Introduction to Electronic System Design (DAT093) Lab 2: Behavioral VHDL and FIR Filters

Sven Knutsson, Lars Svensson

Version 2.1, September 17, 2018

1 Introduction

In Lab 1, you studied several ways to implement arithmetic operations on two's-complement numbers (the most-commonly-used representation for signed integers). This session builds on the previous one and extends it in these directions:

Behavioral VHDL code lets the designer specify hardware behavior without explicitly naming and connecting all components.

Fractional number representation offers a cheap[1] way to handle non-integer data, such as values representing a voltage level between –1V and 1V.

Finite-impulse-response (FIR) filters are typically used to suppress certain frequency components in a stream of data samples. They offer conceptually simple hardware with high-performance potential, although sometimes at a high cost for a given filter performance.

2 Preparation

Read the companion document on **Fractional numbers** to find out how to represent non-integer numbers in a simple way. Pay special attention to how

[1] Floating-point arithmetic, familiar from software programming and from applications such as MATLAB, is more expensive—in terms of hardware amount, processing time, and/or power and energy dissipation—and is therefore most often avoided in special-purpose hardware design.

scaling works, how to add two numbers with different binary-point positions, where the binary point ends up in the result of a multiplication, and how to change the numbers of bits used to represent some value.

An FIR filter implements the following operation:

$$y_n = \sum_{k=0}^{N-1} t_k x_{n-k} \tag{1}$$

where x_n is the stream of input values, y_n is the stream of output values, and t_k is the length-N vector which contains the filter impulse response. Each output value is a weighted average of the current and the $N-1$ previous input samples. N is also known as the filter length; the weights t_k are known as the filter tap values. If you are unfamiliar with these concepts, we recommend that you spend some time with Wikipedia[2] or with a specialized signal-processing reference. The lab-series summary presentation by Sven Knutsson provides an introduction.

Read through this entire lab PM and work through all tasks marked **Preparation**.

3 Direct-form implementation

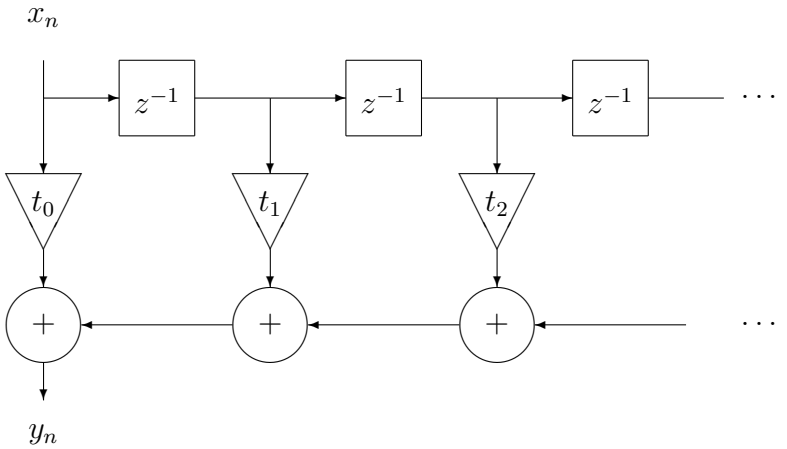

Figure 1: Direct-form implementation of FIR filter. Delay elements (marked z^{-1}) save the previous values of x_n needed to compute y_n. In the simplest case, each delay amounts to one clock cycle.

Figure 1 illustrates a straightforward implementation[3] of Equation 1. This is known as the *direct-form implementation*, or "direct implementation" for short.

[2] `https://en.wikipedia.org/wiki/Finite_impulse_response`

[3] Filters are ubiquitous in embedded systems and specifications such as Equation 1 may be

Your first task is to design and implement such an FIR filter. The first coding step, as usual, is to design the VHDL entity.

- Create a project and directory named `lab2a`.
- Download from the course home page the test bench and *do* file for the direct implementation of the filter.
- Create an entity for the filter in accordance with the test bench file. The asynchronous `reset` signal should clear all the delay elements and thus set the output to 0. Compile entity and test bench together to verify accordance.

Refer again to Figure 1. The implementation is clearly a repetitive, array-like structure, and therefore a good candidate for `generate` statements.

Preparation: Starting with the diagram in Figure 1, consider how to represent the hardware in an array-like form. In particular, consider how to assign names to signals such that port map expressions can be short and clear.

