A Scaleable FIR Filter Implementation Using 32-bit Floating-Point Complex Arithmetic on a FPGA Based Custom Computing Platform

by

Allison L. Walters

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

Approved:

Dr. Peter Athanas, Chair
Dr. Nathaniel J. Davis, IV
Dr. Mark T. Jones

January, 30 1998
Blacksburg, Virginia

Keywords: Reconfiguable Computing, FIR Filters, Digital Signal Processing

A Scaleable FIR Filter Implementation Using 32-bit Floating-Point Complex Arithmetic on a FPGA Based Custom Computing Platform

Allison L. Walters

Committee Chairman: Dr. Peter Athanas
The Bradley Department of Electrical Engineering

Abstract

This thesis presents a linear phase finite impulse response filter implementation developed on a custom computing platform called WILDFORCE. The work has been motivated by ways to off-load intensive computing tasks to hardware for indoor communications channel modeling. The design entails complex convolution filters with customized lengths that can support channel impulse response profiles generated by SIRCIM. The paper details the partitioning for a fully pipelined convolution algorithm onto field programmable gate arrays through VHDL synthesis. Using WILDFORCE, the filter can achieve calculations at 160 MFLOPs/s.

Dedicated to my wonderful parents, Robert and Oanh Walters, and my beautiful fiancée, Joyce.

Acknowledgements

I never would have come this far were it not for the encouragement of my family, the heckling of my friends, and the support of Dr. Athanas. I am extremely grateful for everything they have done for me to make this important goal in my life a reality.

My sincerest thanks go out to Dr. Peter Athanas for all the effort and support he has given me throughout my thesis work. All his technical insight and motivation through my work has made it a success. In addition, I appreciate the funding from the Center for Wireless Technology that they and Dr. Athanas provided me.

I would also like to express my graditude to Dr. Nathaniel Davis, IV who put up with me in several of his classes during my undergraduate and graduate years and then participated on my defense committee.

Many thanks go to Dr. Mark Jones for partaking on my defense committee and taking the time to read through my material in such short notice. I hope to share more ideas for numerical computations on re-configurable computing platforms with him in the future.

I cannot thank the following people enough for the technical support and camaraderie they have given me during my development on Splash 2 and WILDFORCE. Their help accelerated much of my work and made it less frustrating. Thanks to: Bradley Fross, Nabeel Shirazi, Jim Peterson, Mark Musgrove, and Dave Lee.

Without the help of Annapolis Micro Systems, I certainly would not have completed my thesis work. The use of their re-configurable computing products and tools were invaluable to my work. Everyone there made it an enjoyable environment to work in.

Most importantly, I would like to thank my family and fiancée who continuously encouraged and supported me. I appreciate all the patience they have given over the last five years.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Many signal processing tasks frequently necessitate an immense amount of floating-point or fixed-point calculations for real-time or near real-time speeds [20]. Traditional von Neumann architectures cannot provide the performance of special-purpose signal processing architectures using specialized data paths, optimized sequencing, and pipelining. Unfortunately, such systems forego much flexibility despite operating at sufficient speeds. Custom computing machines propose a middle ground that employs flexible, high-performance computing for new algorithms on existing hardware at real-time or near real-time operation.

Digital finite impulse response filtering introduces one of many computationally demanding signal processing tasks. Wireless indoor channel modeling can be represented by an FIR filter using complex arithmetic due to the magnitude and phase responses of the channel impulse characteristics [13,15]. This thesis presents an implementation of such a filter on a custom computing platform called WILDFORCE. Furthermore, custom 32-bit floating-point operators have been devised to support hosts using the IEEE 754 floating-point format. Shorter word formats studied by [21] possess unacceptable loss in precision and accuracy. The pipelined implementation of the multiplier and accumulator elements permit the design to achieve maximum throughput at allowable clock speeds. High-performance yields may still be reached through the use of CCMs without having to depend on application-specific hardware.

Although not new to the realm of programmable devices, field programmable gate arrays (FPGAs) are becoming increasingly popular for rapid prototyping of designs with the aid of software simulation and synthesis. Software synthesis tools translate high-level

language descriptions of the implementation into formats that may be loaded directly into the FPGAs. An increasing number of design changes through software synthesis becomes more cost effective than similar changes done for hardware prototypes. In addition, the implementation may be constructed on existing hardware to help further reduce the cost.

1.1 Application Development on WILDFORCE

The introduction of automated tools and progressively advanced configurable logic devices have facilitated the development environment for custom computing machines. Conventional methods of programmable logic design consists of gate-level designs and schematic capture using complex CAD tools. With current design technology, hardware description language (HDL) compilers and synthesis tools allow designers to make alterations at a higher, abstract level.

Figure 1 illustrates the application design process for WILDFORCE and similar CCMs that use HDLs as the primary implementation tool. To facilitate the design process, the application designer should always generate a solid problem definition as seen in the first step. The designer may begin verification using high-level models with C, MATLAB, or behavioral VHDL to ensure problem definition compliance. In addition, the results obtained from the high-level verification may be used to confirm the implementation during later stages in the design process.

The problem partitioning step divides the algorithm among the processing elements such that the partial computations contribute to the final result in some fashion. Partitioning generally requires consideration from three primary factors as described by [30]: time, area, and communication complexity. According to [31], time and area factors appear to be familiar problems and are discussed further in the high-level synthesis and silicon compiler literature. The *time* factor relates to the amount of desired computation per clock cycle and *area* describes the amount of reconfigurable resources allocated to a given computation, to the total available reconfigurable resources within each processor board, and within each of the total number of processing elements on the board [30]. *Communications complexity* involves careful consideration of how data paths between the

partitioned algorithm should be routed. Since all CCMs do not possess the exact same bus widths, bandwidths, and propagation delays, the designer needs to architect the application to fit the platform being used. For instance, despite Splash 2 and WILDFORCE having systolic array architectures and a 36-bit bus to interconnect the processing elements, memory bus widths differ and must be taken into consideration when using a 32-bit floating-point implementation.

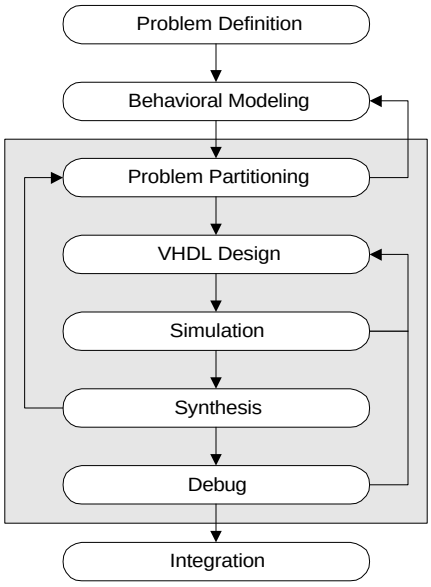

Figure 1. Application design process.

The high-level, partitioned design can be implemented using a variety of FPGA CAD tools, although high-level language synthesis with VHDL provides an advanced development environment for CCMs as well. Contemporary debugging tools exist for most environments that allow designers to step through the high-level code similar to other current high-level language integrated development environments. The VHDL models undergo simulation to provide the designer with a level of correctness before the synthesis stage. Actual propagation delays in the Xilinx FPGAs are highly sensitive to the outcome of

the place-and-route process and can have a disturbing effect on the application behavior [30]. Debugging tools in the development environment allow designers to counter such problems introduced by the limited functional coverage of simulators. As shown in Figure 1, several repetitions back to the coding stage of the design process may be necessary to achieve the desired results after synthesis.

The design approach for partitioned algorithms should be done in an incremental fashion to alleviate design and integration time. When integrating smaller, working components debugging can be facilitated knowing that each part has been functionally tested on an individual basis (when possible). Even though development environments such as with Splash 2 and WILDFORCE have been deemed as state-of-the-art, substantial amounts of time must still be invested to produce optimal, high-performance applications. According to [30], research efforts are underway to improve automation of the stages shaded in gray of Figure 1.

1.2 Task Definition

Earlier floating-point format development on Splash 2 by [21] introduced methods for shorter word formats and relied on the synthesis tools to produce necessary arithmetic units. Implementing 32-bit floating-point formats in the same manner generates functional components at the expense of vast CLB consumption. Furthermore, the use of smaller word lengths degrade either the range or precision of values to be represented. Fortunately due to recent advances in FPGA resource availability over its predecessors, higher density and faster FPGAs have made 32-bit floating-point designs become more feasible for higher bandwidth applications. A custom 32-bit floating-point format provides accurate filter realization and maintains host compatibility with a IEEE 754 format derivation. Chapter 7 provides an error analysis of the format used. Section 2.5 of the paper discusses alternative representations of 16- and 18-bit floating-point format advantages and disadvantages.

Mathematically, finite length sequence convolution is described by [2],

$$y_n = \sum_{k=-N}^{N} c_k u_{n-k} \tag{1.1}$$

where c_k represents a set of coefficients and u_{n-k} represents the unfiltered input sequence. This may be readily computed by hand or by software; however, the implementation on CCMs introduce another level of complexity called algorithm partitioning as described in Section 1.2. The objective behind this work is to implement convolution in a way that scales elegantly over several chips. The number of synthesized components that have to be designed depends on how well the design can be re-used. By examining replication, the design may take advantage of systolic array CCMs such as WILDFORCE and Splash 2 which provide the capability to expand the array across multiple boards. By seamlessly expanding the implementation across multiple boards, the size of the filter increases at the expense of an increased startup latency. Re-circulation techniques introduce another alternative to lessen the cost and at the same time, lengthen the filter for better frequency responses. Since FIR filters generally need to be of higher order than IIR filters, these methods become an important implementation issue [4].

1.3 Contributions

This thesis presents a scaleable FIR filter implemented on a CCM. The filter implementation runs at a maximum clock speed of 20 MHz. The latency of the filter depends on how many processing elements the filter uses in the array; the higher the filter order, the higher the latency. Filter coefficients can be loaded directly into the local memories of the processing elements, thus allowing the host to control the type of filtering being performed on the data dynamically. The CCM acts as a computing engine solely to perform filtering which could possibly be used for real-time filtering depending on the sampling rate. The design currently runs on a WILDFORCE four processing element array allowing up to eight filter taps per board. Implementing the filter on a CCM may benefit the end user by shortening the computation time on convolution in comparison to running it through software. By performing the computation in the time-domain, additional processing to perform an FFT, multiplying, and then converting back to the time-domain can be avoided. With the accelerated computations, sampled signal sources could provide the CCM with continuous data to be filtered.

Sections 2 through 6 of this thesis provides supporting background on filtering and CCMs as well as specific design criteria on how the filter maps onto the CCM being used. Chapter 2 covers all background material associated with filtering algorithms examined, CCM platforms to build the filter on, and different data representations possible. Chapter 3 examines the two different arithmetic logic unit designs used in each processing element. Chapter 4 covers the integration of the state machine to feed the ALUs and ultimately build the dual-tap processing element. Chapter 5 goes in depth on the data flow throughout the entire filter including the internals of each processing element. Chapter 6 discusses the results obtained from the synthesis and place-and-route tools, including area consumption and maximum estimated clock speeds. Coding techniques to produce ideal state machines and logic are presented as well. The results from various runs that verify the design on the CCM hardware contribute to the second part of Chapter 6. Unfiltered and filtered spectrum plots help to visually verify the design. The last chapter, Chapter 7, concludes the paper with the current work and possible future continuing work.

Chapter 2

Background

Digital communications is one of several areas that involve intensive signal processing, and one of the most important techniques found among the processing is digital filtering [5]. The channel model contributes to one of the many steps in a communication simulation in order to give the designer a perspective of how the signal propagates through different mediums [13]. Since channel models can simply be represented by a finite impulse response filter [10], further background material on this matter is included.

An important goal of this thesis is to show that such a model can be constructed on a general purpose custom computing platform which provides a flexible tool for simulation on hardware. In support of this work, some history and system architecture background on WILDFORCE and the Splash 2 system is given.

2.1 Digital Filters

Digital signal processing has been increasing in popularity due to the declining cost of general purpose computers and application specific hardware [5,7]. Since many telephony and data communications applications have been moving to digital, the need for digital filtering methods continue to grow [2]. Hence, simulation techniques to model these complex systems are needed as well. Software simulators offer flexible schemes to code the algorithm from a choice of many languages but cannot always offer the speed that a hardware simulator can. Unfortunately, building hardware prototypes to model different

systems can be costly and time consuming when constant changes have to be made. Therefore, a middle ground might be found using custom computing platforms or programmable logic. Such systems can offer similar flexibility as software and still retain some or all of the hardware acceleration [7] at the cost of a shorter implementation cycle.

2.1.1 Infinite Impulse Response Filters

Two commonly implemented filters in hardware include the finite impulse response filter (FIR) and the infinite impulse response filter (IIR), which may also take on the names non-recursive and recursive, respectively [2]. IIRs not only use the data values that pass through but also use other values of the output, which can be described by the following equation [2]:

$$y_n = \sum_{k=-\infty}^{\infty} c_k u_{n-k} + \sum_{k=-\infty}^{\infty} d_k y_{n-k} \tag{2.1}$$

where c and d represent the IIR coefficients and u the input data. The recursive system can also be described in a Direct Form II structure as shown in Figure 2 below. In order to use these types of filters, future values beyond the current y_n are necessary and is termed a causal filter [6]. Since simulations may possibly have the data to be filtered stored on some non-volatile media, non-recursive filters may be ideal [2]. Unfortunately, the side effect of being unstable, depending on the characteristic transform function, may limit its applicability [12]. Most importantly here, IIR implementation is not as easily realizable as that of the FIR even though IIRs typically require a lower order filter to accomplish the same function. But IIRs are preferred due to fewer parameters, less memory requirements, and lower computational complexity [6]. The non-linear phase characteristic generated by IIRs can be a severe drawback. Some transform techniques, such as the impulse invariance technique, does not have a direct filter frequency interpretation and may give unwanted effects such as aliasing and other phase problems -- more so than FIR filters [5].

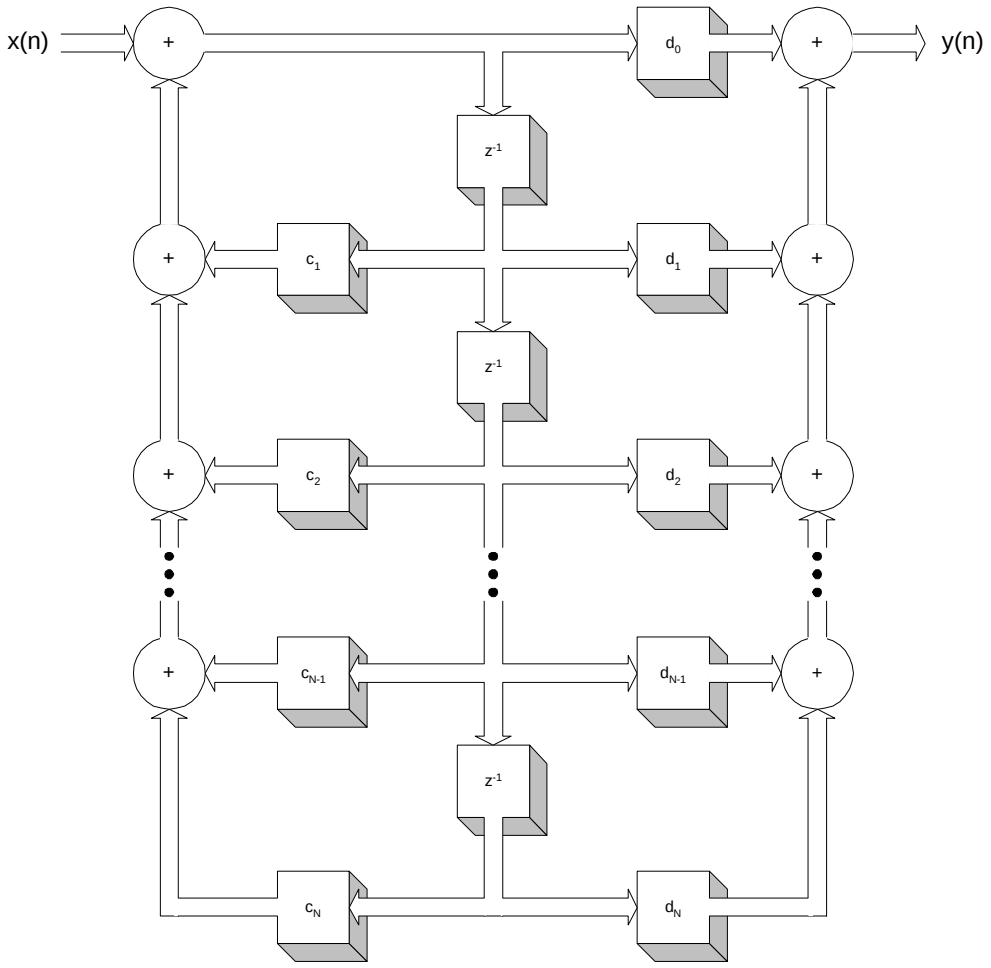

Figure 2: Direct Form II realization signal flow graph of an IIR.

2.1.2 Finite Impulse Response Filters

FIRs have the advantage of being much more realizable in hardware [12] because they avoid division and feedback paths. Despite needing twice the filter order of an IIR, FIRs are dependent on the data coming through the filter and on past values rather than future values like the IIR. Essentially, Equation 1.1 is a 1-D convolution between the filter coefficients and the input data. In performing convolution, one of the two sets of numbers is reversed and "slid past" the other. The resulting stream of numbers is found by taking

the sum of the multiplications at each sliding interval. FIRs can be graphically represented by a Direct Form realization as shown in Figure 3 [6].

Like the IIR structure, the FIR realization can be highly replicatable, which becomes important in the hardware design. One important aspect of FIRs is the linear phase characteristic, which makes it ideal for most digital signal processing applications [6]. Non-recursive filters are always stable unlike the recursive or IIR filters which have to keep the pole placements in perspective. Again, FIRs have to have twice the order of an IIR because they cannot achieve the smaller side lobes in the stopband of the frequency

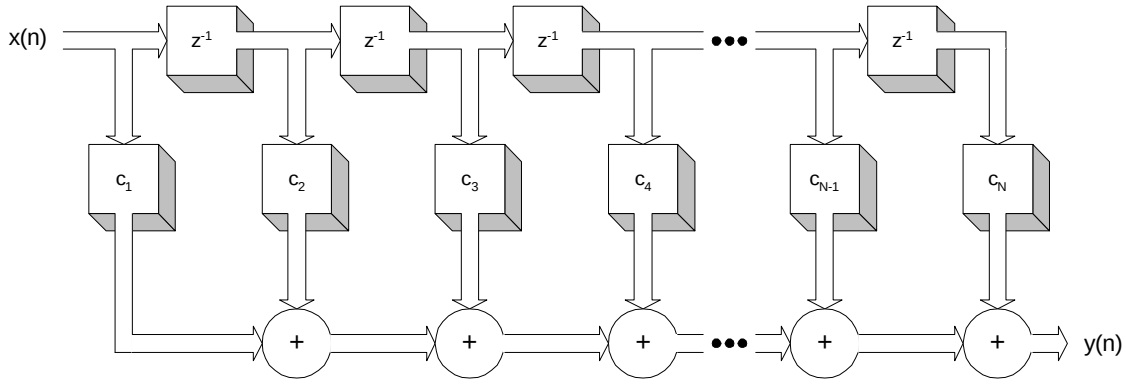

Figure 3: Direct Form realization signal flow graph of an FIR.

response given the same number of parameters as an IIR [6]. To help shape the frequency selective band of the FIR, "windowing functions" are convolved with the filter function. Examples of these functions include the Bartlett (rectangular), Blackman, and Hamming windows. The windowing technique tends to give the frequency response a sharper cutoff and "flatter" response in the passband [4,6]. Typically, a window with a taper and gradual roll off to zero produces less ringing in the sidelobes and lessens the oscillations in both the passband and stopband. The oscillations commonly found in the frequency response are due to Gibbs phenomenon, which is due to abrupt truncations of the Fourier series representation of the frequency response. Unfortunately, when correcting excess ringing in

the sidelobes, the window is widened, which means an increase in the width of the transition band of the filter, or a higher order filter [6]. Despite the higher order of the FIR filter, the implementation is feasible in hardware and possesses the necessary linear phase property needed by channel models [10,13].

Filter properities, design criteria, and the application at hand determine from which filter to chose. In this case, the channel model output is the convolution of the input signal and the impulse response which characterizes the channel. Due to propagation and additional phase shifts, the linear system can be represented as a complex impulse response [13]. Therefore, an FIR with linear phase properties and complex coefficients fulfills the requirements needed to build the channel model.

2.2 Communications Channel Models

Wireless indoor radio channel modeling has been a motivating, potential application for FIR filtering with complex arithmetic. Channel modeling becomes important when the designer must predict the minimum amount of power to transmit a signal through a medium within a specified area [13]. In mobile communications, analysis of the channel may help the designer in frequency re-use techniques or band-sharing schemes. Frequency re-use schemes to obtain high spectrum efficiency is a common way to re-allocate channels in an available spectrum, and therefore, careful consideration and study on interference makes channel models an important part of communication simulations [13]. The channel serves as the link between the transmitter and receiver ends as shown in Figure 4 below [14]. Typically, the channel model includes means to simulate other additive signal features such as noise, distortion, fading effects, and interference [13]. Some or all of these features tend to make channel modeling a very computationally intensive task, involving convolution filters for the signal processing.

For indoor radio channel models, a channel can be based on a statistical impulse response model as presented in [10]. Much of the statistical model is derived from [11], but does not take the effects of path loss as a function of transmitter-receiver separation into account, which is necessary if the power levels must be known [10]. In order to produce

the power delay profiles for the channel model, a program called SIRCIM [15] can be used. The necessary magnitude and phase values from the program are used as the impulse response model coefficients.

