Imperial College London

Real-Time Digital Signal Processing

Laboratory 4

Ahmad Moniri, CID: 00842685

Pranav Malhotra, CID: 00823617

Declaration: We confirm that this submission is our own work. In it, we give references and citations whenever we refer to or use the published, or unpublished, work of others. We are aware that this course is bound by penalties as set out in the College examination offences policy

Signed: Ahmad Moniri, Pranav Malhotra

April 29, 2016

Contents

1 Introduction

In laboratory 3, interrupt-driven programming was introduced. Interrupt-driven programsare more efficient than programs that poll peripherals as they have the capacity to fully utilise the computation power of the processor/embedded device. As such, all programs written for the Real-Time Digital Signal Processing course from this point on will be interrupt-driven.

In laboratory 4, Finite Impulse Response (FIR) filters are introduced. Filter implementation is discussed and efficiency of different programming techniques is considered.

This report starts by introducing the theory behind FIR filters. Following this, design of FIR filters using MATLAB is discussed. Following this an in-depth analysis of the laboratory experiment is presented. The discussion includes multiple functions written in C to implement the convolution sum required for filtering. Next, before the different implementations are compared, the compiler present in the code composer environment is studied. The compiler offers in-built optimisation options and thus understanding this is critical for efficient filtering. The filter's spectrum is also validated using the Audio Precision $APX520$ device.

2 Finite Impulse Response (FIR) Filters

A FIR filter, as the name finite suggests has an impulse response that reaches zero in a finite duration of time. This has an important implication; the memory in the system is finite. FIR filters can only be implemented digitally as analogue filters have an Infinite Impulse Response (IIR).

Equation (1) shows the general form of the difference equation that represents a FIR filter. The number M represents the order of the filter. **The order of the filter represents the amount of memory the system has.**

The difference equation can also be written as an inner product as shown in equation (2). **For a FIR filter, the filter coefficients are equivalent to impulse response of the filter[1].**

$$y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] + \cdots + b_Mx[n-M] \tag{1}$$

$$= \sum_{i=0}^{M} h[n]x[n-i] \tag{2}$$

Taking the z-transform of equation (1) returns the general form of the transfer function of a FIR filter presented in equation (3). Equation (4) corroborates the fact that the order of the filter is M[2].

$$H(z) = \frac{Y(z)}{X(z)} = h[0] + h[1]z^{-1} + \cdots + h[M]z^{-M} \tag{3}$$

$$= \frac{h[0]z^M + h[1]z^{M-1} + \cdots + h[M]}{z^M} \tag{4}$$

A zero in the transfer function causes a dip in the magnitude response of a filter and thus is not considered when stability of the filter is studied. The amount of attenuation is dependent on the distance of the zero, on the z-plane, from the unit circle. A zero placed close to the unit circle will cause greater attenuation than a zero that is placed further away from the unit circle. It is important to note that in the Real-Time Digital Signal Processing course, only real signals are considered and thus all coefficients are real. As a result, all zeroes are either real or occur in complex conjugate pairs. **The frequency response of a FIR filter with real coefficients will be symmetric about the half the sampling frequency[3].**

In addition, it is clear that the FIR filter has M poles located at the origin of the z-plane. Poles located outside the unit circle in the z-plane will result in an impulse response that grows exponentially. The system will not be Bounded-Input Bounded Output (BIBO) stable. **FIR filters are inherently stable as poles are always located at the origin.** Not having the freedom to place poles on the z-plane presents many limitations when

[1] This does not apply for an IIR filter, since the impulse response is infinite.

[2] The order of the polynomial in the denominator of the transfer function determines the order of the filter.

[3] Symmetric about half the sampling frequency also means that the frequency response of symmetric about the y-axis in the frequency domain. Half of the sampling frequency is mentioned because the MATLAB function `freqz` plots the frequency response of the filter from 0 to F_s.

a FIR filter is designed. To compensate, FIR filters have a significantly larger order than IIR filters that attain similar frequency responses. **Although FIR filters present stark limitations, in that poles are always located at the origin, they can be designed to have a linear phase response in the frequencies of interest.**

Equation (5) shows the general form of the transfer function of a FIR filter written as a convolution sum, where $h[n]$ represents the impulse response of the filter.

$$H(z) = \frac{Y(z)}{X(z)} = \sum_{n=0}^{M} h[n]z^{-n} \tag{5}$$

Equation (5) can be implemented using the signal flow graph shown in figure 1.

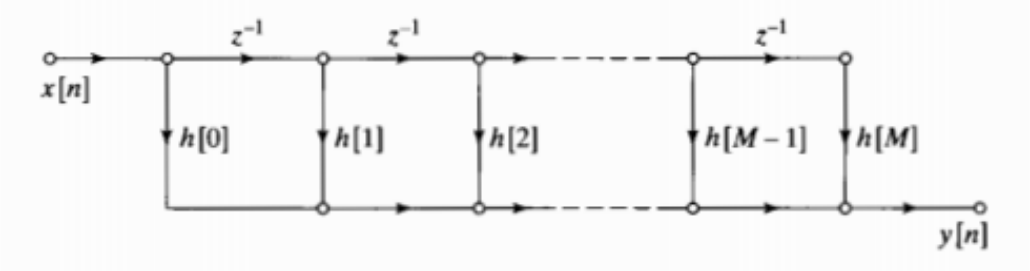

Figure 1: Direct form of FIR filter

Next, consider a filter with an impulse response that is symmetric. The symmetric relation between the coefficients of the filter is explicitly stated in equation (6), where M is the order of the filter.

$$h[n] = h[M-1-n] \tag{6}$$

The symmetric nature of the impulse allows for the following manipulations[4],

$$H(z) = \sum_{n=0}^{M} h[n]z^{-n} \tag{7}$$

$$= \sum_{n=0}^{(M-3)/2} h[n]\left(z^{-n} + z^{-(M-1-n)}\right) + h(\tfrac{M-1}{2})z^{-(M-1)/2} \tag{8}$$

Evaluating equation (8) for $z = e^{j\omega}$ will return the frequency response of the filter.

$$H(e^{j\omega}) = \sum_{n=0}^{(M-3)/2} h[n]\left(e^{-j\omega n} + e^{-j\omega(M-1-n)}\right) + h(\tfrac{M-1}{2})e^{-j\omega(M-1)/2} \tag{9}$$

$$= \sum_{n=0}^{(M-3)/2} h[n]\left(e^{+j\omega(n-(M-1)/2)} + e^{-j\omega(n-(M-1)/2)}\right)e^{-j\omega(M-1)/2} + h(\tfrac{M-1}{2})e^{-j\omega(M-1)/2} \tag{10}$$

$$= \sum_{n=0}^{(M-3)/2} \left(2h[n]cos(\omega(n - \tfrac{M-1}{2}))\right)e^{-j\omega(M-1)/2} + h(\tfrac{M-1}{2})e^{-j\omega(M-1)/2} \tag{11}$$

$$= \left(e^{-j\omega(M-1)/2}\right)\left(\sum_{n=0}^{(M-3)/2} 2h[n]cos(\omega(n - \tfrac{M-1}{2})) + h(\tfrac{M-1}{2})\right) \tag{12}$$

[4]It is assumed that the filter has an odd number of coefficients. Similar calculations can be performed for a filter with an even number of coefficients. These calculations are not presented in this report.

Based on equation (12), the magnitude response, phase response and group delay can be evaluated.

$$\left|H(e^{j\omega})\right| = \sum_{n=0}^{(M-3)/2} 2h[n]cos(\omega(n - \tfrac{M-1}{2})) + h(\tfrac{M-1}{2}) \tag{13}$$

$$\angle H(e^{j\omega}) = -\omega\frac{M-1}{2} \tag{14}$$

$$Group\ Delay = -\frac{\mathrm{d}\phi}{\mathrm{d}\omega} = \frac{M-1}{2} samples \tag{15}$$

Equation (14) shows that the phase response varies linearly with ω. Such a phase response cannot be achieved with an IIR filter. Having a linear phase response is critical is many applications such as audio signal processing. The human ear is more sensitive to changes in phase than changes in magnitude. **A linear phase response means that the relative phase of the spectral components in the signal will be constant. The signal will however be delayed as it passes through the filter.** The delay, in terms of samples, is expressed in equation (15). The delay in the time domain is calculated using, $^{(M-1)t_s}/_2$ where t_s is the sampling period.

Equation (8) can be implemented using the signal flow graph shown in figure 2. **It is clear that the the symmetric nature of the impulse response can be exploited to half the number of multiplications required for filtering.** This will be discussed in greater detail in section 6.

