High Performance FIR and IIR Filter Implementations for the Arm Cortex A7 Processor

SENG 440 — Embedded Systems, Group 18

Khephren Gould, V01012827, khephrengould@uvic.ca
Halle Koyanagi, V00913278, hkoyanagi@uvic.ca

August 9, 2026

Abstract

This report presents the design and optimization of finite impulse response (FIR) and infinite impulse response (IIR) digital filters for a fixed-point ARM Cortex-A7 target, implemented to isolate the resonant frequency of a cantilever beam from accelerometer data. A fourth-order Butterworth lowpass filter was designed and realized in both direct-form FIR and IIR structures. Baseline floating-point implementations were converted to fixed-point arithmetic to reduce computational cost, revealing dynamic range and overflow constraints that produced limit-cycle instability in the direct-form IIR filter; this was resolved by cascading the design into second-order sections (biquads), which reduced the required coefficient dynamic range and suppressed the resulting oscillations. A series of optimization techniques were then applied and evaluated, including loop unrolling with restricted pointers, multiply-accumulate (MAC) instructions, software pipelining, and ARM NEON SIMD vectorization. Performance was assessed on a QEMU-emulated Cortex-A7 using instruction counts, cache miss rates, and cycle-count speedup relative to a software floating-point baseline. The software-pipelined and NEON-vectorized IIR implementations achieved the largest speedups, at 27.2x and 33.5x respectively, demonstrating that exploiting instruction-level parallelism and the processor's dual issue slots is substantially more effective than fixed-point conversion alone for recursive filter structures.

Contents

List of Figures

List of Tables

1 Introduction

This project investigated the implementation and optimization of digital filters on the ARM Cortex-A7 processor. The following sections discuss the background of digital filtering theory, the project structure and UML diagrams, the initial implementation of FIR and IIR filters, and the optimization techniques applied to improve performance.

The goal of the project was to design and optimize lowpass digital filters for accelerometer data from a cantilever beam. The filters were intended to preserve the beam's resonant frequency while attenuating higher-frequency components. Both FIR and IIR filters were implemented so that their numerical behaviour, computational cost, and optimization potential could be compared.

The implementations were developed for a fixed-point ARM target using an emulated Cortex-A7 environment. A baseline floating-point implementation of the FIR filter was created first, then converted to fixed-point arithmetic before applying optimizations such as loop unrolling, multiply-accumulate instructions, software pipelining, and NEON SIMD vectorization. The final implementations were evaluated using timing measurements, instruction counts, cache behaviour, and output plots.

Table 1: Team member contributions

Team Member	**Contributions**
Halle Koyanagi	Initial design planning and UML; naive floating-point FIR; fixed-point FIR; overflow handling and saturating arithmetic; MAC and assembly inlining.
Khephren Gould	Test data and filter theory; fixed-point fourth-order IIR; cascaded biquad implementation; software pipelining; loop unrolling; NEON SIMD optimization.

2 Background

The project background connects the initial UML-driven design work with the digital filter theory, target application, and ARM-based development environment used throughout the implementation.

2.1 Digital Filter Theory

2.1.1 Digital Signal Processing

The theoretical background of digital filters is in discrete time signals and systems theory. A discrete time LTI system is a transformation with the following properties:

- **Linearity:** $T\{ax_1[n] + bx_2[n]\} = aT\{x_1[n]\} + bT\{x_2[n]\}$.
- **Time invariance:** $x[n - n_0] \to y[n - n_0]$ whenever $x[n] \to y[n]$.
- **Convolution:** The output is determined by the input and impulse response of the system as $y[n] = x[n] * h[n] = \sum_{k=-\infty}^{\infty} x[k]h[n - k]$.

The impulse response is the output of the transformation when the input is the Kronecker delta $\delta[n]$. A digital filter can be described as an LTI system where the output of the filter is the discrete-time convolution of the input with the filter's impulse response, producing the difference equation for the filter. The filter is applied by computing the difference equation for each input sample. Note that digital filter design is often performed in the z-domain. The z-domain representation of the filter's impulse response is obtained by computing the z-transform:

$$H(z) = \mathcal{Z}h[n] = \sum_{n=-\infty}^{\infty} h[n]z^{-n} \tag{1}$$

In the z-domain, application of the filter involves multiplying the z-transform of the input by the filter's transfer function $H(z)$. Note that $H(z)$ is a rational function of z, where the roots of the numerator polynomial are called zeros" and the roots of the denominator polynomial are called poles". Two important properties of the z-transform are listed below:

- **Time shift property:**

$$h[n - n_0] \xrightarrow{\mathcal{Z}} z^{-n_0} H(z) \tag{2}$$

- **BIBO stability condition:** All poles of $H(z)$ must lie within the unit circle:

$$|p_i| < 1 \qquad \forall i \tag{3}$$

where p_i is the i^{th} pole of $H(z)$. The z^{-n} terms in the transfer function correspond to time delays.

2.1.2 Filter Types

Often, filters are designed according to the characteristics of their frequency response. For example, a high pass filter is a filter that attenuates frequencies below a certain cutoff whereas a low pass filter attenuates frequencies above the cutoff. The frequency response of an ideal low pass filter is shown in figure 1 [1].

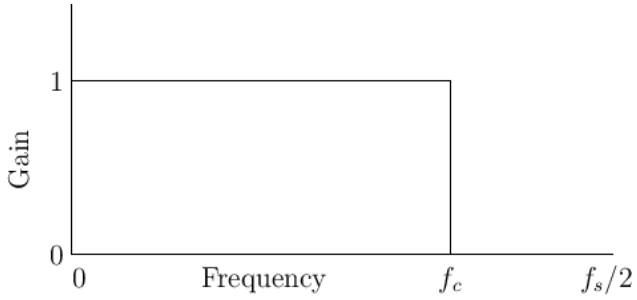

Figure 1: Ideal Low Pass Filter Frequency Response

In practice, it is impossible to obtain a filter with this frequency response. Tradeoffs must be made according to the applications requirements. In practice, filters are often designed according to transfer functions known to exhibit a desirable behavior. For example, a Chebyshev filter exhibits a sharp transition between the pass band and the stop band whereas a Butterworth filter does not distort frequencies within its pass band but has a larger transition between the pass and stop bands. This project focuses on the design of a Butterworth low pass filter [2].

2.1.3 FIR Filters

The difference equation for an FIR filter is shown in equation 4.

$$y[n] = \sum_{k=0}^{M-1} b_k x[n - k] \tag{4}$$

M is the number of filter taps and b_k are the coefficients of the filter's impulse response. Note that the difference equation does not have output terms (does not have feedback). This corresponds to a transfer function with no poles and thus an FIR filter is guaranteed to be stable. As discussed in section 3.1, typical FIR filter designs require M on the order of 50 to 100 and can even require thousands of taps to achieve adequate performance.

2.1.4 IIR Filters

In general, the difference equation for the IIR filter is given by equation 5.

$$y[n] = \sum_{k=0}^{M} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k] \tag{5}$$

M denotes the number of "feedforward" coefficients b_k and N denotes the number of "feedback" coefficients a_k. The feedback term implies the presence of poles in the IIR filter's transfer function. Thus, IIR filter's are subject to the stability condition that all poles must lie within the unit circle. The main advantage of an IIR filter is the reduced number of taps required to achieve performance comparable to the FIR filter.

2.1.5 Roundoff Error, Overflow, Limit Cycles and Instability

Both FIR and IIR filters are susceptible to roundoff error and overflow. Roundoff error is the error that results when a real valued quantity is rounded such that it fits in a certain word length. It occurs when rounding is applied to the result of the multiplication between the input and filter coefficients as shown below:

Rounding is applied because the error incurred is lower than if the result is truncated. Overflow can occur when the summations in equations 4 and 5 are computed. It is desirable for a 32 bit machine to constrain the variables used to store the summations to 32 bits. For 16 bit representations of the input, output and coefficients 32 bits is sufficient to guarantee the multiplication will not overflow however in general, adding two 32 bit numbers can result in overflow. In recursive (IIR) filters, these factors can cause limit cycles. Limit cycles are oscillations in the output of the filter that persist even if the input decays to zero. Limit cycles are described by both a frequency and an amplitude (deadband). The deadband for a two-pole filter with complex-conjugate poles is given by

$$\Delta = \frac{2^{-b-1}}{1 - |a_2|} \tag{6}$$

which depends solely on the magnitude of the coefficient a_2.

The coefficient a_1 sets the oscillation frequency. Typically, this frequency is near the filter's cutoff or resonant frequency [3].

2.2 Project Design and UML Diagrams

Prior to implementation, UML diagrams were created to communicate the intended structure and processing flow of the system. The fixed-point activity diagrams in Figure 2 describe the FIR and IIR processing steps, while the module diagram in Figure 3 shows the relationship between the main software components. These diagrams were used to establish the initial project scope and provide a shared reference for the team. As implementation progressed, requirements were iterated and the diagrams were continuously updated to reflect changes in the design. Components were added, removed, or simplified as the system became better defined. The diagrams shown in Figures 2 and 3 are the final versions, representing the completed processing flow and module structure of the implementation.

The activity diagrams in Figure 2 show the processing flow for the fixed-point FIR and IIR implementations that were used as a base for our optimizations. Both diagrams begin with the same initial stages: loading the accelerometer data, loading the filter coefficients, and converting the required values into fixed-point integer representations. The FIR activity diagram represents the nested coefficient loop explicitly, as each output sample is computed by iterating through the FIR coefficients, checking whether the required input sample is available, accumulating valid products, applying the scale factor shift, and storing the resulting output. In contrast, the IIR activity diagram is presented at a higher level to improve readability and to emphasize the structural differences between the FIR and IIR implementations. Rather than showing each nested coefficient loop, the IIR diagram summarizes the feedforward and feedback computations, scale-factor alignment, subtraction of the feedback contribution, rounding and shifting, and storage of the output sample. This abstraction keeps the diagram readable while still communicating the additional feedback path and scaling considerations introduced by the IIR implementation.

