NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

IMPLEMENTATION OF A QUADRATURE MIRROR FILTER BANK ON AN SRC RECONFIGURABLE COMPUTER FOR REAL-TIME SIGNAL PROCESSING

by

Kevin M. Stoffell

September 2006

Thesis Advisor:	Douglas J. Fouts
Second Reader:	Jon T. Butler

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2006	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE Implementation of a Quadrature Mirror Filter Bank on an SRC Reconfigurable Computer for Real-Time Signal Processing			**5. FUNDING NUMBERS**
6. AUTHOR(S) Kevin M. Stoffell			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Security Agency Fort Meade, MD			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

This thesis describes the design and implementation of a Quadrature Mirror Filter Bank on a general purpose Field Programmable Gate Array. The physical connections and signaling specifications for connecting an Analog to Digital converter to a Reconfigurable Computer system manufactured by SRC Computers Incorporated are discussed. Design and implementation of a fully functional prototype Quadrature Mirror Filter Bank is detailed, with a discussion for extending the functionality to larger more practical designs. Performance and device utilization results between the Quadrature Mirror Filter Bank implemented in VHDL, design elements implemented in the C programming language, and calculations made using high precision mathematical tools are compared, along with relative effort levels required to achieve results using the different hardware instantiation methods.

14. SUBJECT TERMS Quadrature Mirror Filter (QMF), Reconfigurable Computer, Field Programmable Gate Array (FPGA), Signal Processing, VHDL, Analog to Digital Converter (ADC)			**15. NUMBER OF PAGES** 134
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

IMPLEMENTATION OF A QUADRATURE MIRROR FILTER BANK ON AN SRC RECONFIGURABLE COMPUTER FOR REAL-TIME SIGNAL PROCESSING

Kevin M. Stoffell
Captain, United States Marine Corps
B.S., University of South Carolina, 1999

Submitted in partial fulfillment of the
requirements for the degree of

MASTER SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2006

Author:	Kevin M. Stoffell
Approved by:	Douglas J. Fouts Thesis Advisor
	Jon T. Butler Second Reader
	Jeffrey B. Knorr Chairman, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis describes the design and implementation of a Quadrature Mirror Filter Bank on a high-performance Reconfigurable Computer implemented with Field Programmable Gate Arrays. The physical connections and signaling specifications for connecting an Analog to Digital Converter to the Reconfigurable Computer system manufactured by SRC Computers Incorporated are discussed. Design and implementation of a fully functional prototype Quadrature Mirror Filter Bank is detailed, with a discussion for extending the functionality to larger more practical designs. Performance and device utilization results between the Quadrature Mirror Filter Bank implemented in VHDL, design elements implemented in the C programming language, and calculations made using high precision mathematical tools are compared, along with relative effort levels required to achieve results using the different hardware instantiation methods.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank David Caliga and Jon Huppenthal of SRC Computers, Inc. They both provided valuable information about the operation and programming of the SRC Reconfigurable computer system used in this research. Mr. Huppenthal was very gracious in providing his time and expertise in answering my questions concerning the inner workings of the SRC systems.

Professor Phillip Pace of the Naval Postgraduate School was very helpful in making the operations of a Quadrature Mirror Filter bank known to me, as well as providing a clear understanding of some of its uses.

I would also like to thank Professor Herschel Loomis and Professor Jon Butler, both of the Naval Postgraduate School, for the insights I have gained into advanced computer architecture and pipeline construction during their classroom instruction.

Contributions and guidance made by Professor Douglas Fouts as my thesis advisor, as well as classroom instructor, were instrumental in my reaching a level of understanding necessary to complete this work.

Material support from the National Security Agency was a vital element in the development of the hardware interface used for this project.

Finally, I would like to thank my wife Laura for the sacrifices she has made to support me in the completion of this thesis, and for the unwavering love and support she has provided me.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

The purpose of this thesis is the design, construction, and testing of a hardware implementation for a real-time Quadrature Mirror Filter Bank on an SRC-6 reconfigurable computer system. A Quadrature Mirror Filter (QMF) Bank is a type of wavelet decomposition filter system used for Digital Signal Processing (DSP). The use of a Quadrature Mirror Filter Bank as part of a larger system for the detection and classification of Low Probability of Intercept (LPI) radar signals is proposed in [1]. This thesis is part of a larger project working to develop the design methodologies for a practical implementation of LPI detection system on Field Programmable Gate Arrays (FPGA). As a secondary goal, this thesis evaluates the suitability of the SRC-6 system for use as a developmental platform.

