High Speed DSP Implemented in Run-time

Partially Reconfigurable FPGAs

by

Justin D. McBride

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

February 3, 2003

Author_____
Department of Electrical Engineering and Computer Science
February 3, 2003

Certified by_____
Sean Adam
Teradyne Thesis Supervisor

Certified by_____
Dr. Christopher Terman
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

High Speed DSP Implemented in Run-time

Partially Reconfigurable FPGAs

by

Justin D. McBride

Submitted to the
Department of Electrical Engineering and Computer Science

February 3, 2003

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

This thesis investigates the feasibility of utilizing a run-time partially reconfigurable FPGA to implement a sequence of high-speed digital signal processing filters. Rather than reconfiguring the entire device to modify part of a configuration, a modular architecture is designed to allow smaller segments of the device to be individually reconfigured while the remainder of the device continues to operate. This document describes the design, implementation, simulation, and benchmarking of a five-socket modular DSP architecture and compares the results to the performance of alternative digital signal processing methods, particularly that of software DSP subroutines run on a PowerPC processor. The result is a highly flexible architecture that supports the use of timing verified hardware subroutines that could be partially reconfigured onto the FPGA within 3ms. The highly parallel processing power of the FPGA design yields a performance of 5.825 billion multiply and accumulate operations per second while simulated running at 72.8MHz, more than 76 times faster than similar calculations measured on a MPC7410 processor.

Thesis Supervisor: Dr. Christopher Terman
Title: Senior Lecturer, Department of Electrical Engineering and Computer Science

Thesis Supervisor: Sean Adam
Title: Hardware Engineering Manager, Teradyne

Acknowledgements

I would like to acknowledge the guidance and support of a number of supervisors and friends that kept me on course to complete this thesis. First, I would like to recognize Teradyne and specifically my supervisors Sean Adam and Dag Lundstrom. Sean was instrumental in building my interest in digital design and focusing my attention on finding a suitable project that benefited both Teradyne and M.I.T. In addition to serving as the origin of the idea for my thesis project, Dag continued to assist me in developing the project, finding useful informational contacts outside Teradyne, and finalizing this document. At M.I.T., Dr. Christopher Terman gave me the motivation and guidance I needed to focus on the relevant design strategies and presentation techniques that culminated in the creation of this thesis. I would also like to recognize the VI-A Internship program for allowing me to work on this project with Teradyne.

As much as the guidance of my three supervisors was paramount in working on the technical side of the project, I must thank Tiffany for her diligence in keeping me happy and focused on the goal of finishing this thesis in a timely manner. Also, my friends Andy, Larry, Gordon, and others at Phi Beta Epsilon helped me maintain a healthy balance between work and play. Finally and most importantly, I would like to thank my parents Jennifer and Jim along with Grandpa Don and Debbie for pushing me get into M.I.T. in the first place and helping me squeak my way thru these last four and a half years.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Background

Teradyne, the leading producer of automated test equipment (ATE) for digital, analog, and mixed signal testing, uses a complex array of digital signal processing (DSP) tools for a range of testing applications. A common mixed-signal application includes an analog signal capture, some digital hardware based and subsequently some software based signal conditioning and processing. Signal processing needs can vary greatly depending on the nature of the device under test, demanding a computationally intensive process of parameter estimation and other waveform characterization. As the devices in test become faster and more complex, these DSP capabilities must likewise progress with greater speed, complexity, and flexibility. Available DSP tools have improved drastically in the past decade, with many processing chains primarily composed of application specific integrated circuits (ASICs). ASICs, while optimized to provide the desired processing speed, are also subject to costly and time consuming development processes due to the overhead of design revisions and the inflexibility of the devices after development. The cost constraints of designing new ASICs can render this approach impractical, especially in low part volume situations commonly confronted in the design and construction of large automated testers.[1]

Ready-made ASICs with basic signal processing stages, digital filters, mixers, and oscillators provide another often-utilized processing avenue. While these programmable DSP devices do exploit highly parallel processing and are optimized for common processing tasks, the accommodation of a wide range of DSP algorithms increases the number of devices needed. This increase in device count places additional burden on board designers to both fit the devices into the space available and devise a bus architecture to support the devices, rendering this method too complex and too space consuming.

The utilization of general-purpose microprocessors (CPUs) for DSP purposes, on the other hand, offers a more flexible and powerful processing solution. The wide range of processing capabilities provided by CPUs coupled with software development efforts allows systems relying on CPUs to be easily revised during and after development. Unfortunately, CPUs appear unable to meet certain DSP demands anticipated by Teradyne due to the lack of specialization and hardware optimization.

In recent years, FPGAs have drastically improved in terms of size, speed, and features, making the devices suitable candidates for signal processing needs. Given that DSP algorithms typically rely on large-scale parallel multiplication, accumulation, and comparison, FPGA features such as embedded multipliers as well as the configurable logic aspect fit well with DSP requirements. The maturation of FPGA technology means that DSP systems could be designed to approach the processing speed and complexity of an ASIC-based solution, which avoiding recurring engineering development costs. In a 1995 study performed at Brigham Young University to quantitatively compare FPGA performance against DSP processors and ASICs, FPGAs were found to nearly match and in many benchmarks exceed the performance of both alternatives due to the ability of an FPGA to utilize extensive specialization and concurrency.[2] Furthermore, the ability to reconfigure an entire FPGA while in-system offers the capability to optimize the device for a particular processing task, matching the flexibility offered by CPUs while surpassing its processing ability.[3]

For most applications, once an FPGA design is tested and verified, it's seldom changed.[4] In addition to recent improvements in DSP implemented with FPGA technology, an additional capability for run-time partial reconfiguration of an FPGA offers an even more flexible and enticing alternative to both ASIC and CPU based processing. Using a partially reconfigurable FPGA, a device could be theoretically designed such that only a portion of the device would be reconfigured rather than reloading the entire device, a feature that allows for

interchangeability of code modules as well as smaller reconfiguration overhead. This feature would combine the versatility of a programmable solution with the performance of dedicated hardware in a single package, giving the FPGA a noticeable advantage over the use of multiple alternative processing solutions and will therefore be the goal of this project.[5]

Additionally, a run-time aspect would allow for a portion of the device to be reconfigured and optimized for an upcoming task while the rest continues to operate. The scenario is best described as adding a temporal floorplanning aspect to an FPGA, which by definition already utilizes spatial floorplanning in the creation of designs.[6] Given that most DSP applications are configured into a chain of sequential processing operations, DSP algorithms could therefore be designed into a modularized architecture of connected processing blocks and are well suited for partial reconfiguration applications. These processing blocks could be loaded with hardware subroutines representing various DSP algorithms as specified by the signal processing needs of the device under test. From the user's functional perspective, these pre-compiled and timing verified DSP modules loaded onto the FPGA would behave identically to fast software modules executed on a CPU while simultaneously providing a supplementary processing speed advantage.

1.2. Challenge

A high-speed digital signal processing design implemented as a run-time partially reconfigurable FPGA will be presented in this thesis as a feasibility study for future Teradyne applications. Also, the design will utilize an architecture allowing for the replacement of DSP modules. This feasibility study is made possible using a suit of configurations consisting of three separate signal processing algorithms with each offering the capability to enable or bypass the processing chain. All verification results are based on simulation because physical prototype board testing would add additional overhead to the project and is therefore outside of the scope of this thesis. Finally, benchmark results comparing the design implemented to both the current signal processing

performance of a G4 processor as they are used in Teradyne's current IntegraFLEX tester will be presented in this thesis.

1.3. Solution

After researching currently available devices and design methods, this system will be designed as a socket-based modular architecture realized on a single FPGA as seen in Figure 1. Each socket will be capable of holding a range of DSP algorithms designed to interconnect through a standard interface protocol. The standardization of each module's interface will give much greater flexibility in dynamically relocating DSP modules while greatly decreasing the complexity associated with interconnecting these modules. Unfortunately the interface protocol does add inflexibility to the type of module usable by limiting the specialization of the interface. Certain applications, for example, might require additional features such as extra status signals while others might require larger data busses. As will be described later in the thesis, the interface chosen allows for flexibility in the primary data bus width as well as secondary data and control pathways.

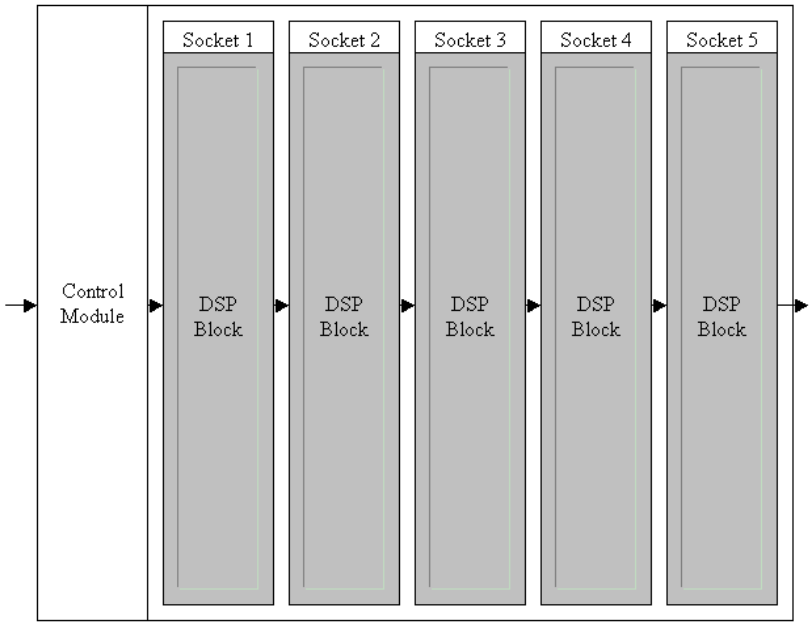

Figure 1 - Socket-based Architecture

As previously stated, when the system requires that a particular DSP algorithm is desired in a particular socket in the DSP chain, that DSP module may be partially reconfigured into the device without requiring the reconfiguration of the remainder of the FPGA. Figure 2 illustrates this reconfiguration process. Partial reconfiguration in an often-reloaded application is desirable as it decreases the reconfiguration time needed to make the chosen modification. Run-time partial reconfiguration is used here to denote that the remainder of the FPGA can continue processing data or operating while the reconfiguration of other sections of the device is in progress. The run-time aspect is realized using a FIFO for temporary storage of incoming data within the FPGA while DSP module(s) are being partially reconfigured. Likewise, the system is capable of modifying the parameters or coefficients contained within the DSP algorithm modules while the system continues to operate.

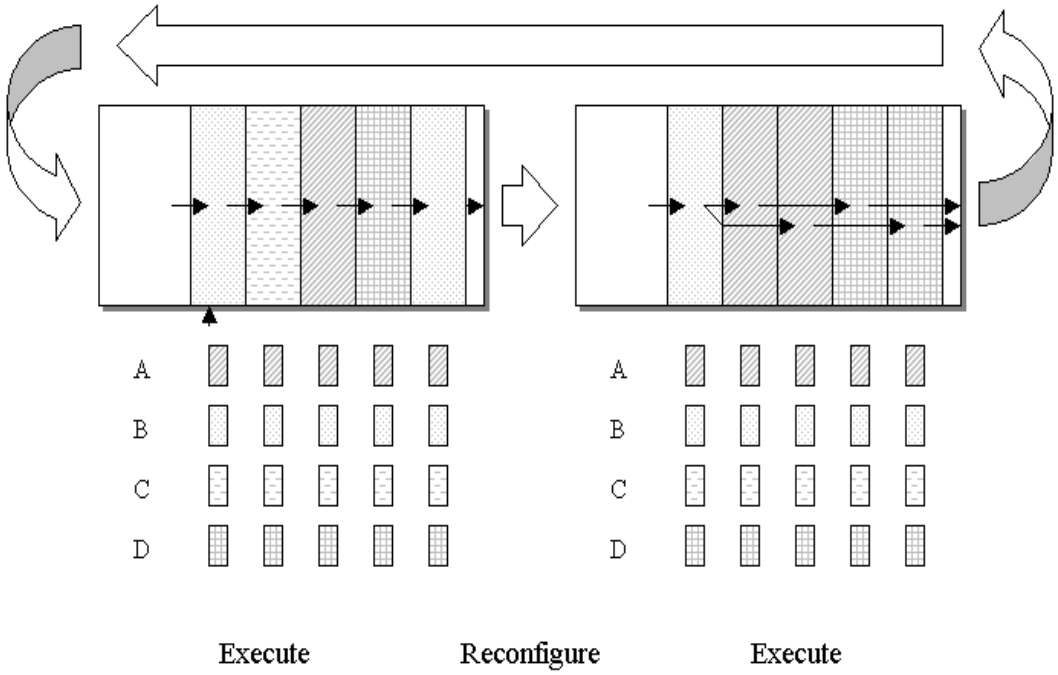

Figure 2 - Dynamic Reconfiguration

Teradyne's automated test equipment is designed to sequentially test a large number of devices for manufacturing verification purposes. As the tester utilizing this DSP FPGA switches between devices under test, the FPGA will utilize this

pause in data flow to empty out internal queues and process all data. Given the goal of operating the FPGA at 100MHz and a data queue that can hold up to 16K data points, a test pause of 164microseconds would be sufficient for the device to process all stored data. This desired speed of 100MHz and the associated data FIFO size implies that reconfiguration time must be less than the time needed to fill the FIFO. Whether or not this allowable queuing time is sufficient will be determined later in this document. The control system in the FPGA will manage reconfiguration and reparameterization scheduling to ensure that all data is processed by the FPGA, as the configuration existed when the data entered the device.

A Xilinx Virtex-II FPGA will be used for this thesis due to its physical support for run-time partial reconfiguration as well as Xilinx's recent efforts to provide support for this capability in its ISE design tool set. Additionally, the Virtex-II series FPGA also boasts advanced features including embedded multipliers and dedicated dual-port RAM blocks. As will be shown in chapter 6, these features will significantly decrease the gate-count necessary to attain the desired features and speed in the FPGA.

The design of this system is not without a number of design concessions due to constraints imposed by both the physical device as well as design tools. While the Xilinx Virtex-II is unique at the time this thesis began in supporting run-time partial reconfiguration in a commercially available part, this reconfiguration is column-based, which eliminates the ability of the designer to route signals through the area of the device undergoing partial reconfiguration.[7] Also, there is currently no support for dynamic module relocation, meaning that a separate version of each DSP algorithm must be created for each possible socket location of the DSP module. While the process of creating copies of each DSP module for all five sockets could be automated, timing may not be consistent among all five modules and some socket-module combinations may require attention. As will be described later in this thesis, the use of strict timing constraints on various

elements of module timing can alleviate these module replication timing issues. Likewise, Xilinx's ISE development suite does offer rudimentary native support for the design of a partially reconfigurable system, however there are no commercially available tools supporting the simulation of a complete run-time partially reconfigurable system. Rather, each permutation of the system must be independently simulated as static designs without the capability of simulating the actual reconfiguration process. These constraints will be elaborated upon in Chapter 4 of this thesis.

Despite these constraints, the design will be able to achieve the intended goal of operating as a run-time partially reconfigurable DSP processor although not quite capable of operating at the desired speed of 100MHz. If successful and cost-effective, Teradyne is likely to further explore this technology to enable the end user to configure and utilize an assortment of pre-compiled DSP algorithms that could be configured into the FPGA architecture's sockets. This dynamically reconfigurable system will effectively give the customer the 'virtual circuitry' it requires on demand with minimal reconfiguration times and little to no interruption in data processing.[8]

As this technology continues to mature, the physical and design tool constraints should disappear, giving the designer even greater flexibility in creating such a system. The potential problems associated with designing a complete system without the capability of verifying the reconfiguration process will hopefully be alleviated with advancements in design tool capability.

1.4. Outline

Following this chapter, this thesis will first consider past and present research involving the use of FPGAs in partial reconfiguration applications along with a detailed description of current FPGA features and capabilities. Chapter 3 will detail the three sample DSP filters chosen for this thesis, namely a 32-tap even-symmetry FIR, a Quadrature Mixer with a built-in NCO, and a 32-tap Time-varying Coefficient even-symmetry FIR filter. This chapter will concentrate on

the functionality of the filters, but will leave implementation-specific details for a later chapter. Next, Chapter 4 will examine the design constraints, both hardware and software, encountered during the design and implementation process. This chapter will close with a comparison of the possible design pathways and the features considered in making the decision to utilize the chosen device and design environment.

Chapter 5 will serve as a design specification for the architecture, starting with a high-level description of the intended design along with a detailed description of every facet of the target design. Starting with the fixed logic elements present in all permutations of the design, this chapter will then move into an implementation-specific description of the DSP filters first presented in Chapter 3. Following this specification, Chapter 6 will explain the actual implementation process, beginning with the top-level initial budgeting phase, moving then to the active module implementation phase, and concluding with the final assembly phase that brings all the pieces together into various design permutations. Next, Chapter 7 will detail the simulation process undertaken to verify the functionality and performance of the design, both at the individual module and overall device levels. Once a few versions of the design have been completely constructed and verified in a simulation environment, Chapter 8 will describe the performance benchmarks derived from these designs and compare them to other processing options, particularly the use of a G4 PowerPC processor to perform similar computations. In addition to detailing the raw processing power of each option, considerations are made for the time needed to partial reconfigure some or all of the FPGA and how this relates to the type of data set analyzed. In the penultimate chapter, this thesis will conclude with a summary of the work performed and knowledge gained in the process. Finally, Chapter 10 will contain a discussion of possible future work to build upon this thesis project.

2. FPGA and Partial Reconfiguration Background

2.1. FPGA Background

The SRAM-based field programmable gate array, or FPGA, was first commercially introduced by Xilinx in 1985.[9] The general purpose of a programmable logic device such as an FPGA is to allow designers to create a physical logic design and produce a finished product without the overhead associated with designing a custom IC. Furthermore, the devices could be reprogrammed with new configurations; a feature most often utilized during the design process, but one that can also be used to customize the device for the given operation. Initially, offerings from Xilinx consisted of a few thousand gates and could only operate at speeds of under 5MHz.[10] Since then, FPGAs have improved in size and speed to over 10 million gates at speeds approaching 300MHz while also incorporating additional features such as embedded multipliers and dedicated SRAM blocks.

The modern SRAM-based FPGA consists of a number of configurable logic blocks (CLBs) that are linked through a series of programmable wire interconnects. In the case of the Xilinx Virtex-II FPGA used, each CLB contains four slices, as seen in Figure 3.[11] Slice logic, as seen in the half slice schematic in Figure 4, is contained within look-up tables (LUTs), each designed as a multi-input, single output SRAM block. Coupled with storage registers, multiplexors, and various other logical mechanisms, each slice can perform a large range of logically operations. The four slices of a single CLB, when combined with neighboring CLBs, can perform almost any logical operation. Like the logic functions themselves, the switch matrix interconnects within and between CLBs are also determined by configuration data stored in SRAM cells, as seen in Figure 5.

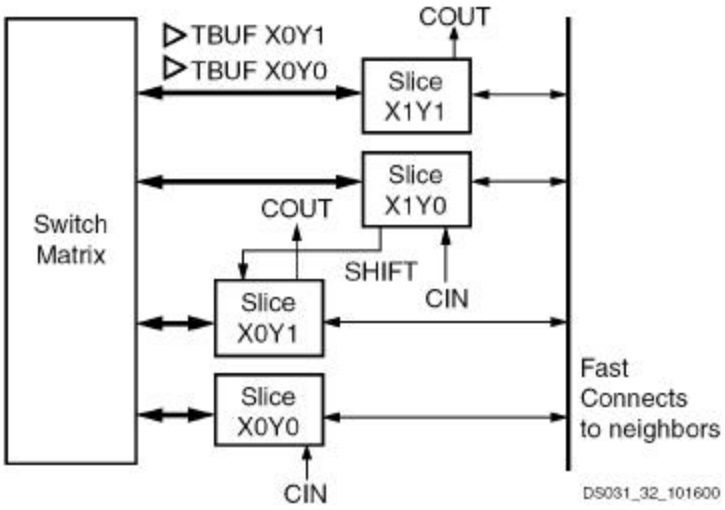

Figure 3 - CLB Schematic

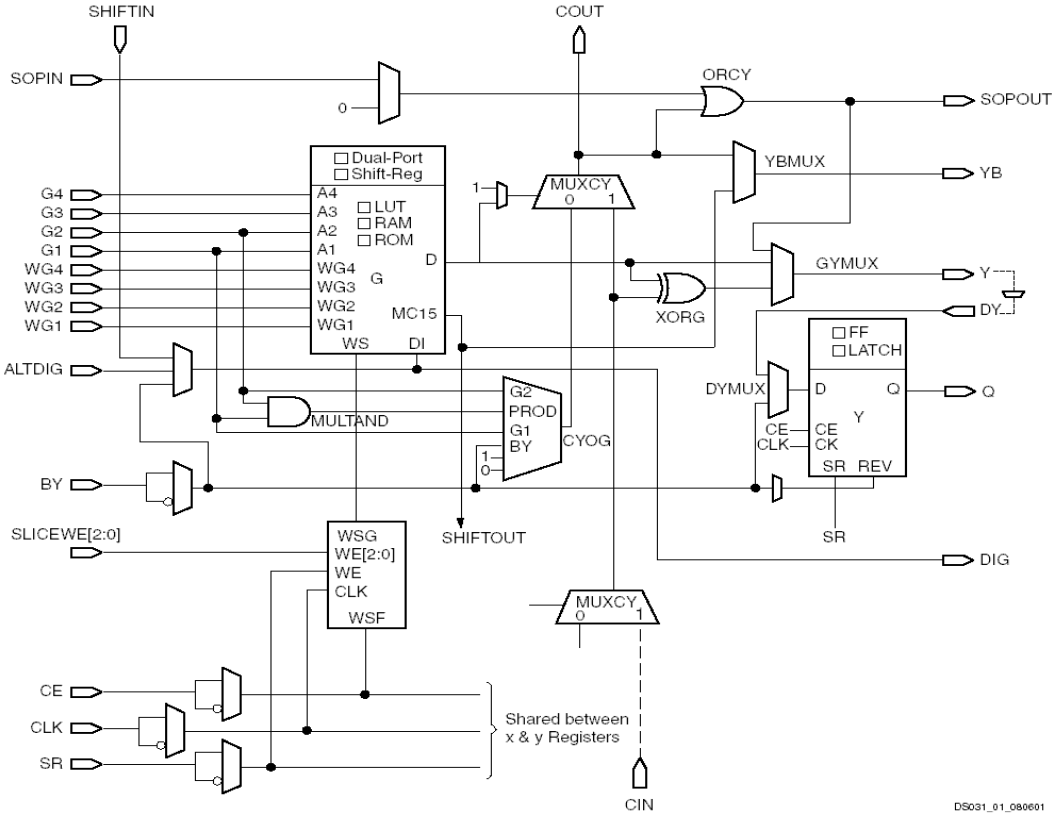

Figure 4 - Slice Schematic

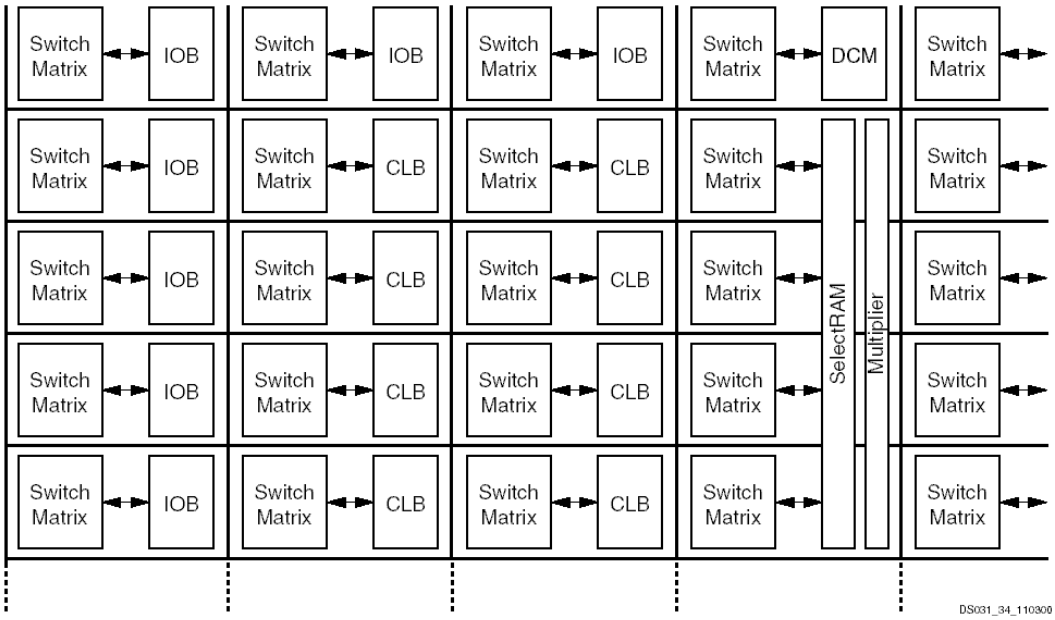

Figure 5 - Interconnected CLBs

The Xilinx Virtex-II FPGA used for this thesis uses an "island-style" architecture characterized by a fine-grained array of logic cells surrounded by a collection of prefabricated routing segments interconnected by programmable switches.[12] Specifically, this project will target the XC2V3000 Virtex-II FPGA, a device containing three million usable system gates. As seen in Figure 6, the device is organized as a 64x56 array of CLBs, each connected to a switching matrix used to interconnected neighboring CLBs. Moreover, each CLB consists of two tri-state buffers and four slices, each of which containing two function generators, two storage elements, and assorted multiplexors, arithmetic logic gates, and cascading chains. The device also includes six columns of 16 embedded 18x18 unsigned multipliers, which will be heavily utilized by the DSP algorithm implemented in this thesis. Each embedded multiplier borders a dedicated 18Kb block of SRAM that will be used to queue data and store coefficients within each module. The internal logic of the device is surrounded by IOB input/output buffers used to connect the device to its host board. Along with a set of multiple clock distribution systems, the FPGA offers a formidable array of capabilities.