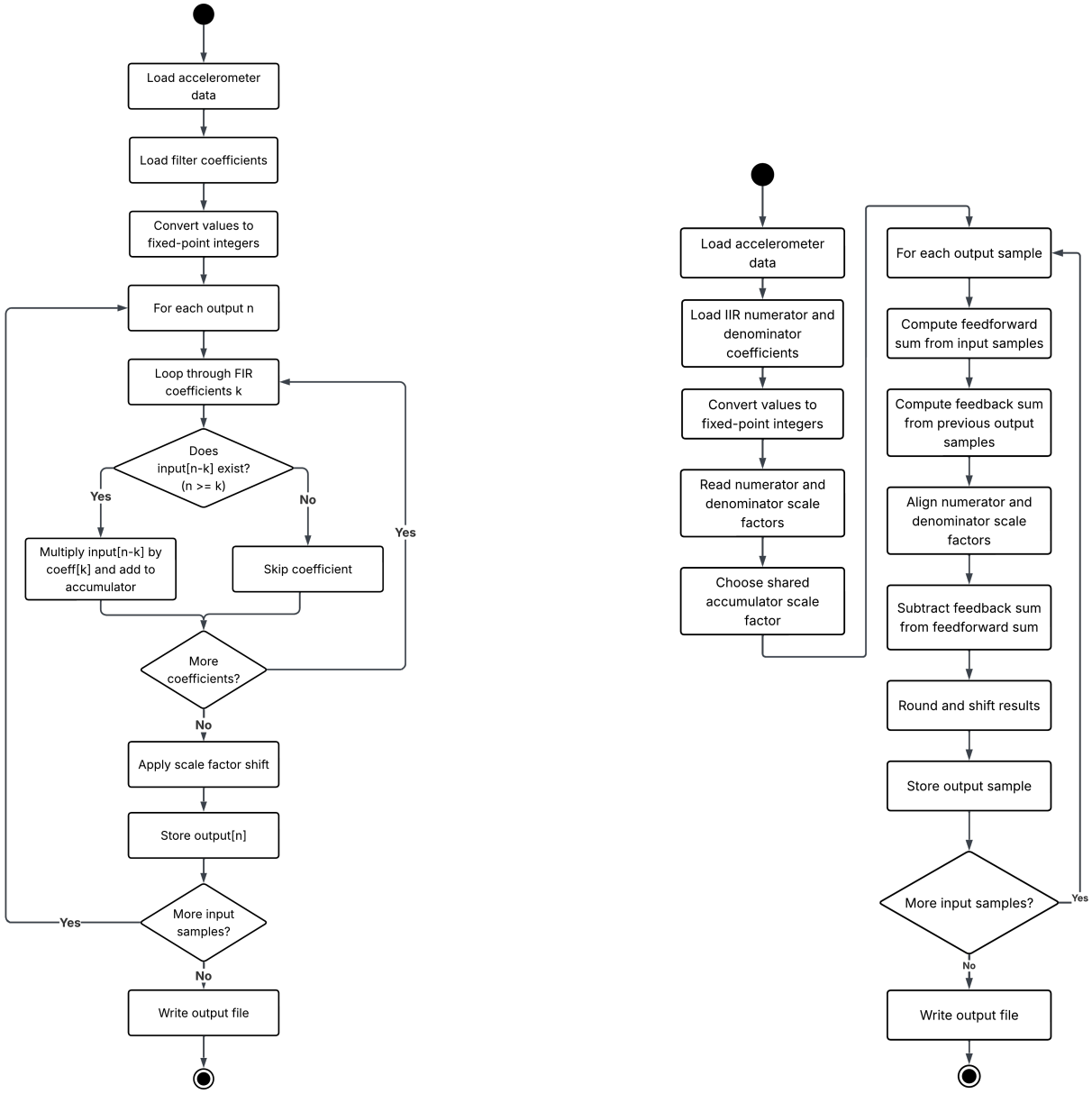

Figure 2: Fixed-point activity diagrams: FIR (left) and IIR (right)

The module diagram in Figure 3 was used to drive the organization of the implementation by identifying the major functional components and the relationships between them. This provided a basis for separating responsibilities and priorities in the codebase, such as loading input data, loading coefficients, converting values to fixed-point representations, applying the FIR and IIR filters, handling overflow, and recording timing information. Defining these components early helped guide how the implementation was structured and made it easier to extend the project as new versions of the filters were added. As the design evolved, the module diagram was updated to match the final scope and to reflect the completed organization of the system.

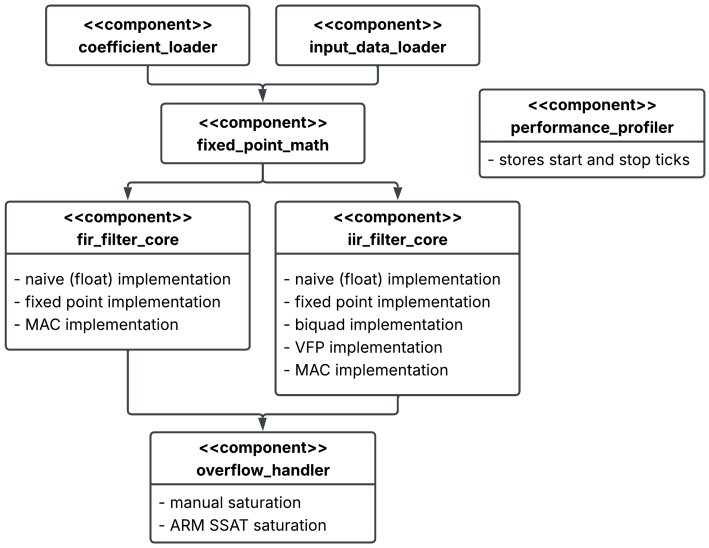

Figure 3: Digital filters module diagram

2.3 Target Application

The objective of this project is to design a Butterworth lowpass filter to isolate the resonant frequency of a cantilever beam. Sample data is obtained from an open-source dataset in which a cantilever beam was struck with a hammer. An accelerometer was placed on the beam which produced the data shown in Figure 4 (note that the data has been normalized between -1 and 1).

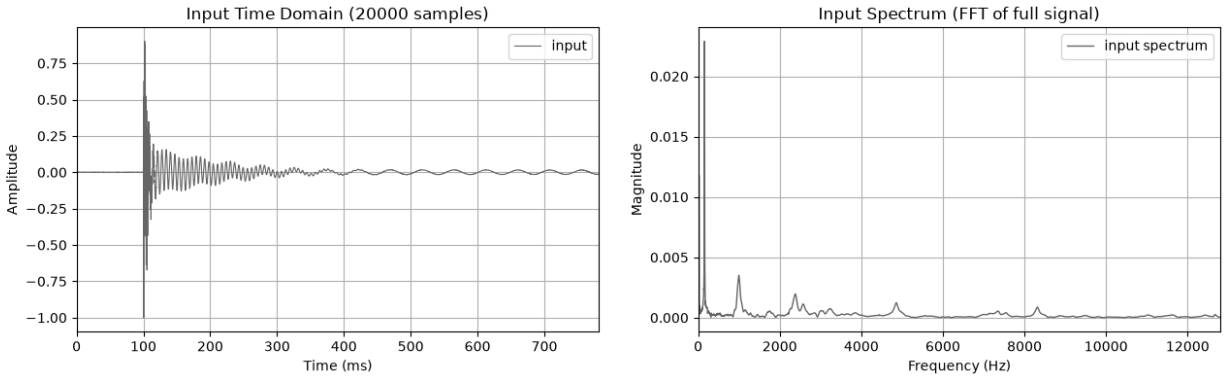

Figure 4: Accelerometer Input Data

The data consists of 20000 samples from a 16 bit output accelerometer. As can be seen in the magnitude spectrum of the input, the resonant frequency of the beam (large peak) is at 157Hz. A lowpass filter should be designed to isolate the resonant frequency of the beam attenuating all the higher frequency peaks. The sampling frequency listed for the data was 25.6 kHz. The required specifications are listed in Table 2.

Table 2: Filter Design Specifications

Parameter	**Value**
Filter Type	Butterworth
Sampling Frequency (f_s)	25.6 kHz
3 dB Cutoff Frequency (f_{d1})	500 Hz
Attenuation (f_{d2})	At least 30 dB at 1.5 kHz

2.4 Development Environment & Target Processor

The target processor is the emulated ARM Cortex-A7 running through QEMU with 2 GB of allocated RAM. This processor is an in-order execution machine with an 8-stage pipeline. The processor supports ARM Neon SIMD instructions and Vector Floating Point (VFP) instructions. The processor has a basic branch predictor. It is a global branch predictor with a 256 entry pattern history table. This is a simpler, less per formant architecture than techniques such as hybrid branch prediction in which both local and global branch history is used. The multiplier in the non-floating-point datapath on the ARM Cortex-A7 is an integer multiplier [4].

The software implementation will be written in C and compiled using GCC version 14.2.0 inside the virtual machine (`arm-linux-gnueabihf`). Compilation flags to be explored when writing code for the project include:

- `-mtune=` Cortex-A7 — optimizes instruction reordering for the ARM Cortex-A7 pipeline.
- `-mfloat-abi=soft` — used to examine the speedup of using fixed-point arithmetic compared to software-based floating point.
- `-O3` — enable compiler optimizations.

3 Implementation

The project implements several MISRA C style guide recommendations. These are summarized in table **??**.

