Semester Project

Architectural exploration of low-pass FIR filters using MATLAB HDL Coder

Student: Nikolaos Stylianou

AEM: 2917

Course: ECE 494 - Processor Design

June 28, 2023

Contents

1 Introduction

A low-pass FIR (*Finite Impulse Response*) filter is a type of digital filter used in signal processing to attenuate or remove high-frequency components from a signal, allowing only low-frequency components to pass through. FIR filters are characterized by their impulse response, which is a finite duration sequence of coefficients. The basic operation of a low-pass FIR filter involves convolving the input signal with the impulse response of the filter. The filter coefficients determine the filter's frequency response, and specifically in a low-pass filter, they emphasize the passage of low-frequency signals and suppress higher frequencies The design of a low-pass FIR filter involves determining the appropriate filter order (length) and selecting the filter coefficients. The filter order determines the length of the filter's impulse response and affects the sharpness of the filter's frequency cutoff. Higher filter orders generally result in steeper roll-off characteristics but require more computational resources.

For this project, I will be focused in exploring

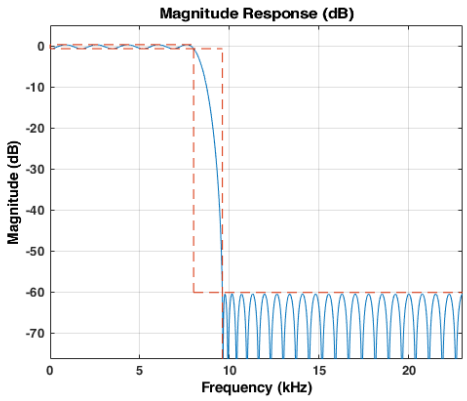


Figure 1: FIR magnitude response

the architecture of an FIR filter with the following characteristics:

- Sampling frequency: 46 kHz
- Pass frequency: 8 kHz
- Stop frequency: 9.6 kHz
- Number of coefficients:
  - Minimum coefficients possible
  - 20 coefficients
  - 30 coefficients
- Arithmetic:
  - Single precision floating point
  - Fixed point 24 bits
  - Fixed point 16 bits
  - Fixed point 8 bits
- Optimizations:
  - 1 stage multiplier pipelining
  - 2 stage multiplier pipelining
  - Multiplier-less using CSD
  - Distributed Arithmetic

The implemented filter has a magnitude response as figure 1 shows.

2 Filter Creation

Before creating any HDL code, the filter has to be normally implemented in MATLAB. Using `designfilt` command, every parameter of the filter is filled with the help of GUI. We call this function 3 times, one for each different order of filter, thus creating 3 different FIR filters. Using

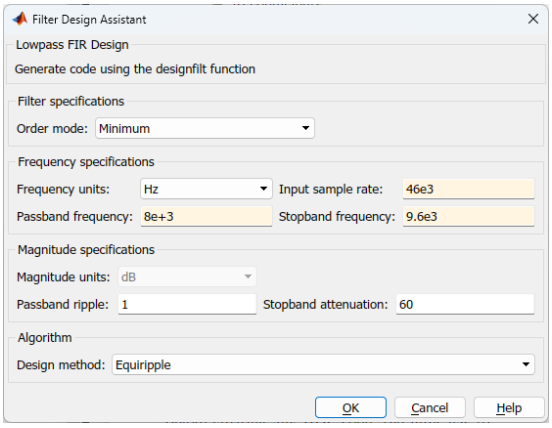


Figure 2: MATLAB's design assist tool for creating lowpass FIR filters

`fvtool`, all three filters can be compared and be checked for compliance.

For HDL Coder to work, all filters must be designed by `fdesign.lowpass()` and `design()` functions, thus all filters created above must be re-created. The first function creates the specification for each filter containing useful information such as passband/stopband frequency, sample rate, etc. In order to actually create the filter, `design()` function has to be called and it returns `dsp.FIRFilter` data type, meaning that the wanted filter is successfully created. Note that, we make use of two specific toolboxes provided by Mathworks, called *DSP System Toolbox* and *DSP HDL Toolbox* that provide those "easy" methods of creating filters alongside with optimizations like pipelining, CSD and others needed for this project.

After each filter is created, `fdhdltool` has to be called with filter specifications and numeric type needed as arguments. Every aspect of the generated filter can be tuned from there such configuring architectures, multiplier pipelining, test bench generator and more. This task has to be executed for each implemented filter.

