Department of Electrical and Computer Engineering
University of Victoria
SENG 440 – Embedded Systems
Section A01

Digital Filters

Final Report

Brett Dionello	V01026046
Ryan Javanmardi	V00895649

June 27, 2024

Table of Contents

Objectives

The first objective of this project was to implement a digital filter in C, targeting deployment on a processor with a 32-bit ARM architecture. The second objective was to apply software optimization techniques to the C code in order to improve its performance on the target platform; this required us to study the instruction set architecture and understand the mechanics of the GCC compiler in order to write C code in a manner that would be compiled into more efficient assembly instructions to be run on the target platform.

Theory

Digital Filters are an essential component in large variety of microprocessor-based systems with applications in areas such as telecommunications, control systems, biomedical devices, and audio/image processing to name only a few. The job of a filter is to remove the unwanted components of an incoming signal, so that only the components of the signal that are of interest are transmitted at its output. The unwanted components of the signal are typically referred to as 'noise' because they are introduced from external phenomena interfering with the signal transmission. **Figure 1** is an illustration of the function of a digital filter.

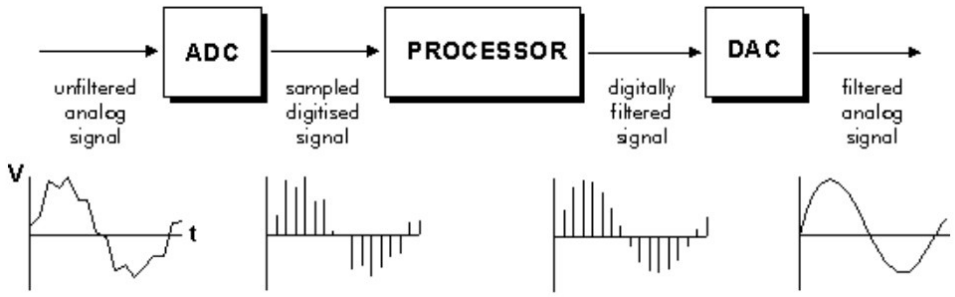


Figure 1 *Signal Smoothing by Digital Filter Implemented on a Processor [1]*

The way digital filters do this, is by performing arithmetic operations on their past input/output binary values to calculate their next output. The arithmetic operations typically involve multiply and accumulate operations with the inputs and chosen constants (see **Figure 2** for a mathematical illustration). The specific way in which these calculations are done determines the class, function, and type of the filter.

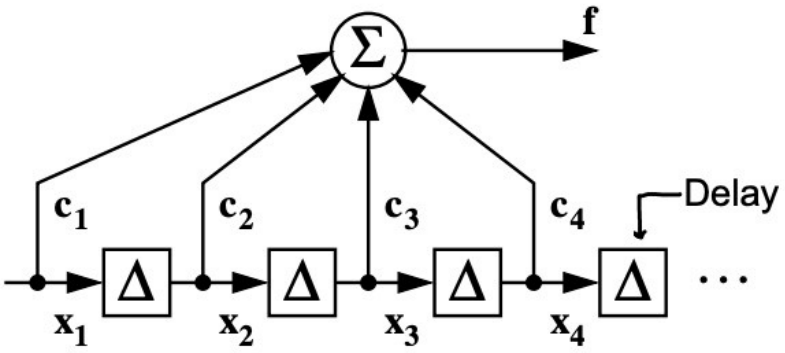


Figure 2 *Mathematical Illustration of a Typical Digital Filter [2]*

There are two main classes of Digital Filters: Finite Impulse Response (FIR) and Infinite Impulse Response (IIR). FIR Filters will produce an output based on the current input and a finite number of past inputs; FIR filters are inherently stable because they are non-recursive (do not have a feedback loop from output to input) and are simpler to implement, but at the cost of greater computational load, memory usage, and latency. IIR Filters will produce an output based on the current input and all/some past inputs, and all/some past outputs making them recursive in nature and thus introducing the possibility for instability; IIR filters, however, generally require less computational resources and achieve a steeper roll-off at the cut-off frequency than their FIR counterpart of equal order (number of poles/zeros of the filter's transfer function). **Table 1** below summarizes the difference equations for both classes of filter.

Table 1 *Mathematical Definitions of Filter Classes [3]*

FIR Filter	IIR Filter
$y[n] = \sum_{i=0}^{M} b_i \cdot x[n-i]$	$y[n] = \sum_{i=0}^{M} b_i \cdot x[n-i] + \sum_{j=0}^{N} a_j \cdot x[n-j]$
Where: • $y[n]$ is the current output. • $x[n]$ is the current input. • b_i are the feedforward coefficients (corresponding to the current and past input values). • M is the order of the filter (which determines the number of past input samples used)	Where: • $y[n]$ is the current output. • $x[n]$ is the current input. • b_i are the feedforward coefficients (corresponding to the current and past input values). • a_j are the feedback coefficients (corresponding to past output values). • M is the order of the feedforward part. • N is the order of the feedback part.

The function of a digital filter is defined by the range frequencies it will pass/attenuate or, in other words, by the shape of its magnitude response (see Figure). For example, a low-pass

filter will attenuate all frequencies above a certain threshold, a high-pass filter will attenuate all frequencies below a certain threshold, and a band-pass will attenuate all frequencies outside of a specified range.

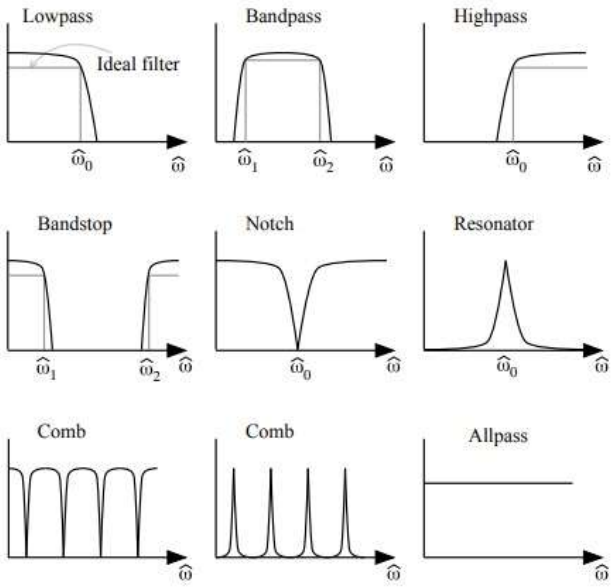


Figure 3 *Some Common Filter Functions [3]*

There are several types of digital filters but the most common are the Bessel, Butterworth, Chebyshev and Elliptic. Each one is distinguished by the characteristics of its frequency response – particularly the presence of ripple in the pass-band or stop-band, and the steepness of the transition from pass-band to stop-band. **Figure 4** below provides an example of the magnitude response of these four filter types used for a low-pass filter, highlighting their differences.

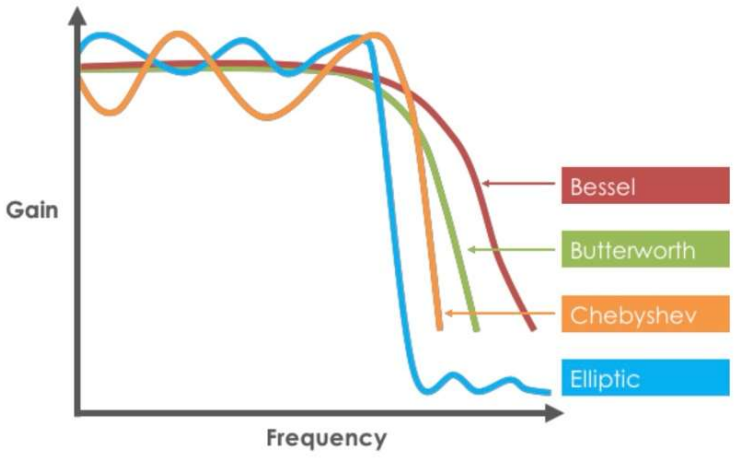


Figure 4 *The Four Common Filter Types*

A specific use-case of a digital filter is in an industrial automation system where the rotational speed of an electric motor driving a conveyor belt must monitored and controlled. A high-speed motor encoder would be used to generate pulses at a frequency proportional to the speed of the motor; this signal would then need to feedback into the control system to display the current speed of the motor and to adjust signals to the motor driver as necessary to ensure that its speed matches the desired setpoint. The signal travelling from the motor encoder to the controller will very likely be subject to some degree of mechanical vibrations and electromagnetic interference from other nearby electrical equipment introducing high-frequency components to the original pulses. These sources of noise could greatly affect the accuracy of the signal received by the controller, causing the system to not function properly. To solve this, a digital filter could be embedded into the controller implementation to remove high-frequency noise before reading the signal for speed measurement and control. The digital filter designed for this project targets such a use case and the specifications (sampling frequency, cut-off frequency, and attenuation) were chosen accordingly.

Project Specifications

Based on the example digital filter use-case scenario described in the previous section, a reasonable maximum frequency to expect from the motor encoder could be 2kHz. Thus, a low-pass filter with a 3dB cut-off frequency of 2kHz is chosen, and an attenuation of at least 29dB at 8kHz is targeted to effectively filter out any high-frequency noise beyond that range; a steeper slope at the cut-off frequency is desirable, but this would require a higher order filter or a different type of filter (ie. Chebyshev or Elliptic); for simplicity, this project will attempt to implement a second order Butterworth filter which also has the advantage of little to no ripple in its pass-band (see **Figure 4**). The AM3358 processor on the Beagle-Board Black has a 12-bit ADC with a maximum sampling rate of 200kHz [4]; for this project a sampling rate of 100kHz is chosen to be able to detect noise of at least up to 10kHz reliably with a safe margin (theoretically, by the Nyquist condition, this would allow detection of noise of up to 50kHz). A 16-bit word length is chosen so that the 32-bit processor [4] will be able to efficiently handle computations with operators of this length while keeping overflow and quantization errors to a minimum. **Table 2** below summarizes these specifications.

