Real-time FIR Filtering on a TI C5502 DSP Core

Real Time Digital Signal Processing - University of Nebraska

Daniel Shchur
62096122

Contents

List of Figures

List of source codes

1 Introduction

Finite Impulse Response filters are a widely used filter design and as their name implies, they have a finite impulse response which makes them desirable for many applications. They operate by specifying filtering coefficients which are convolved with an input signal. This convolution introduces a delay to the input signal, but also removes or accentuates the desired signals. The implementation of a low pass filter for real time applications by use of a real time DSP core is explored.

2 Low Pass FIR

2.1 Design

The filter needs to attenuate a 4000 Hz signal while leaving a 1000 Hz signal in tact. An equiripple FIR design is used to give a high degree of attenuation on the desired frequency while leaving the other signals more or less in tact. The choice to use the equiripple means that attenuated signals will ripple in amplitude, but at a very low magnitude. There is also no ripple in the kept frequencies. The filter was designed to a length of 60 coefficients as per requirements with a passband frequency of 1100 Hz, a stopband frequency of 3955 Hz, and equal weighting on both bands. The filter's frequency response chart is shown in Figure 1. As can be seen from the plot, a stop band attenuation of more than 150 dB is achieved.

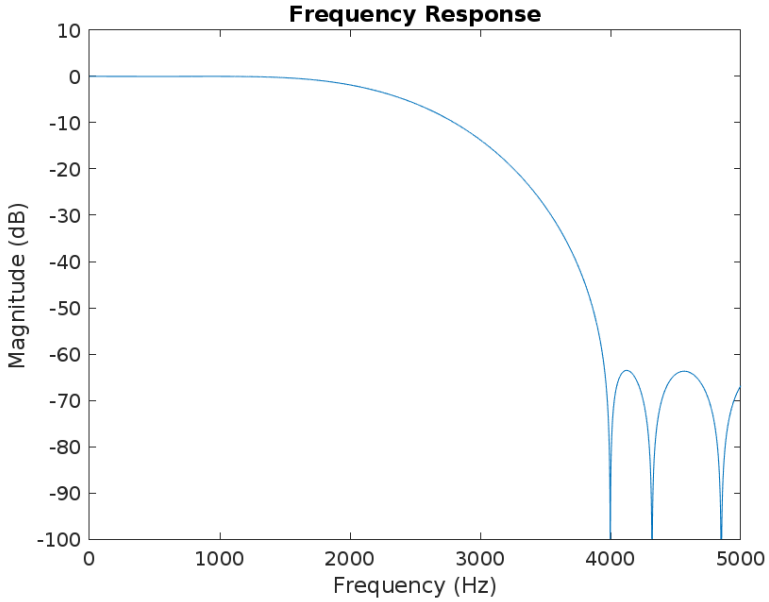

Figure 1: Frequency response of the Low Pass FIR filter

FIR filters have constant delay applied to the input signal based on the number of filter coefficients. This filter has a delay of 30 samples based on its order.

2.2 Implementation

To implement the filter, a function was created which takes in the input signal, filter coefficients, and a delay line to aid in the convolution process. Within the function, the new sample is added

to the front of the delay line, then the next output sample is calculated by taking a convolution of the filter coefficients with the current delay line. This process is repeated for every sample within the given input vector before being returned back. When preforming real time processing, simply a single sample can be input which allows for continuous filtering given the delay line persists.

Data was kept in signed 16 bit integer format with a Q0.15 fixed point specification. When performing the convolution step, a 32 bit signed integer was used to hold the convolution output which was then bit shifted down by 16 bits, keeping the most significant bits.

2.3 Accuracy

Given some truncation error can occur within the real implementation, it was necessary to see how accurate the implemented filter was to the expected design. An input test vector was filtered and then the output dumped for analysis and compared by use of mean squared error. Values were kept in their original 16 bit integer format. The resulting error of the filter was 0.0791. This error is due to when the filter is off by a value of 1 from the ideal. However, this presents a mostly negligible difference difference from expected considering the range of values in signed 16 bit is between -32768 and 32767. This could be down to implementation details for the DSP core where some numbers are truncated in some places. A rounding operation was performed on the accumulated result before shifting the value into a 16 bit form. The rounding had a positive impact on the error reducing it from 0.49 to the current value. Figure 2 shows the raw difference between the real and expected output for 100 samples. The difference never goes above or below 1 through out all 1024 samples.

