Fixed Point vs Floating Point Filters in an FPGA Implementation

Yanni Coroneos and Corey Walsh

Motivation

Audio systems composed of multiple speaker and input sources require different kinds of signal filters for optimal quality. Low pass filters are used for subwoofers, high pass filters for tweeters, and allpass filters are used to compensate for transmission delay in long lines. Unfortunately, one size filter does not fit all, so every audio system needs a specialized set of filters. Audio filters are either implemented in continuous time (CT) using analog circuitry or discrete time (DT) using digital circuitry. CT filters can produce impressive numbers but they are costly to implement because of their size, power consumption, and peculiarity to a single audio system. Conversely, DT filters are especially convenient because they can be implemented on re-programmable hardware such as CPUs or FPGAs. Changing a digital filter to accommodate a new audio system is much cheaper and faster than it is to re-assemble analog circuitry.

Since audio signals are intuitive to work with, this project uses audio as the motivating factor for every design decision and also in the final benchmarks.

FPGAs : Digital Legos

An FPGA is a grid of logic blocks with re-programmable connections. Any combinational or sequential digital circuit can be implemented on an FPGA by describing the interconnections of all the transistors. FPGAs are a fantastic choice for digital filters because all digital filters can be represented in a sequential circuit as a data pipeline. Since all of the processing pathways in an FPGA is described by the programmer, FPGAs can provide extreme timing guarantees and massive parallelism. In this project, we use an xc7a200t FPGA to implement FIR digital filters.

FIR Digital Filter Algorithm

A finite impulse response (FIR) filter, is a filter which has a response of finite length to any finite input, such as an impulse response. The filter itself is defined by a finite set of N coefficients $a_i \in \Re$. An input is defined as a sequence values $x[n] \in \Re$. The output of the filter can be seen as the convolution between the coefficients and the input sequence $y[n] = a * x[n]$.

$$y[n] = \sum_{i=0}^{N} a_i x[n-i]$$

Intuitively, the filter output represents a weighted sum of the previous N inputs. With correct choice of weights (coefficients), this sum can implement a variety of useful filters, such as low pass, high pass, or band pass filters. These filters are very useful in all forms of digital signals

processing applications. For example, such filters are used in crossover systems for multi-way sound systems where different speakers reproduce audio in different frequency bands.

In this project, we normalize the inputs and coefficients to the range [-1,1]. Additionally, the coefficients are scaled such that their sum is also in the range [-1,1] so that the filters do not have a large gain. The coefficient scaling constraint not necessarily present in the general case, but it provides good properties for audio filtering, which is the primary signal that this project focuses on.

FIR Z Transform

It is also insightful to inspect the FIR filter in frequency space by taking the discrete time fourier transform.

$$x[n-k] \ <=> \ X(z)z^{-k}$$

$$Y(z) \ = \ \sum_{k=0}^{N} a_k X(z)z^{-k}$$

The DTFT of the FIR filter gives us insight into the phase response. By recalling that $z^{-1} = e^{-jw}$, we can rewrite the Z transform above as:

$$Y(e^{-jw}) \ = X(e^{-jw}) \ \sum_{k=0}^{N} a_k e^{-kjw}$$

Now it is visible from e^{-kjw} that FIR filters have a linear phase response and a constant group delay! This is very good for audio quality because it means that every frequency will get delayed by an equal amount, so no interference distortion results from phase mismatch. This is an important property of FIR filters that recursive digital (infinite impulse response) filters do not possess.

The computational complexity of an FIR filter scales linearly with the number of coefficients. Assume that there exists an FIR filter with *n* coefficients. That means that the filter computes *n* multiplications and *n-1* additions. Assuming that multiplication and addition are each a single FLOP, this results in a total complexity of 2*n*-1 FLOPs. However, this idealistic estimate is not necessarily accurate since physical hardware constraints can cause the complexity to increase.

Implementation challenges in an FPGA

While the FIR algorithm is conceptually very simple, there are a few practical complications that arise when implementing it on real hardware. On an FPGA, one is limited by the number of computational blocks available for configuration. For example, the operation:

$$Y = A + BX$$

Is so common that there are specialized units on the FPGA, called DSP slices, that perform this single operation. The maximum bit width of any operand into a DSP slice on Xilinx FPGAs is 48 bits. Once the width of a single input increases past 48 bits, *another* DSP slice must be used to

compute the product. This is a problem because the FPGA does not have very many DSP slices to use. Entry level FPGAs typically only have a few hundred DSP slices.

