usepackage[hdivide=1in,*,1in, vdivide=1in,*,1in,]geometry

Finite Impulse Response Filter

ECEN 498: Real-Time Digital Signal Processing
University of Nebraska-Lincoln
Department of Electrical and Computer Engineering

Landon Burk, Evan Cornwell, Tess Jisa

Contents

1 Introduction

Finite Impulse Response (FIR) filters are, as their name implies, digital filters which have an impulse response with a finite duration. FIR filters have a few properties which make them preferable to Infinite Impulse Response filters: FIR filters require no feedback, they are inherently stable, and they can easily have a linear phase.

FIR filters are applied to a signal using convolution. Convolution is a mathematical operation between two functions that results in another function that expresses how the shape of one affects the other. In digital signal processing, convolution is computed using Equation (1), where "h(m)" is the filter coefficient, "x(n-m)" is the signal which is being filtered.

$$y = \sum_{m=0}^{M} h(m) * x(n-m) \tag{1}$$

In embedded applications of digital signal processing, convolution is executed using a series of multiplies and accumulates, in which the coefficients of the filter are multiplied with the samples of a signal and summed together to produce one output signal. This is achieved using what is called a delay line, an array in memory of equal length as the filter which is used to store the current and previous samples from the signal.

For this project, an FIR filter with 62 taps will be designed using Matlab, and implemented on the EZDSPC5502. Steps will be taken to optimize the functions such as: using built-in optimization functions included with the C compiler, intrinsic functions, and other keywords useful in minimizing cycle counts. Results of the implementation will be compared to ideal results generated the mean squared error function in MATLAB. Finally, the program will be used to filter an audio signal with another filter designed specifically to filter out high frequencies.

The memory allocation for this project was 30% DARAM and 50% VECs. The percent CPU used was 5.98%.

2 Program Description

2.1 FIR Filter Design

The filter coefficients needed for convolution were exported from MATLAB. The filter order was specified as 61. A smaller passband of 1500 Hz was chosen along with a wide transition band and a stopband frequency of 3500. This combination allowed for a stopband attenuation of 50dB in the first lobe and more than 60dB in the lobes following. A hamming window with these parameters was made and the 62 coefficients were saved in an array that was copied into the main code. See Figure 1.

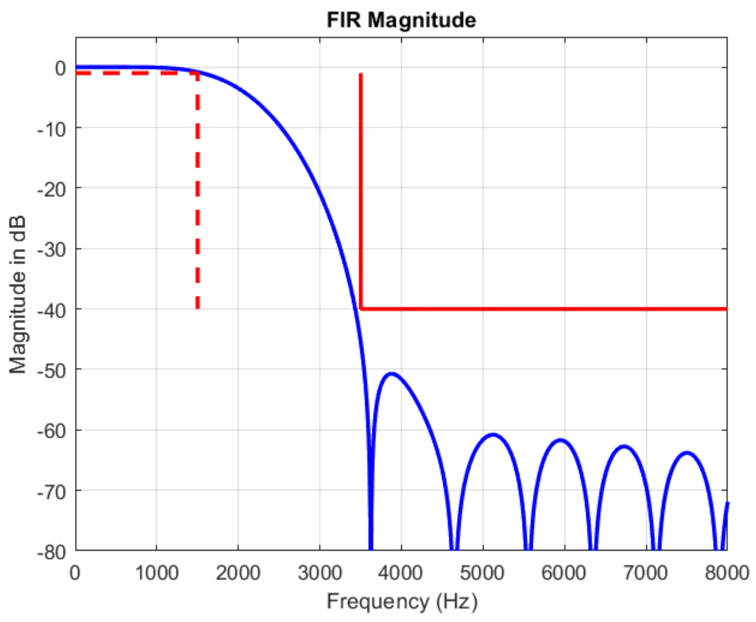

Figure 1: Filter Magnitude Response

2.2 FIR Filter Function

The filter coefficients acquired from the MATLAB filter design were used in the C program implementation of the FIR filter. The filter function was created to take in an input and filter coefficients, do the convolution, and then output the result. A delay line array was the chosen mechanism to do this. The delay line was the size of the FIR filter and it took in one input sample at a time. When the delay line filled up, a loop was done to shift the oldest input value out and shift the newest value in. For each iteration of a loop, the convolution (performed by the _smac intrinsic) of the filter coefficients and the input samples contained in the delay line was taken, and the output was spit out, one sample at a time. This was performed until there were no more samples to be taken in.

