Evaluating Simulink HDL Coder as a Framework for Flexible and Modular Hardware Description

by

Valerie Youngmi Sarge

S.B. Electrical Engineering
Massachusetts Institute of Technology (2018)

Submitted to the
Department of Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Author: ______________________________
Department of Electrical Engineering and Computer Science
May 18, 2018

Certified by: ______________________________
Paul Monticciolo
Group Leader, Lincoln Laboratory Embedded and Open Systems Group
Thesis Supervisor

Certified by: ______________________________
Vivienne Sze
Associate Professor, Research Laboratory for Electronics
Thesis Supervisor

Accepted by: ______________________________
Katrina LaCurts
Chair, Masters of Engineering Thesis Committee

Evaluating Simulink HDL Coder as a Framework for Flexible and Modular Hardware Description

by

Valerie Youngmi Sarge

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2018, in partial fulfillment of the
requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis investigates the performance and viability of Simulink and HDL Coder from MathWorks as an alternative workflow for producing hardware description. Several designs were implemented towards this end. An FFT-based signal analyzer served as a pathfinding application to better understand the tools. In order to directly evaluate the ability of the workflow to faithfully recreate hardware operations, an existing architecture for nonlinear equalization was re-implemented and benchmarked. Finally, a new implementation of polynomial nonlinear equalization was created and benchmarked to explore the possible performance, parameterizability, and flexibility of hardware generated from a Simulink design. It was found that while the generated hardware does not perform quite as well as a hand-optimized design, it does perform well enough to be practical and also can be capable of greater flexibility in structure than a design created with a more traditional workflow.

Thesis Supervisor: Paul Monticciolo
Title: Group Leader, Lincoln Laboratory Embedded and Open Systems Group

Thesis Supervisor: Vivienne Sze
Title: Associate Professor, Research Laboratory for Electronics

Acknowledgments

I would like to thank MIT Lincoln Laboratory for funding and support throughout this thesis. In addition, I would like to express my gratitude to a few people in particular for their help in making this thesis possible.

Paul Monticciolo, for his guidance in planning the thesis and overcoming difficulties along the way. He was invaluable as a source of knowledge and feedback, and helped me work out how to best present my plans and results.

Vivienne Sze, for her advice on planning and writing about my work. She made sure I was asking the right questions, and challenged me to organize and explain my work more clearly.

Ken Teitelbaum, for his help in ensuring I made progress and his advice on the many challenges I encountered.

Karen Gettings, for her assistance with collecting and understanding the existing SPEq work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Specialized Hardware

In recent years, extremely computation-heavy applications such as big data analytics and deep neural networks have created an increasing demand for fast and power-efficient processing methods. Low-power computation is also desirable for mobile and emerging IoT platforms. One approach to reducing power consumption and increasing throughput is to create specialized hardware implemented on a field programmable gate array (FPGA) or as an application-specific integrated circuit (ASIC). This strategy is particularly effective for many signal processing applications; parallel computations can be considerably accelerated by the use of dedicated hardware. ASICs typically show a very significant improvement in area, throughput, and power consumption over a general purpose processor-based implementation (or one in similarly non-dedicated hardware). In cases where a degree of reconfigurability is important, or if a quicker development cycle is desired and requirements are not as stringent, FPGAs represent a good middle ground. They are also often used to prototype a design that will eventually be optimized as an ASIC.

Creating dedicated hardware can improve speed and power by several orders of magnitude; however, the chip design is typically described using a specialized hardware description language (HDL), such as VHDL[1] or Verilog. These languages offer

[1]VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

sophisticated low-level control over the hardware, but require an engineer experienced in the language and months of development time to complete a verified design. Producing a fully-optimized ASIC design can take months to years while requiring several fabrication spins.

As the demand for fast, power-efficient computing grows, there is also a demand for quicker and more accessible workflows for producing hardware designs. This thesis proposal aims to demonstrate and evaluate one potential alternative to traditional HDL, namely the MathWorks suite of tools including HDL Coder for Simulink. These tools enable the user to produce an HDL description from a block-based representation in Simulink, offering several advantages. Simulink models are considerably more accessible than VHDL or Verilog code to engineers inexperienced in hardware. Additionally, it is easy to parameterize subsystems and re-use sections of blocks within a well-designed model. This thesis seeks to investigate the hardware potential of the MathWorks tools by realizing a modern signal processing application within Simulink and evaluating its performance on an FPGA in terms of on-chip area, throughput, and similar standard metrics.

1.2 Approach

1.2.1 Motivation

To better understand the MathWorks tools and create a utility for future use, this thesis will explore the implementation of two modules capable of performing types of nonlinear equalization (NLEQ). Nonlinear equalization is a technique for mitigating certain kinds of common distortions generated in analog and mixed-signal semiconductor devices. When data is recorded from an analog-to-digital converter (ADC), circuit imperfections in the chip can lead to nonlinear behavior. For example, ADC subsystems such as the sample-and-hold and quantizer circuits will result in harmonic and intermodulation distortion produced after processing a pure sinusoidal input signal. These distortions are exacerbated by the common practice of interleaving outputs from multiple low-rate ADCs to produce a high-rate output; any drift in the clocks for these ADCs can result in offset distortion which compounds the effects generated by circuit nonlinearities. As a result, undesirable spurious signals may make it difficult to detect small signals in the presence of large signals at other frequencies.

Spurious signals and nonlinearities are introduced in virtually all applications which require some form of data collection or signal processing. The range of potential uses for NLEQ systems is very broad. At the same time, it is a fairly representative signal processing algorithm. By using the MathWorks tools to create an NLEQ design, several benefits are realized. First, information is gained about the strengths and weaknesses of the tools; specifically, whether the overall performance of resulting VHDL is acceptable for some set of applications, and also whether and how performance varies between different types of common signal processing operations (including finite impulse response filtering, signal mixing, accumulation, and control circuitry). Second, if VHDL performance is found to be acceptable, a set of modular utilities have been created that can be re-purposed to provide NLEQ for a variety of systems in the future.

1.2.2 Pathfinding Model

Before beginning the implementation of an NLEQ design, a simpler baseline system was created with the goal of becoming familiar with the tools and best practice for the workflow. This model performs a Fast Fourier Transform (FFT) on multiple streams of interleaved input data. The workflow was validated successfully using this design, and various methods of structuring and testing the model were attempted and evaluated. The techniques determined to be most suitable formed the basis for the development of the subsequent NLEQ designs.

1.2.3 Comparison to Existing HDL Design

In order to compare VHDL produced using the MathWorks tools to the baseline of hand-optimized VHDL, a design that both exemplifies common signal processing techniques and has been implemented under reasonably stringent power, area, and throughput requirements is desirable. One such design is a type of nonlinear equalizer, the Sparse Polynomial Equalizer (SPEq), produced by Gettings et al[1] for a specific and demanding application.

NLEQ designs in general require many of the basic operations necessary for signal processing (chiefly: filtering, mixing, and accumulating), and the SPEq processor has the added feature of having an existing VHDL implementation which was available for use in the thesis. By re-implementing each modular part of the design using the MathWorks tools with the same overall architecture and comparing the generated VHDL to the existing implementations of SPEq in VHDL, this thesis investigates the ability of the tools to perform efficient hardware description. First, and most importantly, the results must be accurate; the implementation should successfully produce the same output as the reference VHDL, and ideally would do so with the same latency and otherwise be cycle-accurate to the original. Additionally, other aspects of performance are evaluated. By comparing the area, power consumption, and maximum possible throughput of the generated HDL code to that of the hand-optimized VHDL, information is obtained regarding the ability of the MathWorks tools to create efficient VHDL performing basic signal processing operations.

1.2.4 Creating a Parameterizable and Modular Design

In addition to the performance of VHDL created to match an existing design as closely as possible, this thesis examines the efficacy of the MathWorks tools for producing a modular, parameterizable, and understandable design from the ground up. A hardware implementation of polyphase nonlinear equalization (pNLEQ), a more generalizable NLEQ algorithm, described by Goodman et al[2], was created using the tools. The major design goal was to use parameterizable elements wherever possible, allowing compatibility with the maximum range of systems, but to retain a reasonably high clock rate (250 MHz target, thus allowing for 1 GS/s with four interleaved channels). This implementation is analyzed to yield information about the practicality of using the MathWorks tools and workflow to design new modular systems.