This constraint means that one must take care to keep the size of the circuit to a minimum, which is directly at odds with the fact that some filters (such as low pass filters) require a large number of coefficients. The size constraint on an FPGA is related to the primary numerical challenge in designing for an FPGA - discretizing the theoretically continuous inputs, outputs, and intermediate products in a way that does not introduce a large amount of error, or unduly increase size requirements.

To utilize fixed point arithmetic, a rational number must be converted by scaling to the maximum chosen bit-depth and then rounding. For example, for a 24 bit depth, one multiplies the rational inputs and coefficients by 2^{23}, rounds, clips, and converts the values in the range [-2^{23}, 2^{23}] to two's complement.

Since inputs are multiplied by coefficients, intermediate values can potentially require 48 bits to be fully represented. A further $log_2(N)$ bits is required in the worst case to represent intermediate additions. In our implementation, we use 50 bit registers to represent intermediate products. This is problematic because it requires a large number of computational blocks on the FPGA, as we will quantify later in this report. Additionally, this large number of bits does not guarantee a low level of error, because the output values must be scaled back into the 24 bit range for the filter output, causing precision loss.

Given the challenges with fixed point arithmetic, floating point arithmetic seems promising since smaller registers may be used to represent numbers with potentially better precision. However, floating point operations present their own challenges, such as the fact that some numbers are not representable in floating point, or that adding values of very different magnitude can cause loss of precision. In any case, it is more difficult to implement floating point logic on an FPGA since the IEEE 754 standard is significantly more complex than two's complement.

Implementation and Experimental Results

We have implemented the FIR algorithm using both fixed and floating point arithmetic using the fully pipelined Systolic Multiply-Accumulate architecture in Verilog. Additionally, we have performed a variety of tests in simulation in order to compare the error characteristics. Testing was performed in simulation primarily due to difficulties in recording digital inputs and outputs from the real FPGA. Since the simulator is cycle-accurate, we expect the simulation results to exactly match the output of the FPGA with the same circuit. This observation comes from the fact that FPGAs essentially implement boolean logic.

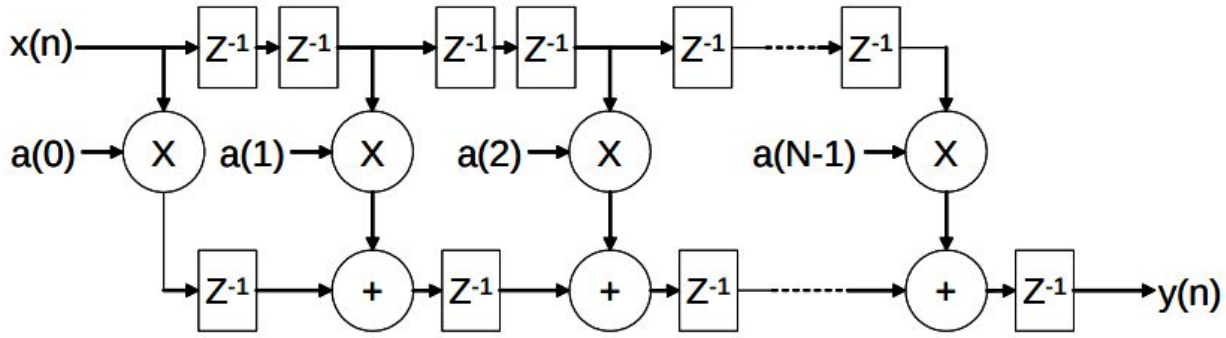

Figure 1. A diagram of the Systolic Multiply-Accumulate architecture.

Implementation of Fixed Point FIR Filter

The figure above shows the pipelined architecture of the fixed-point FIR filter that the FPGA implements. The filter actually takes three inputs even though only two are shown in the figure: input sample, coefficients, and a *sample-rate clock*. On every negative edge of the sample rate clock, there is a new input sample waiting at x[n]. The z^{-1} are unit delays and these are easily implemented with shift registers that are clocked on the sample-rate clock. The X's and +'s represent multiplication and addition, respectively, but these operations are the results of continuous assignments. Consequently, the arithmetic operations are not implemented with shift registers, but rather with wire primitives.