Table 3: MISRA C:2012 Guidelines Applied to the Implementation

Rule	**Status**	**Guideline**
10.1	Followed	Operands have appropriate essential types.
10.3	Followed	Assignments between incompatible essential types are avoided or explicitly cast.
11.3	Followed	Casts between pointers to different object types are avoided.
15.6	Followed	The body of iteration statements is enclosed in braces.
18.1	Followed	Array accesses and pointer arithmetic remain within valid bounds.
8.13	Not followed	Pointers are not consistently declared as pointers to `const` when applicable.
9.1	Not followed	Automatic variables are not formally guaranteed to be initialized before use.
10.8	Not followed	Composite expressions are not systematically checked for MISRA-compliant casting.
14.2	Not followed	The full set of restrictions on loop counters is not formally enforced.
21.3	Not followed	Dynamic memory allocation is not formally prohibited throughout the codebase.

The implementation stage focused on translating the filter designs into C code for the emulated ARM Cortex-A7 target. A baseline floating-point FIR implementation was developed first and used as a reference point for later performance comparisons. Fixed-point FIR and IIR implementations were then developed so that filter behaviour, numerical issues, performance bottlenecks, and optimization opportunities could be evaluated.

3.1 Filter Design

The filter design stage established the FIR and IIR lowpass filters used as the basis for implementation and optimization. Both designs were developed for the accelerometer dataset, with the goal of preserving the beam's resonant frequency while attenuating higher-frequency components. Designing the FIR and IIR filters separately allowed their computational cost, numerical behaviour, and fixed-point implementation requirements to be compared.

Fixed-point arithmetic was used to represent input samples, output samples, and filter coefficients as signed integer values. Scale factors were stored as power-of-two exponents, where an exponent k corresponds to a scale factor of 2^k. The exponent was chosen by selecting the maximum power-of-two factor that allowed the largest value in the range being scaled to fit within a signed 16-bit two's complement representation.

The input and output samples used a scale factor of 2^{15}. The FIR coefficients used a smaller scale factor of 2^{11} because the FIR implementation accumulated products across 51 taps. The IIR implementation required separate scale factors for different coefficient groups: the direct-form IIR feedback path used 2^{28}, the cascaded biquad IIR feedback path used 2^{22}, and the IIR feedforward path used 2^{14}. These separate scale factors were needed because the coefficient ranges differed significantly between the FIR, direct-form IIR, and biquad IIR implementations.

3.1.1 FIR Filter Design

FIR filters require a larger number of filter coefficients to achieve the same performance as IIR filters, as discussed in [5]. However, their lack of feedback makes them useful in applications where guaranteed stability and predictable output behaviour are important. This also made the FIR implementation useful as a stable baseline for comparison. An FIR filter was designed using Python's `scipy.signal` library.

As discussed in [6], a common method of designing FIR filters is the window method. The goal is to truncate the number of coefficients in the filter's impulse response to a finite number, which is required if a microprocessor is to apply the filter. To perform this truncation, the ideal impulse response is multiplied by a window function in the time domain. Multiplication in the time domain is equivalent to convolving the frequency response of the filter with the frequency response of the window function in the frequency domain.

Although a logical choice of window function is the rectangular window (keep coefficients within the window and discard all others), a problem with this approach is that the rectangular function becomes the sinc function in the frequency domain. The convolution will not produce a rectangular function as desired for a lowpass filter; there will be ripples in both the passband and the stopband. To reduce the ripple, a different window function is used. A Blackman window is a popular choice. This window function was used to generate a 100-tap filter using Python. The frequency response showed that the specifications were drastically exceeded. The number of taps was reduced until a suitable balance between computational requirements and filter performance was achieved. The resulting filter has 51 taps [7]. The Bode plot is shown in Figure 5.

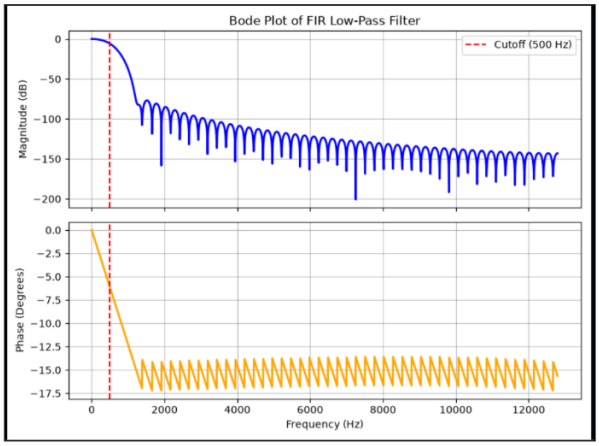

Figure 5: FIR Filter Bode Plot

The resulting FIR coefficients were then used in a direct-form floating point implementation. Listing 1 shows the baseline FIR implementation used before optimization.

In this implementation, the outer loop iterates over the output samples and the inner loop iterates over the FIR coefficients. The condition `n >= k` prevents the filter from accessing input samples before the beginning of the input array. The floating-point FIR implementation did not present correctness issues, but its use of floating-point arithmetic and repeated iteration over the 51 filter taps made it useful for identifying the computational cost that later fixed-point and optimized implementations attempted to reduce.

3.1.2 IIR Filter Design

The IIR filter was first designed using the analog domain design methodology discussed in [3].

By testing the filter, it was observed that the filter was unstable. The poles for the filter resulting from the design procedure in Appendix B are:

$$(z - 1.1317)(1.1624\,z - 0.7883)(z - 0.9532 - 0.1820\,j)(z - 0.9532 + 0.1820\,j) \tag{7}$$

Evidently the filter has a pole outside the unit circle and is therefore unstable. The filter was redesigned using Python's `scipy.signal` library. This resulted in the following difference equation:

$$\begin{aligned} y[n] = {} & 1.2134 \times 10^{-5}\,x[n] + 4.8537 \times 10^{-5}\,x[n-1] + 7.2806 \times 10^{-5}\,x[n-2] \\ & + 4.8537 \times 10^{-5}\,x[n-3] + 1.2134 \times 10^{-5}\,x[n-4] \\ & + 3.6794\,y[n-1] - 5.0886\,y[n-2] \\ & + 3.1345\,y[n-3] - 0.7255\,y[n-4] \end{aligned} \tag{8}$$

The redesigned IIR coefficients were then used in a baseline fixed-point implementation. Listing 2 shows the direct-form IIR implementation used before applying later optimizations. This version was used to translate the redesigned difference equation into fixed-point arithmetic and to provide a reference implementation for later optimized versions.

The implementation separates the feedforward and feedback paths into numerator and denominator accumulators. Since the two paths may use different fixed-point scale factors, the accumulated values are aligned to a shared scale factor before the feedback result is subtracted from the feedforward result. The final value is then rounded, shifted back down, saturated, and stored as the fixed-point output sample. The signal flow graph for the direct form implementation of a second order IIR filter is shown in Figure 6 from [1].

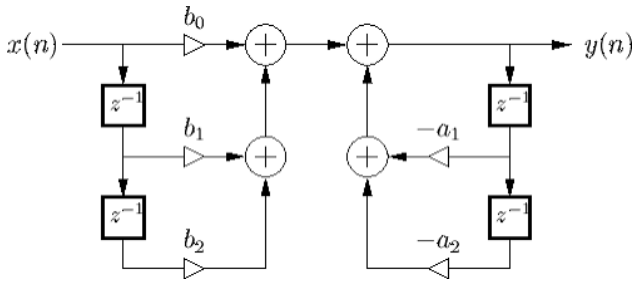

Figure 6: IIR Filter Signal Flow Graph

3.2 Implementation Issues

Both the implementations of the FIR and IIR filters suffered from inadequacies.

The naive fixed point implementation of the FIR filter required significant computational resources. With the test bench discussed in section 5 (20,000 inputs qemu virtual machine running at 1.5GHz), the resulting cycle count totaled 14148. Examining the assembly code of listing 4, the instruction counts in Table 4 can be derived for a single iteration of the outer filter loop (the implementation has a nested structure where both the input array and the filter coefficients are iterated over):

Table 4: Instruction Counts

Instruction	ASR	LSL	ADD	MUL	CMP	B	LDR	STR	MOV
Count (n=2455)	102	211	408	51	204	408	714	255	102

The counts were derived by multiplying the instruction counts of the inner loop by the number of loop iterations. Due to the number of coefficients in the filter's impulse response, the instruction counts for arithmetic, control and memory access are large. The arithmetic instructions are required for the computation of the multiplication of the inputs by the 51 filter taps followed by accumulation into the filter output. Additionally, the number of coefficients is sufficiently large as to cause register spilling of the coefficients into memory (the 32 bit ARM A7 processor is limited to 16 general purpose registers, far fewer than the number of taps). As an example, lines 93 and 94 as well as 119 through 121 demonstrate the accumulator repeatedly being loaded and stored back to memory. Since the FIR filter inner loop is parallel vectorization could be used to not only reduce the instruction count (by using multiple NEON lanes to compute multiple filter outputs in parallel) but also to alleviate register spilling by expanding the number of registers available for use.

The naive IIR filter implementation suffered from limit cycles. As discussed in the implementation section, the input range was limited such that overflow did not occur. The source of limit cycles is thus from numerical instability and quantization. Figure 7 shows the iir_filter output from the direct form implementation of the 4th order filter in the time and frequency domains.