This work uses the SRC-6 reconfigurable computer manufactured by SRC Computer Corporation as the developmental test platform for the development of the QMF. The model of SRC-6 at the Naval Postgraduate School contains four user-programmable Xilinx XC2V6000 FPGAs, which were the specific design targets for the work. The majority of the QMF design work was done in the VHDL Hardware Definition Language. However, a portion of the design was duplicated in the C language based programming environment developed by SRC for their reconfigurable systems. As a development platform, the SRC-6 interface and programming environment simplified portions of the work, and the portion of the work duplicated in C based language proved to require significantly lower development times at the expense of slightly increased hardware utilization at the FPGA level.

The work for this thesis consists of two main parts. The first portion is the construction and testing of a hardware interface between the SRC-6 system and a separate Analog to Digital Converter (ADC) board. The interface consists of the physical cabling and connection devices, the electrical specifications of the interface, and the logical interface designed for implementation on the SRC-6 to accept the incoming data and format it for further processing. The specific design of the interface and components is discussed in depth and complete information for duplication or extension of the interface is provided.

The second portion of the design consists of the design, implementation, and testing of a demonstration QMF in VHDL. Several design choices required for designing a QMF implementation and these choices are discussed in depth. Future extensibility of the QMF is discussed, along with several methodologies to increase the capabilities of the demonstration design. Special attention is given to a method of interleaving separate data streams along a single data path in order to reduce hardware utilization in the design. A method for interleaving data streams along a fully pipelined real-time data path is shown and proven in a working demonstration model.

The demonstration model QMF developed for this work consists of a filter bank that accepts six simultaneous input values from the hardware interface on each clock cycle. It then passes the incoming input values into a three stage QMF built using six tap Finite Impulse Response (FIR) filters as the primary elements of the design. By using delay elements and multiplexers, the design interleaves the filter outputs along a single data path allowing for minimal hardware to be used for the design.

Test results from the demonstration QMF is compared with results from a similar filter designed in MATLAB. The demonstration QMF shows comparable performance to the MATLAB based filter within the limits of accuracy imposed by the real-time processing limitations of the VHDL QMF and design choices made for its implementation. The output of the data path from the VHDL QMF is successfully de-interleaved, returning eight separate data streams from the continuous output.

A discussion of the limitations of the current design is given with recommendations for future work and the extension of the current design into a larger practical design.

I. INTRODUCTION

A. PURPOSE

Reconfigurable computer systems offer a number of performance advantages over general purpose computer systems and significant economy advantages over custom computing devices. A general purpose system is capable of a wide variety of tasks but must necessarily include all the hardware elements required to complete any task. As a result, although modern general purpose systems have become extremely fast, the optimization choices made in their construction require many simple operations, such as arithmetic functions, to require multiple clock cycles to complete. Competing space requirements on the chip restrict the quantity of hardware that can be dedicated to a particular task and subsequently limits the number of parallel operations that can occur in a general purpose processor.

At the other end of the spectrum are Application Specific Integrated Circuit (ASIC) chips, which contain only the hardware elements required for their designed task, and can contain sufficient quantities of hardware elements to complete large numbers of parallel tasks. While very efficient, they must be custom designed for a specific task and are incapable of any other task. As a result, ASIC designs are generally inflexible and have high development costs that tend to restrict their employment to high volume applications where the devices are disposable once the particular application for which they have been designed is obsolete.

In between these extremes lie reconfigurable computing devices. The most common reconfigurable device today is the Field Programmable Gate Array (FPGA). By itself, the FPGA is similar in some ways to an ASIC, although normally of lower performance. Developing FPGA applications is similar in cost to developing ASIC based designs but the FPGA device can be reconfigured to run a different application at a later time. The power of the FPGA is magnified when it is combined with a general purpose computer to provide a reconfigurable hardware asset to a computer system. In systems such as the SRC Computers Corporation SRC-6 reconfigurable computer, the advantages of a high speed general purpose system are joined with the dedicated hardware speed advantages of the FPGA. In addition, the cost of application development can be considerably lower on these systems as they can be programmed in the C programming language without specialized knowledge of hardware engineering.

This thesis explores the practicality of utilizing a reconfigurable computer system for capturing a digital signal in real-time and performing multiple parallel signal processing tasks. The specific application explored involves work done on using a Quadrature Mirror Filter Bank (QMF) in the detection and classification of Low Probability of Intercept (LPI) radar signals.[1] This thesis will detail the issues involved in physically connecting an external Analog to Digital Converter (ADC) to the SRC-6 reconfigurable computer system, the digital logic involved in reading the ADC data and converting it to the internal clock domain, and the digital logic involved in a practical QMF design. In Addition, working prototype designs for a real-time data capture and a QMF bank will be characterized and compared with respect to hardware utilization and accuracy.

B. DESIGN OVERVIEW

Figure 1 shows the basic data flow required to implement a QMF design on the SRC-6. Most of the effort in this thesis was devoted to the physical interface between the ADC and the SRC, the logical interface that reads the ADC data into the SRC, and the implementation of a QMF design. The signal source can be any source readable by the ADC within the Nyquist sampling rate of the ADC clock speed and will not be addressed in depth. The user interface between the MAP and the user will also not be significantly addressed in this work.