The convolution of input signals with these coefficients enables a designer to examine multiple access schemes, coding, diversity techniques, co-channel interference

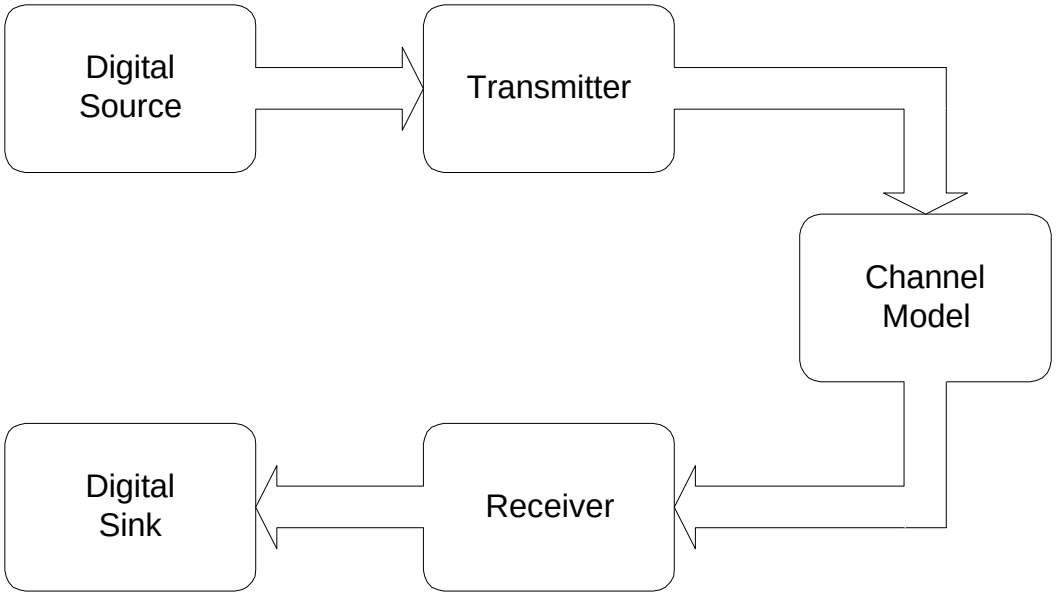

Figure 4: Simplified block diagram of communications system.

detection algorithms, and suitable physical layouts for high data rate factory and open plan office building radio communication systems [10]. Since the channel model can be represented by an impulse response, a linear filter, or FIR filter can be used to convolve the input signal and the channel's coefficients. The coefficients represent a complex baseband to model multi-path channels and can be described in the following equation [10]:

$$h_b(t) = \sum_k a_k e^{-j\theta_k} \delta(t - \tau_k) \tag{2.2}$$

where represents a real voltage attenuation factor, the exponential term represents a linear phase shift due to propagation and additional phase shifts induced by reflection coefficients

of scatterers, and τ_k is the time delay of the k[th] path in the channel with respect to the arrival of the first arriving component [10]. To find the necessary coefficients for the FIR, the power impulse responses given by the equation below [10],

$$|h_b(t)|^2 = \sum_k a_k^2 p^2(t - \tau_k) \tag{2.3}$$

are quantized into groups having temporal widths of 7.8ns. To get the estimated power impulse response values for discrete excess time delay T_K, the equation becomes [10]

$$|h_b(t)|^2 = \sum_K A_K^2 \delta(t - T_K) \tag{2.4}$$

where A_K^2 is a measure of multi-path power. According to [10], an averaging technique is used due to the resolution of the oscilloscope used in the experiments to obtain the data. Also, since the time domain window of the scope was limited to 500 ns, a maximum of 64 resolvable discrete multi-path components can be found in integrals of 7.8 ns [10]. Therefore, the maximum filter length needed for the complex channel model presented is limited to 64. Although this may not seem high for an FIR, analysis of the data described in [10] noted that few components arrive at excess delays greater than 500 ns. The profiles for responses taken at $\lambda/4$ intervals on a 1 meter track indicate that fading occurs in individual multi-path components [10]. Hence, the channel profiles change in space which results in varying sets of coefficients needed for the modeling filter. The FIR filter needed for this channel model requires an array of coefficients for each time delay "tap" as well as complex number convolution.

The examination of indoor, or factory and open plan building, channel models is due to a communications simulator called BERSIM, which is able to use SIRCIM's generated channel impulse response data. BERSIM convolves the transmitter signal with the channel impulse response. At the output of the channel, the co-channel interference and/or Gaussian noise may be added to simulate either noise limited or interference-limited systems [16]. With these operations in mind, the possibility of off-loading the signal convolution of the channel model on a hardware platform might lessen the simulation time.

2.3 Field Programmable Gate Arrays

Custom computing platforms such as WILDFORCE and Splash 2 contain several field programmable gate arrays, or FPGAs, to provide reconfigurability without penalties such as hardware modifications or timely programming. Programming of FPGAs takes on the order of milliseconds through software configuration. Depending on the application size, different sized FPGAs give the designer the flexibility to increase or decrease the resources as needed.

This thesis bases the design work around Xilinx FPGAs. The FPGA architecture consists of columns and rows of configurable logic blocks (CLBs) surrounded by I/O cells. In order to connect signals between CLBs, routing resources and programmable interconnects lay between the logic blocks. The basic nature of FPGAs presents a general set of resources which allow the designer to configure the logic blocks, routing, and I/O cells for a tailored application that runs at the speed of hardware, yet can be easily modified as software. The designer builds the application using a structured, high level language such as VHDL.

2.4 Custom Computing Platforms

The following sections include two reconfigurable computing platforms with systolic array architectures to support pipelined algorithms. Early development on Splash 2 investigated shorter word formats such as in [21] and [22]. Further research with 32-bit floating-point exploited the newer technology of WILDFORCE.

2.4.1 VTSplash and Splash 2

Splash 2 was a preliminary working platform for early filter design stages. The Splash system can consist of between 17 and 272 FPGAs used for special purpose computing, which is accessed primarily through the Sun SPARCstation 2 host SBus. Each Field Programmable Gate Array (FPGA) processor is accompanied by 0.5Mbytes of fast,

static RAM. The crossbar provides a full interconnection network between each of sixteen processing elements on a single array board. The interface board on the Splash 2 system is connected to the SBus on the Sun host through an SBus adapter, and the interface board communicates with a possible 1-16 processor array boards through a Futurebus+ backplane running a custom protocol [3]. In order to develop applications for the boards, the designer can use software development environments such as ViewLogic or the Synopsys tools in conjunction with custom cell libraries designed specifically for Splash 2.

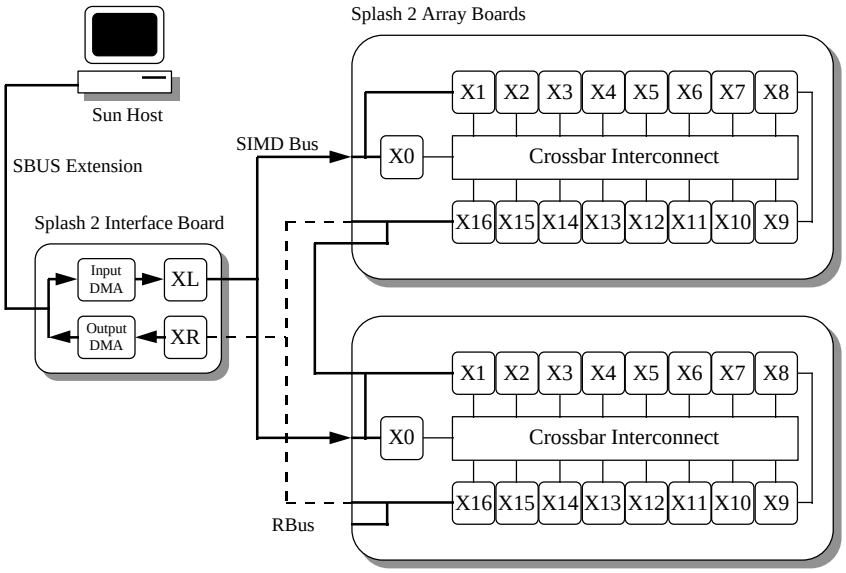

Figure 5: Two board Splash system.

2.4.2 WILDFORCE

Annapolis Micro Systems, Inc. constructs a similar commercial version of the Splash 2 computing engine called WILDFIRE. Over the past few years, a refinement process has brought about smaller versions based on the same architecture but with fewer processing elements, including WILDCHILD with eight FPGAs, WILDFORCE with five FPGAs, and WILD-ONE with two FPGAs. Note that each system does not include an additional control PE accounting for *n+1* PEs on the respective boards. With resources within an

FPGA increasing and intelligent tools becoming available, just a few processing elements can hold an entire application.

The following thesis builds the FIR filter on a WILDFORCE commercially available board which communicates with the host through a PCI bus. The use of PCI allows high bandwidth transfers using master mode DMA and burst mode block access. Currently, up to four boards can be linked together to form a larger computing engine which can share a mastered clock line between all four boards. A SIMD connector on each board allows for direct I/O between each board in addition to the PCI bus. Like, Splash 2 and all its other predecessors, the host controls much of the administrative work for the application such as programming the PEs, setting up DMA transfers, and clock control. Host interaction with the application does not have to be administrative but can also be vital when acting as an accelerator for some applications being run on the host in which data needs to be sent to and from the board.

2.4.2.1 The Host Interface

The application interfaces with WILDFORCE through a set of application programming interface (API) calls. Programming and application data do not have to go through multiple adapters such as those found in WILDFIRE and Splash 2 before reaching the board itself since WILDFORCE plugs straight into a host's PCI slot. With access to the bus and master mode DMA capabilities, large amounts of data can be moved between boards and possibly off the board with a properly designed external I/O connector card. The application designer uses C code in conjunction with the dynamic link library (DLL) to build a program which sets up the board clocks, programs the processing elements, and initializes any external memory for each PE, if necessary.

2.4.2.2 The WILDFORCE Architecture

The architecture and data flow of WILDFORCE still maintains much of the design features of Splash 2 and WILDFIRE but includes additional features, such as internal dual

port memory FIFOs, processing element FIFOs, mailbox capabilities, multiple master mode DMA, PCI burst transfers, and the ability to customize peripheral cards for each PE. The control processing element (CPE0), PE1, and PE4 each have input and output FIFOs which can either go to the host or the external I/O connector. To take advantage of PCI burst mode transfers and DMA, each dual port memory controller (DPMC) maintains an internal FIFO for memory accesses in addition to random memory accesses. Rather than

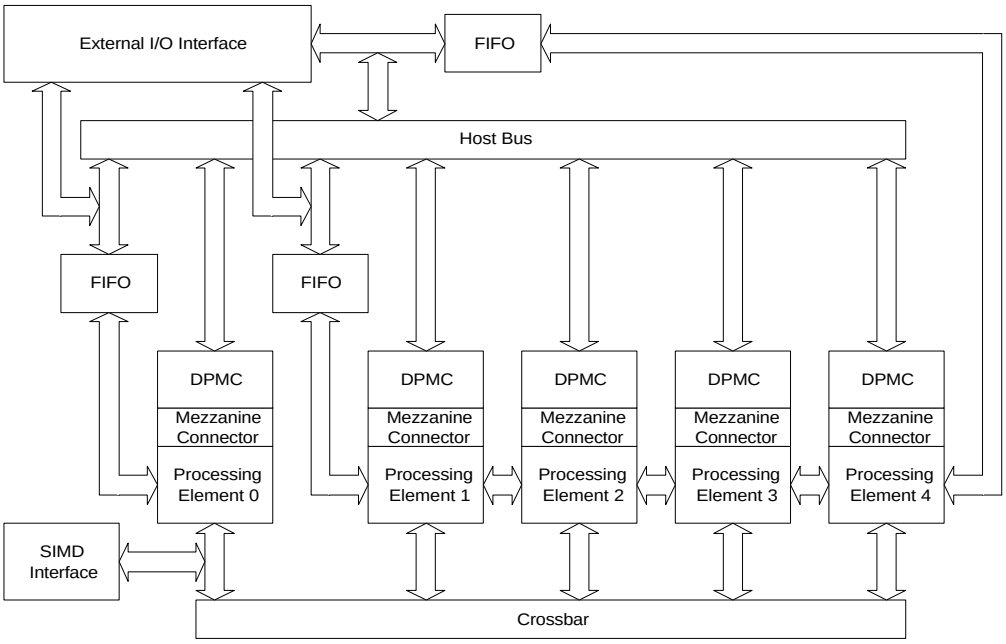

Figure 6: WILDFORCE system architecture.

continue asynchronous handshake lines to the host that need to be polled, mailbox capabilities on each PE with a FIFO (CPE0, PE1, and PE4) can send a single 32-bit word to/from the host using interrupt notification. In addition, each PE can interrupt the host individually.

Unlike the predecessors, WILDFORCE does not limit its local bus accesses to just memory, but instead has a general mezzanine connector interface. The interface allows

local memory to be given to the PE but custom boards other than memory can be attached using the connector, such as a co-processor, DSP, or even another Xilinx FPGA or ASIC. Currently, up to 4 Mbytes of static RAM can be accessed by each PE with the capability to go up to 256 Mbytes. In the near future, the ability to off-load floating-point can be done using a DSP or other co-processor. The newer line of WILDFIRE products, including WILDFORCE and WILD-ONE allow for 32-bit memory accesses rather than the 16-bit accesses found in its predecessors.

2.4.2.3 The WILDFORCE Programming Environment

The programming environment parallels that of the Splash 2. VHDL code in addition to C code is written to construct an application. For WILDFORCE, Model Technology, Inc.'s tools provide compilation and simulation support for application design. After the designer codes the algorithm in VHDL, the MTI environment first compiles the VHDL and then simulates the board model with the designer's application in the appropriate processing elements. Like Splash 2, a VHDL model of WILDFORCE allows the designer to accurately simulate the behavior of the board with the application embedded. Once the application logic has been verified, implementation follows. Synplicity's Synplify synthesis program builds the necessary image to be placed and routed for the FPGAs on the board. Formatted project files provide the program with all the information it needs to build the Xilinx netlist for the place-and-route tool. Xilinx provides a set of place-and-route tools which map the design onto the available FPGA resources (CLBs). Once the signals have been routed, the program creates the binary image that needs to be downloaded to the processing elements.

The host application, written in C, becomes an important part of the application. The host program controls board features not accessible by the PE, such as clock setup, interrupt handling, and DMA. A primary task of the host program includes setting up the environment on the board for the VHDL application. Once everything has been initialized, the application can then be downloaded to the board and started. Other tasks of the host program may include the following:

- data to/from the board either directly to memory or through the PE FIFOs
- interrupt processing
- post-processing of data
- setting up DMA transfers
- mailbox processing

The designer of the application needs to be well versed in not only VHDL, but C programming in order to take advantage of the host capabilities. Some processing, such as file I/O, simply cannot be done on such computing engines and therefore, relies on host interaction to aid in such circumstances.

2.5 Floating Point Representations on CCMs

Showing the feasibility of implementing floating-point on a custom computing machine is one of the primary focuses of this thesis. According to [19], many real-time hardware designs for signal processing applications use fixed-point formats due to size, cost, and speed of the available past hardware. Unfortunately, fixed-point does not offer the increased dynamic range, consistent precision, and normalization features of floating-point which are desirable among signal processing tasks. Recently, the arrival of sophisticated HDLs, like VHDL, and general purpose CCMs offer the ability to quickly prototype custom floating-point formats to suit the designer's application [20].

2.5.1 Custom Formats

Typically, general purpose machines employ 32-bit floating-point computations, providing more accuracy than signal processing applications generally need. But for real-time requirements, signal processing necessitates both speed and accuracy, a combination which general purpose machines find cumbersome to manage [19]. Hardware implementations make a trade-off on a reasonable decrease in precision for a smaller, more

manageable word size to enable floating-point processing. Commercial DSP chips, like Sharp's LH9124 and TRW's LSI chips, use a 24-bit and 22-bit format, respectively, but still retain enough accuracy and range to support most DSP applications. Therefore, even smaller floating-point formats have been investigated by [20] to enable DSP application prototypes and computation acceleration on CCMs. Another concern about using smaller formats involves the dynamic range capabilities. Having a large dynamic range helps to lessen the underflow and overflow problems fixed-point typically runs into [19]. Fixed-point formats try to accomplish a larger range by widening the format, but the hardware complexities and area consumption become obvious with such a large representation. Figure 7 [19], presents a graphical comparison of different formats and their possible ranges.

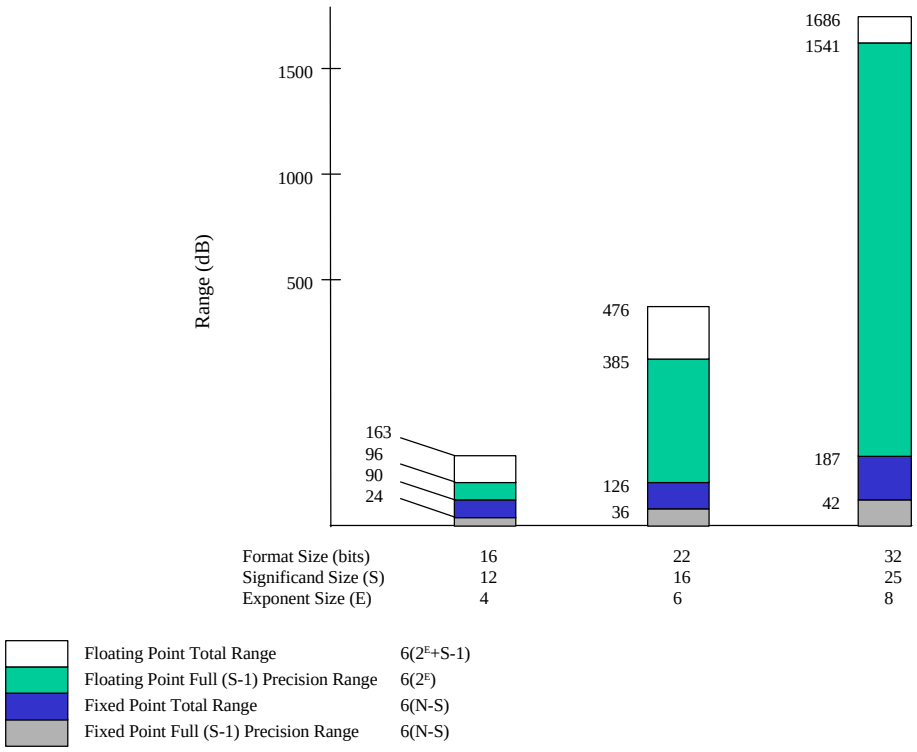

Figure 7: Floating-point format comparisons.

The algorithms presented in [20] try to compromise between the accuracy of a 32-bit format and the speed offered by hardware as well. Area consumption by floating-point operator units becomes the limiting factor on programmable hardware in parallel implementations. The design methodologies shown in [20] are based on HDL constructs which allow designers to tailor and prototype the floating-point format to meet the application requirements. Therefore, many designs depend on synthesis tools to handle the automatic mapping and logic generation. But having such flexibility on CCMs can benefit custom designs, such as the case with Shirazi's 18-bit 2-D Fast Fourier Transform [20]. The FFT could use an 18-bit format since Splash 2 uses a 36-bit wide data path, thus allowing two values per bus word. The ability to customize the format offers more freedom in the implementation rather than constraint around the data.

2.5.2 32-bit Floating-Point Format on FPGAs

Floating-point implementations on FPGAs present the challenge of mapping high resource demanding algorithms for optimal performance on a constrained platform. Not only do FPGA constraints come in the number of logic blocks, routing resources, and I/O capability, but the designer relies heavily on tools to map the implementation onto the chip. For a single stage, combinational logic multiplier to be implemented in an FPGA, the number of CLBs required grows increasingly non-linear. Hence, a compromise in speed and area have to be made in order to map the rest of the design completely and efficiently.

Figure 8: 32-bit floating-point format.

This thesis presents one of several ways to implement a 32-bit floating-point multiplier and adder arithmetic logic units based on the IEEE floating-point format. The IEEE format uses an 8-bit exponent, **E**, with an excess of 127 and a 23-bit mantissa field, **M**. With normalization included, the mantissa value goes to 24-bits. The sign bit, **S**,

determines positive or negative nature of the value. The format allows non-zero magnitudes in the range of approximately 1.18×10^{-38} to 3.40×10^{38} [23].

The design takes into account both real estate with respect to the rest of the filter design and desired speed to run the application. The goal of using 32-bit floating-point includes increased precision and range over fixed-point representations commonly used in signal processing applications. Chapter 4 presents a more in-depth discussion of the compromises made to implement the ALUs in addition to the filter logic on a single FPGA. High speed reconfigurability through a PROM, FLASH memory, or other peripheral device makes FPGAs ideal for custom logic without the need for discrete components. Hence, the progression of FPGA enhancements offer more design options to replace several of these discrete components. The 32-bit floating-point ALU design presented in this paper attempt to take advantage of state-of-the-art FPGAs taking both size constraints and speed into consideration with the rest of the design partitioning. Different VHDL constructs need to be examined since the designer must rely on the tools somewhat to be efficient with the resources used in the FPGA. Only rough estimations can be made from the code of how much real estate the design actually consumes on the FPGAs. Chapter 3 discusses insights of the design and implementation used.

Chapter 3

32-bit Floating-Point Arithmetic Logic Element Design

The following chapter discusses the construction of the arithmetic logic elements from the ground up. To provide compatibility with most host platforms, a format similar in structure to the IEEE 754 32-bit floating-point format has been chosen. Floating-point data to and from the host does not have to undergo any format conversions as found with shorter word representations of 16-bit or 18-bit formats in [21]. However, the coefficients must be pre-processed prior to being written to each processing element local memory space due to the multiplier design. Furthermore, the arithmetic logic elements do not handle exception cases including NaNs, overflow, and underflow. Sections within this chapter cover different design considerations that have to be examined, such as area and speed constraints. Filter calculations simply need to multiply the data with known coefficients and sum them accordingly to produce the convolved result. The adder and multiplier unit both use a fully pipelined design to achieve a floating-point operation per clock cycle.

Designs on earlier CCMs, including Splash, investigated shorter word formats including 16-bit and 18-bit floating-point formats to fit specific application needs. XC4010 Xilinx parts provide less than 10% of the available resources found in current FPGA technology. New technologies allow investigations with larger floating-point formats. As later chapters show, even with such resources available, the design considerations must still

pay attention to area and speed compromises. Some combinational multipliers consume resources in an increasing, $O(n^2)$ fashion, unlike adders.

The essential idea behind floating-point number systems is to formulate representations and computation procedures in which the scaling procedures introduced by fixed-point systems are built-in [25]. In a floating-point system with radix *R*, a number *N* is represented by a pair <*E*,*S*>, where *E* is a signed fixed-point integer and S is a signed fixed-point number, such that $N = S \times R^E$. The value S is also known as the significand and E as the exponent. Addition of two values using this format require that the exponents be of equal value before their significands be added which leads to variable length shifts on the significand, S. Multiplication requires much less overhead and the product more readily normalized. Basically, each operation can be broken down in to smaller, discrete operations on their respective fields which make it ideal for a pipelined architecture.

