ECE 4020

Lab Part 5 Report

Ryan Colon

04.01.22

Introduction

Contained in this report is the documentation of a Direct Form I implementation of an IIR filter on a dedicated digital signal processing board, and its performance evaluation. First is the discussion of the IIR filter’s design, followed by its implementation on a Blackfin BF561, and concluded by verification of its performance metrics.

IIR Filter Design

The IIR filter discussed in this report initially began its existence as a design created using MATLAB. The specifications for the filter are as follows: type I Chebyshev Bandpass, 4 poles; a center frequency of 1.2 kHz; a bandwidth based upon the formula , where M is the month of the filter creator’s birthday; a 1.5 dB passband ripple with a maximum gain in the passband of 5 dB; at a sample rate of 16 kHz. My birthday is in the month of March, so M = 3; implementing this in MATLAB using the cheby1 function is trivial:

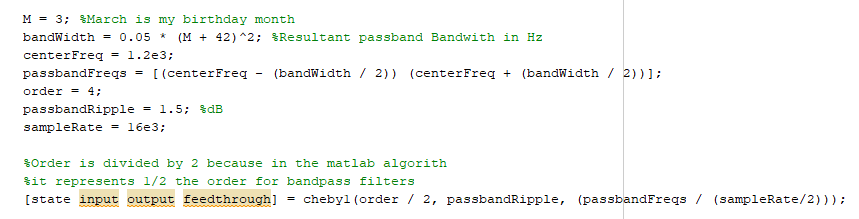


Figure 1. cheby1 Filter design MATLAB snippet

However, there is still some adjustment to be done for this filter to meet spec and be usable for the blackfin implementation. First, the output of the cheby1 function is a filter described by a state-space system, but to use the Direct Form 1 implementation the filter must be described by two second order systems. This can be achieved easily with the MATLAB function ss2sos. Next, the current gain of the is system is centered very high above 0 dB, so we shift it down to a max value of 5 dB by multiplying both second order systems by the square root of our desired gain (5 dB). This is implementation is shown in the code snippet on the next page:

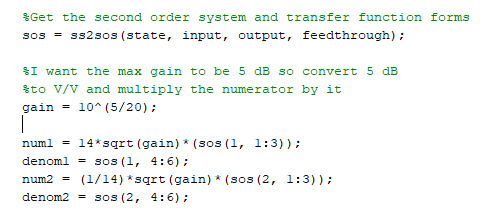


Figure 2. SOS conversion and center gain shift MATLAB snippet

It can also be seen that the second order stages are being multiplied by some factor of 14. This is done to level out the second order systems so both of their max gains are relatively close to each other in level but done in such a way that does not impact the combination of the systems overall performance. This value must be determined either by experimentation or close inspection in MATLAB, and maybe different for every user. This step is important, or else when you combine the two systems in implementation, very lopsided performance will be observed.

Before implementation, we can double check and observe the performance of the filter in MATLAB. This can be done by converting both stages to MATLAB digital filter objects, cascading them, and then plotting their response using freqz. The code and corresponding output are shown below:

Text, letter

Description automatically generated

Figure 3. Filter object conversion and plotting MATLAB snippet

Chart, line chart

Description automatically generated

Figure 4. Overall system magnitude response in MATLAB

Some of the coefficients output by the SOS stages is greater than 1. However, on the BF561 we are limited to fractional math. So before outputting the coefficients to a file for use in the BF561 they are scaled down by a factor of 2 and will be scaled back up by the same amount when implemented on the DSP board. It may seem simple to scale down by a factor of 10 since the digits are just shifted to the right once, but if scaled down by too large a value quantization noise will become a much greater factor and greatly affect filter performance. Therefore, it is recommended to keep the scalar to as minimal a value as possible.