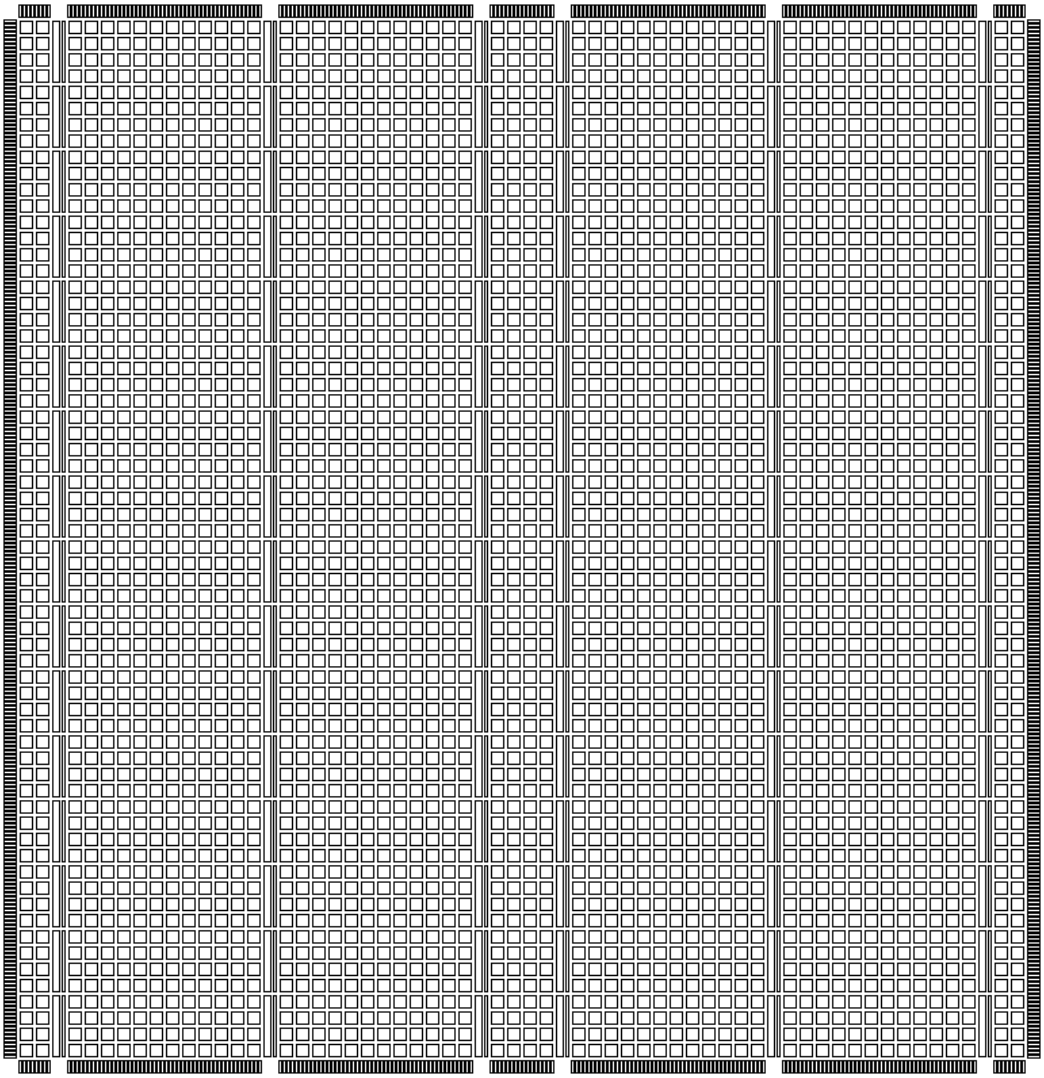

Figure 6 - Xilinx Virtex-II 3000 FPGA

If used as a traditional FPGA, all of these resources may be used for a design and a reasonably optimized mapping, placement, and routing of the design would be determined using standard design tools. In the case of a partially reconfigurable design, however, CLB logic placement and routing must be confined within specific internal boundaries in a manner that allows individual modules to be reloaded without effecting unrelated logic and routing. The partial reconfiguration support offered by the Virtex-II is column-based, meaning that the granularity of reconfiguration is limited to a module that is four slices wide and ranging the full height of the device.[13] As exhibited in Figure 7, this four-

slice minimum width restriction is based on the number of tri-state buffers necessary to create a bus macro barrier between modules, a concept that will be detailed later in this document. The full column restriction is based on the architecture's reliance on a full column as the finest granularity of bitstream loading available. This finest grain reconfigurable area consists of all CLB logic resources within the space as well as all non-clocking routing resources and IOB input/output buffers along the perimeter of the device that border the reconfigurable area. Partial reconfiguration bitstreams may be loaded through the standard reconfiguration interface such as the SelectMAP interface and will only affect the logic and routing within the confines of the target area. As mentioned, this area of the device may be partially reconfigured while the remainder of the device continues to operate. The designer, however, must be privy to contention issues that may arise if the remainder of the device attempts to communicate with portions undergoing run-time partial reconfiguration.

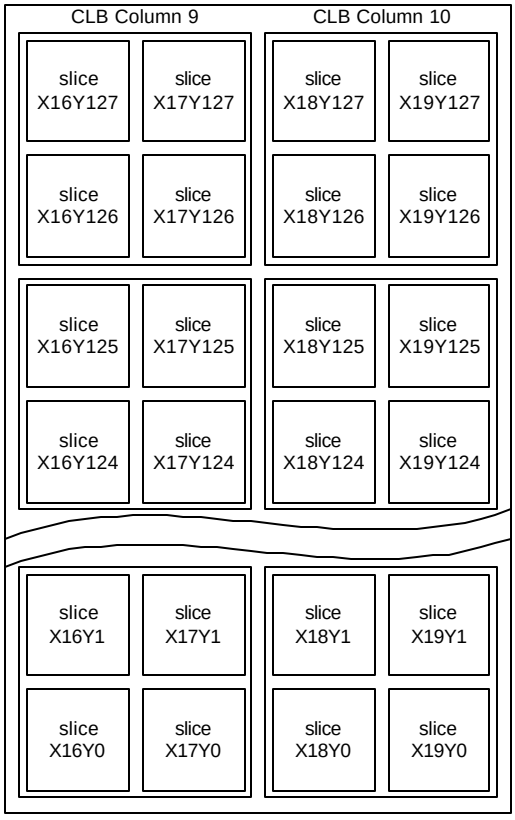

Figure 7 - Column-based Reconfiguration

2.2. Existing Research

In the past decade, a number of research efforts have been undertaken to explore and exploit partial reconfiguration in FPGAs. Partial reconfiguration is defined as any instance where only a portion of the device is undergoing a configuration change, as opposed to the entire device. Support for partial reconfiguration implies that specific regions of the device can be addressed and modified by a reconfiguration bitstream without modifying or disabling unchanged portions of the configuration. As a result, partial reconfiguration applications require special attention to logic and routing resource allocation in order to prevent contention. A number of FPGAs were developed with the capability for partial reconfiguration, such as the Xilinx XC3090 and XC6200 series as well as parts from Atmel and National Semiconductor.[14] While research efforts vary from developing place and routing tools to creating simulation environments, research projects generally fell into one of two categories: partial reconfiguration using dynamically compiled configurations and partial reconfiguration using pre-compiled designs.

The first category of dynamic compilation-based schemes has been primarily targeted towards dynamically recompiling a configuration to create optimized solutions in run-time. The RRANN and RRANN2 projects attempted to create a run-time reconfigurable artificial neural network on an FPGA with the capability to recompile and reconfigure itself as it effectively learned how to process data.[15] This project also considered creating partial reconfiguration bitstreams such that only the differences between two consecutive configurations are defined in the bitstream, which would decrease the size of the bitstream and reduce reconfiguration time. As this thesis project requires reconfiguration sockets to be loaded with any of a large library of DSP modules, it would be prudent to simply store each possible bitstream rather than create differential bitstreams to convert between each possible combination of configuration changes. More recent work such as Xilinx's Jbits tool also attempts to create a real-time redesign capability by leveraging core libraries to quickly map, place, and route designs onto a

device.[16] This project, however, does not attempt to tackle the problem of real-time design compilation and reconfiguration, predominantly due to the lack of commercially available tools to support this endeavor.

A larger effort has been made to develop the design and verification tools necessary to create designs allowing for partial reconfiguration using pre-compiled bitstreams. The DYNASTY project created by Milan Vasilko attempts to create a CAD framework that supports not only the typical spatial floorplanning of an FPGA design but also a temporal floorplanning aspect as well, enabling the designer to visualize the layout of tasks on the FPGA over time.[1718] While this design tool capability would be ideal for this project, the newer and more powerful Xilinx FPGAs are not supported under the design environment. On other projects, researchers have attempted to create module-based designs similar to this project. Gordon Brebner, with his concept of Swappable Logic Units, for example, created an architecture supporting the partial reconfiguration of small logic blocks within a defined interface for a Xilinx XC6200 device.[19] Unlike this thesis, however, Brebner's work attempts to modify the configuration on a much smaller scale than the larger DSP algorithm modules created for this design.

While some research has set out to create design implementation tools and architectures supporting partial reconfiguration, another group of projects have set out to create simulation and verification environments with this same support. As will be evident later in this thesis, the lack of compatible simulation tools drastically hinders the ability of designers to verify partially reconfigurable designs created with standard design tools. The Dynamic Circuit Switching (DCS) CAD framework enables the implementation and verification of partial reconfiguration designs by converting dynamic designs into multiple static designs for verification, and then back to single dynamic system.[20] Dynamic circuitry is defined as any logic or routing resources designed to be modified during partial reconfiguration. This capability of the simulator to model static

circuitry while simultaneously modeling the replacement of dynamic circuitry, however, is not available for use in this thesis and therefore necessitated the manual simulation of each separate design permutation without the ability to simulate the partial reconfiguration process.

2.3. Current Capabilities

While each of the research projects mentioned in the previous section has made academic progress in the field of dynamically reconfigurable FPGAs, much work remains. Current academic and commercial development efforts have attempted to bridge the gap by creating tools that support partial reconfiguration in modern-day architectures such as the Virtex and Virtex-II platform FPGA families.

2.3.1. JBits

Being the leading designer and producer of FPGAs, Xilinx has an inherent interest in providing the design tools necessary to allow for run-time partial reconfiguration. JBits evolved from earlier internal partial reconfiguration tool into a Java-based API supporting the reading, manipulation, and writing of configuration bitstreams for Virtex FPGAs.[21] The tool generally operates at a lower level, allowing fine-grained bitstream and logic manipulation and the ability to draw on automated core generation capabilities. The JBits tool also includes JRoute, which provides access to routing resources in dynamic compilation situations.[22] VirtexDS, perhaps the most significant and useful member of the JBits toolset, allows for device-level simulation of run-time reconfiguration designs by running simulations directly against bitstreams generated with JBits.[23]

While this toolset may seem like the optimal design environment for this thesis, the JBits toolset currently only supports Virtex and older XC4000 series FPGAs and is not compatible with the Virtex-II platform FPGA utilized in this thesis. Since the embedded multipliers intrinsic to the Virtex-II and absent in the Virtex are vital for the DSP application targeted, JBits will not be considered for the remainder of this project. It is important to note,

however, that this project would have greatly benefited from the dynamic simulation features available in VirtexDS.

2.3.2. PARBIT

In a project similar to the run-time partial reconfiguration architecture designed for this thesis, an effort is underway to design a reconfigurable ATM switch architecture called RECATS that utilizes dynamic hardware plugins designed to fit with specific regions of an FPGA.[24] Based on the Xilinx Virtex-E architecture, these dynamic hardware plugins are designed to fit within interface gaskets present on the FPGA. To accomplish the task of creating full column partial bitstreams, PARBIT was created to allow for dynamic hardware plugin bitstreams to be combined with a bitstream representing the default configuration to create valid column-length bitstreams. While this bitstream generation feature may have been useful for this project, the tool is also not compatible with the Virtex-II FPGA family. Additionally, the ability to circumvent some of the routing constraints confronted by both the dynamic hardware plugin project and this project have been accommodated by Xilinx's Modular Design tool and the creation of a native partial reconfiguration design flow.

2.3.3. Modular Design

While academic research projects and experimental tools might be useful for the general advancement of reconfigurable computing technology, the technology is commercially useless unless a viable application or product is derived. To complement the partial reconfiguration support inherent in the Virtex-II FPGA, Xilinx recently augmented their ISE development toolset with a modular design tool, which enables the synthesis, translation, mapping, placing and routing of an individual modules within a larger design. The modular design tool is intended to allow large FPGA designs to be partitioned among multiple engineers as multiple modules that can be synthesized, translated, mapped, placed and routed, timing can be verified, and the modules can be combined into a final design. In order to prevent resource contention, the modular design tool allows a designer to initially budget

individual modules into specific regions of chip and automatically prevent logic and routing from straying beyond those boundaries.[25] This feature solves many of the manual routing issues encountered by earlier research projects.

While designers previously had the ability to specifically place individual logic components, the added ability to control routing gives the modular design tool the ability to create a complete bitstream for a partially reconfigurable module by eliminating resource overlap. Xilinx formalized this capability with the release of application note XAPP290 detailing the steps necessary to creating a partially reconfigurable design using Xilinx's ISE development suite with the modular design add-on.[26] While modular design allowed the creation of complete individual modules, XAPP290 introduced a bus macro scheme utilizing tri-state buffers to bridge the gap between interconnected modules and prevent signal failure during reconfiguration. Although severely lacking in the ability to simulate a partially reconfigurable environment, the modular design tool with partial reconfiguration support does create the capability to design a partially reconfigurable system using standard design tools and a commercially available FPGA. Therefore, this thesis will be performed using this toolset along with a thorough analysis of the advantages and disadvantages of this design flow.

3. DSP Algorithms

Teradyne's DSP requirements are as diverse as the devices under test. In many applications, digitized data must pass through a series of standard processing functions, including an equalizer, a numerically controlled oscillator, quadrature mixers, resamplers, and other filters. A typical processing sequence is given in Figure 8. Without the use of an FPGA, this series of algorithms could exist as a static ASIC chain, however the flexibility of the DSP chips is limited and would necessitate the use of a G4 CPU for additional custom processing. Given that the purpose of this project is to prove that an FPGA-based dynamically reconfigurable processing solution can compare in speed and capabilities to both ASIC and CPU based processing, three demonstration filters will be created for this thesis. Therefore, an equalizer will be designed as a 32-tap even symmetry FIR filter, a numerically controlled oscillator (NCO) that feeds into a quadrature mixer with dual outputs will be implemented, and an interpolator/resampler will be implemented using a 32-tap time-varying coefficient even symmetry FIR filter. The resampler differs from the equalizer by using an addressable array of coefficient values for each tap rather than a single value. All filters will be designed to operate at 100MHz on 16-bit data with no less than 16-bit internal resolution.

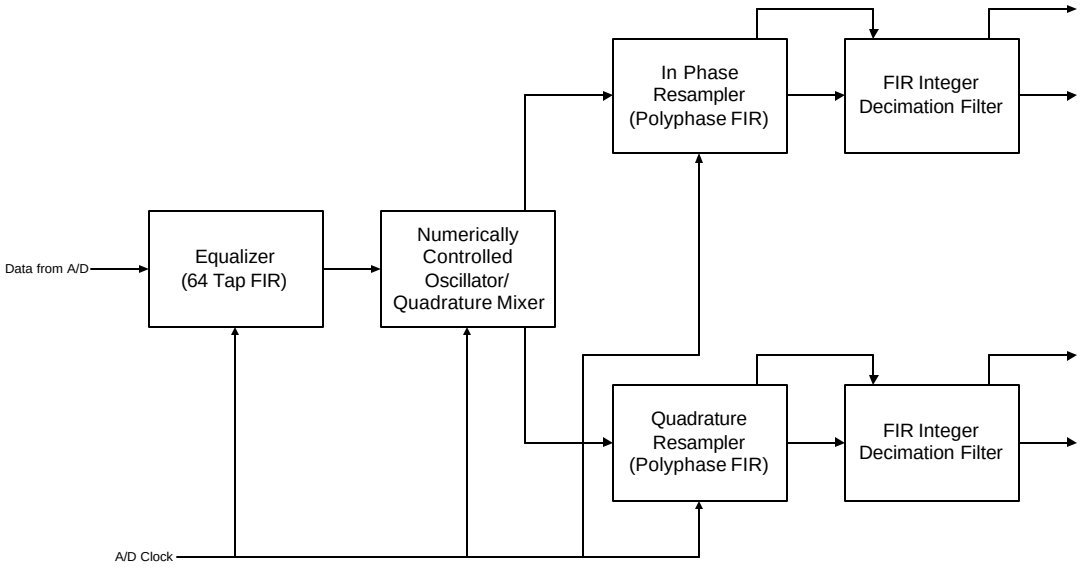

Figure 8 - Sample DSP Filter Series

3.1. FIR

The purpose of the FIR filter is to equalize the digitized signal to compensate for imperfections in the instrument receiver response. Each delayed incoming data is multiplied by a coefficient, summing the results of all 32 taps, and outputting the resulting sum to the next stage of data processing. Linear phase is assumed and the multipliers can be reused according to the symmetry condition, which reduces the number of multipliers needed, as illustrated by Equation 1. Figure 9 gives a simplified 4-tap version of the design. The 32-tap even symmetry FIR filter will be designed to operate on 16-bit signed data with 16-bit coefficients to produce 32-bit products and sums internally, which will then be rounded down to a 16-bit signed output. Coefficients will not be hard-wired into the design and can be reloaded at any time without having to reconfigure any logic.

Equation 1 - 32 tap Even-symmetry FIR Filter

$$Data\_out = \sum_{i=1}^{16} coeff_i \times (tap_i + tap_{33-i})$$

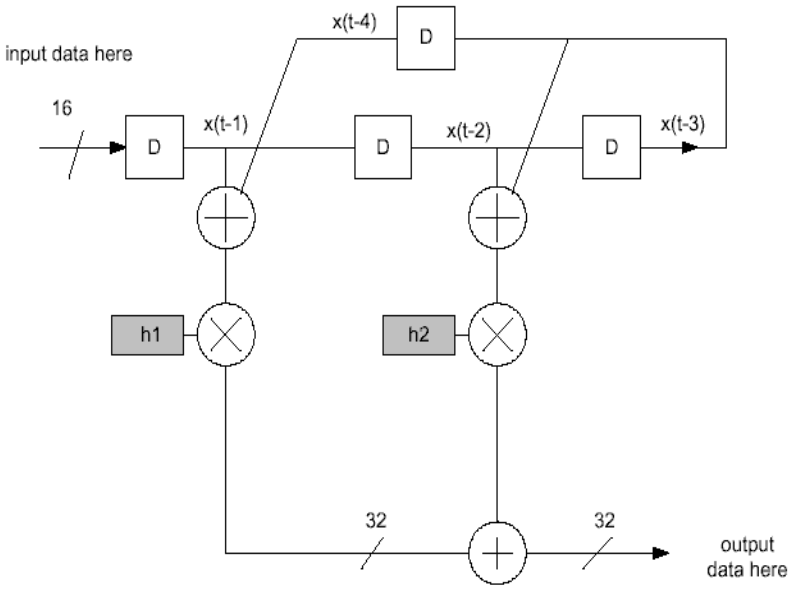

Figure 9 - Even Symmetry FIR Filter

As an added feature, the system will be designed such that two neighboring 32-tap even symmetry FIR filters can be joined to form a single 64-tap even symmetry FIR filter. Data tap interconnects will be provided between modules along with a 32-bit intermediate sum output from the first to second filters in the sequence. The resulting sum of all 64-taps will then be rounded down to a 16-bit value for outputting.

3.2. Quadrature Mixer

The quadrature mixer/downconverter will be implemented with a built-in numerically controlled oscillator, or NCO, that is designed to digitally synthesize a discrete sine and cosine based on the supplied period parameter. The synthesized signals are fed into two multipliers together with the incoming data stream. The NCO portion of this filter will consist of a phase accumulator used to address sine and cosine lookup tables in order generate deterministic waveforms. As seen in Figure 10, the 32-bit phase accumulator is augmented by a 32- bit assignable phase increment register, with the output rounded down to a 16-bit phase angle used to address the lookup tables. The sine and cosine lookup will then output a set of 16-bit discrete output values, which will then be individually multiplied by the 16-bit data input to create discrete output values. In order to minimize lookup table memory requirements, quarter wave symmetry will be utilized, meaning that only one quarter of the sine waveform must actually be stored and that the quadrant designated by the phase angle can be used to determine the sign and value of the output. The 16-bit rounded in-phase and quadrature values will each then be outputted to the next processing module.