The first step of implementing this architecture is to parameterize it by the number of coefficients. The number of taps will decide how many, shift registers, multipliers, and adders to initialize. Also notice that there are two kinds of shift registers: those at the top of the figure and those at the bottom. They perform different functions so it is useful to group them separately. In the code, the ones on the top are called *pipes* and the shift registers on the bottom are called *delays*. From the figure, it is straightforward to determine how many *pipes*, *delays*, multipliers, and adders to create:

$$N_{adders} = N_{taps} - 1$$
$$N_{mult} = N_{taps}$$
$$N_{delay} = N_{taps}$$
$$N_{pipes} = 2(N_{taps} - 1)$$

The FIR coefficients are provided to the filter as a single, very big, number which is meant to be sliced. For example, if a filter was implemented with 3 taps, each with a width of 32 bits, then the coefficients will be provided as a single 96 bit number and each tap of the filter statically slices the input coefficients at the appropriate spot. In Verilog notation, the coefficients are declared as

$$coefficients[NTAPS * WIDTH \ : \ 0]$$

In this project, the FIR coefficients are stored directly in the FPGA's block RAM. This is extremely wasteful for a production system because the coefficients that are stored in BRAM utilize most of the LUTs inside the FPGA. A production system should dedicate a few megabytes of DRAM for coefficient storage instead.

The next step of the implementation is to decide the register widths of each adder, multiplier, and shift register. To avoid integer overflow, each register in the critical path of the final output must be specified for the largest product it can possibly contain. In this implementation, both the input width and coefficient width are 24 bits. This leads to a 48 bit multiplication result for each multiplier. Since each adder can contain the sum of all 48 bit multiplication results that come before it and there are $N_{taps} - 1$ adders, the bit width of each adder should be

$$width_{adder[i]} = \sum_{k=0}^{NTAPS-1} 48 + k$$

The FIR implementation in the FPGA does not follow this rule for the adders though. Instead, the coefficients for the filter are constructed so they sum to 1.0. This ensures that no intermediate product in the FIR pipeline will require more than 48 bits to represent. Unfortunately, rounding error during the discretization process can cause the coefficient sum to exceed 1.0, so the width of each adder is conservatively set to 50 bits.

Implementation of Floating Point FIR Filter

The floating point FIR filter is architecturally similar to that of the fixed point FIR filter. The major difference is that floating point arithmetic cannot be inferred like fixed point arithmetic. Instead of the inferred arithmetic operations, we explicitly instantiate 32 bit floating point adders and multipliers from an external library[1] which implements the IEEE 754 spec. This, alone, is not enough to convert the circuit to floating point though. Since the floating point arithmetic units require more than a single clock cycle to compute the result, an additional clock was created which is 200 times faster than the sample rate clock. The floating point arithmetic is clocked in this faster clock so that way the result is ready before the next sample comes into the filter.

Unlike the fixed-point FIR filter, the register widths in the floating point filter were all 32 bits due to the fact that floating point arithmetic does not increase the bit depth of the output product. This is a great boon for the floating point implementation because it means that the bit depth of the adders can stay constant throughout the whole filter. This prevents unnecessary DSP slices from being consumed and allows the computational complexity of the floating point FIR to scale linearly with the number of taps, just the like idealistic estimate.

[1] https://github.com/dawsonjon/fpu

Experimental Setup, Benchmarks, and Circuit Size

To facilitate testing, we developed a set of tools, and a test automation architecture. One notable utility is a Python script which allows us to convert back and forth between any relevant numerical representation from the command line. A high level overview of our test automation is available in pictorial form below.