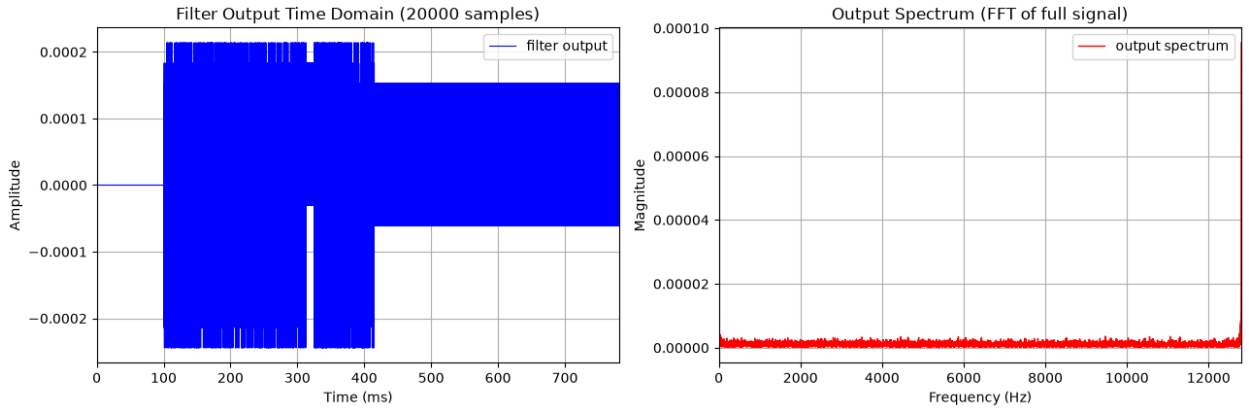

Figure 8: IIR Filter With 32 Bit Accumulator

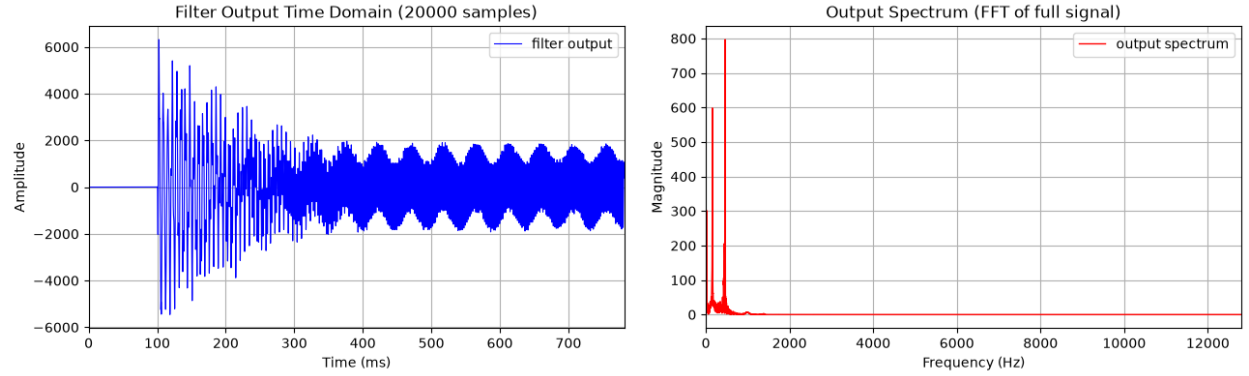

Figure 7: 4th Order IIR Filter Output

As discussed in the background section, the frequency of the limit cycles for a low pass IIR filter occur at the cutoff frequency of the filter. A peak can clearly be observed at the cutoff frequency of 500Hz which dominates the peak corresponding to the beam's resonant frequency (at 157Hz). A further issue was in the dynamic range requirements of the computation. The feedback coefficients (order $\sim 10^{-5}$) yield Q22 scaling while the input and feedforward coefficients (order ~ 1) yield Q15 scaling when the optimal power of two scale factor is determined.

$$Q_{coef} = 15 - \log_2(10^{-5}) \approx 22, \qquad Q_{in} = 15 - \log_2(1) = 15. \tag{9}$$

A single multiply-accumulate product then requires

$$Q_{coef} + Q_{in} = 28 + 15 = 43 \text{ fractional bits}, \tag{10}$$

exceeding the 32-bit accumulator by 11 bits. The coefficient dynamic range is

$$DR_{coef} = 20 \log_{10}\left(\frac{5.0886}{1.2134 \times 10^{-5}}\right) \approx 112.4 \text{ dB} \tag{11}$$

With a 32 bit accumulator, the roundoff error is sufficiently large as to make the filter unstable. This is shown in Figure 8 where the output is shown with a 32 bit accumulator. The accumulator width was expanded to 64 bits, however as seen in listing 6, two registers are required for storing the accumulator. Many MOV and LSL / ASL operations are required to manage storage of the accumulator in 32 bit registers. Loading and storing the accumulator also requires two operations.

Table 5: Biquad Filter Coefficients

Coefficient	Value
$b_{0,1}$	3.48344×10^{-3}
$b_{1,1}$	6.96687×10^{-3}
$b_{2,1}$	3.48344×10^{-3}
$b_{0,2}$	3.48344×10^{-3}
$b_{1,2}$	6.96687×10^{-3}
$b_{2,2}$	3.48344×10^{-3}
$a_{0,1}$	1.00000
$a_{1,1}$	-1.78328
$a_{2,1}$	0.79680
$a_{0,2}$	1.00000
$a_{1,2}$	-1.89614
$a_{2,2}$	0.91050

Optimizations were thus applied to a cascaded biquad implementation of the filter. Note that the new filter coefficients shown in Table 5 are now of order 10^{-3} and the dynamic range between the coefficients and the input is now only $54.67dB$.

Roundoff error induced limit cycles are seen to be significantly reduced as evidenced by the lack of a significant frequency component around the filter cutoff frequency in the output of the filter (shown in figure 9).

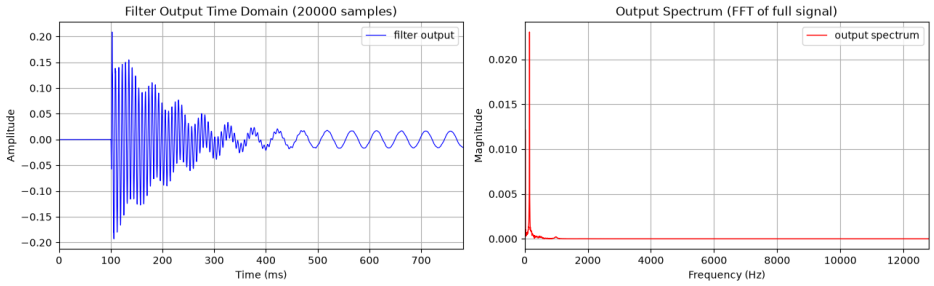

Figure 9: Biquad IIR Filter Output

When this plot is compared to the one in figure 4 the high frequency peaks have successfully been attenuated while the resonant peak at 157Hz is preserved as desired.

In addition to the filter-specific implementation issues discussed above, the overflow handler was also evaluated as a possible source of overhead. Two saturation implementations were compared: a manual saturation function and an inline ARM saturation version. The manual version used conditional checks to clamp the output to the signed 16-bit range, while the ARM version used an inline saturation instruction to remove the explicit `if` statements. The ARM version was expected to reduce branch overhead because the manual comparisons generate branch instructions. However, testing showed that the inline ARM version was slower. Since the saturation function was declared as `static inline` and the code was compiled with optimization enabled, such as `-O2`, function call overhead was already reduced by the compiler. Therefore, the inline ARM instruction did not provide enough benefit to outweigh the extra constraints introduced by the inline assembly.

4 Optimization

4.1 Loop Unrolling - IIR

The principle behind loop unrolling for the IIR filter is that the baseline implementation contains small inner loops that iterate over the feedforward and feedback coefficients for every output sample. Since the optimized implementation uses a second-order IIR section, the number of required input samples, output samples, and coefficients is known ahead of time. The idea is to replace these inner coefficient loops with explicit arithmetic operations so that loop-control instructions, repeated indexing, and unnecessary memory accesses are reduced.

In the unrolled implementation, the feedforward coefficients are loaded into local variables `x0`, `x1`, and `x2`, while the feedback coefficients are loaded into `y1` and `y2`. The numerator and denominator scale factors are also read once before the main loop, and rounding constants are computed before filtering begins. The first two output samples are handled separately because they do not have the full input and output history required by the general recurrence. After this initialization, each output sample is computed directly from the current and previous input samples and the previous output samples.

The function parameters were marked with `__restrict` to indicate that the input buffer, output buffer, and filter structure do not alias each other. This gives the compiler more freedom to keep intermediate values, such as coefficients and accumulator values, in registers rather than repeatedly loading and storing them in memory. This is useful on the ARM Cortex-A7 target because the number of available general-purpose registers is limited and unnecessary memory access can increase loop overhead.

The main changes in the unrolled implementation are summarized below:

- The feedforward coefficient loop is replaced with explicit computations using `x0`, `x1`, and `x2`.
- The feedback coefficient loop is replaced with explicit computations using `y1` and `y2`.
- Rounding constants are computed once before the loop.
- The first two output samples are handled separately to avoid invalid references to previous samples.
- `__restrict` pointers are used to help the compiler keep intermediate values in registers.

This optimization reduces loop-control overhead and makes the coefficient accesses more explicit to the compiler. However, the IIR filter remains limited by its feedback dependency because each output sample depends on previous output samples. As a result, loop unrolling can reduce overhead, but it cannot make the IIR computation fully parallel in the same way as the FIR filter.

4.2 Software Pipelining IIR - Khephren

The assembly code resulting from the software pipelined implementation of the IIR filter is shown in listing 8. The ARM Cortex A7 processor is a dual issue slot machine. This means that two instructions that do not have data or control dependencies can be executed simultaneously. The principle applied in this implementation of software pipelining is that the feedforward computation is independent of the feedback computation in an IIR filter implemented with direct form I. In the loop prologue, the coefficients can be loaded into registers and the feedforward path can be computed for the first input. In the loop, the feedback path for the first input can be computed while the feedforward path for the next input is computed. As shown in table 12 in section 5, software pipelining offered a 27 times speedup over the soft floating point FIR implementation. The code was compiled with $-mtune = cortex - a7$ and $-O2$ because without these flags, the compiler generates load and store instructions for the coefficients and accumulator in between loop iterations which creates a dependency between the feedforward and feedback paths (shown in listings 7 8). It is clear based on the speedup that software pipelining was effective in increasing the utilization of the two issue slots of the processor.