2.3 Filter Performance

Intrinsics and the restrict keyword were used in the original writing of the code so there was not much to add when it came time to get the cycle count down. Initially the cycle count for one iteration of the filter was 3766 cycles. The build settings were changed for the `myFIR.c` file to have debug settings off and an optimization level of 0 for 2821 cycles. The optimization was changed to 1 and the function took 2998 cycles. Optimization levels 2 and 3 each took 1960 cycles.

That being said, this filter could be used to process input samples in real time. One can prove this by taking the number of cycles to run one iteration of the filter (1960) divided by the speed of the DSP processor clock (300MHz): $\frac{1960 cycles}{300{,}000{,}000 cycles/second} = 0.00000653s = 6.53\mu s$. This is quite the trivial amount of time with respect to a real time audio signal, so therefore, this filter can be used to process audio in real time. Since the CPU usage is 5.98%, the maximum possible real-time filter coefficients would be 62/5.98% or about 1036.

A capture of the input to the filter and the output is shown in Figure 2. The yellow wave is a combination of two sine waves at 1kHz and 4kHz. The blue wave is the wave after being run through the DSP board lowpass filter. One can clearly see that the blue wave is a (mostly) clean sine wave, with no residual content from the 4kHz wave. Also note that it took a little bit of time for the filter to take effect, as expected.

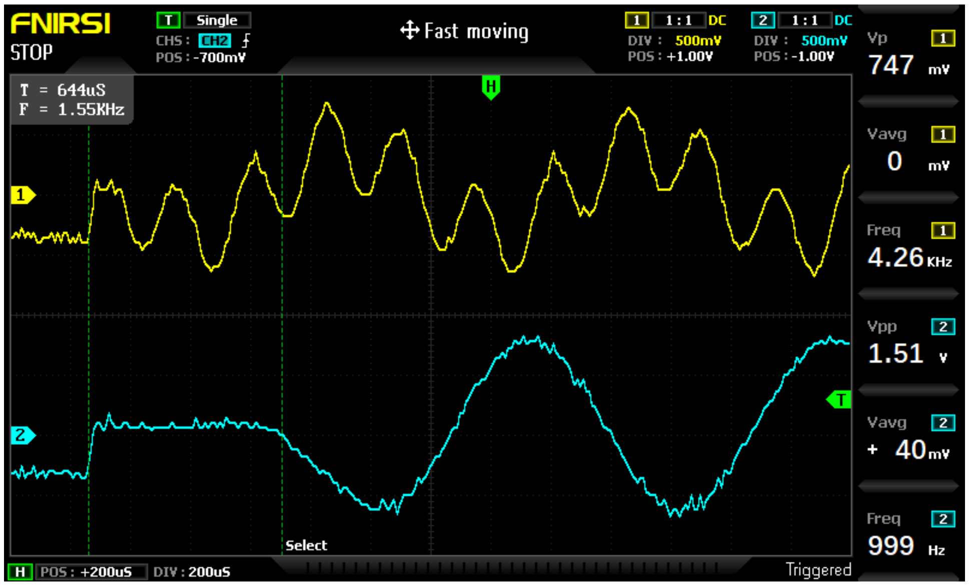

Figure 2: Filtered 1khz Sine Wave

2.4 MATLAB Filter Comparision

A MATLAB script was created to perform a filtering process equivalent to that of the DSP board, and the output sine waves from both filter implementations were compared. Note that the filter had a delay of 31 samples, because the order was 62 (filter delay = filter order/2). One can look at Figure 3 to see that the output of the MATLAB looks identical to that of the DSP board.