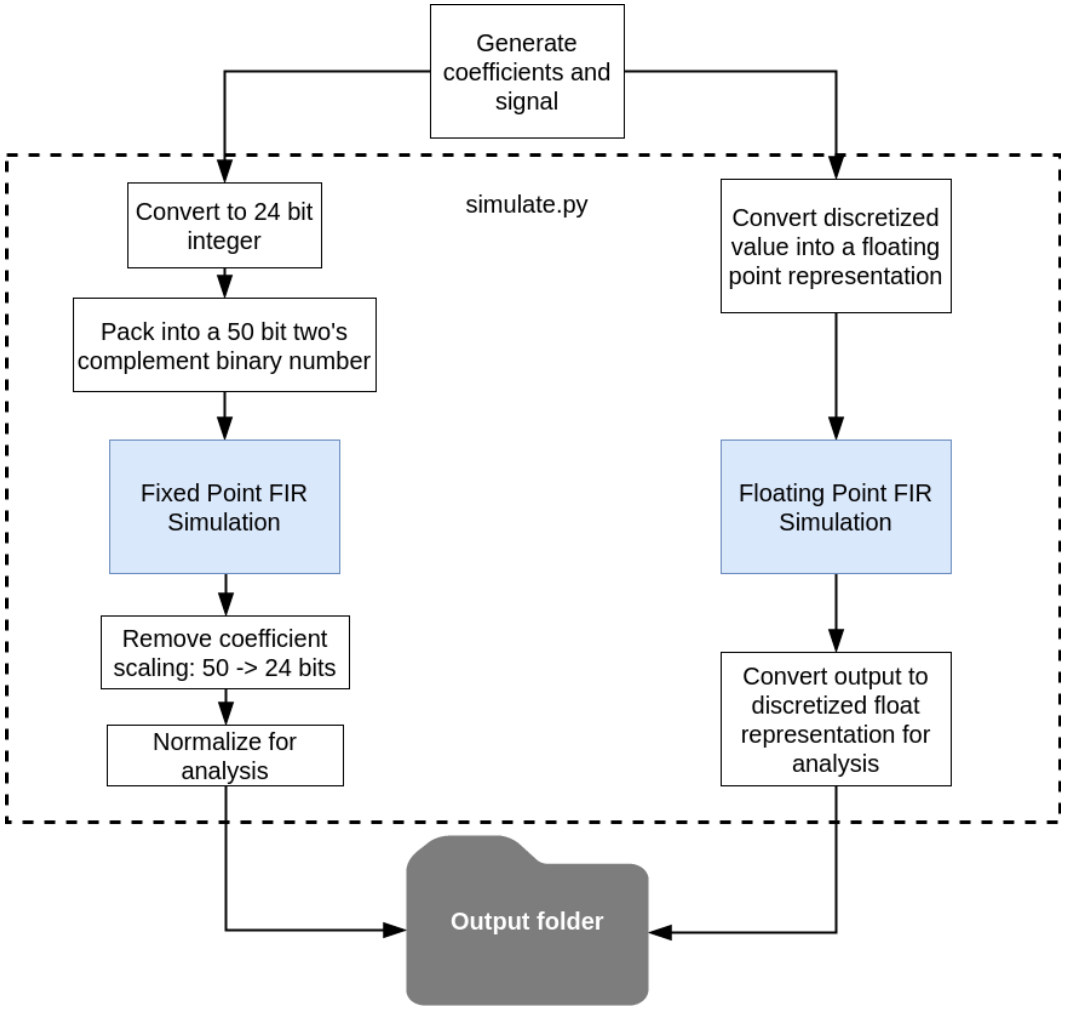

Figure 2. Overview of simulation architecture. Boxes in blue represent behavioral simulation of the FPGA circuit. Other boxes were implemented in Python.

We have also developed utilities that allow us to plot input and output signals, as well as error characteristics. To generate some of our figures, we use the above simulation architecture inside of another loop which generates a wide range of signals.

Filters

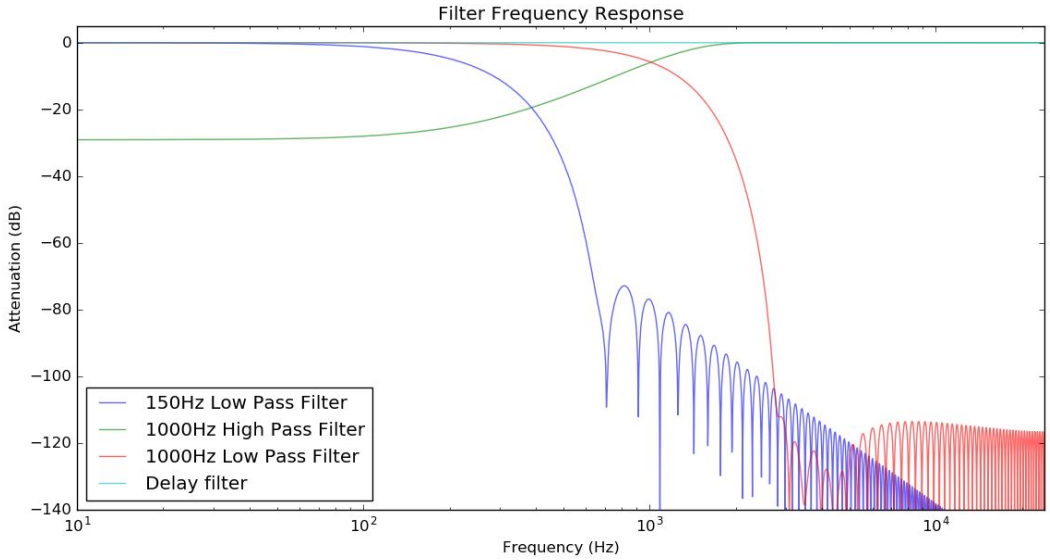

Figure 3. Frequency response for experimental filters.