Table 2 *Digital Filter Project Specifications*

Filter Class	IIR (Recursive)
Filter Type	Butterworth
Filter Function	Low-Pass
Filter Order	Second Order
Sampling Rate	100 kHz
3dB Cut-off Frequency	2 kHz

Attenuation	29dB at 8kHz
Word Length	16-bit

Deployment Platform

Table 3 below summarizes all the key information regarding the hardware, instruction set architecture, and compiler used to implement the digital filter of this project.

Table 3 *Architecture and Hardware Specifications*

Platform	Beagle Bone Black development board
Processor	1GHz ARM Cortex-A8 AM3358 32-Bit RISC [5]
Architecture Features	NEON SIMD Coprocessor [6]
Operating System	Linux Debian 10
Compiler	GCC 8.3.0

Digital Filter Design

The procedure provided in the lecture slides [2] was followed to find the discrete-time transfer function of the digital filter described in the Project Specifications. The derivation of its transfer function can be found in **Error! Reference source not found.**.

$$H(z) = (0.003622)\frac{1 + 2z^{-1} + z^{-2}}{1 - 1.822695z^{-1} + 0.837182z^{-2}}$$

Figure 5 *Discrete-Time Transfer Function of the Digital Filter*

In order to verify the design's frequency response characteristics, we modelled the system in MATLAB and generated its Bode Plot as seen in Figure 6. The plot confirms that the design has a 3dB Cut-off frequency at 2kHz as desired. The attenuation at 8kHz is at around 25dB which is slightly less than the target, however, by 10kHz an attenuation of 29dB is acis was deemed good enough for the intended application of the filter. See Appendix A for the MATLAB script used to generate this Bode Plot.

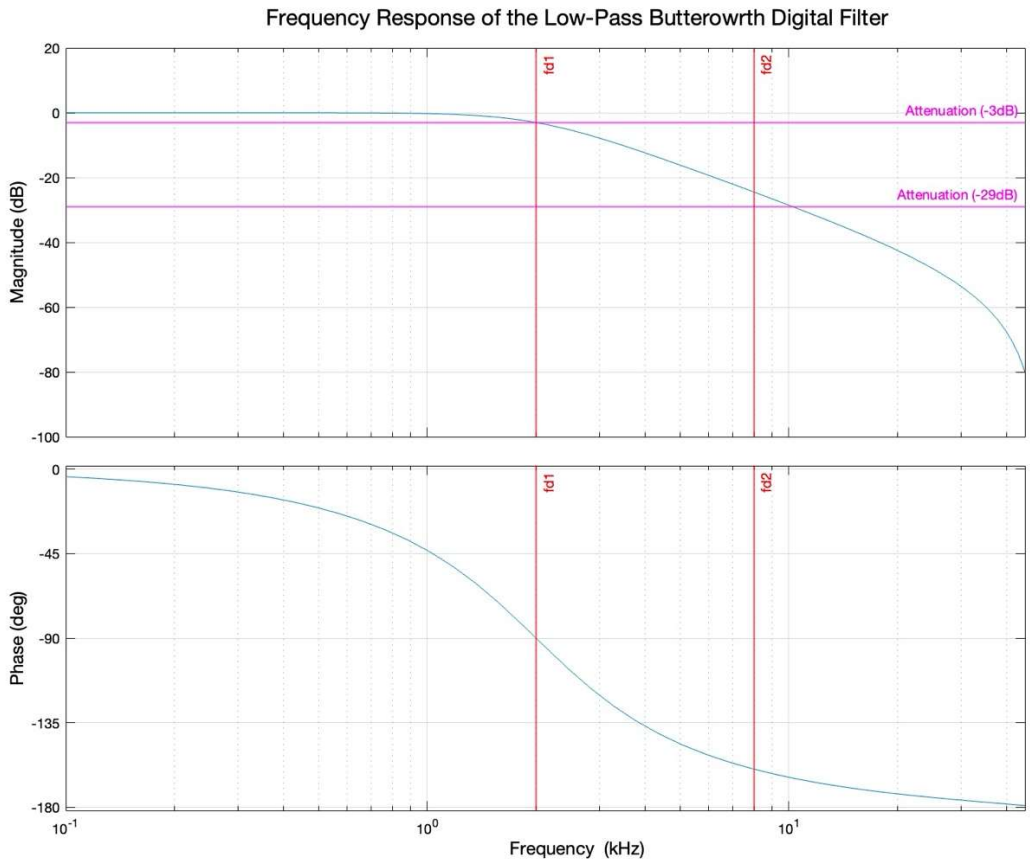


Figure 6 Digital Filter Bode Plot

The calculations in **Error! Reference source not found.** show the process for obtaining the scaled filter coefficients used in the fixed-point arithmetic operations of the embedded filter source code.

$$y[n] = \underset{\uparrow b_0}{0.003622}\,x[n] + \underset{\uparrow b_1}{0.007244}\,x[n-1] + \underset{\uparrow b_2}{0.003622}\,x[n-2] + \underset{\uparrow a_1}{1.822695}\,y[n-1] - \underset{\uparrow a_2}{0.837182}\,y[n-2]$$

Figure 7 *Difference Equation for the Digital Filter*

Figure 8 below is the scatter plot of the filter output for the first 100 samples of a normalized step input from -1 to +1. This plot was generated in Excel by using the difference equation above. The plot reveals that the maximum value of the output is around 1.08005 which needs to be known to determine the scale factor for output values as seen in the calculations that follow.

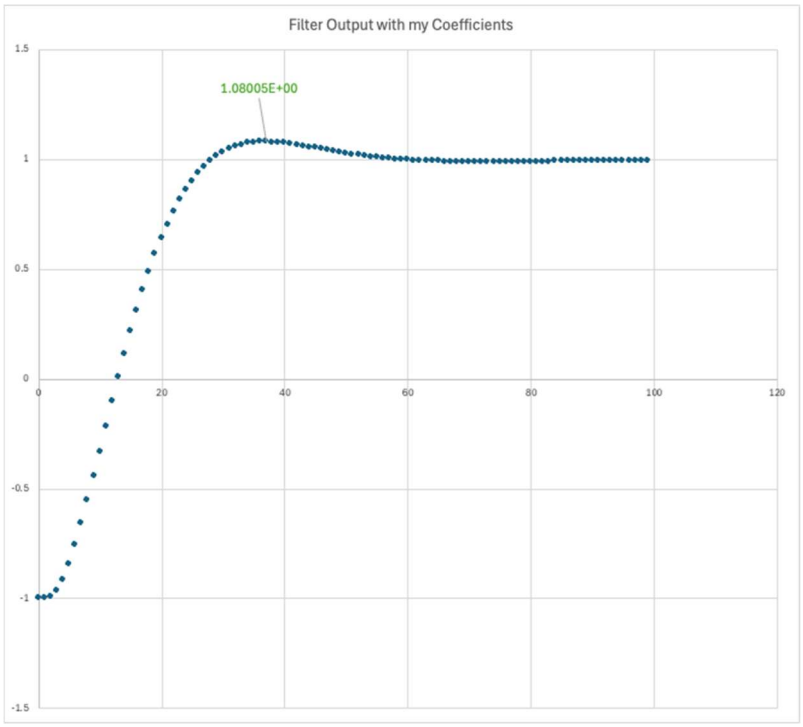


Figure 8 *Filter Output for first 100 Samples of Large Step Input (-1 to +1)*

The scaled difference equation in the box below (derivation in **Error! Reference source not found.**) is the final result of all the calculations and is directly implemented in the code of the digital filter as will be seen in the following section.

$$Y[n] = \frac{0x7680 \cdot X[n]}{2^{24}} + \frac{0x7680 \cdot X[n-1]}{2^{23}} + \frac{0x7680 \cdot X[n-2]}{2^{24}} + \frac{0.74A7 \cdot Y[n-1]}{2^{14}} + \frac{0x94D7 \cdot Y[n-2]}{2^{15}}$$

Figure 9 *Scaled Difference Equation for the Digital Filter*

Digital Filter Implementation

This section explores the code implementation of the IIR Filter designed in the previous section, and the effects of applying different software optimization techniques to it.

Methodology

The code for the digital filter underwent multiple design iterations in an effort to improve its performance. In the following section (Test Results), each version of the code was assessed for its stability (elimination of overflow and limit cycles) and accuracy (minimization of quantization error and truncation error) by examining the output, as well as for its efficiency by examining its assembly code and measuring execution time. The assembly code for each implementation was manually profiled for the total number of instructions, the number of load and store instructions, the number of branching instructions, and the number of multiply instructions as these have the most overhead and greatest impact on execution time. Execution time was measured roughly (in milliseconds) using a bash script that checks the difference in platform system's clock when the executable is run, and again after it is finished running; this is

not metric is not definitive as it could be different on other systems or affected by other processes on the test platform, but at least provides a general sense for the execution time.