3.1 Design Considerations

Choosing the proper 32-bit format enables the arithmetic logic units to coincide within a host that also uses the standard to provide a co-processing, computing engine. Aside from convenience of the chosen format, designers can tailor the format with custom computing machines. In [22], the FFT implementation takes advantage of the 36-bit data path of Splash 2 allowing the application to transfer two data words at a time. Decisions that involve the format depend on the application at hand which include range, accuracy, and precision requirements. By considering the floating-point format, the total decibel range as calculated by [19] is

$$(2^e + m) \times 20\log 2 = 1679.75 \text{ dB} \qquad (3.1)$$

where $e = 8$ for the number of exponent bits and $m = 23$ for the number of mantissa bits. The dynamic range for using the 32-bit format allows more than an order of magnitude over a 22-bit or 16-bit format giving just 480 dB and 440 dB, respectively [19]. In some fixed-point applications, re-scaling can be done to handle comparable ranges found in floating-point rather than accommodating larger word growths becomes an alternative. Unfortunately, the side effect of overflow may occur or precision is lost [19]. Although

overflow and underflow require special circuitry for either format, such occurrences happen less often with floating-point due to a greater dynamic range. The format of the floating-point number depends greatly on the application requirements and expected bounds. Providing excessive resolution comes at a high price and should not be spent if not needed.

Truncation and rounding effects must also be considered when dealing with either fixed-point or floating-point formats. According to [6], these effects introduce an error value which depends on the word size of the original value and how much of the word is truncated or rounded. The attributes of the introduced error rely on the particular form of the number representation. For fixed-point sign magnitude values, the truncation error is symmetric about zero and falls in the range

$$-(2^{-b} - 2^{-bu}) \leq E_t \leq (2^{-b} - 2^{-bu}) \tag{3.2}$$

where E_t is the error, *bu* is the number of bits prior to truncation. For two's complement format, the truncation error is always negative and falls in the range

$$-(2^{-b} - 2^{-bu}) \leq E_t \leq 0 \tag{3.3}$$

Round-off error can be represented by the equation,

$$E_r = Q_r(x) - x \tag{3.4}$$

which is independent of the format of the fixed-point value and only affects the magnitude of the value. Since the maximum error through rounding is

$$E_{max} = (2^{-b} - 2^{-bu})/2, \tag{3.5}$$

round-off error becomes symmetric about zero and falls in the range

$$-(2^{-b} - 2^{-bu})/2 \leq E_r \leq (2^{-b} - 2^{-bu})/2 \tag{3.6}$$

Since floating-point values have a non-uniform distribution, the error introduced from either truncation or rounding becomes proportional to the value being quantized [6]. The quantized value can be represented as

$$Q(x) = x + ex \tag{3.7}$$

where e is the relative error. The following equation can be used to give boundaries to the truncation error, ex, for positive values

$$-2^E 2^{-b} < e_t x < 0 \tag{3.8}$$

Since $2^{E-1} \leq e_t < 2^E$, then

$$-2^{-b+1} < e_t \leq 0, \text{ for all } x > 0 \quad (3.9)$$

For negative numbers,

$$0 \leq e_t < 2^{-b+1}, \text{ for all } x < 0 \quad (3.10)$$

The algorithms in the multiplier and adder units for the filter use truncation, chopping off the lower bits. Instead of a two's complement representation, the arithmetic units in the filter design use a sign-magnitude representation which differs primarily in the distribution of the error value. [6] describes the error value as an equivalent random variable under a uniform distribution serving as additive noise. Chapter 6 presents the actual error analysis found between the expected value and the truncated value which found errors far less than 1% of the expected value.

Aside from mathematical limitations, physical limitations with FPGAs exist. The design elements must fit within a fixed number of configurable logic blocks with a limited number of routing resources to connect the logic cells. The multiplier unit demands a large amount of FPGA resources when implemented through the VHDL multiplication symbol, typically on the order of n^2. The synthesized integer multiplier that computes the product of the two operand mantissas generates a large amount of logic for a single cycle computation. Pipeline alternatives can be investigated to delay the integer multiplication result. The filter design incorporates a pipeline integer multiplier developed by Annapolis Micro Systems. The integer multiplier requires approximately half the amount of space as those generated by the synthesized VHDL multiplier operator. Further sections discuss how the design integrates the multiplier in a pipelined design to achieve a floating-point operation per clock cycle.

3.2 32-bit Floating-Point Adder

The pipelined design of the 32-bit floating-point adder has a latency of eight clock cycles. Once the pipeline has been filled, the adder can generate a result each clock cycle so long as new operands are given every clock cycle thereafter. The design follows that of Shirazi's 18-bit floating-point format adder in [22] except one of the stages has been split into multiple stages to allow the adder to run at a slightly faster clock speed. Basically, by

having more stages with fewer logic levels, the pipeline can be run at faster clock speeds [19]. The following two sections describe the algorithm used to partition the different stages of manipulation and the VHDL constructs used to synthesize the arithmetic logic unit.

3.2.1 Algorithm and Design

A scientifically formatted floating-point value, N, can be represented as $N = S \times R^E$, where S is a fixed-point value multiplied by a radix, R, to some power, E. The algorithm to add two numbers with this representation requires that the radix power, E, be the same in power and sign. If so, the significand components, S, can be summed together while keeping the same radix to some power, E, as the multiplier. If they are not the same, adjustment of the significand needs to be made prior to addition. For instance, in scientific notation, the value 3456.983 can be represented as 3.456983×10^3. Binary based values can also be represented in a similar fashion where 1.101×2^3 might be better understood as $1.625 \times 8 = 13.0$ in base 10. If one adds two operands in a scaled format, or scientific notation, the scaling factor exponents have to be equal. To do this, the decimal point of one of the significands must be shifted so that the exponents become equal in magnitude and sign. After doing so, the significands can be added with the proper decimal alignment. Typically, it is the significand of the operand with the smaller exponent that is shifted; therefore, the shift is to the right. Were the other operand to be shifted, this would be a left-shift, and, as the significance of digits increases towards the left-hand end, the most significant digits could be lost [25]. For example, if the two values $\mathbf{A} = 3.45 \times 10^8$ and $\mathbf{B} = 38.754 \times 10^6$ are to be added, **B** needs to have its significand shift twice to the right to become 0.38754×10^8. With the **A** and **B** values having the same exponents, the two significands can then be added for a sum of 3.83754×10^8. Similar techniques can be done for radix-2 based values. For instance, if the two values $\mathbf{C} = 11.101 \times 2^4$ and $\mathbf{D} = 100.011 \times 2^5$ are to be added, **C** in this case would need to be shifted one place to the right (thus moving the decimal to the left one place). Hence, the values 1.1101×2^5 and 100.011×2^5

give 110.0011 x 2^5. Figure 9 below illustrates the steps necessary to perform floating-point addition. The two values to be added are represented by $N_a = S_a \times R^{Ea}$ and $N_b = S_b \times R^{Eb}$.

During calculations, retaining the significant digits results in more accuracy [25]. Performing operations to delete the non-significant digits (zeros) becomes advantageous in machines since the number of bits to represent the significand is fixed. A significand attains a *normalized* state when the leading (leftmost) non-sign digit is significant (non-zero). In radix-10 values for instance, 0.000456 x 10^7 can be normalized by left-shifting the significand

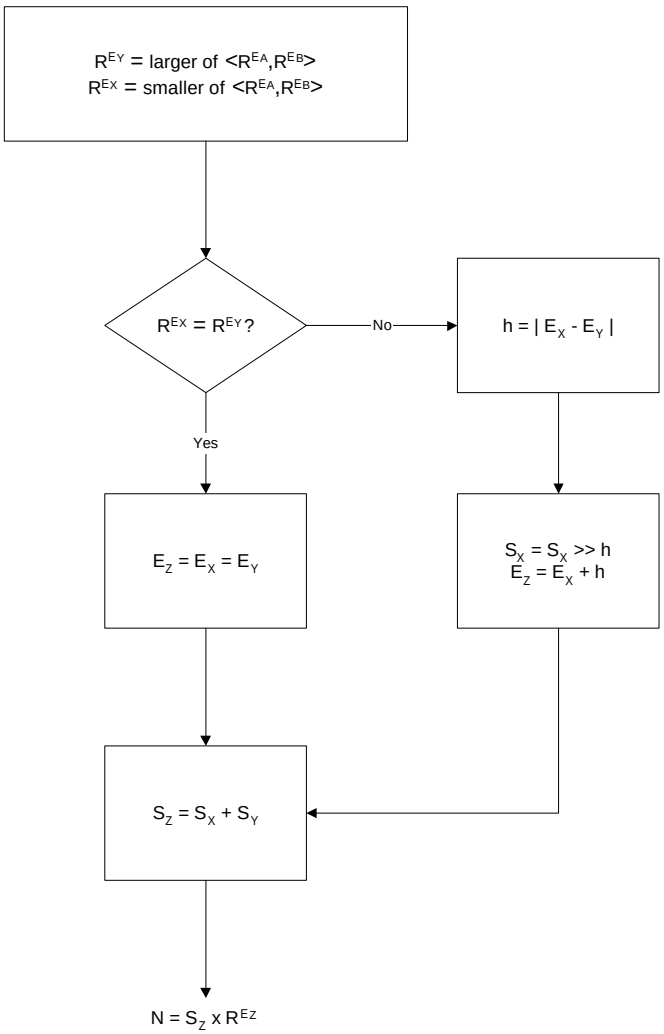

Figure 9: Flow diagram for floating-point addition.

by four spaces which means decrementing the exponent by four to become 4.56×10^3. In the case of binary, or radix-2, numbers the normalization process remains the same; shifting the significand until the most-significant digit has a one while decrementing the exponent by the number of left shifts or incrementing the exponent by the number of right shifts to attain the normalization. In the case from above, the value 110.0011×2^5 can be normalized to 1.100011×2^3. In the IEEE 754 floating-point format, the significand always takes on an implied, or hidden, '1' for the most-significant digit assuming the value represented is normalized. Although the IEEE 754 format according to [23] and [25], does have support for denormalized numbers. Storing normalized numbers allow machines to maximize the number of significant bits for higher accuracy.

The adder design pipelines the steps described in the previous paragraph to achieve a summation every clock cycle. Each pipeline stage performs operations independent of others. Input data to the adder continuously streams in from both the multiplier and the accumulation data from the input bus. The operations have been divided into eight stages in the pipeline to help sustain a target clock rate of 20 MHz. In addition, a particular number of stages have been chosen to properly coordinate with the multiplier to sum the products with the incoming accumulation values. The adder must accumulate the two real parts from the complex multiplication with the real part of the incoming accumulation. The same must be done for the imaginary parts. The filter design interleaves the accumulation steps for the adder so that three passes through the adder have to be done before attaining the complete accumulation of all the partial products. Multiple passes are needed since some partial products cannot be readily available on the next clock cycle due to the pipeline latency. Section 4 presents more detail on how the filter tap design incorporates the adder and multiplier together to compute the accumulation value for the next tap.

Much of the design flow follows that of Shirazi's 18-bit pipelined adder unit as documented in [22]. Modifications have been made to lengthen the pipeline and thus increase the speed at which the design may execute. After further analysis of Shirazi's three stage pipeline, extensive routing and logic have been used to implement the shifting and leading-one detection operations. Different VHDL coding techniques to implement a barrel-shifter helps to reduce the amount of routing for the shifting stages. Additional stages

and a slightly different technique to perform the leading-one detection operation have been changed to reduce the amount of logic used in Shirazi's exhaustive approach. The following sub-sections describe each of the adder pipeline stages in more detail as shown in Figure 10.

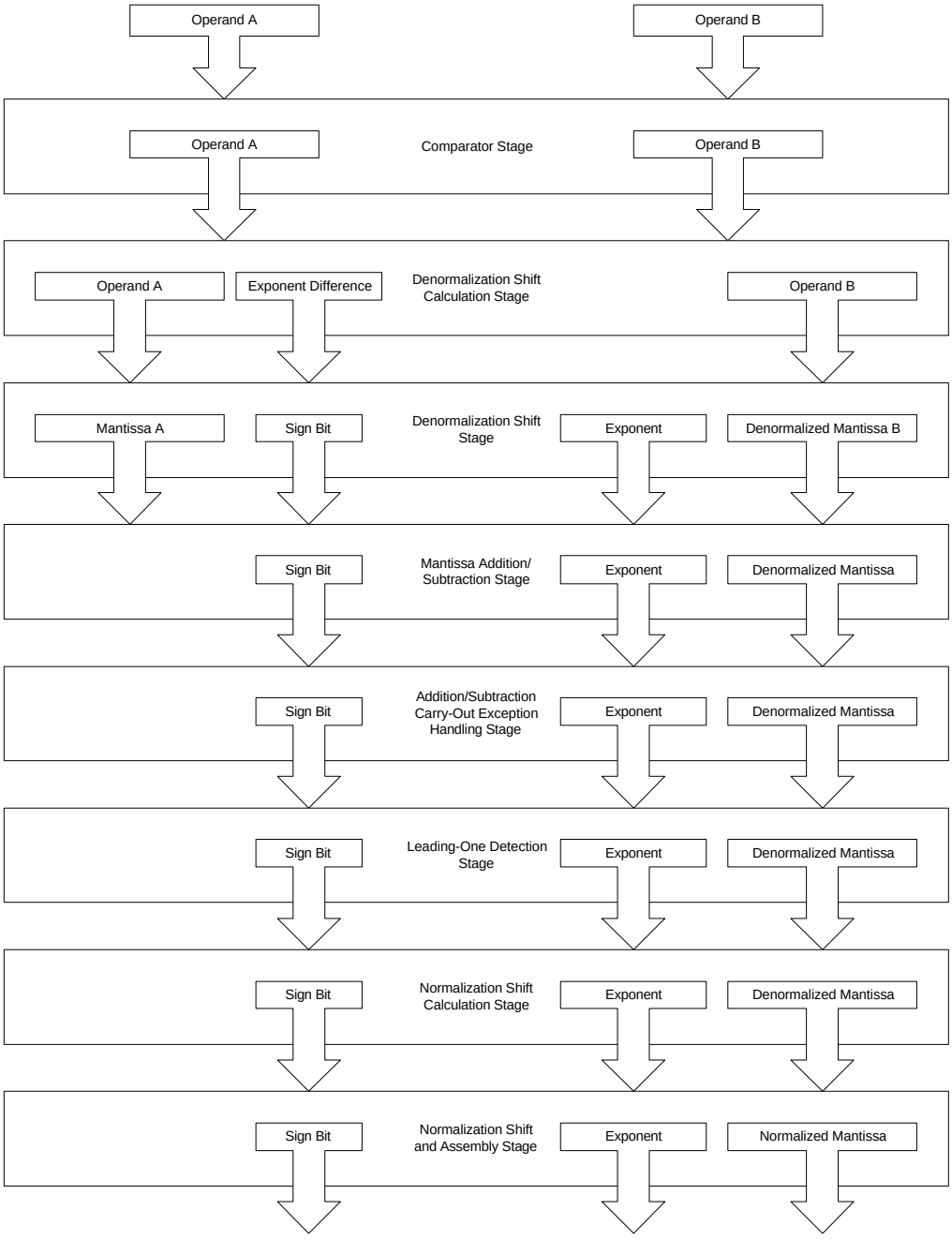

Figure 10: Pipelined multiplier flow diagram.

3.2.2 Implementation in VHDL

As mentioned earlier, VHDL can be used to represent several different abstraction levels [1,26]. The level chosen to represent the 32-bit floating-point adder includes a mixture of register-level and gate-level logic blocks. To describe the register transfer level and the data flow through the adder, VHDL constructs called *processes* provide the sequential instructions that manipulate the data. Each process contains a *sensitivity list* that triggers the actions within a process. [27] provides ideas on how to code more efficiently to use the synthesis tool capabilities. Since more than one process can exist for a single entity, two distinct processes can be used to describe the registered, or clocked part, of the algorithm while the other describes the combinational aspect. The following section describes how the 32-bit floating-point adder uses the two different type of processes to build the pipelined adder architecture.

3.2.3 Stage 0: Comparator Stage

Determining which of the two operands has the greater magnitude alleviates complex conditionals in later stages of the adder. The initial stage in the adder uses comparator logic to place the larger of the two operands as operand A. The combinational VHDL process compares the exponents to make an initial determination. If the exponents are equal, the logic then compares the mantissa values. Sign bits do not effect the comparison. The registered process handles insertion of the implied leading-one for each new mantissa value. The leading-one insertion operation always takes place regardless of the operand values. Stage 2 performs zero value exception handling.

3.2.4 Stage 1: Denormalization Shift Calculation Stage

In order to add two floating-point values in scientific notation, the two values must have the same exponent in both sign and magnitude. The adder must perform this operation by shifting one of the operands and making adjustments to the operand exponent value. Stage 1 of the pipeline takes the difference of the two operand exponents to

determine how many shifts are needed on operand B. By shifting to the right, the operand stands to lose only lower significant bits.

3.2.5 Stage 2: Denormalization Shift Stage

The adder performs a check on the two operands to see if they both have the same magnitude and different signs resulting in a zero sum. If so, the resulting exponent, mantissa, and sign value become zero. Zero checks for the result must be made before further processing to prevent invalid results occurring due to the inserted leading-one of Stage 0. Should the operands not produce a potential zero result, the rest of Stage 2 performs the shifting operation on operand B using a barrel shifter. The adder examines the shift value obtained by taking the difference of the exponents in Stage 1. Shifts in quantities of 1, 2, 4, 8, and 16 are done on the operand. Upon leaving the stage, the adder only needs to keep a single exponent and sign bit along with the two mantissas.

3.2.6 Stage 3: Mantissa Addition/Subtraction Stage

Stage 3 of the adder pipeline performs the addition/subtraction of the two mantissa integer values. The sign bit indicates whether addition or subtraction takes place and if the carry-in bit to the adder/subtractor should be a 0 or 1, respectively. Performing subtraction requires a 2's-complement addition and adding in a 1 for the carry-in bit. Note that since operand A is greater than operand B, a borrow cannot happen, and thus, the carry-out bit of the result is cleared. The carry-out bit becomes important in the next stage of the pipeline which may indicate the result needs no further normalization nor exponent adjustment.

3.2.7 Stage 4: Addition Carry-Out Exception Handling Stage

The resulting mantissa value must always go through the leading-one detection (LOD) logic regardless of the carry-out bit. If the pipeline stage determines a subtraction

took place or if an addition took place with no carry-out, no changes to the mantissa have to be done. But if an addition took place with a carry-out, an immediate adjustment to the exponent must be done prior to the LOD stage since the bit does not take part in the 23-bit mantissa result vector. To do so, the stage must shift the result vector to the right by one to accommodate the carry-out bit as the new leading-one. The LOD logic will then find the leading-one in bit location 23.

3.2.8 Stage 5: Leading-One Detection Stage

Rather than using an exhaustive approach as Shirazi has in [22], another similar technique has been used. The leading-one detection logic has been broken up into two distinct phases. The first, in Stage 5, determines which of the six nibbles of the mantissa value contains the leading-one. The VHDL builds a 6-bit word using gate-level constructs that describe which nibble the leading-one resides in, if one exists. No more than four

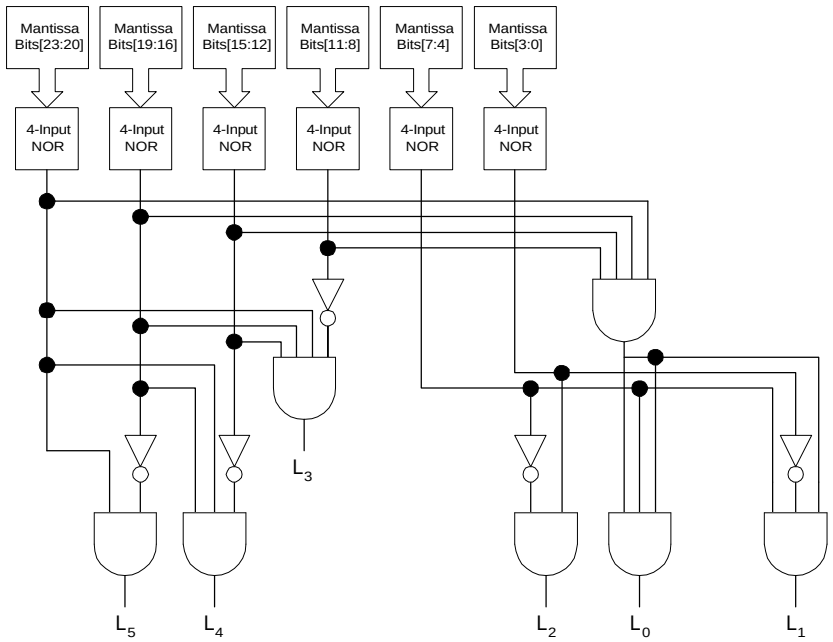

Figure 11: Leading-one detection logic.

levels of logic are used to build such a word. In addition, no gates require more than four inputs to simplify CLB usage in the FPGA. Table 1 illustrates the possible "nibble sector" words that can be constructed from the LOD logic. Nibble sector 5 contains the most significant bits while nibble sector 0 contains the least significant bits. Figure 11: Leading-one detection logic. illustrates the LOD logic where the L_x output represents the different sector word bits.

Table 1: Nibble Sector Words for LOD Detection.

Data Word [5:0]	Description
000000	Nibble sector 5 contains leading-one
000001	None detected
000010	Nibble sector 0 contains leading-one
000100	Nibble sector 1 contains leading-one
001000	Nibble sector 2 contains leading-one
010000	Nibble sector 3 contains leading-one
100000	Nibble sector 4 contains leading-one

3.2.9 Stage 6: Normalization Shift Calculation Stage

Stage 6 works with Stage 5 to produce the number of shifts required to normalize the resulting mantissa value after the addition/subtraction takes place. The stage constructs the 5-bit shift value using the data word from Stage 5 that determines which of the six nibbles in the resulting mantissa the LOD resides in as well as the four bits within that particular nibble. The data word can be used to determine what the upper three bits of the shift value are to be while the lower two bits are determined by the bit values in the nibble containing the leading one. The combinational logic to determine the two bits can be constructed from two, 4-variable logic equations:

$$s_0 = (\sim n_3 \bullet n_2) + (\sim n_3 \bullet \sim n_2 \bullet \sim n_1 \bullet n_0) \tag{3.11}$$

$$s_1 = (\sim n_3 \bullet \sim n_2) \bullet ((n_1 \oplus n_0) + (n_1 \bullet n_0)) \tag{3.12}$$

where s_0 and s_1 are bits 0 and 1 of the constructed shift value, respectively. The n_3, n_2, n_1, and n_0 values represent bits 3 to 0 of the nibble containing the leading-one, respectively. The shift value along with the resulting mantissa, the exponent, and sign are passed onto the next and final stage in the pipeline.