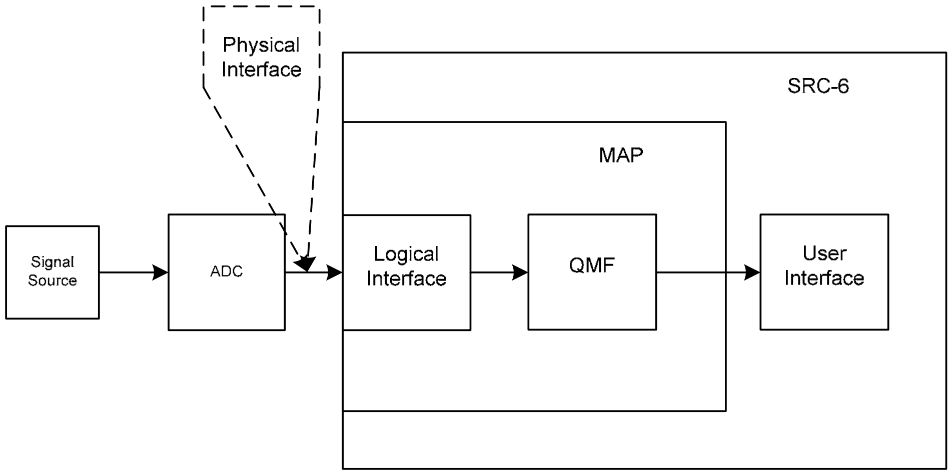


Figure 1. QMF Data Flow Diagram

1. Hardware Elements

This research addresses a specific hardware configuration. However, it will also attempt to draw conclusions about general hardware configurations. The specific hardware utilized for this work is a National Semiconductor ADC081500D Analog to Digital Converter (ADC) mounted on a custom evaluation board produced by Ballenger Creek Consulting. The cable configuration required to transfer the signal from the ADC board to the SRC-6 is examined and the hardware capabilities and configuration of the SRC-6 reconfigurable computer system will be addressed. In particular, elements of the Xilinx FPGA incorporated into the SRC-6 is discussed in detail with respect to their capabilities and limitations, as well as the effects of the hardware elements on the software and mathematical elements of the design.

2. Software Elements

This research covers the logical interface required to read external data into the SRC-6, a three stage QMF written in VHDL, a small Finite Impulse Response (FIR) filter written in VHDL, and a comparable FIR written in C but implemented on the SRC. The support software required to operate the SRC-6, as well as the software tools used to write and test the FPGA related code, is examined. Output comparisons of the performance of the QMF running on the SRC are made against a high precision QMF running in MATLAB. The MATLAB code used to make the comparisons and perform post processing on the data is examined.

3. Mathematical Elements

Due to the limited time available to complete each stage of the QMF processing and the constant flow of incoming data, normal floating point mathematical operations, such as done by a general purpose processor, are not practical for a real-time data capture and processing system. As a result, methods to reduce the processing requirements for the required arithmetic operations are examined along with binary fixed point arithmetic in order to reduce the hardware requirements but still maintain an acceptable degree of precision. In this type of number system, maintaining knowledge of the decimal point location is the responsibility of the designer. In addition, great care must be taken when designing the binary number formats to ensure that overflow conditions are either not possible, within a set of filter coefficients, or can be dealt with effectively by the system. For this project, binary bit widths for each stage of calculations were chosen to exclude the possibility of overflows resulting from the calculations but maintained only the minimum total binary word size necessary to maintain a reasonable precision in the final result.

C. THESIS ORGANIZATION

The remainder of this thesis is organized as follows:

- Chapter II discusses previous and related work in similar areas. It also details some of the potential applications of the QMF bank with respect to detection and classification of ELINT signals.
- Chapter III gives a basic overview of the major hardware elements used in this work, as well as their capabilities and limitations, and will provide a functional reference to the various hardware elements with respect to the requirements of this project.
- Chapter IV provides an in-depth review of the hardware interface between the ADC and the SRC-6. It covers the areas listed in Figure 1 as the ADC, the physical interface, and the logical interface.
- Chapter V provides an in depth discussion of the design concepts developed and employed to provide a real-time QMF implementation. It covers the actual implementation of a prototype QMF design in VHDL, a FIR filter design in C for

implementation on the SRC-6, and the support software written to implement the design.