1.3 Structure of Thesis

Chapter 2 provides background for the MathWorks tools and each type of nonlinear equalization referenced and implemented in this thesis. Chapter 3 describes the pathfinding FFT model and what was learned from designing and iterating upon the system. Chapter 4 gives a more detailed description of the structure and design of the SPEq processor, and the information that was gained from an evaluation of the re-implementation using the MathWorks tools. Chapter 5 describes in detail the implementation of a polyphase nonlinear equalizer using the tools, as well as the design choices that were made, the performance of the final product, and what may be learned about the tools from the design of this system. Chapter 6 summarizes the results found during this thesis regarding the performance of the MathWorks tools in producing hardware descriptions, including the trade-offs involved with using the tools, and the designs produced that may be useful in the future.

The contributions made by this thesis include:

1. The production of a simple signal analysis module for on-chip processing and validation.

2. The production of a near-identical implementation of each module in the SPEq processor using the MathWorks tools for comparison to the original in hand-optimized VHDL.

3. The production of a parameterizable and generalizable polyphase nonlinear equalizer using the MathWorks tools for overall evaluation.

4. A description of the benefits and challenges involved with using the MathWorks tools, and recommendations for best practices and possible improvements to the workflow itself.

Chapter 2

Background

2.1 MathWorks Tools

2.1.1 Description of Workflow

This thesis will involve heavy use of several tools produced by The MathWorks, Inc., primarily Simulink, HDL Coder, Embedded Coder, and plugins such as the Digital Signal Processing Toolbox. All of this software is part of the MATLAB ecosystem.

Simulink[1] is a block-based programming language that supports simulation and verification of designs. Models are composed of fundamental blocks provided in the standard library as well as blocks from toolboxes. Modular design is supported; parts of designs can be designated as subsystems, and subsystems can be reused as part of a custom library. Subsystems may also be masked (parameterized); mask parameters can be input to lower-levels subsystems or fundamental blocks. Additionally, dynamic initialization code can be added to a subsystem. This code will be run if the subsystem's parameters are changed, and can modify block parameters or even add and remove blocks and signal wires.

HDL Coder[2] produces HDL code from a Simulink model or MATLAB code, and

[1]See documentation on the MathWorks website at: `https://www.mathworks.com/help/simulink/index.html`

[2]See documentation on the MathWorks website at: `https://www.mathworks.com/help/simulink/hdl-coder.html`

offers several verification options as well. Embedded Coder performs a similar function for embedded code, and the two products can be used together for hardware-software co-design. For example, when using a System-on-Chip (SoC) integrated FPGA and ARM processor core, a subsystem may be selected to be converted to HDL and run on the FPGA; the remainder of the model will be converted to embedded code and run on the processor. This can allow for easier subsystem verification and data transfer back to the host computer.

When generating HDL code from a Simulink model, a variety of configuration options are available to the user. Either Verilog or VHDL code may be generated, and it is possible to set optimization flags to prioritize area, clock speed, and the like. Individual blocks from the HDL Coder library often also have such options, and sometimes allow the user to sacrifice numerical integrity for additional speed. The tool will attempt to pipeline intelligently such that all signal wires have matched delays and the throughput will be maximized (subject to user-specified priorities). However, as will be discussed in Chapters 4 and 5, pipeline registers will not generally be added within library blocks, and so it is often necessary to manually pipeline parts of the design or even create pipelined versions of existing blocks in order to meet demanding timing constraints. In general, the tool automates many parts of hardware design, but the structure of a model must still be created with an eye towards hardware implementation. Determining which portions of the design process are well-automated is the main focus of this thesis.

2.1.2 Prior Investigation

Some prior work has been done in evaluating HDL Coder and Simulink. Hai et al[3] demonstrated the use of HDL Coder for a video processing application (edge feature extraction), while Besbes et al[4] did the same for a discrete time high gain observer of various properties of an induction motor. More recently, Versen et al[5] benchmarked input and output interfaces on a Zedboard with Simulink, and Shah et al[6] implemented a mobile/low-power communication protocol (MIPI Low Latency Interface, specifically the Data Link Layer) in both hand-written Verilog and Simulink

HDL Coder, comparing the performance of the two.

This thesis aims to expand on previous work by providing an updated evaluation of a more recent iteration (2017a) of the tools, and by implementing a complex, high-throughput signal processing algorithm. The approach to evaluation is similar to the work of Shah et al, but provides new information on the performance of the tools with regard to an application that is primarily based on filtering and manipulating signals. Demonstrating this kind of application further tests the ability of HDL Coder and Simulink to produce sufficiently optimized hardware description for use in practical, real-time systems. Additionally, data is gathered on the tradeoffs involved with implementing signal processing techniques in HDL Coder.

2.2 Nonlinear Equalization

2.2.1 Metrics and Performance

The level of distortion in an ADC's measurements can be expressed as spurious-free dynamic range (SFDR) or intermodulation-free dynamic range (IFDR)[7]. These metrics represent the difference in magnitude (in frequency domain) between the largest signal in the data and the smallest signal that can be distinguished from the distortion resulting from that largest signal. Many signal processing and data gathering applications benefit from increased SFDR/IFDR, and thus those are the main metrics used in evaluating nonlinear equalizers. By modeling and then reducing the aforementioned types of distortions, nonlinear equalization improves SFDR, which is the main focus of this thesis. Ideally, equalization will suppress distortions to the noise floor of the signal, thereby increasing the usable dynamic range for analysis.

2.2.2 SPEq

The Sparse Polynomial Equalizer (SPEq), designed by Gettings et al[1], is a hardware architecture created for the purpose of performing nonlinear equalization under low-power requirements. As a CMOS 65 nm chip, the architecture improved the SFDR of the input signal by a median of 25 dB and consumed under 12 mW of power when clocked at 200 MHz.

The SPEq architecture will be more fully described in Chapter 4, but in short, the design is composed of 30 processing elements (PEs), dataflow and preprocessing logic, and an accumulator to sum the input signal and the outputs of all active PEs. The NLEQ filters representable by SPEq are somewhat limited, as not all higher-order filters may be used, but are sufficient for the intended usage.

This nonlinear equalizer is unique in that, rather than being targeted for general-purpose applications, it was codesigned with a particular anti-alias filter, described in Kim et al[8]. The system composed of this filter and the SPEq chip was designed to have a very high SFDR. Many applications of interest, including those on mobile platforms, have significant size, weight, and power (SWaP) constraints, and thus apply additional restrictions on the system design. Typical solutions for anti-alias filters such as large LC filters will not easily meet these constraints. As a result, an anti-alias filter with lower linearity was devised. The SPEq processor was designed to improve the SFDR of the system, compensating for this trade-off.

The SPEq design also benefits from being created for such a specific purpose. Lowered generalizability allows for improvements in area requirements and power consumption, making the system more suitable for the aforementioned SWaP-constrained applications. This also makes it a good choice for analysis in this thesis, to provide a comparison between the performance of hand-optimized VHDL and the output of the MathWorks tools.

2.2.3 pNLEQ

A more general design will be produced using the MathWorks tools implementing polyphase nonlinear equalization (pNLEQ), a fairly recent semi-blind approach that yields an average improvement of 20 dB in SFDR and which is computed using a series of polynomial filters.

Goodman et al. published pNLEQ in a 2009 paper[2]. This thesis will not fully describe it, but a brief summary follows. The technique is trained by recording a combination of pure sinusoidal tones (i.e. a training sequence) processed by the ADC under test. The training approach is semi-blind, or aware of the number of tones in the signal (typically one to three), but not the exact frequencies. A user-defined number of nonlinearities are selected as best describing the distortions present in the signal after the ADC, and these nonlinearities determine corresponding filter coefficients. More specifically, each nonlinear filter involves applying an FIR filter to the input, then mixing the resulting signal with that same input delayed by a number of cycles (full details on how this is done may be found in Chapter 5, with Figures 5-1 and 5-2 being particularly relevant). These filters are applied to the signal, producing an estimate of the distortion, then subtracted from the original signal to produce an output with improved SFDR/IFDR.

One characteristic that makes pNLEQ suitable for this thesis is its inherent need for flexibility. Many prior analyses of the MathWorks tools have not considered the common need for hardware systems to be reconfigurable and parameterizable, and evaluating an implementation of pNLEQ will address this gap. It is fairly representative of signal processing techniques in its low-level operations, and thus yields information about the suitability of the tools for realizing such techniques in hardware. Additionally, producing a high-throughput system that takes input from multiple interleaved ADCs requires processing multiple streams of data with stringent time requirements, challenging the limits of the tools, but is not so complex that it would be too difficult to realize in the span of this thesis. Most importantly, this design was produced independent of any hardware reference, and so its development yields information about the suitability of the tools for designing parameterizable and modular systems, as well as recommendations for best practices.