We consider four different sets of coefficients with frequency responses demonstrated above.

- 107 tap 1000Hz high pass filter, using a Nuttall window
- 107 tap 1000Hz low pass filter, using a Nuttall window
- 107 tap delay filter (all coefficients 0,0 except for one, which is 1.0)
- 277 tap 150Hz low pass filter, using a Blackman window

Chirp

To provide an overview of the error characteristics of the two FIR implementations, we use a chirp signal in which the frequency of the signal increases with time. We tested the response using a 1000Hz high and low pass filter, as well as a 150Hz low pass filter.

10-2000Hz Chirp, 150Hz low pass filter

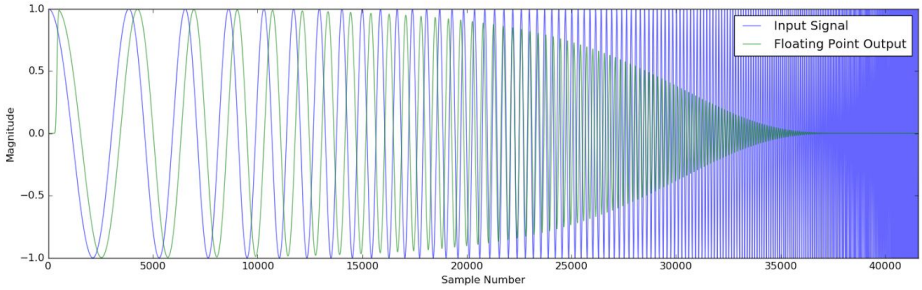

Figure 4. Input (blue) and output (green) signals for the 150Hz chirp with a low pass filter.

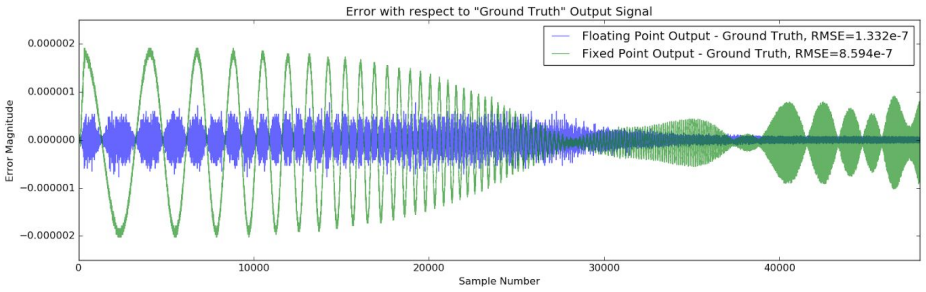

Figure 5. Error with respect to ground truth output signal for fixed (green) and floating (blue) point filters.

10-20000Hz chirp, 1000Hz lpf

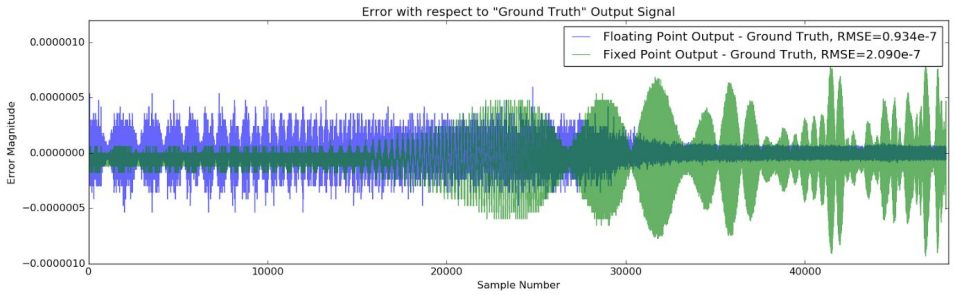

Figure 6. Error with respect to ground truth output signal for fixed (green) and floating (blue) point filters. Input and filtered signals similar to (Fig. 4) and thus not pictured.