3.2.10 Stage 7: Normalization Shift and Assembly Stage

The final stage of the pipeline assembles the 32-bit floating-point result. The previous stage passes in the sign, exponent, and denormalized mantissa result from the Stage 3 addition/subtraction. In addition, Stage 6 passes in a shift value to normalize the mantissa such that the leading-one in the mantissa resides in the most significant bit location. The stage also uses the shift value to adjust the exponent to the number of shifts required. The last stage uses another barrel shifter to perform the shift operation on the mantissa allowing shifts of 1, 2, 4, 8, and 16.

3.3 32-bit Floating-Point Multiplier

The second basic arithmetic logic unit needed to perform digital filtering is a multiplier. Constructing a fast multiplier in an FPGA presents a challenge due to the shear amount of logic required which can be estimated as described in Section 3.1. The implementation still follows some of Shirazi's 18-bit pipelined multiplier [22] but deviates somewhat due to the mere size of the logic and routing produced for a combinational, integer multiplier. Shirazi's design incorporates a straight-forward approach using the VHDL multiplication operator which relies on the synthesis tool to construct the necessary logic for the multiplier. The multiplication operator expects to produce the result after a single clock cycle, thus producing a circuit requiring substantial amounts of CLB resources. Instead, a pipelined approach for the integer multiplier has been examined to continue producing a result each clock cycle. The pipeline latency remains the only drawback which is not a concern since the filter continuously feeds the multiplier every clock cycle. By using

a pipelined multiplier, the resource consumption not only decreases but the speed may actually increase.

3.3.1 Algorithm and Design

According to [25], floating-point multiplication is inherently easier to design than floating-point addition or subtraction. Multiplication requires integer addition of operand exponents and integer multiplication of significands which facilitate normalization when multiplying normalized significands. These independent operations within a multiplier make it ideal for pipelining. Shirazi's multiplier in [22] shares the ideas with [25] in that the following three steps can be done:

1. Unpack the operands, re-insert the hidden bit, and check for any exceptions on the operands (such as zeros or NaNs).
2. Multiplication of the significands, calculation of the sign of the two significands, and addition of the exponents take place.
3. The final result needs to be normalized and the exponent adjusted before packing and removing the hidden bit.

Typically, an additional stage can be used to perform rounding and re-normalization, but in order to save space and cut down on additional logic, truncation has been used. Results may show slight discrepancies between those generated on a host computer but discussions in a later chapter show that they stay well below 1% since the truncation affects low order bits on a 23-bit significand. According to [25], the propagation of errors is such that in the linear processes of addition and subtraction the magnitudes of the absolute errors in the operands add up; whereas in multiplication and division, the magnitudes of the relative errors in the operands add up. The absolute error is the difference between the approximation and the true value, and the relative error is the ratio of the absolute error to the true result. Earlier implementations of a 16-bit multiplier on Splash-2 show much larger discrepancies due to a smaller significand where the lowest order bit in the significand represents 1×10^{-3} of a value as opposed to 1×10^{-7}. The amount of acceptable error depends on the application specifications and tolerances.

Multiplication does not require shifting of the significands or adjustment of the exponents as in the adder unit until the final stage for normalization purposes. For the basic summation of partial products in a floating-point multiplication represented in scientific notation (significand multiplied by the radix to some power), one multiplies the two significands and adds the two radix powers. Normalization of the significand ensures the decimal point of the significand has a exactly one significant digit to the left of it which may or may not need to be done. For example, multiplying **A** = 5.436 x 10^{8} by **B** = 8.995 x 10^{-4} would result in **C** = 48.89682 x 10^{4} and 4.889682 x 10^{5} with normalization. Similar operations are performed on binary numbers as well. The format chosen for this design follows the basic steps as described above to perform multiplication. Shirazi's 18-bit floating-point multiplier uses a single combinational logic multiplier in the second pipeline stage getting the integer result within a single clock cycle [22]. Unfortunately, a 24x24 integer multiplier consumes much more space than that of an 11x11 multiplier; almost six times the configurable logic blocks (CLBs) and considerably more routing resources. Speed of the multiplier depends on how well the tools place-and-route the logic. If the tool inefficiently routes signals within the multiplier and to supporting logic (such as registers and control signals), long delays can affect the overall speed performance. Therefore, other alternatives may need to be examined to compromise between area and speed.

The designed floating-point multiplier consists of the 13 stage pipelined integer multiplier, pipeline delay elements, and an adder/subtractor unit to handle the exponent computation. The multiplier cannot directly accept 32-bit floating-point operands as discussed in Chapter 2 for operand B. Operand A may be issued directly to the multiplier, however. The operand B mantissa must be given to the multiplier on a partial basis but may be interleaved with other operand B values entering the multiplier pipeline. Since the multiplier must be issued static coefficients, an interleaved pattern may be formed for operand B to make implementation and design much easier. The exponent, however, must be calculated before and after the mantissa multiplication. Before entering the exponent pipeline delay, the exponents of operand A and B are added together. Once the mantissa product completes, the most significant bit must be examined to ensure the implied one is properly placed. The final stages of the pipelined floating-point multiplier ensure the

biasing of the exponent is properly done which relies on the most significant bit of the integer multiplication product. The floating-point multiplier block diagram can be seen in Figure 12.

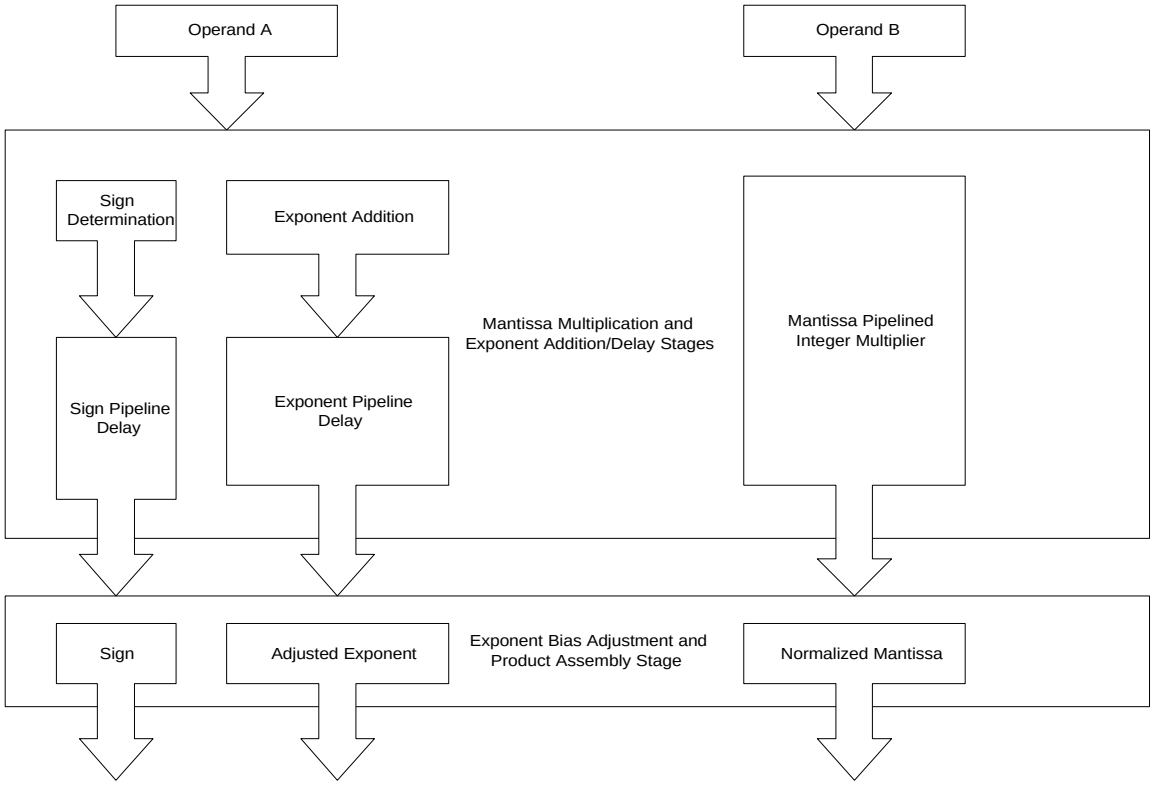

Figure 12: Pipelined Multiplier Block Diagram.

3.3.2 24-bit Pipelined Integer Multiplier

The pipelined floating-point multiplier generates a product every clock when the pipeline has completely filled, and has a latency of 13 cycles. The pipeline stages for the multiplier are much simpler in comparison to the adder stages. Twelve of the 13 stages are used for the computation of the integer multiply. By simply relying upon VHDL, synthesis tools for the creation of the multiplication produced a design consuming 576 CLBs, which was deemed unacceptable considering the planned resource budget. For an XC4036EX Xilinx part, the 24x24 multiplier would consume roughly half of the available CLBs.

An alternative integer multiplier was created using a parameterized multiplier generation program [32]. The generated 24x24 integer multiplier utilizes Booth recoding [33], and inserts RLOC information and pipeline stages to preserve routing, timing, and size of the multiplier. The multiplier consumes roughly 300 CLBs. Two bits of the multiplier are issued at a time for twelve consecutive clock cycles, starting with the lowest two bits. Figure 13 illustrates this process with two eight bit integers $(181)_{10}$ and $(216)_{10}$. Partial results are generated each clock cycle as the bits enter the multiplier. The upper half of the product appears after all operand B bits have been issued. Even though Figure 13 shows only two values being multiplied, the multiplier is constructed to concurrently process $n/2$ multiplications. The remainder of the bits, for instance, of Cycle 0 in Figure 13 may contain bits for three additional multiplies where bits [2:3] belong to a multiplication at time, *t-1*. Bits [4:5] belong to a multiplication at time, *t-2*.

Cycle	Operand A	Operand B	Product
0	10110101	------00	--------------00
1	--------	----10--	------------10--
2	--------	--01----	----------11----
3	--------	11------	--------10------
4	--------	--------	10011000--------

Figure 13: Example integer multiplication.

Figure 12 illustrates the pipeline multiplier stages for the floating-point multiplier. The exponent and mantissa operations can be performed concurrently until the final stage where normalization takes place. In floating-point multiplication, the exponents must be added together as they are in this implementation during the first stages. The result from the exponent addition continues through a pipeline delay until the mantissa result completes. Carry-out logic from the mantissa multiplication informs the control logic not to perform a 1-bit shift since the implied one exists. Note that the exponent must continue through several pipeline delays which requires registered logic.

For additional resource savings, the multiplier implementation does not perform rounding. The design truncates the mantissa value, saving only the upper half of the multiplier result. By truncating the lower bits of the mantissa, [4] describes the error value as an equivalent random variable under a uniform distribution serving as additive noise.

3.3.3 Pipelined Delay Component

The pipelined delay component takes advantage of the RAM capabilities of the Xilinx CLBs. The delays may only be 16 deep but have been constructed in a generic fashion to provide variable pipeline delays from lengths 2 to 16. Each CLB contains a RAM array which can be used in conjunction with a 4-input function generator to provide the write decoding signals. In addition, the delay has been made synchronous in conjunction with other pipelined components.

3.3.4 Implementation in VHDL

The multiplier VHDL consists of several different components that rely on a clocked process and registered signals. The components consist of a pipeline delay element, a 9-bit adder, a 9-bit subtractor, and a 24x24 pipelined integer multiplier as described in Section 0. The VHDL clocked process provides much of the glue logic for the components used in order to ensure signals to each component are registered properly and to avoid timing hazards. The pipelined integer multiplier, for instance, requires that the inputs be registered for expected results. The inputs to the floating-point multiplier need to be checked for a possible zero outcome and assert a flag through the pipeline to indicate a zero value be given as the result during the last stage in the pipeline. The VHDL code provides concurrent operations for some of the initial stages. As the mantissa undergoes integer multiplication, calculations on the exponent are done and passed through a pipeline delay to remain synchronized with the integer multiplier data. The VHDL used in the multiplier differs from the 32-bit floating-point pipelined adder in that no state machines are

required for the multiplier. Instead, the VHDL provides minimal control logic to ensure the components are given data on the correct cycles.

3.3.5 Stages 0-12: Mantissa Multiplication and Exponent Addition

The multiplier undergoes two separate, parallel operations during the first 12 stages. One of the operations includes multiplying the two 24-bit mantissa values using the 24x24 pipelined integer multiplier. As Section 0 explained, the multiplier generates the result on the thirteenth clock cycle. During the mantissa calculation, the exponent addition takes place using the 9-bit integer adder component. Nine bits, instead of eight, are used to handle carry-out situations. The carry-out bit provides important information used in the final stage of the floating-point multiplier to handle exponent biasing adjustments. Since the exponent calculation does not require more than a clock cycle, a pipeline delay component delays the calculated exponent result until the last stage when the bias adjustments are ready to be done. In addition, two smaller logic operations take place. The first determines if either of the input operands are zero. If so, a special zero-flag needs to be set. The second uses XOR logic to determine the resulting sign bit of the two input operands. The zero-flag and sign bit need to be delayed as well until the last stage in the floating-point multiplier. All data going through the pipelined delay must continue to be synchronized with operand B going through the pipelined integer multiplier.

3.3.6 Stage 13: Exponent Adjustment and Product Assembly Stage

The last stage receives the data from the pipelined integer multiplier and the other pipeline delay elements. The stage logic checks the zero-flag bit to see if the output is simply a zero. Otherwise, a one in the most significant bit of the mantissa indicates the resulting mantissa value has already been normalized. If not, the exponent must be adjusted by one and the mantissa output shifted by one. The exponent undergoes subtraction to remove an extra biasing factor from the addition of an earlier stage in the pipeline. Depending on the most significant bit of the mantissa, different values are subtracted from

the exponent. The final stage assigns the resulting values to the output signals of the floating-point multiplier.

Chapter 4

Filter Tap Design and VHDL Implementation

The following chapter presents the operation of the scalable filter tap. The scalable filter tap design integrates the 32-bit floating-point adder and multiplier elements with the necessary control logic and pipeline delays to perform FIR filtering using complex arithmetic. The chapter presents a method to map the filter on a CCM possessing a systolic array architecture using Xilinx XC4036EX FPGAs and 32-bit data buses.

4.1 1-D Time Domain Convolution on a CCM

The design uses the systolic array of processing elements to perform parallel calculations for the filtering. The next two sections discuss how the algorithm for an n^{th} order filter maps onto the linear array of processing elements. The first section presents the basic filter calculations and how the design needs to be partitioned across the CCM processing elements. The second section describes how the algorithm translates into VHDL code in order to be synthesized to execute on the hardware. The most important concept in the first section involves algorithm partitioning which ultimately determines how much of the design each processing element incorporates.

4.1.1 Algorithm and Design Considerations

The algorithm is based upon two taps per processing element and larger filters are formed by cascading PEs. Chapter 2 covered information on convolution and the similar calculations required for FIR filtering. The algorithm presented here uses a series of weighted taps where the different weights represent the FIR filter coefficients. The coefficients come from a specific filter design based on the type of filtering for which the application needs. For time domain filtering, the weights are delayed relative to the input sequence given to the filter, but for each data value entering the filter, the sum of the tap products give the result for the filter output sequence. The partitioning of such an algorithm must take into account speed and area consumption of the logic necessary. In Chapter 6, another important limited resource that needs to be taken into consideration includes routing resources.

Area consumption considerations depend greatly on the platform being chosen. In this case, four Xilinx XC4036EX processing elements on a WILDFORCE board make up the computing array. Since the computing array of processing elements can be expanded over several boards, the algorithm does not have to be limited to these four processors alone. Future expansion of the processing array with multiple boards allows higher order filters to be mapped. The following discusses a mapping that allows one or more processing elements at a time to be appended to the processing array to increase the filter order. The algorithm for each processing element can be used in a cascaded manner such that the data stream enters one side of the array and is then piped through the rest of the processing elements in a linear fashion. The filter results exit the filter tap array in constant time intervals. To provide flexibility, each processing element in the array needs to provide multiplication of the input data stream with the proper filter coefficients as well as accumulation of the tap product.

Some filter realizations offer distinct advantages over others depending on the resources that are available. Block diagrams with interconnected multipliers, adders, and delay elements are referred to as realizations, or structures, to represent the filter system. The structure may exhibit advantages for software or hardware depending upon the arrangement. [6] considers three major factors that influence the choice of a specific

realization including computational complexity, memory requirements, and finite-word length effects in the computations. Computational complexity refers to the number of arithmetic operations required to compute an output value for the system. Memory requirements refer to the number of memory locations required to store the system parameters, past inputs, past outputs, and any intermediate computed values. Thirdly, the finite-word length effects refer to the quantization effects that are inherent in any digital implementation of the system, either in hardware or software. Computations performed are either rounded-off or truncated to fit within the limited precision [6]. Direct forms, cascade, parallel, and lattice structures provide robustness in finite-word-length implementations [4,6]. This thesis uses a direct form structure to implement the filter which does not require computation of feedback paths and has a low computational complexity of M multiplications and (M - 1) additions per output point where M is the number of weighted taps. The direct form realization requires (M - 1) memory locations to store the (M - 1) previous inputs. Figure 3 in Section 2.1.2 illustrates the direct form block diagram. The block diagram manifests the repetitive computations necessary for each tap which simplifies the algorithm partitioning.

A primary objective in designing and partitioning the logic for each processing element includes making the architecture general. Using a general design allows the logic to be easily replicated among several other processing elements. For the design in this thesis, each processing element contains two filter taps which share a single floating-point adder and a single floating-point multiplier. Sharing of resources must be done due to physical constraints. With this design, the processing elements must perform eight multiplies and eight additions for each set of complex data values that come from the input data stream. The data stream format must not only contain both real and imaginary parts of the data values to filter but also the real and imaginary parts of the accumulation values. The control processing element constructs the data stream in a specific format to allow the processing elements to properly compute the filter results as shown in Figure 14. As the data stream enters the filter processing elements, a state machine continuously sends the data to the multiplier and adder units to compute the results. The design maintains a high computation rate through a pipelined design. Finite state machines in the processing element control and

provide exact timing for the different intermediate results that must be sent to the arithmetic, delay, and register components.

Re(Data1)
Im(Data1)
Im(Data1)
Re(Data1)
0
0
0
0
Re(Data0)
Im(Data0)
Im(Data0)
Re(Data0)

Figure 14: Constructed input data stream by CPE0.

The control processing element (CPE0) reads data from the input FIFO which consist of real and imaginary data values only. The zeros must be inserted between each set of complex data values before being inserted into the filter for computation. The data values must be inserted in the order as shown above in Figure 14 since the multiplier issues the coefficients in another order to properly compute the complex multiplication. The zeros inserted into the data stream serve two important roles: (1) provides multiplication delay and (2) carries filter accumulation values. Recall that the multiplier must perform eight multiplies for each set of new data that comes in from the data stream. The first four multiplies include multiplying the four new data stream values with the tap 0 coefficients. The next four multiplies use the previous four data stream values with the tap 1 coefficients. Hence, while the zeros (or possibly accumulation values) are being passed in, the multiplier computes the results for the second tap in the processing element. Figure 15 below provides a high level data flow diagram of a processing element. The following sub-sections discuss the state machines that control the data flow into each of the components of the diagram.

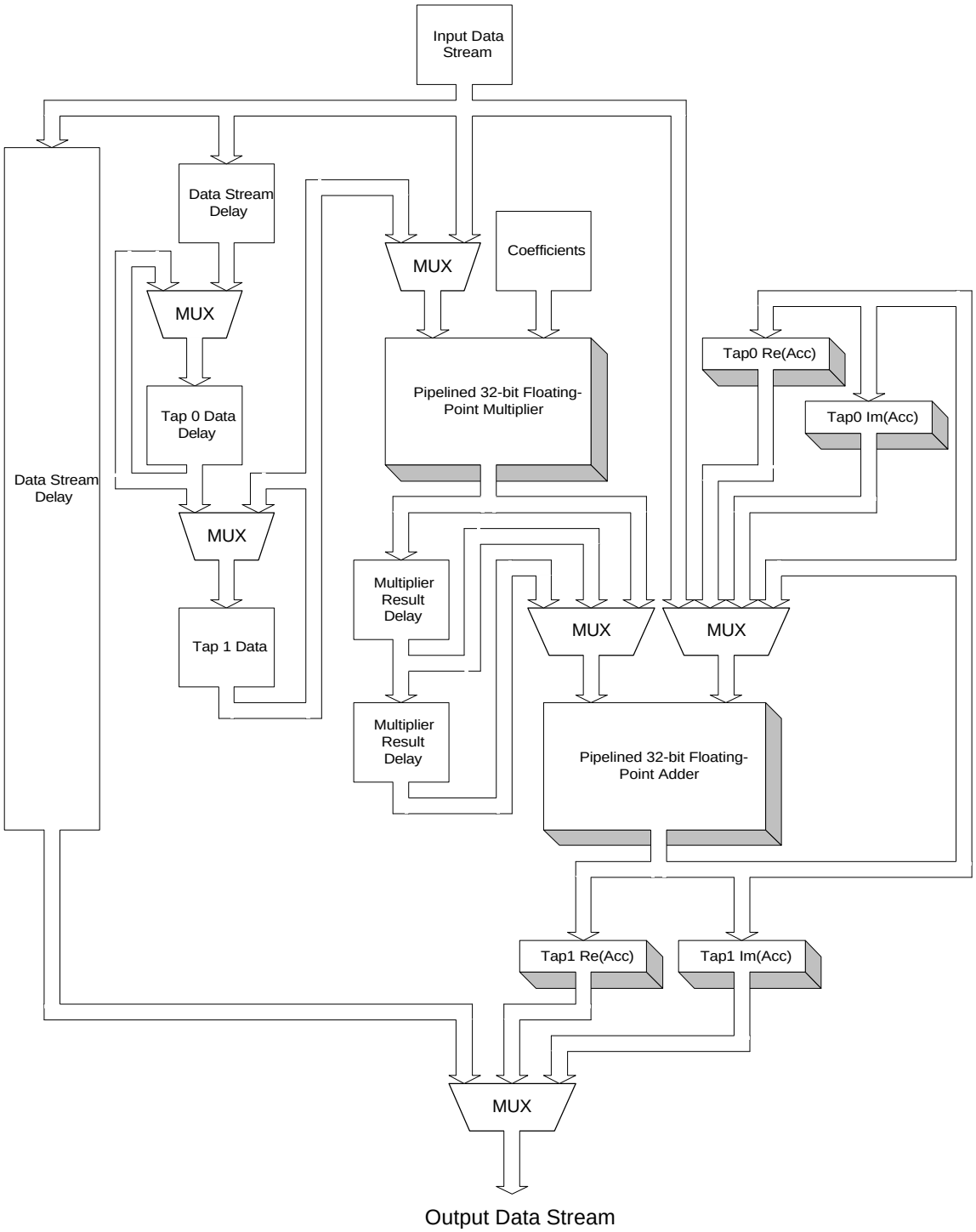

Figure 15: Processing element component connectivity.