Chapter 3

Pathfinding FFT Design

3.1 Design and Implementation

Before beginning work on the NLEQ models themselves, some early work was done towards better understanding the current capability of the MathWorks tools. A simple signal analysis design was implemented in order to gain familiarity with the tools and prepare to create the main system. This model (block diagram in Figure 3-1) performs a Fast Fourier Transform (FFT) on interleaved streams of input data. Code was generated for, and run on, the Xilinx Zynq ZC706 Evaluation Board, and specifically on the Zynq-7000 system-on-chip (SoC) with an integrated FPGA and ARM processor.

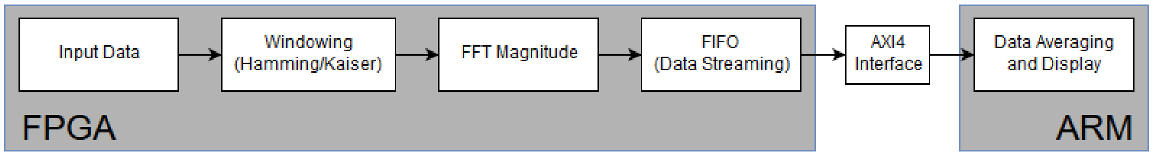


Figure 3-1: Block diagram of the signal analysis model.

Windowing (none, Hamming, or Kaiser, configurable by the user before generating HDL code) is applied to the input data, and the result is streamed into an FFT block, drawn from the built-in HDL Coder library. The result is reordered to account for index bit-reversal, then sent over the AXI4-Lite IP interface[1], to the ARM processor

[1] A Xilinx interface that supports bidirectional data transfer with a width of up to 32 bits.

and then to the host computer. This last data transfer step is by far the limiting factor; on the FPGA, the model produces FFT data results much more quickly than they can be sent to the processor.

3.2 Metrics and Results

Currently, the subsystem accepts as input four streams and outputs the frequency-domain analysis as four segments, lowest frequencies to highest, in parallel streams; however, this is configurable at the subsystem level. See Figure 3-2 for some sample outputs. After some optimization for four streams, data is processed at up to 1 GHz (250 MHz per stream). To test the system, data was generated on-chip and the results were packeted and sent back to the host computer for viewing.

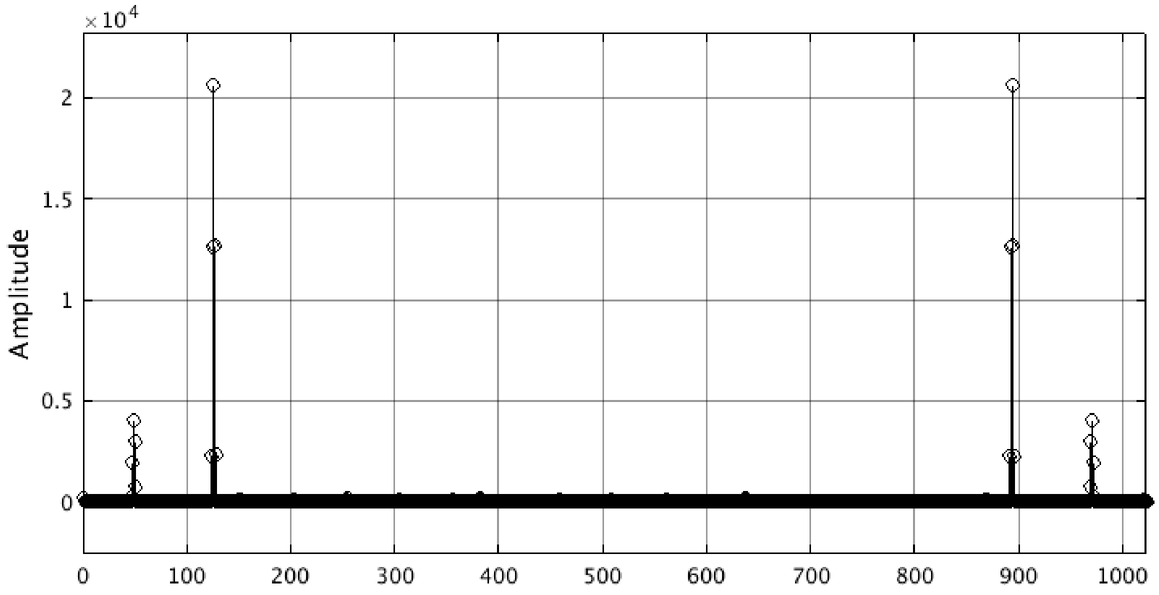


Figure 3-2: Data produced by FFT analysis of a two-tone signal.

It is worth noting that this design, as well as the designs discussed in the following chapters, are not restricted with regard to area. When the resources required to implement a design approach the available area of the chip, it is typically necessary to perform hand-adjustment. The performance of the tools under such circumstances is beyond the scope of this thesis; on a modern FPGA system such as the Zynq-7000 SoC, the designs discussed fall under 50% utilization in all categories. Rather, uti-

Resource	Utilization	Available	Percentage
LUT	12701	218600	5.81
LUTRAM	1273	70400	1.81
FF	17135	437200	3.92
BRAM	14	545	2.57
DSP	49	900	5.44
BUFG	3	32	9.38
MMCM	1	8	12.50

Table 3.1: Breakdown of utilization by the FFT signal analysis design on the Zynq-7000 FPGA.

lization is measured to compare the resource consumption of designs. The utilization of this model, which served as a pathfinding and baseline design, is shown in Table 3.1.

By creating this subsystem, valuable experience and familiarity with the tools was gained. In addition, these results serve as confirmation that all parts of the setup and workflow function correctly. Several different methods of testing the design on-chip were attempted, and the use of data transfer over AXI4-Lite for comparison and processing on the host computer was chosen as easiest for validation and comparison to simulation. Additionally, a few strategies for creating modular and re-usable blocks were used. Simulink allows for the parameterization of blocks via masking, in which programmer-defined parameters may be set by the end user when a block is instantiated in a design. These parameters may control block properties at a lower level, or even be used to dynamically add and remove blocks and connections, which will be referred to as dynamic initialization throughout this thesis. Thus, with some additional effort, it is possible to create fully-generalizable blocks for reuse in other projects or elsewhere in the model.

These techniques were carried forward and used to debug and test the NLEQ designs described in the next chapters. This model can also be useful going forward as a tool for on-chip analysis of other signal processing systems, including the NLEQ systems themselves.

Chapter 4

SPEq Design

4.1 Design and Implementation

4.1.1 High-Level Overview

The Sparse Polynomial Equalizer (SPEq) model is a re-implementation of the design described in Section 2.2.2, referenced from Gettings et al[1]. A high-level block diagram is shown in Figure 4-1. The first step, described in Section 4.1.2, is to ensure that input data is in two's complement format, converting as necessary. Next, as the design prioritizes reducing power and area, the most significant eight bits of the input are extracted and the remainder discarded. This truncated input is exponentiated to produce the square, cube, and fourth power of the input, as discussed in Section 4.1.3. Next, an array of processing elements (PEs) applies the set of required filter terms. The structure of each PE is given in Section 4.1.4. For each PE, a configurable sign-extended shift is applied, allowing for a greater dynamic range of coefficients. Finally, the outputs from each PE are summed together and added to the original input signal by the accumulator (Section 4.1.6). Essentially, the output of each PE is designed to mimic and thereby remove a portion of the undesirable distortions in the input signal. By subtracting out the distortions, an output signal with a higher spurious-free dynamic range may be obtained.