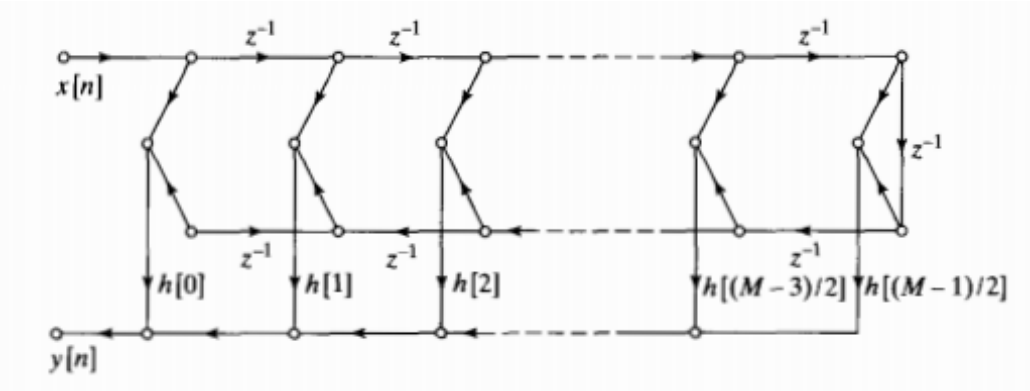

Figure 2: Canonical form of FIR filter

3 Design of Filter using MATLAB

In laboratory 4, the filter described in figure 3 has to be implemented.

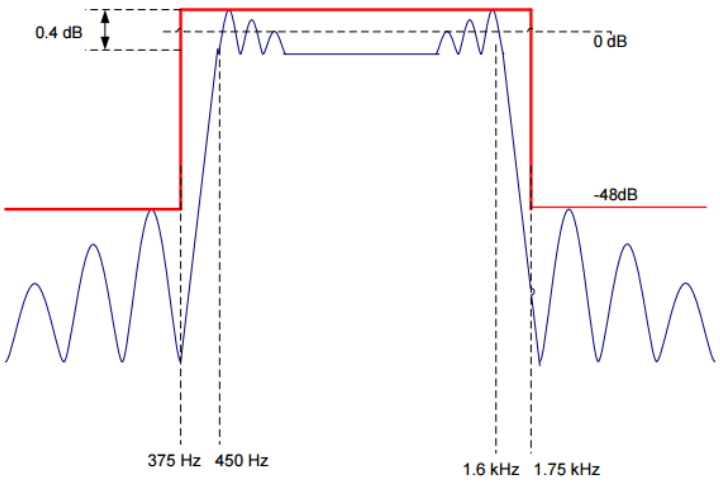

Figure 3: Filter to be implemented in laboratory 4

The filter is designed in MATLAB and the code presented in listing 1. The functions `firpmord` and `firpm` are used to design the filter. **The filter is designed using the Parks-McClellan algorithm and has an optimal frequency response. Optimality is defined as a filter that uses the least number of coefficients to just meet the specifications.** Having an equiripple design minimises the maximum error between the designed filter and the specifications at any particular frequency. Other designs algorithms are optimised to minimise other parameters. For example, the least squares design algorithm aims to minimise the total mean squared error.

```
rp = 0.4;                           % defines the pass band ripple
rs = 48;                            % defines the stop band attenuation
F_sampling = 8000;                  % defines the sampling frequency
F_cutoff = [375 450 1600 1750];     % defines the cutoff frequencies
filter_amplitudes = [0 1 0];        % defines ideal gain of each band

% function used to estimate variables need for firpm function
[N, Fo, Ao, W] = firpmord(  F_cutoff,...
                            filter_amplitudes,...
                            [10^(-rs/20) (10^(rp/20)-1)/(10^(rp/20)+1) 10^(-rs/20)],...
                            F_sampling);


[coefs, error] = firpm(N+4, Fo, Ao, W);      % order of filter is increased by 4 to meet specifications

[h,w] = freqz(coefs, [1], 2^16);
```

Listing 1: MATLAB code to generate coefficients for a FIR filter

The function `firpmord` requires the pass band deviation and stop band attenuation to be expressed in absolute terms as opposed to, in decibels. Notice that the filter is specified in figure 3 is specified in decibels and thus a conversion is necessary. The appropriate value for pass band deviation is calculated using the formula stated in equation (16) where rp represents the pass band ripple.

$$Passband\ Deviation = \frac{10^{rp/20} - 1}{10^{rp/20} + 1} \tag{16}$$

It should be noted that the function `firpmord` only generates an estimate for the order of the filter. To meet all specifications of the filter, the estimated filter order was increased by 4. Figure 4 shows the magnitude and phase response of the filter that was generated.

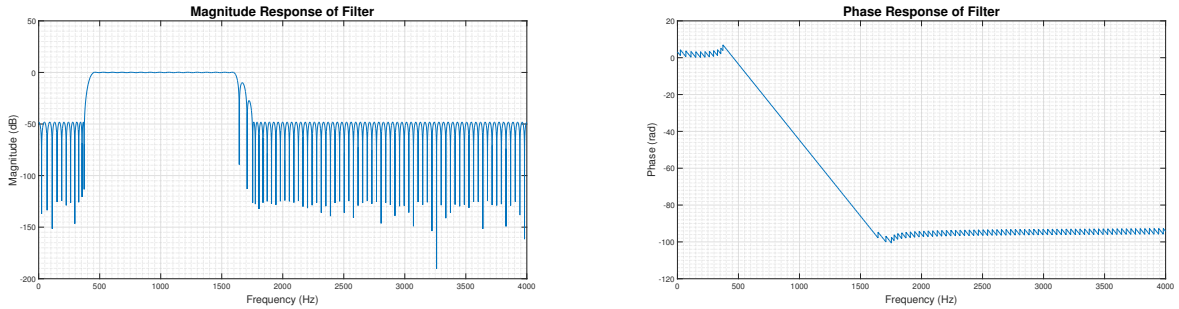

Figure 4: Frequency response of designed filter

It is clear that the filter has a linear phase response in the pass band. There is a non-linear phase response in the transition band and because zeros are extremely close to each other. Also, the zeros are not equidistant from the unit circle in the stopband. This is clear from the figure 5. **In the transition band, the dominant zeroes are placed equidistant from the unit circle to ensure a linear phase response. As discussed in section 2, an FIR filter has a linear phase response if the impulse response has the form described in equation (6).** To verify this, the impulse response of the filter is graphed in figure 5.

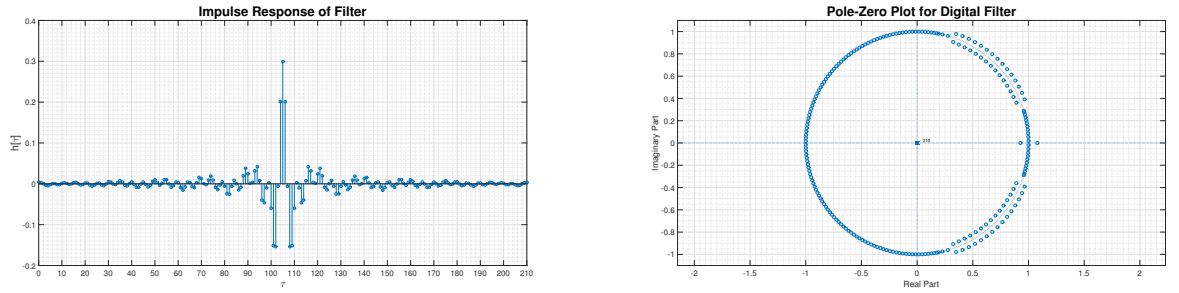

Figure 5: Impulse response of designed filter

To affirm that the filter meets all the specifications, specific frequencies are graphed in figure 6.

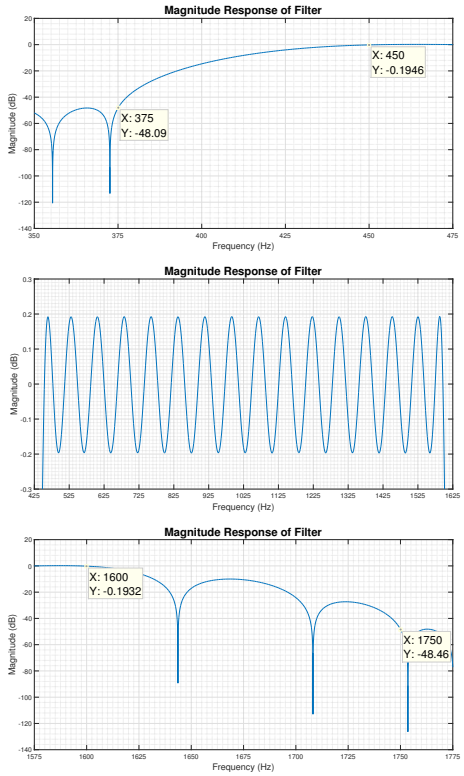

Figure 6: Graphs showing that designed filter meets specifications

4 FIR Filter Implementation: Non-Circular Buffering

The first method of implementing the filter on the $TMS320C6713$ DSK board is conceptually very simple. As discussed in section 2, a FIR filter of order M will have a memory of M. This means that a buffer of size M is required to keep track of the previous samples. In C, this buffer is implemented in the form of an array, `x`. The output is calculated by computing the inner product of the filter coefficients, which are stored in array `b`, and the past inputs as described in equation (1).

