logic_vector(G_DATA_WIDTH - 1 DOWNTO 0); --input signal
		data_in_vld : IN std_logic; --1 if data is sent to FIR
		data_out : OUT std_logic_vector(G_DATA_WIDTH - 1 DOWNTO 0); --filter output
		data_out_vld : OUT std_logic --1 if the FIR has data to sent back
	);
END my_FIR_filter;

ARCHITECTURE behav OF my_FIR_filter IS
	TYPE coefficients IS ARRAY (0 TO Filter_count - 1) OF signed(coef_bits - 1 DOWNTO 0); --data line to store h values
	TYPE Multiply IS ARRAY (0 TO Filter_count - 1) OF signed(bits_out - 1 DOWNTO 0); --multiplied data line to store values
	TYPE in_bit IS ARRAY (0 TO Filter_count - 1) OF signed(bits_out - 1 DOWNTO 0);
	TYPE add IS ARRAY (0 TO Filter_count - 1) OF signed(bits_out - 1 DOWNTO 0); --data line to store added values
	-- initialisation of the 36 h values for the FIR filter --
	CONSTANT h : coefficients := ("1111111111110011", "1111111111010101", "1111111110100111",
		"1111111101101101", "1111111100110011", "1111111100000111",
		"1111111100000011", "1111111100111111", "1111111111001111",
		"0000000011000011", "0000001000100000", "0000001111011100",
		"0000010111011011", "0000011111111000", "0000101000000001",
		"0000101111000010", "0000110100001101", "0000110110111100",
		"0000110110111100", "0000110100001101", "0000101111000010",
		"0000101000000001", "0000011111111000", "0000010111011011",
		"0000001111011100", "0000001000100000", "0000000011000011",
		"1111111111001111", "1111111100111111", "1111111100000011",
		"1111111100000111", "1111111100110011", "1111111101101101",
		"1111111110100111", "1111111111010101", "1111111111110011"
	);
	SIGNAL x_in : signed (G_DATA_WIDTH - 1 DOWNTO 0) := (OTHERS => '0'); --store to copy the in data
	SIGNAL padding : signed(4 DOWNTO 0) := (OTHERS => '0'); --count to wait before stopping to send out data
	SIGNAL added : add := (OTHERS => (OTHERS => '0')); --signal line to store added dara
	SIGNAL M : Multiply := (OTHERS => (OTHERS => '0')); --signal line to store multiplied data
	SIGNAL Q_array : in_bit := (OTHERS => (OTHERS => '0')); --signal line to hold multiplied data before being sent through
	                                                        the flipflop
	SIGNAL clear : signed(bits_out - 1 DOWNTO 0) := (OTHERS => '0'); --signal to set data_out to when reset is called
	SIGNAL buffer_out : signed (bits_out - 1 DOWNTO 0); --store to copy the 24bit out data
	--Data PIPELINE internal signal
	SIGNAL pipe_2_vld : STD_LOGIC;
	SIGNAL pipe_2_data : STD_LOGIC_VECTOR(G_DATA_WIDTH - 1 DOWNTO 00);

BEGIN
	x_in <= signed(data_in); --set internal signal to external data in pin
	pipe_2_data <= std_logic_vector(buffer_out(22 DOWNTO 15)); --set the data out internal pin to the top 23 to 16 bits
	data_out <= pipe_2_data; --return internal data out ot output pin
	M(Filter_count - 1) <= x_in * H(Filter_count - 1); --set the last h value to mutiply the x in
	Q_array(0) <= M(Filter_count - 1); --set up the last mutiply to be transfred to the first Q line
	added(0) <= M(Filter_count - 1); --set the first added line to be maped to the last mutiplied value
	-- Construction of the clock and clear logic for the FIR Filter Outputs --
	PROCESS (Clk)
	BEGIN
		IF (rising_edge(Clk)) THEN
			-- if data is being sent output data from the FIR filter --
			IF (data_in_vld = '1') THEN
				buffer_out <= added(Filter_count - 1); --set data to come out of filter
				data_out_vld <= '1'; --set the FIR filter it tell other components it is outputing
				-- if all data has been oututed stop sending data --
			ELSIF (padding = "11111") THEN
				data_out_vld <= '0'; --tell other components no data is being sent from the FIR filter
				buffer_out <= clear; --set FIR to send no data out
				padding <= "00000"; --reset offset counter
				-- added to ensure that the 36 clock cycle delay from last bit in to final output value --
				-- is still outputed via counting up a 36 counter before shutting off the data out line --
			ELSE
				padding <= padding + "00001";
			END IF;
		END IF;
	END PROCESS;

	-- begin the construction of the FIR filter --
	fIR_fflops :
	-- creates the 35 flip flops needed --
	FOR i IN 1 TO Filter_count - 1 GENERATE
		BEGIN
			-- imply a flip flop through tieing output and input to the clock signal --
			PROCESS (Clk)
			BEGIN
				-- Flip flop Logic --
				IF (rising_edge(Clk)) THEN --each clock cycle
					IF (data_in_vld = '1') THEN --if FIR filter still needs to send data out
						Q_array(i) <= added(i - 1); --copy over the last added array to the next
						                            Q holding line (next flip flop)
					END IF;
				END IF;
				-- end Flip flop Logic --
			END PROCESS;
			added(i) <= Q_array(i) + M(Filter_count - 1 - i); --setting what each added line is calculated
			END GENERATE fIR_fflops;
```

```
                        -- create the mutiplication for the data in and h values --
                        MUL :
                        -- create the other 35 mutiplication lines --
                        FOR i IN 0 TO Filter_count - 2 GENERATE
                                BEGIN
                                        M(i) <= x_in * H(i); --set each M line to be the data in and one of the 0 to 35 h
                                                                values
                                END GENERATE MUL;

