# CSE565M: Acceleration of Algorithms in Reconfigurable Logic

## Learn by Doing: Finite Impulse Response (FIR) Filters

Anthony Cabrera

FL24::L03

Washington University in St. Louis

# Table of contents

# Overview

# Learning Goals

The goal of this lecture is to provide a basic understanding of the process of taking an algorithm and creating a good hardware design using high-level synthesis. The first step in this process is always to have a deep understanding of the algorithm itself. This allows us to make design optimizations like code restructuring much more easily.

With this in mind, we'll

- cover a brief overview of FIR filter theory and computation.
- combine this knoweldge with the intuition developed from last class to introduce HLS optimizations to create an FIR filter in hardware.

# What are FIR filters? p. 1

- Finite Impulse Response (FIR) filters are commonplace in digital signal processing (DSP) applications.
- They are well suited for hardware implementation since they can be implemented as a highly optimized architecture.
- A key property is that they are a linear transform on **contiguous elements of a signal**.
  - This maps well to data structures (e.g., FIFOs or tap delay lines) that can be implemented efficiently in hardware.
- In general, streaming applications tend to map well to FPGAs, e.g., most of the examples in the book have some sort of streaming behavior [1].

# What are FIR filters? p. 2

Two fundamental uses for a filter are signal restoration and signal separation.

- Signal separation is perhaps the more common use case: here one tries to isolate the input signal into different parts, e.g., low-pass, high-pass, and band-pass filters.
- Signal restoration relates to removing noise and other common distortion artifacts that may have been introduced into the signal, e.g., signal smoothing or removing the DC component.

# What are FIR filters? p. 3

Digital FIR filters often deal with a discrete signal generated by sampling a continuous signal. The most familiar sampling is performed in time, i.e., the values from a signal are taken at discrete instances. These are most often sampled at regular intervals.

- e.g., we might sample the voltage across an antenna at a regular interval with an analog-to-digital converter.

Alternatively, samples may be taken in space.

- For instance, we might sample the value of different locations in an image sensor consisting of an array of photo-diodes to create a digital image.

# What are FIR filters? p. 4

The format of the data in a sample changes depending upon the application.

- Digital communications often uses complex numbers (in-phase and quadrature or I/Q values) to represent a sample.
- In image processing we often think of a pixel as a sample.
- A pixel can have multiple fields, e.g., red, green, and blue (RGB) color channels.
- We may wish to filter each of these channels in a different way again depending upon the application.

# Background

# Impulse Response, Taps, Convolutions

- The output signal of a filter given an impulse input signal is its **impulse response**. The impulse response of a linear, time invariant filter contains the complete information about the filter.
- Given the impulse response of an FIR filter, we can compute the output signal for any input signal through the process of **convolution**. This process combines samples of the impulse response (also called **coefficients** or **taps**) with samples of the input signal to compute samples of the output signal.

# Convolution of an N-tap FIR filter

The convolution of an N-tap FIR filter with coefficients $h[]$ with an input signal $x[]$ is described by the general difference equation:

$$y[i] = \sum_{j=0}^{N-1} h[j] \cdot x[i-j] \tag{1}$$

Note that to compute a single value of the output of an N-tap filter requires N multiplies and N-1 additions.

# Moving Average Filters

**Moving average filters** are a simple form of lowpass FIR filter where all the coefficients are identical and sum to one. For instance in the case of the three point moving filter, the coefficients are $h = [\frac{1}{3}, \frac{1}{3}, \frac{1}{3}]$. It is also called a **box car filter** due to the shape of its convolution kernel. Alternatively, you can think of a moving average filter

$$y[i] = \frac{1}{N} \sum_{j=0}^{N-1} x[i-j] \tag{2}$$

# Moving Average Filters (cont.)

Each sample in the output signal can be computed by the above equation using $N-1$ additions and one final multiplication by $1/N$. Even the final multiplication can often be regrouped and merged with other operations. As a result, moving average filters are simpler to compute than a general FIR filter. Specifically, when $N = 3$ we perform this operation to calculate $y[12]$:

$$y[12] = \frac{1}{3} \cdot (x[12] + x[11] + x[10]) \tag{3}$$

This filter is **causal**, meaning that the output is a function of no future values of the input. It is possible and common to change this, for example, so that the average is centered on the current sample, i.e., $y[12] = \frac{1}{3} \cdot (x[11] + x[12] + x[13])$.