A simple buffer is implemented such that index of the array indicates the delay between the stored sample the current sample. This is depicted in figure 7.

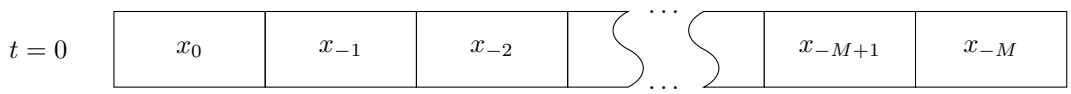

Figure 7: Simple Buffer

To maintain the simplicity of this buffer requires a great deal of overhead. Each time a new sample is read, every value in the array has to be shifted to the right. The shifting of the samples within the buffer is shown is figure 8.

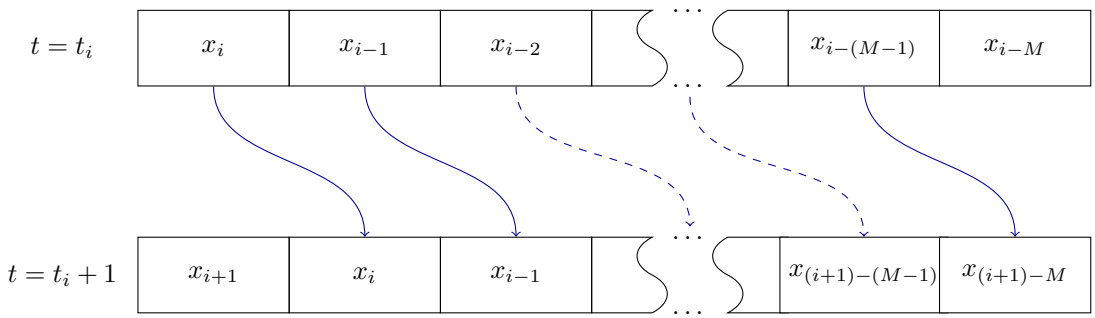

Figure 8: Complete reorganisation of the buffer each time a new sample is read

Shifting of the buffer each time a new sample is read is computationally expensive. Nonetheless, it is implemented in C and will form the baseline for performance analysis.

```
1  void ISR_filter(void){
2      short sample_in;
3      int i;
4
5      // newest sample is read into a temporary variable
6      sample_in = mono_read_16Bit();
7
8      // shifting of the buffer is performed
9      for(i=M-1; i>0; i--)
10         x[i] = x[i-1];
11
12     // newest sample is moved from the temporary variable to its correct position in buffer
13     x[0] = sample_in;
14
15     // function that performs convolution is called
16     // function returns a double and is converted to a short before being sent to output
17     mono_write_16Bit((short)non_circ_FIR());
18 }
19
20 // function that performs convolution using a non-circular buffer
21 double non_circ_FIR(void){
22     int i;
23     double y = 0;
24
25     // inner product of array containing filter coefficients and buffer containing samples
26     for(i=0; i<M; i++)
27         y += b[i]*x[i];
28
29     return y;
30 }
```

Listing 2: `non_circ_FIR`

4.1 Verification of Filter Implementation

To ensure that the code in listing 2 performs the filtering correctly, sine waves of different frequencies are used as inputs to test the frequency response[5]. Note that there are 2 analogue filters, 2 digital filters and a network of resistors on the board to prevent aliasing and adjust gains; these will result in attenuation. **As such, the amplitude of input waves is kept constant such that relative changes in amplitude of the output waves are clear.**

The scope traces obtained are graphed in figure 9. The results are exactly expected.

- There is significant signal attenuation in the stop bands.
- The strength of the signal grows progressively in the transition band.
- There is slight variation in the peak-to-peak amplitude at different frequencies within the pass band.

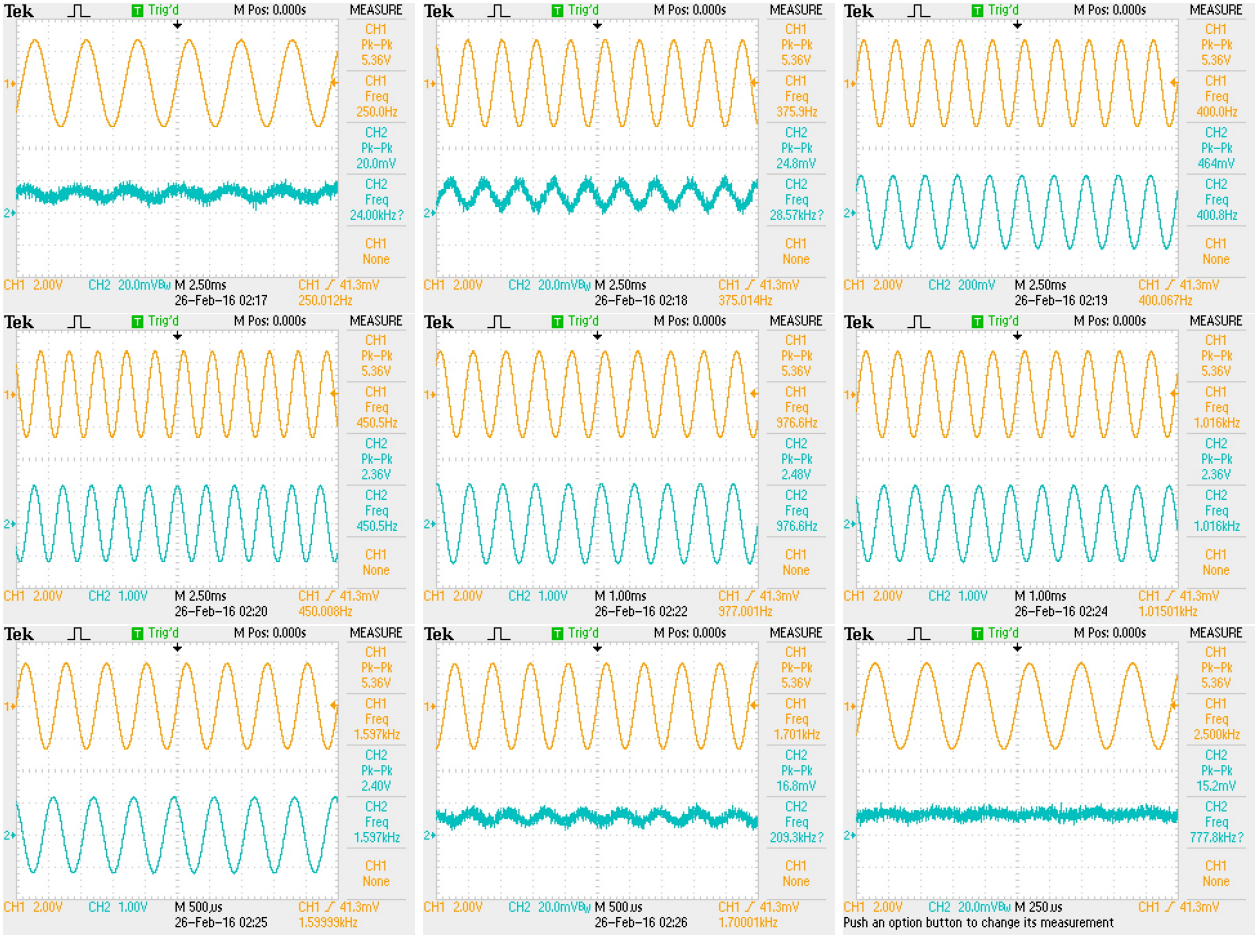

Figure 9: Frequency spectrum analysis of filter

A detailed analysis of the frequency response is presented in section 11.

[5]This is done for the sake of completion. In section 11, the spectrum analyser is used to check the frequency response of the filter

5 FIR Filter Implementation: Circular Buffering

In this section, a more involved implementation of the buffer is explored. Previously, the index of the array represented the relative delay between the previous samples and the current sample. T**he array had a very regular structure and computing the inner product between the array containing the filter coefficients and the array containing the samples was simple.** Maintaining the regular structure of the array required a great deal of overhead as each iteration required complete reorganisation of the samples stored in the buffer.