One common challenge that will be discussed throughout this section is the design decision of what data type to use within Simulink. For hardware synthesis, a fixed-point signal is typically desirable; in Simulink, it are defined as having a given number of bits and also by whether or not it is signed. Arithmetic library blocks such as addition and multiplication produce output values accounting for whether inputs are signed or unsigned, and also ensure that the output data type is the same as the input data type. In other words, when given signed input data, a library block will perform signed arithmetic and produce signed output data. Data in the SPEq Simulink model is typically treated as signed for this reason (because the value itself is intended to be signed), but in cases when bit operations are being applied, it is sometimes necessary to convert signals back to unsigned data. For example, when extracting bits from a signed value, the output is also treated as signed for comparison reasons, and a single bit cannot be extracted at all, so an unsigned conversion usually will be performed just prior to bit selection. In some sub-modules, this is handled by using primarily signed data but converting to unsigned just prior to bit extraction; in other sub-modules, an explicit sign bit is held and data is processed as unsigned instead, then converted back to signed at the ouptut. Additionally, fixed-point data may include a number of fractional bits. Data in this design is in most cases treated as an integer, as this makes it easier to precisely match the reference design, which uses the std_logic_vector type.

Additionally, many blocks are created using dynamic initialization, with which blocks are programmatically added, removed, configured, and connected. This is useful in cases where vector (multichannel) data needs to be processed in a certain way, but the built-in library blocks for this purpose are only capable of processing scalar data; with dynamic initialization, it's possible to simply replicate those blocks once for each channel. Likewise, in pipelined structures similar to the multipliers discussed in Section 4.1.5, similar processing circuitry must be created for each parallelization, and this was generally done using dynamic initialization. Pipelining in these types of blocks was performed manually and will be discussed in the following sections; additionally, virtually all blocks are pipelined to some degree by HDL Coder to satisfy the timing constraints corresponding to the desired data processing frequency.

Blocks that are referred to as parameterizable are configurable at HDL code generation time by the end user. These parameters set the bit widths of data signals. The function of the processor is also controllable by a set of input signals, which modify the type of data pre- and post-processing as well as the number and type of filters applied. Up to 30 PEs may be enabled (scan_pe_enable), each outputting the product of two signals generated from the input data and a coefficient (coe_in). An independently controlled delay is applied to both signals (dly_reg_sel and exp_reg_sel). The first, a simple passthrough of the input, may be delayed by between zero and seven cycles; the second, which may be any power of the input signal up to the fourth (controlled by sel_exp), is delayed by between 0 and 15 cycles. Another master delay is applied to the passthrough ADC signal to manage syncing and shift both delays as necessary (scan_adc_dly_sel). Dynamic range of coefficients and outputs is improved by a set of configurable shifts (scan_output_atten_sft and sft_control). Thus, a subset of polynomial filters on the input data, up to the fifth order, may be created for equalization.

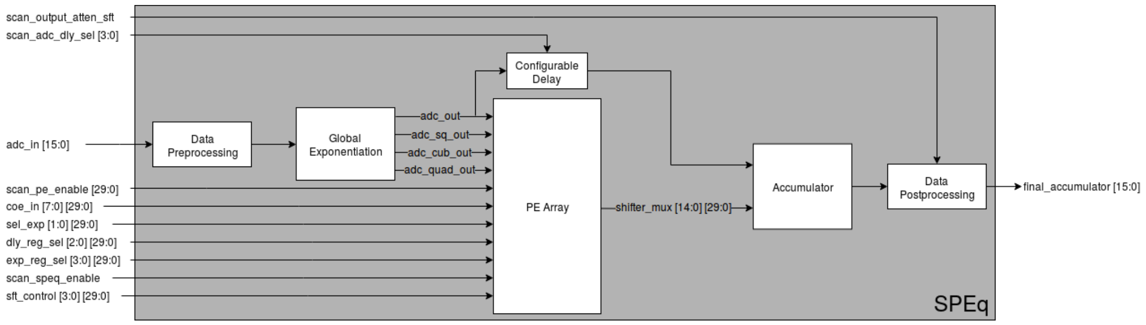


Figure 4-1: High-level block diagram of the SPEq processor design. Each block shown here is pipelined at the output and some additional registers have been added manually or by HDL Coder.

4.1.2 Data Preprocessing

Depending on the format of the input data, it may or may not be necessary to do several types of preprocessing. First, an input gain is applied by shifting the data left by between zero and eight bits (controlled by the input_gain_sft input). Next, if input data is given as unsigned values (biased so that all will be positive), the twos_sel_hi signal should be asserted. If so, the top bit of the input will be flipped to provide a simple conversion to two's complement. If twos_sel_hi is not asserted, the data is assumed to already be in two's complement format.

If bits of the input data are interleaved, preprocessing may also be controlled via the intrlv_sel_hi and intrlv_fall_first control inputs, which rearrange the bits of the data stream to convert it to a straightforward two's complement representation.

This module was fairly readily reproduced in Simulink; library blocks already exist for HDL Coder-friendly implementations of bit shifts and bit flips. Fixed-point data is accepted into the block without asserting whether its type is signed or unsigned; the output, however, is always signed.

4.1.3 Global Exponentiation

In order to allow various polynomial combinations of the input signal, a set of pipelined multipliers, described in Section 4.1.5, are used to produce powers ranging from two to four of that signal. These values are truncated to a parameterizable number of the most significant bits (in the original implementation and thus most testing, this value is set to eight bits) to reduce power consumption. Each multiplier operates on sign-extended two's complement data.

Rather than creating each power of the signal independently for each term of the polynomial filter, SPEq saves power by performing some of the exponentiation in this module. The input signal, and its powers produced here, will be used as inputs to each processing element.

When re-creating this block, the main challenge was creating the pipelined multipliers that perform the bulk of the computation. Input data with preprocessing is input to the block, then multiplied by itself repeatedly to produce a square, cube, and fourth power output, keeping only the top eight bits from each operation. Signed data was used throughout this block.

4.1.4 Processing Element

The array of PEs performs the actual signal filtering and equalization within the processor. Each PE accepts as input the outputs from the global exponentiation, including the input data signal, and its square, cube, and fourth power. In addition, a number of control signals are input, including the power to select for mixing, the delays to apply to the input signal and selected power, and the coefficient to apply to the product of the two before outputting. The bit size of the output is, like the bits received from global exponentiation, parameterizable at HDL code generation time. In testing, the output is set to fifteen bits.

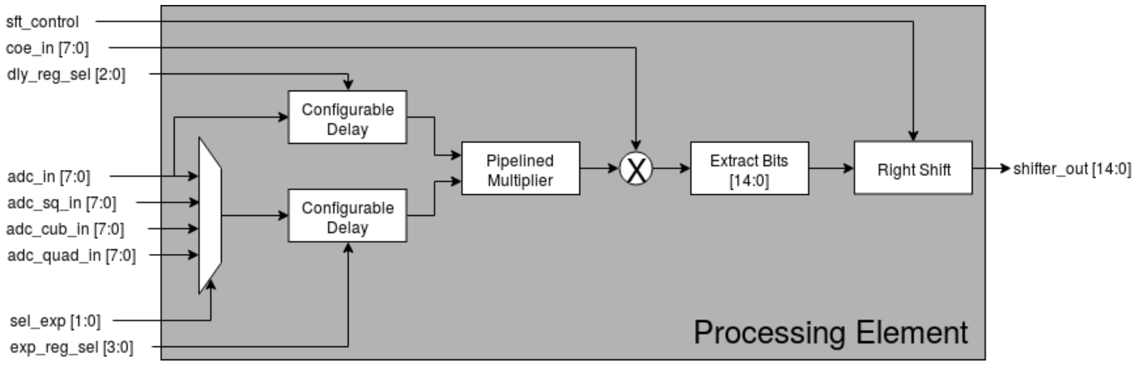


Figure 4-2: Block diagram of a single processing element within SPEq. Note that generation-time parameters here are set as eight bits to receive and multiply, and fifteen bits to accumulate (output from PE).

Subsequently, the output data from each PE may be shifted between zero and nine bits right to allow for selective scaling of the outputs of various filters and thus greater dynamic range of coefficients. This is managed by the sft_control signal, which is set independently for each PE.

4.1.5 Pipelined and Combinational Multipliers

Multiplication in the processor is performed either entirely in one cycle (combinatorial) or in a configurable number of cycles (pipelined). The first is quite simple to reproduce in Simulink, and was done by using the built-in product block. The second was more challenging; the built-in product block cannot be pipelined using a parameter, and to ensure cycle-accuracy with the reference VHDL it was desirable to manually recreate the division of calculation between cycles. This was done using dynamic initialization to create and connect the necessary blocks for pipelining, arithmetic, and signal routing.