The filter will use fractional number representation; we will assume a wordlength of 8 bits (including the sign bit) for both the signals and the filter-tap values.

Preparation: Consider the wordlength of the miscellaneous signals. The filter tap constants and multipler outputs should have the same wordlength as the input signal (consult the **Fractional Numbers** document on how to arrange that). How can you select the wordlengths of the adder outputs so that no overflow can occur within the filter? What is the necessary wordlength at the output of the final adder?

The tap values for the filter are given in Table 1. As you can see, the filter is rather short, at only 4 taps; most practical filters are significantly longer. This choice was made for ease of debugging.

In Table 1, the filter tap values are given in decimal format, which must be translated to signed fractional numbers. Exact correspondence beween decimal and binary fractions is not possible except in special cases, so approximation will be necessary. Either truncation or rounding could reasonably be used; the test benches assume truncation, so use that.

---

realized in myriad ways. Consequently, there is a vast literature on filter designs which improve on this implementation. We will mostly stay simple in this lab (one optimization is shown as an optional task on page 6).

k	t_k
0	-0.32
1	0.23
2	0.23
3	-0.32

Table 1: Filter tap values

You should now be equipped to implement a VHDL architecture for the FIR filter.

- Implement the FIR filter architecture to fit with the entity you already built. Use behavioral-style code, that is, use `*` for multiplication and `+` for addition rather than explicitly stating what component to use.

  You will eventually want to be able to parameterize your design for different filter lengths and wordlengths; if you can consider this possibility already now, you may save time in the long run.

  Depending on how you choose to arrange your `generate` statements, you may find that you don't want identical hardware for each index value in the `for` range. A *conditional* `generate`, using an `if`–`generate` construct, may be of use then.

- Compile the architecture with the entity and the test bench. Fix compile-time errors, if any.

Compared with the designs of previous labs, this filter has a more complex behavior: its output depends on the input values over the past several cycles. Design verification therefore requires a bit more care. After the `reset` signal has been de-asserted and before any input data has been shifted into the filter, the input and all register values are at zero, so the output must also be at zero; as values are shifted in, signal values change progressively along the filter. This property may be used to locate errors with input sequences chosen to sequentially exercise small parts of the hardware; but you may have to modify the test bench and *do* file. The provided files are intended primarily as a final test for correctness rather than as a debugging aid.

- Verify the functionality of your filter architecture using the provided test bench and *do* file.

- In case of errors, make copies of the test bench and *do* file. Edit your copied files, re-compile, and re-simulate to find the bugs in your filter. Repeat until your design passes the tests in the original provided files.

The main benefit of parameterization is to increase reusability of a design. Not coincidentally, in Lab 6 in a few weeks, you will need to use a larger FIR filter. You will save much effort by being able to re-use this design then.

- Review your filter design once more, with special consideration of parameterizability. Would anything (except some constants and generics) need to be changed to implement a 30-tap filter instead?

4 Serial implementation

FPGAs intended for embedded/signal-processing include a limited number of hardwired multiplers (which require much less resources than multipliers built from general-purpose LUTs). As you have seen, the direct-form FIR implementation uses one multiplier per tap, which will limit the reasonable filter length for a given FPGA. One way around this limitation is to serialize the filter computations in a similar way as you did for adders and multipliers in Lab 1. The principle is illustrated in Figure 2. As one multiplier-accumulator now carries out the tap computations in sequence, this implementation will have a lower maximum processing throughput than the direct-form implementation.