Instead, a circular buffer is used. A variable `ptr` keeps track of the index that the newest sample is stored in. **It is important to note that the variable `ptr` is not an actual pointer, in that it does not contain a memory address; it is an integer that stores an array index.** New samples are placed to the left of older samples; the array grows towards `x[0]` instead of towards `x[M-1]`. When a new sample is read, it simply replaces the oldest sample in the buffer and the variable `ptr` is decremented by 1. **This implementation is referred to as a circular buffer because the variable `ptr` wraps around when it reaches the first element of the array.** This is clearly depicted in figure 10. For simplicity, a 6^{th} order filter is depicted.

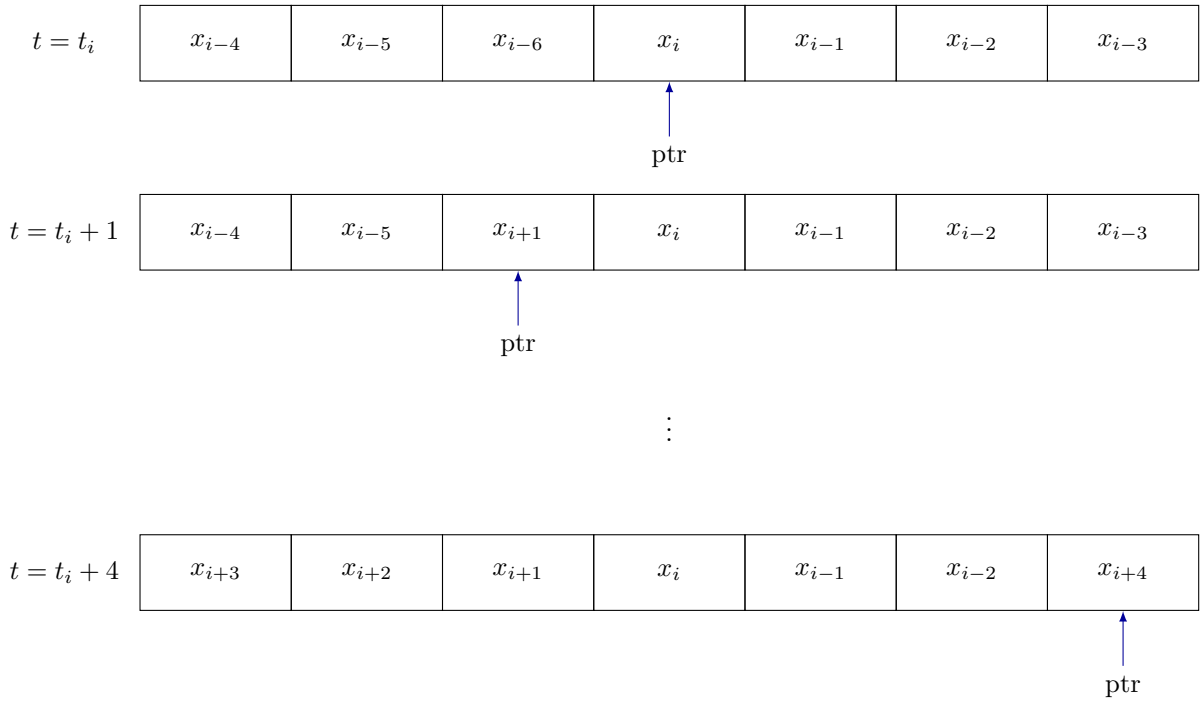

Figure 10: Wrapping of `ptr` in a circular buffer

The circular buffer requires significantly less overhead in terms of maintaining the structure of the buffer. This is however compensated by the increased complexity needed to implement the convolution sum. An `if` statement needs to be used to ensure that the variable `ptr` does not exceed its bounds of $[0, M]$, where M represents the filter order. To implement a filter of order M, $M+1$ coefficients and $M+1$ samples are necessary. This is clear from equation (1). Thus, the variable `ptr` can take $M+1$ distinct values.

An `if` statement is also necessary when reading data into the buffer.

Listing 5 shows the code required to implement the circular buffer in C. **Updating of the buffer and all read and write operations are performed in `ISR_filter` whereas `circ_FIR` only performs the convolution sum and returns the result.** Separating the two distinct tasks into unique functions makes for increased readability. The overhead of making function calls within the ISR is eliminated at compile time[6].

```
1  void ISR_filter(void){
2
3      // new sample is stored directly into the buffer
4      x[ptr] = mono_read_16Bit();
5
6      // function that performs convolution is called
7      // function returns a double and is converted to a short before being sent to output
8      mono_write_16Bit((short)circ_FIR());
9
10     // updates the position of the pointer
11     ptr--;
12
13     // checks if the ptr needs to be wrapped around
14     if(ptr<0)
15         ptr = M-1;
16
17 }
18
19 double circ_FIR(void){
20     short i;
21     double y = 0;
22
23     // inner product of array containing filter coefficients and buffer containing samples
24     // after M iterations, the value of ptr is unchanged
25     for(i=0; i<M; i++){
26         y += b[i]*x[ptr];
27         ptr--;
28
29         // checks if the ptr needs to be wrapped around
30         if(ptr<0)
31             ptr = M-1;
32     }
33
34     return y;
35 }
```

Listing 3: `circ_FIR`

[6]More details about compiler optimisations are provided in section 8.

6 FIR Filter Implementation: Circular Buffering Exploiting Symmetry

6.1 Function that Uses `if` Statements

As mentioned in section 2, a FIR filter is usually implemented when a linear phase response is required. A linear phase response can by achieved by placing zeroes to produce a symmetric impulse response. The symmetric relationship between coefficients of a linear phase FIR filter is stated in equation (6) and figure 5 corroborates the fact that the designed filter's coefficients are indeed symmetric. **This symmetry can be exploited to halve the number of multiplication operations that need to be computed to perform filtering. Since pairs of samples are going to be multiplied by the same coefficient, the samples can first be added, then multiplied.** This is depicted in the block diagram shown in figure 2.

Computationally, this implementation provides a significant improvement since the `for` loop[7] required to compute the convolution sum only needs $^{M-1}/_{2}$ iterations. However, for correct implementation, two pointers[8] need to be managed. One pointer will traverse the samples in the order x_0, x_{-1}, $\cdots$, whereas the other pointer will traverse the samples in the order x_{-M}, x_{-M+1}, $\cdots$.

- **The first pointer is called `move_right` and initialised to be equal to `ptr`.**
- **The second pointer is called `move_left` and initialised to be equal to `ptr` − 1.**

This improved implementation also utilises the circular buffer and thus it has a similar flaw; there is a potential of the pointers going out of range. Unless the initial position of `ptr` is right in the middle of the buffer `x`, either `move_right` or `move_left` will reference an invalid memory location at a certain iteration of the `for` loop. This needs to be dealt with, in a similar manner to that presented in listing 5, with the use of `if` statements. **Both pointers do not need to be checked at every iteration of the for loop because each time the function `circ_FIR_symmetric` is called, only one of the pointers will overflow.** The list below summarises which initial condition will lead to which pointer overflowing.

- `ptr` $\leq {}^{M-1}/_{2}$**:** `move_left` **will overflow**
- `ptr` $\geq {}^{M-1}/_{2}$**:** `move_right` **will overflow**
- `ptr` $= {}^{M-1}/_{2}$**: neither** `move_left` **nor** `move_right` **will overflow**

The case where `ptr` $\leq {}^{M-1}/_{2}$ is depicted in figure 11.