10-20000Hz chirp, 1000Hz hpf

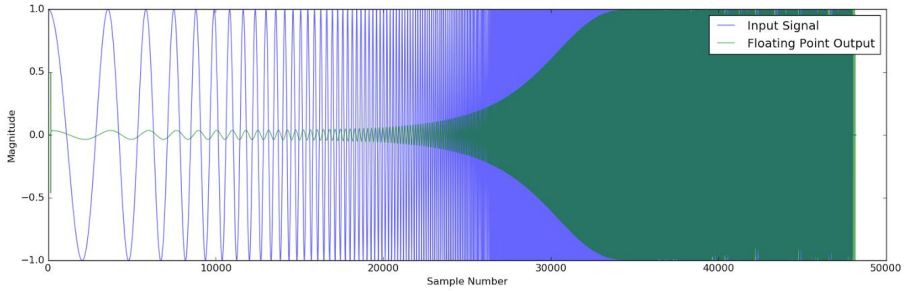

Figure 7. Input (blue) and output (green) signals for the 10-20000Hz chirp with a 100Hz high pass filter.

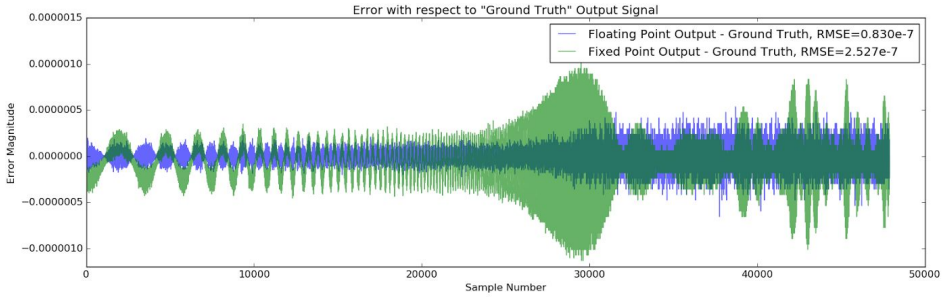

Figure 8. Error with respect to ground truth output signal for fixed (green) and floating (blue) point filters. Input and filtered signals similar to (Fig. 4) and thus not pictured.