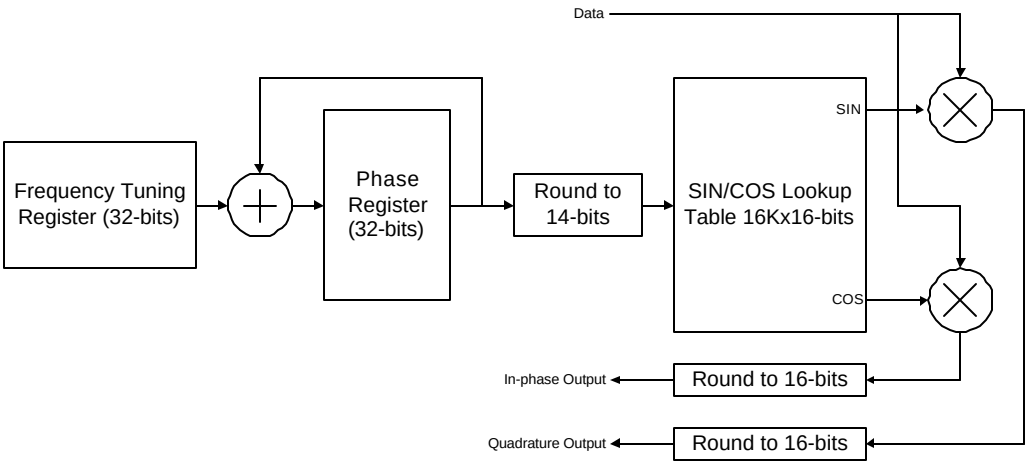

Figure 10 - Quadrature Mixer with NCO

3.3. Time-varying Coefficient FIR

The time-varying coefficient FIR filter will be implemented as a 32-tap even symmetry FIR filter with time-varying coefficients. The even symmetric tap accumulators, multipliers, and sum accumulation structure is identical to the 32-tap even symmetry FIR filter detailed above. Rather than a single coefficient for each multiplier, however, a memory of 1Kx16-bit coefficients is connected to each multiplier. As seen in Figure 11, all 16 sets of 1K memories are addressed by an accumulator that increments the address using an assignable 32-bit delta value and taking only the rounded 10 most significant bits as the coefficient address. The coefficients and delta can be chosen to give a range of filtering capabilities. Like the other filters, all coefficients and the delta can be reloaded once the filter has been configured onto the FPGA. While the capability of combining two 32-tap time-varying coefficient FIRs into a single 64-tap time-varying coefficient FIR will be not supported for this thesis, two neighboring 32-tap time-varying coefficient FIRs can be configured to pass thru the necessary data values and sums to allow for simultaneous in-phase and quadrature filtering.

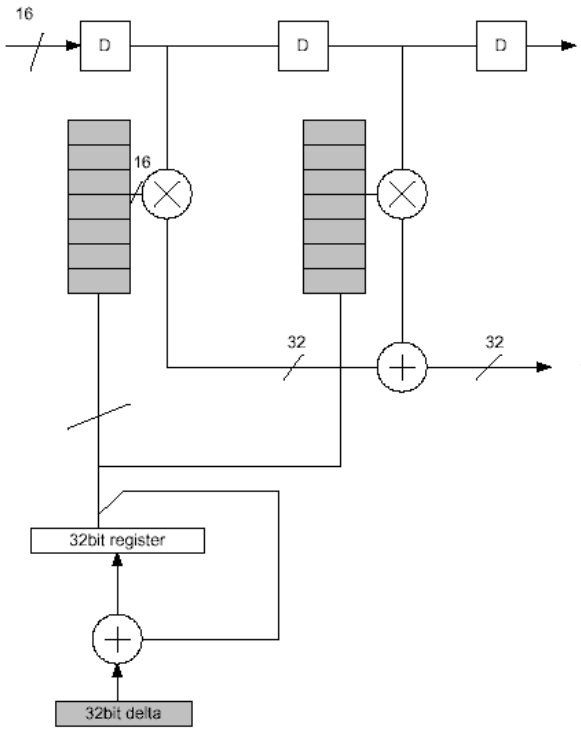

Figure 11 – Time-varying Coefficient FIR

4. Design Decisions

4.1. Device Constraints

The Xilinx Virtex-II FPGA, while physically capable of partial reconfiguration, is not without limitations. It is possible to address and dynamically reconfigure specific portions of the device, but this dynamic reconfiguration must correspond to a column-based bitstream-reloading scheme. As mentioned in Chapter 2, the smallest portion of the device that can be dynamically reconfigured consists of an area four slices wide by the full column height of the device. Any area to be reconfigured must therefore be an integer multiple of a region this size. This limitation has the effect of heavily constraining the type of reconfigurable architectures and designs supportable on the Xilinx Virtex-II. Using this device, it would not be possible to create a socket-based architecture consisting of a grid of interconnected modules, as seen in Figure 12. The PARBIT project currently in progress intends to create a gasket-based modular architecture along with a bitstream modification tool capable of combining multiple bitstreams to create valid full column bitstreams.[27]

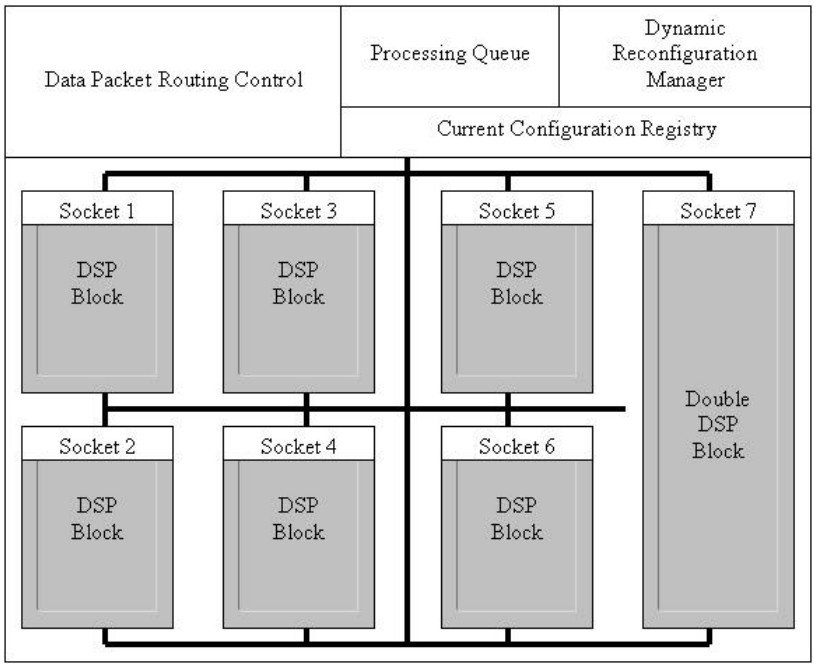

Figure 12 - Socket-based Architecture without Column Restraints

While the PARBIT tool might solve the problem of allowing the partial reconfiguration of modules that do not extend the full height of the device, the routing constraints of the Virtex-II continue to pose a formidable hurdle to unrestricted module design. Since all non-clocking routing resources are designated by the reconfiguration bitstream, all routing resources within a partially reconfiguring module are not available during reconfiguration. In the case that a partial reconfiguration is altering a set of columns in the middle of the device, this restriction thus prevents signals from passing between the left and right sides of the device without making use of external pin connections. This routing restriction limits the ability to freely route signals inside the device, leading to the design of a chained module architecture in which modules can communicate with their immediate neighbors, but cannot directly send or receive to more distant modules, a scheme resembling that given in Figure 1.

Finally, while the internal structure of the Xilinx Virtex-II FPGA is standardized and uniform with the sole exception of embedded multiplier and block RAM columns, the device does not support the ability to dynamically relocate modules to different locations on the device. For example, while areas A and B of Figure 13 are identical in size, logic and routing resources, inclusion of embedded multipliers and block RAM, and access to IOB resources, it would not be possible to create single bitstream that could be loaded into either location. Rather, a separate version of the bitstream would need to be built for each possible location.

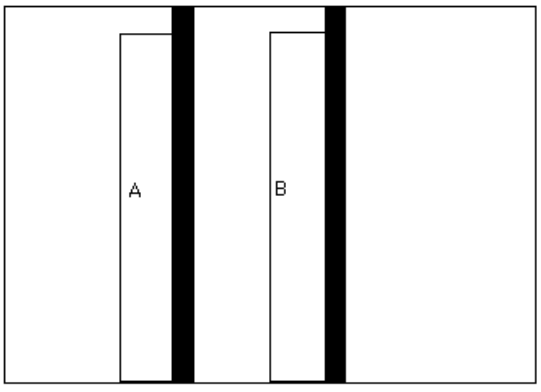

Figure 13 - Dynamic Relocation Restriction

Despite being the most advanced commercially FPGA architecture available at the time of this project, the Xilinx Virtex-II's device constraints drastically curtail the types of reconfigurable designs currently possible. As a result of these restrictions, the architecture described in the next chapter consists of a socket-based architecture that includes five full column-height modules in a chain from left to right on the device. These sockets are connected to their immediate neighbors via bus-macro routing protocols, which make use of tri-state buffers to communicate across module boundaries in accordance with a standard interface scheme. Additionally, since a single DSP module can be instantiated in all five of the sockets, five versions of the module bitstream must be created to support all possible locations.

4.2. Design Flow Constraints

Although one of a few manufacturers claiming to support partial reconfiguration of its devices, Xilinx is unique in its initial support for partial reconfiguration using standard design tools and methodologies. As mentioned in Section 2.2.3, Xilinx's ISE framework with the modular design tool does give the user the ability to create a partially reconfigurable design, but a number of roadblocks remain to prevent the more effective design and proper verification of such a design. On a superficial ease-of-design level, the framework does not offer a temporal floorplanning capability that allows designer to visualize configuration change over time. It is therefore necessary to design modules separately and manually combine them to create complete permutations of the design. This flaw in the design flow, however, is minor compared to others and will likely be corrected in future design flows as the demand for such features increases.

Along with the partial reconfiguration design flow, Xilinx has included a bus macro that enables the simple creation of boundary connections between partially reconfigurable modules. Figure 14 shows that each bus macro consists of four tri-state buffer bits that can be driven and read from either side of the module boundary.[28] Each 4-bit bus macro is one CLB in height and four CLBs wide -

two on each side of the module boundary. While the functionality of this macro could have been manually created, Xilinx has also encoded placement and routing directives into the macro for use by the ISE place and route tool. These placement and routing directives ensure that signals from both connecting DSP sockets interconnect at a single defined location. Without the use of these bus macros, it would not currently be possible to constrain signals to specific routing locations for interconnect purposes. Each of the 189 bus macros used in this design must be manually fixed to a specific location on the device, as will be illustrated in Chapters 5 and 6. Unfortunately, the design flow documentation only recommends for the bus macros to be uni-directional within the design to prevent signal contention, which eliminates the possibility of designing a bi-directional network or bus for routing signals between connecting modules.

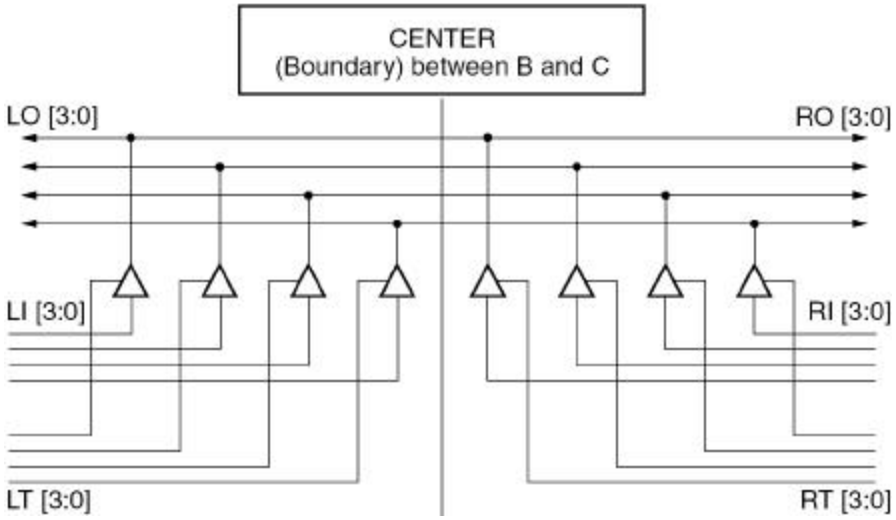

Figure 14 - Bus Macro

It should be reiterated that, while cumbersome, the design flow does allow for the effective creation of a partially reconfigurable design. Verification, however, cannot be properly accomplished because no commercially available simulator supporting partial reconfiguration currently exists. Like any standard FPGA design, a single assembled static permutation of the partially reconfigurable design can be verified for functionality and timing using a standard testbench. As

will be explained in later chapters, a permutation for this design would consist of the initial Fixed_Logic module, all five sockets filled with a DSP module, and the End_Logic module. Limited to this methodology, however, it is not possible to actually simulate the process of partially reconfiguring the device or to simulate the functionality of the FPGA in an in-system use situation. Since the available time and scope of this thesis forbids prototype board testing that would confirm the validity of this design in a real world scenario, the simulation of multiple static permutations of the design will have to suffice for both benchmarking and conclusion-drawing purposes. Information on reconfiguration times and the performance of the FPGA during partial reconfiguration must be extrapolated from information provided in Xilinx's data book as well as observed simulation results, which will be presented in Chapter 8.

4.3. Decision Matrix

While the decision to proceed with the project using the Xilinx Virtex-II FPGA along with Xilinx's ISE and modular design tool may seem obvious given the overall design tool capabilities and device features, it seems important to quantify that decision with a decision matrix. Table 1 details the three design routes under consideration at the commencement of this thesis project along with itemized features, allowing for a quantification of the advantages and disadvantages of each design path. Not all features are weighted equally, as indicated by the subjective scaling factor on the far right of the table. The chosen scaling factors reflect both information garnered during the research of previous projects and the recommendations of fellow engineers or supervisors.

Table 1 - Design Decision Matrix

Thesis Decision Matrix

	Options / Criteria	Xilinx ISE,MD Virtex-II	Jbits Virtex	PARBIT Virtex-E	scaling factor
Features	18x18 Multipliers	5	0	0	1
	Device Size	5	1	3	1
	Device Speed	5	3	3	1
Capabilities	Dynamic Relocatability	0	0	5	2
	Module Shape Limitations	1	1	3	2
Process Ease	Overall Design	5	1	3	3
	DSP Algorithm Coding	5	1	5	1
	Routing Constraints	5	1	3	2
	Module Timing Constraints	5	5	5	1
	Overall Timing Constraints	5	1	3	3
Support	External Design Support	5	3	3	1
	Internal Design Support	5	1	3	1
	Confidence in Process	5	3	3	2
	Total	87	30	68	

Using either the Xilinx Virtex or Virtex-E, for example, would have eliminated the ability to take advantage of the Virtex-II's size, speed, and embedded multiplier features. The Java-based JBits tool may support partial reconfiguration, but the design flow did not permit for simple and reliable design creation or verification and would have also carried with it the added overhead of requiring extensive experience with both standard FPGA design and Java programming. The PARBIT tool does present attractive features such as the added flexibility of dynamic module relocation and fewer module shape limitations, but is also inhibited by design flow concerns, validation, and tool support concerns. Utilizing the Xilinx ISE with the modular design tool offered the most attractive design pathway despite dramatic limitations in dynamic module relocatability and strict module shape limitations. In all cases, simulation support for dynamic partial reconfiguration was not available and therefore this factor was not taken into account in the decision matrix.

5. Device Architecture

5.1. Design Overview

Although aspects of the device architecture have been described or alluded to on an introductory level prior to now, this chapter will detail the exact specification of the design. The architecture, as illustrated by Figure 15 consists of seven distinct modules arranged left to right across the Xilinx Virtex-II 3000 series FPGA and separated by bus macro boundary interconnects. The first and last modules, referred to as Fixed_Logic and End_Logic respectively, exist in all permutations of the device and act as the interface connecting the internal DSP structure to the external interface. The five interior modules are the DSP sockets into which DSP modules can be loaded, referred to as sockets 1 thru 5. Between each module, a series of tri-state buffer bus macros has been placed to act as an interface, which is intended to prevent signal contention during reconfiguration. In this scheme, modules can only communicate with their immediate neighboring modules. Data and status signals therefore propagate through the system in order to reach their destinations. If the End_Logic module is ready for data, for example, that signal will propagate through the five sockets in reverse order until it reaches the Fixed_Logic module.

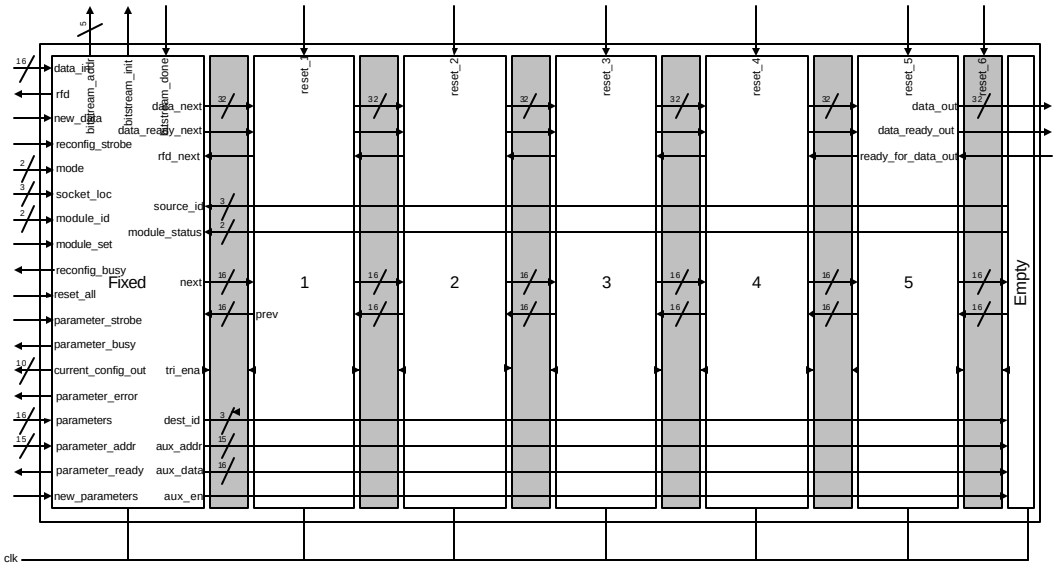

Figure 15 - Chip Layout

In order to make this device less dependent on outside micromanagement, the Fixed_Logic module orchestrates all data flow, parameter loading, and partial reconfiguration control. Data is then processed through the DSP modules in a left-to-right fashion to be transmitted externally via the End_Logic module. While the data bus and auxiliary parameter loading bus, which consists of the aux_addr, aux_data, dest_id, and aux_en signals and is referred to as aux_bus, travels rightward across the device, a few status signals communicate leftward to facilitate Fixed_Logic module operation. The details surrounding each intra-module and inter-module signal will be expounded upon in the following sections.

5.2. Fixed_Logic Architecture

The Fixed_Logic module essentially acts as the brain of the FPGA design - controlling data flow, parameter loading, and maintaining the operational state of the device. In the process of facilitating the intended operation of this run-time partially reconfigurable design, the current operational state of the device must constantly be monitored and coordinated with any attempts to either modify the configuration or load any new parameters or coefficients into any of the existing DSP modules, a process referred to as reparameterization. If either a reconfiguration or reparameterization is desired, a finite state machine sequence is followed to ensure that all necessary DSP modules are disabled prior to said modifications. Likewise, the affected DSP must also be re-enabled post reconfiguration or reparameterization so that the device may resume operation. Any data entering the data_FIFO prior to either a reconfiguration or reparameterization initialization signal must be processed with the old configuration before any changes occur. Similarly, any data input to the device after the initialization signals must be processed with the new configuration. In order to simplify the description of this fixed logic module, Figure 16 exhibits the internal compartmentalization of this module into coherent operational sub-modules. The operation and construction of each of these sub-modules will be described in the following sections.

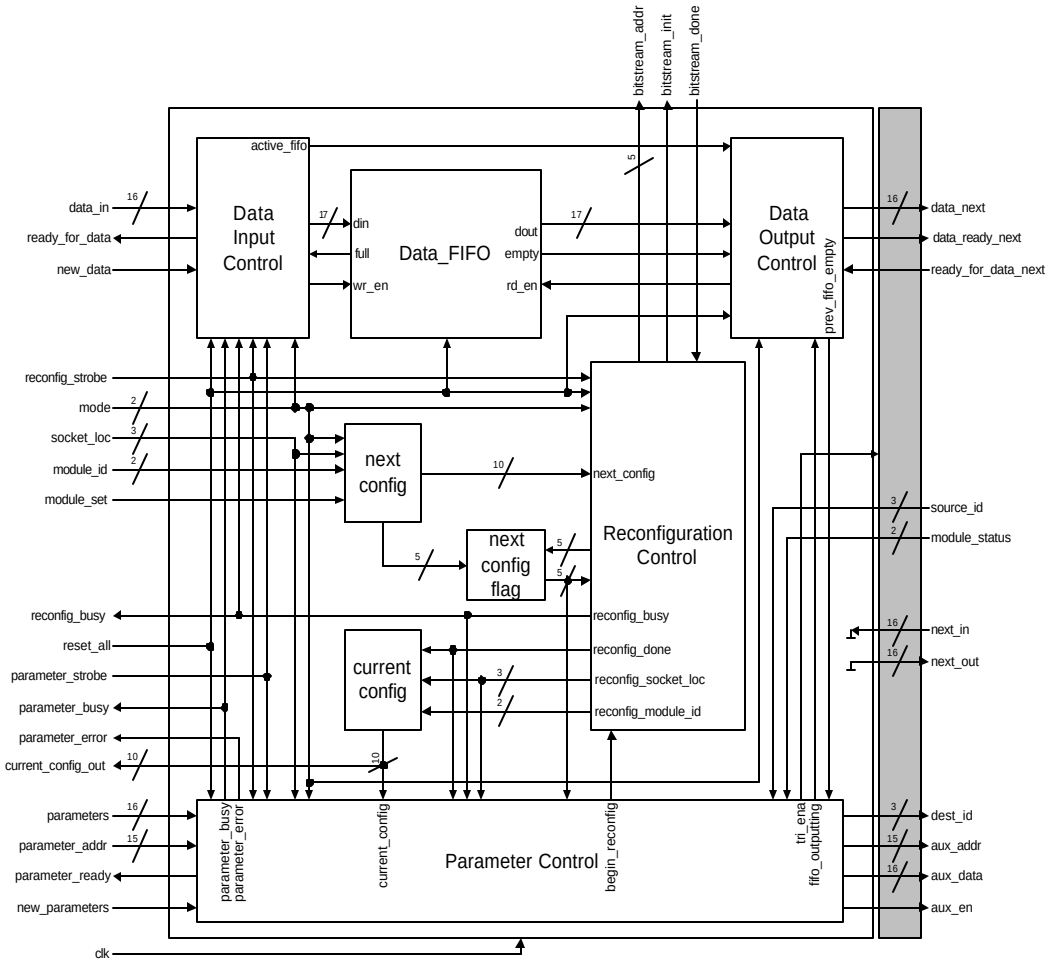

Figure 16 - Fixed Logic

Preceding the descriptions of each sub-module, a number of general-purpose input signals should be detailed. Like all other regions of the FPGA, the fixed logic module runs synchronously with the rising edge of the input clock signal, referred to as clk. The reset_all signal, with a single exemption in the current configuration register, effectively resets the device to a default operational state. The mode input defines the current operational mode of the device and can be externally set to stop, run, partial reconfiguration, or reparameterization modes.