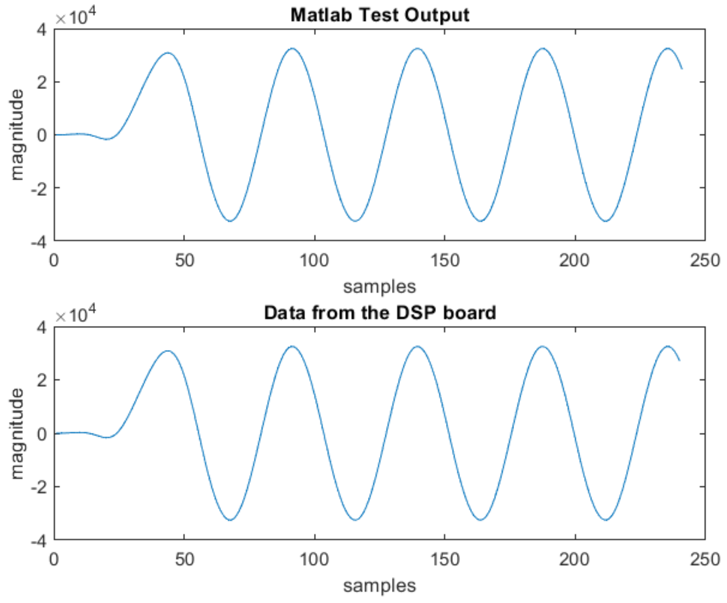

Figure 3: MATLAB Filtered Sinusoid and DSP Filtered Sinusoid

To compare these two filter implementations further, the mean squared error (MSE) of the filtered signals was taken. The average mean squared error acquired was 5.8878, which is not ideal. Efforts were taken to reduce the MSE value such as using integer values for calculations and rounding the sine wave values, but this did not affect much. Even so, the filters were quite similar to each other, and their performance was almost exactly identical. One can see a graph of the difference between the two signals in Figure 4.

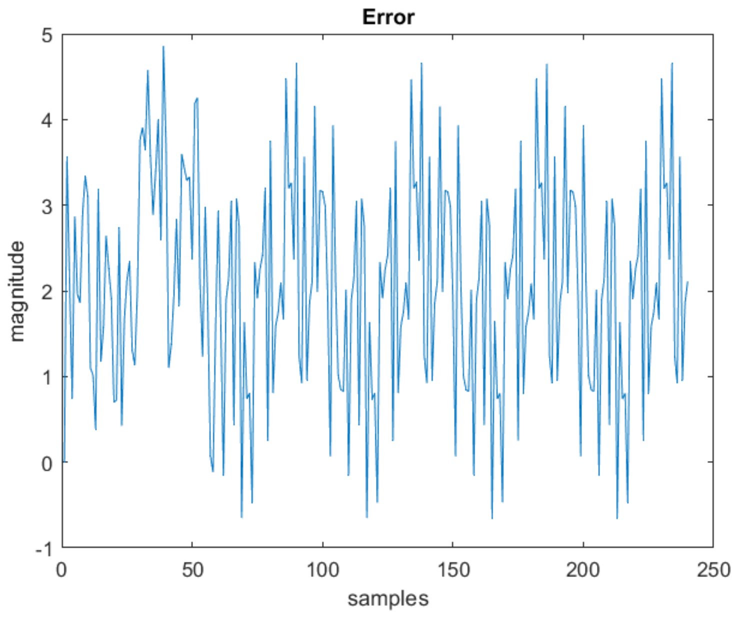

Figure 4: Difference Between DSP Filtered Signal and MATLAB Signal

3 Summary

All in all, this project was a success. A simple lowpass FIR filter was successfully explored and designed in a way that allows an input signal to be filtered in real time (now that's pretty neat!) Concepts like the delay line were introduced, where one can process and filter input samples one sample at a time. In addition, C intrinsics were used to make this implementation as quick and smooth as possible on the DSP board. After implementing the lowpass FIR filter on the DSP board to filter a simple sinusoid, the result of this was compared to a MATLAB implementation where the same exact signal was filtered. The MATLAB implementation confirmed that the DSP board did a good job of filtering the input signal, and with not much delay at all. Overall, the concept of real time FIR filtering was established and will prove to be quite useful in future projects in this class.

