Introduction

This report discusses implementation of two different digital filters in with embedded environment. The design and implementation of both FIR and IIR filters will reviewed. Techniques for optimizing the resulting C and assembly code will also be discussed. Finally, a prototype hardware and firmware function will be proposed with the intention of further reducing the code footprint.

Digital Filter Background Theory

Digital filters are used in numerous applications, such as audio signal processing, control systems, communication networks and data acquisition. There are two main classifications of digital filters: finite impulse response, and infinite impulse response.

Finite Impulse Response (FIR) Filter

Finite impulse response filters are considered to be the most simple form of digital filter. In digital signal processing, finite impulse response (FIR) filters exhibit a response of finite period when provided with a finite input. In other words, the filter output will settle to zero in a finite time. The following feed-forward difference equation describes the simplest FIR filter.

$$y(n) = \sum_{k=0}^{N} h(k)x(n-k)$$

Equation 1: Feed-forward Difference Equation [1]

Inspecting equation 1 reveals that an FIR filter is not dependant on any past outputs and therefore does not provide any feedback. Implementing an FIR filter therefore requires N multiplication and summation operations when computing one output value y[n], where N is the number of filter taps.

Design Requirements and Specifications

The following section provides technical specifications and requirements to which the FIR and IIR filter implementations will be designed to satisfy.

Finite Impulse Response

The FIR filter implementation will be designed to meet the following specifications:

- The filtering program will accept and output 16 bit audio sampled at 44.1 kHz
- Input and output audio files will be Waveform Audio File Format (WAV)
- The program must support FIR filters with no less than 128 signed fixed point coefficients
- Only fixed point arithmetic will be utilized

For the purposes of this project, the actual FIR filter and it's coefficients will be generated in MATLAB. In order to test the functionality of the filtering program, an audio file contaminated with a 250 Hz sine wave will be filtered with a notch filter centered at 250 Hz. This scenario also serves as an ideal test bench as the functionality of the filter implementation can be reviewed not only by listening to output audio but also by observing the reduction in power at 250 Hz in the waveform's spectrum. The following figure displays a portion of the unfiltered test audio's power spectral density plot. Clearly, a large power density is present at 250 Hz, the frequency of the contamination signal.

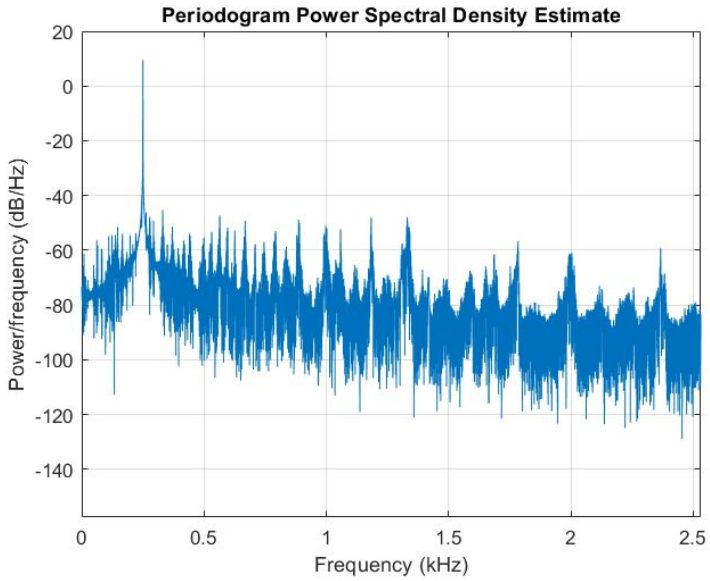

Figure 1: Spectral Power Density of Test Audio from 0 to 2.5 kHz

Design Process and Implementation

FIR Implementation

The following section discusses implementation of the FIR filter and includes a UML chart, discussion regarding a functional C code implementation and several optimization techniques employed to improve execution performance.

UML Chart

The following UML chart provides a basis for how the FIR filter implementation will behave . After opening the input file, the difference equation (Equation 1) is implemented by performing the following math operations.

$$x[n]\,h[0] + x[n-1]\,h[1] + x[n-2]\,h[2] + ... + x[n-k]\,h[k] = accumulated\ result$$

Were x[n] is the current 16 bit sample of the input file and h[0] to h[k] are 16 bit coefficients of the FIR filter. The individual multiplication results are accumulated into a single 32 bit result. This results is rounded back to a 16 and stored in the output audio file as a filtered sample. This process is repeated until n is equal to the length of the input file at which point the entire input file has been filtered and the routine stops.