4.3 Neon SIMD - IIR

The principle behind the use of Neon SIMD for the IIR filter is that the assembly code of the naive implementation exhibits register spilling (listing 6). The idea is to use the Neon register file which consists of 32 registers each 64 bits wide. These are the d registers in the assembly code. This register file can also be treated as 16, 128 bit q registers. Arm provides a c library (arm_neon.h) which allows vector operations to be invoked in the c code directly [8]. If the 5 IIR filter coefficients, 3 input samples and two outputs are represented in 16 bits, the multiply result in 32 bits and the accumulators for the feedforward and feedback paths also in 32 bits we can divide the variables into the registers as described below:

Table 6: NEON register allocation for the IIR filter.

Registers	Variables	Data Width
`d20`	b_0, b_1, b_2	16-bit
`d21`	a_1, a_2	16-bit
`d18`	$x[n], x[n-1], x[n-2]$	16-bit
`d16`	$y[n-1], y[n-2]$	16-bit
`q11-q12`	shift amounts	32-bit
`q8-q9`	multiplication results	32-bit
`d6-d7`	accumulators	32-bit

This allocation is shown in lines 52 through 62 of the assembly code in Listing **??**. Four vector operations are required to compute the feedforward and feedback paths:

- vmull.s16
- vaddq_s32
- vshrq_n_s32
- vpadd_s32

These instructions are discussed in the neon reference manual [8]. The vmull operation computes a multiplication for each lane in the input argument. In this case, the input is divided into four int16 lanes (one int16 coefficient is multiplied by one int16 sample in each lane). Vaddq denotes vector addition. The q suffix indicates that the register file should be treated as 128 bit q registers (the 4 multiplication results occupy a q register while an immediate round factor is duplicated into a second q register). Vshrq shifts the result to complete the process of rounding and vpadd sums across the lanes in the neon register. This is required to obtain a single accumulator value to be used in computing the filter output. The results of the test bench indicate a speedup of 33x when compared to the soft floating point implementation of the FIR filter.

4.4 MAC

Multiply-accumulate instructions were tested because FIR and IIR filters repeatedly multiply samples by coefficients and add the products to an accumulator. The first implementation used the ARM `SMLAL` instruction with a 64-bit accumulator. Although this matched the wider accumulator used in the fixed-point IIR implementation, it required splitting the accumulator into lower and upper 32-bit values and recombining them after the instruction. This added extra overhead on the 32-bit ARM Cortex-A7 target.

The implementation was then changed to use the ARM `MLA` instruction with a 32-bit accumulator. This reduced the overhead of managing a 64-bit value, but made the implementation more sensitive to fixed-point scaling and accumulator range. The FIR MAC output was largely unaffected as seen in Figure 10 because FIR outputs depend only on input samples and coefficients. The IIR MAC output behaved similarly to the earlier direct-form fourth-order IIR output (Figure 7) because any scaling or quantization error can be reused through the recursive feedback path and appear as sustained oscillation. This behaviour is shown in Figure 11.

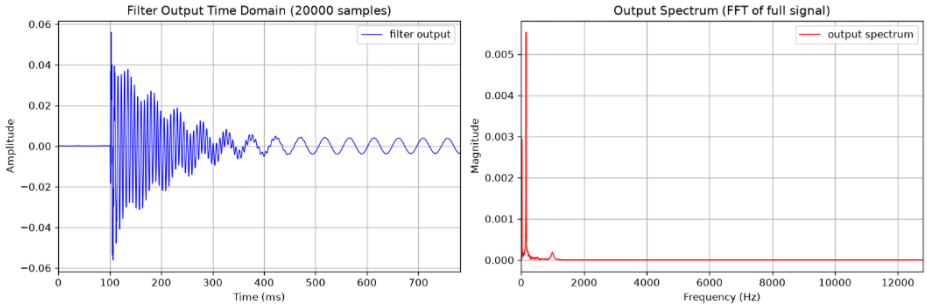

Figure 10: FIR fixed-point MAC output

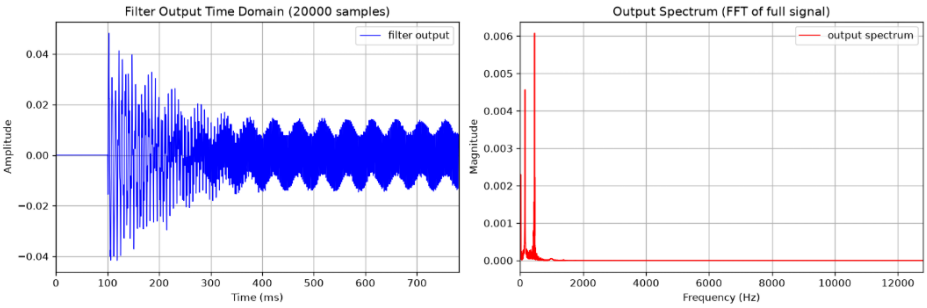

Figure 11: IIR fixed-point MAC output

Overall, the MAC implementation showed that using a processor-supported instruction does not automatically improve performance or numerical behaviour. The normal C multiply-accumulate pattern was already optimized effectively by the compiler, while the inline MAC version introduced additional scaling concerns.

4.5 Neon SIMD - FIR

The final optimization technique tested was the use of NEON SIMD instructions in the implementation of the FIR filter. The goal of this optimization technique was to investigate if the performance of a vectorized FIR implementation could be comparable to the performance of the IIR filter. As the transfer function of an FIR filter does not have any poles, there is no feedback in the filter's difference equation. The implications for optimization is that subsequent filter output computations do not have data dependencies. This allows them to be computed in parallel. The NEON implementation of the FIR filter computes 8 filter outputs in parallel. Listings 3 and 4 show both the c code and the assembly for a single iteration of the filter computation. The steps below illustrate the computations performed:

- Eight inputs are loaded into a q register with the vdupq instruction
- A loop iterates over the 51 filter taps, copying a single tap across a second q register
- The multiplication and accumulation is computed with a single vmlal intrinsic
- The accumulator result after iterating over the 51 filter taps are shifted back to the scale factor of the input (Q15 for the FIR) and the vcombine intrinsic is used to sum the accumulators in the q register into a scalar

Note that the vmlal intrinsic multiplies the 8 inputs by the current tap and adds the result to an accumulator which is computed for each 32 bit wide lane. This optimization technique resulted in a speedup of 7.2x over the floating point FIR implementation.

5 Testing - Khephren

The filter implementations were subjected to a common test bench to judge the filter's stability, susceptibility to overflow and limit cycles, and speed.

5.0.1 Methodology

As discussed in section 2.3, a test input of 20000 samples normalized between 1 and -1 was applied to each filter implementation. Testing was performed on the QEMU virtual machine specified in section 2.4. A python routine was used to plot the filter input and output in the time and frequency domain in order to visually confirm the proper operation of the filter. Speedups were computed by using the *time.h* library to record the CPU time of the filter code sections. A simple performance profiling utility was written to facilitate computation of the CPU time of each filter. An excerpt from *tests.c* exhibiting the usage of the utility is shown in listing 1 below.

```
93 profiler_start();
94 iir_filter_fixed_point_mac(input_data, filter_output, data_samples, filter);
95 profiler_stop();
96 ...
97 printf("%d", profiler_get_elapsed_time());
```

Note that the CPU times computed are not cycle accurate. The results returned by the profiler_get_elapsed_time() function are from the local machine and thus are much shorter than on a real ARM machine running at 1.5GHz. As a result, the times are only used to measure relative speed between implementations. To evaluate the performance of each implementation, the assembly generated by the compiler (obtained using the $-S$ flag) was examined and counts of the common instructions were derived manually. Valgrind was used to obtain global instruction counts for the computation over 20000 inputs. Valgrind is a multipurpose profiling and debugging and profiling software that packages many tools together. The Callgrind and Cachegrind tools were used for this project. Callgrind provides instruction counts related to each line of c source code and Cachegrind estimates the cache performance of your program. In particular, cachegrind simulates a machine with separate level 1 data and instruction caches and a combined last level (ll) cache. This is the same general structure of an ARM processor cache. The baseline implementations of each filter were compiled without the -mtune or -O3 flags. Additionally, the floating point implementation of the FIR filter was compiled with the -mfloat-abi=soft flag. For the optimized implementations, the -mtune=cortex-a7 and -O3 flags were used.

5.0.2 Results

Tables 7 and 8 summarize the results of running the test bench on the basic FIR and IIR filters.

Table 7: Profiling results for `fir_filter_core_fixed`

Metric	Value
Speedup over floating point	5.5
Instruction count (20,000 inputs)	28,878,325
Instruction count (1 input)	2,455
ASR (arithmetic shift right)	102
LSL (logical shift left)	211
ADD	408
MUL	51
CMP	204
B	408
LDR	714
STR	255
MOV	102
DMR	627

Table 8: Profiling results for `iir_filter_core_fixed`

Metric	Value
Speedup over floating point	9.6
Instruction count (20,000 inputs)	8,399,537
Instruction count (1 input)	170
ASR (arithmetic shift right)	3
LSL (logical shift left)	7
ADD	40
MUL	15
CMP	4
B	9
LDR	37
STR	19
MOV	8
DMR	629

Note that DMR corresponds to the data miss rate output from Cachegrind. The tables illustrate the fact that IIR filter implementations are inherently faster than FIR filter implementations. The assembly code used to profile these implementations is shown in listings **??** and **??**.

Table 9 shows the profiling results for the biquad implementation of the IIR filter.

