Report: Lab 4

EE3-19 Real-time Digital Signal Processing

Tayyab Bhandari {01334547}, **Timo Thans** {01668411},
Department of Electrical and Electronic Engineering, Imperial College London

I. Introduction

The goal of this lab was to design a real time Finite Impulse Response (FIR) filter. We have done this by retrieving the filter coefficients from Matlab and implementing this using the C6713 DSK system in C. The implementation of the filter in real time requires the design of a buffer. The most obvious design of this is the delay operator but as it turned out this is not the most quickest and efficient design. Therefore we have also designed two versions of circular buffers and evaluated their results in terms of clock cycles.

II. Filter Design

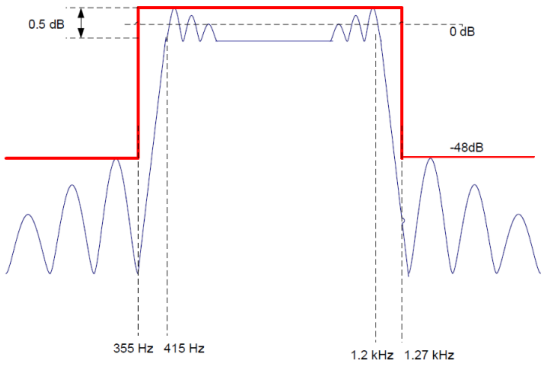


Fig. 1. FIR Filter with the required specification including passband and stopband regions

The goal was to firstly design the FIR filter in Matlab and determine the coefficients that we could use in CCS, to do this the Parks-McClellan algorithm was used which is one of the most commonly used algorithms for calculating FIR filter coefficients. Figure 1 shows the specification our filter had to meet including the passband, stopband and cut-off frequency regions.
The stop band ripple is equivalent to the minimum attenuation in the stop band, the pass band ripple indicates the maximum allowed ripple in the pass band. Listing 1 shows how we approached to meet these design specifications in Matlab.

```
%passband, stopband and cut-off frequencies
f = [355 415 1200 1270];
a = [0 1 0]; % amplitudes
fs = 8000; % sampling frequency
rp = 0.5; % passband ripple
rs = 48; % stopband ripple
% maximum ripples allowable for each band
dev = [10^(-rs/20) (10^(rp/20)-1)/(10^(rp/20)+1)
    10^(-rs/20)];
% Parks-McClellan FIR filter order estimation,
    normalised frequency band edges and amplitudes
[n,fo,ao,w] = firpmord(f,a,dev,fs);
% Parks-McClellan FIR filter design via
    calculation of filter coefficients
b = firpm(n,fo,ao,w);
% Plot the frequency response
[h, f] = freqz(b, 1 , 5000, fs);
plot(f, db(h));
% Print all the values of b on one line so we can
    copy it to CCS
for i = 1:249
    fprintf('%.8d ,', b(i))
end
```

Listing 1. Design of the FIR filter in Matlab using ***firpm*** and ***firpmord***

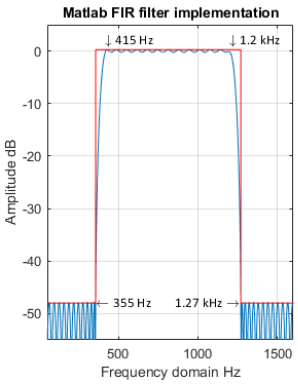


Fig. 2. Design of FIR filter in Matlab

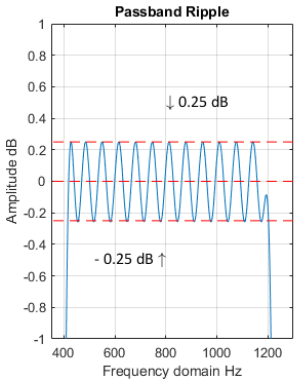


Fig. 3. Pass band ripple of FIR design

The resultant coefficients from the filter were written to a text file in a format suitable for inclusion in a C program.
Figures 2 and 3 show the frequency response of the filter and we can see that the passband and stopband specifications are met. One aspect we noticed was that we had to increase the number of evaluation points to make sure the filter was displayed correctly which explains why the number 5000 is used in the code when using the ***freqz*** function. This however could have been any arbitrary number higher than the order of the filter. In addition, the ***firpmord*** function only generates the approximate order required for the filter.