Some common challenges with digital filters are overflow, quantization error, truncation error, error and limit cycles. Overflow was first eliminated by implementing saturating addition, then by choosing scale factors that accommodate the full range of possible values that variables in the difference equation could assume. Quantization error was minimized by using the power-of-2 scale factors that are as near as possible to the limits (max and min) of the analog signal being represented and the largest possible word length based on the system specifications. Truncation error was minimized by implementing rounding. Limit Cycles are directly attributed to overflow errors in addition and round-off errors in multiplication [2]. Limit cycles were eliminated completely in our tests by applying all the aforementioned techniques.

In addition to attempting manual optimization of the code using techniques such as loop unrolling and pipelining (instruction reordering), usage of the -O3 compiler optimization flag was also explored to observe the performance gains it offers. The following paragraph provides some important background information about optimization flags.

According to the GCC manual (version 8.3.0), without the use of any optimization flags, the compiler is designed to reduce the time and memory required for compilation and to produce predictable machine code with a straightforward correspondence to the source code for simplified debugging [7]. Using optimization flags tells the GCC compiler to attempt to improve the performance (reduce code size or decrease execution time) by activating a set of built-in optimization techniques to be applied to the source code during compilation. The set of optimization techniques applied depends on the flag that is used (read the GCC manual tonight for more details). The following are the most common optimization flags: -O/-O1, -O2, -O3, -Os, -Ofast. The effects of each of these flags is summarized in the table below.

According to the GCC manual (version 8.3.0), without the use of any optimization flags, the compiler is designed to reduce the time and memory required for compilation and to produce predictable machine code with a straightforward correspondence to the source code for simplified debugging [7]. Using optimization flags tells the GCC compiler to attempt to improve the performance (reduce code size or decrease execution time) by activating a set of built-in optimization techniques to be applied to the source code during compilation. The set of optimization techniques applied depends on the flag that is used (read the GCC manual tonight for more details). The following are the most common optimization flags: -O/-O1, -O2, -O3, -Os, -Ofast. The effects of each of these flags is summarized in the table below.

Table 4 *Common GCC Optimization Flags*

Flag	**Description**
-O1 (or -O)	Applies a set of optimization techniques that attempt to reduce code size and execution time without increasing compile time by a great deal [7].
-O2	Applies a larger set of optimization techniques (including all from -O1) that attempt to further improve performance; may increase compile time substantially [7].

-O3	Applies the largest set of optimization techniques (including all from -O2) that attempt to maximize performance gains; may increase compile time substantially [7].
-Os	Optimize for size; Applies all optimization techniques from -O2 but with the exception of a few specific ones which typically would increase the code size [7].
-Ofast	Optimize for speed; Applies all optimization techniques from -O3, and additionally enables a few extra ones for potential speed boost, but which are not in strict compliance for all language standards [7].

Test Results

This section briefly outlines the design features and statistics for each implementation of the digital filter.

Filter V0: Base Filter Code

Filter V0 is the example digital filter implementation provided in the lecture slides [2] with only very minor changes (fixed typos, and additional comments). It was verified that when running this code on the 32-bit ARM system, the expected limit cycles do occur. For convenience, the relevant loop code is included in the code snippet below.

```
// Input Coefficients
const short int B0 = 0x10C8;
const short int B1 = 0x2190;
const short int B2 = 0x10C8;
int tmp_B0, tmp_B1, tmp_B2;

// Output Coefficients
const short int A1 = 0x5FB7;
const short int A2 = 0xDD28;
int tmp_A1, tmp_A2;

// Filter Loop
register int i;
for (i=2; i<100; i++) {

    tmp_B0 = ((int)B0 * (int)X[i  ] + (1 << 14)) >> 15;
    tmp_B1 = ((int)B1 * (int)X[i-1] + (1 << 14)) >> 15;
    tmp_B2 = ((int)B2 * (int)X[i-2] + (1 << 14)) >> 15;
    tmp_A1 = ((int)A1 * (int)Y[i-1] + (1 << 14)) >> 15;
    tmp_A2 = ((int)A2 * (int)Y[i-2] + (1 << 14)) >> 15;

    Y[i] = (short int)(tmp_B0 + tmp_B1 + tmp_B2 + tmp_A1 + tmp_A2);
}
```

The following table summarizes the results from manually profiling the assembly code and recording the execution time measurement.

Table 5 *Filter V0 Results*

V0 - Base	In Loop	Total
Load Instructions	24	29
Store Instructions	6	13
Branch Instructions	2	2
Multiply Instructions	5	5
Instructions	70	103
Loop Iterations	98	
Execution Time (ms)	20	
Stability Test	Fail (Limit Cycles)	

This filter implementation is not well-designed; thus, it encounters overflow often and exhibits limit cycles. These issues will be reduced /corrected using two different techniques in the following sections.

Filter V0.1: Base Filter Code with Truncation

In this version, rounding was removed and thus each multiplication result was simply truncated by the scale factor right-shift operation.

```
// Filter Loop
register int i;
for (i=2; i<100; i++) {

    tmp_B0 = ((int)B0 * (int)X[i  ] >> 15;
    tmp_B1 = ((int)B1 * (int)X[i-1] >> 15;
    tmp_B2 = ((int)B2 * (int)X[i-2] >> 15;
    tmp_A1 = ((int)A1 * (int)Y[i-1] >> 15;
    tmp_A2 = ((int)A2 * (int)Y[i-2] >> 15;

    Y[i] = (short int)(tmp_B0 + tmp_B1 + tmp_B2 + tmp_A1 + tmp_A2);
}
```

This technique, however, did not eliminate limit cycles.

Filter V0.2: Base Filter Code with Saturating Addition

In this version, a saturating addition function, provided in the lecture slides [2], was added to the base filter code and tested.

```
// Function for Saturating Addition
short int short_int_clipping ( int a) {
    int tmp ;
    tmp = a;
    if( tmp >= (int) (+32767) ){
```

```
        tmp  = (int) (+32767) ;
    }
    if( tmp <= (int) ( -32767) ){
        tmp  = (int) ( -32767) ;
    }
    return( (short int) tmp );
}

// Filter Loop
register int i;
for (i=2; i<100; i++) {

    tmp_B0 = ((int)B0 * (int)X[i  ] + (1 << 14)) >> 15;
    tmp_B1 = ((int)B1 * (int)X[i-1] + (1 << 14)) >> 15;
    tmp_B2 = ((int)B2 * (int)X[i-2] + (1 << 14)) >> 15;
    tmp_A1 = ((int)A1 * (int)Y[i-1] + (1 << 14)) >> 15;
    tmp_A2 = ((int)A2 * (int)Y[i-2] + (1 << 14)) >> 15;

    Y[i] = short_int_clipping ( tmp_B0 + tmp_B1 + tmp_B2 + tmp_A1 + tmp_A2 );

}
```

This resulted in much better performance and effectively solved the problem of limit cycles for sample input.

Filter V1: Base Custom Filter

Filter V1 uses the same implementation as V0 except that it employs all the scaling factors and coefficients calculated earlier for the filter specification (see Digital Filter Design). For convenience, the relevant code is included in the figure below.

```
// Input Coefficients (See Calculations in Report)
const short int B0 = 0x76B0; //30384
const short int B1 = 0x76B0; //30384
const short int B2 = 0x76B0; //30384
int tmp_B0, tmp_B1, tmp_B2;

// Output Coefficients (See Calculations in Report)
const short int A1 = 0x74A7; //29863
const short int A2 = 0x94D7; //-27433
int tmp_A1, tmp_A2;

register int i;
for (i=2; i<100; i++) {

    tmp_B0 = ((int)B0 * (int)X[i  ] + (1 << 23)) >> 24;
    tmp_B1 = ((int)B1 * (int)X[i-1] + (1 << 22)) >> 23;
    tmp_B2 = ((int)B2 * (int)X[i-2] + (1 << 23)) >> 24;
    tmp_A1 = ((int)A1 * (int)Y[i-1] + (1 << 13)) >> 14;
```

```
    tmp_A2 = ((int)A2 * (int)Y[i-2] + (1 << 14)) >> 15;

    Y[i] = (short int)(tmp_B0 + tmp_B1 + tmp_B2 + tmp_A1 + tmp_A2);
}
```

The following table summarizes the results from manually profiling the assembly code and recording the execution time measurement.

***Table 6** Filter V1 Results*

V1 – Base Custom	In Loop	Total
Load Instructions	24	29
Store Instructions	6	13
Branch Instructions	2	2
Multiply Instructions	5	5
Instructions	69	103
Loop Iterations	98	
Execution Time (ms)	21	
Stability Test	Pass	

As seen in the results above, this filter does not encounter the issues of overflow or limit cycles because the scale factors were carefully chosen. The performance can, however, be improved as will be seen in the next sections.

Filter V2: Filter with Inline C Function

Filter V2 makes only one small change to Filter V1: it adds the 'inline' keyword before the filter initialization function where the input array is generated so that the body of that function will be included in the calling function (main in our case), avoiding a branching operation to the filter initialization function definition. Since this function is now an inline routine, the compiler requires the use of the -static flag which simply instructs the linker to create an executable which includes all necessary libraries and dependencies within itself so that no external references are generated that could throw errors during linking.

The following table summarizes the results from manually profiling the assembly code and recording the execution time measurement.