5.2.1. Data_Input_Control

The Data_Input_Control sub-module essentially handles the loading of 16-bit data into the data_FIFO based on a number of input control signals sent to the device, acting as the first step in the data flow management process. In simple

terms, if Data_Input_Control is ready for data and new data is present on the data input bus, Data_Input_Control will load this data into the data_FIFO. This loading process, however, must be coordinated with both external signals and internal signals asserted by other sub-modules. When in run mode, Data_Input_Control will manage data entering the data_FIFO by tracking the reconfig_strobe and parameter_strobe signals. These signals instruct the FPGA to initiate the reconfiguration or reparameterization processes, respectively. In the case that either strobe signal is asserted, Data_Input_Control must differentiate between pre-strobe and post-strobe data by adding an extra data bit to each 16-bit data input word, enabling the 17-bit data_FIFO to virtually act as two separate 16-bit data_FIFOs. For example, after a system reset, data entering the system will have a 0x0 set as the 17^{th} data_FIFO bit to indicate that the data occupies the first virtual FIFO. After a reconfig_strobe or parameter_strobe signal, however, 0x1 will be appended as the 17^{th} bit to indicate that this new data belongs in the second virtual FIFO. This distinction between old and new data will allow the system to process all old data before changing the system. If another strobe signal occurs after the configuration change is made, the 17^{th} bit will toggle back to 0x0 and the process will repeat. The active_fifo signal will reflect which virtual FIFO is currently in use. In order to prevent subsequent reconfig_strobe or parameter_strobe signals from confusing the Data_Input_Control by beginning an illegal FIFO toggle operation, the reconfig_busy and parameter_busy signal asserted by other sub-modules are checked. Finally, Data_Input_Control can be returned to a default state if the reset_all signal is asserted.

5.2.2. Data_FIFO

In order to enable the run-time aspect to this FPGA design, data is stored in data_FIFO queue during either reconfiguration or reparameterization. The goal of this feature is to decrease the overhead of reconfiguration or reparameterization by allowing Teradyne's tester to continuously transmitting digitized waveforms into the FPGA. The system will rely of the FPGA's

ability to process the data with multiple separate configurations. The FPGA would then utilize pauses in input data flow to complete processing of current data and empty out the data_FIFO. The data_FIFO queue is implemented as a 16Kx17-bit FIFO, with the 17th bit acting to distinguish between two virtual 16-bit FIFOs. On the input side of the FIFO, the fifo_full signal will be used by Data_Input_Control to determine whether the FPGA is ready for data. Also, the new data input to Data_Input_Control will be used to assert the fifo_wr_en signal that enables the writing of data to the FIFO. Likewise, the fifo_empty signal is used by Data_Output_Control to determine whether data_ready_next can be asserted. Finally, the ready_for_data_next signal sent to Data_Output_Control by DSP Socket_1 dictates whether fifo_rd_en is asserted to read data from the FIFO.

5.2.3. Data_Output_Control

When in run mode, the Data_Output_Control sub-module acts as the final stage of data flow control within the Fixed_Logic module, reading data from the data_FIFO and sending the output data to DSP Socket_1 as long as the data_FIFO is not empty and DSP Socket_1 is ready for data. The complexity of this sub-module arises from determining which of the two virtual FIFOs is active and transmitting the appropriate output data. While the active_fifo signal emitted by Data_Input_Control conveys the current active FIFO to Data_Output_Control, older data may still exist in the data_FIFO that must be transmitted to DSP Socket_1 prior to any FPGA configuration or parameter modifications. Therefore, once active_fifo changes from its previous value, Data_Output_Control will continue to output the previous FIFOs data from data_FIFO until the 17th data bit matches active_fifo, indicating that all old FIFO data has been flushed. At this time, Data_Output_Control will assert the prev_fifo_empty bit to notify the Parameter_ Control sub-module that it may begin reconfiguration or reparameterization. Data_Output_Control will then monitor the fifo_outputting signal asserted by the Parameter_ Control sub-module and, when fifo_outputting matches active_fifo, Data_Output_Control resumes the transmission of data from data_FIFO to the first DSP socket.

5.2.4. Next_Configuration and Next_Configuration_Flag

The next_config and next_config_flag registers are the first stages related to the partial reconfiguration of the system. Prior to a reconfig_strobe signal initiating partial reconfiguration, the system must be informed of which DSP modules are to be loaded into the DSP sockets. The next_config registers perform this function based on the mode, socket_loc, module_id, and module_set input signals. The next_config registers consist of five 2-bit registers, one for each of the five DSP sockets. As will be described later in this chapter, four DSP modules have been designed for this feasibility study, each corresponding to a 2-bit value. If in either run mode or partial reconfiguration mode, the external assertion of module_set will result in the value of module_id being stored in the next_config register indicated by socket_loc. Values of socket_loc outside of the range of one through five inclusive are invalid and are ignored by the Fixed_Logic module. Since multiple configuration changes can be initiated by a single reconfig_strobe signal, any or all of the next_config registers can be modified prior to reconfiguration. In the case of a reset signal, the values of next_config will default to the current configuration of the system, as indicated by the current_config registers.

The next_config_flag registers exist as a 5-bit array that informs the Reconfiguration_Control sub-module as to which next_config registers have been modified. This information is then used to determine which DSP sockets are to be reloaded with next DSP modules during partial reconfiguration. In the case that only DSP Socket_1 is to be reconfigured, for instance, next_config_flag[1] will be asserted while the other bits remain constant. Once the Reconfiguration_Control sub-module has made the necessary reconfigurations, it will de-assert the affected next_config_flag bits.

5.2.5. Current_Configuration

Identical in size to the next_config registers, the current_config registers reflect the current configuration loaded into the five DSP sockets. In addition

to being used internally by the Parameter_ Control sub-module, these five 2-bit values are also routed to output buffers allowing externally connected devices to monitor the internal configuration of this FPGA. Only the Reconfiguration_Control sub-module can modify the current_config registers, reflecting any configuration changes made to the system. Like the next_config register loading mechanism, the value of reconfig_module_id will be loaded into the appropriate current_config register as dictated by reconfig_socket_loc and the reconfig_done strobe. Unlike all other sub-modules of the Fixed_Logic module, a reset signal while in run mode will not change the current_config registers as they must continue to accurately reflect which DSP modules are loaded into each of the five DSP sockets. Only a reset signal asserted while in stop mode can override this safeguard.

5.2.6. Reconfiguration_Control

While the previously mentioned sub-modules have not been especially complex in functionality, the Reconfiguration_Control sub-module is more convoluted as it must accurately coordinate the reconfiguration process based on a vast array of input status signals and registers. Simply, the Reconfiguration_Control sub-module, when instructed via reconfig_strobe, must determine which sockets are to be reloaded with new DSP modules, coordinate the disabling and re-enabling of any affected DSP sockets with the Parameter_Control sub-module, and output the necessary bitstream control signals to facilitate partial reconfiguration. The following paragraph will more accurately define this operation in greater detail, in accordance with Figure 17.

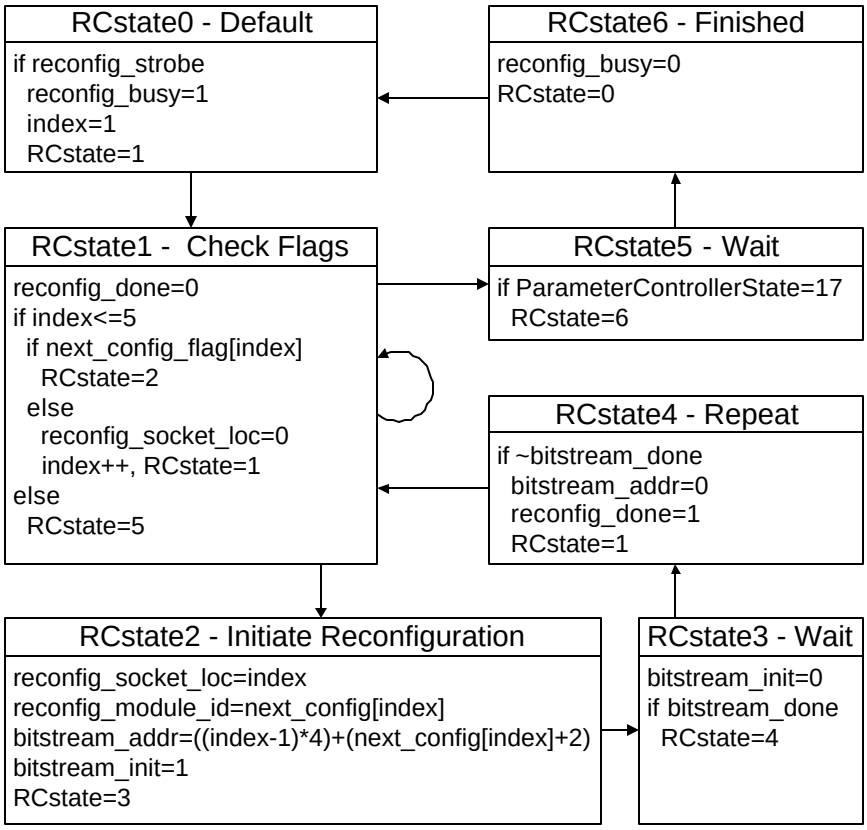

Figure 17 - Reconfiguration_Control FSM

As cited, the partial reconfiguration process is commenced by the assertion of the reconfig_strobe signal while in the appropriate mode, resulting in the immediate activation of the reconfig_busy signal. At this time, the Reconfiguration_Control sub-module must wait for the Parameter_Control sub-module to properly deactivate the affected DSP sockets, a process that will be explained in the next section. Once this deactivation process is completed, as indicated by the begin_reconfig signal declared by the Parameter_Control sub-module, Reconfiguration_Control will utilize next_config_flag to determine which DSP socket to begin reconfiguring. In the case that multiple DSP sockets are to be reconfigured, Reconfiguration_Control will start with the leftmost socket. As each socket is undergoing reconfiguration, the socket location will be reflected in the reconfig_socket_loc signal. This reconfig_socket_loc signal in conjunction

with reconfig_busy will prevent any other operations from interfering with partial reconfiguration.

Along with internally coordinating this reconfiguration process, Reconfiguration_Control must also output the appropriate bitstream address and initialization signals, which will instruct an external bitstream storage device to load a particular bitstream. For this feasibility study, each of the four sample DSP modules has a separate bitstream address for each of the five DSP sockets. Once the bitstream has been loaded into the appropriate socket, the bitstream_done signal will inform reconfiguration control that it may proceed with the next stage of the reconfiguration process. Reconfiguration_Control then asserts the reconfig_done signal and, if necessary, repeats the process by triggering the partial reconfiguration of another DSP socket. Once all desired reconfiguration changes have been completed, Reconfiguration_Control will wait for the Parameter_Control sub-module to reactivate the appropriate DSP sockets, terminate the reconfig_busy signal, and return to a default state.

5.2.7. Parameter_Control

By far the largest and most complex facet of the fixed logic module, the Parameter_Control sub-module revolves primarily around a 41-state finite state machine that control the operational status of all five DSP sockets in combination with both reconfiguration and reparameterization operations. Figure 18 illustrates this mechanism. In the case of either a reconfig_strobe or parameter_strobe, this module first initiates a shutdown sequence designed to involve all sockets to the left of the rightmost DSP socket targeted by the respective operation. The rightmost-targeted DSP socket would be defined as the most significant next_config_flag bit asserted in the case of reconfiguration or the value of source_id in the case of reparameterization. Using the aux_bus as a transmission medium, this module will send signals instructing these target sockets to finish processing all pre-strobe data. After Data_Output_Control notifies Parameter_Control that the previous FIFO is

empty, Parameter_Control will then wait for the necessary sockets to return a status signal indicating that the socket is finished processing data and ready for reconfiguration or reparameterization. This status signal is returned via the module_status and source_id signals, also collectively known as status_bus, which indicate the type of status message and the socket source of the message, respectively. For example, take the case that DSP Socket_3 was to be reloaded with a new DSP module. Sockets 1, 2, and 3 would all be instructed to finish processing and send the appropriate deactivation notification back to Parameter_Control. After prev_fifo_empty is asserted and all three status signals are received, Parameter_Control would then instruct Reconfiguration_Control to begin reconfiguration. In the case of a parameter loading operation, the shutdown and notification sequence would be identical except that the sequence would be followed by parameter loading rather than partial reconfiguration.

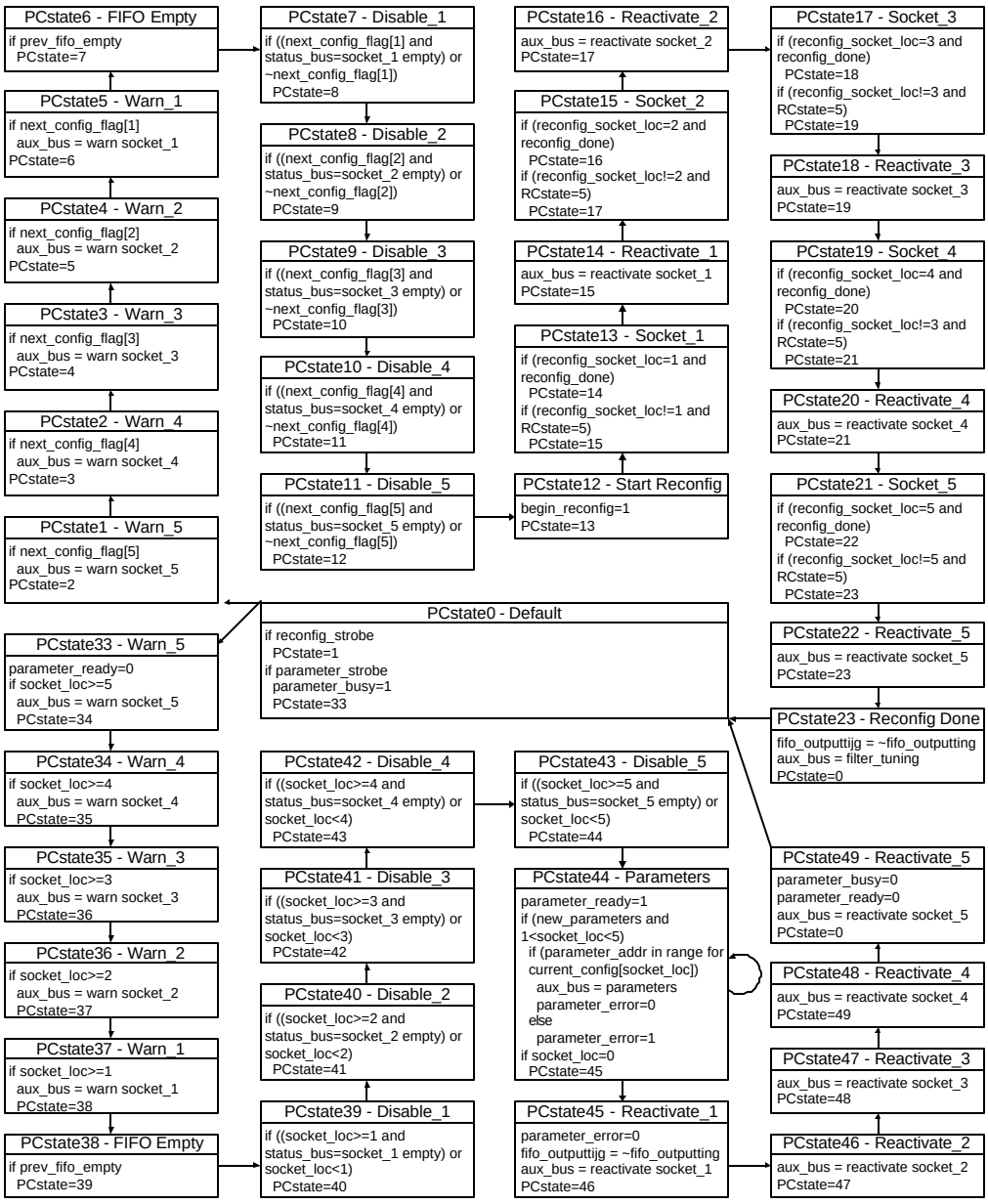

Figure 18 - Parameter_Control FSM

The aux_bus used to transmit both instruction and parameters to the DSP sockets consists of four separate signals. The 3-bit dest_id signal designates which DSP socket should receive the signal. While each DSP socket will forward the aux_bus signals to the next socket, each socket is responsible for whether or not to heed the command or parameter. A dest_id value of 0x7 will be received and acted upon by all five DSP sockets. The 16-bit aux_data signal will reflect either parameter data or command data depending on the

situation. The 15-bit aux_addr signal allows the DSP sockets to differentiate between commands and parameters. The address space for both circumstances will be described later in this chapter. Finally the assertion of aux_en will inform the DSP sockets that valid data is present on the aux_bus.

In the instance of a parameter loading operation, the parameter_busy signal would be immediately triggered to both notify external devices and prevent multiple parameter_strobe signals from illegally disturbing Data_Input_Control. Once the necessary socket shutdown sequence has been completed as indicated by source_id, parameter control will then assert the parameter_ready signal to indicate that parameters can now be loaded. The 16-bit parameters signal is used for data, the 15-bit parameter_addr bus used to address specific memory locations within a DSP socket, source_id indicates the target DSP socket, and new_parameters indicates that this data is available for transmission to the DSP sockets. If parameters are to be loaded into multiple DSP sockets as indicated by source_loc, they must be loaded into the rightmost socket first. During this parameter loading process, parameter control will compare the parameter_addr and source_loc of each parameter to current_config to confirm that it is within the acceptable address space of the target DSP module. If an illegal address is entered, the parameter will be ignored and the parameter_error signal will be triggered. Once all parameters have been loaded, externally asserting source_loc to 0x0 will allow Parameter_Control to reactivate the affected sockets and continue data processing operation. While dest_id will reflect source_loc, aux_data reflects parameters, and aux_en reflects new_parameters, respectively, aux_addr does not merely forward parameter_addr, as the address space for each type of DSP module is unique. Again, the particulars of this address transformation will be described in the DSP Modules section.

If reconfiguration is occurring, Parameter_Control will assert begin_reconfig to command Reconfiguration_Control to begin the reconfiguration process.

During this process, Parameter_Control will monitor both reconfig_done and reconfig_busy to determine when it may proceed with the next stage, as described below.

Once either the parameter loading or partial reconfiguration processes have been completed by the respective control elements, Parameter_Control must reactivate the necessary DSP sockets in preparation for resumed data flow. Beginning with DSP Socket_1, each necessary socket will be instructed via the aux_bus to reactivate itself. In addition to the previously stated uses of the current_config registers, the type of DSP module in each location is also used by Parameter_Control to construct filter_tuning information. In the case that a FIR filter is loaded into only DSP Socket_1, for instance, it will be given a filter_tuning value of 0x1, instructing it to act as a single 32-tap FIR filter and transmit its truncated sum output to the next DSP socket. In the case that two FIR filters are placed in neighboring DSP sockets, however, Parameter_Control notices this characteristic and modifies the filter tuning such that the first FIR is given a filter_tuning of 0x2 and the second a filter_tuning of 0x3, indicating that the filters should operate as the first of two and second of two filters, respectively. As the current design stands, if two 32-tap FIR filters are placed adjacent to one another, they will automatically be combined into a single 64-tap FIR. After all necessary DSP sockets have been reactivated, Parameter_Control calculates the 2-bit filter_tuning of each of the five DSP sockets and transmits the resulting 10-bit filter_tuning word via the aux_bus with a dest_id of 7, which instructs all DSP sockets to heed the data. The particulars of each filter_tuning value for each DSP module type will be specified in the following section. Once the filter_tuning word has been transmitted to the DSP sockets, Parameter_Control will return to a default state and await the next reconfig_strobe or parameter_strobe signal.

5.3. DSP Modules

The three separate DSP algorithms detailed in Chapter 3 have been implemented for this thesis project. As part of the flexible socket architecture designed, each

module conforms to a defined and standardized interface that allows each to communicate with neighboring modules. In order to simplify both the design and verification processes, a fourth Empty module has been implemented that contains a basic module skeleton on which all DSP filters are built. The next section will outline the Empty module, followed by implementation-specific details for each of the three remaining DSP modules.

5.3.1. Empty Module

The Empty module contains all of the components necessary to act as a template on which DSP filters can be built. First, the Empty module contains all of the connections necessary to interface with the bus macro boundaries between DSP sockets. The primary rightward-moving data flow elements consist of 32-bit data inputs and outputs along with the corresponding data_ready and ready_for_data status signals. Next, the Empty module accepts aux_bus inputs on the left side of the module and forwards those signals unaltered to the next DSP socket. In the opposite direction, the module accepts status bus signals from the right side of the module and routes those signals leftward towards the Fixed_Logic module. Also, the module contains 16-bit data busses traveling to and from both the previous and next module. While unused for the Empty module, these secondary data pathways are used to connect identical neighboring filters, such as in the creation of a 64-tap FIR filter by automatically combining two neighboring 32-tap FIR filters, as dictated by the filter_tuning registers. Besides the obligatory reset signal connected independently to each module, the Empty module also contains tri-state signals used to enable and disable bus macro interconnects on both sides of the module. These tri-state enabling signals are disengaged prior to reconfiguration in order to prevent signal contention.

In addition to the standard interconnects, the Empty module also contains a logical frame that controls the module status as dictated by the Fixed_Logic module via the aux_bus. A series of internal registers track the current status of the module along with the both the previous module's status and the next

module's status. In the case that reconfiguration or reparameterization is occurring, the internal module status can be altered to instruct the module to finish processing all old data if an aux_addr of 'h0002 accompanies and aux_data value of 'h0001 and a matching dest_id. Both the previous and next modules will also note this data on the aux_bus. Once the module finishes processing old data, it will send a notification signal to the Fixed_Logic module via the status_bus while simultaneously notifying the next module of the change via the aux_bus. In the case of impending reconfiguration, the tri-state enable signals are disengaged after the aforementioned notifications have been transmitted. Other than the finite state machine that controls these module status interactions, the Empty module also contains logic to read filter_tuning information transmitted from the Fixed_Logic module via the aux_bus and make that filter_tuning information available to any DSP filter built on top of the Empty module.

While the logic described can be used in any of the modules, the Empty module also contains simple routing to simply forward data from left to right without alteration as long as the next module is ready for data and the previous module has data ready to transmit. Likewise, the next and previous secondary data busses are connected to zero, as they have no purpose in an Empty module.

5.3.2. FIR

The 32-tap even symmetry FIR filter has been implemented to match the description given in Chapter 3. As seen in Figure 19, the least significant 16-bits of the 32-bit data input are entered into a tap shifting mechanism that, given a single operation filter_tuning, shifts the data along until all 32 taps are filled. Once this process is completed, the signed 16-bit data values of each tap pair are summed to produce a 16-bit signed tap value that is then multiplied by the 16-bit signed coefficient value connected to each multiplier. Rather than use a multiplier constructed using standard FPGA logic resources, the 18-bit fast, embedded multipliers inherent in the FPGA fabric are utilized.

The 32-bit signed output values of each multiplier are then summed using a series of two-entry 32-bit signed adders until a single 32-bit signed sum is ready for truncation to 16-bits and transmission to the next module.