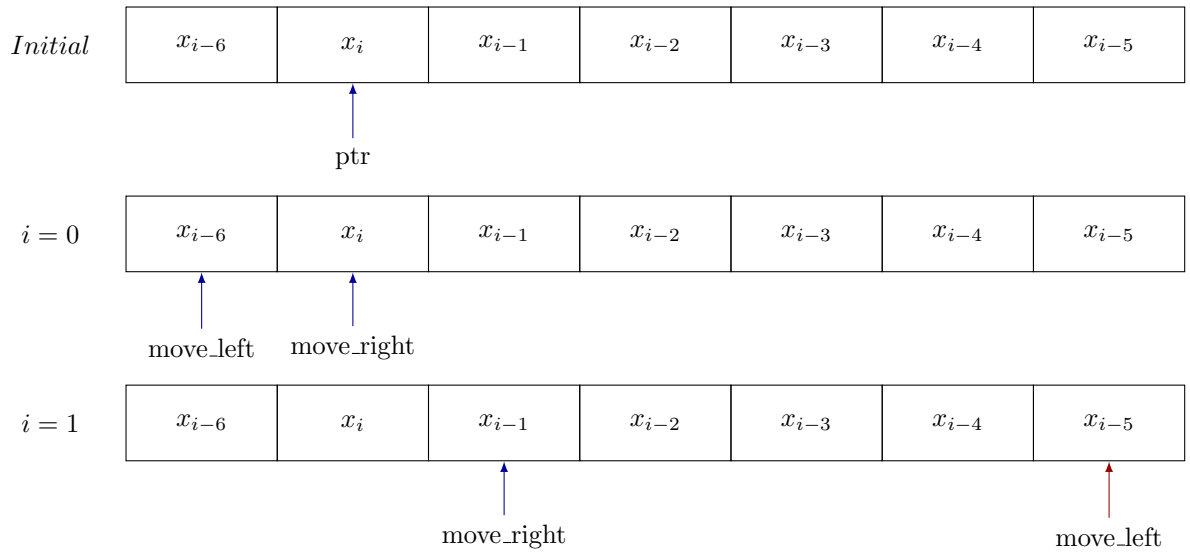

Figure 11: `move_left` will overflow before `move_right`

[7] Number of multiplies per iteration of the `for` loop is constant. There is only one multiply operation per iteration.

[8] Again it is key to note that the term pointer is used loosely. In fact, two variables defined as integers are created and used to keep track of two distinct positions in the buffer.

Thus, if the initial position of the variable `ptr` is checked, only one pointer will overflow and only one `if` statements is needed in the `for` loop. Listing 4 presents the code that is used to implement a linear phase FIR filter by exploiting the symmetric nature of the impulse response. **Note that, the filter designed in MATLAB has an odd number of coefficients and thus one multiply accumulate step is implemented outside the `for` loop.**

```
// symmetry of linear phase FIR filter is exploited
// elements are added before multiply
// filter has odd number of coefficients, last term calculated separately
double circ_FIR_symmetric(void){
    int i;
    int move_right =  ptr;
    int move_left = ptr - 1;
    double y = 0;

    // special case that move_left initialised to element not in the array
    if(move_left == -1)
        move_left = M-1;

    // move_left will go out of range before move_right
    if(ptr<(M-1)/2){
        for(i=0; i<(M-1)/2; i++){
            y += b[i]*(x[move_left]+x[move_right]);

            // incrementing move_right without checking since there is no danger of overflow
            move_right++;

            // move_left needs to be checked before decrementing, can go out of bounds
            if(move_left<=0)
                move_left = M-1;
            else
                move_left--;
        }
        y += b[(M-1)/2]*x[move_right];
    }
    // move_right will go out of range before move_left
    else if(ptr>(M-1)/2){
        for(i=0; i<(M-1)/2; i++){
            y += b[i]*(x[move_left]+x[move_right]);

            // decrementing move_left without checking since there is no danger of overflow
            move_left--;

            // move_right needs to be checked before incrementing, can go out of bounds
            if(move_right>=M-1)
                move_right = 0;
            else
                move_right++;
        }
        y += b[(M-1)/2]*x[move_left];
    }
    // neither move_right nor move_left will go out of bound
    else{
        for(i=0; i<(M-1)/2; i++){
            y+= b[i]*(x[move_left]+x[move_right]);

            // neither move_left nor move_right needs to be checked
            move_left--;
            move_right++;
        }
        y += b[(M-1)/2]*x[move_right];
    }
    return y;
}
```

Listing 4: `circ_FIR_symmetric`

6.2 Function that Completely Eliminates the Use of `if` Statements

Next, it is possible to completely eliminate the use of `if` statements from the C code. As described in section 6.1, only one of the pointers, either `move_left` or `move_right` will overflow during the `for` loop used to compute the convolution sum. **It is possible to calculate the number of iterations before the pointers will overflow using the value of `ptr`.** The list below summarises this argument.

- **`ptr` $\leq {}^{M-1}/_{2}$: `move_left` will overflow after `ptr` iterations, where `ptr` is an integer within the range** $[0, {}^{M-1}/_{2}]$
- **`ptr` $\geq {}^{M-1}/_{2}$: `moveright` will overflow after $M -$ `ptr` iterations, where $M -$ `ptr` is an integer within the range** $[1, {}^{M-1}/_{2}]$
- **`ptr` $\geq {}^{M-1}/_{2}$: neither `move_left` nor `move_right` will overflow**

Note that, in case 1, `move_left` can overflow even before the `for` loop begins. This special case will be dealt with separately.

Listing 5 provides the C code that implements a FIR linear phase filter without the use of any `if` statements. There are several things that should be noted for the code below:

- `move_left++` and `move_right--` are used to reduce the amount of code required.
- In between the two `for` loops, the pointer that overflows is wrapped around.
- Again, because the number of filter coefficients is odd, one multiply accumulate step is implemented outside the `for` loops.

```
1  // two for statements are used to eliminate the use of if statements
2  double circ_FIR_two_for_loops(void){
3      int i;
4      double y = 0;
5      int end;
6      int move_right = ptr;
7      int move_left = ptr-1;
8
9      // special case, ptr points to the first element
10     if(ptr==0){
11         // wrap left pointer around
12         move_left = M-1;
13
14         for(i=0; i<(M-1)/2;i++)
15             y+= b[i]*(x[move_left--]+x[move_right++]);
16
17         y+= b[(M-1)/2]*x[(M-1)/2];
18     }
19     // left moving point will go out of bounds
20     else if(ptr<(M-1)/2){
21         // calculate the number of iterations for the first loop
22         end = ptr;
23
24         for(i=0; i<end; i++)
25             y += b[i]*(x[move_left--]+x[move_right++]);
26
27         move_left = M - 1;
28
29         // second for loop accounts for the overflow
30         for(i=end; i<(M-1)/2;i++)
31             y += b[i]*(x[move_left--]+x[move_right++]);
32
33         y += b[(M-1)/2]*x[ptr+(M-1)/2];
34     }
35     // right moving point will go out of bounds
36     else if(ptr>(M-1)/2){
37         // calculate the number of iterations for the first loop
38         end = M-ptr;
39
40         for(i=0; i<end; i++)
41             y += b[i]*(x[move_left--]+x[move_right++]);
42
43         move_right = 0;
44
45         // second loop accounts for the overflow
46         for(i=end; i<(M-1)/2;i++)
47             y += b[i]*(x[move_left--]+x[move_right++]);
48
49         y += b[(M-1)/2]*x[ptr-1-(M-1)/2];
50     }
51     // neither left moving pointer, nor right moving pointer overflows
52     else{
53         for(i=0; i<(M-1)/2; i++)
54             y+= b[i]*(x[move_left--]+x[move_right++]);
55
56         y += b[(M-1)/2]*x[M-1];
57     }
58     return y;
59 }
```

Listing 5: `circ_FIR_two_for_loops`

7 FIR Filter Implementation: Circular Buffering with Doubled Memory Size

Up until this point, filter implementation has been improved by:

- Using a circular buffer
- Exploiting the symmetry of the impulse response

Using the circular buffer significantly increases the efficiency of the program however, there is a possibility of overflow which has to be accounted for. An overflow can occur both when reading new samples into the array as well as computing the inner product between the array containing the samples and the array containing the filter coefficients. **Overflow during the inner product can be avoided if a buffer of size $2M$ instead of size M.** This implementation does have overhead costs:

- **Buffer of size $2M$ instead of M is needed**
- **New sample needs to be read into two positions instead of just one; positions are offset by M**

This implementation is illustrated in figure 12.