***Table 7** Filter V2 Results*

V2 – Inline Function	In Loop	Total
Load Instructions	27	37
Store Instructions	6	18
Branch Instructions	1	5
Multiply Instructions	5	5
Instructions	69	147

Loop Iterations	98
Execution Time (ms)	20
Stability Test	Pass

As seen in the result above, this technique actually resulted in a small increase in load, store, and branch operations, and in an increase in total instruction count by 44. As a result, it was decided that instead of generating the input with filter_init() and using function inlining, the input array would be stored as a static array in the C code moving forward (see code snippet below).

```
short int X[128] = {
0x8001, 0x8001, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
};
short int Y[128]; // Previous Outputs Array (stores up to 128, 16-bit signed integers)

void main(void){ ...
```

Filter V3: Filter with Instruction Reordering

In this version the concept of instruction reordering was applied to try and produce more efficient assembly from the compiler. The idea was to interleave the multiply and the add operations in the loop (see code snippet below).

```
  for (i=2; i<100; i++) {

      tmp_B0 = ((int)B0 * (int)X[i  ] + (1 << 23)) >> 24; // Scale Factor = 2^24
      Y[i] = tmp_B0;
      tmp_B1 = ((int)B1 * (int)X[i-1] + (1 << 22)) >> 23; // Scale Factor = 2^23
      Y[i] += tmp_B1;
      tmp_B2 = ((int)B2 * (int)X[i-2] + (1 << 23)) >> 24; // Scale Factor = 2^24
      Y[i] += tmp_B2;
      tmp_A1 = ((int)A1 * (int)Y[i-1] + (1 << 13)) >> 14; // Scale Factor = 2^14
      Y[i] += tmp_A1;
      tmp_A2 = ((int)A2 * (int)Y[i-2] + (1 << 14)) >> 15; // Scale Factor = 2^15
      Y[i] += tmp_A2;
      Y[i] = (short int)Y[i]; // Recall: y[n] = Y[n] / SFy
```

```
}
```

This did not prove to be effective however, because the results of each operation were independent so there were no true dependencies which this would help to mitigate. The statistics for this attempt are therefore omitted.

Filter V4: Custom Filter Code with Two-Iteration Loop Unrolling

In this version, the concept of loop unrolling was implemented and tested by computing two output values within each loop iteration in order to halve the for-loop branching overhead. The relevant code snippet is provided below for convenience. Notice the need for extra temporary variables to store the intermediate values for the output result.

```
// Input Coefficients
const short int B0 = 0x76B0;
const short int B1 = 0x76B0;
const short int B2 = 0x76B0;
int tmp_B0, tmp_B1, tmp_B2;
int tmp_B0_nxt, tmp_B1_nxt, tmp_B2_nxt;

// Output Coefficients
const short int A1 = 0x74A7;
const short int A2 = 0x94D7;
int tmp_A1, tmp_A2;
int tmp_A1_nxt, tmp_A2_nxt;

register int i;
for (i=2; i<100; i+=2) {

    tmp_B0     = ((int)B0 * (int)X[i  ] + (1 << 23)) >> 24;
    tmp_B1     = ((int)B1 * (int)X[i-1] + (1 << 22)) >> 23;
    tmp_B2     = ((int)B2 * (int)X[i-2] + (1 << 23)) >> 24;
    tmp_A1     = ((int)A1 * (int)Y[i-1] + (1 << 13)) >> 14;
    tmp_A2     = ((int)A2 * (int)Y[i-2] + (1 << 14)) >> 15;
    Y[i]   = (short int)(tmp_B0 + tmp_B1 + tmp_B2 + tmp_A1 + tmp_A2);
    tmp_B0_nxt = ((int)B0 * (int)X[i+1] + (1 << 23)) >> 24;
    tmp_B1_nxt = ((int)B1 * (int)X[i  ] + (1 << 22)) >> 23;
    tmp_B2_nxt = ((int)B2 * (int)X[i-1] + (1 << 23)) >> 24;
    tmp_A1_nxt = ((int)A1 * (int)Y[i  ] + (1 << 13)) >> 14;
    tmp_A2_nxt = ((int)A2 * (int)Y[i-1] + (1 << 14)) >> 15;
    Y[i+1] = (short int)(tmp_B0_nxt + tmp_B1_nxt + tmp_B2_nxt + tmp_A1_nxt + tmp_A2_nxt);

}
```

The following table summarizes the results from manually profiling the assembly code and recording the execution time measurement.

Table 8 *Filter V4 Results*

V4 – Loop Unroll (x2)	In Loop	Total
Load Instructions	48	54
Store Instructions	12	19
Branch Instructions	1	3
Multiply Instructions	10	10
Instructions	135	169
Loop Iterations	49	
Execution Time (ms)	21	
Stability Test	Pass	

As seen above, the number of total instructions increased by 66, but the number of iterations is cut in half meaning less branches are taken which is an overall improvement. This trend continues in the next two versions.

Filter V5: Custom Filter Code with Four-Iteration Loop Unrolling

This version is identical to the one before except, as the name suggests, it computes four output values within each loop iteration, bringing the number of looper iterations down again. The following table summarizes the results.

Table 9 *Filter V5 Results*

V5 – Loop Unroll (x4)	In Loop	Total
Load Instructions	96	101
Store Instructions	24	31
Branch Instructions	1	3
Multiply Instructions	20	20
Instructions	270	302
Loop Iterations	25	
Execution Time (ms)	21	
Stability Test	Pass	

This further optimizes the code by reducing the number of branches taken in the code, however, it does require significantly more load and store operations for all the intermediate variables.

Filter V6: Custom Filter Code with Three-Iteration Loop Unrolling

This filter was built simply to see the result of loop-unrolling three times. The results were unsurprising and therefore omitted from this report.

Filter V7: Filter with NEON Intrinsics and Vector Operations

In this version the code was significantly refactored in order to leverage the NEON SIMD (Single Instruction Multiple Data) instruction set on the ARM Cortex-A8. The NEON

architecture provides thirty-two 128-bit vector registers, each capable of being evenly subdivided into lanes of 8-bits at the minimum and 64-bits at the maximum. NEON instructions When declaring variables with NEON intrinsic types, they get stored in these registers. [8]

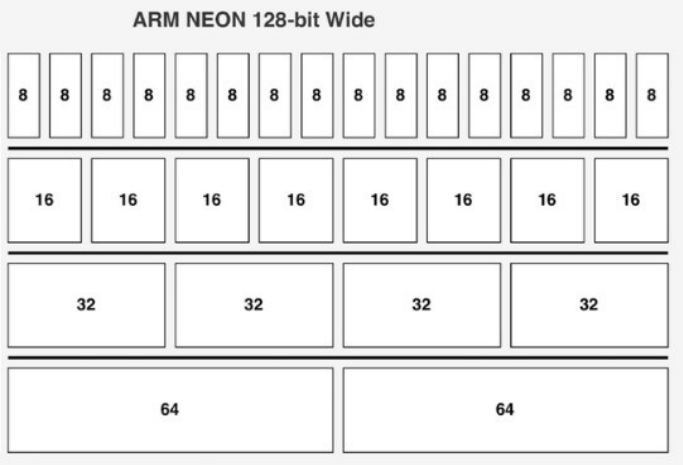


Figure 10 *NEON Register Possible Subdivisions [9]*

The goal with using NEON intrinsics is to perform vector operations. This allows for some of the computations in the digital filter algorithm to occur in parallel in order to minimize loads, stores, and multiplies. Since our digital filter deals with signed 16-bit integers, and products that could be up to 32-bits, we made use of the int16x4_t and int32x4_t NEON intrinsic types, which are vectors of four 16-bit, and four 32-bit integers respectively. The implementation below is again a four-iteration loop-unroll, but this time with the most of the multiply, accumulate, and shifting operations occurring in parallel for the four output calculations. Please refer to Appendix C for code listing demonstrating the filter implementation with NEON intrinsics.

The following table summarizes the results from manually profiling the assembly code and recording the execution time measurement.

V7 – NEON	**In Loop**	**Total**
Load Instructions	107	116
Store Instructions	54	64
Branch Instructions	1	3
Multiply Instructions	8	8
Instructions	138	353
Loop Iterations	25	
Execution Time (ms)	19	
Stability Test	Pass	

Notice that the instruction count is higher that the Base Custom Filter implementation, but execution time has actually decreased below 20ms. Another important thing to notice is that notice that the number of multiply operations was reduced from 20 to 8 compared with the Filter V5; this is because the other 12 multiplication operations have been compressed into 3 vector

multiplications. Due to these improvements this optimization method was selected to be implemented in the final version.

Filter V8: Filter with Integer Array Access and Two – Iteration Loop unrolling

This version of the code was attempting to decrease the number of load operations in each loop iteration by storing the input value arrays as 32bit integers. The 16bit input values are combined in pairs to form 32bit integers which after loading are split using two shift operations, for example, the first four input values are 0x8001, 0x8001, 0x7FFF, 0x7FFF, and can be stored as 0x80018001, 0x7FFF7FFF. When the 32-bit integer is loaded the first value (upper half word) can be extracted with a right shift by 16 bits, and the second value (lower half word) can be extracted with a left shift by 16 bits, followed by a right shift by 16 bits. In this implementation 2 temporary variables were used to store the two 32-bit integers, which will contain 4 input values for the cost of 2 load operations. Note: This filter was used in combination with two – iteration loop unrolling since the loop contained the correct number of inputs.

Table 10 *Filter V8 Results*

V8 - int array + Unroll 2	In Loop	Total
Load Instructions	46	51
Store Instructions	14	21
Branch Instructions	1	3
Multiply Instructions	10	10
Instructions	138	173
Loop Iterations	49	
Execution Time (ms)	20	
Stability Test	Pass	

Notice that there was a reduction of two load operations per iteration, but an increase of two store operations due to the use of the temporary variable. This could be corrected by not using temporary variables; however, filter version 7 already uses a similar technique by accessing four halfwords in a single load operation, therefore this technique was not investigated further.