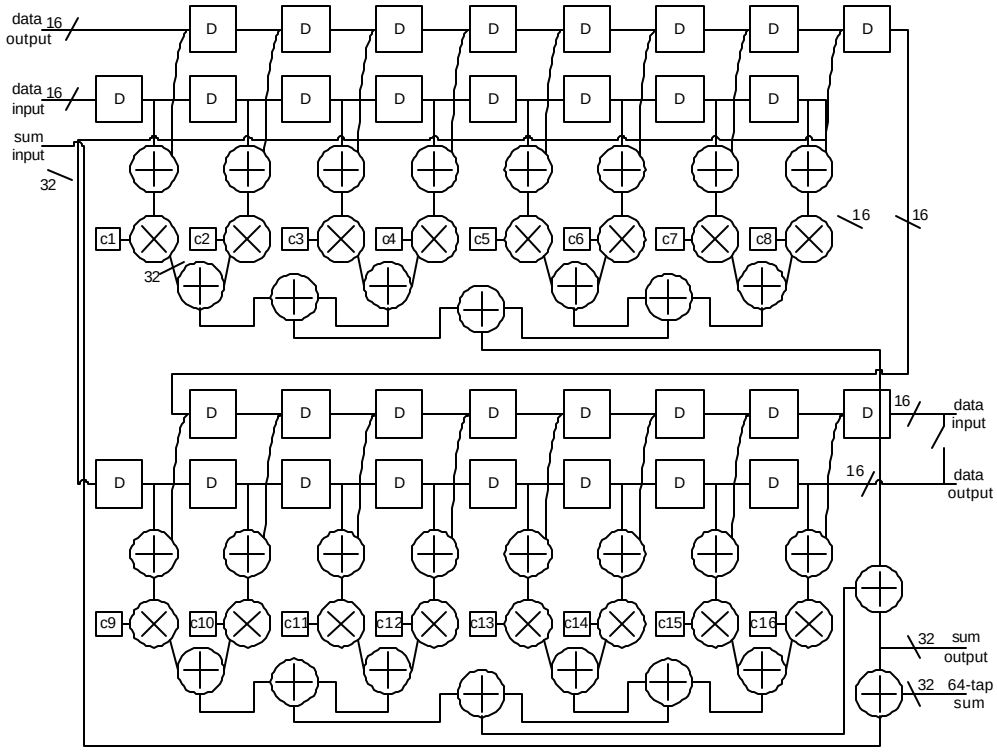

Figure 19 - 32-tap FIR Implementation

In addition to the mathematical aspect of this FIR filter, data flow control has been designed to speed the processing of data in the FPGA. When all 32 taps are empty, the filter will emit a ready for data signal to the previous module regardless of whether or not the next module is ready for data. Once the taps are full, however, the ready_for_data signal sent to the previous module will correspond to the ready_for_data signal emanating from the next module. The data_ready_next signal, which indicates to the next module that new data is present on the data_out bus, is delayed by seven cycles to allow for the computational delay in preparing the final sum output value. In the single filter_tuning configuration, the next and previous secondary data busses are not utilized.

When two 32-tap FIR filters are combined into a single 64-tap FIR filter, the default configuration when two 32-tap FIR filters neighbor each other and as determined by the filter_tuning value sent from the Fixed_Logic module, the tap shifting mechanism is altered to permit proper sum computation. Figure 20 illustrates the method described below. For the first filter in a two-filter configuration, data is shifted from tap_1 toward tap_16 in a normal manner. Rather than shifting the value from tap_16 to tap_17, however, the tap_16 value is sent along the 16-bit next_out bus to be used by the second filter. Correspondingly, the tap_17 value is read off of the 16-bit next_in bus and shifted towards tap_32. Once all of the first filter's 32 taps are filled, the filter will compute a 32-bit sum value and transmit the entire non-truncated value to the second filter in the sequence using the data_out bus. In this configuration, the data_ready_next signal relates to the presence of data on the next_out bus rather than data_out bus. The internal shift register that tracks whether or not all taps are full accounts for the full 64-tap design and delays the output of sum data accordingly.

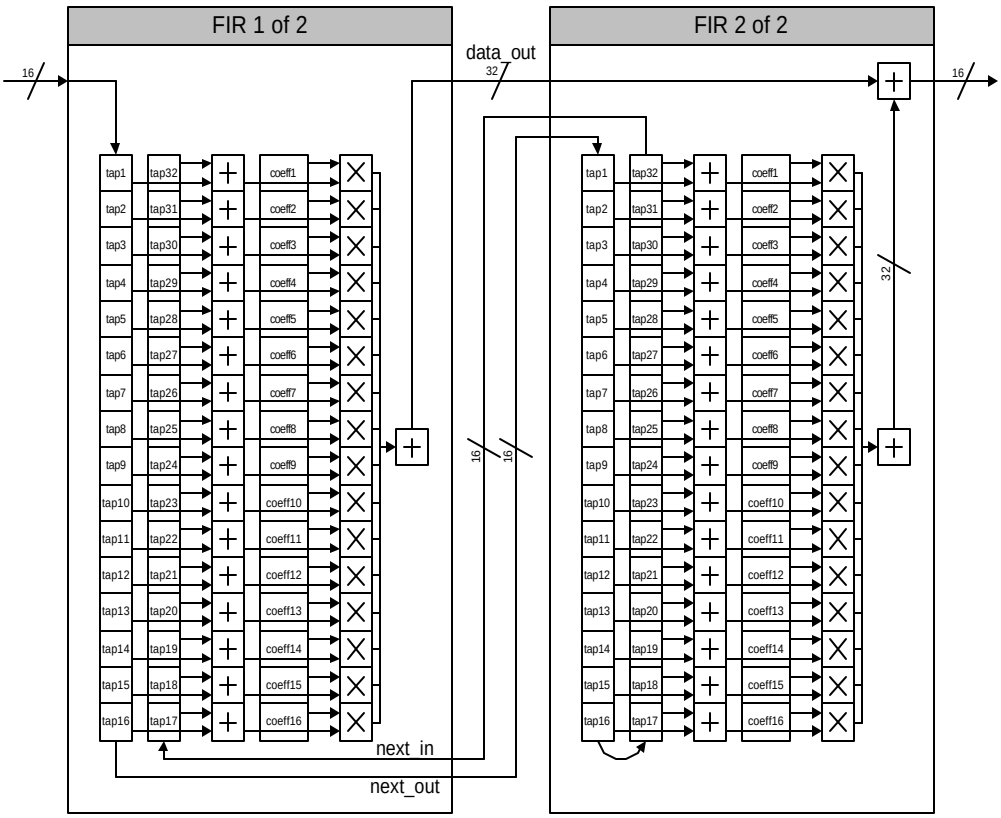

Figure 20 - 64-tap Two-filter Operation

The second filter of the two-filter sequence shifts data through its 32 tap registers as it typically would. In accordance with the scheme described in the previous paragraph, data is read off of the prev_in bus and written to tap_1. Likewise, data is read from tap_32 and written to the prev_out bus. When the taps are full and the second filter begins receiving 32-bit sum output values from the first filter, the second filter will add this value to the internally computed 32-bit sum value and output the truncated 16-bits as the final 64-tap output value.

The 16-bit coefficient values used in this tap filter are loaded as parameters via the aux_bus. Using the reparameterization process described previously, any or all of these values can independently be modified at any time without requiring reconfiguration. The sixteen coefficient values correspond in order to aux_addr values of ‘h4000 through ‘h400F, which correspond to

parameter_addr values of ‘h0000 through ‘h0000F, respectively. The modification of the 15th bit is performed by the Fixed_Logic module’s Parameter_Control sub-module in order to both simplify the external interface and complement the address space requirements of other DSP filters, particularly the Time-varying Coefficient FIR filter.

5.3.3. Quadrature Mixer

The Quadrature Mixer, which only operates in a single filter_tuning configuration is designed to produce 16-bit sine and cosine values using a built-in numerically controlled oscillator, independently multiply both values by the incoming 16-bit input data, and truncate the 32-bit results down to 16-bits each for output to the next processing stage. In order to ease the implementation of this filter, a Direct Digital Synthesizer IP core provided by Xilinx will be used as the numerically controlled oscillator. Using a 32-bit phase increment value that can be set as a parameter via the aux_bus, the NCO will cycle through the phase accumulator address space used by the sine and cosine lookup tables. Using quarter wave symmetry, which adds 2-bits worth of lookup table addressing accuracy, and the 16K entry BlockRAM capacity of the DSP socket, which is addressed by 14-bits of the phase accumulator value, 16-bit sine and cosine values can be derived from the NCO. Two embedded multipliers are then used to create separate 32-bit in-phase and quadrature output values. Given the data output bandwidth limitation of 32-bits, each 32-bit value is then truncated down to 16-bits for output. In order to accommodate the three-cycle latency of the NCO, the data input values are pipelined accordingly.

The data control characteristics of this filter are quite basic as the ready_for_data_prev signal simply reflects the ready_for_data_next signal. The data_ready_next signal accounts for the computational latency of the NCO and multipliers when data_ready_prev is active. This filter does not use either the next or previous data busses. Finally, the 32-bit phase increment value is loaded as two separate 16-bit parameters due to the data bandwidth

limitation of the aux_bus. A parameter_addr value of ‘h4000, which translates to ‘h3FFE on aux_addr, will write the lower 16-bits of the delta value while a parameter_addr value of ‘h4001, translating on aux_addr to ‘h3FFF, will write the upper 16-bits. The parameter addresses ranging from ‘h0000 to ‘h3FFF were originally allocated to sine and cosine lookup table values, but were not needed due to the utilization of the Xilinx IP core for the NCO.

5.3.4. Time-varying Coefficient FIR

The computational elements of the Time-varying Coefficient 32-tap even symmetry FIR filter is constructed identically to the previously described FIR filter with the primary exception of the coefficient storage system. Like the regular FIR filter in single-filter operation, the Time-varying Coefficient FIR will shift data until its 32-taps are filled and then compute the output sum value. The ready_for_data and data_ready status signals operate like the regular FIR filter and the 32-bit sum output value will also be truncated down to 16-bits.

Instead of a single 16-bit coefficient register connected to each embedded multiplier, each multiplier is linked to the data output of a 1K-entry coefficient storing BlockRAM, referred to as coeffRAMs. These 16-bit coefficients are also loaded via the aux_bus albeit with a significantly larger address space. Parameter_addr values of ‘h0000 through ‘h3FFF are translated in the Fixed_Logic module to correspond to aux_addr values of ‘h4000 through ‘h7FFF. Parameter addresses from ‘h0000 to ‘h03FF correspond to the coeffRAM_1, addresses from ‘h0400 to ‘h07FF correspond to coeffRAM_2, and so forth. Like the phase accumulator present in the Quadrature Mixer’s NCO, a 32-bit delta value loadable through the same parameter addresses is used to increment a 32-bit address accumulator, the most significant 10-bits of which are used to address all sixteen coeffRAMs in unison.

Rather than support the capability of combining two 32-tap Time-varying Coefficient FIR filters into a single 64-tap Time-varying Coefficient FIR, this filter is designed with a distinctly different filter combination scheme, as dictated by the filter_tuning register. As a single filter, the Time-varying Coefficient FIR will take the lower 16-bits of the input data, perform the necessary processing, and output the truncated sum as the lower 16-bits of data_out. As an additional feature, if a Quadrature Mixer filter is followed by two adjacent Time-varying Coefficient FIR filters, the two Time-varying Coefficient FIRs will be automatically set by filter_tuning to work concurrently to independently process both in-phase and quadrature data. As seen in Figure 21 below, the first of two Time-varying Coefficient FIR filters will utilize the lower 16-bits of data_in, process the data through the filtering mechanism, and output the truncated output sum via data_out. Simultaneously, the first filter will also immediately make the upper 16-bits available to the second filter by way of the next_out secondary data bus. Meanwhile, the second filter will accept this data from the first filter through the prev_in bus and process this data through its 32-tap Time-varying Coefficient FIR filter apparatus. Since both filters operate concurrently, the 16-bit sum output of the first filter will be computed in time with the 16-bit sum output of the second filter. The second filter will then combine both values and output the two 16-bit sums using the 32-bit data_out bus.

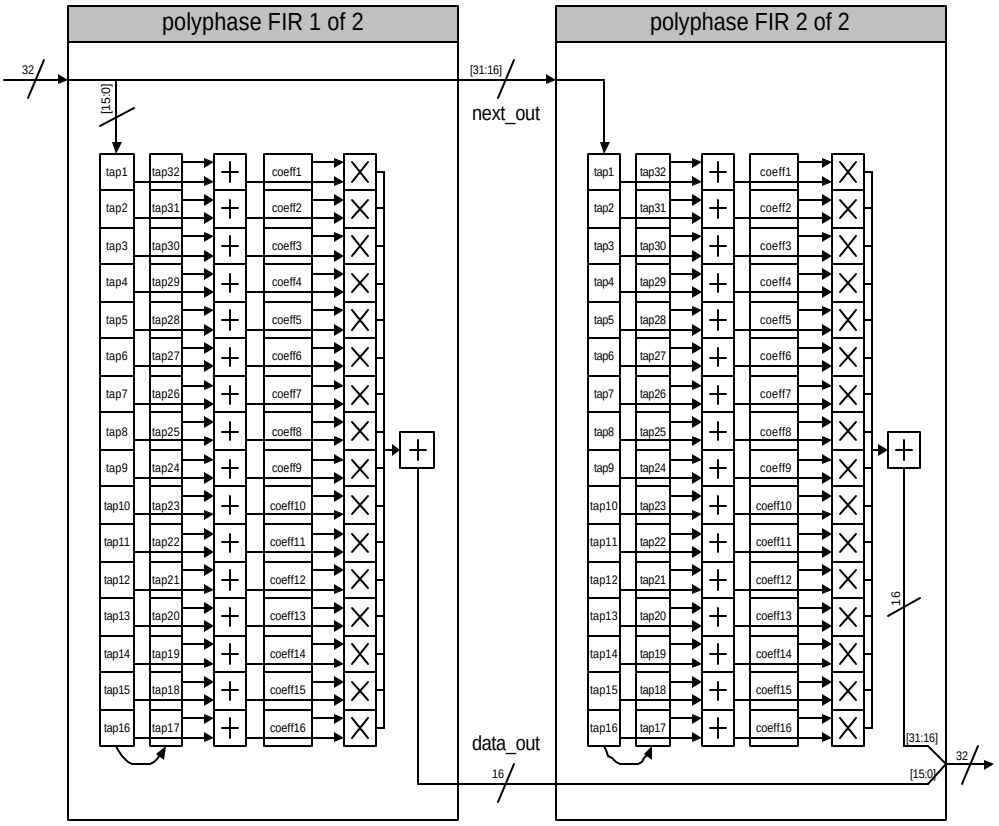

Figure 21 - Two Time-varying Coefficient FIR Operation

5.4. End_Logic Module

As the final stage of the DSP architecture of the FPGA, the End_Logic module is responsible for the simple task of making the data output of the DSP Socket_5 available externally. As long as the End_Logic module receives an external ready_for_data signal along with a data_ready signal from DSP Socket_5, output data will be conveyed on each cycle. This module also serves as the termination point of the aux_bus and the source of module status bus.

5.5. Bus Macros and Partially Reconfigurable Socket Architecture

The FPGA specifically chosen for this project, the XC2V3000-FG676-6, was used because its size and embedded feature set was well suited to the types of filters desired for this project. In addition to the three million usable system gates, the XC2V3000 contains six separate columns of 16 embedded multipliers and 16 18Kbit BlockRAMs each. Given that both FIR filters conveniently require 16

multipliers, the Time-varying Coefficient FIR requires 16 separate coefficient memories, and the Fixed_Logic module requires extensive data queue storage, this device offered an excellent embedded feature set while also offering more than adequate programmable logic and routing resources. Figure 22 shows that the sizing of each module was chosen such that the Fixed_Logic module and each of the five DSP sockets each contain a column of embedded multipliers and BlockRAMs. Due to the previously mentioned partially reconfigurable module sizing constraints, each of the DSP sockets must also be an even number of CLBs wide. Because the number of CLBs between each embedded multiplier column is not uniform, the DSP sockets are not all equal in size. As a result, the fifth DSP socket is ten CLB columns wide compared to eight CLB columns for the first four sockets.

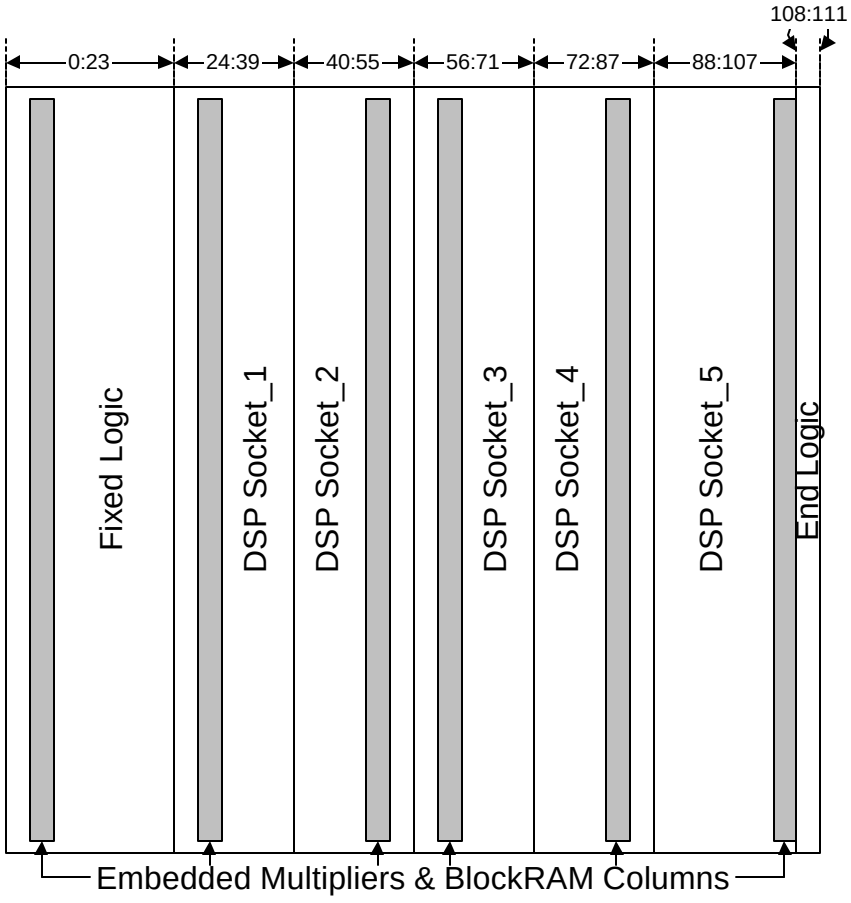

Figure 22 - Module Placement

Physically, each bus macro utilizes four tri-state buffers on each side of the module boundary and occupies an area one CLB high by four CLBs wide. For example, as seen in Figure 23, since the boundary between the Fixed_Logic module and the first DSP socket is between CLB columns 12 and 13, a bus macro across this boundary will occupy CLB columns 11 through 14 for that given CLB row.

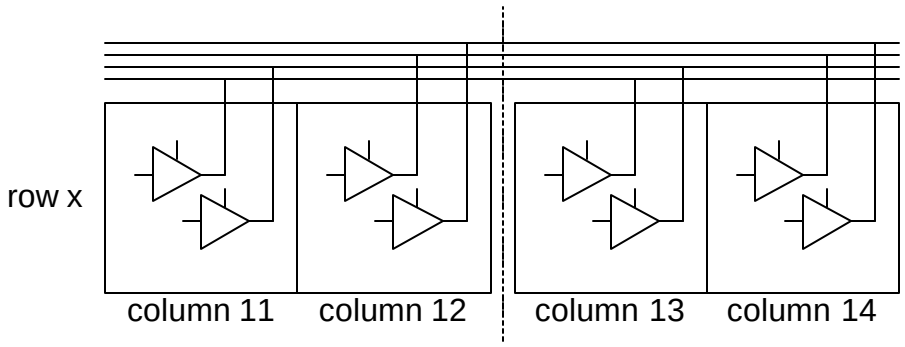

Figure 23 - Sample Bus Macro Placement Across a Module Boundary

Once the architecture is assembled, it becomes obvious that a large number of bus macros will be needed in between each module to satisfy the design's large data and status bus requirements. For this design, each bus macro boundary therefore consists of 106-bits of data, addressing, or status bits constructed using 28 separate bus macro instantiations. Additionally, each of these bus macros must be manually constrained to a specific location early in the design process. This manual location constraint has dramatic and noticeable effects on place and route performance and will be elaborated upon later in this document. An exception to this requirement for 28 bus macros occurs between the fifth DSP socket and the End_Logic module. Because these bus macro placements are at the rightmost possible location on the FPGA, physical limitations of the transmission lines used by the tri-state buffers limits each bus macro to 2-bits of usable rightward moving signal compared to the usual 4-bits. This limitation results in the use of 49 bus macros rather than 28. Since the device used contains a 64 CLB high array of logic and each 4-bit bus macro requires only one CLB row, the device is limited to a maximum of 64 bus macros in a single column, which can typically carry 256-bits, although not in the rightmost or leftmost physical locations. Also, in the

case of the 6th bus macro column, the bus macros simply bridge over 6th embedded multiplier and BlockRAM column to link the two rightmost CLB columns of DSP Socket_5 with the two CLB columns of the End_Logic module. Figure 24 illustrates the placement of each bus macro in the design. Now that the specification of the intended design has been described, the following chapter will elaborate on the implementation process.

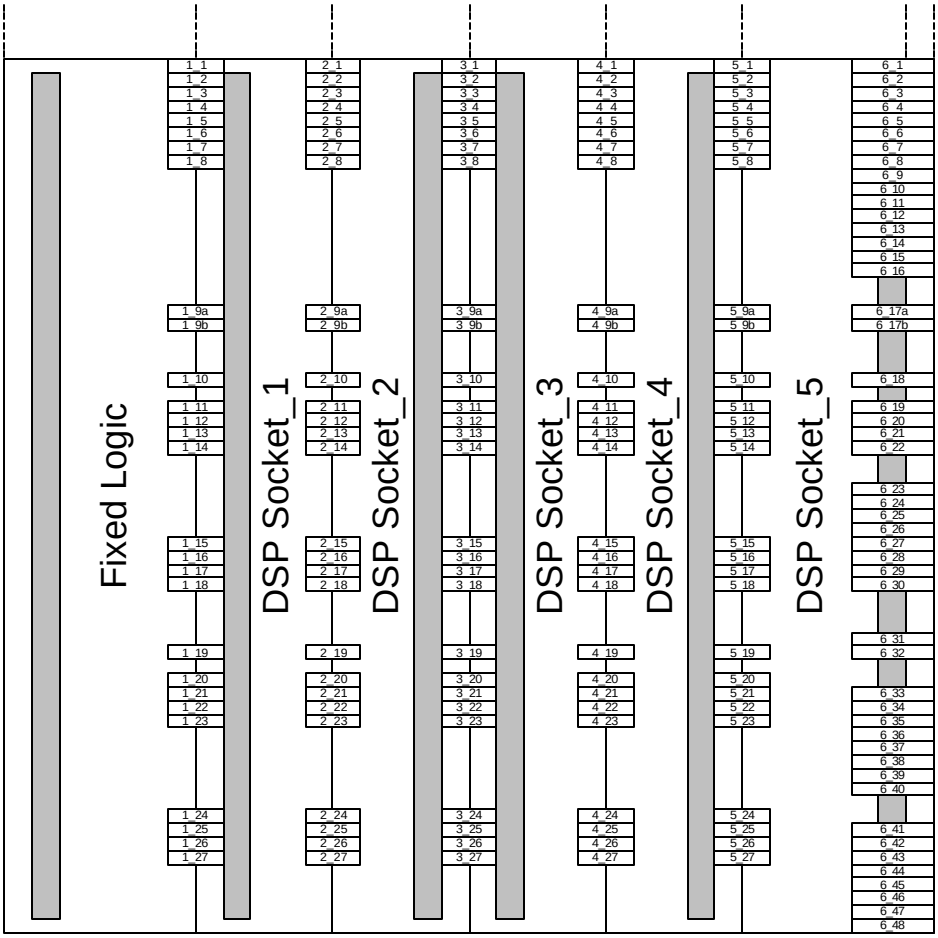

Figure 24 - Overall Bus Macro Placement

6. Design Implementation Process

The three-stage design flow process detailed below is the primary feature of the modular design flow as provided by Xilinx. The initial budgeting stage defines the top-level modular framework for the design. Next, the active module implementation stage allows each module, regardless of whether reconfigurable or fixed, to be fully designed and built to the extent that a module bitstream can be derived from the design. Thirdly, the final assembly stage incorporates the individually created modules into one or more assembled design permutations for both simulation and bitstream generation purposes. For all stages of this design, the requirement to meet a 10ns clock period and run at 100MHz is the only timing constraint. Simulation of the design, which will be discussed in the next chapter, can be performed during both the active module implementation and final assembly stages.