BF561 Implementation:

As stated, this filter was implemented as two Digital Biquad Filter Direct Form Is cascaded together. The Wikipedia page for digital biquad filters contains a significant amount of useful information for this implementation and was used as the basis for this lab (see reference i). The relevant information in [i] is shown below:

Diagram, schematic

Description automatically generated

Figure 5. Wikipedia Direct Form I entry

The resulting algorithm on the BF561 is a direct implementation of the difference equation shown in the Wikipedia article using fractional mathematics. The delayed inputs, delayed outputs, and filter coefficients are all hardcoded and shifted manually to keep the code very simple. The resulting C code implemented on the blackfin is shown below:

Text

Description automatically generated

Figure 6. BF561 Process Data C Code

As stated previously, this is far from the most efficient implementation of the algorithm. However, the intent was to keep the code as simple stupid (KISS) as possible to minimize the risk of error.

Performance Evaluation:

List of equipment used:

* Oscilloscope
* Arbitrary Function Generator
* Blackfin BF561 Board
* Spectrum Analyzer

The evaluation of the filter began with a simple test to visually verify that filter followed the general shape as displayed in MATLAB. For this test, the function generator output was connected to the right channel of Audio In 1 of the BF561 and the left/right channel of the Audio Out 1 of the BF561 was hooked up to the channel 1/2 input of the oscilloscope respectively. A block diagram of the connections is show on the next page:

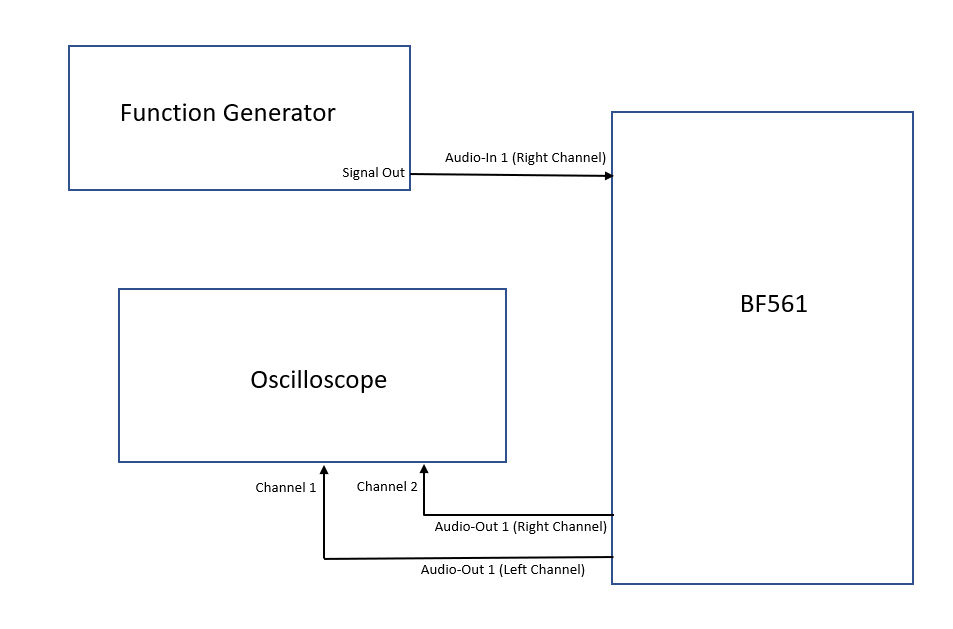


Figure 7. Basic Test Connection Setup

In this configuration, channel 1 is the output of the filter and channel 2 is the passthrough signal, and displaying both allows for filter behavior to be easily differentiated when the two signals are overlayed on top of each other. Sweeping the function generator over the entire input range showed the filter was behaving relatively as it should, so the filter can move into the next phase of testing to ensure exact performance metrics. The output of the oscilloscope at a filter maximum peak is shown below:

Graphical user interface

Description automatically generated

Figure 8. Filter Peak Amplitude Difference

For this test, a network analyzer was connected through an attenuator as an input to the BF561 and the blackfin output (Audio Out 1 Right) was plugged into T1 on the spectrum analyzer. A block diagram of this configuration is shown on the next page:

Diagram

Description automatically generated

Figure 9. Network Analyzer Test Connection Diagram

The purpose of this test is to check and make sure that the filter continues to the meet the specs outlined in the design section. For this, the magnitude and group delay of the filter and passthrough were measured over the operating range of the filter (10 Hz to 8 kHz). The filter overall and passband measurements are shown below, followed by overall passthrough measurements:

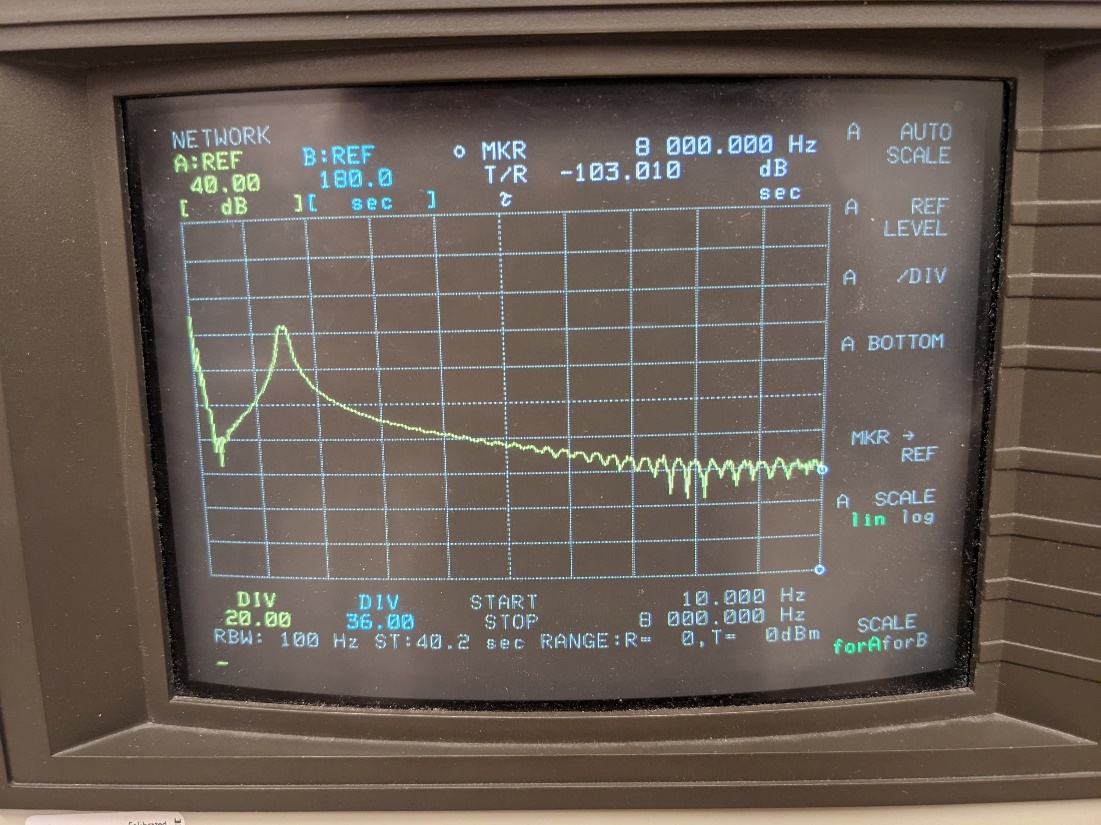


Figure 10. Overall Filter Magnitude Response

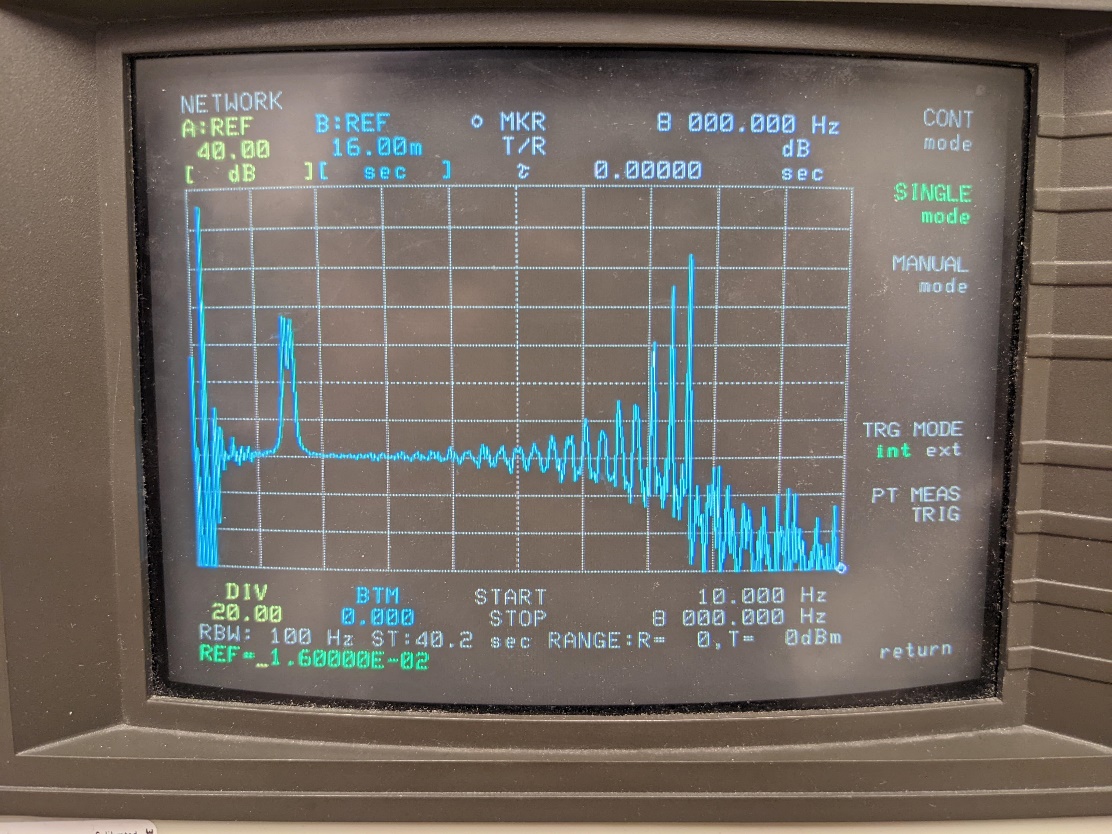


Figure 11. Overall Filter Group Delay

It can be seen in the above images that the filter has an overall performance as expected. Zooming in to the passband:

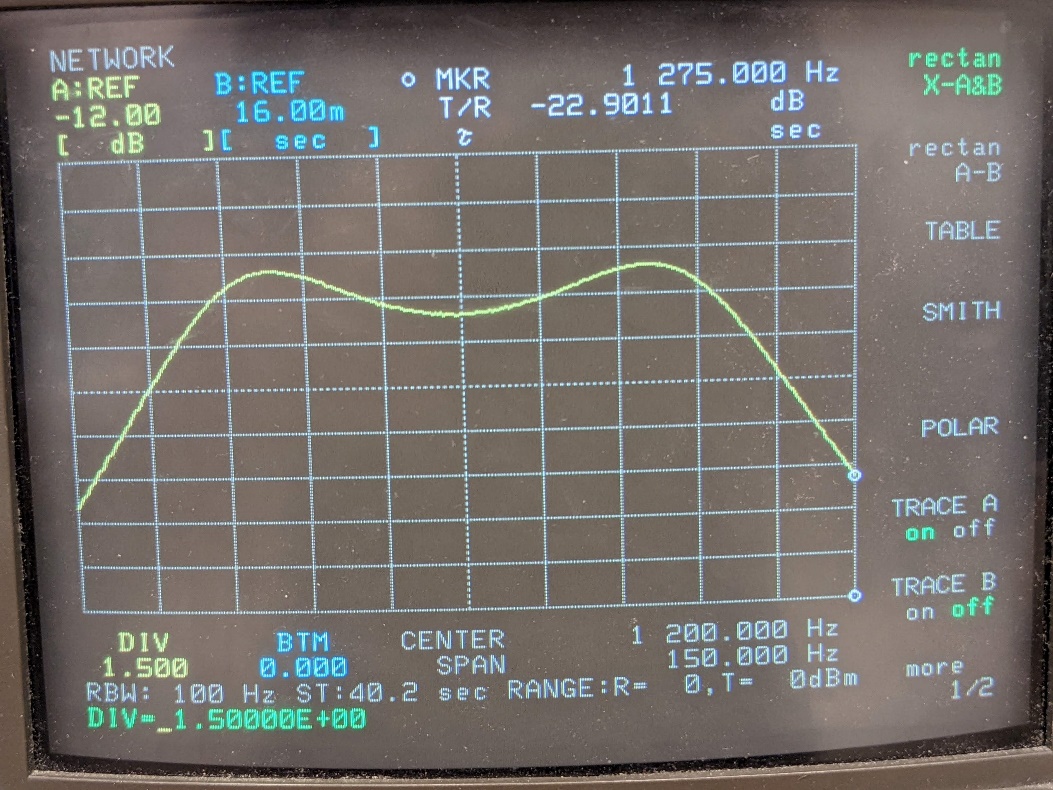


Figure 12. Passband Filter Magnitude Response

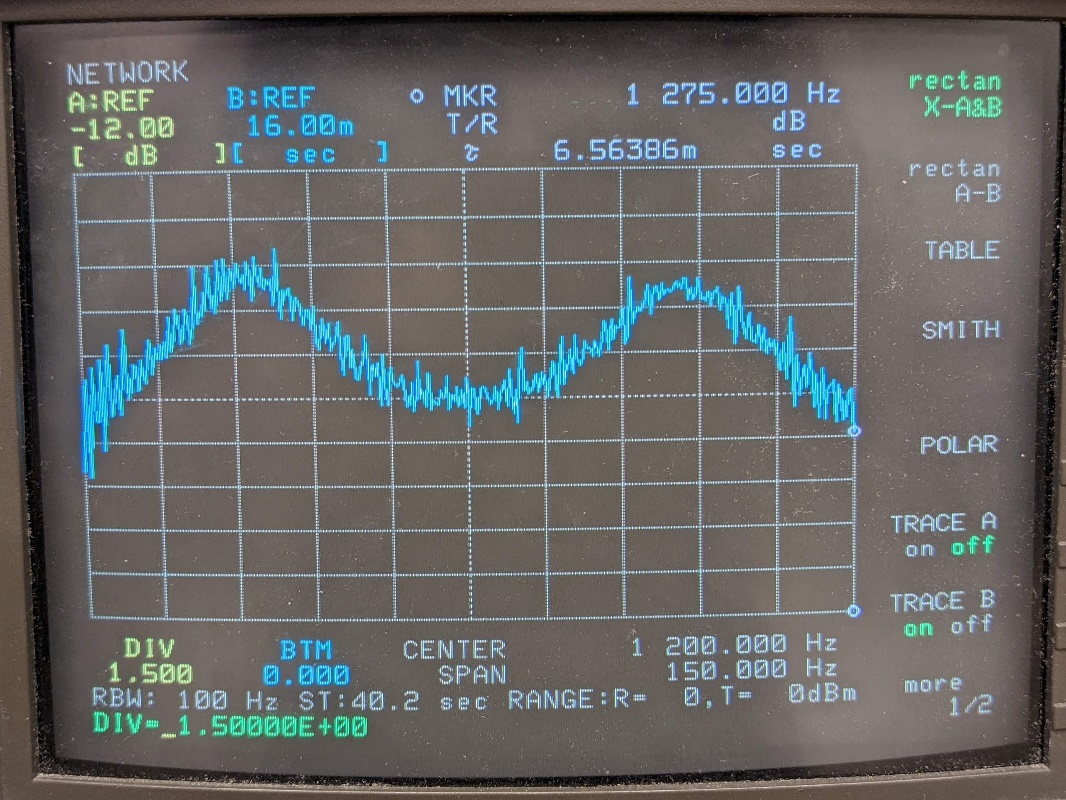


Figure 13. Passband Filter Group Delay

As can be seen by the magnitude response, the ripple is well within spec, and given the overall and passband group delay we can see that the filter is meeting the set performance standards. For reference, measurements of the overall passthrough response were taken and they are shown below:

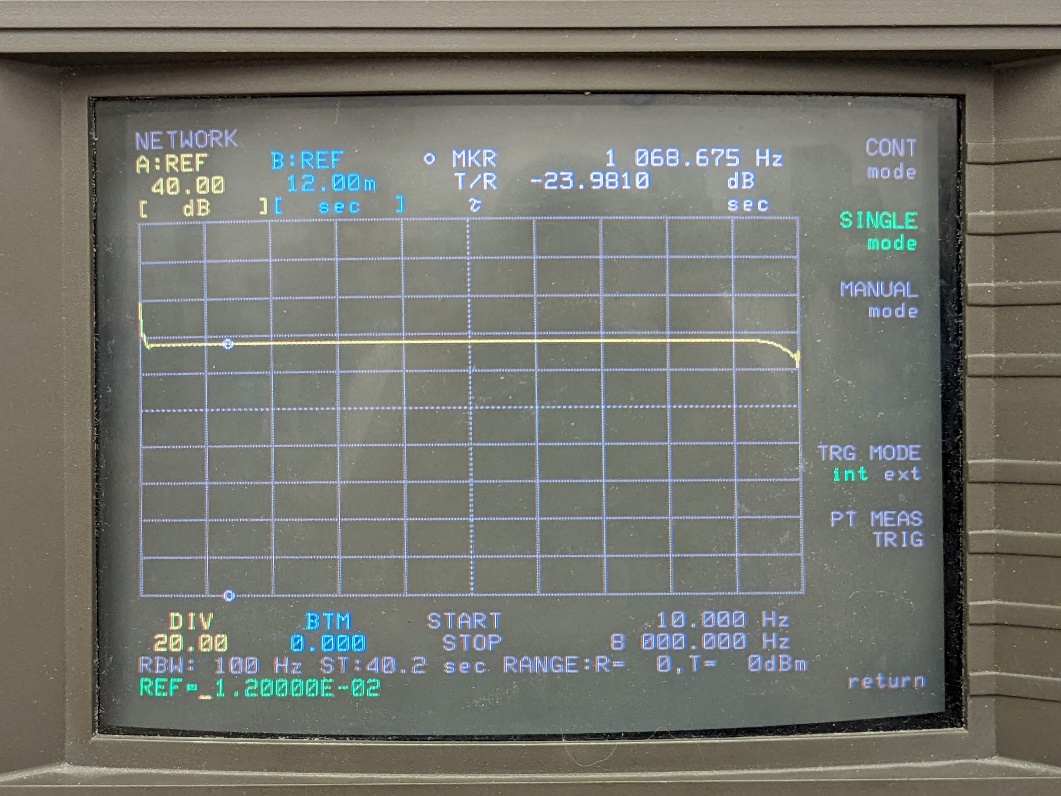


Figure 14. Overall Passthrough Magnitude Response

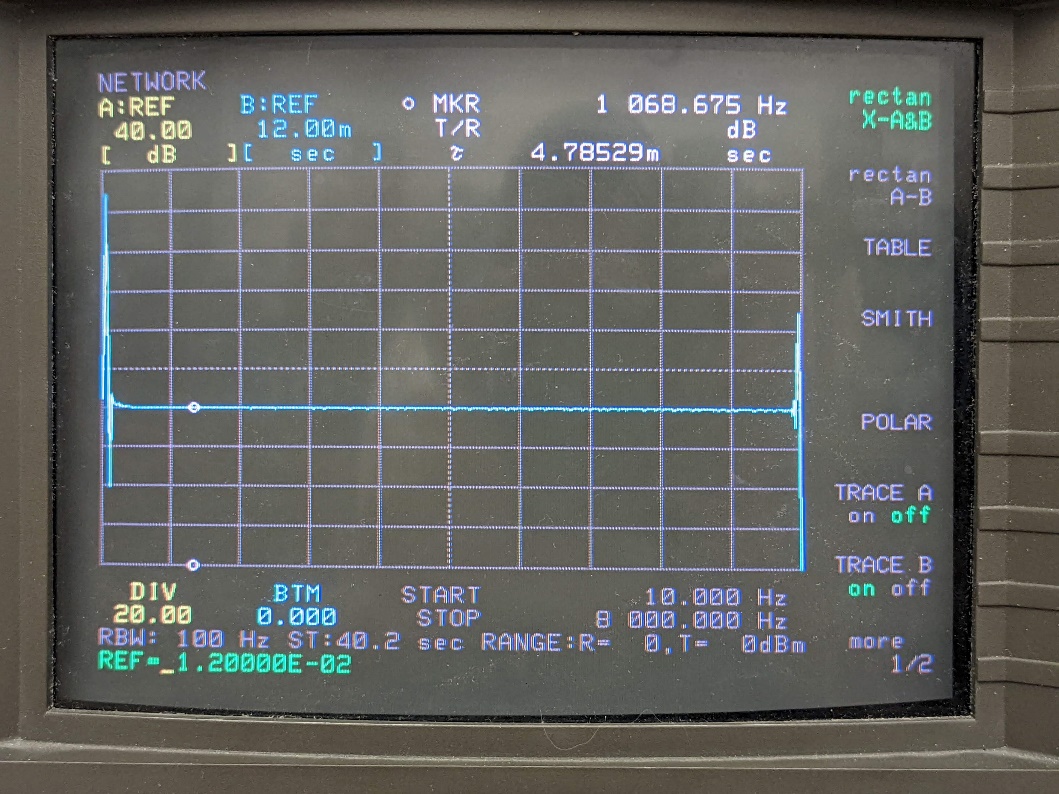


Figure 15. Overall Passthrough Group Delay

The passthrough behavior appears as expected, and the filter measurements show that it is operating well within specifications, so the goal of this experiment has been achieved.

Concluding Remarks

In this report, an IIR filter meeting discussed specifications has been designed and implemented using two cascaded Direct Form I biquad filters on a DSP board and shown to meet specs.

Citations

1. “Digital biquad filter,” *Wikipedia*, 30-Mar-2022. [Online]. Available: https://en.wikipedia.org/wiki/Digital\_biquad\_filter. [Accessed: 01-Apr-2022].

Appendix

1. Process Samples C Program

#include "Talkthrough.h"

#include <stdfix.h>

//--------------------------------------------------------------------------//

// Function:    Process\_Data()                                              //

//                                                                          //

// Description: This function is called from inside the SPORT0 ISR every    //

//              time a complete audio frame has been received. The new      //

//              input samples can be found in the variables iChannel0LeftIn,//

//              iChannel0RightIn, iChannel1LeftIn and iChannel1RightIn      //

//              respectively. The processed data should be stored in        //

//              iChannel0LeftOut, iChannel0RightOut, iChannel1LeftOut,      //

//              iChannel1RightOut, iChannel2LeftOut and iChannel2RightOut   //

//              respectively.                                               //

//--------------------------------------------------------------------------//

void Process\_Data(void)

{

    /\* Old Program, merely passthrough. Holding onto it for debugging purposes.

    iChannel0LeftOut = iChannel0LeftIn;

    iChannel0RightOut = iChannel0RightIn;

    iChannel1LeftOut = iChannel1LeftIn;

    iChannel1RightOut = iChannel1RightIn;\*/

    //Constant coefficents for the difference equations

    static const fract b10 = 0.00282002964871r;

    static const fract b11 = 0.00564005929742r;

    static const fract b12 = 0.00282002964871r;

    static const fract a11 = -0.87502859678969r;

    static const fract a12 = 0.49062148093976r;

    static const fract b20 =  0.04762576543440r;

    static const fract b21 = -0.09525153086881r;

    static const fract b22 = 0.04762576543440r;

    static const fract a21 = -0.89065625680738r;

    static const fract a22 = 0.49121467651756r;