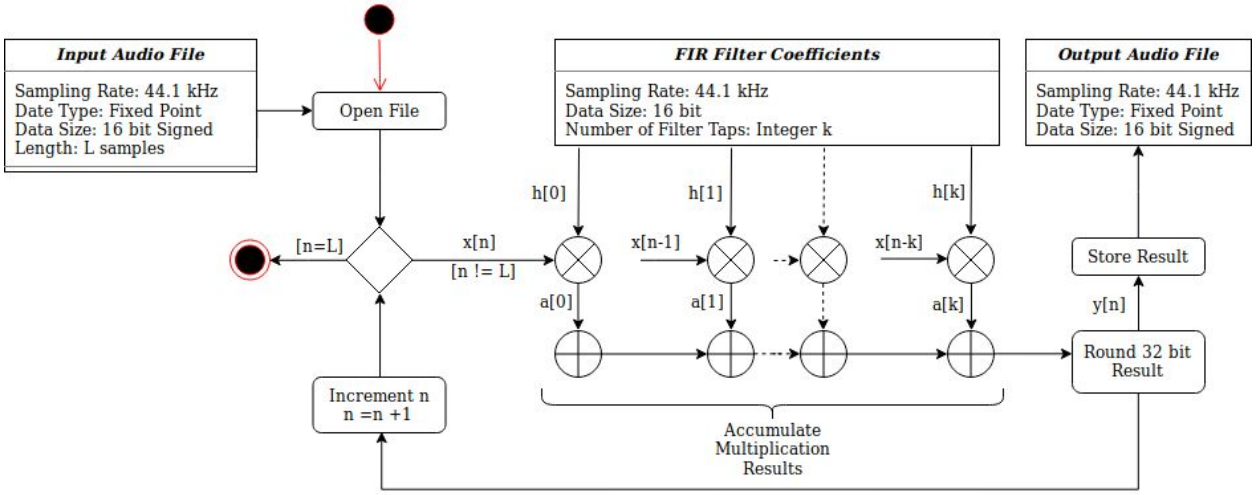

Figure 2: UML Chart for the Implementation of a FIR Filter

Software

C code written to implement FIR filtering while meeting the previously mentioned design specifications is included in package submitted with this project. The provided code filters the input audio file by loading frames of sampled audio into a buffer reserved in memory. This portion of the input is then filtered by the FIR routine. The routine simply implements the process described in the UML chart above. The outer for loop sets the pointer x to the current input sample and the pointer h to the address of the first filter coefficient. The inner for loop then multiplies the operands each pointer refer to. The x pointer is decremented such that it refers to the sample one previous while the h pointer is incremented to refer to the next FIR filter coefficient. The 32 bit result of the multiplication is rounded and shifted 7 bits left. This is done to help prevent overflowing the accumulator, to which each multiplication result is added to. This process is repeated until every filter coefficient is applied.The 32 bit accumulator, which

contains the sum of all previous multiplication results, is then rounded and shifted into a 16 bit result. This result is stored into the output frame. Once this routine runs through the entire input frame the output frame will be filled with filtered samples, which are in turn written to the output file. This process of filtering input frames is continued until the end of input file is reached.

Compiling and executing this code on an arm architecture linux system provides a filtered output with a reduced power density at the contamination frequency of 250 Hz, as can be seen by comparing the two images below.

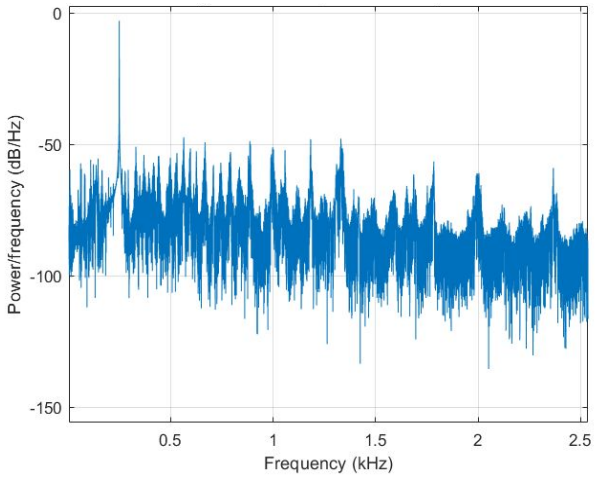

Figure 3 a): Signal Spectral Power Density of Filtered Test Audio from 0 to 2.5 kHz

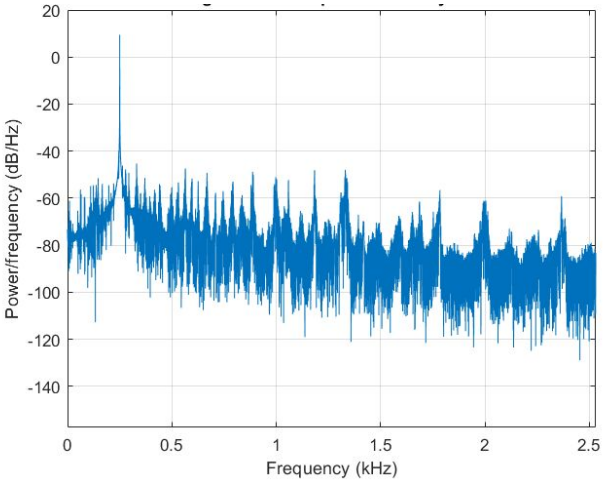

Figure 3 b): Signal Spectral Power Density of Unfiltered Test Audio from 0 to 2.5 kHz

Clearly, there still exists a contamination in the filtered signal at 250 Hz. Considering this results was obtained with a 500 tap filter designed in matlab, the shortcomings of implementing FIR filters on embedded ARM processors become evident. That is, a filter with many more taps would be required to remove the contamination to a point were it's lost to the human ear. Unfortunately, processing the thousands of taps required to achieve this would inevitably cause overflow when accumulating the multiplication results. However, for the purposes of this project, we are only interested in using the contaminated signal to benchmark the code.