4.1.2 Implementation in VHDL

The VHDL code consists of four state machines that provide control logic for the components shown in Figure 15 above. The processing element does not process any data coming in from the input data bus until the data detection state machine identifies valid data. Once valid data has been detected, a signal notifies the coefficient loading state machine to begin reading the coefficient data from local memory in order to issue the correct values to the floating-point multiplier. To correctly synchronize the coefficient data with the input data stream values, the coefficient loading state machine notifies the multiplier operand loading state machine when to begin accepting data from the input data stream delay line. Finally, the multiplier operand loading state machine notifies the adder operand loading state machine when to begin performing accumulations with the multiplier results and the input data stream accumulation values. The state machines synchronously transfer from one state to another using a variety of handshake signals to notify one another when to start processing. Additional signals from each state machine provide control for different components, such as registers, in order to store the correct values for each computation cycle.

The VHDL code uses several pipeline delay elements to properly synchronize the different data streams throughout the design. The floating-point adder and multiplier do not compute results in the same amount of time and thus require pipelined delay insertions between some data paths. During accumulation, certain multiplier values cannot be used immediately and must be inserted into a pipeline delay until the correct operand becomes available. Section 0 describes the state machine responsible for accumulation and the timing pattern that must be used to correctly add the real and imaginary products with the incoming accumulation value of the data stream from the previous tap.

4.1.3 Data Detection State Machine

The design incorporates a small state machine to determine whether or not the filter should begin processing the incoming data. The series of handshake signals to begin processing rely on the "dataDetected" signal to indicate when valid data has been found on the input data bus. The state machine examines the four tag bits of the input data bus and remains in a non-detected state until a single valid tag has been detected. Once detected, the state machine continues to stay in the detected state forever, regardless of future tag values. Some registers depend on the "dataDetected" signal to work in conjunction with other control signals to enable storage of particular values only when valid processing occurs. For instance, the tap accumulation registers do not register anything unless the proper state machine enable signal logically ANDed with the "dataDetected" signal has been asserted.

4.1.4 Coefficient Loading State Machine

Each processing element must provide a complex coefficient for each tap. The coefficients are multiplexed as operand B to the multiplier to be multiplied with the input data. Since operand B to the multiplier must be issued in a special format as described in Section 0, different parts of each coefficient must be interleaved to form the operand value. One approach to the problem uses twelve 4-to-1 multiplexers to build the mantissa part of the B operand from the four registered 32-bit coefficients read from memory. Another approach involves pre-processing of the coefficients such that they can be read directly from memory and issued to the multiplier B operand. The design incorporates the second approach that not only simplifies the implementation but also reduces the amount of CLB logic and flip-flop consumption. Instead, a 3-bit counter continuously provides the eight memory addresses from which to read the B operand patterns. In addition, the state machine awaits the data detected signal and begins reading the memory contents for the multiplier operand. The coefficient state machine must also signal the multiplier operand loading state machine to begin accepting data from the input data stream pipeline delay.

4.1.5 Multiplier Operand Loading State Machine

The 32-bit floating-point multiplier must be shared between two filter taps in a single processing element. For each new complex data value coming in on the input bus, the multiplier performs four calculations for the first tap (tap 0) and another four for the next tap (tap 1). The multiplier A operand multiplexes between the current input data stream and a delayed version of the input data stream. The multiplier B operand accepts data directly from the memory data register. The control and select signals to synchronize

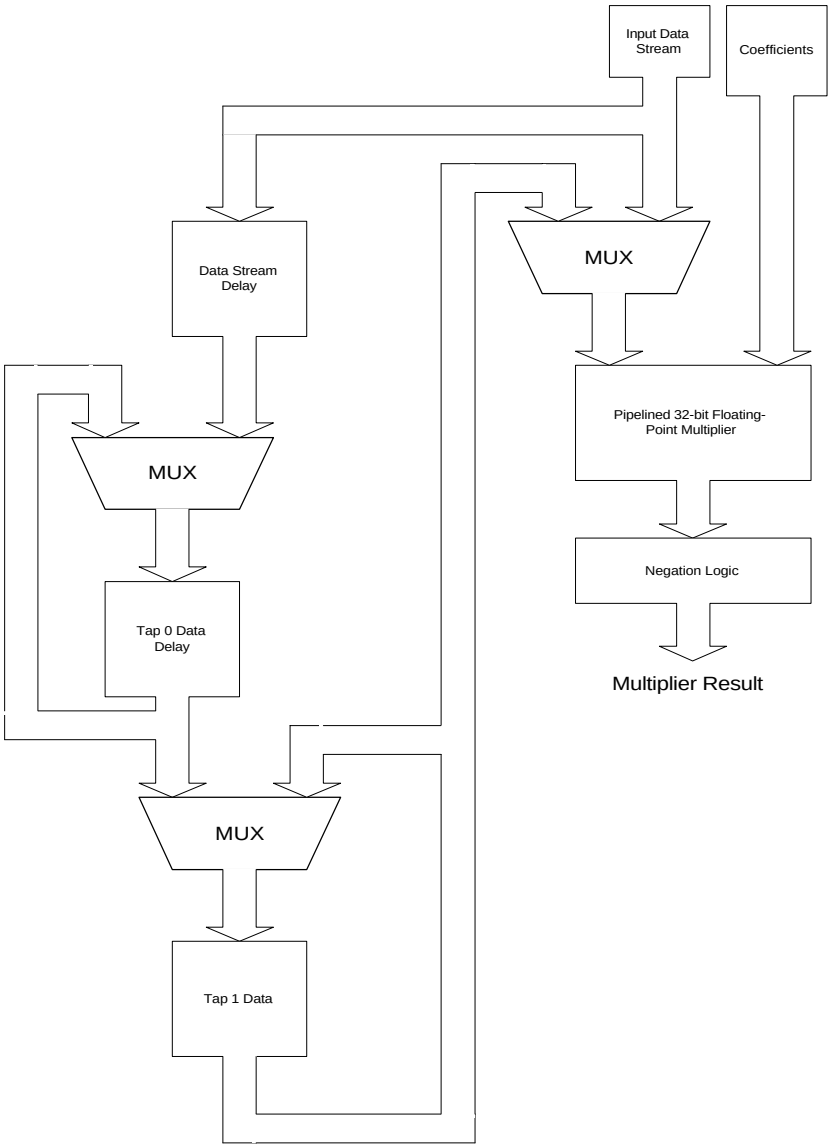

Figure 16: Multiplier component interconnections.

the data correctly between the two multiplier operand sources comes from the multiplier operand loading state machine.

The state machine must provide the control signals at precise moments to synchronize data to the multiplier. In addition, the state machine enables the adder operand loading state machine to begin accumulation. When multiplication for the first tap occurs, the control logic selects the data stream input values to be multiplied with the incoming coefficients being read from memory. The coefficient loading state machine enables the multiplier operand loading state machine at precisely the right moment for synchronization. As the values come in from the input data bus, the four values not only go into the multiplier but also into a pipeline delay component for the second tap. In addition to this, another set of delay elements holding the current data values for the second tap are re-circulated until the state machine is ready to perform the multiplication for the second tap (see Figure 15). When the state machine finishes with the first tap calculations, the second tap data values are selected from the multiplexer. During these states, the re-circulation feedback paths to the second tap pipeline delays are no longer used but rather accept data from the previous pipeline delays. The two pipeline delays can be seen in Figure 16 above as the “Tap 0 Data Delay” and “Tap 1 Data” blocks. The state machine controls the two input multiplexers to each of these pipeline delay elements. Aside from data flow control to the multiplier input, the state machine controls the negation logic, as seen in Figure 16 above, accepting the pipelined 32-bit floating-point multiplier output. On certain states, the negation logic inverts the sign bit to handle multiplication of two imaginary values.

4.1.6 Adder Operand Loading State Machine

The accumulation state machine to issue operands to the adder must provide selection for two multiplexers. Operand A may be selected between different multiplier result pipeline delay outputs or directly from the multiplier output. Operand B may be selected directly from the input data stream, either of the two tap 0 accumulation registers, or possibly a feedback path from the adder output. Pipeline delay mechanisms and feedback paths provide appropriate synchronization for partial real and imaginary

summations. The adder has an eight cycle latency which prevents results coming out of the multiplier to be correctly summed within a single pass through the adder. Hence, multiplier result pipeline delays of eight and sixteen hold necessary multiplier results until needed by the adder. Complete accumulation for a particular complex data input value requires three passes through the pipelined adder. The following table outlines the eight states of the adder operand loading state machine.

Table 2: Adder Operand Data Path Selections.

State	Operand A	Operand B
0	Multiplier Output Direct	Input Data Stream
1	Multiplier Output Delayed 8	Adder Output Feedback
2	Multiplier Output Direct	Input Data Stream
3	Multiplier Output Delayed 8	Adder Output Feedback
4	Multiplier Output Delayed 8	Tap 0 Re(Acc) Register
5	Multiplier Output Delayed 16	Adder Output Feedback
6	Multiplier Output Delayed 8	Tap 0 Im(Acc) Register
7	Multiplier Output Delayed 16	Adder Output Feedback

The adder operations in the state machine split the accumulations between the two taps where the first four operations are associated with tap 0 and the second four associated with tap 1. Real and imaginary results exiting the multiplier must be summed accordingly with the real and imaginary accumulation values coming from the input data stream (which are essentially accumulations from previous taps). The multiplier performs the following calculations for t=0:

For Tap 0:

$Re(D0) \times Re(C0) = Re_0(0)$
$Im(D0) \times Im(C0) = Re_1(0)$
$Im(D0) \times Re(C0) = Im_0(0)$
$Re(D0) \times Im(C0) = Im_1(0)$

For Tap 1:

$Re(D1) \times Re(C1) = Re_2(0)$
$Im(D1) \times Im(C1) = Re_3(0)$
$Im(D1) \times Re(C1) = Im_2(0)$
$Re(D1) \times Im(C1) = Im_3(0)$

where D0 is the complex data value to be multiplied with the Tap 0 coefficients producing two real and two imaginary values, and D1 is the complex data value to be multiplied with the Tap 1 coefficients producing two real and two imaginary values. The processing element must accumulate the real parts and imaginary parts of the Tap 0 products with an incoming complex accumulation value made up of Re(Acc) and Im(Acc). During the same computation phase, the processing element also begins a similar accumulation for the Tap 1 products. The primary difference between the two accumulation procedures is that Tap 0 accumulates with data coming from the input data stream while tap 1 accumulates with the previous tap 0 accumulation values held in registers (Tap0 Re(Acc) and Tap0 Im(Acc) registers as shown in Figure 15). Thus, the accumulation equations for the adder become:

For Tap 0:

$Re_0(0) + Re(Acc) = Re_X(0)$
$Re_1(0) + Re_X(0) = Re0(Acc)$ (registered)
$Im_0(0) + Im(Acc) = Im_X(0)$
$Im_1(0) + Im_X(0) = Im0(Acc)$ (registered)

For Tap 1:

$Re_2(0) + Re(Acc) = Re_Y(0)$
$Re_3(0) + Re_Y(0) = Re1(Acc)$ (registered)
$Im_2(0) + Im(Acc) = Im_Y(0)$
$Im_3(0) + Im_Y(0) = Im1(Acc)$ (registered)

The accumulation procedure essentially follows the above sequence of additions to produce the intermediate and final accumulation values. Since the adder has an eight cycle latency, the intermediate summations cannot be used immediately as implied by the equations above. Instead, pipeline delays hold off certain input values for either eight or sixteen clock cycles as shown in Figure 15 by the two delay elements following the multiplier. For instance, the $Re_1(0)$ value needs to be added to the intermediate summation of $Re_X(0)$. Since the result does not appear until eight clock cycles later, the $Re_1(0)$ multiplier output must be delayed. Table 2 above describes where each state must obtain its input to correctly accumulate the final tap 1 values, Re1(Acc) and Im1(Acc). The adder latency prevents accumulation calculations to be interleaved and produces a larger

latency due to the addition of intermediate results. Since some multiplier results must be delayed as many as sixteen cycles, the latency turns out to be 24 clock cycles from the introduction of the first two adder operands.

4.1.7 Processing Element Output Stage

The processing element must interleave the data values with the accumulation values, as similarly shown in Figure 14. Rather than zeros, between the two sets of data values, accumulation values are inserted. The input data stream passes through a cascade of pipeline delay elements such that the accumulation values are synchronized for the following processing element. The input data stream actually follows the accumulation stream by 24 clock cycles - the latency of the accumulation. Rather than create another state machine, the multiplier operand loading state machine suffices to issue the multiplexer select signals to build the output data stream.

4.2 MATLAB Filter Design Techniques

The MATLAB mathematical computation package in conjunction with digital signal processing toolboxes help generate coefficients for the filter. In addition, the program provides a means to help verify the design. Within the DSP toolbox, standard FIR filtering functions can be used to filter signals, generate FIR coefficients with or without windowing techniques, and graphically display the time-domain or frequency spectrum results.

The filter design meta-file allows for arbitrary length filters, selective frequency cutoff, and the ability to apply different windowing techniques provided by MATLAB which include Hamming, Hanning, Blackman, triangular, rectangular, and tapered rectangular. Once the type of filter has been selected with the few parameters needed, the program generates the filter coefficients and displays the frequency response. To display the filtering capabilities with the selected parameters, four summed sinusoids undergo filtering. The before and after filtered power spectrums have been plotted. Appendix B

illustrates the different type of filters the MATLAB meta-files can generate, which includes lowpass, highpass, bandpass, and bandstop filters.

The use of windowing functions to supplement the filter design reduce Gibbs phenomenon [4,6,24]. Gibbs phenomenon stems from the abrupt truncation of the infinite series described by the ideal impulse response [24]. When dealing with FIR filters, the length, or number of coefficients, used to represent the filter determines the effective cutoff of frequencies outside the passband. Hence, a longer filter results in a smaller transition band. Increasing the size of the FIR filter has drawbacks, which include increased delay and computational burden [6,24]. The use of windowing functions in the design techniques help reduce the width of the transition band and energy of the sidelobes for the shorter length filters constructed. In the previous sections of the chapter, the design can only fit two taps per processing element which comes to a total filter length of L= 8. Although the length may be short with a single board, larger FIR filters can be constructed as Section 5 discusses.

4.3 SIRCIM Channel Model Coefficient Generation

Coefficients to model wireless indoor channel impulse responses can be generated by a program developed by Ted Rappaport, Scott Seidel, Prabhakar Koushik, and Scott McCulley, called SIRCIM (Simulation of Indoor Radio Channel Impulse response Models) [10,15]. The program is based on measurements taken in indoor locations from which statistical models are developed [10]. The model data includes the number of distinct multi-path signals, the arrival times, and amplitudes of individual multi-path components [15]. The results of the program give detailed information about the channel model, including wide band path loss, RMS delay spread statistics, and temporal fading effects over time - all which can be plotted to be visually examined.

The channel model coefficients are computed based on several different user input parameters, such as carrier frequency, LOS (Line of Sight) or OBS (Obstructed Sight) visibility, transmitter-receiver separation, and even the type of partitioning the building

possesses which includes open, hard, or soft. The information found in [10], [11], and [15] give a better understanding of how the different parameters affect the channel impulse response. Therefore, depending on the designer's needs for modeling, SIRCIM offers the means to customize the channel according to the environment. The SIRCIM output files contain the necessary data which has the complex impulse response coefficients in magnitude-phase form, which can be translated into filter coefficient memory files. According to [15], SIRCIM could be used to provide the necessary data to drive hardware simulators. Hence, the FIR filter design on a CCM which supports complex computations may accelerate large simulations.

Chapter 5

FIR Filter Data Flow Design on CCMs

The implementation of the algorithm presented in Chapter 4 has been tailored for the WILDFORCE custom computing machine. On WILDFORCE as described in Section 0, five processing elements make up the processing array in which the algorithm has been partitioned over. The algorithm for the filter can be mapped onto any CCM that possesses a similar processing array architecture. Slight modifications to the implementation can be done to port the design over to other CCMs. Although the design attempts to stay away from data flow control based on timing constraints, injection of the data into the filter requires specific delay intervals which Section 5.1.2 covers. In conjunction with the hardware, the host software plays an important role in continuously issuing data to the input FIFO on the board while at the same time reading filtered data from the output FIFO. In addition, the host software control program is responsible for initializing the filter, setting up the clocks, and downloading the algorithm to the board. Despite having only four processing elements to represent the filter, Section 0 presents a method to increase the filter length by re-circulating the data through the filter and dynamically loading the filter coefficients.

5.1 Filter Data Flow Through the WILDFORCE Architecture

Although the processing elements have a pipelined adder and multiplier, the calculations cannot be completed so that a complex result occurs each clock cycle. The input stream from the data source issues the real part of the data value first, then the imaginary part. The filter must also share the data stream bus between the data to be filtered and the accumulation results as Figure 14 in Section 0 shows. Each complex value input consumes four data slots while accumulation values consume another four slots before repeating the same pattern for the next input data value and accumulation value. Thus, each new complex data value requires eight slots, or cycles, to represent a calculation cycle. Once valid data has been detected from the input source, the filter continuously receives data from the control processing element.

CPE0 reads data in from its FIFO and sends them to the first filter processing element by means of the crossbar in the case of the WILDFORCE implementation. Since the processing element expects a particular pattern for the data stream, CPE0 must format the data stream accordingly. The data stream pattern passed into the first processing element must issue the real and imaginary part of each data value twice followed by four zeros for accumulation data value place holders. Hence, CPE0 does not read data from the input FIFO every clock cycle. To mark the data as valid, the control processing element sets the tag bits which the filter processing elements recognize and acts upon. The control processing element continues such a cycle of operations until all the data has been read from the FIFO. The following sections describe the timing issues involved with injecting the data into the filter processing array and how the processing element state machines move the data through the WILDFORCE custom computing machine.

5.1.1 Data Flow Specifications

The number of data items to be filtered can be of any size and, therefore, is typically stored on disk or some other non-volatile storage media. The host software role includes

reading the data from the media and continuously sending it into the CCM input FIFO for filtering. The host needs to monitor the FIFO full and empty flags in order to regulate the input flow. The control processing element of the filter regulates the intake of the data into the FIFO, injecting data from the FIFO at a fixed rate. If the FIFO runs out of data and starves the filter, the end of the sequence simply finishes and does not get acknowledged by the control PE. Since the FIFO runs out of data, the read does take place from the FIFO and no valid data goes to the first processing element.

Essentially, two data streams flow through the filter: (1) the data values to be filtered, and (2) the accumulation of products from the multiplication of the filter coefficients and the data values. In order to make the proper accumulations of the products, the input data values associated with accumulation values going through the filter need to be delayed behind the accumulation values by exactly two computation cycles. Each processing element contains internal delay registers to provide the delay needed. Depending on the architecture of the CCM, the data flow from the control processing

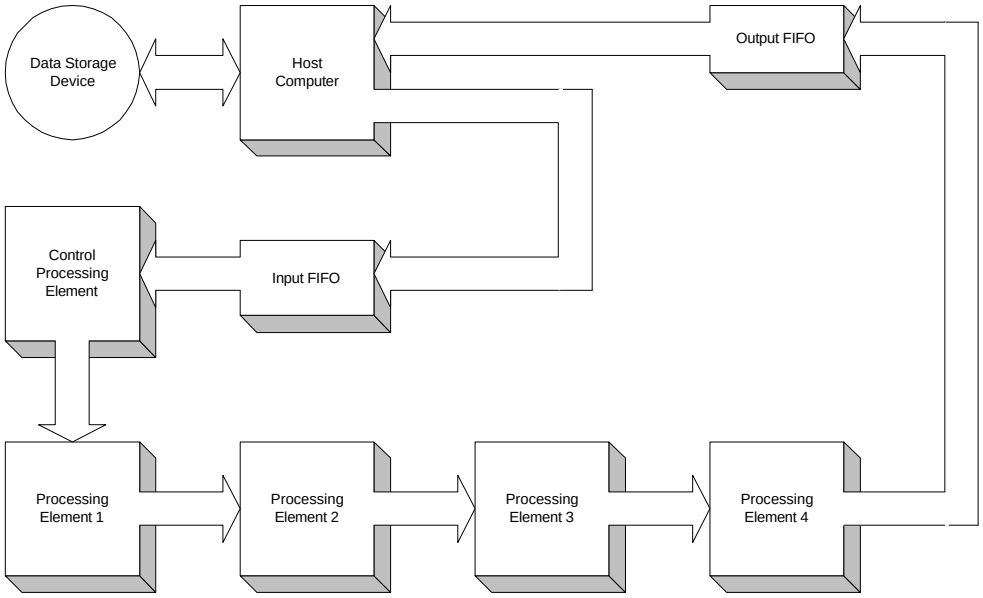

Figure 17: Data flow through WILDFORCE.

element may differ. On WILDFORCE, the data comes into the first processing element through the crossbar. After the data has gone through the internal taps within each processing element, the data and accumulation values get transferred out the right systolic array bus and into the neighboring PE. The above diagram in Figure 17 shows the flow of data coming from the storage media, through the host computer, through WILDFORCE, and back to the host computer to be possibly stored on the media. Should the filter need to be expanded with more processing elements, WILDFORCE and similar platforms allow the data to be sent to another board via a SIMD connector. The data can then flow into the processing array of another board. The following section provides more detail on the internal flow of the tap state machines. The last processing element in the array may either transfer both, the accumulation values and the data values, or just the accumulation values depending on whether or not the filter length is to be lengthened by re-circulation. The host software must also monitor the output FIFO in order to keep the FIFO from completely filling and storing the data to either a large memory buffer or non-volatile media for post-processing and analysis.

5.1.2 Data Flow Control within the Processing Element

The finite state machines within each processing element move the data from the input bus to several holding and delay registers. To ensure the data moves about correctly within the processing element, one state machine moves data into the processing element and another moves data out. The state machine that moves data into the processing element looks for a specific tag on the input data bus and continues to select different multiplexers to register the data coming in on the bus. The output shifting state machine not only shifts data out of the processing element but also moves data from the first internal tap to the next by means of delay registers. In the figure above, the data comes from the input data bus which on WILDFORCE could be the crossbar or left systolic bus. Holding registers within the PE keep the data values and running accumulation for the computing engine state machines. Data values enter the computation cycles through the adder and multiplier which are regulated by separate adder and multiplier state machines.

Data values that have to be filtered need to go through both the feedback path into the multiplier unit and also through the delay registers depending upon the state of the PE. Accumulation values coming from the adder need to go through the delay registers and if from the second tap of the PE, directly out to the right systolic bus. Feedback data paths to the adder allow accumulation of products from the complex multiplication to be performed using multiple passes.