6.1. Initial Budgeting Phase

The partially reconfigurable design implementation process begins with the initial budgeting phase. The process starts with the usual verilog top-level module instantiation followed by the module instantiations for the Fixed_Logic, End_Logic, and DSP modules. Each interconnection between modules is accomplished in 4-bit increments using the bus macros. Unlike a typical FPGA design, however, the top-level design must also include manually placed constraints to fix the location and size of each module. Likewise, the embedded multipliers and BlockRAMs available for each module must also be confined. Using Xilinx's constraint file, or UCF, the placement constraints are defined according to the x-axis and y-axis slice coordinates of the module boundaries. Beyond module location, the bus macro boundaries between modules must also be manually placed in the UCF file. As each of the 189 bus macros used in the design occupies an area four CLBs wide, the location of the macro is defined by the leftmost slice occupied. These macros must also symmetrically straddle the module boundaries. Finally, the location of all external pin connections must be constrained such that each exists within the appropriate target module's margins.

If this were a typical FPGA design, the usual synthesis, translation, mapping, placement, and routing tools would be utilized in a rather automated fashion to build the final FPGA design. At this stage of development in this partially reconfigurable situation, however, only the top-level modules and interconnects are defined without any lower-level logic. As a result, special synthesis and translation commands must be used to prevent the tools from attempting to physically synthesize and model any internal logic. The build files created in this translation stage will be used in the active module implementation stage described below. Lastly, the use of mapping, placement, and routing tools at this stage offers the designer a glimpse at module locations, but does not offer any significant value since they do not account for any actual logic placement or related timing information.

6.2. Active Module Implementation Phase

The active module implementation phase occupies the bulk of the design cycle as it is in this stage that all fixed and partially reconfigurable modules are built. Each module must be constructed in its own project and directory structure to prevent overlap with other modules. This project structure starts with a copy of the top-level UCF file along with the translation build file, or NGO, created in the initial budgeting phase. The use of these files along with specific active module implementation commands conveys to the design tools that the module must be designed within the aforementioned location and timing constraints. While the Fixed_Logic and End_Logic modules each required only a single module version, each of the four DSP filters had to be implemented in each of the five possible DSP socket locations. To simplify the process, each DSP filter was created as a Socket_1 version with the intent of later replicating the design for sockets two through five.

Starting with the Fixed_Logic module, a moderate learning curve was encountered and surmounted in dealing with the Xilinx design tools, specifically in regard to the modular design flow. At first, only the top-level constraints were used without the addition of any module-specific timing constraints. The design

was entirely manually coded verilog with the exception of the use of a Xilinx IP core for the 16K-entry 17-bit data FIFO. After the expected debugging steps, the design placed and routed surpassing the required 10ns minimum clock period with a resulting minimum clock period of 8.821ns, or 113.3MHz. The module also utilized far fewer CLB logic resources than expected, requiring only 110 of the 768, or 14.3%, of the CLBs allotted, which amounts to only 3% of the total 3 million system gates available on the FPGA. If not for the timing and scope limitation of this thesis, it would have been prudent to resize the Fixed_Logic module to waste fewer resources. Fortunately, the Fixed_Logic module utilized all 16 of its allotted embedded BlockRAMs for use in the 16K-entry data FIFO. As indicated by the total equivalent gate count of 1.06 million gates and an actual utilization of less than 100,000 gates, the use of BlockRAMs rather than distributed RAM for the data_FIFO resulted in a dramatic system gate utilization savings.

Post placing and routing, a verilog model including back-annotated timing information of this module was derived for simulation and verification purposes. As will be discussed in the next chapter, it was quickly discovered through analysis of both the synthesized and placed and routed design that synthesis fan-out restrictions have a dramatic effect on whether or not the resulting built model actually operates. The End_Logic module, which does not contain any significant logic, was built in a similar manner and also easily satisfied the timing requirements while using only 13 of the 128 CLBs allotted. Due to the physical placement limitations of the bus macros with respect to the edge of the device, however, shrinking the size of the End_Logic module to better employ logic resources was not an option.

Next, the DSP modules were implemented beginning with the Empty module. The decision to completely build and verify this module before moving onto the more elaborate DSP filters greatly decreased backtracking and debugging time. The DSP Socket_1 version of the Empty module placed and routed to run at over

150MHz, again exceeding the then-specified timing requirements. The simple design required only 26 of 512 allocated CLB logic blocks with an equivalent gate count of only 1,599 gates. While testing the performance limitations of this module, it was found that the module could successfully place and route up to almost 200MHz. The synthesis, mapping, placing, and routing decisions made by the Xilinx tools to achieve this general timing specification, however, would have an adverse effect on the timing of a few important signals, as will be discovered in the final assembly stages of the design.

Unlike the Empty module, the FIR filter required additional design iterations in order to meet the timing requirements. First, a simple version of the FIR containing only the multiplier and accumulator structure along with basic 32 tap shifting mechanism was constructed to weigh the benefits of utilizing the embedded multipliers. Clearly, as seen in Table 2, the 79% savings in CLB utilization along with the 45% increase in speed realized with the use of embedded multipliers is especially promising for this design. After constructing the final version of the FIR filter, including the proper tap shifting and status signal mechanisms, the module employed 302 out of 512 CLBs available, resulting in a more respectable 59% logic utilization. Still, the multiplication and addition structures needed to be further optimized to achieve the 100MHz goal. After modifying the 16-bit and 32-bit adders to register only output data and not input data, the design began approaching specifications. Using the timing analyzer tool provided with the Xilinx ISE toolset, it was discovered that the fan-outs of some signals, particularly the reset and tri-state enabling signals, was far too high, resulting in a larger than desired delay. Unfortunately, as will be discussed in the verification phase, setting the fan-out synthesis guide significantly lower effectively disrupts the operation of the filter. Once an appropriate middle ground was determined, however, the FIR filter in DSP Socket_1 placed and routed to run at 102.8MHz.

Table 2 - FIR Comparison

	Embedded Multipliers	LUT-based Multipliers
CLB use	186	888
Logic levels	5	20
Max speed	83.7MHz	57.7MHz

Next, the Quadrature Mixer module was constructed fairly quickly due to the use of the Xilinx IP core Numerically Controlled Oscillator. With only minor optimization, the Quadrature Mixer placed and routed to run at 103.4MHz while exploiting only 73 of the 512 CLBs available. Again, this low 14.3% utilization of module CLBs results heavily from the use of 14 of the 16 embedded BlockRAMs for sine and cosine data storage along with two embedded multipliers for mixing, an equivalent gate count of 932,000 system gates. Without the embedded features of the Virtex-II FPGA, this filter would have required a module twice the size as the one apportioned.

Lastly, the Time-varying Coefficient FIR filter was implemented building off of the regular FIR with major modifications to filter tuning-related functionality, tap shifting mechanisms, and coefficient storage system. Starting with the optimizations made for the original FIR filter, the coefficient storage memories were created using 16-sets of a Xilinx IP core 1Kx16-bits single-port RAM. As these coeffRAMs and the associated accumulator addressing system operated faster than the computational components of the module, the Time-varying Coefficient FIR in Socket_1 placed and routed to run at 101.7MHz. Even with the use of all 16 embedded multipliers and BlockRAMs within the module, this filter required 332 of 512 available CLBs, resulting in 64.8% logic utilization and an equivalent gate count of 1,144,000 gates.

As the logic utilization data suggests, an average CLB utilization of only 46% for the three DSP filter indicates that there is definitely room for more elaborate filters within the given architecture. While moving to a smaller FPGA is an option if higher CLB utilization is desired, that move would also come at the cost of fewer embedded multipliers and BlockRAMs. Furthermore, limited gains

would be realized from resizing the module boundaries since each of the five DSP sockets and the Fixed_Logic module require an embedded multiplier and BlockRAM column. Given the space between the 3rd and 6th embedded columns and the requirement that modules be an even number of CLBs wide, for example, it would not be possible to increase the size of all sockets to ten CLBs wide, as seen in Figure 25.

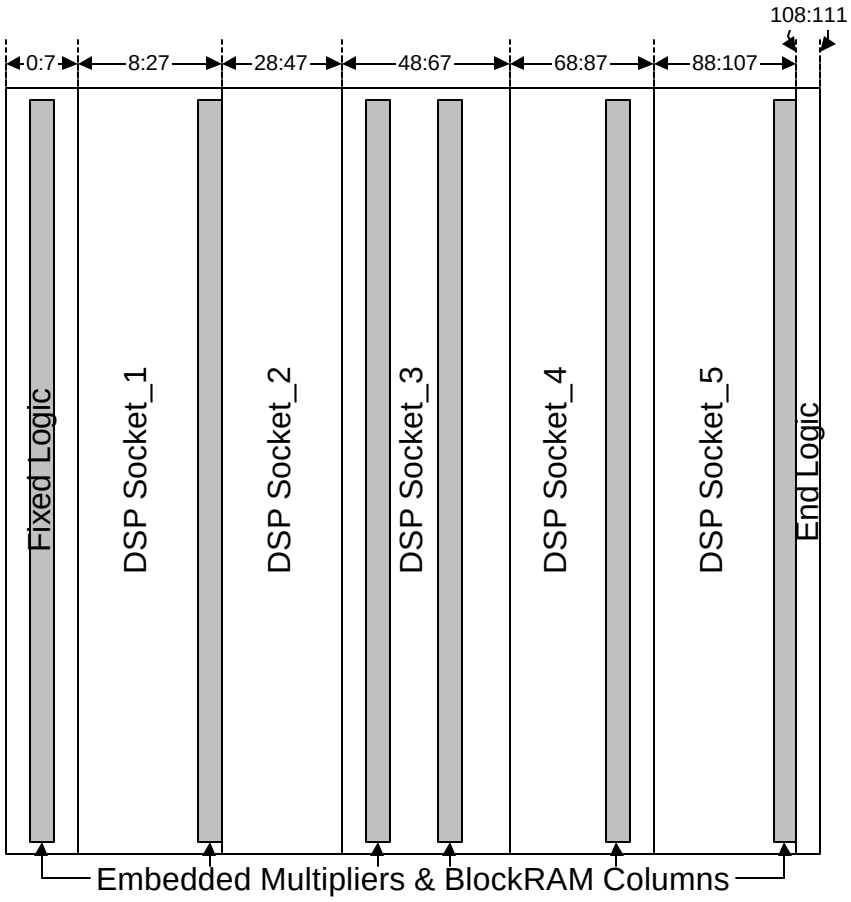

Figure 25 - Sample Module Resizing

Once all four DSP modules were constructed, each needed to be replicated to create a copy for each of the five DSP sockets. Quickly, it became apparent that any changes in socket layout, such as the placement of the embedded multiplier and BlockRAM column within the DSP socket, could have an adverse effect on placed and routed timing. In the case of the FIR filter, modules for DSP sockets 1, 3, 4, and 5 met the 100MHz requirement while DSP Socket_2 failed by 2MHz.

While the problem was easily corrected by modifying the structure of the mechanism that detected when all 32 taps were filled, the experience does raise a significant concern about socket layout regularity. In the case that a larger library of DSP filters were designed for this architecture, a suitable timing buffer accounting for layout irregularity must be created to differentiate between the actual desired speed of the device and the speed at which the first socket will place and route. Given the proximity of the rightmost embedded multiplier and BlockRAM column to the edge of DSP Socket_5, it was expected that routing issues could be encountered in the place and routing of a filter into that socket. Since all 16 embedded multipliers and BlockRAMs are used for the Time-varying Coefficient FIR filter, for example, it was incorrectly anticipated that the switching matrices connected to this embedded column would not have enough bandwidth to connect to the rest of Socket_5. This, however, was not the case and all twenty DSP module-socket combinations placed and routed to meet the 100MHz specification. Table 3 shows the final place and route maximum clock rates generated.

Table 3 - DSP Module Timing

	Socket_1	Socket_2	Socket_3	Socket_4	Socket_5
Empty	159.1	138.2	152.5	159.5	182.6
FIR	102.8	101.9	100.2	102.5	102.5
Quadrature	103.4	104.1	104.4	104.5	103.4
Polyphase FIR	101.6	100.9	102.5	103.1	102.2

6.3. Final Assembly Phase

After completely implementing each of the individual modules, the task turned to creating various permutations of the final design for verification and benchmarking purposes. Due to the time constraints of the thesis project, only three different permutations were created. The first version, referred to as Top_1, consists of the standard Fixed_Logic and End_Logic modules along with five Empty modules filling the five DSP sockets. Top_1 provided a platform on which the general operation of overall architecture could be observed without the added complexity of functioning DSP. The implementation process benefited

from a feature of the modular design flow enabling the design tools to build on the place and routing work performed during active module implementation when creating an assembled build. Rather than placing and routing the entire design from scratch, the toolset would utilize the place and route output files of each individual module and simple interconnect the routing and generate the necessary timing analysis. Once combined, Top_1 mapped out to use only 230 of the 3,584 CLBs, or 6.4% logic utilization, and an equivalent gate count of 1.06 million system gates due to the BlockRAM data_FIFO. While this low utilization was not surprising since this design does not contain any DSP functionality, the place and route timing result of 80.9MHz was both unexpected and discouraging.

Unfortunately, an unforeseen timing issue related to the tri-state buffer interconnects became apparent during this final assembly process. At the onset of the design process, the requirement to run at 100MHz was the only timing constraint considered. Each module was explicitly designed with all output signals stored in registers. As each module met this simple 100MHz requirement in the active module implementation phase, it was incorrectly assumed that connecting these modules would not have any significant effect on timing. Due predominantly to the unexpectedly significant tri-state buffer delay of 3.593 ns, the total delay from the output of one module to the input registers of the next reached a maximum of 12.366ns. Given that this is the simplest assembled permutation of the architecture, the speed of the Top_2 and Top_3 was now expected to be even worse. These larger than desired routing delays, however, may be correctable through the employment of additional stringent timing constraints at the module level and some redesign of the internal logic. Regrettably, the remaining time available for the completion of this project does not allow for further timing-based optimization, leading to the concession that the design will not operate at the expected 100MHz, despite using the fastest grade part available.

The second assembled permutation, called Top_2, consists of the following sequence of filters filling the DSP sockets: FIR_1, FIR_2, NCO_3, polyFIR_4, and polyFIR_5. Filter_tuning generated in the Fixed_Logic module will instruct the both sets of matching FIR and polyFIR filters to operate in conjunction. Figure 26 shows that the result is a DSP processor that processes data through a 64-tap even symmetry FIR filter, running the result through the Quadrature Mixer, and then sending both the in-phase and quadrature data independently through 32-tap even symmetry Time-varying Coefficient FIR filters to generate two 16-bit output values. When mapped, the resulting design used 1,380 of 3,584 CLB blocks, or 38.5% utilization, while also using 66 of 96 embedded multipliers and 62 of 96 BlockRAMs, resulting in an equivalent gate count of 4.47 million system gates. Unfortunately, the design placed and routed to reach a maximum frequency of only 68.7MHz due principally to extraordinarily long delays in addressing and writing coefficients into the FIR filters. The tri-state buffers provided the same delay as the Top_1 configuration which, when added to aux_addr and aux_data delays in reaching the 16-bit coefficient registers, resulted in a total delay of 14.553ns.

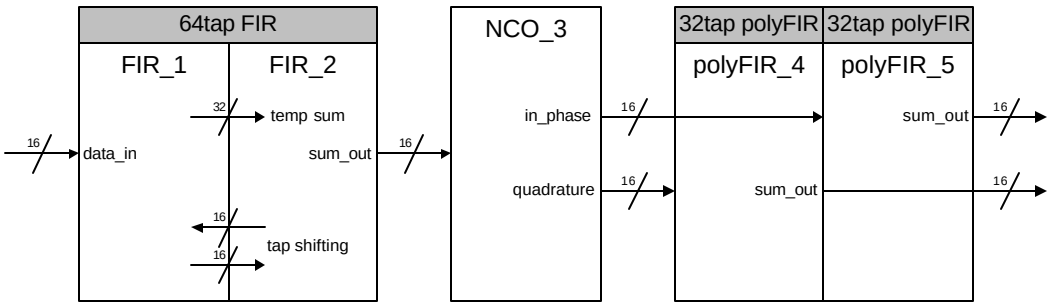

Figure 26 - Top_2 Configuration

As an experiment, an additional timing constraint was added to the FIR filters and the filters were individually rebuilt, followed by a rebuilding of the Top_2 configuration. The timing requirement that the FIR filters must have a minimum delay between inputs and registers of 8ns resulting in an 818ps improvement in the Top_2 configuration's critical delay path. Now, the Top_2 configuration is able to run at 72.8MHz. While still not close to the desired 100MHz target speed,

the experiment does indicate that modest improvements in speed can be realized with increased timing constraints. Further improvement could possibly be realized through the use of additional pipeline stages, specifically for the aux_addr and aux_data signals since they often must fan-out to all 16 sets of embedded multipliers or BlockRAMs.

The third and final design permutation, called Top_3, consists of the following filters: NCO_1, polyFIR_2, polyFIR_3, polyFIR_4, and polyFIR_5. Illustrated in Figure 27, the desired functionality of this configuration is a Quadrature Mixer followed by two independent 32-tap even symmetry Time-varying Coefficient FIRs, with each of the two data paths going independently into separate 32-tap even symmetry FIR filters. As designed, however, two neighboring 32-tap FIR filters would combine into a single 64-tap even symmetry FIR due to the filter_tuning options implemented. Rather, Time-varying Coefficient FIR filters can be used in place of the FIR filters if the accumulator delta value is set to 0x0 and only the first location in each coeffRAM is used. When built, Top_3 uses 1,376 of 3,584 of the available CLBs, or 38.4% utilization, along with 66 of 96 embedded multipliers and 94 of 96 BlockRAMs, resulting in an equivalent gate count of 6.57 million system gates. This powerful display of the usefulness of embedded components in reconfigurable logic, when placed and routed, runs at 78.1MHz without the luxury of additional module-level timing constraints. If timing constraints similar to those employed in the FIR filters were utilized, the expected 800ps reduction in delay would result in a speed of over 83.3MHz.

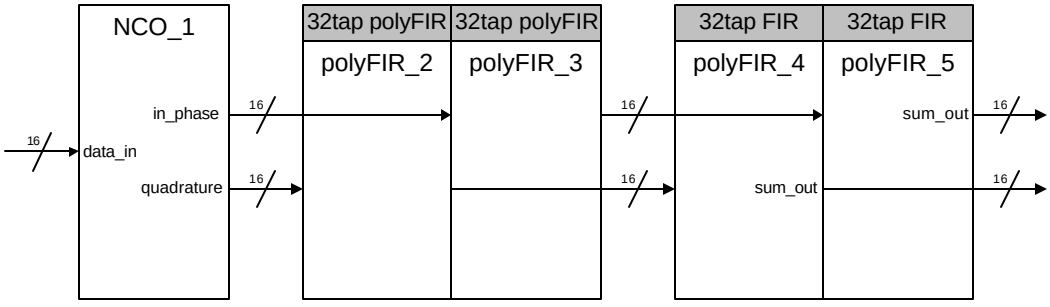

Figure 27 - Top_3 Configuration

The operational speed of this FPGA, however, cannot be dependent on the configuration currently loaded. As a result, the device can only be expected to run as fast as the slowest possible assembled permutation, which is expected to be five FIR filters. If this architecture were to be deployed, rigorous timing constraints would be required at both module-level and device-level to ensure proper operation at 100MHz. For example, since the tri-state buffer delay between modules is approximately 3ns, both the flip-flop to output and input to flip-flop delays would need to be limited in a manner that meets this 10ns requirement.

More importantly, the essential reason for this undesired delay cannot be overlooked: the modular nature of the partially reconfigurable architecture inherently adds a significant amount of delay to the design. The signals within each DSP module must be routed within the eight CLB-wide module boundaries, which can be very inefficient compared to a design where module location and separation is not restricted. Furthermore, bus macro interconnects between modules are manually placed with utter disregard for the optimal location for each signal, fostering an even more inefficient assembled design. Figure 28 illustrates a likely scenario, as interpolated from timing analysis data from the Top_3 configuration. The aux_bus signals are factors in the worst critical path delays as they may be routed to any portion of the DSP module while being required to pass through the location-confined bus macro interconnects. In the example given, coefficient information on the aux_data bus may be routed to the first BlockRAM of the rightmost socket, which would result in a lengthy routing delay that can approach 10ns. Again, additional pipeline stages would be appropriate to alleviate this delay.