Two inputs of the same bit size are accepted to the pipelined multiplier each cycle. One is split into a number of bit sections equal to the number of cycles of processing desired. In parallel, over the course of those cycles, each input section is multiplied by the entire other input and added to a running total. Each cycle, the running total that was just completed (with the final partial product being added) is output. This is conceptually similar to simply processing these multiplications in parallel each cycle, but makes timing easier to manage with regard to signal fanout and accumulation.

Due to the necessity of bit slicing throughout the process, the pipelined multiplier only operates directly on unsigned data. To process signed data, a wrapper module was created to take the absolute value of each input while recording the signs, then attach the appropriate sign to the resulting unsigned product before outputting.

The pipelined multiplier is parameterizable with regard to cycles to producing an output and bits to multiply. The combinational multiplier is also parameterizable with regard to bits.

4.1.6 Accumulator

The accumulator block receives filtered data from each PE as well as the passed-through input signal and sums those inputs in three stages, pipelining each partial sum. The output of this block is intended to be the input signal with distortions mimicked and removed by the filters. Control logic at the output ensures that the accumulator will saturate upon a positive or negative overflow rather than wrapping around.

This block is parameterizable with respect to the bit size of each input and the bit size of the output sum. The reference VHDL implementation of the block is also nominally configurable by the number of PEs, but in reality does not support beyond 30 PEs, and thus a similar approach was taken with the Simulink implementation to save time and complexity.

Resource	Utilization	Available	Percentage
LUT	8034	218600	3.68
LUTRAM	16	70400	0.02
FF	5830	437200	1.33
IO	6	362	1.66
BUFG	2	32	6.25
MMCM	1	8	12.50

Table 4.1: Breakdown of utilization by the SPEq VHDL reference design on the Zynq-7000 FPGA.

Most of the arithmetic operations in this block are performed on signed data. However, due to the necessity of bit selection in detecting and dealing with an overflow, those portions of the model operate on unsigned data instead.

4.1.7 Coefficient Scan Register

In the reference VHDL implementation, many configuration signals including filter coefficients, delays, and shifts were read in through a serial data stream and processed by a scan register module. Due to some complications in how this would need to be integrated with a test harness as well as time constraints, this module was not implemented using the Simulink workflow. Testing was done by directly passing in these configuration values instead, for both the reference VHDL and Simulink model versions.

4.2 Metrics and Results

To evaluate the Simulink workflow implementation of SPEq, both the reference VHDL and Simulink-based modules were synthesized and implemented with the same validation suite and setup. Testing was performed on the Zynq-7000 SoC (Xilinx Zynq ZC706 Evaluation Board). The ARM processor was not used; the equalizer was run and validated on the FPGA.

The reference VHDL successfully met timing with a 200 MHz clock. An estimated

Resource	Utilization	Available	Percentage
LUT	9730	218600	4.45
LUTRAM	18	70400	0.03
FF	6076	437200	1.39
IO	6	362	1.66
BUFG	2	32	6.25
MMCM	1	8	12.50

Table 4.2: Breakdown of utilization by the SPEq Simulink-based design on the Zynq-7000 FPGA.

0.817 W of power are consumed[1]. Utilization is described in Table 4.1. As expected, performance is poorer than the ASIC implementation described by Gettings et al. However, less power and area are consumed by this design than by the Simulink implementation. The Simulink design met timing with a 200 MHz clock as well, with similar negative and hold slacks. This version of the design consumes an estimated 0.899 W. Utilization is described in Table 4.2.

As expected, the Simulink model-based implementation performed somewhat worse than the reference VHDL in resource consumption. This supports the hypothesis that hand-optimization remains the best way to get a high-performance design. However, the difference is not significant, particularly when implemented for a modern FPGA. The most prominent categories of resource usage for both designs are Look-up Tables (LUTs) and Flip-Flops (FFs), with the reference VHDL design requiring 8034 LUTs and 5830 FFs while the Simulink-based design required 9730 LUTs and 6076 FFs. This is a difference of under a percent in the usage of total LUTs and under a tenth of a percent of FFs on the Zynq ZC706 FPGA, which is unlikely to be important on such a system.

Despite the relative insignificance of this difference, it yields information about what types of operations are well-optimized by HDL Coder. Upon comparing the utilization of LUTs by analogous modules in the two designs, it was discovered that although many modules showed an area overhead of roughly 20% in the Simulink

[1]This power figure, and the others presented, were simulated by the Xilinx tools. Annotated switching activity was not used, as this feature of simulation is not supported for VHDL.

model-based design (relative to 21.1% overall), a few blocks were significantly more or less demanding. The accumulator had less overhead at 293 LUTs compared to 288 in the reference VHDL design. This suggests that, in general, HDL Coder is less efficient at generating multipliers than adders, as the accumulator contains no multipliers, unlike most other modules. On the other hand, the blocks performing bit shifting within each PE were found to use many more LUTs in the Simulink-based design (a total of 1607 LUTs compared to 1189 LUTs in the reference design for a net overhead of 35.2%). This seems to reflect a particular inefficiency related to the manner in which Simulink interprets and HDL Coder generates this module. A four-bit unsigned value is input to the block indicating by how many bits the data should be shifted. However, the shift is only performed when the value given is between zero and nine; for all other values, the data value will simply be passed through. In the VHDL design, circuitry is only generated for shifts of zero to nine bits. On the other hand, likely due to limitations of the library block and the optimization performed by HDL Coder, the Simulink-based design generates logic for shifts of zero to fifteen bits (i.e. all shifts that could be specified with a four-bit value). It is possible that this utilization could be reduced by structuring the block differently or by creating a custom shift block; however, since the issue was identified late in development, further investigation will be left for future work.

Additionally, this model demonstrates that the signal manipulations performed in SPEq are fully replicable using the Simulink workflow; the two designs were tested and found to be cycle-accurate with the exception of resets. This discrepancy in reset handling is due to the manner in which reset pins are generated by HDL Coder. During generation from a Simulink model, one type of reset must be chosen to apply to all submodules generated at that time, and clock and enable pins are also automatically added. While the reference SPEq implementation mixed styles of resetting and enabling depending on the module, and sometimes grounded outputs for a certain duration after reset, this behavior proved difficult to replicate with Simulink. It is possible to do something of the sort by creating an manual reset signal, but this can become problematic in terms of fanout and timing, and is inherently limited by the cycle-based nature of Simulink; only synchronous resets may be implemented in this way. For many applications, this is not important (SPEq's function is essentially unharmed by changes to the nature of resets) and in fact the limitations on reset type may be useful for inexperienced users to avoid metastability issues and the like. A workaround exists in the form of generating HDL for submodules separately, then connecting them with a simple top level module in traditional HDL; however, this was outside the scope of this thesis and thus has not been thoroughly tested.

In summary, the results of this implementation demonstrate that a Simulink-based implementation of SPEq, while lagging somewhat in resource utilization, is capable of producing equivalent outputs to the reference implementation. This suggests that Simulink can perform most essential operations for signal processing applications, and supports its use as a potential alternative to traditional HDL in cases where best performance is not necessary.

Chapter 5

pNLEQ Design

5.1 Design and Implementation

5.1.1 High-Level Overview

This design is, at the highest level, describable as a configurable filter bank. As discussed in the background section, the pNLEQ technique works by mimicking, and then subtracting out, types of distortion applied to the input signal. The first filter is simply a passthrough and delay to retain the desirable linear portion of the signals; the other filters each re-create sections of the undesirable nonlinear distortions, and their outputs are subtracted from the output of the first filter to produce the cleaner output signal. A high-level block diagram is shown in Figure 5-1.

Within each filter are two major components: a finite impulse response (FIR) linear filter, described in Section 5.1.2, and a mixer, described in Section 5.1.3. In combination, a higher-order polynomial filter is applied. A block diagram representing this component is shown in Figure 5-2.