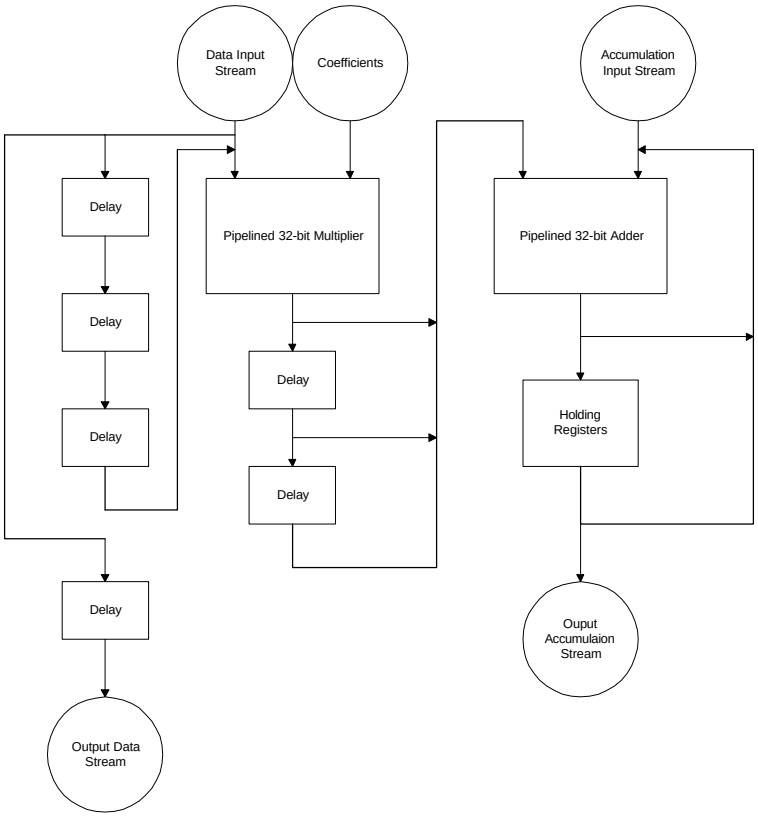

Figure 18: Processing element internal data flow paths.

5.2 Filter Architecture on Other CCMs

Although each PE has been customized for WILDFORCE, portability to other custom computing machines has not been excluded but do require similar capabilities. The

architecture must first possess a data path through an array of computational elements. Having such an architecture allows for the array to be either lengthened or shortened as needed. Handshake signals need to exist between each element to signal when valid data transfers from one PE to another in the array. In the case of WILDFORCE, the tag bits allow detection of valid data packets off the input data bus. To regulate data flow into the filter, a control processing element at the head of the computing array needs to be able to accept data and transfer the data with some indication of valid data packets. WILDFORCE again uses the tag bits to identify valid data packets. The last processing element in the array needs to have the capability to transfer data to the host for possible post-processing. Each last element on WILDFORCE allows the data to continue on to the next board for filtering through the SIMD connector or can place the results in an output FIFO for the host to read.

5.3 Variable Filter Lengths Using Re-circulation

The design presented implements two filter taps per processing element which severely limits the length of the filter and thus, may reduce the effectiveness of the filter. In order to increase the length of the filter, the results may be re-circulated through the filter multiple times. For each pass through the filter, new coefficients to represent the next consecutive taps must be loaded. Note that the results from the filter may already have the data stream interleaved with the accumulation values which only require that the stream be directly passed through the filter once the new coefficients have been loaded. Section 6 illustrates the differences between using filter lengths of 7 and 31 to perform lowpass, highpass, bandpass, and bandstop filters on similar signals. Higher order filters provide better frequency cutoff and lower sidelobes to minimize unwanted frequencies.

Chapter 6

Synthesis Results and Implementation Verification

This chapter covers the results from synthesizing the VHDL and from the verification of the implementation on a WILDFORCE board. The verification process includes filter comparisons between a 7-tap and 31-tap FIR filter. Each set of filter coefficients have also been convolved with a rectangular window. Error analysis plots have been included to compare the implemented floating-point format with the IEEE 754 floating-point format. In addition, performance numbers have been included in terms of MFLOPs based on the clock speed at which the filter may be executed.

6.1 VHDL Synthesis Results

Incremental construction and synthesis of the design help determine how much a particular resource consumes. Furthermore, an incremental approach alleviates design debugging and helps narrow the possibilities when errors occur during the implementation. In the process of creating the filter in this paper, the first few iterations from the VHDL coding step to synthesis did not utilize CLB resources efficiently and failed to make use of the global reset line. As refinements were made in the coding style, more efficient use of the logic blocks became apparent as the total usage of CLBs dropped by 15% in some cases. Registers and other elements in the design can be connected to the global reset by using the

asynchronous reset signal in the registered process as shown in Figure 19. Therefore, not all VHDL code produced the same results which depended greatly on the constructs and how they were used.

The filter used three different tap designs to incorporate FIFO input (Block A), FIFO output (Block C), and the processing elements in between (Block B). Figure 17 illustrates that Block A must receive data from the crossbar and pass the results onto the next processing element through the right systolic data bus. Block B receives the data and accumulation stream from the left systolic data bus and passes the data and accumulation

```
P_Reg : process( PE_Reset,
                 PE_Pclk )

    begin

        if ( PE_Reset = '1' ) then
            presentDetectState <= InitDetectState;
            presentCoeffLoadState <= InitLoadState;
            presentMultFeedState <= MultFeedState0;
            presentAdderFeedState <= AdderFeedState0;

            PE_MemAddr_OutReg( 21 downto 0 ) <= (others => '0');

            tap0ReAcc_Reg <= (others => '0');
            tap0ImAcc_Reg <= (others => '0');
            tap1ReAcc_Reg <= (others => '0');
            tap1ImAcc_Reg <= (others => '0');

            PE_Right_Out( 35 downto 0 ) <= (others => '0');

        elsif ( rising_edge( PE_Pclk ) ) then
```

Figure 19: Asynchronous global reset coding technique.

stream out the right systolic data bus. Block C receives data from the left systolic data bus and passes the results to the host through the output FIFO. Table 3 lists the final synthesis results of each different block (less the resources consumed by the 24-bit integer multiplier and several of the pipeline delay elements) after using Synplify's Synplicity tool to produce a Xilinx netlist file. Table 4 provides the actual resource consumption during the place-and-

route phase which may also include redundant or replicated logic to possibly meet timing specifications or reduce lengthy routing.

Table 3: VHDL synthesis results for a XC4036EX device.

Block A	34% of CLB FG function generators 14% of CLB H function generators 49% of CLB flip-flops
Block B	34% of CLB FG function generators 14% of CLB H function generators 49% of CLB flip-flops
Block C	33% of CLB FG function generators 13% of CLB H function generators 44% of CLB flip-flops

Table 4: Place and route results for a XC4036EX device.

Block A	98% CLB usage 69% of CLB flip-flops
Block B	98% CLB usage 69% of CLB flip-flops
Block C	88% CLB usage 65% of CLB flip-flops

Unfortunately, the consumption of CLBs does not always give a good indication of whether or not the processing element will both place and route on the chip. The place-and-route phase depends greatly on the tool being used to perform these operations. Efficient placement typically leads to efficient routing so that signals are not only routed correctly but routed so that delays do not affect the entire design performance. Specially constructed pipeline delay elements helped to reduce the number of 32-bit registers that would have been required to provide the necessary delays between operations in the design. The delay elements took advantage of the CLB RAM resources rather than consuming flip-flops. The RAM resources within the CLB allowed for up to 16 different bit values to be stored at any one time. By stacking 32 CLBs in such a manner allows for up to 16 delay registers. Constructing the same number of 32-bit registers using VHDL synthesis would

consume 512 flip-flops. A combination of such techniques may be required to alleviate resource consumption.

6.2 Results Verification Using MATLAB

MATLAB is a software mathematical computing environment for numeric computation and visualization [29]. The environment allows users to enter problem specifications through a command-line interface or scripts called meta-files. MATLAB possesses its own language constructs in order to build meta-files. In addition to computation, built-in functions allow users to graphically represent data through charts, plots, or graphs. With additional libraries, signal processing support functions for filtering and spectrum analysis can be done. MATLAB has been used to quickly generate coefficients using techniques described in [24]. Meta-files provided by [24] allow users to select the type of filter with a variation from six different windowing functions. As a baseline, the filter coefficients are convolved with a rectangular window. To verify the implementation on WILDFORCE, four types of filters have been designed in MATLAB including a lowpass, highpass, bandpass, and bandstop filter.

6.2.1 Filter Verification with Real Numbers

One part of the verification exhibits the capability of the filter to process values using real numbers. Since the filter accepts complex values, the imaginary part of the data can be zeroed. The following section covers the bandpass filter verification. Appendix B provides similar plots for the other filter types. Filter lengths of 7 and 31 shall be used to illustrate differences and trade-offs between single and multiple passes through the filter. The input signal to pass through each filter has been generated with the addition of four sinusoids at different frequencies. The frequencies have been chosen to show not only the distinct properties of each type of filter but the effectiveness according to their lengths. Figure 20 shows a spectrum plot of the signal frequencies.

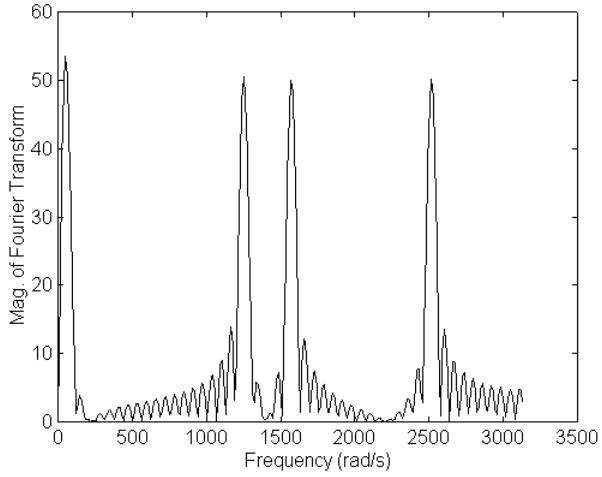

Figure 20: Frequency spectrum of input signal.

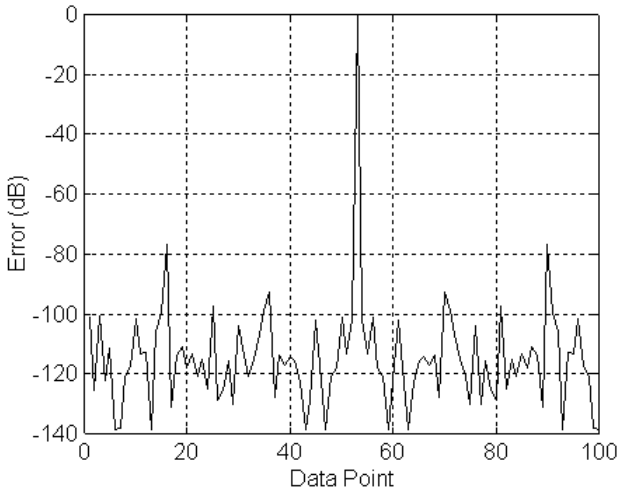

Figure 21: Filter of length 7 error analysis.

The first part of the verification involves an error analysis between MATLAB and WILDFORCE executions. Data obtained from WILDFORCE are compared against the calculated IEEE floating-point values by MATLAB. The **conv()** or **filter()** function provides the same filtering operation performed on the WILDFORCE board. Figure 21 and Figure 22 provide the error in decibels for the 7-tap and 31-tap filters, respectively. The

average error stays well below the -100 dB range for both filters. Peak errors depicted in the analysis plots indicate data points where the expected and actual values equaled zero. For the 31-tap filter, re-circulation of the output may compound the error introduced in each consecutive pass through the four processing elements. The mean error for the 7-tap filter is approximately -122 dB and -118 dB for the 31-tap. With the anomalies excluded, peak errors of -78 dB and -59 dB exist for the 7-tap and 31-tap filters, respectively. The average error comes out to be about 0.0001% for both lengths. Table 5 provides mean errors for the other filter types found in Appendix B.

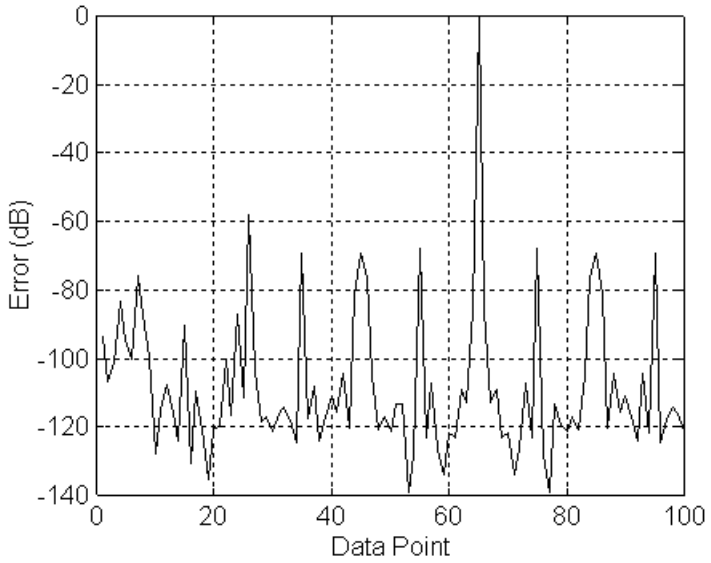

Figure 22: Filter of length 31 error analysis.

Table 5: Mean filter error comparisons with IEEE 754 standard.

Filter Type	Filter Length 7 Error	Filter Length 31 Error
Lowpass	-124 dB	-114 dB
Highpass	-125 dB	-114 dB
Bandpass	-122 dB	-118 dB
Bandstop	-123 dB	-115 dB

The second part of the verification exhibits the capability to expand the filter to an arbitrary length by simply re-circulating the output from the systolic array. As Figure 17 shows from Section 0, the FIFO output going to the host may be issued back to the input FIFO once the new filter coefficients have been loaded. This technique has been done to create filter lengths greater than the eight provided through the four processing elements. Filter lengths may also be extended by cascading several boards but will not be as cost effective as re-circulating the output. The trade-off poses the typical cost-performance situation. By reducing on the number of boards, the host must intervene in order to reload new coefficients and re-issue the data to the filter, whereas cascading several boards for one large filter allows for possible real-time applications of the filter. Higher order filters affect several aspects of the filter, including the sidelobes, the roundness of the cutoff, and the transition band of the frequency response. Appendix B illustrates the different filter frequency responses between 7-tap and 31-tap filters. Figure 23 shows some of the different characteristics of the two filters. In the frequency response plot, notice the considerable gain on the left sidelobe of the 7-tap filter which allows substantial amounts of energy to pass for the lower frequencies outside the cutoff.

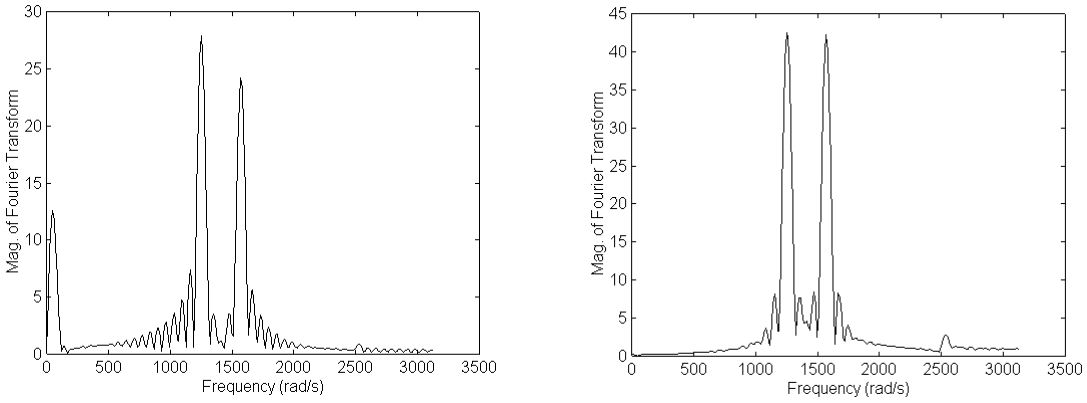

Figure 23: Spectrum plot of bandpass filtering for 7 taps (left) and 31 taps (right).

The typical cost-performance trade-off determines whether to use a lower order filter with speed or a higher order filter with significant overhead processing. Lower order filters provide simplicity and may be enhanced to an extent through the use of different windowing techniques. If the signal possesses large gaps between different frequencies, lower order filtering may be adequate. Applications requiring a more narrowband selection of frequencies use larger order filters to achieve proper signal filtering. The overhead processing associated with large data transfers may be alleviated using host and bus-master DMA channels. In addition, high-priority interrupt service routines may handle new coefficient loading for the filter. Such host processing techniques may reduce the turn around time to get the data back into the filter for multiple passes.

6.2.2 Filter Verification with Complex Numbers

One of the motivating ideas behind this thesis includes a means to perform channel model simulations in conjunction with SIRCIM as described in Section 0. SIRCIM generated data may be used to provide the necessary complex valued coefficients to represent different channel mediums for wireless indoor radio simulations. The verification

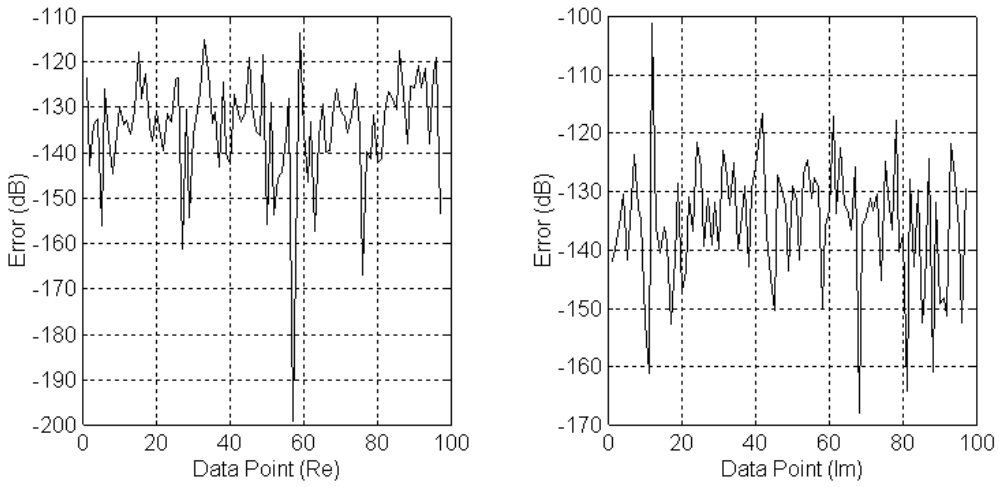

Figure 24: Spectrum plot of bandpass filtering for 7 taps (left) and 31 taps (right).

filtering shown in Figure 24 illustrates the error between complex convolution provided by MATLAB and the same execution done on WILDFORCE using the same complex coefficients and input data stream through a 32-tap filter. The average error in decibels turns out to be about the same for both at -133 dB.

6.2.3 Filter Performance on WILDFORCE

The filter implementation incorporates a fully pipelined design which utilizes the floating-point multiplier and adder units within each processing element to the fullest. With each ALU producing a floating-point result each clock cycle, and the clock running at 20 MHz, each processing element achieves **40** MFLOPs/s. **160** MFLOPs/s can be achieved with a single WILDFORCE board having four processing elements in the systolic array. Cascading four WILDFORCE boards in a single host can achieve **640** MFLOPs/s for the system.

Re-circulating data through the filter allows higher order to filters to be achieved but at the expense of overhead processing done by the host. For each pass through the filter, new coefficients need to be loaded into the filter memories which require the host control program to halt processing, write to the PE local memories, and start the filter again. By far, loading new coefficients consume the most time during the overhead processing. As the number of filter taps increase, the overall performance degrades linearly since the time to load coefficients remains the same for each pass through the filter. The time to write the coefficients to memory for a single processing element takes roughly 8.5 milliseconds. Additional overhead expense to reset and start up the filter again is about 6.5 milliseconds. Therefore, the following equation can be used to roughly figure out the overhead for re-circulation:

$$((8.5x)y + 6.5y)z = \text{total overhead in milliseconds} \quad (6.1)$$

where x is the number of PEs on the board, y is the number of iterations, and z is the number of boards in the host. The overhead calculated here accounts for the down-processing time that the input signal is not being filtered. Hence, another important factor includes the length of the data stream. For instance, if the filter executes at 20 MHz and the

host needs to perform a re-circulation every second, the effective PE MFLOPs/s rating would drop from 40 MFLOPs/s to 36.76 MFLOPs/s. But as the size of the input stream increases to where the overhead occurs once every 10 seconds, the effective computing rate becomes 39.67 MFLOPs/s. From the empirical overhead data of Equation 6.1, the effective MFLOP rating for higher filters can be quantified as shown in Figure 25. The plot illustrates the MFLOP rating change as the size of the input data stream increases in length. As the length increases, the distance in the time between re-circulation overhead occurrences increases. The number of times the host code must re-circulate the data is fixed depending on the size of the filter.

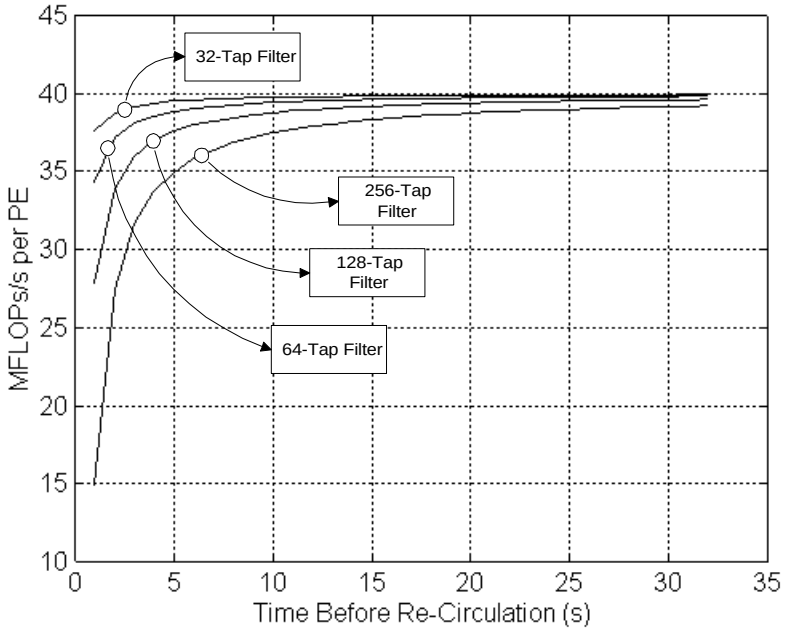

Figure 25: Effective filter performance with data re-circulation.