Sine Waves

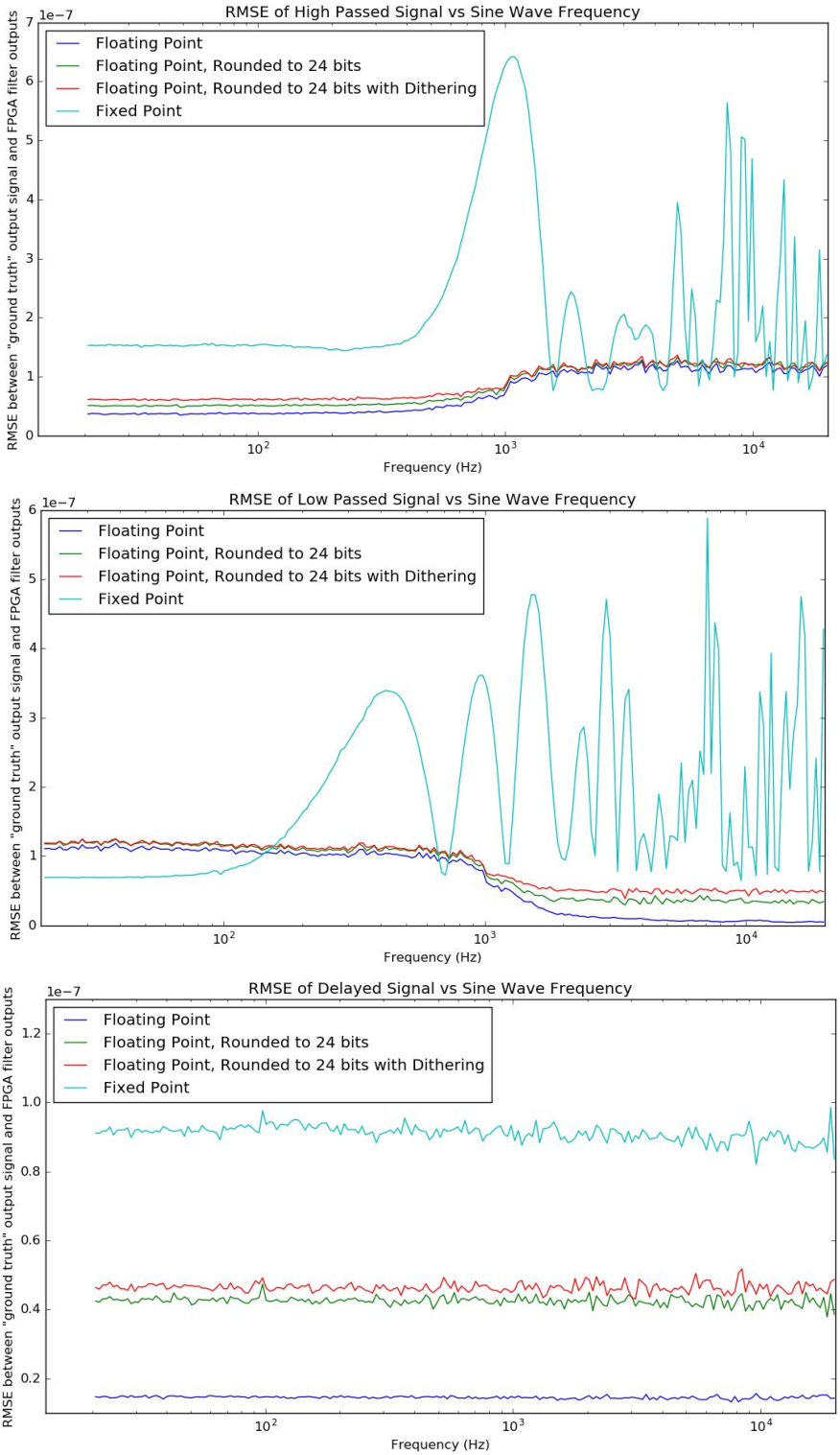

Figure 9. We generated 200 sine waves of various frequency ranging from 20 to 20000Hz. These sine waves were fed through three test filters, and an RMSE value was computed by comparing the filter output to the ground truth value computed with 64 bit float precision. The error is shown in this figure for each of the fixed and floating point filters. The green line represents the error the floating point filter would

incur. The teal line represents the fixed point error. Blue line is float output without discretization to 24 bits.