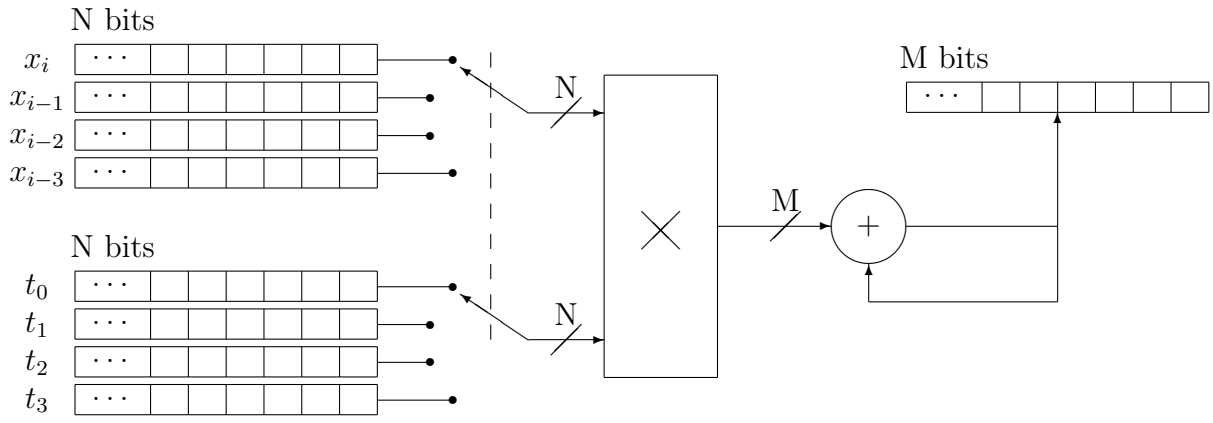

Figure 2: Serial FIR filter. The x values on the top left change as new data are shifted in; the t values are constant.

An illustration of the interface for the serial filter is shown in Figure 3. Because now several multiply-and-add operations will be performed for each new input signal value, a `start` signal is used to indicate that a new input value is to be read and the new evaluation of Equation 1 is to start. A `finished` signal is used to indicate when the new output value has been computed.

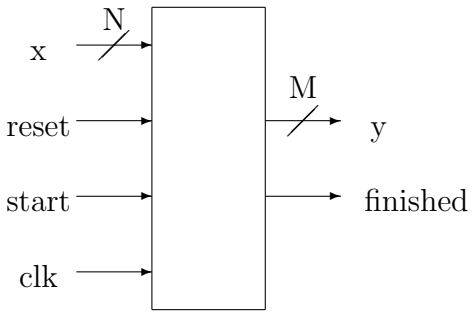

Figure 3: Interface for serial FIR filter

- Download the test bench and *do* file for the serial FIR filter from the course home page. Study the test bench to determine ports, generic parameters, and types expected for the filter.
- Create an entity for the serial filter implementation in accordance with the test bench and with Figure 3. Compile with the test bench to validate.

As usual, consider the hardware implementation of the serial filter before starting to implement the VHDL architecture. The `reset`, `start`, and `finished` signals should be active-high. The output value `y` should not be updated until evaluation is complete and the `finished` signal is asserted (it is often a good idea to avoid incorrect signals on output ports during computations).

- Implement the serial-FIR architecture in accordance with the entity you created above. Hint: you may find that your implementation can be quite similar to the direct-form implementation of Section 3.
- Verify your design through compilation and simulation. The serial implementation should compute the same results as the direct-form implementation did, but it will obviously take longer for each new value to be generated. After how many clock cycles is a new value generated?

What are the major similarities and differences between the direct-form and serial FIR architectures?

Optional task

The vector of filter tap values given in Table 1 is *symmetric*: the first and last values are identical, as are the second and the second-last values:

$$t_k \equiv t_{N-1-k}, k \in 0 \ldots N-1 \tag{2}$$

Symmetric filter tap vectors occur often in practice, so optimized implementations have been developed for this case. If N is even, Equation 1 may be rewritten:

$$\begin{aligned} y_n &= \sum_{k=0}^{N-1} t_k x_{n-k} \\ &= \sum_{k=0}^{N/2-1} t_k x_{n-k} + \sum_{k=N/2}^{N-1} t_k x_{n-k} \\ &= \sum_{k=0}^{N/2-1} t_k x_{n-k} + \sum_{k=N/2}^{N-1} t_{N-1-k} x_{n-k} \\ &= \sum_{k=0}^{N/2-1} t_k (x_{n-k} + x_{n-(N-1)+k}) \end{aligned} \quad (3)$$

The formulation of Equation 3 contains only half as many multiplications as Equation 1. The number of additions has increased by the same amount; but adders are cheaper than multipliers to implement.

- Consider how to modify the hardware of the direct-form filter of Section 3 to reduce the number of multipliers by half. Be careful to select the internal wordlengths to avoid overflow!
- Re-implement the VHDL architecture of the filter in accordance with your findings. As the behavior should be identical to the original architecture, you can re-use the test bench and *do* file from Section 3.

How would you implement this optimization for a serial FIR filter such as that described in Section 4?

5 Wrap-up

After completing this lab session, you are expected to be able to carry out the following tasks:

- Describe hardware using "abstract" mathematical operations such as `*` rather than by instantiating components
- Implement fractional arithmetic and scale fractional values appropriately when using them
- Implement simple FIR filters with parameterized filter lengths and word-lengths