The purpose of creating this design was to test the suitability of the Simulink workflow for creating and testing a large modular design, and also to investigate the design choices that make a model easier to work with in the Simulink/HDL Coder ecosystem. Additionally, the final design is fully customizable with regard to the number of channels processed in parallel, bit width of input, and filter structure and

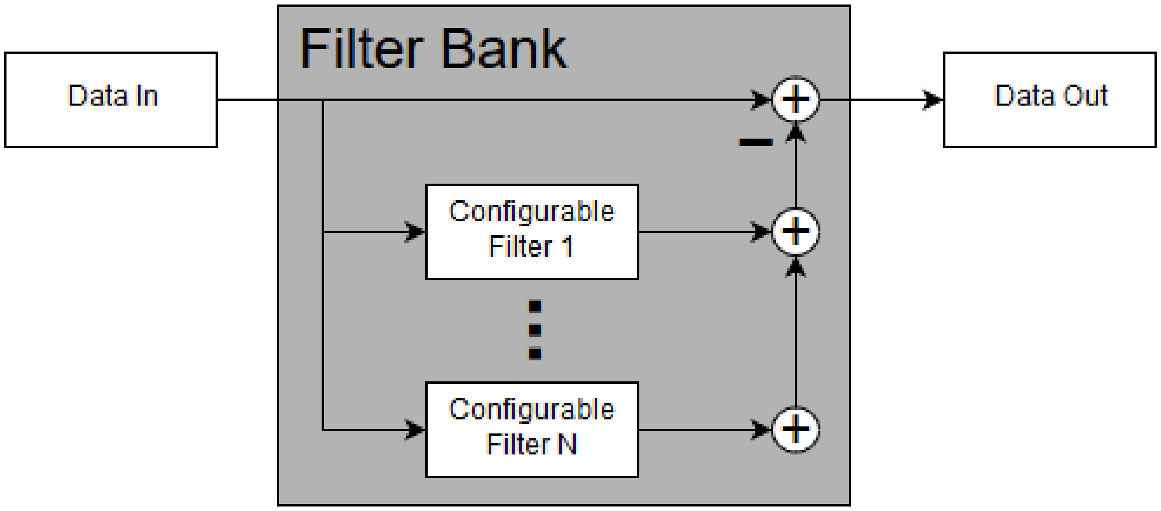


Figure 5-1: High-level block diagram of the pNLEQ model.

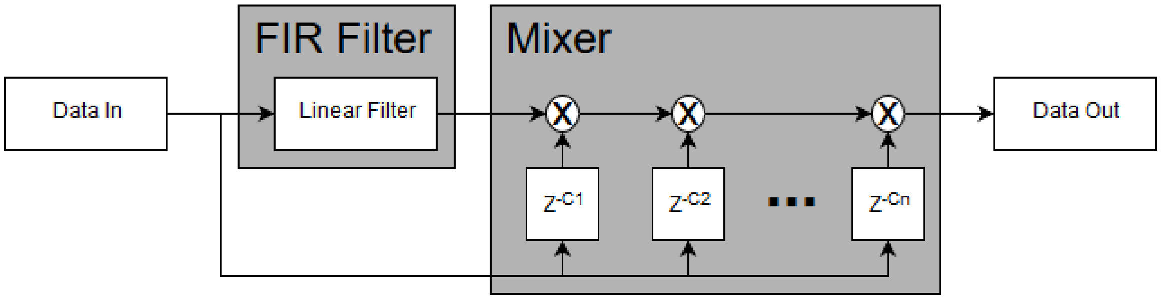


Figure 5-2: Block diagram of an individual filter. Number of multipliers and corresponding delays may be customized by the end user.

coefficients, meaning that it can be useful going forward as a utility for projects that need a nonlinear equalization module and are not operating under tight area, power, or throughput constraints such that they require a custom design. Testing was done with four parallel channels and a clock rate of 250 MHz (thus, a throughput of 1 GS/s); this overall throughput was chosen as challenging to implement in the tools, but not impossible under the scope of the thesis.

5.1.2 Linear FIR Filter

Although an FIR filter exists in the HDL Coder provided block library (and this block was indeed used in simulations of this design), a custom version was created for the greater control over pipelining necessary to allow the design to be synthesized for higher frequencies. This block was implemented using dynamic initialization.

Another feature added in this FIR filter is the ability to handle multichannel (interleaved) inputs. This is challenging because it requires delaying and re-combining inputs from each channel to contribute to the outputs. For example, if there are four input channels and three filter taps, two of the inputs from the previous cycle will need to be retained, as they contribute to the four outputs to be calculated. Multichannel inputs were represented by the Simulink vector type, and were arranged, multiplied, and summed as needed using builtin blocks.

This block also automatically sets bit precisions of data values throughout the calculation to improve output accuracy. Two parameters are set at the block level to control the fixed-point precision of the coefficients given; these parameters also indirectly control the bit precision of the output from the block. The precision values are set with the signal mixers in mind as well, and are meant to allow the entire filter bank to produce the most accurate result possible while still meeting timing. However, since precisions are set automatically, it is important to note that requesting very precise coefficients could cause very high data precisions to be used in certain multipliers and that the resulting generated code could fail to meet the necessary timing constraints. This did not occur naturally with the filters tested, and from the timing observed using realistic data and coefficients, it seems that issues are actually more likely to arise due to increases in the desired output precision. Thus, it is likely that these automatic precisions will work well for most use cases.

5.1.3 Signal Mixer

This module performs mixing of the filtered signal with the pre-filter signal after various delays. The number of multiplications to perform and corresponding delays are configurable using dynamic initialization; additional multiplications will increase area and latency, but shouldn't impact throughput. Theoretically, any polynomial filter may be implemented by configuring one or more filters as shown in Figure 5-2.

The challenges with implementing this module mostly involved managing pipelining to allow synthesis at 250 MHz when processing signals with a larger bit width.

When increasing precision, it became necessary to create a custom product block, similarly to the FIR filter issue mentioned in Section 5.1.2.

Balancing accuracy was also a challenge. As discussed in Section 5.1.2, the fixed-point precision of data values was managed more or less automatically to allow the outputs to be accurate almost to the final bit, while restricting bit widths as much as possible to reduce timing issues. Again, it is possible that this system will fail timing when given a large number of compound multiplies to perform, but this did not occur with any filter tested.

5.1.4 Staged Multiplier

An early version of the custom multiplier block was implemented as a simple staged multiply, shown in Figure 5-3. Each of two input values is split into two sub-values, each containing half the bits. Four partial products are created by multiplying each pair of sub-values, and are summed to produce the output. Breaking down the multiply in this manner allowed higher-precision products to be built from lower-precision ones.

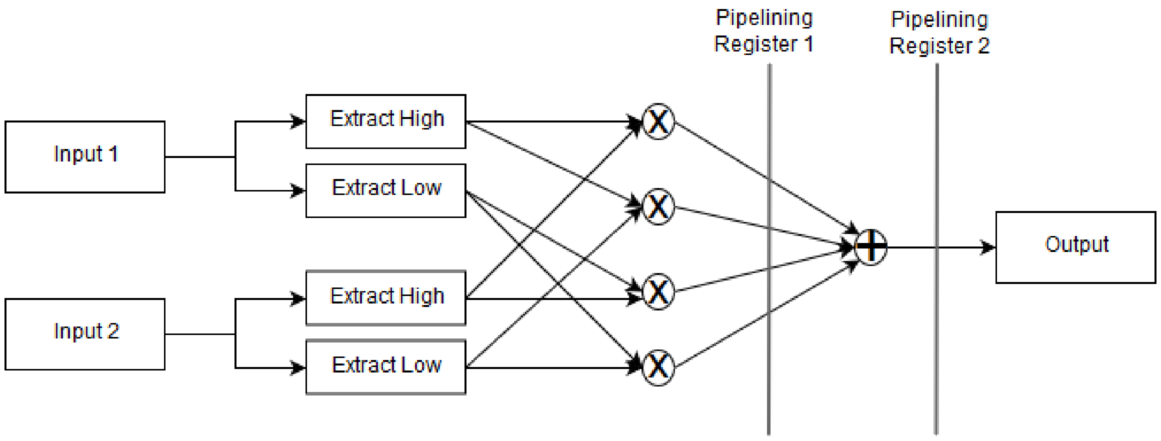


Figure 5-3: Block diagram of the initial staged multiplier.

Later in development, a more flexible custom multiplier was developed for the SPEq model. Using that configurable multiplier (described in Section 4.1.5) instead of this block could enable higher clock speeds, but would require some reworking of the data pipeline and was thus left as a potential future extension.

5.1.5 Vector Delay

To handle the vector data and also improve readability, a helper vector delay block was created. This block accepts vector data, presumably multichannel interleaved streams of one higher-frequency signal, and outputs multichannel vector data that would correspond to that higher-frequency signal delayed by a certain number of cycles. This involves a combination of direct delays of the signal and rearranging of channels, and is performed with dynamic block initialization.