Mahal

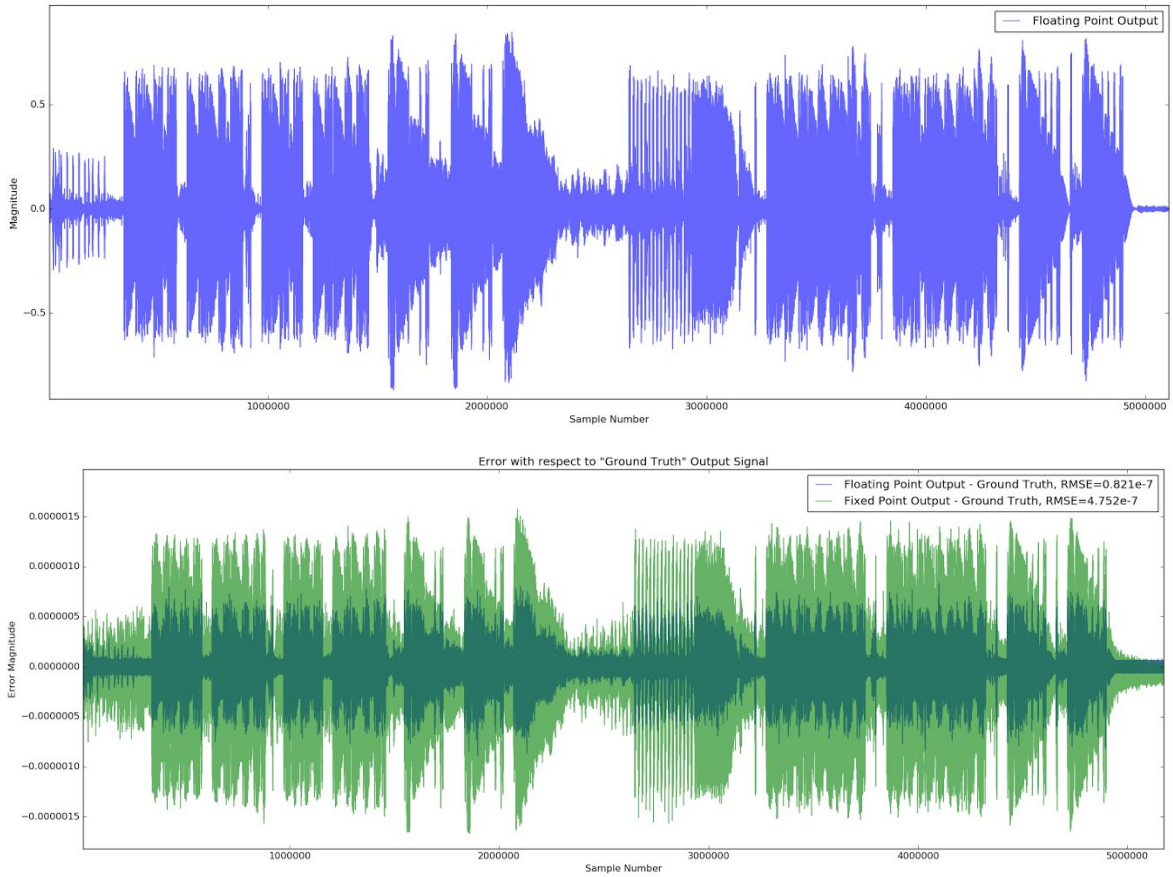

Figure 10. FIltered signal (above) and error with respect to ground truth for the fixed and floating point outputs (below). Floating point output (dark green) is discretized to 24 bit fixed point range on the output since this would be necessary in the real FPGA system. Fixed point error shown in light green.

To demonstrate real world operating conditions, we have simulated the response of a low pass filter to the song *Mahal* by Gent and Jawns. We choose a 150Hz low pass filter since it allows us to isolate the bass drop from the rest of the song. In the author's opinion, while using this filter, one may always listen to the best part of music.

Simulation is very slow as compared to real-time operation on the FPGA. Thus, we sampled Mahal and divided it into ten chunks. The low filter was simulated with these chunks using eight processor cores for approximately ten hours. After simulation completed, we recombined the chunks by aligning the phase of the output signals, and plotted error metrics over the course of the whole song.

We found that in this representative example, the fixed point output had roughly 5.78 times more error than the floating point alternative.

Measurement Results

FIR Type	Taps	Bit Depth	DSP Slices Used
Fixed Point	100	50	900/740
Floating Point	100	32	200/740

In order to measure circuit size, we synthesized two FIR filters that compute the same result using the Xilinx Vivado synthesizer. One is a fixed point filter and the other is floating point. The synthesis summary is shown above. In this case, the fixed point filter is too big to fit on the FPGA while the floating point filter barely consumes 25% of the DSP slices. For the engineer who uses floating point, the work is over. But the engineer who uses fixed point must now dig into the FIR implementation and attempt to time multiplex some arithmetic units so that the filter can fit on the FPGA; this is a laborious process and completely avoidable with the floating point FIR filter.

Conclusions

In our work, we found that using floating point arithmetic in an FPGA based FIR filter yields better error characteristics than fixed point arithmetic in most test cases, despite having a physically smaller circuit. We attribute the accuracy of the floating point results to the fact that it does not require lossy scaling operations which introduce error into the calculation. The larger size of the fixed point circuit is mostly due to the fact that integer arithmetic overflows necessitate the use of more bits to represent intermediate than both the input values and filter coefficients combined.

Despite lack of native support in Verilog, we found that implementing the FIR filter with floating point logic was in some ways more straightforward than doing so with fixed point logic. In particular, we did not have to manually consider the scaling and bitness required at each of the intermediate stages of the algorithm to achieve correct results, as was the case with fixed point.

One potential drawback of the floating point method is that arithmetic operations require many clock cycles to resolve, as compared to the single clock cycle required by fixed point operations. In an audio context, this does not pose a significant challenge since the clock rates available to FPGAs are significantly in excess of the audio sample rate.

In future work, it would be interesting to investigate the use of an extended-precision floating point standard which utilizes 48 bits instead of the 32 that we used experimentally. Doing so could potentially reduce error further, while not significantly increasing the size of the resultant circuit.

References

1. *IEEE Standard for Floating-Point Arithmetic.* IEEE Standard 754, 2008.
2. Dawson, J. IEEE 754 floating point arithmetic. Github repository. 2013. <https://github.com/dawsonjon/fpu>
3. XILINX, Inc. “IP LogiCORE FIR Compiler v5.0.” San Jose, CA, 2011. pp 17.
4. Gent & Jawns. *Mahal*. Los Angeles: Lowly Palace, 2017. Online. <https://soundcloud.com/lowlypalace/gent-jawns-mahal>