Chapter 7

Conclusions

This thesis has been motivated by finding ways to accelerate and off-load the computing burden of complex FIR filtering to hardware. Thirty-two bit floating-point representations attempt to make the FIR filter more feasible for simulation purposes and possibly real-time applications. The representation and implementation provides another means of 32-bit floating-point computing on re-configurable platforms and may contribute to future continuing work. The following few sections discuss possible continuing work on the filter to perhaps improve or utilize its current capabilities, describe some of the limitations the filter implementation encountered, and finally, presents a reflection on the results of the work presented.

7.1 Suggestions for Future Work

The current design of the FIR filter utilizes the floating-point multiplier and adder units within each processing element every clock cycle. The current pipelined adder requires 14% more resources than the multiplier according to the synthesis results and approximately 26% more routing resources according to the place-and-route tools. Possibly a more efficient pipelined adder may reduce the size and complexity of the design to fit into smaller FPGAs and reduce the cost. Also, with a shorter pipeline, the adder may perform all the necessary additions in a shorter time frame, thus reducing the size of pipeline delays associated with feedback to the adder - another means to lessen the cost. Since the design

is fully pipelined, performance increases would mean increasing the clock rate which may lead to simpler, yet faster components and data paths within the processing element. For example, an early design trade-off issues pre-processed coefficient patterns to the filter rather than using multiplexers to select from registers. Tremendous savings were made in CLB resources and even more importantly, routing resources. Similar design trade-offs that can be moved off-chip and pre-processed for the filter may help performance and resource consumption.

Aside from the performance work that can continue, applications that use the filter can be constructed with the use of external I/O interfaces. WILDFORCE provides a means to send and receive data to and from the PE FIFOs directly. With such connections, properly sampled digital data can be fed directly into the filter and directly out of the filter to provide real-time filtering depending on the sampling rate. Near future capabilities to WILDFORCE allow higher data transfers to and from the board which provide better performance measures for host driven applications. Software programs that need to filter stored data can use the CCM filter to perform intensive computations with little or no host processor intervention through the use of DMA and multi-threading techniques.

7.2 Design Limitations

Limitations exist for the design, which impose different design strategies to compensate. The challenge of constructing the filter not only lies in the design speed but also in a design that will fit into low cost FPGAs. Limited number of CLBs and routing resources bring about two of the foremost concerns when developing with FPGAs. Fast but inefficient VHDL coded arithmetic units consume vast amounts of logic and routing resources which may possibly limit the amount of supplemental control logic. Even efficiently coded VHDL may result in unwanted, large arithmetic units. In turn, fewer taps may be designed into each processing element. In this particular thesis, larger arithmetic units which may result in shorter pipelines have been traded-off for longer pipeline latencies and lesser resource consumption. Providing simpler logic stages and a longer pipeline, the multiplier may still achieve feasible calculation speeds. Automated synthesis tools may not

always produce the most optimized results and therefore, designers may have to manually optimize certain areas. For example, the integer multiplier provided by Annapolis Micro Systems, was not generated by VHDL or synthesis but from C code that directly generated a Xilinx netlist formatted file to be used directly by the place-and-route tool. Similar techniques to generate the pipeline delay units had to be done. For simulation purposes, models had to be created to reflect the exact timing and results for each of these entities. In other cases, careful VHDL coding techniques may be used to produce the desired results to help reduce resource consumption and possibly increase operating speeds. The floating-point adder that uses two shifters for the mantissa denormalization and normalization originally incorporated variables in the implementation. Instead, a VHDL coding technique using signals rather than variables to implement a barrel shifter has been conceived. Signal usage eliminated more than 42% of the routing according to the place-and-route tool when performing a direct comparison of the two different types of implementations.

7.3 Conclusions on the Work

The design and implementation meet the primary FIR filter goals which include 32-bit floating-point multiplication and addition with a scaleable filter architecture for real or complex filtering. The filter can be enhanced by simply adding more processing elements to the CCM computing array or by re-circulating the filter results. In addition, coefficient loading into local memories provide dynamic filter changes without having to re-program the processing elements. With larger FPGAs that have recently become available, better implementations can be done to help improve routing and provide more logic for the arithmetic units to help shorten the pipeline delays. The current pipeline design must interleave the input data stream with the accumulation stream which limits the input bandwidth by half. Future versions on larger FPGAs may introduce 64-bit data paths to allow both streams to flow contiguously or possibly send the real and imaginary parts of each value simultaneously where either one would increase the bandwidth. CCMs provide such flexibility for DSP in that the hardware can be tailored precisely to the algorithm from

configurable logic. Whereas with DSPs, the algorithm must be fitted to a generic set of fixed resources.

Appendix A

Filter Processing Element Finite State Machines

The design of the processing element architecture has been created such that it may be replicated across all FPGAs in the systolic array. Subtle differences to adjust for different I/O buses may have to be done but the basic core set of state machines and control flow remain the same across all processing elements. The following appendix provides some diagrams to supplement Section 0 on the state machines that control the data flow throughout each processing element. The four state machines include:

- Data Detection State Machine
- Coefficient Loading State Machine
- Multiplier Operand Loading State Machine
- Adder Operand Loading State Machine

A.1 Data Detection State Machine

Table 6: Data detection FSM states.

State	Description	Output(s)
0	The FSM starts in this state during a reset and continues to stay in this state until the correct tag word has been identified on the input bus. The tagDetected flag indicates a valid tag has been found on the input bus and continues to the next state.	dataDetected = 0
1	After detecting valid data on the input bus, the FSM asserts the dataDetected signal to notify the rest of the control that the processing element should begin processing the incoming data.	dataDetected = 1

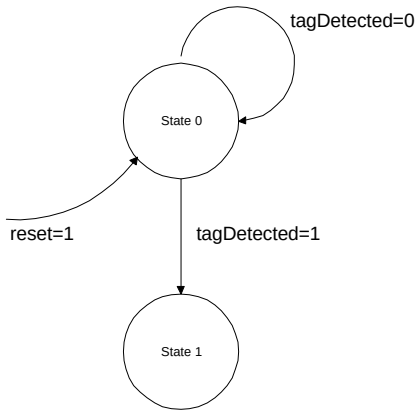

Figure 26: Data detection finite state machine.

A.2 Coefficient Loading State Machine

Table 7: Coefficient loading FSM states.

State	Description	Output(s)
0	When the processing element resets, the FSM returns to this state. Continue to request for the memory bus from the DPMC. Continue to the next state if we have a memory bus grant and valid data has been detected.	multFeed = 0 memBusReq = 1 memStrobe = 1
1	The FSM has been granted access to the local memory by the DPMC. Once a read has been initiated, the data should arrive two clock cycles later. Continue on into the next state.	multFeed = 0 memBusReq = 1 memStrobe = 1
2	Wait state for a memory read. Continue on into the next state.	MultFeed = 0 memBusReq = 1 memStrobe = 1
3	The FSM remains in this state until a reset on the processing element occurs. Continuous memory reads are being performed to read the coefficient data. The multiplier operand loading state machine should be engaged.	MultFeed = 1 memBusReq = 1 memStrobe = 1

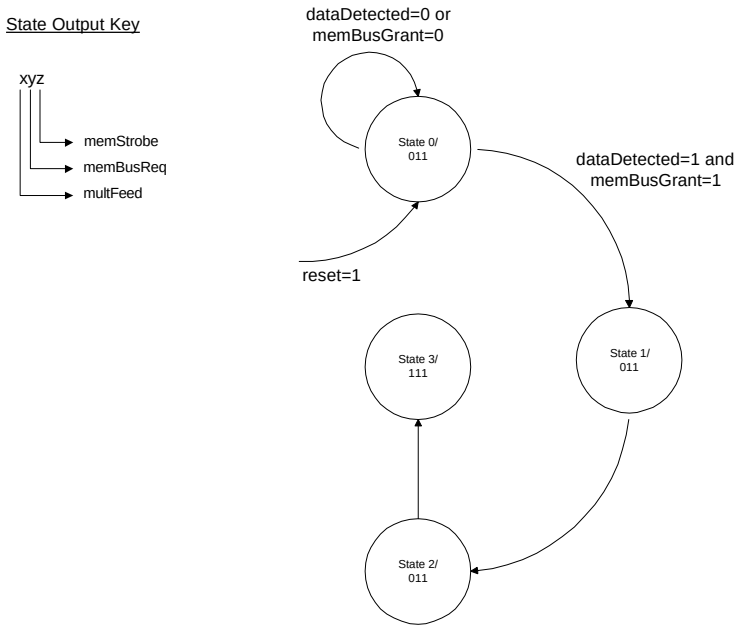

Figure 27: Coefficient loading finite state machine.

A.3 Multiplier Operand Loading State Machine

Table 8: Multiplier coefficient loading FSM states.

State	Description	Output(s)
0	Upon a processing element reset, the FSM enters and continues to stay in the state until the coefficient loading FSM asserts the multFeed signal.	tapLoad_en = 0 negateMultResult = 0 startAdderFsm = 0 outSelects = 00
1	Selecting the tap 1 imaginary register value output enable. Loading multiplier with data directly from input bus and memory coefficients.	tapLoad_en = 0 negateMultResult = 0 startAdderFsm = 0 outSelects = 01
2	Selecting data stream delay as the output source and loading the multiplier with data directly from the input bus and memory coefficients.	tapLoad_en = 0 negateMultResult = 0 startAdderFsm = 0 outSelects = 11
3	Selecting data stream delay as the output source and loading the multiplier with data directly from the input bus and memory coefficients. Output from the multiplier should be negated.	tapLoad_en = 0 negateMultResult = 1 startAdderFsm = 0 outSelects = 11
4	Selecting data stream delay as the output source and loading the multiplier with data from the tap 1 delay line and memory coefficients. Not negating the multiplier output anymore.	tapLoad_en = 1 negateMultResult = 0 startAdderFsm = 0 outSelects = 11
5	Selecting data stream delay as the output source and loading the multiplier with data from the tap 1 delay line and memory coefficients.	tapLoad_en = 1 negateMultResult = 0 startAdderFsm = 0 outSelects = 11
6	Selecting tap 1 Re(Acc) component as the output source and loading the multiplier with data from the tap 1 delay line and memory coefficients.	tapLoad_en = 1 negateMultResult = 0 startAdderFsm = 0 outSelects = 10
7	Selecting data stream delay as the output source and loading the multiplier with data from the tap 1 delay line and memory coefficients. Not negating the multiplier output anymore.	tapLoad_en = 1 negateMultResult = 0 startAdderFsm = 0 outSelects = 10

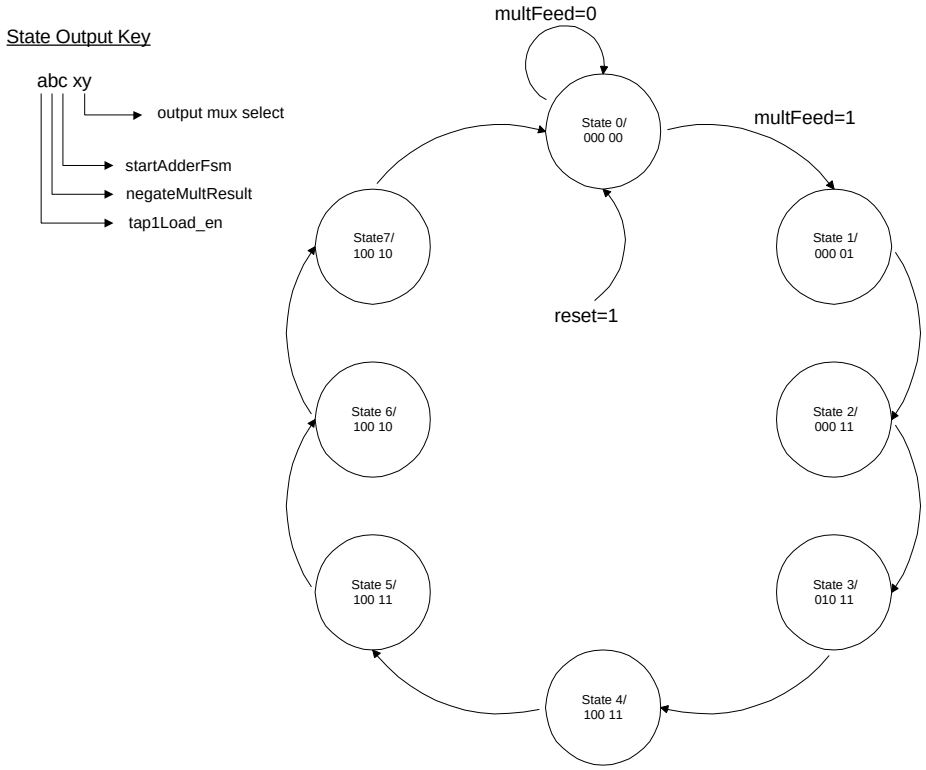

Figure 28: Multiplier loading finite state machine.

A.4 Adder Operand Loading State Machine

Table 9: Adder operand loading FSM states.

State	Description	Output(s)
0	Upon a processing element reset, the FSM enters and continues to stay in the state until the multiplier operand loading FSM asserts the adderFeed_en signal. The mux selectors currently choose the immediate multiplier result and the input data stream. The state registers the adder output into the tap 1 Im(Acc) holding register.	tap0ReAcc_en = 0 tap0ImAcc_en = 0 tap1ReAcc_en = 0 tap1ImAcc_en = 1 muxAddOpA = 00 muxAddOpB = 00
1	The FSM selects the delayed by 8 multiplier result and the feedback path from the adder output as the two operands. No adder results are registered.	tap0ReAcc_en = 0 tap0ImAcc_en = 0 tap1ReAcc_en = 0 tap1ImAcc_en = 0 muxAddOpA = 01 muxAddOpB = 01
2	The FSM selects the immediate multiplier result and the input data stream value as the two values to add. The tap 0 Re(Acc) value should be ready during this adder cycle and needs to be registered.	tap0ReAcc_en = 1 tap0ImAcc_en = 0 tap1ReAcc_en = 0 tap1ImAcc_en = 0 muxAddOpA = 00 muxAddOpB = 00
3	The FSM selects the delayed by 8 multiplier result and the feedback path from the adder output as the two operands. No adder results are registered.	tap0ReAcc_en = 0 tap0ImAcc_en = 0 tap1ReAcc_en = 0 tap1ImAcc_en = 0 muxAddOpA = 01 muxAddOpB = 01
4	The FSM selects the delayed by 8 multiplier result and the tap 0 Re(Acc) register value to be added. During this state, the tap 0 Im(Acc) value needs to be registered.	tap0ReAcc_en = 0 tap0ImAcc_en = 1 tap1ReAcc_en = 0 tap1ImAcc_en = 0 muxAddOpA = 01 muxAddOpB = 10
5	The FSM selects the delayed by 16 multiplier result and the feedback path from the adder result as the two operands. No adder results are registered.	tap0ReAcc_en = 0 tap0ImAcc_en = 0 tap1ReAcc_en = 0 tap1ImAcc_en = 0 muxAddOpA = 10 muxAddOpB = 01
6	The FSM selects the delayed by 8 multiplier result	tap0ReAcc_en = 0

State	Description	Output(s)
	and the tap 0 Im(Acc) register value to be added. The tap 1 Re(Acc) value needs to be registered.	tap0ImAcc_en = 0 tap1ReAcc_en = 1 tap1ImAcc_en = 0 muxAddOpA = 01 muxAddOpB = 11
7	The FSM selects the delayed by 16 multiplier result and the feedback path from the adder as the two operands. No adder results need to be registered at this time.	tap0ReAcc_en = 0 tap0ImAcc_en = 0 tap1ReAcc_en = 0 tap1ImAcc_en = 0 muxAddOpA = 10 muxAddOpB = 01

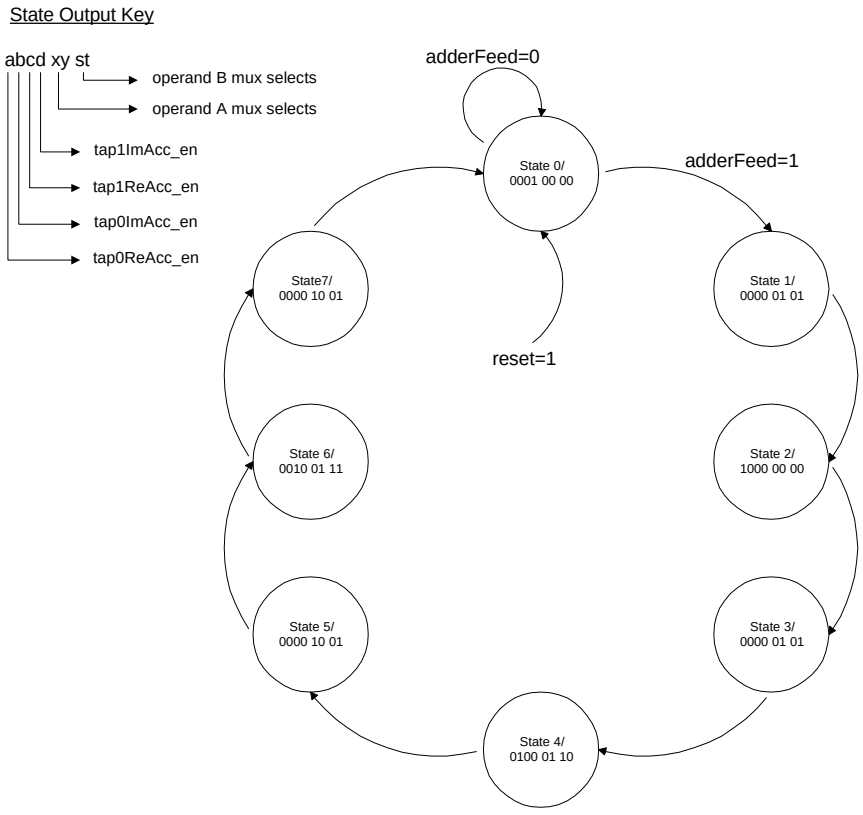

Figure 29: Adder operand feeding finite state machine.

Appendix B

Real Number Values Filter Verification

The real number values verification not only included the bandpass filter presented in 0 but also included 7-tap and 31-tap lowpass, highpass, and bandstop filters as well. Each of the filter frequency responses, filtered output signal, and error analysis plots have been included for further comparisons. The input signal includes a mix of four different sinusoids having the following frequencies:

- 62.83 rads/s (10 Hz)
- 1256.63 rads/s (200 Hz)
- 1570.79 rads/s (250 Hz)
- 2513.27 rads/s (400 Hz).

B.1 Lowpass Filter Verification

The frequency responses significantly differ between the two filter lengths where the 7-tap filter allows considerable energy for unwanted frequencies to pass through. Whereas the 31-tap filter provides a sharper roll-off at the cutoff frequency and possesses sidelobes lower than the -20 dB mark. Figure 31 illustrates the results of failing to provide a narrower transition band. Energy from the 250 Hz sinusoid were allowed to pass through the filter whereas the 31-tap filter completely eliminates the frequency.

Figure 32 shows the error comparison between the two different length filters which continue to stay well below the -100 dB mark.

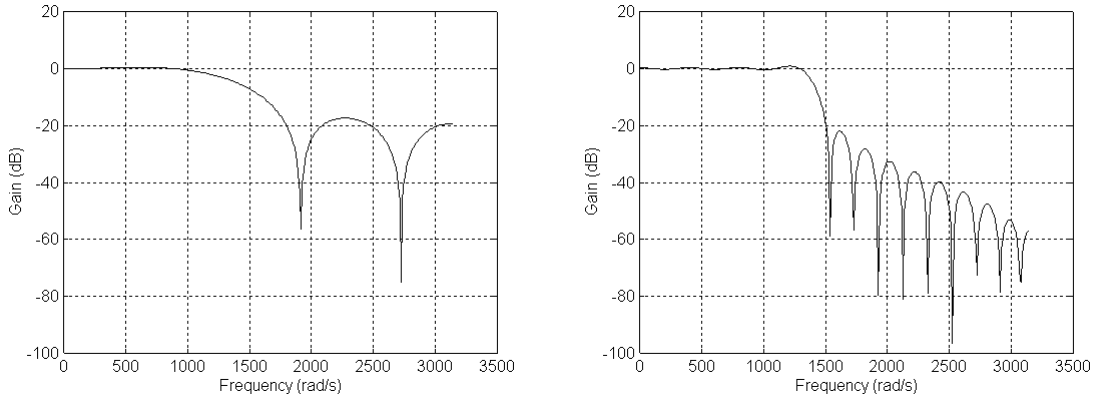

Figure 30: Lowpass filter frequency responses for 7-tap (left) and 31-tap (right) filters with a target cutoff frequency of 1413.7 rad/s (225 Hz).

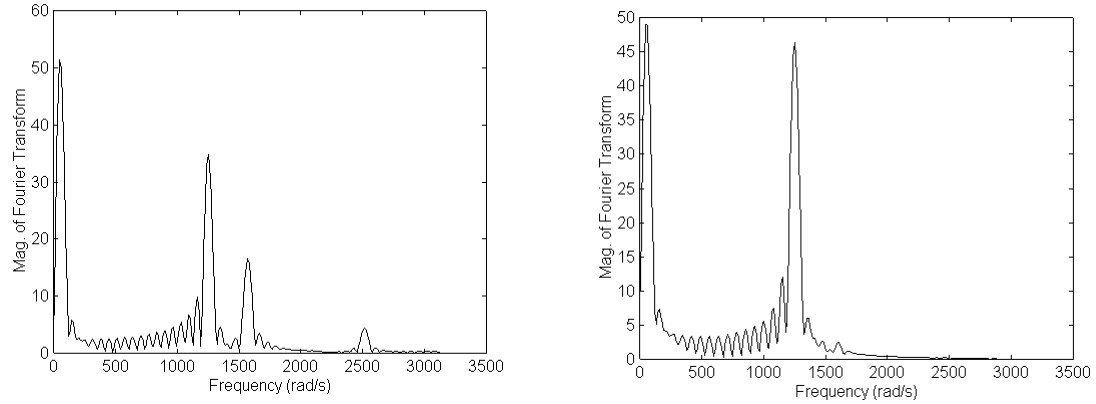

Figure 31: Lowpass filtered signal spectrums for 7-tap (left) and 31-tap (right) filters with a target cutoff frequency of 1413.7 rad/s (225 Hz).