    //Variables for the delayed values

    static fract x11 = 0r;

    static fract x12 = 0r;

    static fract y11 = 0r;

    static fract y12 = 0r;

    static fract x21 = 0r;

    static fract x22 = 0r;

    static fract y21 = 0r;

    static fract y22 = 0r;

    fract in0 = rbits((short)(iChannel0LeftIn>>16));

    accum in1 = 2\*(b10\*in0) + 2\*(b11\*x11) + 2\*(b12\*x12) - 2\*(a11\*y11) - 2\*(a12\*y12);

    in1 = (in1 > FRACT\_MAX) ? FRACT\_MAX : in1;

    in1 = (in1 < FRACT\_MIN) ? FRACT\_MIN : in1;

    accum sum = 2\*(b20\*((fract)in1)) + 2\*(b21\*x21) + 2\*(b22\*x22) - 2\*(a21\*y21) - 2\*(a22\*y22);

    //Clip the signal

    sum = (sum > FRACT\_MAX) ? FRACT\_MAX : sum;

    sum = (sum < FRACT\_MIN) ? FRACT\_MIN : sum;

    //Shif  o7

    x12 = x11;

    x11 = in0;

    y12 = y11;

    y11 = (fract)in1;

    x22 = x21;

    x21 = (fract)in1;

    y22 = y21;

    y21 = (fract)sum;

    iChannel0LeftOut = bitslr((long fract)sum);

    iChannel0RightOut = iChannel0LeftIn;

}

B.

% IIR Filter Design Code

% 03.29.22

% Ryan Colon

% might need to use tf2sos I think

% Specifications:

% Bandpass IIR, Chebyshev Type I

% 4 poles

% center freq: 1.2 kHz

% passband bandwidth: 101.25 Hz, X = 0.05(M+42)^2

% passband ripple: 1.5 dB

% gain at max points: 5 dB

% Sample Rate: 16 kHz

clear

clc

M = 3; %March is my birthday month

bandWidth = 0.05 \* (M + 42)^2; %Resultant passband Bandwith in Hz

centerFreq = 1.2e3;

passbandFreqs = [(centerFreq - (bandWidth / 2)) (centerFreq + (bandWidth / 2))];

order = 4;

passbandRipple = 1.5; %dB

sampleRate = 16e3;

%Order is divided by 2 because in the matlab algorith

%it represents 1/2 the order for bandpass filters

[state input output feedthrough] = cheby1(order / 2, passbandRipple, (passbandFreqs / (sampleRate/2)));

%Get the second order system and transfer function forms

sos = ss2sos(state, input, output, feedthrough);

%I want the max gain to be 5 dB so convert 5 dB

%to V/V and multiply the numerator by it

gain = 10^(5/20);

num1 = 14\*sqrt(gain)\*(sos(1, 1:3));

denom1 = sos(1, 4:6);

num2 = (1/14)\*sqrt(gain)\*(sos(2, 1:3));

denom2 = sos(2, 4:6);

d1 = dfilt.df1(num1, denom1);

d2 = dfilt.df2(num2, denom2);

Hd = dfilt.cascade(d1, d2);

freqz(Hd)

% freqz(d1)

% freqz(d2)

%Write the coefficients to a file for submission

FID = fopen('RyanColon\_IIR\_Coefficients\_Rev1.txt','w');

if FID > 0

fprintf(FID, 'Difference Equation 1: ')

fprintf(FID, '%.14fr %.14fr %.14fr %.14fr %.14fr\n', num1(1)/2, num1(2)/2, num1(3)/2, denom1(2)/2, denom1(3)/2);

fprintf(FID, 'Difference Equation 2: ')

fprintf(FID, '%.14fr %.14fr %.14fr %.14fr %.14fr\n', num2(1)/2, num2(2)/2, num2(3)/2, denom2(2)/2, denom2(3)/2);

end