# On Filter Coefficients

Filter coefficients can be crafted to create many different kinds of filters: low pass, high pass, band pass, etc. In general, a larger value of number of taps provides more degrees of freedom when designing a filter, generally resulting in filters with better characteristics.

When implementing a filter, the actual values of these coefficients are largely irrelevant and we can ignore how the coefficients themselves were arrived at.

However, the structure of the filter, or the particular coefficients can have a large impact on the number of operations that need to be performed

- e.g., symmetric filters have multiple taps with exactly the same value which can be grouped to reduce the number of multiplications.

But we will ignore that for the time being, and focus on generating architectures that have constant coefficients, but do not take advantage of the values of the constants.

# Base FIR Architecture pt. 1

# Base FIR Architecture

```
1  #define N 11
2  #include "ap_int.h"
3
4  typedef int coef_t;
5  typedef int data_t;
6  typedef int acc_t;
7
8  void fir(data_t *y, data_t x) {
9    coef_t c[N] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
10   static data_t shift_reg[N];
11   acc_t acc;
12   int i;
13
14   acc = 0;
15 Shift_Accum_Loop:
16   for (i = N - 1; i >= 0; i--) {
17     if (i == 0) {
18       acc += x * c[0];
19       shift_reg[0] = x;
20     } else {
21       shift_reg[i] = shift_reg[i - 1];
22       acc += shift_reg[i] * c[i];
23     }
24   }
25   *y = acc;
26 }
```

**Figure 1:** A functionally correct, but highly unoptimized, implementation of an 11 tap FIR filter.

# Base FIR Architecture pt. 2

The coefficients for the filter are stored in the `c[]` array declared inside of the function. These are statically defined constants. Note that the coefficients are symmetric. i.e., they are mirrored around the center value `c[5] = 500`. Many FIR filter have this type of symmetry. We could take advantage of it in order to reduce the amount of storage that is required for the `c[]` array.

The code uses `typedef` for the different variables. While this is not necessary, it is convenient for changing the types of data. As we discuss later, bit width optimization – specifically setting the number of integer and fraction bits for each variable – can provide significant benefits in terms of performance and area.

# Base FIR Architecture pt. 3

The code is written as a streaming function. It receives one sample at a time, and therefore it must store the previous samples. Since this is an 11 tap filter, we must keep the previous 10 samples. This is the purpose of the `shift_reg[]` array. This array is declared `static` since the data must be persistent across multiple calls to the function.

# Base FIR Architecture pt. 4

The `for` loop is doing two fundamental tasks in each iteration. First, it performs the MAC operation on the input samples (the current input sample `x` and the previous input samples stored in `shift_reg[]`). Each iteration of the loop performs a multiplication of one of the constants with one of the sample, and stores the running sum in the variable `acc`. The loop is also shifting values through `shift_array`, which works as a FIFO. It stores the input sample `x` into `shift_array[0]`, and moves the previous elements "up" through the `shift_array`:

```
shift_reg[10] = shift_reg[9]
shift_reg[9]  =  shift_reg[8]
shift_reg[8]  =  shift_reg[7]
            ...
shift_reg[2]  =  shift_reg[1]
shift_reg[1]  =  shift_reg[0]
shift_reg[0]  =  x
```

# Base FIR Architecture :: Exercise 1

## Implementing logic gates from LUTs

Rewrite the code so that it takes advantage of the symmetry found in the coefficients. That is, change `c[]` so that it has six elements (`c[0]` through `c[5]`). What changes are necessary in the rest of the code? How does this effect the number of resources? How does it change the performance?

# Base FIR Architecture pt. 5

After the for loop completes, the `acc` variable has the complete result of the convolution of the input samples with the FIR coefficient array. The final result is written into the function argument `y` which acts as the output port from this `fir` function. This completes the streaming process for computing one output value of an FIR.

**This function does not provide an efficient implementation of a FIR filter.** It is largely sequential, and employs a significant amount of unnecessary control logic. The following sections describe a number of different optimizations that improve its performance.

# References i


R. Kastner, J. Matai, and S. Neuendorffer.
**Parallel Programming for FPGAs.**
*ArXiv e-prints*, May 2018.