5.2 Metrics and Results

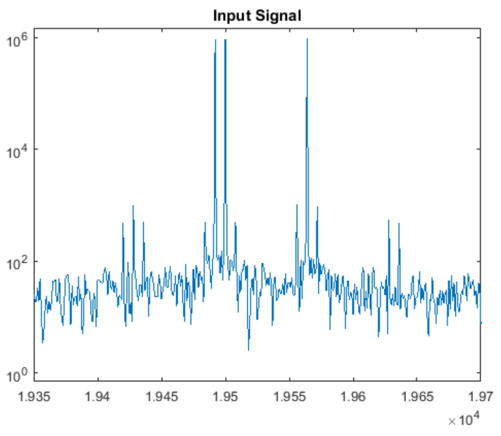
Unlike the SPEq-based design described in the previous section, this model was not directly based on any reference VHDL code; thus, there is no direct performance comparison to be made. Instead, this design was tuned with the intention of exploring the level of control allowed by the Simulink workflow with regard to timing optimizations.

The final design is capable of processing 20-bit signed data (8 bits integer, 12 bits fractional) in four parallel channels with a clock speed of 250 MHz to yield a maximum data rate of 1 GS/s. The model may freely be configured for additional channels at HDL code generation time. Timing is limited by the multipliers within the FIR filters and mixers; thus, substituting the more highly pipelined multipliers described in Section 4.1.5 could allow for improvements in data rate. Additionally, adding channels will typically not impact timing, and thus data rate could be improved by small integer factors at the cost of a corresponding increase in area usage.

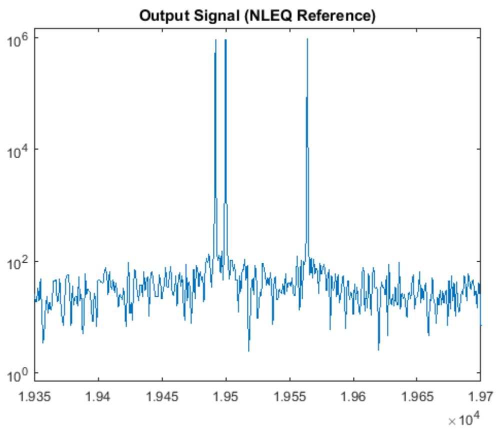
No noticeable loss of precision occurred in this hardware implementation. See Figures 5-4 and 5-5, where no performance difference can be seen. Using the same filters, both implementations successfully lower intermod peaks to the noise floor, improving spurious-free dynamic range by 16.54 dB (from 59.39 dB to 75.93 dB).

More specifically, differences between the outputs of the reference and hardware implementations had a mean of 1.0684e^{-4} and a standard deviation of 6.6166e^{-5}. The smallest representable difference in the hardware data format (with 12 fractional bits) falls at 2.4414e^{-4}, which is greater than two standard deviations above the mean of differences. Thus, the outputs are in the grand majority of cases within discretization error.

A less precise configuration of the design was also tested which processed and output 16-bit data. This output data is visually identical when graphed and improves SFDR equivalently (±0.01 dB). Differences between these outputs and the reference outputs had a mean of 1.5397e^{-3} and a standard deviation of 1.0717e^{-3}. The smallest representable difference of 16-bit data with 8 fractional bits is 3.9063e^{-3}, which is over a standard deviation above the mean. As expected, compared to the results from the 20-bit equalizer, it is clear that reducing the bit precision of the equalizer reduces the precision of the output. However, at least at 16 bits and above, this does not appear to significantly degrade the improvement in dynamic range for the data tested. It is worth noting that this result is not surprising; the test data used to produce these outputs was recorded using a 16-bit ADC. These results are more important as evidence that the hardware implementation is true to the reference software code than as a benchmark of precision.



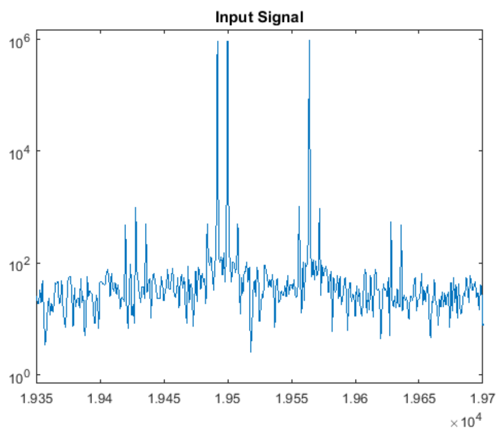
(a) Pre-equalization input data.



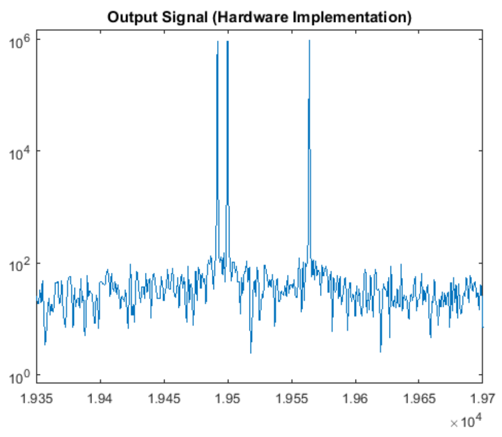
(b) MATLAB code output.

Figure 5-4: Pre- and post-equalization data for reference MATLAB software implementation of pNLEQ.

This system was prototyped and tested on the Xilinx Zynq ZC706 Evaluation Board (Zynq-7000 SoC specifically). The equalizer itself, in addition to other hardware portions including testing infrastructure, was hosted on the integrated FPGA; the test harness was run on the ARM processor. On this platform, an estimated 3.082



(a) Pre-equalization input data.



(b) Hardware implementation output.

Figure 5-5: Pre- and post-equalization data for hardware implementation of pNLEQ.

Resource	Utilization, 16 bit	Utilization, 20 bit	Available
LUT	24729	21536	218600
LUTRAM	790	883	70400
FF	28744	31835	437200
DSP	48	384	900
BUFG	5	5	32
MMCM	1	1	8

Table 5.1: Breakdown of utilization by the Simulink-described pNLEQ implementations, one producing 16 bits of output and the other 20 bits of output, on the Zynq-7000 FPGA.

W and 3.505 W of power are consumed by the block when configured to output 16-bit and 20-bit data respectively[1]. Utilization is described in Table 5.1.

The area and power consumption of this design are both significantly higher than those of the reference SPEq design as well as the Simulink implementation of SPEq. It is not easy to make any sort of clear comparison due to various differences between the implementations: the SPEq design is clocked at 200 MHz, while this design meets timing at 250 MHz; and the filters used to test this design somewhat correspond to 18 PEs (6 taps per filter and 3 filters meaning 18 independent terms) rather than 30 if used in SPEq, but some of those filters are not actually representable by SPEq's

[1]This power figure, similar to those in the previous chapter, were simulated by the Xilinx tools. Annotated switching activity was not used, as this feature of simulation is not supported for VHDL.

hardware.

Some of this discrepancy can be explained by the higher data rate of this implementation of NLEQ (enabled by the automatic pipelining performed by HDL Coder with no effort required of the user). SPEq produces one output per cycle at 200 MHz, thus yielding data at 200 MS/s. This pNLEQ implementation produces four outputs per cycle at 250 MHz, producing data at 1 GS/s, five times as quickly as SPEq. Certainly the higher clock rate contributes to the higher power, and parallel data stream processing is a culprit for higher area usage.

At the same time, the greater flexibility of the design likely also contributes to resource consumption. SPEq is capable of implementing a subset of polynomial filters that are adequate to equalize most distortions. However, the measures taken to save power, including global exponentiation, also restrict the kinds of filters that may be implemented. By contrast, this implementation of pNLEQ can implement any polynomial filter term desired (with the caveat that very high-order terms will likely cause timing and/or area issues). This design is also capable of producing a theoretically unlimited number of terms, while SPEq is limited to a maximum of 30 terms by the accumulator module. These changes make this implementation potentially more powerful, but also more energy- and area-intensive.

Unexpectedly, when comparing the utilizations of the two NLEQ designs, it was discovered that the 16-bit design was actually more resource-intensive in certain categories than the 20-bit design. For example, the 16-bit implementation uses 24729 LUTs, while the 20-bit implementation uses only 21536 LUTs. This difference is largely attributable to greater usage of DSP slices in the 20-bit design (384 slices, as compared to 48 slices in the 16-bit design), specifically within the FIR filters. It appears that the larger multipliers required by the higher-precision design allows for additional optimizations and mapping to DSP slices and thus a decrease in the utilization of LUTs.