Table 9: Profiling results for `iir_filter_biquad`

Metric	Value
Speedup over floating point	8.2
Instruction count (20,000 inputs)	8,247,748
Instruction count (1 input)	142
ASR (arithmetic shift right)	8
LSL (logical shift left)	6
ADD	38
MUL	15
CMP	5
B	8
LDR	21
STR	13
MOV	5
D1MR (data miss rate)	1,251

The instruction count is similar to the direct implementation of the IIR filter however the data miss rate has increased (by approximately a factor of 2) and the speedup has slightly decreased. The IIR filter invokes the function `iir_filter_biquad` twice when performing the computation. Due to the large size of the input (20000 16 bit samples) it is evident that the entire input will not fit in the cache. Thus, iterating over the input twice is expected to double the miss rate (misses should occur on the second iteration in addition to the first). The tradeoff in performance was deemed worthwhile due to the biquad implementation's increased resilience towards limit cycles which is evident in the plot of figure 9.

Tables 10 through 14 summarize the profiling results for each optimization technique.

Table 10: Profiling results for `iir_filter_biquad_2` (`__restrict` / loop unrolling)

Metric	Value
Speedup over floating point	10.2
Instruction count (20,000 inputs)	7,239,688
Instruction count (1 input)	162
ASR (arithmetic shift right)	8
LSL (logical shift left)	4
ADD	6
MUL	2
CMP	8
B	12
LDR	74
STR	48
MOV	6
D1MR (data miss rate)	1,254

Table 11: Profiling results for `iir_filter_biquad_3` (with MAC instructions)

Metric	Value
Speedup over floating point	3.8
Instruction count (20,000 inputs)	9,279,898
Instruction count (1 input)	324
ASR (arithmetic shift right)	16
LSL (logical shift left)	22
ADD	20
MUL	2
CMP	24
B	36
LDR	148
STR	56
MOV	70
D1MR (data miss rate)	1,254

Table 12: Profiling results for `iir_filter_biquad_4` (with software pipelining)

Metric	Value
Speedup over floating point	27.2
Instruction count (20,000 inputs)	3,999,416
Instruction count (1 input)	129
ASR (arithmetic shift right)	7
LSL (logical shift left)	6
ADD	24
MUL	15
CMP	1
B	2
LDR	34
STR	7
MOV	5
D1MR (data miss rate)	1,249

Table 13: Profiling results for `iir_filter_biquad_5` (with NEON)

Metric	Value
Speedup over floating point	33.5
Instruction count (20,000 inputs)	279,980
Instruction count (1 input)	35
`vmull.s16`	2
`vadd.i32`	4
`vshr.s32`	2
`vpadd.i32`	4
`vld1.s16`	2
ADD	3
CMP	1
B	3
LDR	8
STR	6

Table 14: Profiling results for `fir_filter_core_fixed_vector`

Metric	Value
Speedup over floating point	7.2
Instruction count (20,000 inputs)	5,617,500
Instruction count (1 input)	40
`vdup`	1
`vstr`	1
`vmlal.s16`	2
`vqrshrn.s32`	2
`vmov`	4
ADD	12
CMP	10
B	2
LDR	1
STR	0

The fastest implementations corresponded to the software pipelined and vectorized IIR filter implementations. These implementations offered speedups of 27.2 and 33.5 respectively over the floating point FIR implementation.

6 Conclusion & Recommendations

This project analyzed the implementation and optimization of FIR and IIR digital filters on an ARM Cortex-A7, focusing on the trade-offs between computational performance, numerical stability, and filter structure. A floating-point FIR implementation was used as the common performance reference, allowing the speedups of the fixed-point FIR and IIR implementations to be compared consistently. The baseline fixed-point FIR achieved a 5.5x speedup, while the direct fixed-point IIR achieved 9.6x.

Several optimization strategies were evaluated, including loop unrolling, inline MAC instructions, software pipelining, and NEON SIMD. Loop unrolling improved the biquad IIR implementation to 10.2x, while inline MAC instructions were less effective, achieving only 3.8x. The largest improvements came from exploiting parallelism: software pipelining achieved a 27.2x speedup, while NEON SIMD achieved the highest performance at 33.5x. These results demonstrate that optimization strategies targeting register utilization and instruction-level parallelism were substantially more effective than optimizing individual arithmetic operations.

Overall, platforms with a vector register file should prioritize SIMD implementations to exploit additional registers and prevent register spilling or thrashing. On platforms without vector registers, software pipelining should instead be prioritized to maximize instruction-level parallelism and processor utilization.

References

[1] I. Smith, Julius O., *Introduction to Digital Filters with Audio Applications*. online: W3K Publishing, 2007. Accessed: Aug. 7, 2026.

[2] BMS College of Engineering, "Butterworth filters." `https://bmsce.ac.in/Content/TE/Butterworth_filters.pdf`. Accessed: 2026.

[3] M. Sima, "SENG440 embedded systems – lesson 106: Digital filters." Lecture notes, University of Victoria, 2024.

[4] ARM Limited, *Cortex-A7 MPCore Technical Reference Manual*, 2012.

[5] M. Azin, H. J. Chiel, and P. Mohseni, "Comparisons of FIR and IIR implementations of a subtraction-based stimulus artifact rejection algorithm," in *29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pp. 1437–1440, 2007.

[6] R. G. Lyons, *Understanding Digital Signal Processing*. Prentice Hall, 3rd ed., 2011.

[7] K. Gould, "SENG440 Digital Filters — FIR filter coefficients." `https://github.com/KhephG1/SENG440DigitalFilters/blob/main/tools/FIR_filter_coeffs.txt`, 2026.

[8] Arm Limited, "NEON registers." Introducing NEON, Document DHT0002A. Accessed: 2026-08-08.

7 Appendix A Filter Design Process

7.1 IIR Filter Design Using Analog Methods

The design methodology used is based on the Butterworth transfer function in the analog domain, the pre-warping transform must be used to determine the analog frequencies that correspond to the digital frequencies. This is because the bilinear transform used to obtain the system function $H(z)$ from the transfer function $H(s)$ scales the frequency axis nonlinearly. The bilinear transform operation is:

$$s = \frac{2}{T} \cdot \frac{z-1}{z+1} \tag{12}$$

The analog angular frequencies resulting from the pre-warping are found using the formula:

$$\Omega_{a1} = \frac{2}{T} \tan\left(\frac{\omega_{d1} T}{2}\right) = \frac{2}{39.0625 \times 10^{-6}} \tan\left(\frac{1000\pi \times 39.0625 \times 10^{-6}}{2}\right) = 3145.54 \text{ rad/s} \tag{13}$$

$$\Omega_{a2} = \frac{2}{T} \tan\left(\frac{\omega_{d2} T}{2}\right) = \frac{2}{39.0625 \times 10^{-6}} \tan\left(\frac{3000\pi \times 39.0625 \times 10^{-6}}{2}\right) = 9466 \text{ rad/s} \tag{14}$$

These correspond to the analog frequencies:

$$f_{a1} = \frac{\Omega_{a1}}{2\pi} = \frac{3145.54}{2\pi} = 500.63 \text{ Hz} \tag{15}$$

$$f_{a2} = \frac{\Omega_{a2}}{2\pi} = \frac{9466}{2\pi} = 1506.64 \text{ Hz} \tag{16}$$

The order of the filter corresponds to the number of poles required. The minimum filter order is obtained using the formula [2]:

$$n \geq \frac{\log\left(\frac{10^{A_2/10} - 1}{10^{A_1/10} - 1}\right)}{2 \log\left(\frac{f_{a2}}{f_{a1}}\right)} \tag{17}$$

where A_1 is the filter gain at the cutoff frequency, A_2 is the filter gain at the rejection frequency, f_{a1} is the cutoff frequency, and f_{a2} is the rejection frequency. Inserting the values results in:

$$n \geq \frac{\log\left(\frac{999}{0.995}\right)}{2\log\left(\frac{1506.64}{500.63}\right)} \approx 3.14 \tag{18}$$

Thus, the required order of the Butterworth filter is 4.

The transfer function for a 4th-order Butterworth filter is:

$$H(s) = \frac{\Omega_c^4}{s^4 + 2.6131\,\Omega_c\,s^3 + 3.4142\,\Omega_c^2\,s^2 + 2.6131\,\Omega_c^3\,s + \Omega_c^4} \tag{19}$$

In this case, $\Omega_c = \frac{2}{T} \cdot 0.061436$. Applying the bilinear transform yields the transfer function in the z-domain:

$$\begin{aligned} H(z) = \frac{\left(\frac{2}{T}\right)^4 (0.061436)^4}{\left(\frac{2}{T}\right)^4 \left(\frac{z-1}{z+1}\right)^4 + \left(\frac{2}{T}\right)^3 (0.160539) \left(\frac{z-1}{z+1}\right)^3} \\ + \left(\frac{2}{T}\right)^2 (0.00128866) \left(\frac{z-1}{z+1}\right)^2 \\ + \left(\frac{2}{T}\right) (0.00060594) \left(\frac{z-1}{z+1}\right) + \left(\frac{2}{T}\right) (0.00001424) \end{aligned} \tag{20}$$

$$H(z) = \frac{\begin{array}{c} 0.00001424 + 0.00005696\,z^{-1} + 0.00008544\,z^{-2} \\ + 0.00005696\,z^{-3} + 0.00001424\,z^{-4} \end{array}}{\begin{array}{c} 1.1624 - 4.3198\,z^{-1} + 5.9975\,z^{-2} \\ - 3.68\,z^{-3} + 0.84016\,z^{-4} \end{array}} \tag{21}$$

This leads to the difference equation:

$$\begin{aligned} y[n] = {} & 0.00001225\,x[n] + 0.00004900\,x[n-1] + 0.00007350\,x[n-2] \\ & + 0.00004900\,x[n-3] + 0.00001225\,x[n-4] \\ & - 3.716\,y[n-1] + 5.15958\,y[n-2] \\ & - 3.16586\,y[n-3] + 0.72278\,y[n-4] \end{aligned} \tag{22}$$