Filter V9: Filter with Single-Line Multiply-Accumulate

In this version the code was refactored to remove all temporary variables and excessive load/store operations by combining everything into a single line.

```
Y[i] = (((int)B0 * (int)X[i] + (1 << 23)) >> 24)
     + (((int)B1 * (int)X[i-1] + (1 << 22)) >> 23)
     + (((int)B2 * (int)X[i-2] + (1 << 23)) >> 24)
     + (((int)A1 * (int)Y[i-1] + (1 << 13)) >> 14)
     + (((int)A2 * (int)Y[i-2] + (1 << 14)) >> 15);
```

Table 11 *Filter V9 Results*

V9 - Single line MAC	In Loop	Total
Load Instructions	19	24
Store Instructions	1	8
Branch Instructions	1	3
Multiply Instructions	5	5
Instructions	57	93
Loop Iterations	98	
Execution Time (ms)	20	
Stability Test	Pass	

Notice the reduction of total load store operations five loads and five stores when compared to the custom base version, and the single store operation in each loop iteration. Due to these improvements this optimization method was selected to be implemented in the final version.

Filter V10: Filter with Constant Definitions for Coefficients

In this version the code the filter coefficient variables were replaced with define statements to reduce the use of load store operations.

Table 12 *Filter V10 Results*

V10- Define Contants	In Loop	Total
Load Instructions	10	11
Store Instructions	6	8
Branch Instructions	1	3
Multiply Instructions	5	5
Instructions	75	99
Loop Iterations	98	
Execution Time (ms)	20	
Stability Test	Pass	

Notice in the above results the reduction in 18 load and five store operations when compared to the custom base version, which is the result of converting five variables into constants. Due to these improvements this optimization method was selected to be implemented in the final version.

Filter V11: Filter with Constant Definitions and Single Line Operation

In this version the code was a continuation of versions 9 and 10 to see if the improvements will add together, the results were as expected and both optimizations can be used together for added benefits.

Table 13 *Filter V11 Results*

V8 - int array + Unroll 2	In Loop	Total
Load Instructions	14	19
Store Instructions	1	3
Branch Instructions	1	3
Multiply Instructions	5	5
Instructions	65	87
Loop Iterations	98	
Execution Time (ms)	20	
Stability Test	Pass	

Notice above, within the loop there are fifteen load/store operations, which is further reduced from filter V9 and filter V10 which have twenty and 16 load store operations respectively.

Filter V12: Final Filter

In this version the code is a combination of filter versions 7, 9 and 10 as they showed the greatest improvements individually and their implementations are compatible with each other. To summarize the optimization, 1. NEON was used to parallelize the input coefficient multiplications as well as loading multiple input values in a single load operation. Additionally, the NEON implementation facilitates Four-Iteration loop unrolling while providing the previously mentioned optimization. 2. Constant definitions were used instead of variables for coefficients. 3. Temporary variables and excessive load/stores were removed by putting as many operations on a single line as possible. Additionally, all global variables were moved within the main function, and the -O3 compiler flag was used. One notable improvement made by the -O3 flag was that the assembly now uses the multiply-accumulate operator instead of a multiply followed by an add. This greatly reduced the multiply and add operations in the loop with a single operation that takes advantage of the MAC unit that is included in the ARM Cortex A8.

Table 14 *Filter V12 Results*

V8 - int array + Unroll 2	In Loop	Total
Load Instructions	21	25
Store Instructions	16	19
Branch Instructions	1	1
Multiply Instructions	4	4
Instructions	134	161
Loop Iterations	25	
Execution Time (ms)	19	
Stability Test	Pass	

The major change with this design iteration, however, was the use of the -O3 compiler optimization flag which significantly reduced the number of assembly instructions for this digital filter. Below is a table comparing the total operations of the final filter and the custom base version.

Table 15 *Improvement Comparison*

Operation	V1 Custom	V 12 Final	Percent Improvement
Load	2357	529	22.44%
Store	595	403	67.73%
Multiply	196	25	12.76%

Conclusion

There are many ways to optimize C code such that the compiler translates it into truly efficient assembly instructions for the target architecture. In this report, the design of a digital filter was presented along with multiple ways of implementing in software. The implementations demonstrated the effects on performance of a few techniques and the tradeoffs between code size and speed that must be considered when developing an embedded system. The most effective techniques based on this study which were applied in the final design of the digital filter were loop unrolling, vectorization with SIMD, compiler optimization flags, and simple ways of writing code to reduce overhead. The evolution of Filter V0 to FilterV12 (the final version) resulted in the elimination of limit cycles, a 12% reduction in multiply operations, 22% reduction in load operations, and a 68% reduction in store operations, and a 75% reduction in loop iterations.

Bibliography

[1] T. Tyson, "Physics 123 / 253," 2013. [Online]. Available: https://123.physics.ucdavis.edu/week_5_files/filters/digital_filter.pdf.

[2] M. Sima, "Digital Filters," Victoria, 2024.

[3] F. Gebali, "Filters," 2021.

[4] Texas Instruments, "AM335x Sitara Processors Datasheet," March 2020. [Online].

[5] BeagleBoard, "BeagleBone Black," Dec. 21, 2023. [Online]. Available: https://www.beagleboard.org/boards/beaglebone-black.

[6] Y. Zhang, "Arm NEON programming quick reference," 2015. [Online]. Available: https://community.arm.com/arm-community-blogs/b/operating-systems-blog/posts/arm-neon-programming-quick-reference.

[7] G. D. C. Richard M. Stallman, Using the GNU Compiler Collection For GCC version 8.3.0, Boston, MA: GNU Press, 2018.

[8] ARM Ltd., Learn the architecture - Optimizing C code with Neon intrinsics, Cambridge, England , 2024.

[9] T. K. e. al., "An energy efficient multi-target binary translator for instruction and data level parallelism exploitation," *ResearchGate,* 2022.

① Filter Specifications

→ Sampling Rate :
→ 3dB cut-off frequency :
→ Attenuation :

$f_s = 100\text{kHz}$
$f_{d_1} = 2\text{kHz}$
$A \geq 29\text{dB}$ at $f_{d_2} = 8\text{kHz}$

② Sampling Period

$$T_s = \frac{1}{f_s} = \frac{1}{100\text{kHz}} = \frac{1}{10^5 \frac{1}{s}} \Rightarrow \boxed{T_s = 10^{-5}\text{s} \quad \text{or} \quad 10\mu\text{s}}$$

③ Digital Angular Frequencies

$$\omega_{d1} = 2\pi f_{d_1} = (2\pi_{\text{rad}})(2\text{kHz}) \qquad \omega_{d2} = 2\pi f_{d_2} = (2\pi_{\text{rad}})(8\text{kHz})$$

$$\boxed{\omega_{d_1} = 4\pi \times 10^3 \frac{\text{rad}}{\text{s}} \qquad \omega_{d2} = 16\pi \times 10^3 \frac{\text{rad}}{\text{s}}}$$

④ Analog Angular Frequencies

Use Pre-Warping Transformation: $\omega_a = \frac{2}{T_s}\tan\left(\frac{\omega_d T_s}{2}\right)$

$$\omega_{a1} = \frac{2}{T_s}\tan\left(\frac{\omega_{d1} T_s}{2}\right)$$

$$= \frac{2}{10^{-5}\,s}\tan\left(\frac{(4\pi \times 10^3 \frac{rad}{s})(10^{-5}\,s)}{2}\right)$$

$$= (2\times 10^5 \tfrac{1}{s})\tan(2\pi \times 10^{-2}\,rad)$$

$$= (2\times 10^5 \tfrac{1}{s})(0.06291467\,rad)$$

$$\omega_{a2} = \frac{2}{T_s}\tan\left(\frac{\omega_{d2} T_s}{2}\right)$$

$$= \frac{2}{10^{-5}\,s}\tan\left(\frac{(16\pi \times 10^3 \frac{rad}{s})(10^{-5}\,s)}{2}\right)$$

$$= (2\times 10^5 \tfrac{1}{s})\tan(8\pi \times 10^{-2}\,rad)$$

$$= (2\times 10^5 \tfrac{1}{s})(0.25675636\,rad)$$

$$\boxed{\omega_{a1} = 12,582.933 \ \tfrac{rad}{s} \qquad \omega_{a2} = 51,351.272 \ \tfrac{rad}{s}}$$

⑤ Analog Frequencies

$$f_{a1} = \frac{\omega_{a1}}{2\pi} = \frac{12,582.933\,rad/s}{2\pi\,rad} \Rightarrow \boxed{f_{a1} = 2002.64\,Hz \approx 2.003\,kHz}$$

$$f_{a2} = \frac{\omega_{a2}}{2\pi} = \frac{51,351.272\,rad/s}{2\pi\,rad} \Rightarrow \boxed{f_{a2} = 8172.81\,Hz \approx 8.173\,kHz}$$

⑥ Order of the filter

$$\text{Order: } N \geq \frac{\log\left(\frac{\sqrt{10^{0.1A_s}-1}}{\sqrt{10^{0.1A_p}-1}}\right)}{\log\left(\frac{f_{a_2}}{f_{a_1}}\right)}$$

A_s : Stopband Attenuation in dB = 29dB

A_p : Passband Ripple in dB = 6dB

↑ we choose reasonable values such the the filter order is kept at 2

$$N \geq \frac{1.2125}{0.6107}$$

$$N \geq 1.9855 \implies \boxed{N = 2}$$

⑦ Transfer Function (continuous-time domain)