4 Appendix

```c
/////////////////////////////////////////////////////
//                                                 //
//  Project 1: Finite Impulse Response Filter      //
//                                                 //
//  By: Landon Burk, Tess Jisa, Evan Cornwell      //
//                                                 //
//  Created: 2023-02-15                            //
//                                                 //
//  Last Revision: 2023-02-17                      //
//                                                 //
/////////////////////////////////////////////////////

#include "stdio.h"
#include "stdint.h"
#include "ezdsp5502.h"
#include "ezdsp5502_i2cgpio.h"
#include "ezdsp5502_mcbsp.h"
#include "myFIR.h"
#include "testVector.h"
#include "demo_filt.h"

#define NH (68) //change to number of coeffs in filter being used


extern Int16 aic3204_setup( );
extern void aic3204_process(void);
void aic3204_output_sample(int16_t left, int16_t right);


const int16_t fir1Coeffs[62] =
{
        12,     20,     28,     37,     46,     54,     56,     51,     35,
                 6,    -37,    -95,   -162,   -234,   -300,   -351,
      -373,   -356,   -286,   -157,     37,    294,    609,    969,   1358,
          1754,   2135,   2475,   2754,   2951,   3053,   3053,
      2951,   2754,   2475,   2135,   1754,   1358,    969,    609,    294,
              37,   -157,   -286,   -356,   -373,   -351,   -300,
      -234,   -162,    -95,    -37,      6,     35,     51,     56,     54,
              46,     37,     28,     20,     12,
};
const int16_t demoFilter[];

void main( void )
{
    /* Initialize BSL */
    EZDSP5502_init( );

    /* Set to McBSP1 mode */
    EZDSP5502_I2CGPIO_writeLine(  BSP_SEL1, LOW );
    EZDSP5502_I2CGPIO_configLine( BSP_SEL1, OUT );

    /* Enable McBSP1 */
    EZDSP5502_I2CGPIO_configLine( BSP_SEL1_ENn, OUT );
    EZDSP5502_I2CGPIO_writeLine(  BSP_SEL1_ENn, LOW );

    aic3204_setup();

    // Pointer + Variable Declaration
    const int16_t* restrict fir1Coeffsptr;
    const int16_t* restrict demoFilterptr;
    int16_t* restrict fir1_delayLineptr;
    int16_t* testVectorOutput;
    int16_t* testVectorptr;
    int16_t fir1_delayLine[NH];
    Int16 dataLeft;


    //Passing Arrays to Pointers
    testVectorptr=testVector;
    fir1Coeffsptr=fir1Coeffs;
    demoFilterptr=demoFilter;

    volatile int k,i;
    //fill delay line with zero
    for(i=0; i<NH; i++)
        fir1_delayLine[i]=0;

    //Assign delayline to pointer
    fir1_delayLineptr = fir1_delayLine;

    //Dummy variable for watching output
    int16_t datOutput[240];
    for(i=0;i<240;i++)
        datOutput[i]=0;

   //Start with left channel input at 0
   dataLeft = 0;
   while(1)
   {    //Filter and output signal (also output original signal for fun)
        for( k=0; k < INPUT_TEST_VECTOR_LENGTH ; k++)
        {
            EZDSP5502_MCBSP_read(&dataLeft);

            myFIR(//&testVectorptr[k], //function takes a vector or realtime
                input and outputs to testVectorOutput
                    (int16_t*)&dataLeft,
                    //fir1Coeffsptr,
                    demoFilterptr,
                    testVectorOutput,
                    fir1_delayLineptr,
                    1,
                    NH);
            datOutput[k] = *testVectorOutput;

            //aic3204_output_sample(*testVectorOutput, *testVector);
            aic3204_output_sample(*testVectorOutput, (int16_t)&dataLeft);

        }
        k=0;//dummy instruction for debugging
    }
}
```