This design and its implementation results demonstrate the potential utility of the Simulink workflow for HDL code generation. Generated code is not remarkably high-performance with regard to resource utilization or timing, but works correctly and is highly flexible. The Simulink workflow also lends itself well to the production of modular, readable, and parameterizable designs for signal processing. Block-based programming typically makes it easy to see the structure of the design at a high level, and also is much more readable than traditional HDL for an inexperienced user. Reusing and refactoring code is easier to do correctly because of the visible connections involved. Additionally, block parameterization support via masks and dynamic initialization allows for the creation of generalizable modules fairly intuitively.

Similarly to traditional HDL, a number of tools exist in the Simulink workflow for testing, validation, and debugging. Prior to HDL code generation, blocks may be tested using the built-in simulator. Any signal may be displayed via a scope block and/or logged to the MATLAB workspace for further analysis. Once HDL code is generated, it may be tested using the HDL toolchain as well; in this thesis, ModelSim was the primary simulator used for validating generated HDL.

For platforms with both an ARM processor and an FPGA, including the Zynq-7000 SoC, it is also possible to do testing in hardware using any existing test harnesses created for use with the Simulink simulator. Embedded code implementing test and communication infrastructure is run on the ARM processor, and sends those stimuli to the FPGA, which is programmed with HDL code implementing the device under test. The results received back from the FPGA are then sent back to the host computer as well, which displays them in Simulink. This technique is especially useful when a design has been confirmed to work in the Simulink simulator, as it can be used to debug a number of issues that can arise from cycle mismatches between a simulated block and its HDL implementation. (Built-in library blocks generally work out of the box, but occasionally the pipelining tool or simulator behavior can cause such an error to occur.)

In summary, this design shows that the Simulink workflow is capable of producing reasonably practical HDL code. Performance is likely not as high as a design optimized in traditional HDL, but this implementation is sufficient to be used in less-demanding applications and also supports a variety of potential filter configurations that it would be difficult to reproduce in an HDL implementation, made possible by parameters that must be set before HDL code generation. This design demonstrates one possible use case of the workflow as an alternative to traditional HDL: to create flexible and reusable hardware descriptions that perform well enough to be practical.

Chapter 6

Conclusion and Future Work

In this thesis, several hardware designs were created using the Simulink workflow, including a signal analysis block, an implementation of the sparse polynomial equalizer (SPEq) architecture, and a new implementation of the polyphase nonlinear equalization technique (pNLEQ). These designs were analyzed for performance on the Zynq ZC706 Evaluation Board (Zynq-7000 SoC).

6.1 Conclusion

This thesis investigated the viability of the Simulink/HDL Coder workflow as a potential alternative to more traditional methods of hardware description. Several designs were created and benchmarked for performance towards this end.

The pathfinding FFT signal analyzer served as an initial launching point to discover design principles that work well with the tools and confirm that the setup was functioning. This block can also serve as a utility going forward for testing other designs and visualizing their results.

The re-implementation of the sparse polynomial equalizer (SPEq) allowed for a direct comparison of two nearly cycle-accurate designs, one produced in traditional HDL and the other generated from a Simulink model. It was found that the Simulink-generated hardware fell behind in performance, but that there were very few parts of the architecture that could not be replicated. Resets and enables are automatically generated by the software and thus could not easily be made to precisely match the functionality of the reference VHDL design. However, this did not impact the ability of the design to produce correct outputs, and the Simulink-generated code otherwise behaved as desired.

The new polyphase nonlinear equalizer block was created to explore the potential of the workflow with regard to creating a reasonably well-performing, parameterized, and well-structured design from scratch. This design is somewhat more flexible than the SPEq architecture; parameters that are set prior to HDL code generation were used to alter the structure of processing elements in a way that is not possible in traditional hardware description, and an arbitrarily large number of processing elements can theoretically be created. On the other hand, the tradeoff for this increased flexibility is a greater utilization of hardware resources and the corresponding increase in power consumption. The module may also be useful going forward as an equalizer for use in or with future hardware designs.

By creating and analyzing these modules, this thesis explored the benefits and costs associated with using Simulink and HDL Coder as opposed to VHDL or Verilog to describe and synthesize hardware. The Simulink workflow was found to offer reasonable performance with greater ability to parameterize a design and easily visualize and re-use portions of an existing design. On the other hand, more traditional forms of hardware description can be more highly optimized and offer a greater degree of control over pipelining and reset and enable behavior. These features are useful to experienced users, but the automation and easier startup provided by Simulink and HDL Coder may appeal to users newer to hardware description.

In conclusion, the MathWorks tools seem likely to be a good alternative to VHDL and Verilog in situations where demanding power, area, or timing constraints are not present, and the user is interested in using a visual block-based programming language to define hardware. The types of parameterization offered by dynamic initialization may also appeal to more advanced users wishing to create generalizable hardware.

6.2 Future Work

While nonlinear equalization is fairly representative of the kinds of fundamental operations that make up many signal processing algorithms, there are also many other types of architectures and applications that were not investigated. This work could be extended to explore the ability of Simulink and HDL Coder to implement hardware of many different kinds. As one example, the designs created in this thesis are fairly light on control logic. One possible extension could be to investigate whether systems with more significant amounts of readable and well-performing control logic can readily be created with the MathWorks tools.

This thesis focused primarily on a workflow in which a block is defined in Simulink and then HDL code is generated for that block as a unit. It is also possible to generate HDL code for sub-blocks, then combine them in VHDL or Verilog, or alternatively to import existing HDL modules into Simulink for simulation or integration into a larger design. These alternative workflows can be useful when a user wants to take advantage of features of both VHDL/Verilog design and Simulink modeling; for example, if it was desirable to describe hardware primarily through Simulink, but also necessary to have more control over reset and enable behavior than is easily possible with Simulink and HDL Coder, a user might generate HDL separately for each sub-block and then create the necessary control logic in a VHDL or Verilog module. The possibilities and issues of this kind of approach were not investigated in this thesis, but form an interesting foundation for potential future work.

Another direction of investigation could involve comparisons between the Simulink/HDL Coder workflow and other high level synthesis (HLS) methods. HLS workflows represent additional alternative ways of producing hardware, and the relative strengths and weaknesses of different types of HLS are thus of interest.

Bibliography

[1] K. Gettings, A. Bolstad, S.S. Chen, M. Ericson, B. Miller, M. Vai, "Low Power Sparse Polynomial Equalizer (SPEQ) for Nonlinear Digital Compensation of an Active Anti-Alias Filter," *Signal Processing Systems, 2012 IEEE Workshop on* Oct 2012.

[2] J. Goodman, B. Miller, M. Herman, G. Raz, and J. Jackson, "Polyphase Nonlinear Equalization of Time-Interleaved Analog-to-Digital Converters," *IEEE J. Selected Topics in Signal Processing*, vol. 3, no. 3, pp. 362–373, June 2009.

[3] J.C.T. Hai, O.C. Pun, and T.W. Haw, "Accelerating video and image processing design for FPGA using HDL Coder and Simulink," *Sustainable Utilization And Development In Engineering and Technology, IEEE Conf. on*, Oct. 2015.

[4] M. Besbes, S.H. Saïd, and F. M'Sahli, "FPGA implementation of high gain observer for induction machine using Simulink HDL Coder," *Control, Engineering, and Information Technology, 3rd Int. Conf. on*, May 2015.

[5] M. Versen, S. Kipfelsberger, and F. Soekmen, "Model-Based Reference Design Projects with MathWorks' HDL Workflow Advisor for Custom-Specific Electronics with the Zedboard," *ANALOG 2016; 15. ITG/GMM-Symposium, Proc. of*, Sept. 2016.

[6] R.K. Shah, T. Kumar, A. Fell, M. Dohadwala, and R. Malik, "Executable model based design methodology for fast prototyping of mobile network protocol: A case study on MIPI LLI," *Signal Processing and Integrated Networks, 4th Int. Conf. on*, Feb. 2017.

[7] D. Martinez, M. Vai, and R. Bond, *High Performance Embedded Computing Handbook: A Systems Perspective.* Boca Raton, FL: CRC, June 2008.

[8] H. Kim, M. Green, B. Miller, A. Bolstad, and D. Santiago, "An Active Filter Achieving 43.6dBm OIP_3," *Radio Frequency Integrated Circuits Symposium (RFIC)* June 2011.