The transfer function of a second-order Butterworth low-pass filter is:

$$H(s) = \frac{\omega_c^2}{s^2 + \sqrt{2}\,\omega_c s + \omega_c^2}, \text{ where } \omega_c = \omega_{a_1} = \frac{2}{T_s}(0.06291467\ \text{rad})$$

⑧ Transfer Function (discrete-time domain)

To obtain the transfer function in z, use the bilinear transformation: $s = \frac{2}{T_s}\frac{1-z^{-1}}{1+z^{-1}}$

$$H(z) = \frac{\left(\frac{2}{T_s}\right)^2 (0.06291467\ \text{rad})^2}{\left(\frac{2}{T_s}\right)^2\left(\frac{1-z^{-1}}{1+z^{-1}}\right)^2 + \sqrt{2}\left(\frac{2}{T_s}\right)(0.06291467\ \text{rad})\left(\frac{2}{T_s}\right)\left(\frac{1-z^{-1}}{1+z^{-1}}\right) + \left(\frac{2}{T_s}\right)^2 (0.06291467\ \text{rad})^2}$$

factor out $\left(\frac{2}{T_s}\right)^2$

$$H(z) = \frac{(0.00395826\ \text{rad}^2)}{\left(\frac{1-z^{-1}}{1+z^{-1}}\right)^2 + (0.08897478\ \text{rad})\left(\frac{1-z^{-1}}{1+z^{-1}}\right) + (0.00395826\ \text{rad}^2)}$$

factor out z^{-1}

$$H(z) = \frac{(0.00395826)}{\left(\frac{z-1}{z+1}\right)^2 + (0.08897478)\left(\frac{z-1}{z+1}\right) + (0.00395826)}$$

$\times \frac{z+1}{z+1}$ $\quad \times \left(\frac{z+1}{z+1}\right)^2$

$$H(z) = \frac{(0.00395826)}{\frac{(z-1)^2}{(z+1)^2} + \frac{(0.08897478)(z^2-1)}{(z+1)^2} + \frac{(0.00395826)(z+1)^2}{(z+1)^2}}$$

$$H(z) = \frac{(0.00395826)}{\frac{z^2-2z+1 + 0.08897478z^2 - 0.08897478 + (0.00395826)(z^2+2z+1)}{(z+1)^2}}$$

α

$$H(z) = \frac{(0.00395826)(z+1)^2}{1.08897478z^2 - 2z + 0.91102522 + \alpha z^2 + 2\alpha z + \alpha}$$

$$H(z) = \frac{(0.00395826)(z^2+2z+1)}{1.09293304z^2 - 1.99208348\ z + 0.91498348}$$

$$H(z) = \frac{0.00395826}{1.09293304} \cdot \frac{z^2+2z+1}{z^2 - 1.822695\ z + 0.837182}$$

$$\boxed{H(z) = (0.003622)\ \frac{1 + 2z^{-1} + z^{-2}}{1 - 1.822695z^{-1} + 0.837182z^{-2}}}$$

⑨ Difference Equation

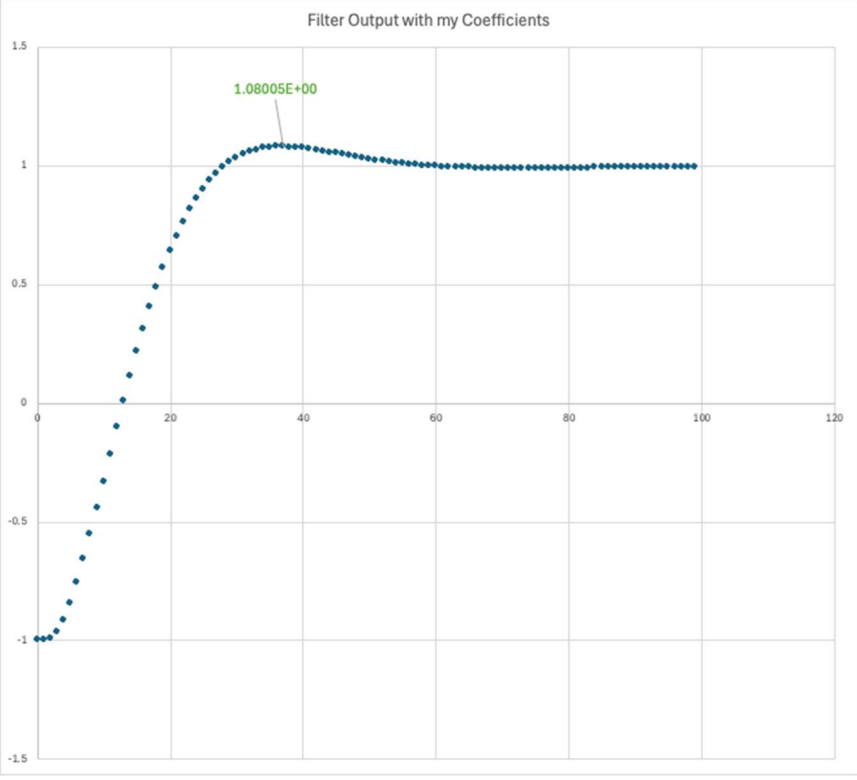
$$\frac{Y(z)}{X(z)} = \underbrace{(0.003622)}_{\beta} \frac{1 + 2z^{-1} + z^{-2}}{1 - 1.822695z^{-1} + 0.837182z^{-2}}$$

$$Y(z)\left(1 - 1.822695z^{-1} + 0.837182z^{-2}\right) = \beta X(z)\left(1 + 2z^{-1} + z^{-2}\right)$$

$$Y(z) - 1.822695z^{-1}Y(z) + 0.837182z^{-2}Y(z) = \beta X(z) + 2\beta z^{-1}X(z) + \beta z^{-2}X(z)$$

$$Y(z) = (0.003622)X(z) + (0.007244)z^{-1}X(z) + (0.003622)z^{-2}X(z) + 1.822695z^{-1}Y(z) - 0.837182z^{-2}Y(z)$$

$$y[n] = \underset{b_0}{0.003622}\,x[n] + \underset{b_1}{0.007244}\,x[n-1] + \underset{b_2}{0.003622}\,x[n-2] + \underset{a_1}{1.822695}\,y[n-1] - \underset{a_2}{0.837182}\,y[n-2]$$



⑩ Coefficients

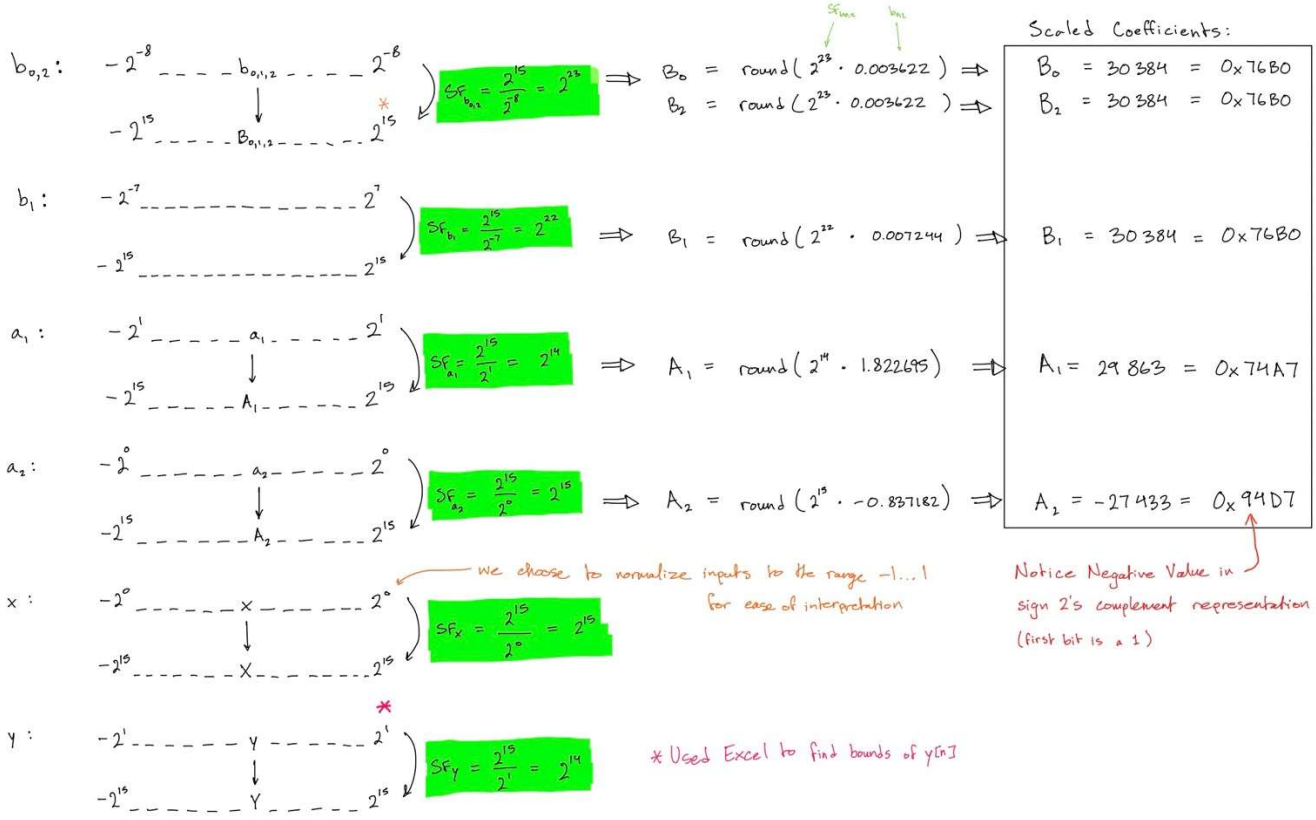
$b_0 = 0.003622$ *
$b_1 = 0.007244$ ⊛
$b_2 = 0.003622$ *