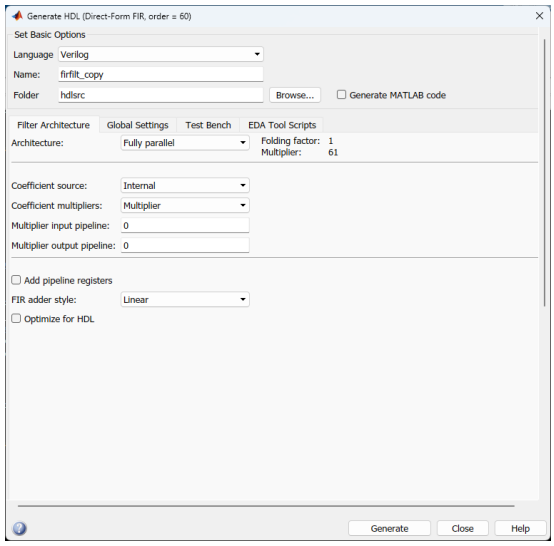


Figure 3: MATLAB's tool for creating HDL filters.

3 Filter Synthesis

Once creating every filter with its own configuration, each HDL file has to be synthesized and/or implemented in order to compare each architecture. For this process, Xilinx Vivado seems to be the best option to use as other great EDA tools like Synopsys and Cadence suites weren't available. Since I'm using Vivado, I should as well target an FPGA that might have access such as the Zedboard. Zedboard isn't a very big FPGA in terms of memory size, but, hopefully, it might be able to implement some filters at the end.

3.1 Importing MATLAB HDL files

In Vivado, a new project is created for each filter architecture. Files are imported using the standard GUI procedure while also adding constraints for the target device mentioned above. MATLAB produces source HDL and test-bench for each filter created as well as some `.do` files that tell Xilinx's compiler what to do.

3.2 Synthesis in Vivado

After importing all necessary files into Vivado, we must assure the functionality of the exported circuit. First we set the target device, which as said before is the *Zedboard*, the target clock ($\approx$ 10 ns) and then simulation starts with the included simulator, *vsim*. Also, MATLAB streamlines this process by adding some specific tests inside testbenches, to lighten designer's workload. After functionality is ensured, synthesis takes place where utilization of LUTs (*Look Up Tables*), flip-flops and of other critical components is measured. So, this will be the way of comparing all architectures.

In order to get more accurate results, *constraints* for the target chip must be set-up. Those include mostly timing constraints that help Vivado predict the circuit's power consumption and whether it passes/fails the timing checks.

Finally, each project gets implemented to check if the systems meets the timing criteria introduced in the step before. After implementation is achieved, we start to compare each result and from an opinion about each architecture presented.

4 Filter Comparison

4.1 Architectural differences

The main difference of each pack of filters is the architecture itself. Using a standard multiplier, the filter coefficients are multiplied with the input samples using multipliers. The output is obtained by summing the products of the coefficients and input samples. This architecture is straightforward and produces accurate results but can be resource-intensive in terms of hardware implementation.

The second implementation is by using factored CSD., This is a technique used to reduce the number of partial products required for multiplication by decomposing the coefficients into a sum of powers of two. This architecture employs a combination of shifters and adders to implement the multiplication operation. The output is obtained by accumulating the partial products. Factored CSD reduces hardware complexity and power consumption compared to the multiplier architecture, but it may introduce some additional round-off errors due to approximation.

The last architecture used in this project is

called Distributed Arithmetic (*DA*). Distributed arithmetic is another technique for efficient multiplication using look-up tables (LUTs). In this architecture, the filter coefficients are precomputed and stored in a LUT. The input samples are used as indices to retrieve the corresponding precomputed values from the LUT. These values are then summed to obtain the filter output. Distributed arithmetic offers advantages in terms of reduced hardware complexity and power consumption but can introduce quantization errors due to the finite precision of the LUT.

So, when comparing these architectures, we can expect the multiplier architecture to provide the most accurate results but at the cost of increased hardware complexity and power consumption. The factored CSD and distributed arithmetic architectures offer trade-offs by reducing hardware requirements and power consumption while introducing minor approximations that result in slight differences in the output.

4.2 Minimum Order filter

Beginning with the minimum order one, overall utilization is fairly low due to the low number of coefficients used. From MATLAB, the minimum coeff. number that was able to produce an `FIR` filter of those specifications was **3**. Creating a filter with such low number of coefficients does not have a great filtering capability as can be seen from figure 4.