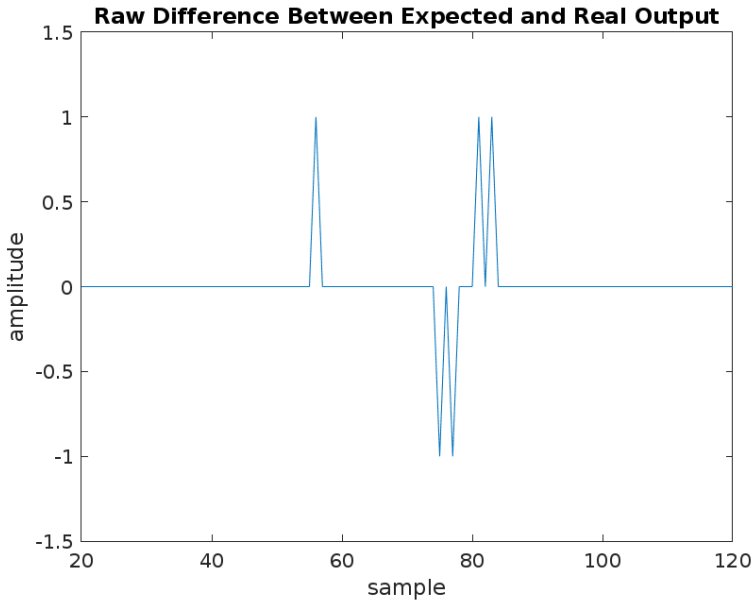

Figure 2: Raw difference of expected and actual filters

Figure 3 shows the original signal on top with the filtered outcome below. The bottom graph shows both real and expected filters overlapped over each other. The filters are shifted by 30 samples to align them with the original signal.

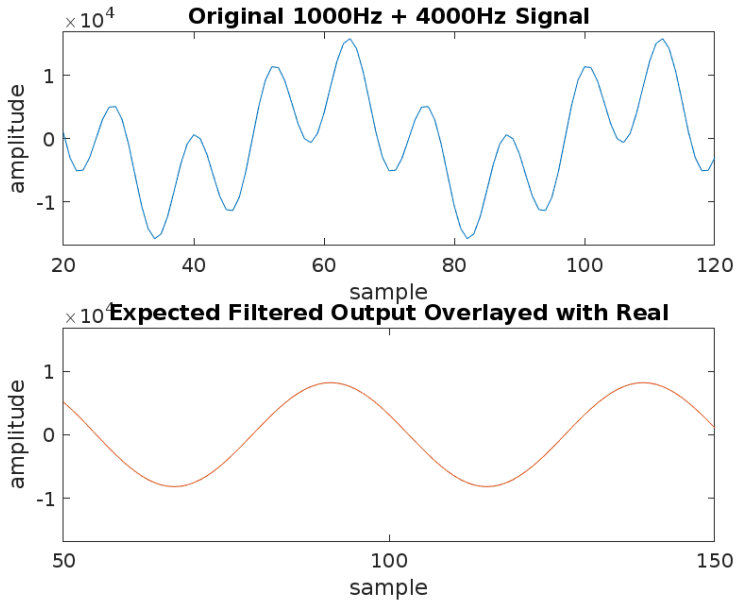

Figure 3: Original and filtered signal

3 Efficiency

Convolutions are a costly operation, especially when it comes to FIR filters as steep filters tend to have many coefficients which means many multiply and accumulate operations. Because of this, this DSP core comes with "intrinsics", or hardware optimizations related to signal processing to keep these operations quick. Two intrinsics were utilized to speed up operation of the filtering process, a multiply and accumulate as well as a 32 to 16 bit round operation. Even with these in play, a non optimized filtering operation takes about 2645 clock cycles per sample to complete or about 42% of the processor. This puts it very close to being out of a real-time range of operation as a stereo sample write takes 6251 cycles to complete. That would need to include a read, filter, and write step for each channel which would put the cpu at nearly 100% usage. If the codec were to be set up to be mono, this would give it enough head room to comfortably perform filtering.