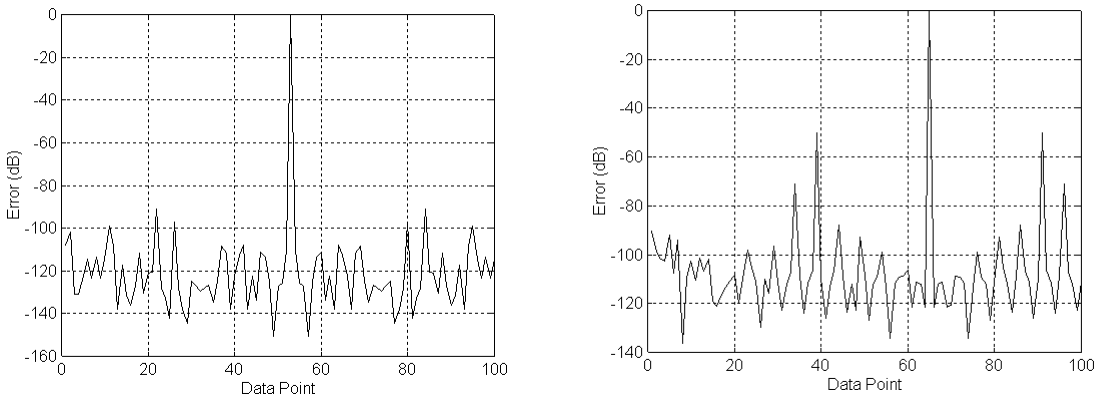

Figure 32: Lowpass filter output error analysis for 7-tap (left) and 31-tap (right) filters.

B.2 Highpass Filter Verification

The highpass filter design simply reverses the allowable frequencies from those found in the lowpass filter above. Using the same cutoff frequency, the lower sidelobes of the 7-tap filter appear to stay well below the -20 dB mark but still does not provide a sharp enough roll-off for the transition band. Thus, energy from unwanted frequencies continue to pass through the filter. The higher order filter provides a better roll-off for the transition band which performs the proper filtering out of the neighboring 200 Hz sinusoid component of the signal. The error for each length of the FIR filter continues to operate below the -100 dB mark.

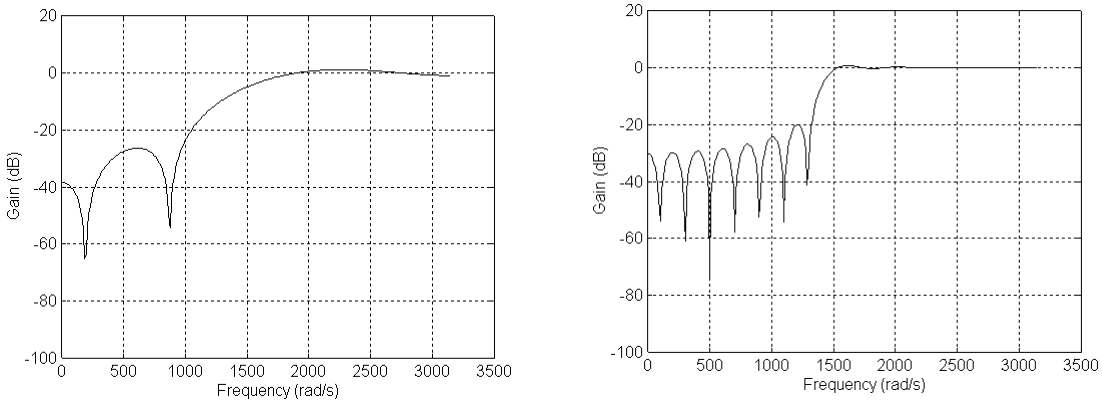

Figure 33: Highpass filter frequency responses for 7-tap (left) and 31-tap (right) filters with a target cutoff frequency of 1413.7 rad/s (225 Hz).

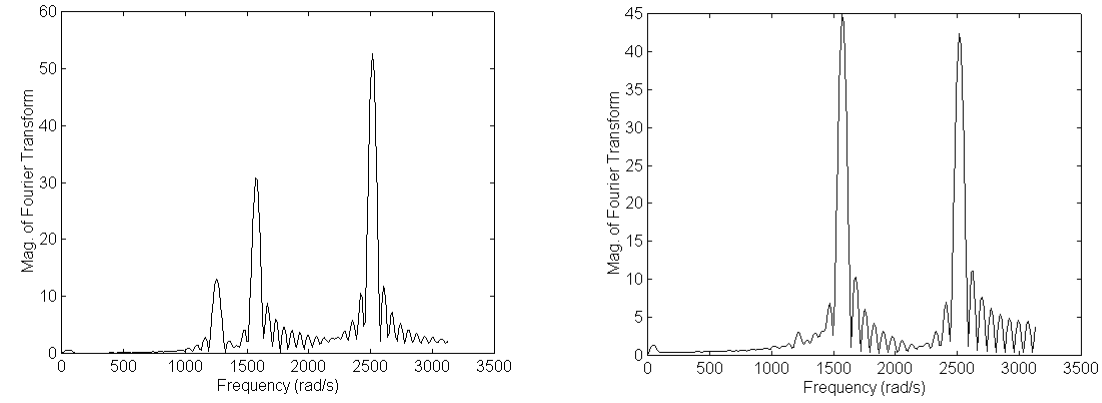

Figure 34: Highpass filtered signal spectrums for 7-tap (left) and 31-tap (right) filters.

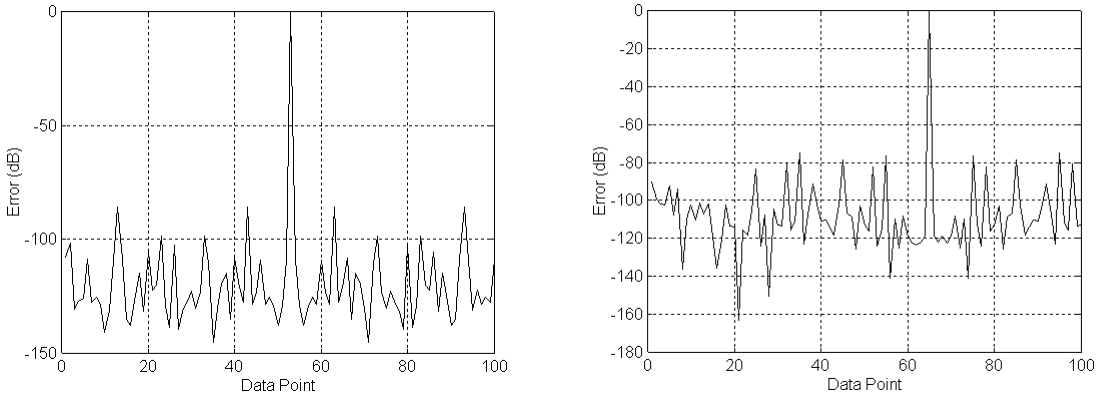

Figure 35: Highpass filter output error analysis for 7-tap (left) and 31-tap (right) filters.

B.3 Bandpass Filter Verification

The bandpass filter attempts to only pass frequencies in the 180 Hz - 270 Hz range. A significant difference between the two filter frequency responses can be seen where the 7-tap filter allows much more energy from unwanted frequencies to pass. In addition, the left sidelobe for the lower frequencies does not fall below the -20 dB mark. However, the 31-tap filter provides much sharper roll-offs for both cutoff frequencies and exhibits a better sidelobe characteristic. Operating error conditions continue to stay well below the -100 dB mark as shown in Figure 38.

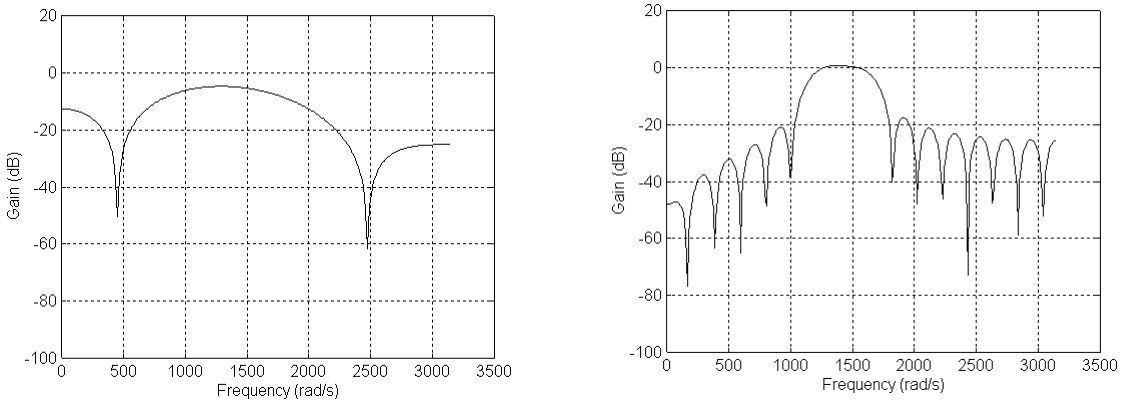

Figure 36: Bandpass filter frequency responses for 7-tap (left) and 31-tap (right) filters with a target cutoff frequency of 1130.97 rad/s (180 Hz) and 1696.46 rad/s (270 Hz).

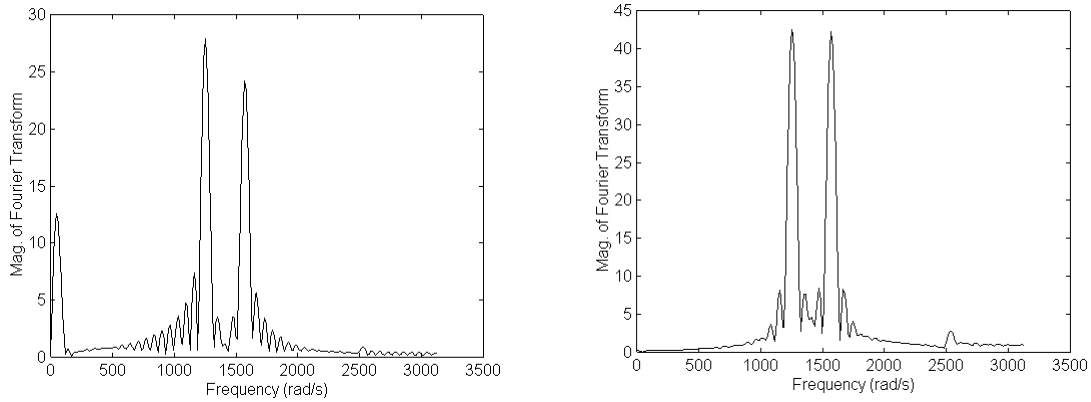

Figure 37: Bandpass filtered signal spectrums for 7-tap (left) and 31-tap (right) filters.

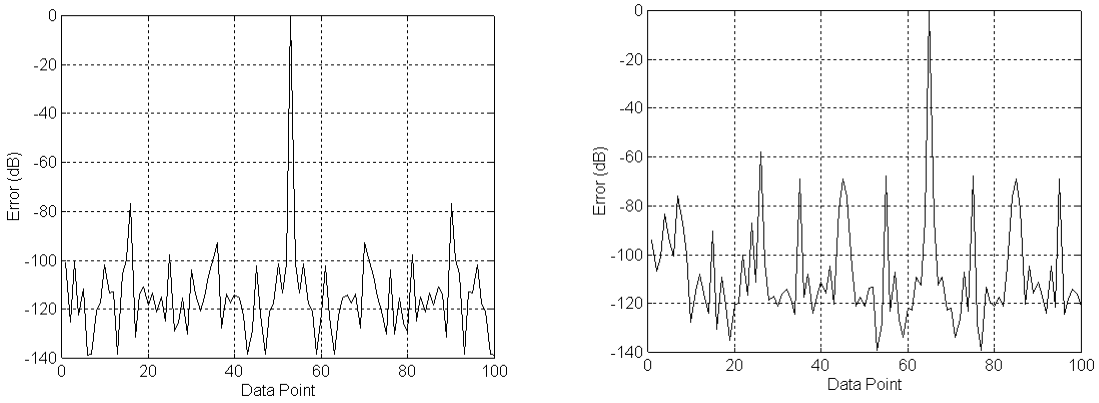

Figure 38: Bandpass filter output error analysis for 7-tap (left) and 31-tap (right) filters.

B.4 Bandstop Filter Verification

The bandstop filter manifests the advantages of using a 31-tap filter of a 7-tap filter. The frequency response of the 7-tap filter provides very little, if any, filtering capabilities for a narrowband exclusion. In fact, no cutoff below the -20 dB mark can be found. However, the 31-tap filter continues to show sharp roll-offs for both cutoff frequencies and eliminates frequencies in between below the -20 dB mark. Despite frequency responses, the operating error falls well below the -100 dB mark as shown in Figure 41.

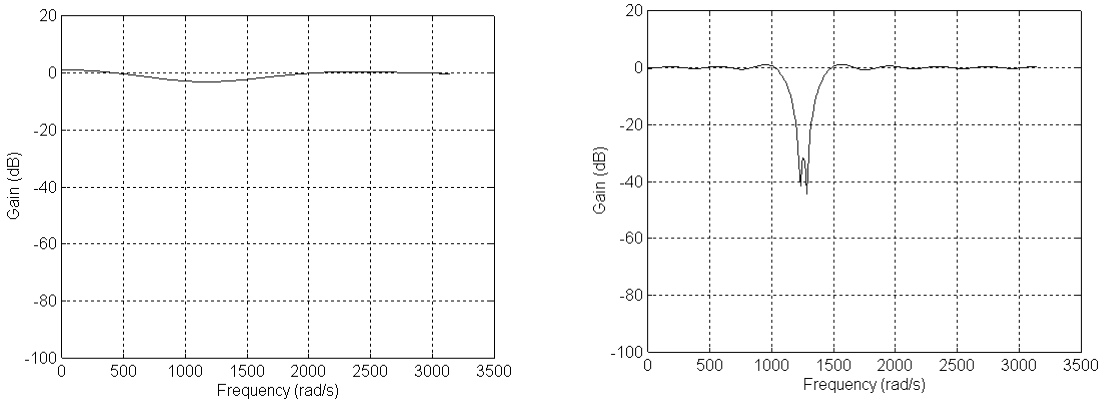

Figure 39: Bandstop filter frequency responses for 7-tap (left) and 31-tap (right) filters with a target cutoff frequency of 1130.97 rad/s (180 Hz) and 1382.30 rad/s (220 Hz).

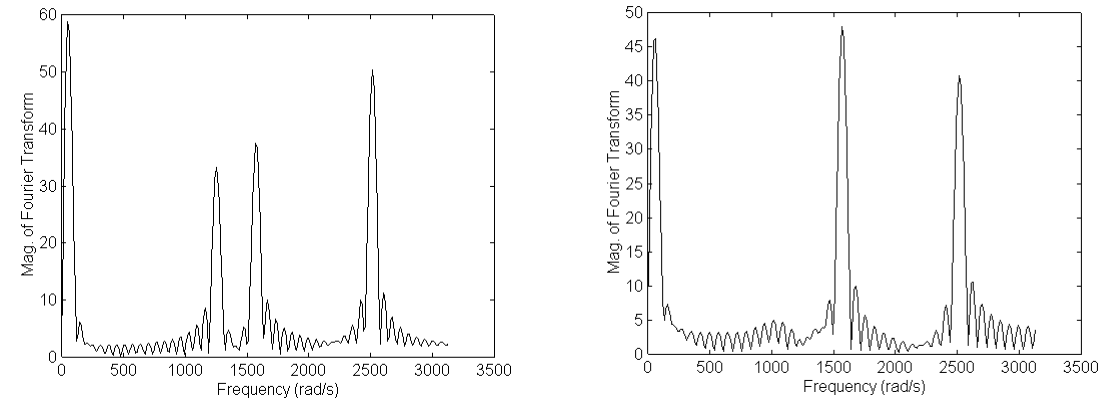

Figure 40: Bandstop filtered signal spectrums for 7-tap (left) and 31-tap (right) filters.

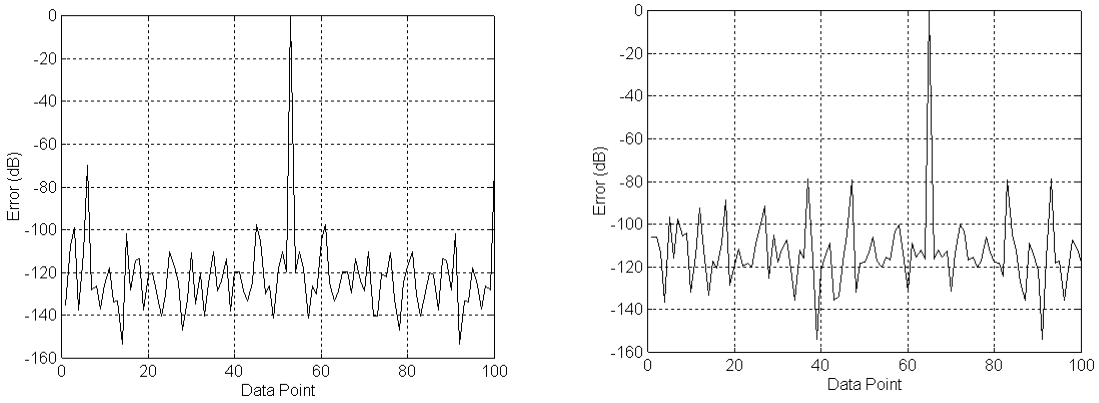

Figure 41: Bandstop filter output error analysis for 7-tap (left) and 31-tap (right) filters.

Bibliography

[1] J. R. Armstrong and F. G. Gray, *Structured Logic Design with VHDL*, Prentice Hall, 1993.

[2] R. W. Hamming, *Digital Filters*, Prentice Hall, 1983.

[3] Jeffrey M. Arnold and Margaret A. McGarry, *Splash 2 Programmer's Manual*, Supercomputing Research Center (unpublished), 1993.

[4] A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing*, Prentice Hall, 1975.

[5] V. Cappellini, A.G. Constantindes, and P. Emiliani, *Digital Filters and Their Applications*, Acedemic Press, Inc., 1978.

[6] John G. Proakis and Dimitris G. Manolakis, *Digital Signal Processing*, Macmillan Publishing Company, 1988.

[7] R. E. Bogner and A. G. Constantinides, *Introduction to Digital Filtering*, John Wiley & Sons, Ltd., 1975.

[8] Andreas Antoniou, *Digital Filters: Analysis and Design*, McGraw-Hill, 1979.

[9] Herman J. Blinchikoff and Anatol I. Zverev, *Filtering in the Time and Frequency Domains*, John Wiley & Sons, Inc., 1976.

[10] Theodore S. Rappaport, Scott Y. Seidel, and Koichiro Takamizawa, *Statistical Channel Impulse Response Models for Factory and Open Plan Building Radio Communications System Design*, IEEE Transactions on Communications, Vol. 39, No. 5, May 1991.

[11] Theodore S. Rappaport, *Characterization of UHF Multipath Radio Channels in Factory Buildings*, IEEE Transactions on Antennas and Propagation, Vol. 37, No. 8, August 1989.

[12] John N. Little and Loren Shure, *Signal Processing Toolbox*, The MathWorks,

Inc., 1988-93.

[13] David Parsons, *The Mobile Radio Propagation Channel*, John Wiley & Sons, 1994.

[14] Rodger E. Ziemer and Roger L Peterson, *Introduction to Digital Communcations*, Macmillan Publishing Co., 1992.

[15] Theodore Rappaport, Scott Y. Seidel, Prabhakar M. Koushik, and Scott L. McCulley, *A User's Manual for SIRCIM: Simulation of Indoor Radio Channel Impulse response Models*, VTIP, 1992.

[16] Mobile Portable Radio Group of Virginia Polytechnic Institute and State University, *A Users Manual for BERSIM: Bit Error Rate SIMulator*, Virginia Polytechnic Institute and State University/VTIP, 1992.

[17] Dan Burns and Wally Kleinfelder, *Splash 2 Interface Board Engineering Document*, Supercomputing Research Center (unpublished), 1994.

[18] Xilinx, Inc., *The Programmable Logic Data Book*, 1993.

[19] Kai Hwang, *Advanced Computer Architecture*, McGraw-Hill, 1993.

[20] John A. Eldon and Craig Robertson, *A Floating Point Format for Signal Processing*, Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, 1982, pp. 717-720.

[21] Nabeel Shirazi, Al Walters, and Peter Athanas, *Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines*, FCCM, 1994.

[22] Nabeel Shirazi, *Implementation of a 2-D Fast Fourier Transform on an FPGA Based Custom Computing Platform*, Master's Thesis (in progress), Virginia Polytechnic Institute and State University, 1995.

[23] J.L. Hennessy and D.A. Patterson, *Computer Architecture a Quantitative Approach*, Morgan Haufmann Publishers, Inc., 1990.

[24] Samuel D. Stearns and Ruth A. David, *Signal Processing Algorithms in MATLAB*, Prentice Hall, Inc., 1996.

[25] Amos R. Omondi, *Computer Arithmetic Systems: Algorithms, Architecture, and Implementations*, Prentice Hall International (UK) Limited, 1994.

[26] Peter J. Ashenden, *The Designer's Guide to VHDL*, Morgan Kaufmann Publishers,

Inc., 1996.

[27] Synplicity, Inc., *Synplify User's Guide*, Synplicity, Inc., 1994-1996.

[28] Duncan A. Buell, Jeffery M. Arnold, and Walter J. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*, IEEE Computer Society Press, 1996.

[29] The MathWorks, Inc., MATLAB *Reference Manual*, The MathWorks, Inc., 1992.

[30] Peter M. Athanas and Lynn Abbott, *Addressing the Computational Requirements of Image Processing with a Custom Computing Machine: An Overview*, The Ninth International Parallel Processing Symposium, 1995.

[31] D. Gajaski, *Silicon Compilation*, Addison-Wesley, Reading, Massachusetts, 1988.

[32] J. Peterson, Multiplier Module Generation Program, Annapolis Micro Systems, Inc., 1997.

[33] K. Eshraghian and N.H.E. Weste, *Principles of CMOS VLSI Design, A Systems Perspective*, 2nd Edition, Addison-Wesley Publishing Company, 1993.

Vita

Allison L. Walters

Al Walters was born in April, 1970 and grew up in Sterling, Virginia. After graduating from Park View High School, he received early acceptance to Virginia Tech's engineering program. In 1992, he completed his undergraduate degree in Computer Engineering and continued immediately to start a Master's program in Electrical Engineering. Al worked under Dr. Peter Athanas and the Center for Wireless Technology to research high-performance algorithms on re-configurable computing platforms. While working for Annapolis Micro Systems, he received his Master's degree in 1998. Seeking further goals in the field of wireless communications, he now works for ioWave in Georgetown.

After completing his work at Virgina Tech, he continues to learn more about re-configurable computing and wireless communications using spread spectrum technology. Most of all, he can enjoy more time playing his favorite sport of volleyball with his fiancée, Joyce.