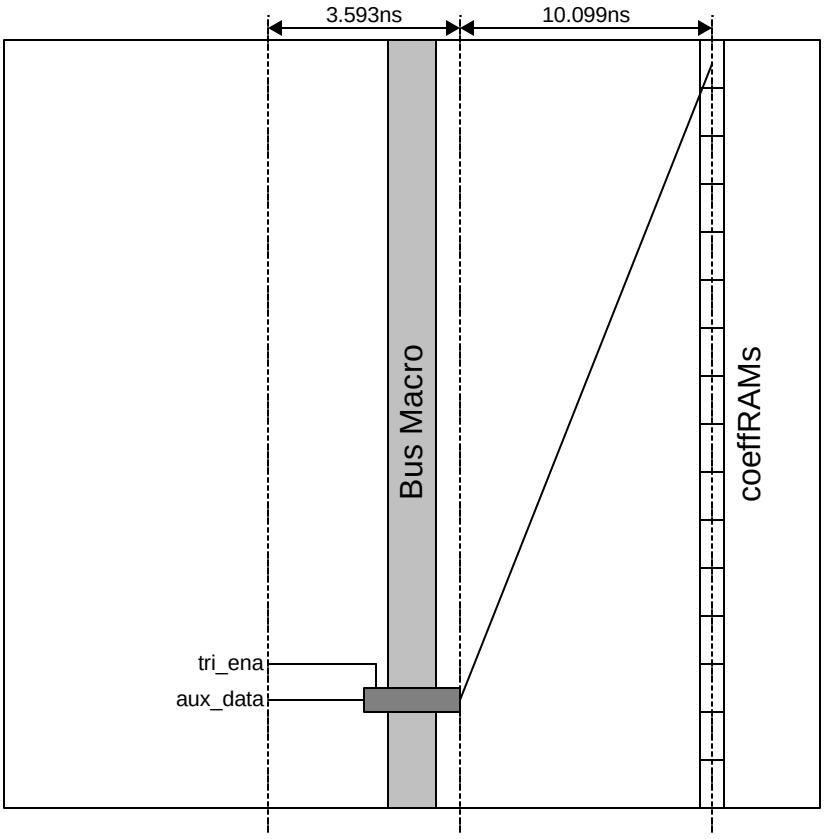

Figure 28 - Connected Module Delays

The costs associated with using a partially reconfigurable design rather than a conventional design are quantifiable with the following results. The design described was rebuilt using a conventional FPGA design methodology to fit onto the same Xilinx Virtex-II 3000 FPGA without pre-defined module locations or bus macro boundaries. The CLB logic overhead associated with the use of the partially reconfigurable architecture given is surprisingly small, requiring an average of only 3.4% more CLBs than a conventional FPGA design. The speed advantages of a non-partially reconfigurable design, however, were more pronounced. The Top_2 configuration that required a period of 13.735ns for a partially reconfigurable design is capable of running at 10.008ns using a conventional flow. The tri-state buffer and extensive routing delays are eliminated. Likewise, the Top_3 configuration that ran on a 12.812ns period now runs at 9.790ns. So while roughly the same size device can be used for both design flows, a conventional design can make use of a slower grade part to realize

the same performance as a partially reconfigurable design targeted towards a high-speed part.

Given the current design flow, however, the bus macros are absolutely necessary to ensure proper signal connection within the partially reconfigurable framework. In addition to making sure that tri-state signals are only driven when the source module is active, the tri-state buffers contain specific placement and routing directives that both connecting DSP sockets communicate through a common routing node. A similarly constrained connection node between modules cannot currently be replicated manually using the standard design flow process and would require manual manipulation of the place and route tool using proprietary Xilinx methods.

As an experiment to find possible improvements in timing performance, the locations of bus macro columns one through five were relocated as indicated in Figure 29. When the Top_3 configuration was rebuilt under these new constraints, performance actually slowed down from the previous 78.1MHz to 76.3MHz. Clearly, the placement of such a large number of signals is a delicate procedure that can have consequences on the overall performance of the design.

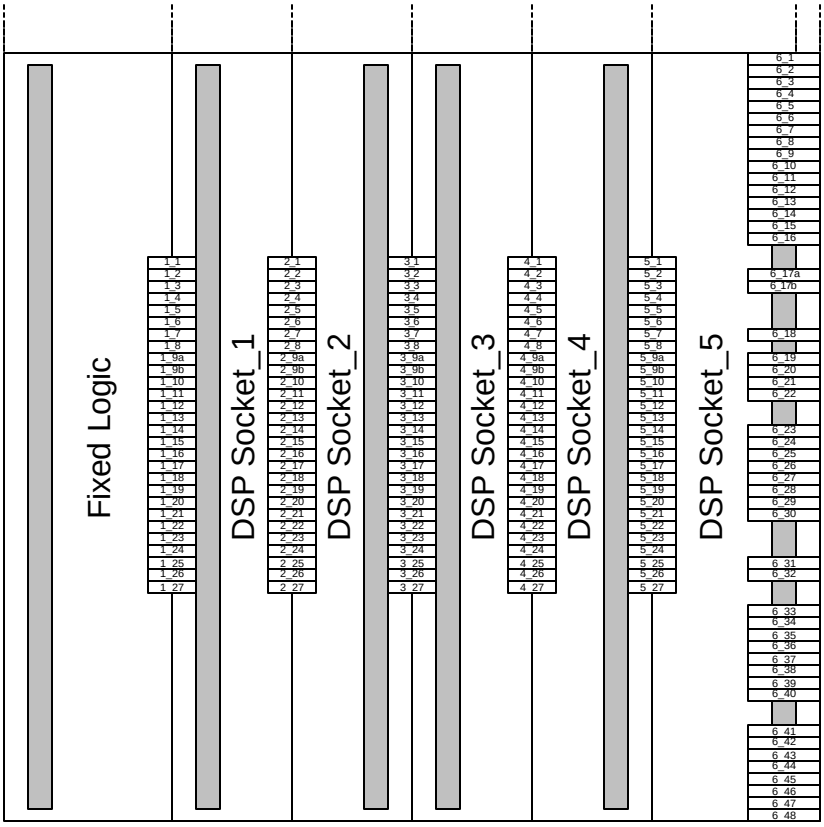

Figure 29 - Bus Macro Location Modification

Xilinx's partial reconfiguration design flow using modular design would be greatly improved if the toolset had the capability to automatically optimize the location of module interconnects based on a library of all possible assembled permutations. Given the set of the four DSP modules implemented for this study, for example, it may be best for the aux_addr signal between the Fixed_Logic module and Socket_1 to be in a row X while the same signal between Socket_1 an Socket_2 may be better suited to row Y. Whichever optimal placement is chosen, the results would need to be applied consistently to all DSP sockets to ensure identical placement and guarantee functionality. Along with the prerequisite temporal floorplanning aspect, this feature would greatly enhance the ability of a designer to create a stable modular architecture without the added concern of delays associated with unexpected assembled design permutations. Beyond the inflexibility of the interconnect mentioned in the first chapter, the logical and routing resource limitations within each module can strangle the

ability of the toolset to refine the design. These limitations, however, must be tolerated as the ability to dynamically resize DSP sockets greatly increases the complexity of the design process and architecture to unmanageable levels.

7. Simulation

During the design implementation process described in the previous chapter, each module must be simulated to verify proper functionality. The verilog modules can be directly simulated in Verilog-XL using a simple testbench structure that effectively acts as a virtual fixture in which the module can tested by stimulating and reading the appropriate inputs and outputs, respectively. While this method will allow for verification of basic functionality, it fails to take into account any device-specific timing characteristics. Therefore, the back-annotated timing-based verilog modules generated from a placed and routed design allows for the most complete verification available at the simulation level. In order to expedite the debugging process, testbenches are initially run repeatedly against the raw verilog modules in order to correct as many detectable functional faults as possible before the Xilinx ISE building process is started. Since a single place and route task requires anywhere from two to nine hours of CPU compilation time, avoiding unnecessary rebuilds was definitely desired. As any changes were made to the modules later in the design process at either the module or device level, both the regular and back-annotated testbenches were rechecked. Each module type and location was independently simulated followed by the larger-scale simulation of entire assembled designs.

Before the process is described in detail, however, it should be emphasized that self-verifying a design can be both difficult and incomplete due to the plethora of assumptions, preconceptions, and expectations of the designer. After designing a module, for instance, the designer will create a verification environment that transmits a series of expected signals and detects expected outputs. To verify that illegal operations do not occur to adversely effect the operation of the system, the designer may also create a series of such situations. Without an independent audit of the design, however, it is extremely difficult to postulate unexpected operational scenarios. The number of previously overlooked module bugs detected during the device-level simulation phase reinforces this argument.

7.1. Module-level Simulation

Beginning with the Fixed_Logic module, a testbench was constructed to interface with all input and output signals of the module, including all signals passing to and from the first bus macro boundary. The testbench consists of a series of test sequences targeted towards various aspects of the design. The first, for example, tests a simple data flow scenario passing through the Data_Input_Control, data_FIFO, and Data_Output_Control sub-modules. After a stream of data is loaded into the data_FIFO, the output ready signals are activated to flush the FIFO. In a second test, the reading and writing operations are performed simultaneously to ensure that the data_FIFO is capable of such functionality. Next, the operation of the next_config and next_config_flag registers was simulated in a variety of sequences, followed by much more elaborate partial reconfiguration loading tests.

This testbench can only directly verify that the Fixed_Logic module generates the appropriate aux_bus and bitstream status signals, so it becomes necessary to manually inspect the finite state machines using the SignalScan tool during the testbench design process. For example, during an early partial reconfiguration test, a deadlock situation occurred in which the module and testbench both ceased to operate, a scenario difficult to debug without any insight into the internal state of the module. Using SignalScan, the failing Parameter_Control sub-module state was quickly detected and corrected. During reconfiguration or reparameterization socket shutdown sequences, the aux_bus and bitstream output signals needed to be carefully monitored and coordinated with the appropriate module status or parameter input signals. After all conceivable working scenarios were exhausted, a series of illegal operation tests were introduced to ensure that the complex Fixed_Logic module would not fail once incorporated into the larger assembled designs.

After a quick confirmation that the End_Logic module functioned, the DSP modules were systematically attacked with the opening salvo assailing the empty

module. Since all other DSP modules were based on the status signal and state machine framework of the Empty module, thorough interrogation of this design resulted in a significant reduction in overall verification time. Like other DSP modules, the Empty module testbench for Socket_1 was designed for easily replication to other socket locations. Moreover, the setup time of signals asserted in the testbench was designed to mimic the clock-to-output delay time of signals exiting the Fixed_Logic module. Most tests for the Empty module revolved around the proper forwarding, creation, and use of aux_bus and status_bus signals, especially in regard to module shutdown to activation directives. If a module in Socket_2 were to be deactivated, for example, it would further monitor the aux_bus to determine when previous modules have finished processing data so that it may finish its tasks and transmit a confirmation signal on both aux_bus and status_bus. Lastly, the proper operation of the tri-state enabling signal is paramount to preventing signal contention during partial reconfiguration.

Next, the FIR filter testbench built in the tests encompassed in the Empty module testbench. Rather than the verification of simple data flow, however, this testbench needed to mimic the operation of the full 32-tap even symmetry FIR filter to ensure that all tap shifting, addition, and multiplication operations proceeded correctly. Given that all 16-bit and 32-bit internal values are signed, the process of generating the appropriate output sums was daunting. The pipeline latency of the post-multiplication adder tree needed to be accounted for in the assertion of the data_ready_next status output. In addition to a data input stream scenario, an intermittent data input situation was also simulated. In all cases, coefficients were first loaded via the aux_bus using a reparameterization process. While the verification of the full 32-tap design may itself seem complex, the testbench also simulated combined 64-tap operation by verifying filter operation in both the first and second positions. In accordance, the appropriate intermediate signals on the secondary data bus needed to be generated and observed where applicable. While this verification phase concluded that the FIR filter operated correctly, the upcoming device-level simulation section will refute this claim.

In contrast to the FIR simulations, complete coverage of the Quadrature Mixer was more difficult to achieve due to the nature of the numerically controlled oscillator. In one test, input data remained at a constant value while the NCO cycled through the range of sine and cosine output values, which were then captured from the simulation log file and confirmed using an spreadsheet. Due to the range of possible data input, phase increment, sine, and cosine values, however, it was not possible to verify every possible corner case within the time available for this thesis. The extent of the verification performed will have to suffice. Finally, since this filter can only operate in a single module configuration, it was not necessary to simulate any multi-filter situations.

Lastly, the Time-varying Coefficient FIR module testbench requires a series of tests very similar to that of the original FIR filter with the addition of more complex coefficient loading. In addition to the larger addressable size of the coefficient storage memories, the coeffRAMs add an additional cycle of latency that must be accounted for in the tap shifting mechanism. Once the coefficients are loaded along with an addressing delta value, the operation of a single 32-tap Time-varying Coefficient FIR filter is similar to previous FIR tests. Simulating the operation of two Time-varying Coefficient FIRs working in conjunction, however, is another matter. In either the first or second filter position, it is vital both filters require the same latency to process data so that both sets of 16-bit sums are output in the same clock cycle. Precise reproduction of the appropriate intermediate signals in both steady and intermittent data flow situations resulted in satisfactory confirmation that the modules work.

7.2. Device-level Simulation

Once the individual modules were acceptably tested, the mission changed to verifying the coordinated operation of permutations of the DSP modules in an assembled design. While verification coverage for the aux_bus, status_bus, and data bus signals was thought to have been adequate during the module-level simulation stages, this was quickly determined to not be the case when the Top_1

configuration was initially tested. After reversing the polarity of the tri-state enabling signals to the correct orientation, it was determined that the bus macros were being deactivated earlier than necessary to facilitate proper aux_bus and status_bus signal forwarding. These issues, along with some other minor issues with data bus status signals, should have been detected at the module level, which serves to illustrate the difficulties associated with achieving adequate certification in testing one's own design.

The testing process included not only checking the flow of data through the five-socket sequence, but also mimicking as much of the reconfiguration process as possible given the simulation environment. As previously highlighted, currently available simulation environments due not support the ability to physically modify the design logic as is required during partial reconfiguration. Therefore, it is impossible to ascertain whether the bus macro-based module boundaries actually prevent signal contention during reconfiguration. The partial reconfiguration sequence of deactivating one or more modules, requesting the loading of appropriate bitstreams, and reactivating the affected modules can be simulated, but only without any actual change to the contents of the DSP sockets. Since the next_config and current_config registers contain default values of 0x0, indicating the presence of Empty modules, the Top_1 was testbench was written to call for the reconfiguration of the sockets with various other module types. As observed through SignalScan, the internal state of each of the five Empty modules responded correctly to aux_bus signals emitted by the Fixed_Logic module. Similarly, although the Empty modules do not actually contain any loadable parameters, the reparameterization process was emulated to confirm proper operation of that aspect of the design as well. Errors detected during the verification of Top_1 were reflected in all other DSP modules, where appropriate. Since the speed at which the back-annotated code could run varied as optimizations were made in the place and routing process, simulations were run at 50MHz to avoid complications.

While Top_1 verification problems centered predominantly on general module interaction and interconnection issues, the verification of the Top_2 configuration involved a number of problems related to two-filter combinations. Some of the problems probably should have been detected during module level simulation, but again such complex multi-module interactions are difficult to envision in a single module environment. The first test in the Top_2 testbench entails loading the next_config registers values reflecting that actual modules existing in the DSP sockets. Once the reconfiguration process was emulated and the appropriate coefficients loaded, it became apparent that the FIR filters loaded in the first two DSP sockets failed to operate properly. For instance, the second filter had been designed to wait until it detected a data_ready signal from the first filter before it would start accepting data from the secondary data bus and shift the data taps. Unfortunately, the first filter would not assert its data_ready_next signal until it was ready to actually output an intermediate sum value on the data_out bus. This problem was corrected by changing the nature of the data_ready signal to reflect the availability of data on the secondary data bus. Since the second filter still needed to know when all 64-taps were filled and intermediate sum data was available, a 64-bit shift register was enacted to keep track of how many taps were actually filled. The first filter in the sequence keeps track of all 64-bits while the second filter tracks 48-bits since it is oblivious to the first 16 taps of the first filter.

A second problem with the interaction of the two FIR filters involved the actual 64-tap shifting process, as depicted in Figure 30. As designed, the first filter would shift the value of tap_16 onto the next_out bus, making it available to the second filter. On the next clock cycle, the second filter would read this value and shift it into its tap_1. The process was repeated in shifting tap_32 of filter two into tap_17 of filter one. Because of two one-cycle delays across the bus macro boundary, the combined 64-taps would not contain 64 consecutive values but rather 64 of 66 consecutive values, with the 17^{th} and 50^{th} values in-transit on the secondary data bus. To correct this error, the value of the first filter's next_out bus would be derived from tap_15 with a similar method employed using tap_31

of the second filter. Figure 31 shows how the modification fixes the problem, enabling proper operation of the circuit.

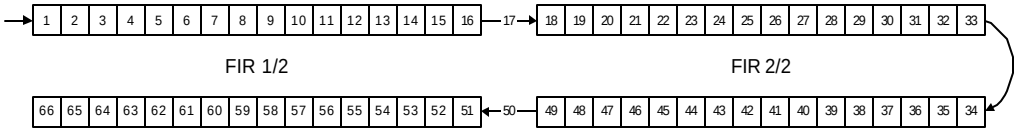

Figure 30 - Improper 64-tap Shifting

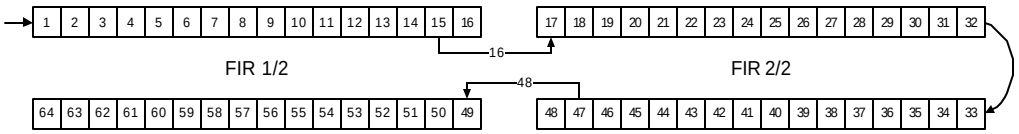

Figure 31 - Proper 64-tap Shifting

After confirming that the Quadrature Mixer in Socket_3 operated properly, another interconnect problem was detected between the Time-varying Coefficient FIR filters occupying the fourth and fifth sockets. The implementation section on the Time-varying Coefficient FIR filter shows how, in two-filter operation, the lower 16-bits of data are to be processed in the first filter while the upper 16-bits are bypassed to the second filter. The results are then combined in the second filter and transmitted on to the next stage. As designed, the first filter would accurately shift the lower 16-bits into the Time-varying Coefficient FIR sequence, but would not forward any values from the upper 16-bits of data_in until it had computed the 32-tap Time-varying Coefficient FIR sum and attempted to transmit that value on the data_out bus. In the process, a number of data values intended for the second filter were wrongly discarded. Like the original FIR problem, the upper 16-bit value on the data_in bus was immediately forwarded on the secondary data bus along with the corresponding data_ready_next signal. The second Time-varying Coefficient FIR filter would then utilize this value to compute the resulting FIR value, which would be combined with the first filter's output value to create a pair of 16-bit sum values. A 32-bit version of the shift register incorporated into the FIR filter was implemented in the Time-varying Coefficient FIR to better track the presence of data in the 32-taps. Once these

interconnection issues were corrected, the Top_2 configuration operated as expected.

The verification of the Top_3 configuration took advantage of corrections made during Top_2 verification, resulting in a much more efficient authentication process. Since the two-module interconnection issues were no longer present, no problems were detected in this verification phase, thus completing the verification process as required for this thesis project.

8. Benchmarking

Now that the three permutations of the design architecture have been created and verified, this chapter will contain an analysis of whether the architecture succeeds in surpassing the performance of DSP running on a G4 CPU, as currently implemented for a Teradyne application. Since the critical path of the architecture's operation involves signals passing between two interconnected FIR filters, as detected in the Top_2 configuration, the maximum operational speed of the FPGA design will be set at 72.8MHz. The G4 PowerPC 7410 processor, on the other hand, will be operating at 400MHz and will be connected to a 128MB SDRAM via an intermediate FPGA using a 64-bit 100MHz memory bus. Additionally, the G4 will also be directly connected to a 2MB SDRAM cache via a 64-bit 160MHz dedicated bus, as indicated by Figure 32.

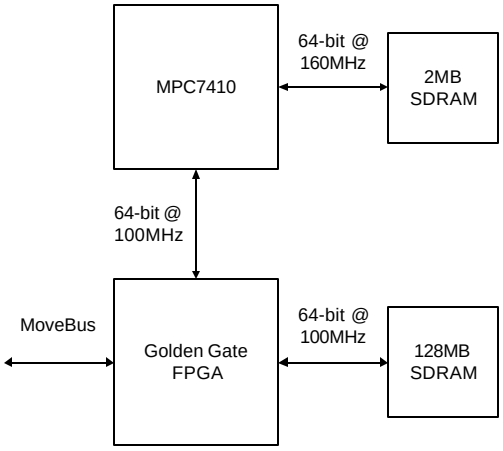

Figure 32 - G4 in DSP Module

Initially, consider a scenario where a single 32-tap even-symmetry FIR filter exists on the FPGA connected to four Empty modules. In the case of a long uninterrupted stream of data, a single FIR module will be able to process 16 16-bit additions, 16 multiplications, and 15 32-bit additions in a single cycle. For comparison purposes, simplify this situation to state that the FIR can perform 16 multiply and accumulate, or MAC, operations per cycle. Given the clock speed of 72.8MHz, this simplified FIR filter can perform 1.165 billion MACs, or 1.165 GigaMACs, per second. If all

five sockets were loaded with FIR or Time-varying Coefficient FIR filters, the architecture could support a maximum throughput of 5.825 GigaMACs/second. Since the Top_2 and Top_3 configuration both contain four FIR or Time-varying Coefficient FIR filters, the simulated FPGA produces a throughput of 4.660 GigaMACs/second.

As a benchmark, internal Teradyne testing produced the following performance numbers using the PowerPC 7410 G4 processor running at 400MHz. To produce bulk data processing numbers, 4096 MAC operations were run on the G4 to simulate a 4096-tap constant coefficient FIR filter. Due to the serial processing nature of this test, which in this example does not utilize the vector processing ALU present on the G4, this operation took an average of 102.9µs, or 25.11ns per MAC. This results in an effective throughput of 39.82 MegaMACs. In a separate 4096-tap time-varying coefficient FIR filter test, the G4 required an average of 53.52µs to perform the calculation, or 13.07ns per MAC, resulting in an effective throughput of 76.53 MegaMACs. It is suspected that the manner in which the time varying coefficients were generated and stored in the cache resulted in an increase in coefficient fetch and subsequent overall processing speeds over the constant coefficient version. As evident in Table 4, the FPGA architecture provides a significant processing speed advantage over this G4 in the case of a long, uninterrupted data stream.

Table 4 - FPGA to G4 Performance Comparison

		MegaMACs/second	Performance Advantage
FPGA	Single FIR	1,165	15.22x
	Top_2	4,660	60.89x
	Maximum	5,825	76.11x
G4		76.53	

Since the FPGA is designed as a run-time partially reconfigurable architecture, however, DSP module-reloading times must be considered in order to fully compare the true performance of the FPGA. On the G4 processor, the time required to call a subroutine from the instruction cache is negligible compared to data and coefficient bus access and processing time requirements. For the FPGA, on the other hand, module reconfiguration times are significant compared to processing speeds. Using

the Virtex-II's 8-bit SelectMAP configuration loading interface running at 66MHz, the following loading times can be achieved, as given in Table 5. The partial reconfiguration of all five sockets, for example, would require 13.62ms.

Table 5 - Partial Reconfiguration Times (ms)

Sockets 1-4	2.59
Socket_5	3.24
All Sockets	13.62

It quickly becomes apparent that the 16Kx17-bit data_FIFO is not sufficiently large enough to hold all of the required data. Running at 72.8MHz, the data_FIFO would need to be increased to 992Kx17-bits in order to accommodate all the data. Effectively, in order for the existing data_FIFO to be sufficiently large enough to hold all incoming data during a five-socket reconfiguration, the system could only run at 1.2MHz. This FPGA architecture would be better implemented with an external SRAM data_FIFO connected to a memory controller embedded in the Fixed_Logic module. To simplify this viability study, however, utilizing an embedded data_FIFO reduced the complexity of the design and simplified the verification of the design as well.

Aside from the data_FIFO issue, the size of the data stream to be processed with a given configuration can determine whether or not using the partially reconfiguration FPGA architecture is a better option than a G4. For the following scenario, assume the G4 memory architecture is designed such that larger data sets do not adversely affect processing speed. Also, assume again that that Time-varying Coefficient FIR filter performs 16 MACs per second. Consider a stream of 10 million data points running through a single Time-varying Coefficient FIR loaded into Socket_1 on the FPGA. Running at 72.8MHz, the device would only require 137.4ms to process all 10 million data points. Even if the 2.59ms partial reconfiguration time is included, the device still only needs 140ms to process the data. At the previously given speed of 13.07ns per MAC, a 16-tap time varying coefficient FIR filter implemented on a G4 would 2.09 seconds to accomplish the same task. If the data stream length is only 1,000 data points, on the other hand, the picture drastically changes. The FPGA

processing time of only 13.74µs is overshadowed by the constant 2.59ms partial reconfiguration time for that module. Meanwhile, the G4 processor provides superior performance by completing the task in 209µs. As displayed in Figure 33, the crossover point between FPGA and G4 processing advantage occurs at 13,256 data samples in this case. In the case that Time-varying Coefficient FIR is already loaded into Socket_1 and reparameterization does not occur, the FPGA will surpass the G4 in all instances. Finally, in the case of a full reparameterization of all 16K coefficients present in the filter, which requires approximately 225.1µs, the FPGA is superior as long as the data stream is longer than 1,152 data points, as seen in Figure 34.