⊛ $a_1 = 1.822695$
* $a_2 = -0.837182$

Nearest power of 2 for each coefficient

* $2^{-8} = 0.00390625$
⊛ $2^{-7} = 0.0078125$
* $2^{0} = 1$
⊛ $2^{1} = 2$

* ∵ we are dealing with 16-bit word length and signed values



⑪ Scaled Difference Equation

$$\frac{Y[n]}{SF_y} = \frac{B_0}{SF_{b_0}} \cdot \frac{X[n]}{SF_x} + \frac{B_1}{SF_{b_1}} \cdot \frac{X[n-1]}{SF_x} + \frac{B_2}{SF_{b_2}} \cdot \frac{X[n-2]}{SF_x} + \frac{A_1}{SF_{a_1}} \cdot \frac{Y[n-1]}{SF_y} + \frac{A_2}{SF_{a_2}} \cdot \frac{Y[n-2]}{SF_y}$$

$$\frac{Y[n]}{2^{14}} = \frac{0x7680}{2^{23}} \cdot \frac{X[n]}{2^{15}} + \frac{0x7680}{2^{22}} \cdot \frac{X[n-1]}{2^{15}} + \frac{0x7680}{2^{23}} \cdot \frac{X[n-2]}{2^{15}} + \frac{0x74A7}{2^{14}} \cdot \frac{Y[n-1]}{2^{14}} + \frac{0x94D7}{2^{15}} \cdot \frac{Y[n-2]}{2^{14}}$$

$$Y[n] = \frac{0x7680 \cdot X[n]}{2^{24}} + \frac{0x7680 \cdot X[n-1]}{2^{23}} + \frac{0x7680 \cdot X[n-2]}{2^{24}} + \frac{0.74A7 \cdot Y[n-1]}{2^{14}} + \frac{0x94D7 \cdot Y[n-2]}{2^{15}}$$

Appendix B: MATLAB Script for Frequency Response Verification

Clear Workspace

```
clear;
```

Specify the Discrete-Time Transfer Function

```
K     = 0.003622;
num   = K*[1 2 1];
denom = [1 -1.822695 0.837182];
Ts    = 10^-5


filterTF = tf(num, denom, Ts)
```

Configure Bode Plot Settings

```
% Display Properties
plotSettings = bodeoptions;


plotSettings.Title.String = 'Frequency Response of the Low-Pass Butterowrth Digital Filter';
plotSettings.Title.FontSize = 16;
plotSettings.XLabel.FontSize = 13;
plotSettings.YLabel.FontSize = 13;
plotSettings.TickLabel.FontSize = 10;
plotSettings.FreqUnits = 'kHz';
plotSettings.Grid = 'on';


% Frequency-Axis Limits for Clearer Presentation
fmin = 100;       % Hz
fmax = 45000;     % Hz
wmin = fmin*2*pi; % rad/s
wmax = fmax*2*pi; % rad/s
```

Generate Bode Plot

```
figure;
h = bodeplot(filterTF, {wmin,wmax}, plotSettings);
```

Bode Plot Annotations

```
% Add red lines to highlight frequencies of interest (fd1=2kHz, fd2=8kHz)
% Add green line to indicate target attenuation (-29dB at 8kHz)


% Note: gcf = "get current figure": Built-in MATLAB function,
% returns the handle to the current figure.
```

```
% Get the handles to the magnitude and phase plot axes
[mag_ax, phase_ax] = findBodeAxes(gcf);


% Define the frequencies to add vertical lines
highlight_frequencies = [2, 8]; % Frequencies in kHz (from plotSettings)


% Add red vertical lines at the specified frequencies on both plots
for f = highlight_frequencies
    if f == 2
        label = 'fd1';
    else
        label = 'fd2';
    end
    xline(mag_ax, f, 'r', {label}, 'LineWidth', 1);
    xline(phase_ax, f, 'r', {label}, 'LineWidth', 1);
end


% Add green horizontal line at -29 dB on magnitude plot
yline(mag_ax, -29, 'm', {'Attenuation (-29dB)'}, 'LineWidth', 1);
yline(mag_ax, -3, 'm', {'Attenuation (-3dB)'}, 'LineWidth', 1);


% Print Crossover Frequencies and Gain/Phase Margins
[Gm,Pm,Fgc,Fpc] = margin(filterTF)
```

Function the returns handles for the two axes of a Bode Plot figure

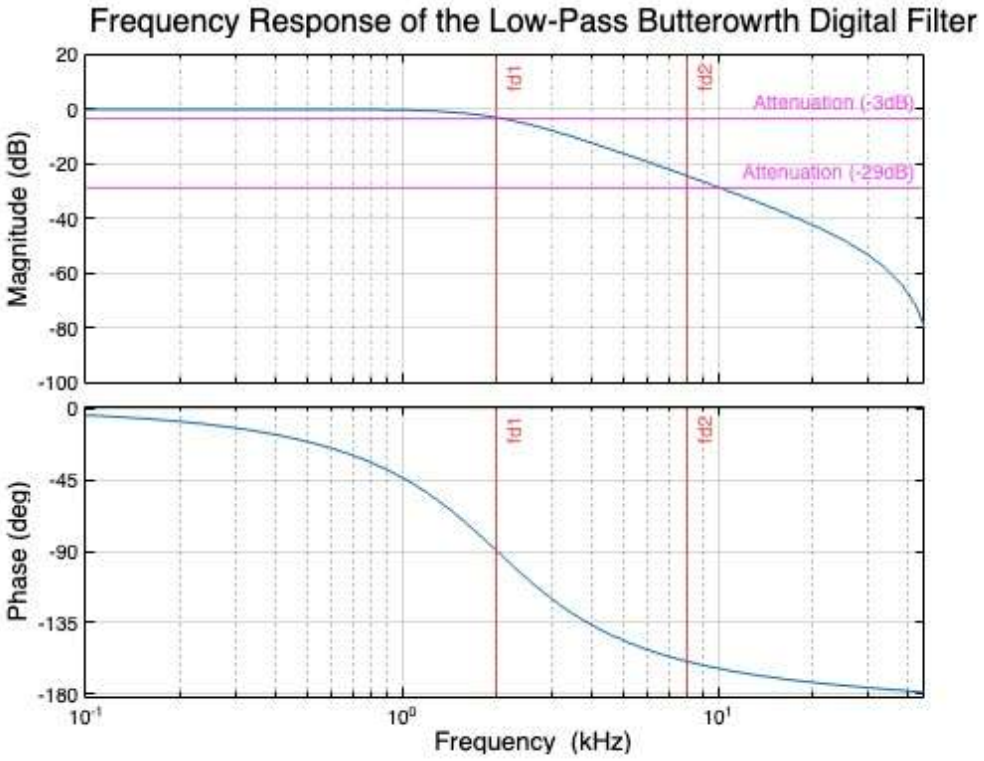
```
function [mag_ax, phase_ax] = findBodeAxes(fig)
    ax = findall(fig, 'Type', 'axes');
    for k = 1:length(ax)
        if contains(ax(k).YLabel.String, 'Magnitude')
            mag_ax = ax(k);
        elseif contains(ax(k).YLabel.String, 'Phase')
            phase_ax = ax(k);
        end
    end
end
```

Output

```
Ts = 1.0000e-05
filterTF =

  0.003622 z^2 + 0.007244 z + 0.003622
  ------------------------------------
         z^2 - 1.823 z + 0.8372

Sample time: 1e-05 seconds
Discrete-time transfer function.
Model Properties
```




```
Gm = Inf
Pm = 171.1740
Fgc = NaN
Fpc = 1.3653e+03
```