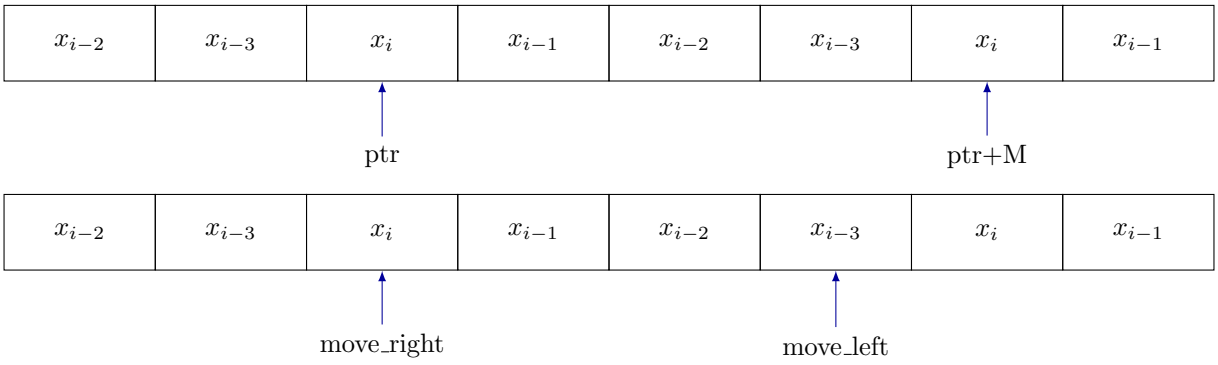

Figure 12: Circular buffer of size $2M$

Listing 6 shows the C code that implements the linear phase FIR filter using a buffer of size $2M$. Again `move_left++` and `move_right--` are used to reduce the amount of code required. **Also, note that `move_left` is initialised to `ptr` $+ M - 1$.**

```
// size of the buffer is doubled
// neither move_left nor move_right will overflow
void ISR_filter(void){
    // sample has to be read into two positions in the buffer
    // positions are offset by M
    x[ptr] = mono_read_16Bit();
    x[ptr+M] = x[ptr];

    // function that performs convolution is called
    // function returns a double and is converted to a short before being sent to output
    mono_write_16Bit((short)circ_FIR_double_buff());

    // updates the position of the pointer
    ptr--;
    // checks if the ptr needs to be wrapped around
    if(ptr<0)
        ptr = M-1;
}
double circ_FIR_double_buff(void){
    int i;
    double y=0;
    double move_right= ptr;
    double move_left= ptr+M-1;

    // only one for loop is necessary
    for(i=0; i<(M-1)/2; i++)
        y+=b[i]*(x[move_right++]+x[move_left--]);

    y+=b[i]*x[move_right];

    return y;
}
```

Listing 6: `circ_FIR_double_buff`

8 Compiler Optimisation

Performance of the filter has been improved through smart programming however, it can also be enhanced using the compiler's in-built optimisations options. The compiler offers multiple levels of optimisations. They are described below.[1]

- No optimisation
- Level 0
  - Performs control-flow-graph simplification
  - Allocates variables to registers
  - Performs loop rotation
  - Eliminates unused code
  - Simplifies expressions and statements
  - Expands calls to functions declared in line
- Level 1
  - Performs all optimisations from lower levels
  - Performs local copy/constant propagation
  - Removes unused assignments
  - Eliminates local common expressions
- Level 2
  - Performs all optimisations from lower levels plus:
  - Performs software pipe lining
  - **Performs loop optimisations**
  - Eliminates global common sub expressions
  - Eliminates global unused assignments
  - **Converts array references in loops to incremented pointer form**
  - **Performs loop unrolling**
- Level 3
  - Performs all optimisations from lower levels plus
  - Removes all functions that are never called
  - Simplifies functions with return values that are never used
  - In line calls to small functions
  - Reorders function declarations; the called functions attributes are known when the caller is optimised
  - Propagates arguments into function bodies when all calls pass the same value in the same argument position
  - Identifies file-level variable characteristics

In laboratory 4, optimisation 1 and 3 are not tested.

- Optimisation 1 is typically applied to individual blocks of code rather than across functions and files.
- Optimisation 3 does not allow the use of breakpoints; this would prevent using the clock to profile the speed of the ISR.

9 Performance Measure

Table 1 compares all of the different implementations that have been discussed thus far. **It should be noted that the clock cycles stated below measure the time taken for the entire ISR to run; the read and write operations are also included in the time taken.**

Performance Measure			
Function	Optimisation		
	No Optimisation	Level 0	Level 2
`non_circ_FIR`	13090	10761	2762
`circ_FIR`	11457	10180	4301
`circ_FIR_symmetric`	7465	5884	867
`circ_FIR_two_for_loops`	5937	5170	1152
`circ_FIR_double_buff`	5781	4151	1855

Table 1: Performance measure analysis

The functions in column 1 of the table are written in increasing order of expected efficiency. All functions using circular buffers are expected to out perform `non_circ_FIR` since complete organisation of the buffer is extremely costly. `circ_FIR` computes twice the number of multiplications that `circ_FIR_symmetric`, `circ_FIR_two_for_loops` and `circ_FIR_double_buff` compute and thus is expected to be significantly slower. `circ_FIR_symmetric` is expected to be slower that `circ_FIR_two_for_loops` since the former utilises `if` statements. `if` statements are mapped to branches in assembly. If the condition placed on the branch is satisfied, the program counter jumps and executes a new piece of code. This jump causes a stall in the pipeline. For this reason, `if` statements are expected to produce code that is slower. `circ_FIR_two_for_loops` does not use any `if` statements and thus is not expected to stall the pipeline. `circ_FIR_double_buff` is expected to be slightly faster than `circ_FIR_two_for_loops` since only 1 for loop is utilised. `circ_FIR_double_buff` does however perform an extra write operation, which is computationally expensive, and thus a measured increase in efficiency is expected.

The number of clock cycles taken for each ISR correlates very strongly with the expected efficiency of the function when no optimisation is applied. However, once compiler optimisations are included, the program does become more efficient however `circ_FIR_symmetric` out performs all the other functions. **This is slightly unexpected however, as mentioned in section 8, in-built optimisations are compiler dependent.** The compiler is easily able to identify the multiply-accumulate loop in `circ_FIR_symmetric` and thus produces machine code with a throughput of 2 multiply-accumulate operations per clock cycle. For the other functions, the compiler does not identify the C code and does cannot produce the most efficient code.

The best way to ensure optimal efficiency is to write the program in assembly.

10 FIR Filter Implementation: Fastest Implementation

Up until this point, the term pointer has been used loosely; it has been used to describe a variable `ptr`. It is critical to note that the variable `ptr` is defined as an `int` and holds a value between $[0, M-1]$. It is used to indicate the index of the newest sample in the buffer. All samples have been read from the buffer using array indexing. This is not the most efficient implementation since a basic compiler will only store the address of the first element in the array. In listing 7, accessing a sample from the buffer requires 3 steps:

- Load the address of the first element of the array and the value of `i`
- Add the offset to the address of the first element of the array
- Read the value from the computed position in the array

```
1  y = 0;
2  for(i = 0; i < M; i++){
3      y += b[i] * x[i]
4  }
5  return y;
```

Listing 7: `for` loop used to illustrate array indexing concept

Using pointers will reduce the number of operations. The address of memory locations are tracked instead of array indexes and thus reading the sample value does not require any offset calculations. **Also, in assembly, it is possible to read the value stored at a certain memory address and increment\decrement the address stored in a variable such as `*move_left` or `*move_right` in just 1 clock cycle. This is achieved by the instruction the `LDH` instructions. It is not possible to read a sample using array indexing and increment\decrement a variable such as `ptr` that holds the index of the array in 1 clock cycle.**

Modern compilers convert array indexing to a pointer implementation automatically. Nonetheless, it is advisable to use pointers explicitly as:

1. Compiler optimisations are not always possible, for example, in the case of interrupts
2. Optimisations are heavily dependant on the compiler; a piece of code may have different levels of performance when ported from one compiler to another
3. **Low-level languages such as C allow complete control over what the processor. Using pointers over array indexing gives a clearer description of what should happen at the assembly code level.**

As such, listing 8 provides the C code that implements `circ_FIR_symmetric` with actual pointers containing memory addresses.

```
// pointers are used and in line incrementing/decrementing is used
double circ_FIR_symmetric_pointer(void){
    int i;
    double *move_left;
    double *move_right =  &x[ptr];
    double y = 0;

    // special case, ptr points to the first element
    if(move_right == &x[0])
        move_left = &x[M-1];
    // regular case
    else
        move_left = &x[ptr-1];

    // move_left will go out of range before move_right
    if(ptr<(M-1)/2){
        for(i=0; i<(M-1)/2; i++){
            y += b[i]*(*move_left--+*move_right++);

            // move_left needs to be checked before decrementing, can go out of bounds
            if(move_left==&x[0])
                move_left = &x[M-1];
            else
                move_left--;
        }
        y += b[(M-1)/2]*(*move_right);
    }
    // move_right will go out of range before move_left
    else if(ptr>(M-1)/2){
        for(i=0; i<(M-1)/2; i++){
            y += b[i]*(*move_left--+*move_right);

            // move_right needs to be checked before incrementing, can go out of bounds
            if(move_right==&x[M-1])
                move_right = &x[0];
            else
                move_right++;
        }

        y += b[(M-1)/2]*(*move_left);
    }
    // neither move_right nor move_left will go out of bound
    else{
        for(i=0; i<(M-1)/2; i++)
            y+= b[i]*(*move_left--+*move_right++);

        y += b[(M-1)/2]*(*move_right);
    }
    return y;
}
```