III. Non-Circular FIR filter

As figure 4 shows, the implementation of the filter can be thought of as a tapped delay line. A method of implementation would be to store all the samples in an array and each time a new sample arrives, shift all the samples and insert the new sample. We implemented this delay operator using a loop as

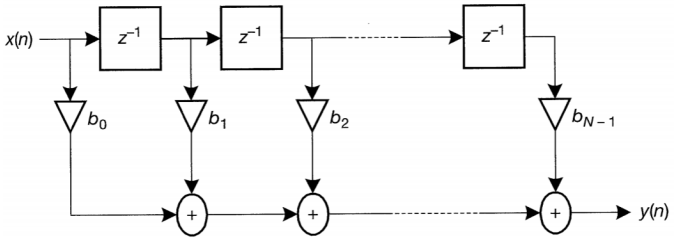


Fig. 4. Block diagram of the implementation of a digital FIR filter

shown in listing 2 where data is moved along the buffer and new samples are read into the buffer.

```
for (int j = N - 1; j > 0; j--)
{
    x[j] = x[j-1];
}
x[0] = mono_read_16Bit();
```

Listing 2. Implementation of the delay operator using an array shuffling method

Our Interrupt Service Routine (ISR) had to perform the following tasks: read the input sample from the codec, perform the delay operator, perform the convolution function, output *y(n)* to the codec. Therefore, the code from lab 3 was copied and subsequently changed so that the interrupt function included this delay buffer which in our case had a length of 249. Listing 3 shows the modified ISR and here, the global variable *y* is the output to the codec.

```
void InteruptFunc()
{
    non_circ_FIR();
    mono_write_16Bit(y);
}
```

Listing 3. The modified Interrupt Service Routine (ISR)

The *non_circ_FIR* function is explained in listing 4 and includes the delay buffer and the MAC algorithm. In the MAC algorithm, the values in the buffer are multiplied by the filter coefficients using the dot product.

```
double non_circ_FIR()
{
    double y = 0;
    int i;
    //Perform the MAC using the dot product
    for (i = 0; i < N-1; i++){
        y += b[i]*x[i];
    }
    //Move data along the buffer
    for (j = N-1; j>0; j--){
        x[j] = x[j-1]
    }
    //Put new sample into buffer
    x[0] = mono_read_16Bit();
    return y;
}
```

Listing 4. Implementation of the non circular buffer function

To test this implementation, figures 5 and 6 show what happened when we attached a sine wave to the input of the board. We had to ensure that the amplitude was small enough so no damage is done to the input of the DSK and this meant the input being rated at 2Vrms. The output showed a sinewave signal identical to the input with no harmonics or distortions which increases in amplitude around the first transition band and the opposite occurring at the next transition band.

To test the implementation at different compiler optimization

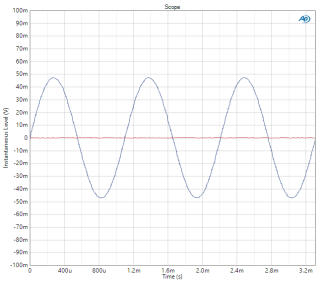


Fig. 5. Output of Scope in Passband Frequency

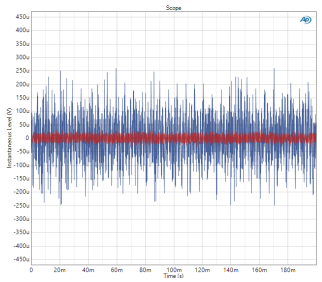


Fig. 6. Output of Scope in Stopband Frequency

levels, we measured how many instruction cycles the interrupt function takes to finish using the profiling clock method, we achieved this by using breakpoints before and after the ISR. Table 1 shows repeated measurements for each optimization

Optimisation Level	Measurement 1 (clock cycles)	Measurement 2	Measurement 3
Nothing	15968	15380	15379
-o0	12630	13149	12630
-o2	4243	3704	3703

TABLE I
ISR IN TERMS OF INSTRUCTION CYCLES WITH DIFFERENT OPTIMIZATION OPTIONS

level until the cycle count minimised so we can then use the best case for each filter implementation.

IV. Circular Buffer

The previous method using the non circular method involved shifting data in such a way so that the first sample in the array was always the newest sample and the last element in the array always the oldest. To do this, each time we got a new input sample we had to copy data. Therefore, in this method, we have a pointer to the array element which is the most current sample. Listing 5 shows the implementation of our circular buffer.

```
void circ_FIR()
{
    //Out and buffer array are global variables
    //N = 249, ptr = 248,
    out = 0;
    //Store incoming samples in the buffer
    buffer[ptr] = mono_read_16Bit();
    for (i = 0; i < N-ptr; i++){
    //Perform the convolution right of the pointer
        out += buffer[ptr+i]*b[i];
    }
    for (i=N-ptr; i<N; i++){
    //Perform the convolution left of the pointer
        out += b[i]*buffer[ptr+i-N];
    }
```

```
    //Decrement the pointer
    ptr--;
    //Reset pointer back to right of the buffer
    if (ptr < 0){
          ptr = N - 1;
     }
}
```

Listing 5. Implementation of the circular buffer function

The array we are using is called ***buffer*** and has length N which is 249 in our case and this stores the current and past samples. The pointer is set to the last element in this buffer and to store subsequent samples, it was important to decrement the pointer until we fill the array. To store samples after this point, the oldest sample to the far right of the array are no longer needed and therefore replaced with a new sample. In this way we will always have the pointer pointing to the most current sample with the samples getting older towards the right meaning in a circular fashion.