Listing 1: Main Code

```
/*
```

```c
 * myFIR.c
 *
 *  Created on: Feb 15, 2023
 *      Author: Landon Burk, Tess Jisa, Evan Cornwell
 */

#include "stdio.h"
#include "stdint.h"



void myFIR(int16_t* x, const int16_t* restrict  h, int16_t* y, int16_t*
    restrict delayLine, uint16_t nx, uint16_t nh)
{
    //copy new samples into delayLine
    int40_t sum = 0;
    uint16_t i = 0;

    for(i=nh-1; i>0;i--)
        delayLine[i]=delayLine[i-1];

    delayLine[0] = *x;

    //Filter

    #pragma MUST_ITERATE(2,,2) //commented out for the first part
    for(i = 0; i < nh; i++)
        sum = _smac(sum, (int)delayLine[i], (int)h[i]);

    //output results

    *y = sum>>15;

    //in debug, tools, save memory can specify that we want output buffer saved

}
```

Listing 2: FIR Filter Function

```matlab
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                   %
%  Linear Phase Filter              %
%                                   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear all; close all;

N = 61; % order of FIR filter
Fs = 48000;
Fp = 1500;
Fsby2 = Fs/2;
Rs = 40; %dB
Rp = 1; %dB
t = 0:1/Fs:240/Fs;

```

```
wp = Fp/(48000/2);

fdco = 1.5*Fp;
%fst=Fp*1.6;
fst = 3500;

Fpass = [0 Fp Fp];
Rpass = [-Rp -Rp -Rs];

Fstop = [fst fst Fsby2];
Rstop = [-Rp -Rs -Rs];

ws = fdco/Fsby2; %vector of stop band frequencies

% win = window(@hann,N+1);
win = window(@hamming,N+1);
b3 = fir1(N, ws,win); %create filter coefficients

% rounded for integer math on the board
b3Good = round(b3*32767);

str = sprintf('int16_t data[] = \n{');

for i = 1:16:length(b3Good)
    strln = sprintf('%6d, ', b3Good(i:min(i+15,length(b3Good))));
    str = [str sprintf('\n     ') strln];
end
str = [str(1:end-1) sprintf('\n};')];
clipboard('copy', str);


[H3,w3] = freqz(b3,1,2^16,Fs); % generate freqeuency response

% plot filter resopnse
plot(w3, 20*log10(abs(H3)), 'b' ...
    ,Fpass,Rpass,'r--',...
    Fstop,Rstop,'r','LineWidth',2)
axis([0 8000 -2*Rs,5])
grid
xlabel('Frequency (Hz)')
ylabel('Magnitude in dB')
title('FIR Magnitude')


% COMPARING A MATLAB SIGNAL FILTERED TO THE ACTUAL DSP FIR
% datOutput(signed) file. Done on 2/19/23 :)
xbad = sin(2*pi*1000*t) + sin(2*pi*4000*t);
y = filter(b3Good, 1, xbad);
```

```

% data from the DSP filter
y_struct = importdata('datOutput(singedint).dat', '', 1);
y_CC15 = y_struct.data;

figure
subplot(2,1,1)
plot(y)
title('Matlab test output')
xlabel('samples')
ylabel('magnitude')

subplot(2,1,2)
plot(y_CC15);
title('Data from the DSP board')
xlabel('samples')
ylabel('magnitude')


% Messi
y_MSE = mean((round(y(1:240)) - y_CC15').^2, 'all');

figure
plot(y(1:240) - y_CC15')
title('Error')
xlabel('samples')
ylabel('magnitude')





% %plot phase response
% pause
% plot(w3, angle(H3),'LineWidth',2)
% xlabel('frequency')
% ylabel('Phase (radians)')
% title('Linear Phase Phase')
% axis([0 1 -3 3])
% grid
% pause
%
% %plot Group Delay
% [grpL,wg] = grpdelay(b3,1,N);
% plot(wg, grpL,'LineWidth',2)
% xlabel('frequency')
% ylabel('Group Delay (Samples)')
% title('Linear Group Delay')
```

Listing 3: MATLAB Code