Listing 8: `circ_FIR_symmetric_pointer`

Lastly, table 2 shows the `circ_FIR_symmetric_pointer` is the fastest implementation of the linear phase FIR filter on the $TMS320C6713$ DSK board using optimisation level 2. **This confirms the assertion above that using pointers is advisable even if the compiler utilises in-built pointer based optimisations.**

Performance Measure			
Function	Optimisation		
	No Optimisation	Level 0	Level 2
`non_circ_FIR`	13090	10761	2762
`circ_FIR`	11457	10180	4301
`circ_FIR_symmetric`	7465	5884	867
`circ_FIR_two_for_loops`	5937	5170	1152
`circ_FIR_double_buff`	5781	4151	1855
`circ_FIR_symmetric_ptr`	**7169**	**5125**	**778**

Table 2: Results showing `circ_FIR_symmetric_pointer` is the fastest implementation

11 Spectrum Analyser

11.1 Original Filter

Before the frequency response of the filter can be studied, it is important to realise that the $TMS320C6713$ DSK will have a non-flat frequency response even if no filtering is applied. This is due to the 2 analogue filter, 2 digital filters and multiple network of resistors that are present on the board. For accurate analysis, the frequency response of the DSK board has to be taken into account. Figure 13 shows the response of the DSK board.

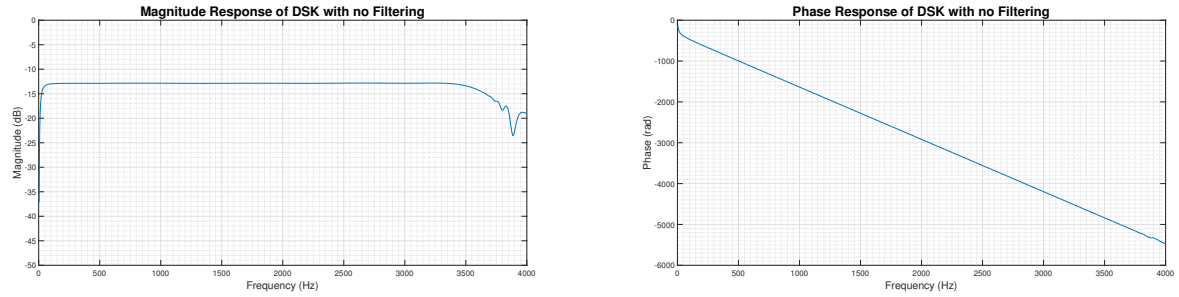

Figure 13: Frequency response of $TMS320C6713$ DSK board without any filter

Pass band gain of $-13dB$ corresponds to a gain of approximately $1/4$. This is exactly as expected. The $AIC23$ audio codec is connected to a potential divider with gain 0.5 externally. In addition, there is a potential divider built into the line inputs with of gain of 0.5 as well. **Cascaded, these potential dividers produce a gain of $1/4$.**

Next, figure 14 shows the frequency response of the filter designed in section 3. Note that the magnitude response in the pass band is centred about $-13dB$. This is discrepancy comes about because of the inherent frequency response of the DSK board as discussed above.

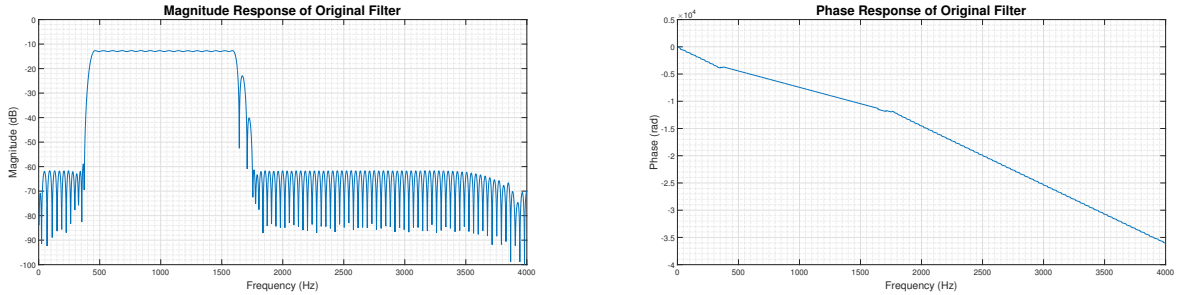

Figure 14: Frequency response of the filter designed in section **??**

Finally, figure 15 shows the frequency response once the offset introduced by the DSK board is taken into account.

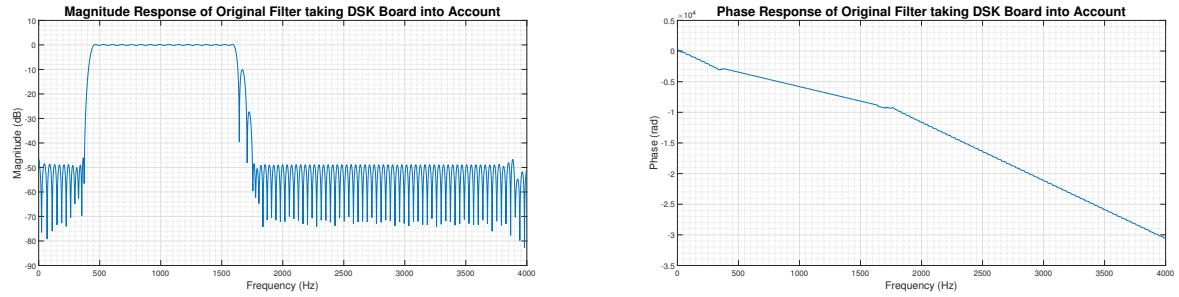

Figure 15: Frequency response of filter taking $TMS320C6713$ DSK board's inherent response into account

The filter designed in section 3 meets all of the specifications on MATLAB, however when implemented on the $TMS320C6713\ DSK$ board, it fails to satisfy the stop band attenuation[9]. Figure 16 shows that the stop band attenuation surpasses the limit of $-48dB$.

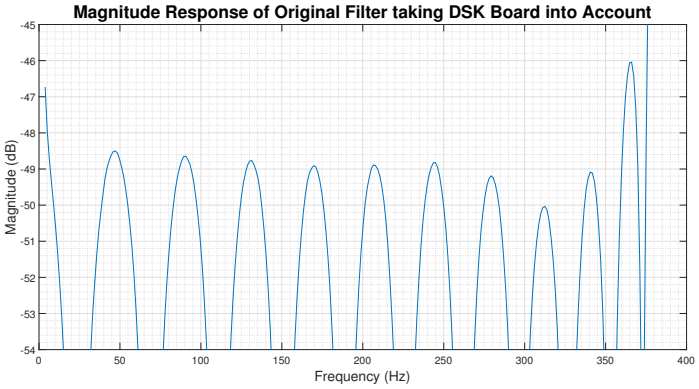

Figure 16: Filter does not meet specified stop band attenuation

11.2 Improved Filter Design

To ensure that the filter meets all of the specifications, a new filter is designed on MATLAB. Listing 9 shows the MATLAB code used to generate the new filter. **The stop band attenuation and pass band ripple are over specified to $-52dB$ and $0.36dB$ respectively.** In addition, the transition band edges have been tweaked slightly.

```
rp = 0.36;                          % defines the passband ripple
rs = 52;                            % defines the stop band attenuation
F_sampling = 8000;                  % defines the sampling frequency
F_cutoff = [373 450 1600 1700];     % defines the cutoff frequencies
filter_amplitudes = [0 1 0];        % defines ideal gain of each band

% functiuon used to estimate variables need for firpm function
[N, Fo, Ao, W] = firpmord(  F_cutoff,...
                            filter_amplitudes,...
                            [10^(-rs/20) (10^(rp/20)-1)/(10^(rp/20)+1) 10^(-rs/20)],...
                            F_sampling);


coefs = firpm(N+4, Fo, Ao, W);      % order of filter is increased by 4 to meet specifications

[h,w] = freqz(coefs, [1], 2^16);
```

Listing 9: MATLAB code to generate coefficients for an improved FIR filter

The improved filter requires 221 coefficients instead of 211 to be implemented. Figure 17 shows the magnitude response of the new filter. Note that this plot already takes into account the inherent frequency response of the $TMS320C6713\ DSK$ board. **The plot also includes the theoretical frequency response and it is clear that the filter performs exactly as expected.**

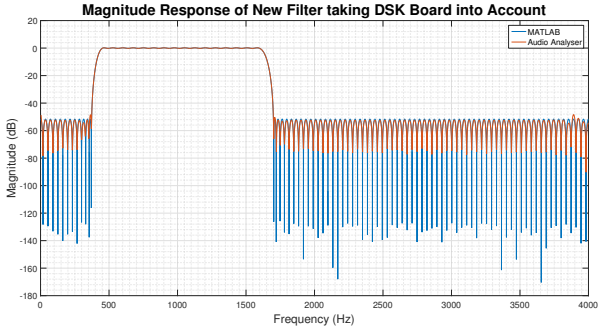

Figure 17: Magnitude response of improved filter

[9]The phase response of the filter designed in section 3 is not discussed in detail. The filter does not meet satisfy the specifications set out in laboratory 4 and thus a new filter is designed in the next section. Phase response of the new filter is presented.