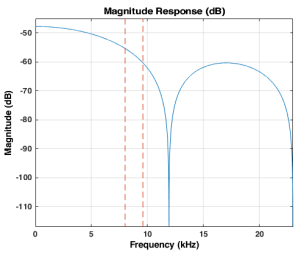


Figure 4: Magnitude response of the FIR filter with minimum number of coefficients.

Increasing pipeline stages from one to two, theoretically, increases operations per cycle but it didn't have the same impact on execution time. From figure 6, we can see that execution time of 1-stage pipeline is faster than 2-stage pipeline by a significant margin. This decrease in execution time is due to several overheads from the second pipeline stage but the increase of those stages can increase efficiency by dropping operating frequency while doing the same amount of computations in the same time (*because of the reduced slack*).

Knowing the latency of each architecture, we can calculate the overall execution time for a specific sample set. MATLAB's sample set consists of 3499 samples. So, using the following formula, we can calculate each architecture's execution time.

$$Execution\ time = \\ = \frac{Latency\ per\ sample \cdot Number\ of\ Samples}{Clock\ frequency} \tag{1}$$

By inserting numbers in the formula above, we get the diagram that is displayed in fig 6. Clearly, execution time follows the gradient of latency per sample. Moving on to the multiplier-less factored CSD architecture, we can start to observe a significant decrease in latency per sample, at first, and then in execution time, as this architecture outputs a new value at least 2 time-units quicker than any multiplier architecture mentioned above. This architecture achieves this speed by using small approximations on the output, which increase overall speed but does not affect much the filter's output.

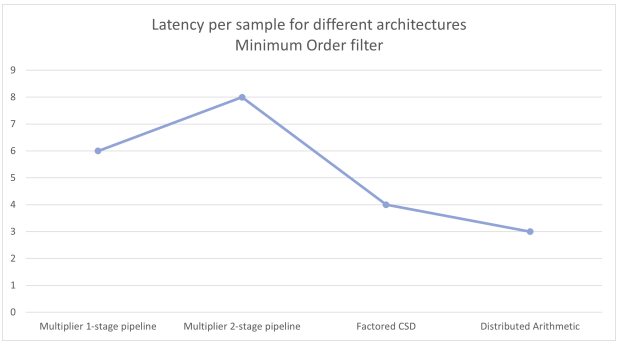


Figure 5: HDL latency for different architectures in samples.
Note: This is for 32bit arithmetic, the same pattern is observed for the other arithmetics.

One of the many things changing with the reduction of the computation bit width as well as with the different arithmetic is the `SNR` (*Signal-to-Noise Ratio*) of the filter. In order to measure the `SNR` of the exported HDL filters, we have to run

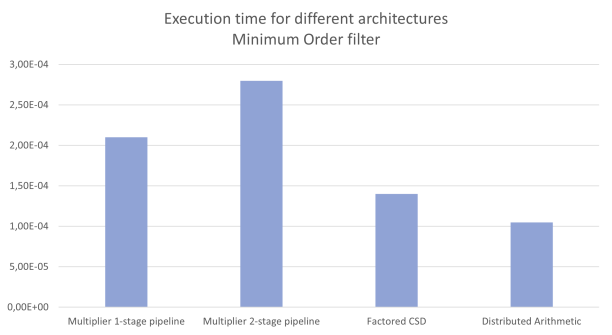


Figure 6: Execution time for different architectures in seconds.

the simulation in Vivado and capture two outputs, the expected one and the actual one and import them into MATLAB for further analysis. Then, filter expected error is calculated by subtracting the expected output from the actual one and by calling the `snr()` function we get the Signal to Noise Ratio to compare different arithmetics and architectures.

Figure 7 shows SNR values for this filter across multiple arithmetic bit-width and architectures. We can observe that SNR relatively stays the same for both multiplier architectures and multiplier-less CSD with values around 30.2 dB, with only DA having a very small value of 6.5 dB. This trend is followed for every bit width and is caused by the "bad" filter capabilities of the generated filter. Seeing the magnitude response of it in fig 4, we can clearly see that even from 0 Hz, magnitude is already at $-50\,dB$ which is not desired. If we were to add the losses of some architectures, as described in section 4.1, those numbers start to appear normal.