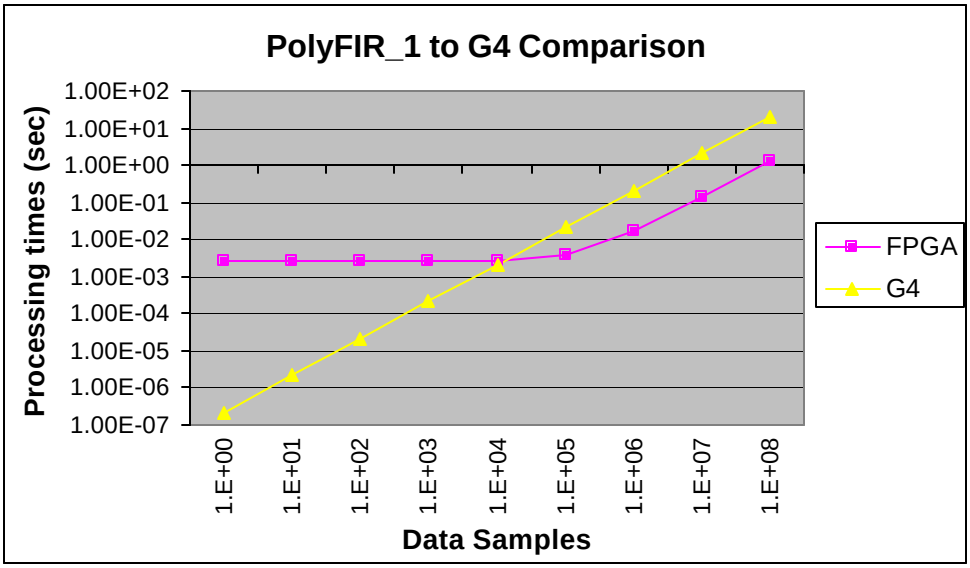

Figure 33 - PolyFIR_1 to G4 Comparison

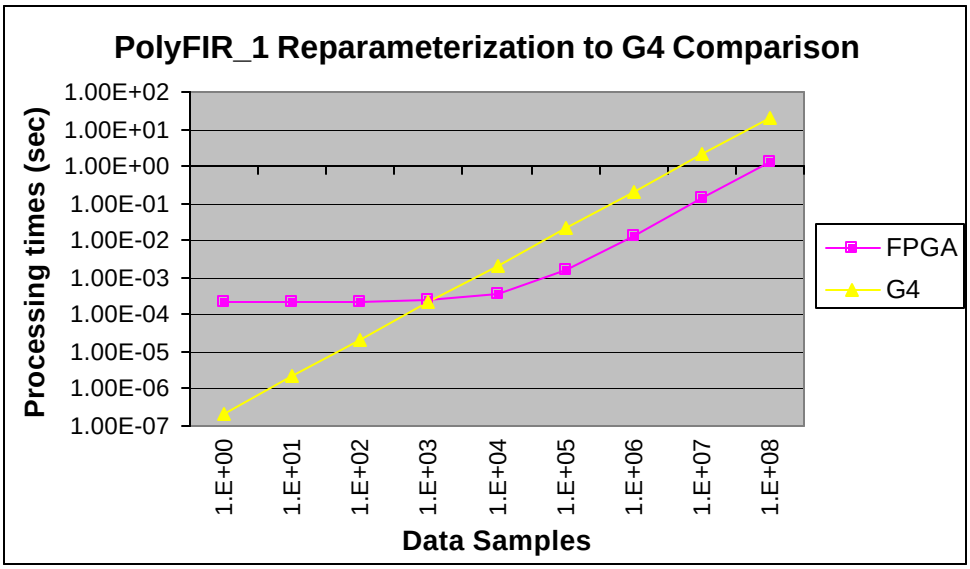

Figure 34 - PolyFIR_1 Reparameterization to G4 Comparison

Now, consider the Top_3 configuration, as described in previous chapters, along with a full five-socket partial reconfiguration. While the processing speed remains the same, the partial reconfiguration time increases to 13.62ms and becomes an even larger component of overall processing times. On the G4, even without considering the Quadrature Mixer, a set of four 16-tap time-varying coefficient FIRs would require 836µs per sample to process. As a result, as seen in Figure 35, the performance crossover occurs at 16,554 samples. In the separate case that the five sockets are already loaded and only the parameters are being adjusted, Figure 36 shows that the crossover is reduced to 548 samples.

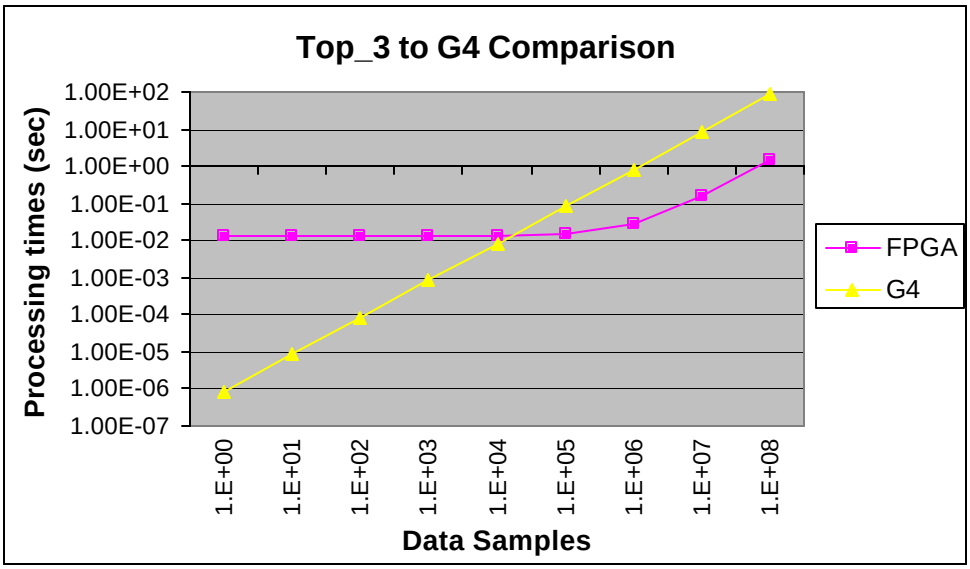

Figure 35 - Top_3 to G4 Comparison

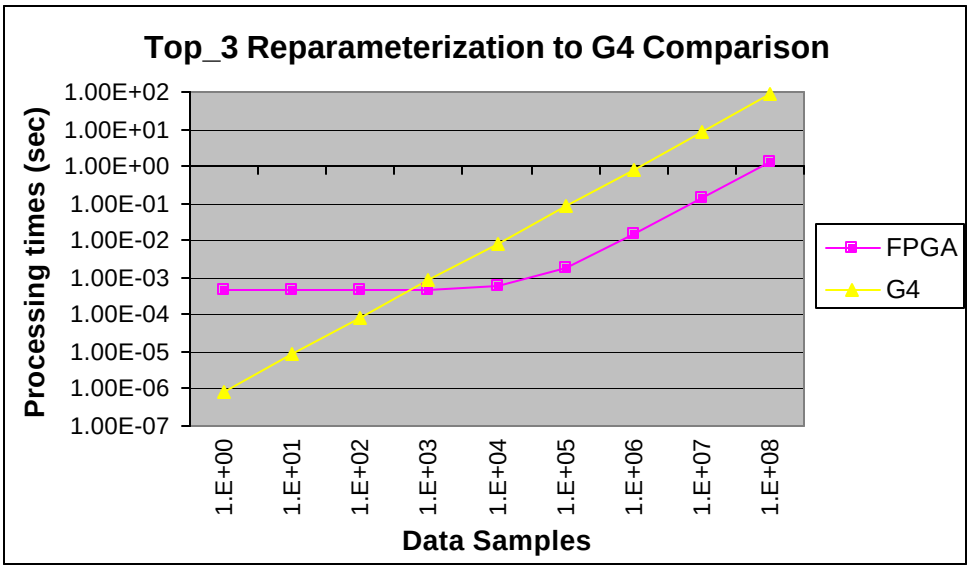

Figure 36 - Top_3 Reparameterization to G4 Comparison

Clearly, the processing scenario can contribute greatly in determining which processing solution is most applicable. In the case that a single configuration is repeatedly used in the testing of a batch of devices, the run-time partially reconfigurable FPGA solution is always superior to a G4 given that processing chain can be fit within the FPGA architecture. In the case that a single device under test

must be run through multiple processing configurations, the FPGA maintains its advantage as long as the data set is sufficient. Since the data set length can vary from under one hundred to over a one million samples based on the users requirements, the system should designed to be able to select the optimal processing method of either the FPGA or G4. Effectively, the test program would need to select between either hardware or software subroutines, respectively.

9. Conclusion

All things considered, the goal of implementing a high-speed DSP architecture within a run-time partially reconfigurable FPGA was achieved. While the device as designed will only operate at 72.8MHz, the five-socket modular architecture will support a wide range of DSP algorithms that can make use of the advanced embedded features of the Xilinx Virtex-II FPGA. If a desired processing algorithm is too large for a single DSP socket, the architecture compensates by allowing multiple neighboring sockets to work together in a coordinated manner. The resulting large-scale parallel computational ability reigns superior to a serial processing solution given the available processing speeds and the fact that the G4's vector processing ALU was not utilized. Additionally, secondary considerations such as ease of instrument design, ease of system design, field upgradability, and thermal properties also lean in favor of utilizing an FPGA running at 100MHz rather than a series of ASICs or a G4 running at 400MHz.

The result, from the user's perspective, is in an architecture that can support a library of timing-verified and guaranteed hardware subroutines that could be utilized in a fast and flexible FPGA architecture just as a software subroutine could be processed on a CPU. A programming interface could just as easily run through a sequence of run-time partial reconfigurations as desired by the application while enjoying the parallel processing speed advantages offered by the FPGA. While the same processing algorithms could be realized by using a vast array of interconnected programmable DSP devices or ASICs, a single FPGA could duplicate this functionality with a fraction of the power consumption and footprint.

Additionally, the exact specifications of the design implemented were based on the parameters desired for this feasibility study. The Xilinx Virtex-II 3000 FPGA was chosen because it features six columns each with 16 embedded multipliers and BlockRAMs, which easily supported a partially reconfigurable five-socket architecture. Each DSP Module would receive a set of 16 embedded blocks, a

convenient number to work with given the desired FIR-based filters and the storage requirements of the desired NCO. Within the architecture, the data and secondary bus sizes were arbitrarily chosen for the given DSP modules, but could be increased if desired. Since the embedded multipliers and BlockRAMs are 18-bits wide, all computational, storage, and bus elements could have been increased to this size. Furthermore, the number of supportable DSP modules could easily be increased from its current value of four, which again was arbitrarily chosen for this feasibility study.

There were, as expected, a number of concessions made in the design process as dictated by constraints present in either the available hardware, supporting software, or the scope of this project. While a workaround was found and utilized, the column-based reconfiguration scheme present in the Xilinx Virtex-II devices presents a significant design hurdle. A FPGA without this constraint could be used to design a much more flexible and less wasteful architecture. For example, since the FIR filter implemented required less than 40% of the CLB resources allocated, reducing the size of the DSP module by half would result in a corresponding reduction in partial reconfiguration time.

Furthermore, the removal of this column-based constraint would also greatly simplify the control scheme designed into the Fixed_Logic module. As described, shutdown notifications and directives must propagate through intermediate modules in order to reach the target module. This process is complicated by the fact that module undergoing partial reconfiguration temporarily disrupts this communication method. Preferably, the Fixed_Logic module would directly connect to each DSP Module either through independent channels or through a central uninterrupted bus.

Along these lines, the utilization of a flexible data bus could also increase the flexibility and efficiency of both the architecture itself and specifically the partial reconfiguration process. Referring back to Figure 12, consider an architecture devoid of column-based restrictions in which a number of specific DSP modules are loaded. Consider the scenario in which a partial reconfiguration is desired to remove a single

DSP module from the processing chain. Rather than reloading the affected socket with an Empty module, which would require a couple milliseconds depending on the size of the module, the same processing chain modification could be utilized by a simple parameter change that altered data routing through a central bus. The ability to specifically target a routing change rather than a larger-scale logical change would significantly reduce effective partial reconfiguration time and make the FPGA architecture an even more desired processing solution.

The software constraints are similarly difficult to work around, leaving much room for future improvements in the design process. As stated, a temporal floorplanning-based design environment would again greatly simplify the design process by removing a great deal of complexity that for now must be dealt with manually. DSP modules were created in separate design environments and continually checked to ensure that they operated within the scope of the architecture. Any major modification to the architecture would require the designer to scour through a number of design environments and implementation stages to propagate the desired changes.

Verification alone presents an obstacle that simply cannot be completely overcome given the current design process. Modules can independently be verified to a satisfactory degree, but the combined design can only be confirmed at the rudimentary level. Like the design process itself, various permutation of the design must be verified separately without the much-needed ability to verify the run-time partially reconfigurable aspect of the design. If this degree over coverage were available, it would be possible to construct a simulation suite that included a number of partial reconfigurations, which could be more directly compared to the G4 processor's ability to sequentially process a chain of subroutines.

Finally, the time constraints and scope of this thesis limited the extent to which this architecture could be explored. The design, implementation, and simulation of the architecture were deemed sufficient for this thesis. It would be desirable, however, to delve into a physical prototype scenario in which the run-time partially reconfigurable

design was tested in a real-world situation. The following chapter outlines potential future work using this architecture and technology.

10. Future Work

Given the opportunity to continue work on this project, a number of challenges would be undertaken to refine the current architecture, physically test the architecture, and find adapt this architecture to suit other potential applications. Clearly, there are a few issues for which additional work would yield a superior final product than the one presented in this thesis. First, further optimization of the module timing requirements should lead to reductions in bus macro interconnect delay and result in improvements in overall system speed. Also, an external version of the Fixed_Logic module's data_FIFO would result in a more appropriately sized data storage capability. To accomplish this, for example, the Data_Input_Control and Data_Output_Control sub-modules could remain in the Fixed_Logic module while connected through external pins to a separate SRAM device. In the case that the SRAM device is designed to act as self-addressing FIFO, the data and control connections would be quite simple. On the other hand, if a regular SRAM memory were used, the Fixed_Logic module would need to contain the addressing mechanism and either a single memory bus or dual-port memory access scheme would be required.

Prototype testing of the device architecture using either a Xilinx prototyping board or an in-system configuration would result in a more comprehensive glimpse at the final product. Although this degree of testing would have also resulted in a significantly longer development timeline, data collected from an actual run-time partial reconfiguration could reveal certain power drain, stability, or other physical-level issues that are undetectable in simulation.

On a separate topic, the issue of the variable latency of the FPGA architecture was not considered until the final phases of this project.[29] Despite the fact that every module runs on the same clock, depending on the configuration loaded into the device, the raw data input to processed data output latency can range from ten cycles for the Top_1 configuration to 28 cycles for the Top_3 configuration. As designed, the

device reading data from the End_Logic module would simply wait until the data_ready_out status signal indicated available data. Given the complex pipeline timing required for Teradyne testing system or any other comparably complex system, additional process latency information may be required from the FPGA. This design enhancement would not be complicated and given that the Fixed_Logic module already tracks the current configuration and filter_tuning of the device, it would only be a matter of adding an additional output bus to communicate this process latency.

In addition to the designated application, this run-time partially reconfigurable architecture could be adapted to serve other signal processing purposes for other applications. While this version was designed to act as an alternative to the use of software processing in a CPU, this FPGA could also be used to serve as a dedicated co-processor to the CPU. This modification could simplify the design process by allowing the control scheme and external interface of the FPGA, as dictated by the software on the CPU, to be modified along with the FPGA's contents.

Furthermore, rather than reserving this technology for larger systems, the same architecture could be adapted for use in portable digital devices. A similar architecture would be well suited for use in a software radio application, for instance. As encoding schemes and processing requirements change not only with advancements in technology but also the region of use, the ability to reprogram the DSP capability of a phone over the air could be extremely useful.[30] It is unreasonably burdensome to prepare for a number of configuration or processing possibilities by placing more DSP devices into small portable devices, especially when newer, faster, and more capable FPGAs excel at the same sequential signal processing tasks.[31] Advancements in design methodologies to make partially reconfigurable modules on an FPGA more analogous to hardware subroutines, such as object oriented reconfigurable processing work currently underway at Jet Propulsion Laboratories, could make the design of similar time-multiplexed processing architectures more efficient.[32]

Overall, the utilization of the run-time partially reconfigurable aspect of current FPGAs opens up a wealth of design opportunities for applications unavailable to designers only a few years ago. This design capability is of course nicely complimented by the increasing logical and memory potential of FPGAs, which now includes PowerPC cores embedded in the fabric of the Xilinx Virtex-II Pro FPGA. As emphasized, however, there remains much need for improvement in design tool features and capability, but this improvement should occur swiftly as the demand for such capability increases.

References

[1] Bradly Fawcett and John Watson, "Reconfigurable Processing with Field Programmable Gate Arrays", 1996 International Conference on Application-Specific Systems, Architectures, and Processors, August 1996, pg. 293.

[2] Russell Petersen and Brad L. Hutchings, "An Assessment of the Suitability of FPGA -Based Systems for use in Digital Signal Processing", 5th International Workshop on Field-Programmable Logic and Applications, Oxford, England, August 1995., pg. 293-302.

[3] Russell Tessier and Wayne Burleson, "Reconfigurable Computing for Digital Signal Processing: A Survey", Journal of VLSI Signal Processing 28, July 27, 2001, pg. 9-10.

[4] Scott McMillan and Steven A. Guccione, "Partial Run-Time Recongfiguration Using JRTR", Proceedings of the 10th International Workshop on Field-Programmable Applicatins, Lecture Notes in Computer Science 1896, 2000.

[5] Bradly K Fawcett and John Watson, "Reconfigurable Processing with Field Programmable Gate Arrays", 1996 International Conference on Application-Specific Systems, Architectures, and Processors, August 1996, pg. 293.

[6] Milan Vasilko, "DYNASTY: A Temporal Floorplanning Based CAD Framework for Dynamically Reconfigurable Logic Systems", Field-Programmable Logic and Applications, LNCS 1673, Springer-Verlag, 1999, pg. 124-133.

[7] Xilinx, "Virtex-II Platform FPGA Handbook", Xilinx, Inc., 2001.

[8] Gordon Brebner and Adam Donlin, "Runtime Reconfigurable Routing", Proceedings of the Reconfigurable Architectures Workshop, March 30, 1998, pg. 25-30.

[9] Xilinx, www.xilinx.com, Xilinx, Inc., 2002.

[10] Xilinx, "The Programmable Logic Data Book 1999", Xilinx, Inc., 1999.

[11] Xilinx, "Virtex-II Platform FPGA Handbook", Xilinx, Inc., 2001, pg. 56.

[12] Russel G Tessier, "Fast Place and Route Approaches for FPGAs", Ph.D. thesis, Department of Electrical Engineering and Computer Science, MIT, February 1999.

[13] Davin Lim and Mike Peattie, "Two Flow for Partial Reconfiguration: Module Based or Small Bit Manipulation", Xilinx, Inc., May 17, 2002.

[14] Katherine Compton, James Cooley, Stephen Knol, and Scott Hauck, "Configuration Relocation and Defragmentation for Reconfigurable Computing", IEEE Symposium on Field-Programmable Custom Computing Machines, April 2000, pg. 279.

[15] J.D. Hadley and B.L. Hutchings, "Design Methodologies for Partially Reconfigurable Systems", Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, 1996, pg 78-84.

[16] Philip James-Roxby and Steven A. Guccione, "Automated Extraction of Run-Time Parameterisable Cores from Programmable Device Configuration", IEEE Workshop on Field Programmable Custom Computing Machines, April 2000, pg. 153-161.

[17] Milan Vasilko, "DYNASTY: A Temporal Floorplanning Based CAD Framework for Dynamically Reconfigurable Logic Systems", Field-Programmable Logic and Applications, LNCS 1673, Springer-Verlag, 1999, pg. 124-133.

[18] Ian Robertson, James Irvine, Patrick Lysaght, and David Robinson, "Timing Verification of Dynamically Reconfigurable Logic for the Xilinx Virtex FPGA Series", ACM International Symposium on Field-Programmable Gate Arrays, 2002, pg. 127-135.

[19] Gordon Brebner, "Automatic Identification of Swappable Logic Units in XC6200 Circuitry", Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications, September 1997, pg. 173-182.

[20] John Stockwood and Patrick Lysaght, "A Simulation Tool for Dynamically Reconfigurable Field Programmable Gate Arrays", IEEE Transactions on VLSI Systems, September 1996, Vol. 4, No. 3, pg 381-390.

[21] Philip James-Roxby and Steven A. Guccione, "Automated Extraction of Run-Time Parameterisable Cores from Programmable Device Configuration", IEEE Workshop on Field Programmable Custom Computing Machines, April 2000, pg. 153-161.

[22] Eric Keller, "JRoute: A Run-Time Routing API for FPGA Hardware", 7th Reconfigurable Architectures Workshop, May 2000.

[23] Scott P. McMillan, Brandon J. Blodget, and Steven A. Guccione, "VirtexDS: A Virtex Device Simulator", SPIE, November, 2000.
[24] Edson Horta, John Lockwood, David E. Taylor, and David Parlour, "Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration", Design Automation Conference, June 10-14 2002.
[25] Xilinx, "Development System Reference Guide: Modular Design", Xilinx, Inc., 2002.
[26] Davin Lim and Mike Peattie, "Two Flow for Partial Reconfiguration: Module Based or Small Bit Manipulation", Xilinx, Inc., May 17, 2002.
[27] Edson L. Horta and John W. Lockwood, "PARBIT: A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs)", Department of Computer Science, Applied Research Lab, Washington University, July 6, 2001.
[28] Davin Lim and Mike Peattie, "Two Flow for Partial Reconfiguration: Module Based or Small Bit Manipulation", Xilinx, Inc., May 17, 2002.
[29] D. Lund, B. Honary, and M. Darnell, "A New Development System For Reconfigurable Digital Signal Processing", IEE 3G Mobile Communication Technologies, Conference Publication No. 471, March 2000.
[30] Hiroyuki Shiba, Takashi Shono, Kazuhiro Uehara, and Shuji Kubota, "Design and Evaluation of Software Radio Prototype with Over-the-Air Download Function", NTT Network Innovation Laboratories, 2001.
[31] Mark Cummings and Shinichiro Haruyama, "FPGA in the Software Radio", IEEE Communications Magazine, February 1999, pg. 108-112.
[32] Andrew A. Gray, Clement Lee, Payman Arabshahi, and Jeffrey Srinivasan, "Object-Oriented Reconfigurable Processing for Wireless Networks", Proceedings of the IEEE ICC 2002, April 28-May2, 2002.