Appendix C: Filter V7 Code Listing

```
/* SENG 440 - Digital Filters Final Project */

// Compile in the terminal using the following commands:
// gcc -S -static -mfpu=neon filter.c -o filter_nonO3.s
// gcc -S -static -mfpu=neon -O3 filter.c -o filter_O3.s

# include <stdio.h>
#include <arm_neon.h>

// Initalize Input ad Output Arrays
short int X[128] = {
0x8001, 0x8001, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
};
short int Y[128]; // Previous Outputs Array (stores up to 128, 16-bit signed integers)

void main(void){

    // Initial Conditions
    Y[0] = (short int)0xC000; //-16384 --> Normalized (Y[0] / 2^14) to y[0] = -1
    Y[1] = (short int)0xC000; //-16384 --> Normalized (Y[1] / 2^14) to y[1] = -1

    // Display initial values of the output array (scaled decimal, scaled hex, unscaled decimal)
    printf("Y[ 0] = %+6hi = 0x%04hX ....... y[ 0] = %8.5f\n", Y[0], Y[0], ((float)Y[0]) / 16384);
    printf("Y[ 1] = %+6hi = 0x%04hX ....... y[ 1] = %8.5f\n", Y[1], Y[1], ((float)Y[1]) / 16384);

    // Input Coefficients (See Calculations in Report)
    const int16x4_t B0 = vdup_n_s16(0x76B0); //30384
    const int16x4_t B1 = vdup_n_s16(0x76B0); //30384
    const int16x4_t B2 = vdup_n_s16(0x76B0); //30384
    int32x4_t tmp_B0_combined, tmp_B1_combined, tmp_B2_combined; // 32bit x 4 NEON q-registers to store MAC
results for each iteration
    int16x4_t x_curr, x_prev1, x_prev2;                    // 16bit x 4 NEON d-registers to store input values for each
iteration
```

```
    // Output Coefficients (See Calculations in Report)
    const short int A1 = 0x74A7; //29863
    const short int A2 = 0x94D7; //-27433
    int tmp_A1, tmp_A2;
    int tmp_A1_nxt1, tmp_A2_nxt1;
    int tmp_A1_nxt2, tmp_A2_nxt2;
    int tmp_A1_nxt3, tmp_A2_nxt3;

    // Compute the scaled output Y[n] for all n beyond initial conditions (from 2 to 99)
    register int i;
    for (i=2; i<102; i+=4) {

        // Load current and previous inputs and outputs into NEON registers
        x_curr  = vld1_s16(&X[i]);      // 4 values starting from i (16-bits each)
        x_prev1 = vld1_s16(&X[i - 1]);  // 4 values starting from i-1 (16-bits each)
        x_prev2 = vld1_s16(&X[i - 2]);  // 4 values starting from i-2 (16-bits each)

        // MAC Computations for BX values
        tmp_B0_combined = vshrq_n_s32(vmlal_s16(vdupq_n_s32(1 << 23), B0, x_curr), 24);  //x_curr is 4 different 16-bit values, B is 4 identical 16-bit values, vdupq_n_s32(1 << 23) is 4 identical 32-bit values (result register for multiply-accumulates with rounding bits)
        tmp_B1_combined = vshrq_n_s32(vmlal_s16(vdupq_n_s32(1 << 22), B1, x_prev1), 23);
        tmp_B2_combined = vshrq_n_s32(vmlal_s16(vdupq_n_s32(1 << 23), B2, x_prev2), 24);

        // Store Results for BX values into arrays in memory
        int tmp_B0[4], tmp_B1[4], tmp_B2[4];
        vst1q_s32(tmp_B0, tmp_B0_combined);
        vst1q_s32(tmp_B1, tmp_B1_combined);
        vst1q_s32(tmp_B2, tmp_B2_combined);

        // Compute the scaled output for iteration i (Y[i])
        tmp_A2      = ((int)A2 * (int)Y[i-2] + (1 << 14)) >> 15;
        tmp_A1      = ((int)A1 * (int)Y[i-1] + (1 << 13)) >> 14;
        Y[i]        = (short int)(tmp_B0[0] + tmp_B1[0] + tmp_B2[0] + tmp_A1 + tmp_A2);

        // Compute the scaled output for iteration i+1 (Y[i+1])
        tmp_A2_nxt1 = ((int)A2 * (int)Y[i-1] + (1 << 14)) >> 15;
        tmp_A1_nxt1 = ((int)A1 * (int)Y[i  ] + (1 << 13)) >> 14;
        Y[i+1]      = (short int)(tmp_B0[1] + tmp_B1[1] + tmp_B2[1] + tmp_A1_nxt1 + tmp_A2_nxt1);

        // Compute the scaled output for iteration i+2 (Y[i+2])
        tmp_A2_nxt2 = ((int)A2 * (int)Y[i  ] + (1 << 14)) >> 15;
        tmp_A1_nxt2 = ((int)A1 * (int)Y[i+1] + (1 << 13)) >> 14;
        Y[i+2]      = (short int)(tmp_B0[2] + tmp_B1[2] + tmp_B2[2] + tmp_A1_nxt2 + tmp_A2_nxt2);

        // Compute the scaled output for iteration i+3 (Y[i+3])
        tmp_A2_nxt3 = ((int)A2 * (int)Y[i+1] + (1 << 14)) >> 15;
```

```
        tmp_A1_nxt3 = ((int)A1 * (int)Y[i+2] + (1 << 13)) >> 14;
        Y[i+3]      = (short int)(tmp_B0[3] + tmp_B1[3] + tmp_B2[3] + tmp_A1_nxt3 + tmp_A2_nxt3);

        // Display output for each iteration
        printf( "Y[%2d] = %+6hi = 0x%04hX ..... y[%2d] = %8.5f\n", i, Y[i], Y[i], i, ((float)Y[i])/16384 );          // SFy =
2^14;
        printf( "Y[%2d] = %+6hi = 0x%04hX ..... y[%2d] = %8.5f\n", i+1, Y[i+1], Y[i+1], i+1, ((float)Y[i+1])/16384 ); // SFy
= 2^14;
        printf( "Y[%2d] = %+6hi = 0x%04hX ..... y[%2d] = %8.5f\n", i+2, Y[i+2], Y[i+2], i+2, ((float)Y[i+2])/16384 ); // SFy
= 2^14;
        printf( "Y[%2d] = %+6hi = 0x%04hX ..... y[%2d] = %8.5f\n", i+3, Y[i+3], Y[i+3], i+3, ((float)Y[i+3])/16384 ); // SFy
= 2^14;

    }

} /* main */
```

Appendix D: Filter V12 Code Listing (Final Design)

```
/* SENG 440 - Digital Filters Final Project */

// Compile in the terminal using the following commands:
// gcc -S -static -mfpu=neon filter.c -o filter_nonO3.s
// gcc -S -static -mfpu=neon -O3 filter.c -o filter_O3.s

#include <stdio.h>
#include <arm_neon.h>

#define B0 0x76B0
#define B1 0x76B0
#define B2 0x76B0
#define A1 29863
#define A2 -27433

// Initalize Input ad Output Arrays
short int X[128] = {
0x8001, 0x8001, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF, 0x7FFF,
};
short int Y[128]; // Previous Outputs Array (stores up to 128, 16-bit signed integers)

void main(void){
    // Initial Conditions
    Y[0] = (short int)0xC000; //-16384 --> Normalized (Y[0] / 2^14) to y[0] = -1
    Y[1] = (short int)0xC000; //-16384 --> Normalized (Y[1] / 2^14) to y[1] = -1

    // Display initial values of the output array (scaled decimal, scaled hex, unscaled decimal)
    printf("Y[ 0] = %+6hi = 0x%04hX ....... y[ 0] = %8.5f\n", Y[0], Y[0], ((float)Y[0]) / 16384);
    printf("Y[ 1] = %+6hi = 0x%04hX ....... y[ 1] = %8.5f\n", Y[1], Y[1], ((float)Y[1]) / 16384);

    // Compute the scaled output Y[n] for all n beyond initial conditions (from 2 to 99)
    register int i;
```

```
   for (i=2; i<102; i+=4) {
      // Store Results for BX values into arrays in memory
      int tmp_B0[4], tmp_B1[4], tmp_B2[4];
      vst1q_s32(tmp_B0, vshrq_n_s32(vmlal_s16(vdupq_n_s32(1 << 23), vdup_n_s16(B0), vld1_s16(&X[i    ])), 24));
      vst1q_s32(tmp_B1, vshrq_n_s32(vmlal_s16(vdupq_n_s32(1 << 22), vdup_n_s16(B1), vld1_s16(&X[i - 1])), 23));
      vst1q_s32(tmp_B2, vshrq_n_s32(vmlal_s16(vdupq_n_s32(1 << 23), vdup_n_s16(B2), vld1_s16(&X[i - 2])), 24));

      // Compute the scaled output for iteration i (Y[i])
      Y[i]   = (short int)(tmp_B0[0] + tmp_B1[0] + tmp_B2[0] + (((int)A1 * (int)Y[i-1] + (1 << 13)) >> 14) + (((int)A2 *
(int)Y[i-2] + (1 << 14)) >> 15));
      Y[i+1] = (short int)(tmp_B0[1] + tmp_B1[1] + tmp_B2[1] + (((int)A1 * (int)Y[i  ] + (1 << 13)) >> 14) + (((int)A2 *
(int)Y[i-1] + (1 << 14)) >> 15));
      Y[i+2] = (short int)(tmp_B0[2] + tmp_B1[2] + tmp_B2[2] + (((int)A1 * (int)Y[i+1] + (1 << 13)) >> 14) + (((int)A2 *
(int)Y[i  ] + (1 << 14)) >> 15));
      Y[i+3] = (short int)(tmp_B0[3] + tmp_B1[3] + tmp_B2[3] + (((int)A1 * (int)Y[i+2] + (1 << 13)) >> 14) + (((int)A2 *
(int)Y[i+1] + (1 << 14)) >> 15));

      // Display output for each iteration
      printf( "Y[%2d] = %+6hi = 0x%04hX ..... y[%2d] = %8.5f\n", i, Y[i], Y[i], i, ((float)Y[i])/16384 );          // SFy =
2^14;
      printf( "Y[%2d] = %+6hi = 0x%04hX ..... y[%2d] = %8.5f\n", i+1, Y[i+1], Y[i+1], i+1, ((float)Y[i+1])/16384 ); // SFy
= 2^14;
      printf( "Y[%2d] = %+6hi = 0x%04hX ..... y[%2d] = %8.5f\n", i+2, Y[i+2], Y[i+2], i+2, ((float)Y[i+2])/16384 ); // SFy
= 2^14;
      printf( "Y[%2d] = %+6hi = 0x%04hX ..... y[%2d] = %8.5f\n", i+3, Y[i+3], Y[i+3], i+3, ((float)Y[i+3])/16384 ); // SFy
= 2^14;
   }

} /* main */
```