8 Appendix B Assembly Code Excerpts

Listing 1: FIR floating point implementation

```
void fir_filter_naive(const float *input, float *output, uint32_t input_length,
                      float *filter_x, int *coeffsx)
{
    for (uint32_t n = 0; n < input_length; n++)
    {
        float acc = 0;

        for (uint32_t k = 0; k < *coeffsx; k++)
        {
            if (n >= k)
            {
                acc += input[n - k] * filter_x[k];
            }
        }
        output[n] = acc;
    }
}
```

Listing 2: Baseline fixed-point IIR implementation

```
void iir_filter_fixed_point(const input_data_t *input, int16_t *output,
                            uint32_t input_length,
 filter_t *filter)
{
    const int num_sf = filter->num_scale_factor_exp;
    const int den_sf = filter->den_scale_factor_exp;
    const int acc_sf = (num_sf > den_sf) ? num_sf : den_sf;

    for (int i = 0; i < (int)input_length; i++)
    {
        // Feedforward accumulation
        int64_t num_acc = 0;
        for (int j = 0; j < filter->x_coeffs; j++)
        {
            if (i >= j)
            {
                num_acc += (int64_t)((int32_t)filter->x[j] *
                                     input->input_data_buffer[i - j]);
            }
        }

        // Feedback accumulation; y[0] is normalized to 1.0
        int64_t den_acc = 0;
        for (int j = 1; j < filter->y_coeffs; j++)
        {
            if (i >= j)
            {
                den_acc += (int64_t)((int32_t)filter->y[j] *
                                     output[i - j]);
            }
        }

        // Align scale factors before combining terms
        int64_t total =
            (num_acc << (acc_sf - num_sf)) -
            (den_acc << (acc_sf - den_sf));

        // Round, shift, and saturate the output
        total = (total + (1 << (acc_sf - 1))) >> acc_sf;
        output[i] = saturate(total);
    }
}
```

Listing 3: Inner loop of the baseline FIR filter in C

```
for (uint32_t k = 0;
     k < coeffs_length; k++)
{
    if (n >= k)
    {
        acc += (int32_t)
            input->input_data_buffer[n - k]
            * coeffs[k];
    }
}
```

Listing 4: Assembly for the baseline FIR filter inner loop

```
movs    r3, #0
str     r3, [r7, #20]    @ k
b       .L40

.L42:
ldr     r2, [r7, #28]    @ n
ldr     r3, [r7, #20]    @ k
cmp     r2, r3
bcc     .L41

ldr     r2, [r7, #28]    @ n
ldr     r3, [r7, #20]    @ k
subs    r2, r2, r3       @ n-k

ldr     r3, [r7, #12]    @ input
ldrsh   r3, [r3, r2, lsl #1]
mov     r1, r3

ldr     r3, [r7, #20]    @ k
lsls    r3, r3, #1
ldr     r2, [r7]         @ coeffs
add     r3, r3, r2
ldrsh   r3, [r3]

mul     r3, r1, r3

ldr     r2, [r7, #24]    @ acc
add     r3, r3, r2
str     r3, [r7, #24]    @ acc

.L41:
ldr     r3, [r7, #20]    @ k
adds    r3, r3, #1
str     r3, [r7, #20]    @ k

.L40:
ldr     r2, [r7, #20]    @ k
ldr     r3, [r7, #44]    @ coeffs_length
cmp     r2, r3
bcc     .L42
```

Listing 5: IIR filter denominator accumulation in C

```
239         for (int j = 1; j < filter->y_coeffs; j++)
240         {
241             if (i >= j)
242             {
243                 den_acc += (int64_t)((int32_t)
    filter->y[j] * output[i - j]);
244             }
245         }
```

Listing 6: ARM Cortex-A7 assembly for the accumulation

```
1  ldr  r2, [r7, #56]   @ tmp176, filter
2  ldr  r3, [r7, #92]   @ tmp178, j
3  adds r3, r3, #100    @ tmp177, tmp178,
4  ldrsh r3, [r2, r3, lsl #1] @ _12, filter_46(D)->y[j_43]
5  mov  r1, r3          @ _13, _12
6  ldr  r2, [r7, #124]  @ tmp179, i
7  ldr  r3, [r7, #92]   @ tmp180, j
8  subs r3, r2, r3      @ _14, tmp179, tmp180
9  lsls r2, r3, #1      @ _16, _15,
10 ldr  r3, [r7, #64]   @ tmp181, output
11 add  r3, r3, r2      @ _17, _16
12 ldrsh r3, [r3]       @ _18, *_17
13 mul  r3, r1, r3      @ _20, _13, _19
14 asrs r2, r3, #31     @ tmp182, _20,
15 str  r3, [r7, #32]   @ _20, %sfp
16 str  r2, [r7, #36]   @ tmp182, %sfp
17 ldrd r2, [r7, #96]   @ tmp184,,
18 ldr  r1, [r7, #32]   @ tmp221, %sfp
19 adds r1, r2, r1      @ tmp220, tmp184, tmp221
20 str  r1, [r7, #8]    @ tmp220, %sfp
21 ldr  r1, [r7, #36]   @ tmp223, %sfp
22 adcs r3, r3, r1      @ tmp222,, tmp223
23 str  r3, [r7, #12]   @ tmp222, %sfp
24 ldrd r3, [r7, #8]    @ den_acc_62,,
25 strd r3, [r7, #96]   @ den_acc_62,,
```

Listing 7: Loop body -O0

```
b    .L40
.L41:
ldrsh r2, [r7, #26]  @ _56, y1
ldr  r3, [r7, #80]   @ i.6_57, i
add  r3, r3, #-2147483648  @ _58, i.6_57,
subs r3, r3, #1      @ _58, _58,
lsls r3, r3, #1      @ _59, _58,
ldr  r1, [r7, #16]   @ tmp316, output
add  r3, r3, r1      @ _60, tmp316
ldrsh r3, [r3]       @ _61, *_60
mul  r2, r3, r2      @ _63, _62, _56
ldr  r3, [r7, #64]   @ tmp317, den_add
add  r2, r2, r3      @ _64, tmp317
ldr  r3, [r7, #72]   @ tmp319, den_sf
asr  r3, r2, r3      @ temp4_172, _64, tmp319
str  r3, [r7, #32]   @ temp4_172, temp4
ldrsh r2, [r7, #24]  @ _65, y2
ldr  r3, [r7, #80]   @ i.7_66, i
add  r3, r3, #-2147483648  @ _67, i.7_66,
subs r3, r3, #2      @ _67, _67,
lsls r3, r3, #1      @ _68, _67,
ldr  r1, [r7, #16]   @ tmp320, output
add  r3, r3, r1      @ _69, tmp320
ldrsh r3, [r3]       @ _70, *_69
mul  r2, r3, r2      @ _72, _71, _65
ldr  r3, [r7, #64]   @ tmp321, den_add
add  r2, r2, r3      @ _73, tmp321
ldr  r3, [r7, #72]   @ tmp323, den_sf
asr  r3, r2, r3      @ temp5_173, _73, tmp323
str  r3, [r7, #28]   @ temp5_173, temp5
ldr  r2, [r7, #32]   @ tmp325, temp4
ldr  r3, [r7, #28]   @ tmp326, temp5
add  r3, r3, r2      @ den_acc_174, tmp325
str  r3, [r7, #56]   @ den_acc_174, den_acc
ldr  r2, [r7, #84]   @ tmp327, num_acc
ldr  r3, [r7, #56]   @ tmp328, den_acc
subs r3, r2, r3      @ _74, tmp327, tmp328
asrs r2, r3, #31     @ tmp329, _74,
mov  r4, r3          @ _75, _74
mov  r5, r2          @ _75, tmp329
ldr  r3, [r7, #80]   @ i.8_76, i
lsls r3, r3, #1      @ _77, i.8_76,
ldr  r2, [r7, #16]   @ tmp330, output
adds r6, r2, r3      @ _78, tmp330, _77
mov  r0, r4          @, _75
mov  r1, r5          @, _75
bl   saturate(PLT)
mov  r3, r0          @ tmp331,
strh r3, [r6]        @ movhi @ tmp332, *_78
ldrsh r3, [r7, #54]  @ _80, x0
ldr  r2, [r7, #80]   @ i.9_81, i
adds r2, r2, #1      @ _82, i.9_81,
lsls r2, r2, #1      @ _83, _82,
ldr  r1, [r7, #60]   @ tmp333, in
add  r2, r2, r1      @ _84, tmp333
ldrsh r2, [r2]       @ _85, *_84
mul  r2, r3, r2      @ _87, _80, _86
ldr  r3, [r7, #68]   @ tmp334, num_add
add  r2, r2, r3      @ _88, tmp334
ldr  r3, [r7, #76]   @ tmp336, num_sf
asr  r3, r2, r3      @ temp1_177, _88, tmp336
str  r3, [r7, #44]   @ temp1_177, temp1
ldrsh r3, [r7, #52]  @ _89, x1
ldr  r2, [r7, #80]   @ i.10_90, i
lsls r2, r2, #1      @ _91, i.10_90,
ldr  r1, [r7, #60]   @ tmp337, in
add  r2, r2, r1      @ _92, tmp337
ldrsh r2, [r2]       @ _93, *_92
mul  r2, r3, r2      @ _95, _89, _94
ldr  r3, [r7, #68]   @ tmp338, num_add
add  r2, r2, r3      @ _96, tmp338
ldr  r3, [r7, #76]   @ tmp340, num_sf
asr  r3, r2, r3      @ temp2_178, _96, tmp340
str  r3, [r7, #40]   @ temp2_178, temp2
ldrsh r2, [r7, #50]  @ _97, x2
ldr  r3, [r7, #80]   @ i.11_98, i
add  r3, r3, #-2147483648  @ _99, i.11_98,
subs r3, r3, #1      @ _99, _99,
lsls r3, r3, #1      @ _100, _99,
ldr  r1, [r7, #60]   @ tmp341, in
add  r3, r3, r1      @ _101, tmp341
ldrsh r3, [r3]       @ _102, *_101
mul  r2, r3, r2      @ _104, _103, _97
ldr  r3, [r7, #68]   @ tmp342, num_add
add  r2, r2, r3      @ _105, tmp342
ldr  r3, [r7, #76]   @ tmp344, num_sf
asr  r3, r2, r3      @ temp3_179, _105, tmp344
str  r3, [r7, #36]   @ temp3_179, temp3
ldr  r2, [r7, #44]   @ tmp345, temp1
ldr  r3, [r7, #40]   @ tmp346, temp2
add  r3, r3, r2      @ _106, tmp345
ldr  r2, [r7, #36]   @ tmp348, temp3
add  r3, r3, r2      @ num_acc_180, tmp348
str  r3, [r7, #84]   @ num_acc_180, num_acc
```