Figure 18 plots the frequency response of filter for specific frequencies. **The plot confirms that the magnitude spectrum of the improved filter satisfies all the specifications set out in laboratory 4.**

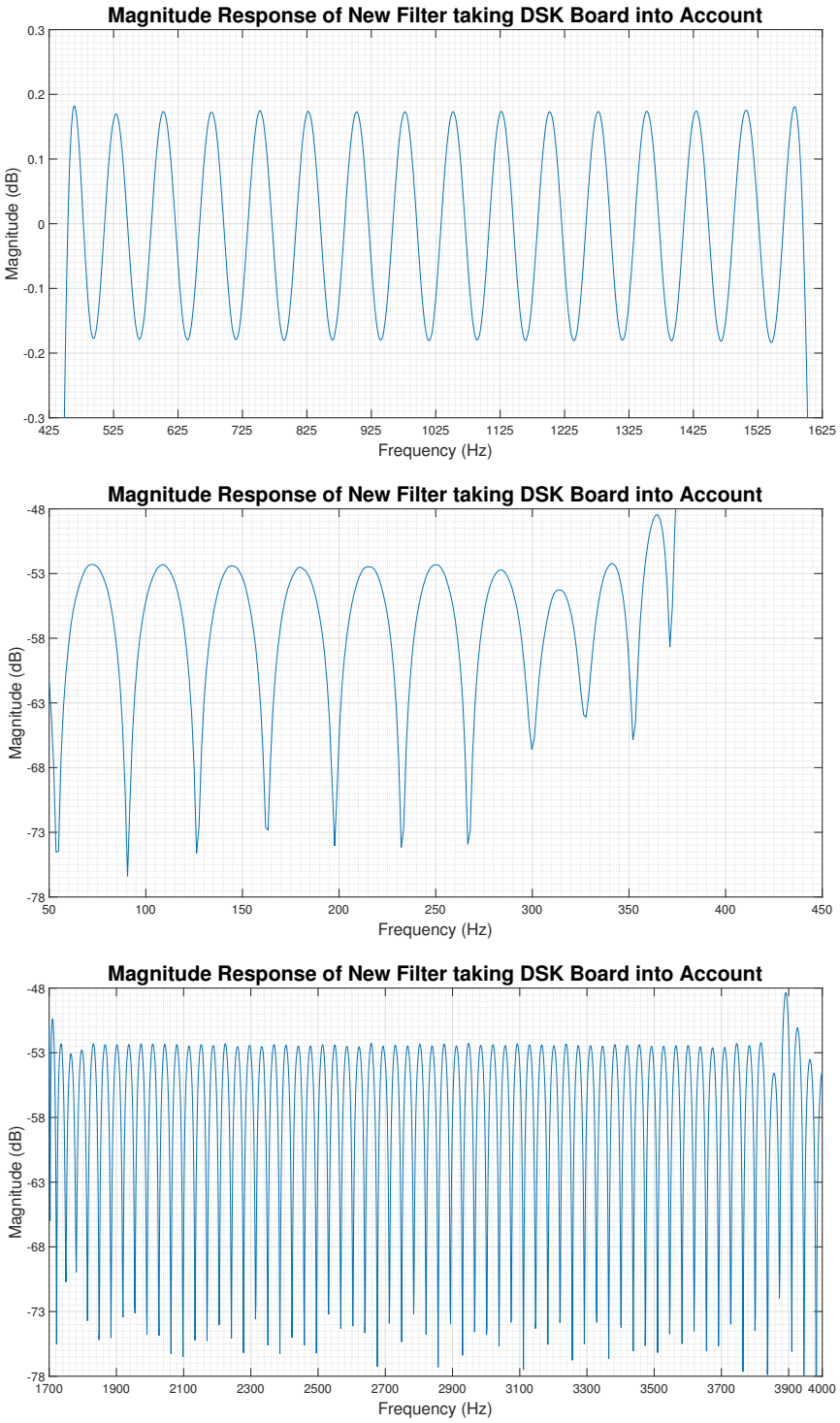

Figure 18: Magnitude response of improved filter

Next, the phase response of the improved filter is studied. Figure 19 shows the phase response of the filter and confirms the fact that the phase in indeed linear in the pass band.

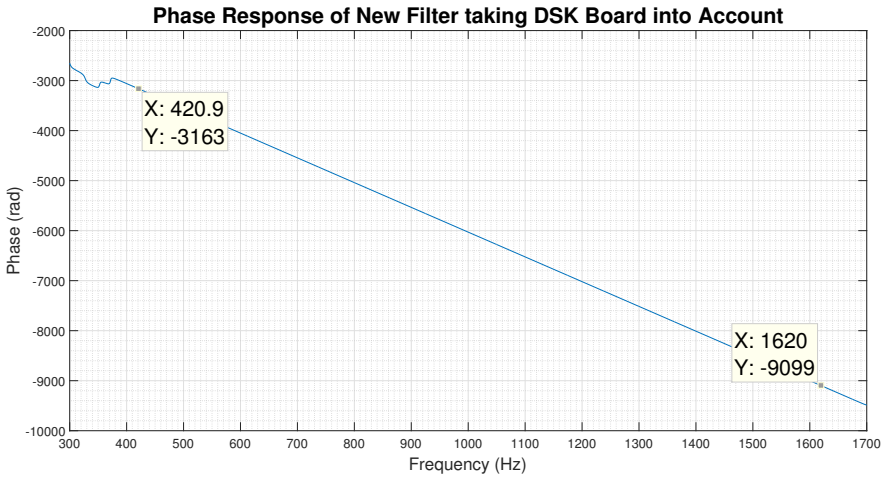

Figure 19: Phase response of improved filter

Two data cursors are included in the plot. These are the points used to calculate the observed group delay in equation (17).

$$Group\ Delay\ (/s) = -\frac{(\phi_1 - \phi_2)(\frac{\pi}{180})}{(\omega_1 - \omega_2)(2\pi)} t_s = -\frac{(-3163 \cdot 10^4 - -9099 \cdot 10^4)(\frac{\pi}{180})}{(420.9 - 1620)(2\pi)} \frac{1}{8000} = 17.2ms \tag{17}$$

The theoretical group delay is calculated in equation (18).

$$Group\ Delay\ (/s) = \frac{M-1}{2} t_s = \frac{221-1}{2}(1/8000) = 13.8ms \tag{18}$$

The observed value is approximately equal to the theoretical value. **The improved filter performs as extremely well both in terms of its magnitude and phase response.**

Finally, `circ_FIR_symmetric_pointer` is tested with the new filter. The results obtained are listed in table 3.

New Filter: 221 Coefficients			
Function	Optimisation		
	No Optimisation	Level 0	Level 2
`circ_FIR_symmetric_ptr`	7546	5497	**812**

Table 3: Performance measure of `circ_FIR_symmetric_ptr` for new filter

12 Conclusion

This report presented the theory behind FIR filters and the discussed generation of filters using the MATLAB functions `firpmord` and `firpm`. A detailed discussion about implementing FIR filters on the $TMS320C6713\ DSK$ board is presented. **Compiler optimisations were considered; the function that is expected to be the fastest might not actually be the fastest once compiler optimisations are applied. More details about the type of structures that the compiler recognises and optimises can be found in the Texas Instrument TMS320C6000 Optimisation Compiler User Manual.** Lastly the frequency spectrum of the designed filter was observed. The designed filter did not meet all the specifications and thus a new filter was designed.

The improved filter meet all specifications listed in laboratory 4. The filter was implemented in C and 812 **clock cycles were required for a sample to be read, the convolution sum to be computed and the output to be sent to the AIC23 audio codec. Without including the read or write operation, the filter only required** 632 **clock cycles.**

References

[1] Instruments, T. (july 2012). TMS320C600 Optimizing Compiler. Retrieved from `http://www.ti.com/lit/ug/spru187u/spru187u.pdf`