END behav;
```

Appendix 13 VHDL code for the file my_FIR_filter.vhd

```
----------------------------------------------------------------------------------
-- Create Date: 29.10.2021 15:03:26
-- Design Name:
-- Module Name: fir_test - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

ENTITY fir_test IS
        -- Port ();
END fir_test;

ARCHITECTURE Behavioral OF fir_test IS

        SIGNAL CLK : std_logic := '0';
        SIGNAL rst_n : std_logic := '1';
        SIGNAL data_in : std_logic_vector(7 DOWNTO 0) := (OTHERS => '0');
        SIGNAL data_out : std_logic_vector(7 DOWNTO 0) := (OTHERS => '0');
        SIGNAL data_in_vld : std_logic := '0';
        SIGNAL data_out_vld : std_logic := '0';

BEGIN
        CLK <= NOT CLK AFTER 10 ns;

        -- Instantiate the Unit Under Test (UUT)
        uut : ENTITY work.my_FIR_filter
                PORT MAP(
                        CLK => CLK,
                        rst_n => rst_n,
                        data_in => data_in,
                        data_in_vld => data_in_vld,
                        data_out => data_out,
                        data_out_vld => data_out_vld
                );

                        -- Stimulus process
                        stim_proc : PROCESS
                        BEGIN
                                WAIT FOR 20ns;
                                rst_n <= '0';
                                WAIT FOR 20ns;
                                rst_n <= '1';
                                WAIT FOR 20ns;

                                data_in_vld <= '1';

                                WAIT FOR 200 ns;

                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 200 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
```

```
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000001";
WAIT FOR 20 ns; --0
data_in <= "00000010";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "01001000";
WAIT FOR 20 ns; --0
data_in <= "10000000";
WAIT FOR 20 ns; --0
data_in <= "00001000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00011000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
WAIT FOR 20 ns; --0
data_in <= "00000000";
```

```
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "01001000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00100000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00001100";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00001100";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0
                                data_in <= "00000000";
                                WAIT FOR 20 ns; --0

                                WAIT;
                        END PROCESS;
END Behavioral;
```

Appendix 14 VHDL code for the fir_test.vhd

```
clear all;
StudentNo = 10489045;   % Enter your student number here
filterType = 'lp';      % choose low pass filter
%filterType = 'hp';    % choose high pass filter
NumRecords = 348;       % number of lines in params file, do not edit


[fp, fs, delta1, delta2] = findMyParams (StudentNo,NumRecords,filterType);
[h,Nfilter] = myFIRFilter (fs, fp, delta1, delta2, filterType);


i_ar =  strings(36,1);
i_neg = strings(36,1);
num = ones(36,1);
for i=1:36
   i_ar(i)=to_binary_frac(abs(h(i)),16);
   format long
   num(i,1) = str2double(i_ar(i));
   if (abs(h(i)) ~=h(i))
      i_neg(i) = to_binary_frac(abs(h(i)),16) ;
   end
end


for i=1:36
    (i_ar);
end
num = ones(16,1);
for i=1:16
    num(i)=2^-(i-1);
end
fix_bin = strings(36,1);
for ii=1:36
    if (abs(h(ii)) ~=h(ii))
        fix_bin(ii)=neg_bin_covf(i_ar(ii));
    else
        fix_bin(ii)=i_ar(ii);
    end
end
fix_bin
```

```
function bin = neg_bin_covf(b)
carry = 1;
bin="";
 for i=0:15
     b
        bp = extract(b,16-i);
        carry = str2double(bp) && carry;
        xx = int8(xor(int8(xor(str2double(extract(b,16-i)),1)),carry));
        bin= strcat (int2str(xx),bin);
 end
```

x

```
end
function d = bin2dec(b)
d=0;
if (extract(b,16)=="1")
    d=-1;
    for i=2:16
        bp=extract(b,i);
        d=d+2^(-(i-1))*str2double(bp);
    end
else
    for i=2:6
        bp=extract(b,i);
        d=d+2^(-(i-1))*str2double(bp);
    end
end
end
function [ outstr ] = to_binary_frac( num, ndigits )
%converts a fractional number (between 0 and 1) to a string of binary digits
% ndigits - number of digits required
%
% result will contain (ndigits - 1) digits after the radix point.
% MSB will be of 2^0 weighting,
% following digits will be of 2^-1, 2^-2 ... 2^(ndigits-1) weighting
%




%ndigits=8;
%num = 0.5;


outstr = '0';
num1 = num;


for n=1:ndigits-1

    num1 = num1* 2;

    if (num1 >= 1.0)
        outstr = strcat (outstr,'1');
        num1 = num1 - 1.0;
    else
        outstr = strcat (outstr,'0');
    end
end
end
function [fp, fs, delta1, delta2] = findMyParams (StudentNo,N,filt_type)


%N = 218;
%StudentNo = 00817996;
%filt_type='lp';
```

```
fptr = fopen('params.txt','r');
[A] = fscanf(fptr,'%d %g %g %g %g %d',[6 N]);
fclose(fptr);


for i=1:N
     if (StudentNo == A(1,i))
         if (strcmp(filt_type,'hp') && (A(6,i) == 1))
           fp = A(2,i);
           fs = A(3,i);
           delta1 = A(4,i);
           delta2 = A(5,i);
         elseif (strcmp(filt_type,'lp') && (A(6,i) == 0))
           fp = A(2,i);
           fs = A(3,i);
           delta1 = A(4,i);
           delta2 = A(5,i);
         end
    end
end
end
```

Appendix 15 MATLAB code USED to generate 16bit coefficients for the FIR Filter