In terms of implementation of the convolution, we needed to multiply ***b[0]*** with the most current input sample (where the pointer points) and ***b[1]*** with ***buffer[ptr + 1]*** and continue this way for all the filter coefficients denoted by ***b***. In order to iterate through all the values in the buffer we developed a clever iteration algorithm. We split the buffer into values right of the pointer and left of the pointer and then iterated through both separately. This implementation required very little branching.

As for the non circular buffer, we tested this implementation at different compiler optimization levels by using the same profiling clock method. Table 2 shows repeated measurements for each optimization level. Again, we took 3 measurements

Optimisation Level	Measurement 1 (clock cycles)	Measurement 2	Measurement 3
Nothing	12454	12150	12152
-o0	11716	11411	11417
-o2	2266	1974	1974

TABLE II
ISR IN TERMS OF INSTRUCTION CYCLES WITH DIFFERENT OPTIMIZATION OPTIONS FOR CIRCULAR BUFFER

for each level to get a minimised cycle count which we can use as the best case. It is evident from table 2 that there was a significant reduction at each optimization level in the number of cycles used by the ISR which shows the increased efficiency of this implementation.

There is still room for optimization. The coefficients of a linear phase FIR filter are always symmetric by definition and so are our coefficients as is shown in figure 7. We used this property to make the ***for*** loop twice as short. The idea here was to use a longer buffer table to deal with overflow. In the extra indices the same values can be stored so that these can be used for the mirrored values of the coefficients. This allows us to shorten the ***for*** loop without having to use multiple ***if*** statements. Branching has an enormous influence on the performance of the code so it is important to keep the amount of branching as low as possible. Listing 6 displays

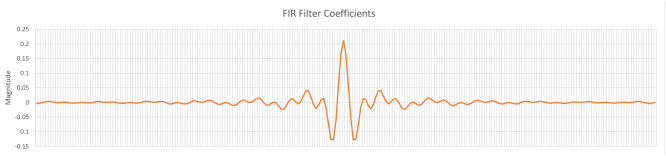


Fig. 7. Symmetry of FIR Filter Coefficients

the revised implementation of the circular buffer using the idea of symmetry of the coefficients.

```
void double_circ_FIR()
{
    //ptr2 = 2*N -1;
    output = 0; //set output to 0
    //store sample in first part of buffer
    large_buffer[ptr] = mono_read_16Bit();
    //store sample in second part of buffer
    large_buffer[ptr2] = mono_read_16Bit();
    for (i=0; i < (N-1)/2; i++){
    //use both coefficient values in the MAC
        algorithm
        output += b[i]*(large_buffer[ptr+i]+
            large_buffer[ptr2-i-1]);
    }
    //N is odd
    output += b[124]*large_buffer[ptr+124];
    ptr--;
    if (ptr < 0){
        ptr = N-1;
    }
    ptr2 = ptr + N;
}
```

Listing 6. Implementation of the improved circular buffer function

Table 3 shows the increase in performance whilst using this revised method, this is a faster implementation of the circular buffer. With the highest level of optimization, we achieved 574

Optimisation Level	Measurement 1	Measurement 2	Measurement 3
Nothing	5739	5740	5740
-o0	6376	6112	6111
-o2	574	575	574

TABLE III
ISR IN TERMS OF INSTRUCTION CYCLES WITH DIFFERENT OPTIMIZATION OPTIONS FOR IMPROVED CIRCULAR BUFFER

number of cycles run by the ISR as the best case which is a significant improvement to previous methods used.

V. Fastest Implementation

We were intrigued by the frequency response for our fastest implementation method and it was satisfying to see that the expected response from Matlab and the bode plot from the designed filter obtained from the spectrum analyser were similar in style and specification. Figure 8 displays the result of the passband and in particular the ripple, we see that the deviation in the ripple matches that of the Matlab plot.

Figure 9 and 10 display the frequency response and linear phase of our designed FIR filter using the fastest implementation method, the specification is closely matched to the expected response previously designed in Matlab, we also see

that the designed filter has linear phase which means a constant group delay.