Optimizing the code using the C compiler optimizations gave very positive gains in performance. O0 and O1 optimizations give a nearly identical performance of 1604 and 1598 cycles to complete a single sample filter. This is a near doubling of performance to no optimizations. O2 and O3 optimizations had no discernable difference but brought performance to a very fast 155 cycles per filtering operation. At 155 cycles real time filtering is without a doubt possible since, as explained previously, it takes 6251 cycles to push a sample onto the audio codec to maintain a constant sampling rate, which gives plenty of room to work with at 155 cycles. It would amount to about 2.5% processor usage.

Based on the O3 optimized version of the filter function, it can be extrapolated that the the number cycles it would take to filter a signal based on number of coefficients and samples can be given by the following equation:

$$cycles = 2.58333 * num\_of\_Coeff * num\_of\_samples$$

If we were to take a naive calculation and set the target cycles to be the sampling rate of the codec, then a theoretical ideal maximum of 2419 coefficients can be used in a single sample filtering process while still maintaining a real-time status. In the real world, this would be a much lower number however because of the time it takes to both read and write from the codec, especially when considering the doubling of operations required for stereo operation.

4 Summary

Finite Impulse Response filtering was explored in real time application by utilizing the C5502 DSP core. The filtering function implemented has a fairly decent performance efficiency allowing for real time filtering of input data without distortions or delays. It was found that many compiler optimizations can be used to help improve performance as well as intrinsics which utilize hardware optimizations of the DSP core.

5 Appendix

```
64      int16_t outputL, outputR, inputL, inputR;
65      int16_t delayLineL[LPF_COEFF_LEN], delayLineR[LPF_COEFF_LEN];
66
67      // Initialize the delay lines
68      memset(delayLineL, 0, LPF_COEFF_LEN);
69      memset(delayLineR, 0, LPF_COEFF_LEN);
70
71      while(1)
72      {
73          // Left Channel
74          inputL = read_sample();
75          myfir((const int16_t *) &inputL, lpfCoeff, &outputL, delayLineL, 1,
            ↪  LPF_COEFF_LEN);
76          play_sample(outputL, 2);
77
78          // Right Channel
79          inputR = read_sample();
80          myfir((const int16_t *) &inputR, lpfCoeff, &outputR, delayLineR, 1,
            ↪  LPF_COEFF_LEN);
81          play_sample(outputR, 2);
82      }
```

Listing 1: Main read, filter, and write operation

```
#include "myfir.h"

const int16_t lpfCoeff[] =
{
       -12,      0,      7,     18,     34,     52,     70,     84,     88,
        ↪  77,     46,     -7,    -80,   -170,   -268,   -360,
      -428,   -456,   -423,   -314,   -120,    165,    533,    969,   1450,
       ↪  1945,   2419,   2834,   3159,   3365,   3436,   3365,
       3159,   2834,   2419,   1945,   1450,    969,    533,    165,   -120,
       ↪  -314,   -423,   -456,   -428,   -360,   -268,   -170,
        -80,     -7,     46,     77,     88,     84,     70,     52,     34,
        ↪  18,      7,      0,    -12,
};

void myfir(const int16_t* input, const int16_t* filterCoeffs, int16_t* output,
           int16_t* delayLine, uint16_t numberOfInputSamples,
           uint16_t numberOfFilterCoeffs)
{
    int i, j, k;
    int32_t sum;

    for (i = 0; i < numberOfInputSamples; i++)
    {
        // update delay line with current sample
        for (k = numberOfFilterCoeffs-1; k > 0; k--)
        {
            delayLine[k] = delayLine[k-1];
        }
        delayLine[0] = input[i];
        // Perform filter
        sum = 0;
        for (j = 0; j < numberOfFilterCoeffs; j++)
        {
            // main filter loop y[n] += x[n] * h0 + x[n-1]*h1 + ...
            sum = _smac(sum, delayLine[j], filterCoeffs[j]);
        }
        // update output
        output[i] = _rnd(sum) >> 16;
    }
}
```

Listing 2: FIR filtering implementation

```c
/*
 * myfir.h
 *
 */

#ifndef MYFIR_H_
#define MYFIR_H_

#include "stdint.h"

#define LPF_COEFF_LEN 61

void myfir(const int16_t* input, const int16_t* filterCoeffs, int16_t* output,
           int16_t* delayLine, uint16_t numberOfInputSamples,
           uint16_t numberOfFilterCoeffs);

#endif /* MYFIR_H_ */
```

Listing 3: FIR filtering header