Profiling this code with perf and gprof reveals it requires an average of 11.43 seconds to filter 30 seconds or 2.6 MB of audio. With 95% of that time being spent in the FIR function. With the bottleneck identified to be the FIR function, several optimization techniques were employed on the C code to improve execution time. The following techniques were implemented:

- *Locality*: The compiler is hinted to store the pointer values, counters, accumulator variable and temporary variables in registers to prevent the unnecessary store and load operations in the scope of the FIR function
- *Loop Unrolling/Grafting*: Temporary variables were declared such that two multiplication operations could be processed in parallel. This pair of multiplication is repeated three more times per iteration to reduce the amount of compare and branch operations

required to complete the inner for loop roughly by a factor of 4. Since this code was tested to not produce overflows without rounding after every multiplication operation, the temporary accumulators are only rounded and shifted once per iteration.

- *Exit Condition*: The inner for loop was edited to decrement and compare against zero. Therefore, theoretically the zero flag can simply be checked before branching. This saves a compare operation for every iteration.

It's important to note that the C code itself was also simplified by declaring the majority of the variables as globals. These optimizations resulted in an average execution time on only 1.55 seconds when filtering the same 30 second, 2.6 MB audio file. This is reduction by roughly a factor of 7.5. Next, the assembly code generated when compiling this code with GCC was optimized manually. Specifically the assembly code which represents the body of the inner loop of the FIR function was optimized. Multiply and accumulate operations were added manually in place of the separate multiply and add commands. This reduces the number of assembly operations from 52 to 37. When executing the resulting executable an average time saving of only 0.02s is achieved from the edited assembly.

Hardware Improvement

Even after manual optimization of the assembly code, the bottleneck in the execution is the multiply and accumulate operations central to the implementation of an FIR filter. Therefore an integrated hardware unit which performs this functionality plus the desired rounding has been proposed.

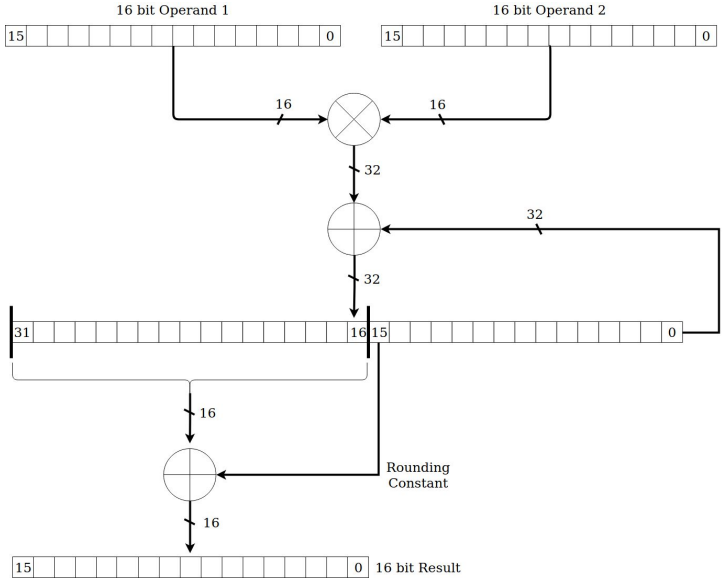

Figure 4: System Schematic for the MACSR Function

A simplified system schematic for this so called multiply and accumulate with shifting and rounding (MASCR) is displayed below. The following latency improvements are estimated by

assuming that each stage of the MASCR requires one cycle to complete, were the multiplication and addition are performed in a pipeline (ie. the previous multiplication result is accumulated as the current multiplication results is computed). For example, when operating on a 500 tap filter, the MASCR operation would require 502 cycles. Were 501 cycles are required to complete the multiplication/accumulation, while the final cycle is simply for rounding and shifting. Operands would simply be shifted out of a buffer into the hardware function, removing the need to execute LDR instructions. Assuming that the MLA assembly instruction requires 6 cycles, the ADD, LDR and ASR instructions all require 1 cycle, then the estimated number of cycles saved per iteration of the FIR function is as follows.

$$125((\#MLA \times 6) + LDR\ (\#ADD) + (\#ASR)\ + (\#LDR))\ -\ 502\ =$$
$$125((6 \times 8) + 2 + 2 + 16)\ - 502 = 8500 - 502 = 7998\ cycles$$

Implementing this hardware function would therefore provide a significant improvement in latency.

Performance and Cost Evaluation

FIR

While large improvements were observed when optimizing the C code, the improvements gained when implementing the MACSR operation in hardware would be well worth the cost. Especially if the filtering was to complete in real time in a control system or similar applicati

References

[1] M. Sima, "Lesson 106 - Digital Filters", 2019.