Listing 8: Pipelined IIR Loop body -O2

```
cmp  r3, #2        @ _201,
bls  .L35          @,
ldr  r3, .L62+4    @ tmp314,
mov  r8, r6        @ ivtmp.69, output
ldr  r1, [sp, #36] @ tmp222, %sfp
adds r5, r4, #6    @ ivtmp.74, input,
ldrsh r9, [r6]     @ D__lsm0.62, *output_125(D)
ldr  r1, [r1, r3]  @ tmp302,
ldr  r3, [sp, #28] @ input_length, %sfp
strd r6, r1, [sp, #40]  @ output, tmp302,,
add  r3, r4, r3, lsl #1 @ _52, input, input_length,
str  r3, [sp, #20] @ _52, %sfp
ldr  r3, [r1]      @ overflow_count_lsm.55, overflow_count
ldrsh r4, [r8, #2]! @ D__lsm1.63, MEM[(int16_t *)output_125(D)+2B]
str  r3, [sp, #24] @ overflow_count_lsm.55, %sfp
movs r3, #0        @ overflow_count_lsm_flag.56,
str  r3, [sp, #32] @ overflow_count_lsm_flag.56, %sfp
b    .L39
.L36:
cmn  r3, #32768    @ _314,
sbcs r2, r2, #-1   @ tmp324, tmp263,
blt  .L54          @,
.L37:
ldr  r3, [sp, #16] @ x2, %sfp
mov  r9, r4        @ D__lsm0.62, D__lsm1.63
ldrsh r6, [r5], #2 @ _300, MEM[(const int16_t*)_145]
mov  r4, r1        @ D__lsm1.63, _307
mla  r10, r3, r10, r0  @ _285, x2, D__lsm0.59, num_add
strh r1, [r8, #2]!  @ movhi @ _307, MEM[(int16_t*)_56]
ldr  r3, [sp, #4]  @ pretmp_392, %sfp
ldr  r1, [sp]      @ pretmp_386, %sfp
mla  r3, lr, r3, r0  @ _291, D__lsm1.60, pretmp_392, num_add
mla  r1, r6, r1, r0  @ _297, _300, pretmp_386, num_add
asr  r3, r3, ip    @ temp2_290, _291, num_sf
asr  r1, r1, ip    @ temp1_296, _297, num_sf
asr  r2, r10, ip   @ temp3_284, _285, num_sf
mov  r10, lr       @ D__lsm0.59, D__lsm1.60
add  r3, r3, r1    @ _283, temp1_296
mov  lr, r6        @ D__lsm1.60, _300
add  r2, r2, r3    @ num_acc, _283
ldr  r3, [sp, #20] @ _52, %sfp
cmp  r3, r5        @ _52, ivtmp.74
beq  .L55          @,
.L39:
ldr  r3, [sp, #12] @ y2, %sfp
mla  r9, r9, r3, fp  @ _317, D__lsm0.62, y2, den_add
ldr  r3, [sp, #8]  @ pretmp_394, %sfp
asr  r9, r9, r7    @ temp5_316, _317, den_sf
mla  r3, r4, r3, fp  @ _325, D__lsm1.63, pretmp_394, den_add
asrs r3, r3, r7    @ temp4_324, _325, den_sf
add  r9, r9, r3    @ den_acc_315, temp4_324
sub  r3, r2, r9    @ _314, num_acc, den_acc_315
cmp  r3, #32768    @ _314,
sxth r1, r3        @ _307, _314
asr  r2, r3, #31   @ tmp263, _314,
sbcs r6, r2, #0    @ tmp323, tmp263
blt  .L36          @,
ldr  r3, [sp, #24] @ overflow_count_lsm.55, %sfp
movw r1, #32767    @ _307,
adds r3, r3, #1    @ overflow_count_lsm.55,,
str  r3, [sp, #24] @ overflow_count_lsm.55, %sfp
movs r3, #1        @ overflow_count_lsm_flag.56,
str  r3, [sp, #32] @ overflow_count_lsm_flag.56, %sfp
b    .L37
```

Listing 9: Inner loop of the IIR filter in C

```
for (int i = 2;
     i < input_length; i++)
{
    int16x4_t v_in =
        vld1_s16(in + i - 2);
    int32x4_t res =
        vmull_s16(v_in, vx);
    res = vaddq_s32(res, vround_num);
    int32x4_t shifted =
        vshrq_n_s32(res, 22);
    int32x2_t lo =
        vget_low_s32(shifted);
    int32x2_t hi =
        vget_high_s32(shifted);
    lo = vpadd_s32(lo, hi);
    lo = vpadd_s32(lo, lo);
    int32_t num_acc =
        vget_lane_s32(lo, 0);

    int16x4_t v_out =
        vld1_s16(output + i - 2);
    res = vmull_s16(v_out, vy);
    res = vaddq_s32(res, vround_den);
    shifted = vshrq_n_s32(res, 14);
    lo = vget_low_s32(shifted);
    hi = vget_high_s32(shifted);
    lo = vpadd_s32(lo, hi);
    lo = vpadd_s32(lo, lo);
    int32_t den_acc =
        vget_lane_s32(lo, 0);

    output[i] =
        saturate(num_acc - den_acc);
}
```

Listing 10: ARM Cortex-A7 NEON assembly for the inner loop

```
.L7:
    subs   r3, r1, #2
    vld1.16 {d18}, [r0]
    vld1.16 {d16}, [r3]
    vmull.s16 q9, d18, d20
    vmull.s16 q8, d16, d21
    vadd.i32  q9, q9, q12
    vadd.i32  q8, q8, q11
    vshr.s32  q9, q9, #22
    vpadd.i32 d6, d18, d19
    vpadd.i32 d6, d6, d6
    vmov      r3, s12
    vshr.s32  q8, q8, #14
    vpadd.i32 d7, d16, d17
    vpadd.i32 d7, d7, d7
    vmov      ip, s14
    sub    r3, r3, ip
    cmp    r3, #32768
    sxth   r5, r3
    asr    ip, r3, #31
    sbcs   r6, ip, #0
    blt    .L4
    ldr    r3, .L23+4
    movw   r5, #32767
    ldr    r6, [lr, r3]
    ldr    r3, [r6]
    adds   r3, r3, #1
    str    r3, [r6]
    b      .L5
.L4:
    cmn    r3, #32768
    adcs   ip, ip, #0
    blt    .L22
.L5:
    strh   r5, [r1, #2]!
    adds   r4, r4, #1
    cmp    r4, r2
    add    r0, r0, #2
    beq    .L1

    ...

.L22:
    ldr    r3, .L23+4
    mov    r5, #32768
    movt   r5, 65535
    ldr    r6, [lr, r3]
    ldr    r3, [r6]
    adds   r3, r3, #1
    str    r3, [r6]
    b      .L5
```

Listing 11: Input loop of the FIR filter in C

```
// Process 8 output samples
// per iteration
for (; n + 8 <= input_length;
     n += 8)
{
    int32x4_t acc_lo =
        vdupq_n_s32(0);
    int32x4_t acc_hi =
        vdupq_n_s32(0);

    for (uint32_t k = 0;
         k < coeffs_length; k++)
    {
        int16x8_t c =
            vdupq_n_s16(coeffs[k]);

        const int16_t *src =
            &input_with_history[
                n + coeffs_length
                - 1 - k];

        int16x8_t x =
            vld1q_s16(src);

        acc_lo = vmlal_s16(
            acc_lo,
            vget_low_s16(x),
            vget_low_s16(c));

        acc_hi = vmlal_s16(
            acc_hi,
            vget_high_s16(x),
            vget_high_s16(c));
    }

    int16x4_t out_lo =
        vqrshrn_n_s32(acc_lo, 15);
    int16x4_t out_hi =
        vqrshrn_n_s32(acc_hi, 15);

    int16x8_t out =
        vcombine_s16(out_lo, out_hi);

    vst1q_s16(&output[n], out);
}
```

Listing 12: ARM Cortex-A7 NEON assembly for the FIR input loop

```
.L3:
vmov.i32 q11, #0
vmov     q10, q11

mov      r7, r9
movs     r0, #0

.L4:
ldrsh    lr, [ip, #2]!
adds     r0, r0, #1

vld1.16  {d16-d17}, [r7]
vdup.16  q9, lr

vmlal.s16 q10, d16, d18
vmlal.s16 q11, d17, d19

sub      r7, r7, #2
cmp      r4, r0
bne      .L4

vqrshrn.s32 d22, q11, #15
vqrshrn.s32 d20, q10, #15
vmov     d21, d22

add      r0, r10, #8
add      r9, r9, #16

vst1.16  {d20-d21}, [r5]!

cmp      r2, r0
bcc      .L2
mov      r10, r0
b        .L3
```