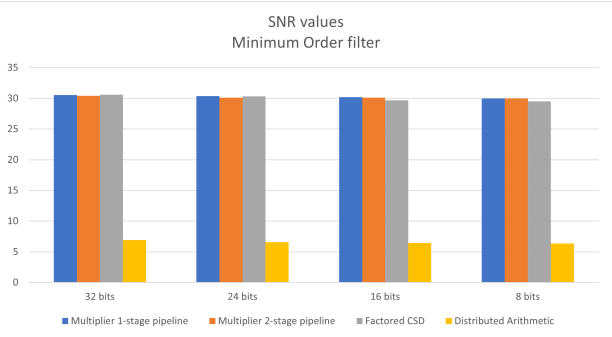


Figure 7: SNR values of the minimum coefficient filter

Finally, while the distributed arithmetic architecture seems to be the most quick of all above architectures as viewed in figure 6 for a specific arithmetic bit width and appears to be the most efficient of all (since LUT and flip flop utilizations are smaller than that of other architectures), it clearly isn't the correct one. This is the cost of every approximation technique applied by the architecture. Those approximations introduce errors that are more significant from those introduced in factored CSD architecture since those errors accumulate throughout the computation stages. Each stage involves a lookup operation and subsequent accumulation of values thus its impact is bigger on overall system accuracy.

4.3 20 Order filter

Moving to a filter which uses more coefficients, things start to distinguish in the filter architecture field.

Starting with execution times in figure 8, we can clearly see the same pattern occurred in the previous filter. Both sample latency and execution times have almost the same gradients and this is normal, as the only thing changing is the operations per sample (*increased to 41 from 7 in min. order filter*). As we change architectures from multiplier to multiplier-less (*either f-CSD or DA*), the number of operations per sample increase but multiplication is not taking place. Those operations are generally additions which are more cheap to implement (*mainly implemented using LUTs*).

Utilization also increased, since more computations are executed in one sample cycle. The filter order directly corresponds to the number of filter taps or coefficients used in the design. As the filter order increases, more taps are required to achieve the desired frequency response and filtering characteristics.

As far as SNR values are concerned, there is an improvement over the previous filter in both consistency and magnitude of numbers. Beginning with multiplier architectures, we observe a solid $\approx 38\,dB$ value for both 32 and 24 bits with steadily decreasing values for 16 and 8 bits arithmetic width ($\approx 34\,dB$).

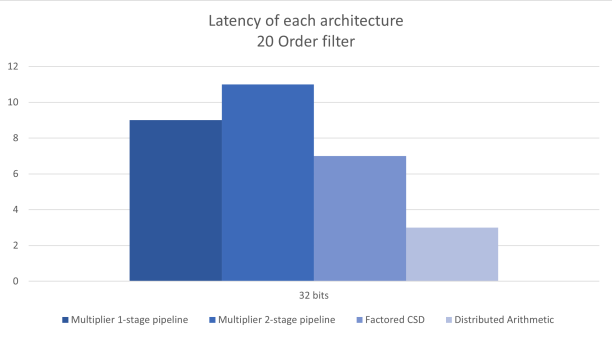


Figure 8: Latency per sample for different architectures.

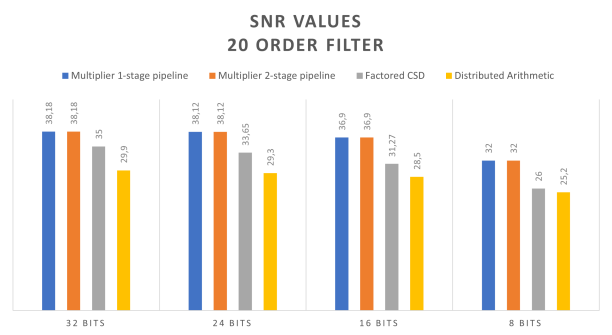
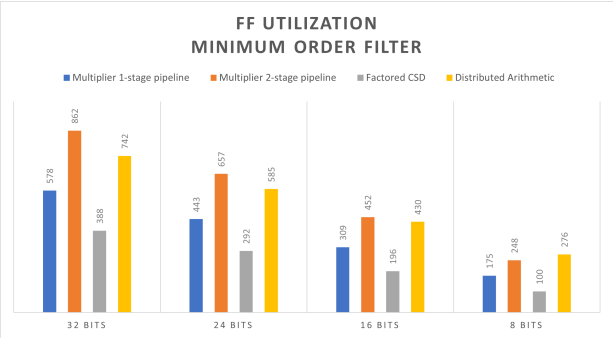