However, there were a few differences that are important to note. Firstly, the gain has an offset of 17 dB and this is because the amplitude of the output signal is 4 times reduced in comparison with the input amplitude due to the two potential dividers at each input port of the DSK board. When we tested an all pass filter, we gathered that this is due to the measurement hardware and not our software.

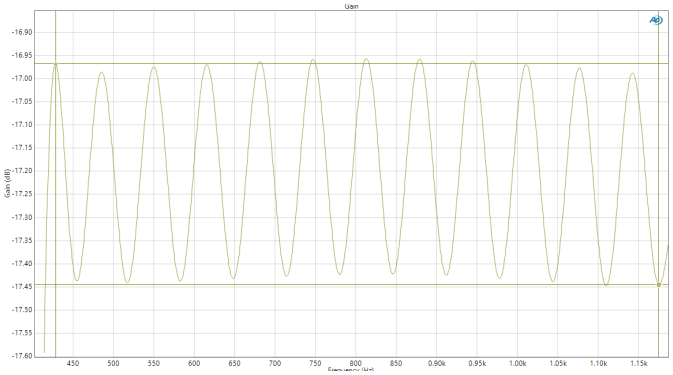


Fig. 8. Result of the passband ripple

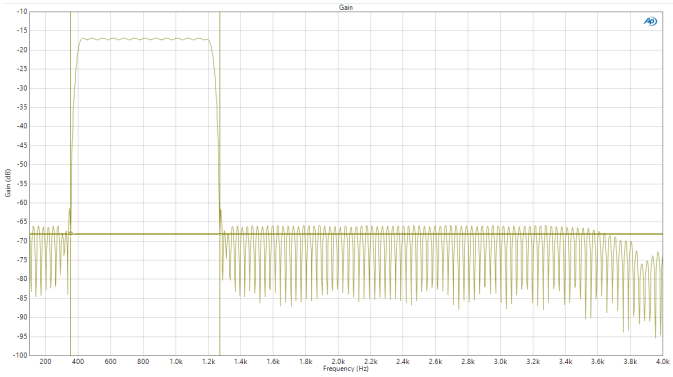


Fig. 9. Performance of the FIR filter

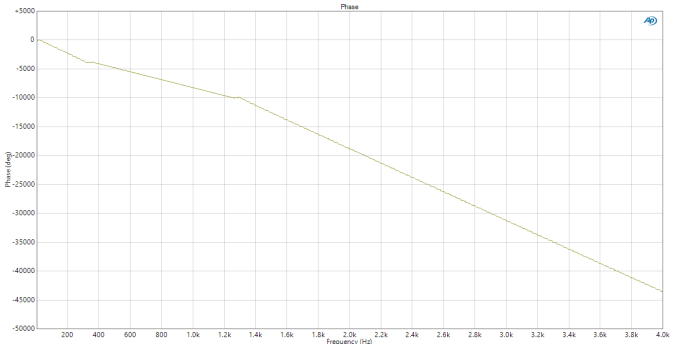
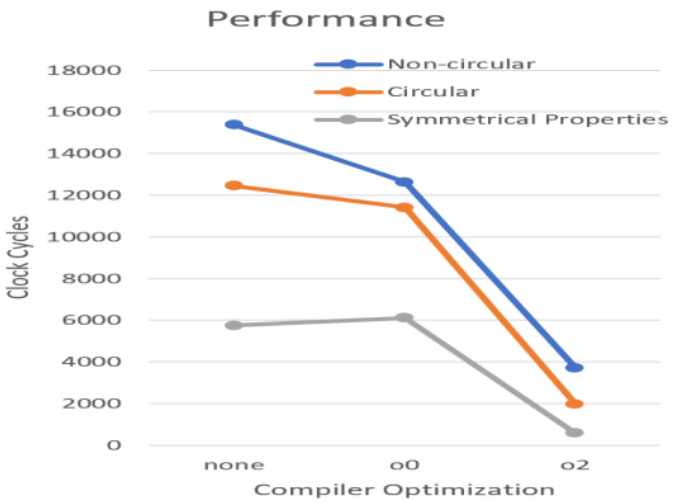


Fig. 10. Linear phase FIR filter



h

Fig. 11. Relative Performance of the Different Implementations

VI. Conclusion

In this lab we designed a specific FIR filter to match a specification and implemented this on a DSK board to test with an input signal. In general, code optimization is extremely important and therefore we have looked into three different compiler optimizations but also into three code optimizations: the (non circular) delay buffer, the circular buffer and an improved circular buffer making use of the symmetrical properties of the coefficients of the filter. This resulted in an improvement from 15380 clock cycles in the slowest case towards 574 clock cycles in the fastest case. Figure 11 is an illustration of the different implementations we used in terms of differences in performance for a visual perspective on how they differ in the number of clock cycles used.