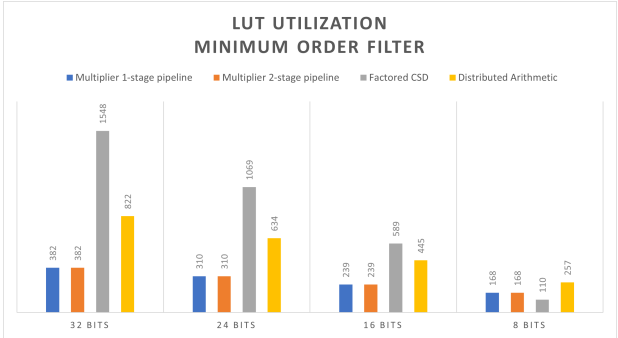
Figure 9: SNR values of 20 order filter for different arithmetics.

4.4 30 Order filter

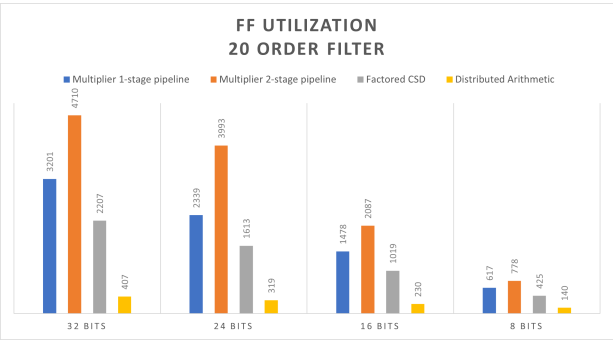
4.5 Overall utilizations



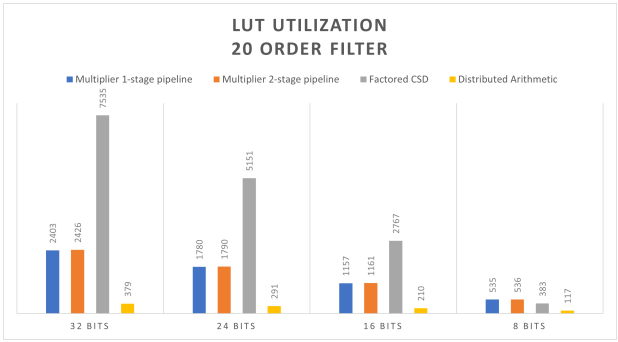
(a) FF utilization of min order filter



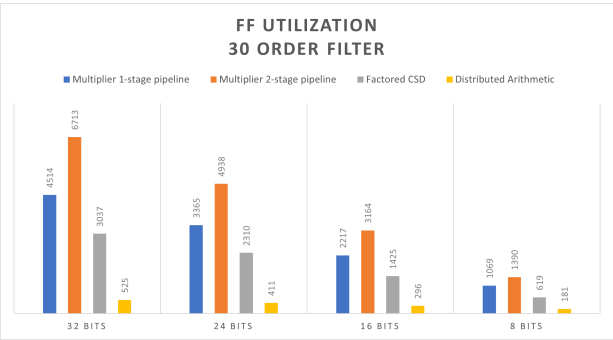
(b) LUT utilization of min order filter



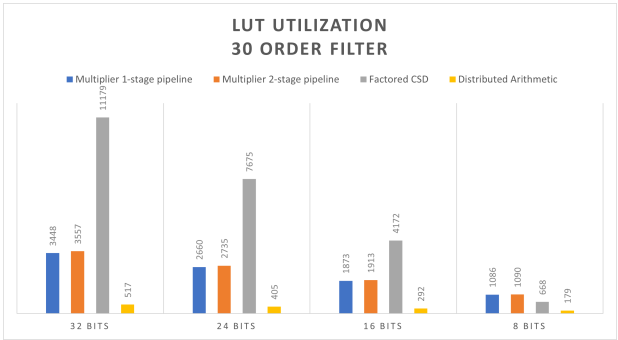
(c) FF utilization of 20 order filter



(d) LUT utilization of 20 order filter



(e) FF utilization of 30 order filter



(f) LUT utilization of min order filter

Figure 10: Various utilizations of LUTs and flip-flops for different architectures.