- Chapter VI discusses the results of the implementations with respect to hardware utilization and relative error rates compared to high precision calculations made using MATLAB on the same raw ADC outputs as the VHDL filters.
- Chapter VII provides a summarization of results, draws some conclusions based on the work, and suggests some areas for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

A. QUADRATURE MIRROR FILTER BANK AS PART OF AN LPI ELINT DETECTION SYSTEM

A detailed description of a possible Electronic Intelligence (ELINT) detection and classification system is given by [1]. Figure 2 shows a block diagram of the proposed system. This work is concerned with the portion of the design connecting the Digital Receiver to the system and the Quadrature mirror filtering. Work on implementing other portions of the design is ongoing but is not the focus of this thesis. Results from porting the Image Analysis section shown in Figure 2 to the SRC-6 are given by [2].

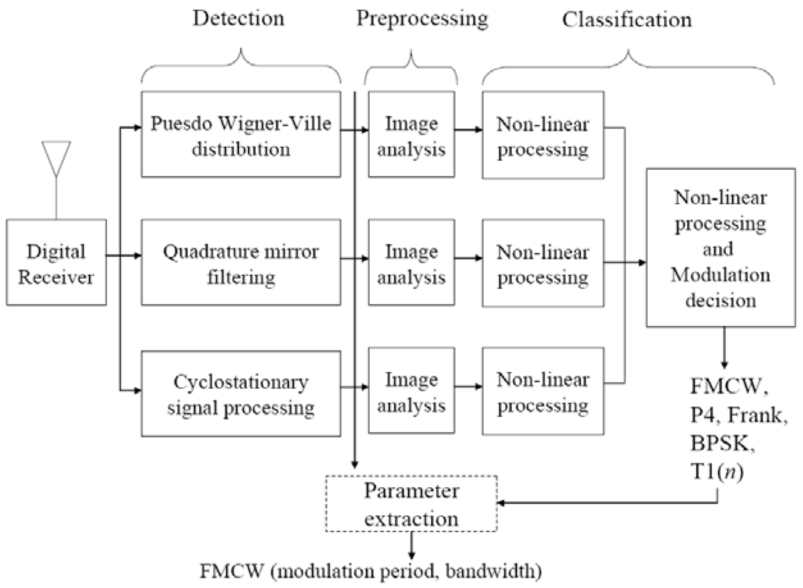

Detection
Preprocessing
Classification
Puesdo Wigner-Ville distribution
Image analysis
Non-linear processing
Digital Receiver
Quadrature mirror filtering
Image analysis
Non-linear processing
Non-linear processing and Modulation decision
Cyclostationary signal processing
Image analysis
Non-linear processing
FMCW, P4, Frank, BPSK, T1(n)
Parameter extraction
FMCW (modulation period, bandwidth)

Figure 2. LPI Detection and Classification system (From [1])

A Quadrature Mirror Filter bank is a type of wavelet decomposition filter that reduces the time domain resolution of a signal to produce a higher resolution in the frequency domain. A more detailed explanation of the elements of a QMF is given in Chapter V but the basic design is shown in Figure 3. The blocks labeled G and H in Figure 3 represent high and low pass digital filters. The focus on this work is to determine practical methods for the implementation of a QMF on a particular hardware suite and to provide generalized guidelines for QMF implementation in similar types of reconfigurable logic.

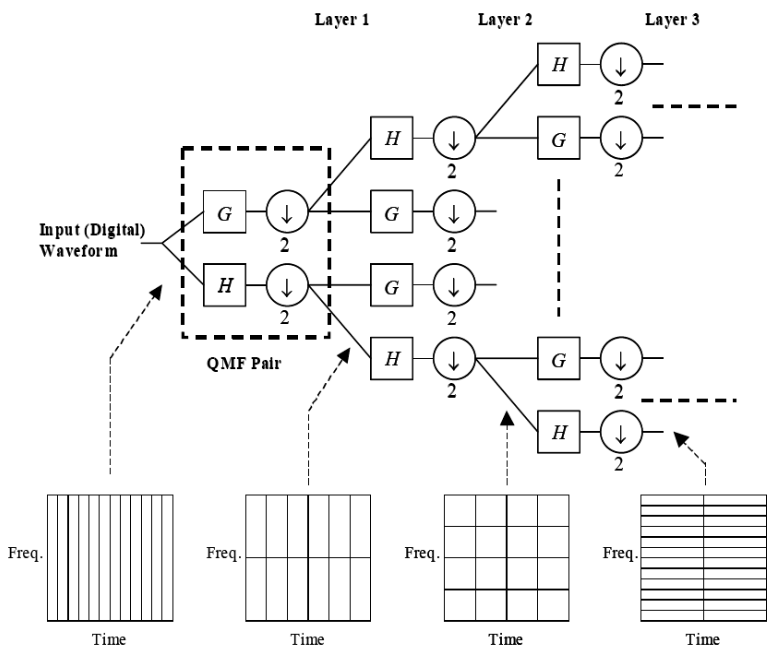


Figure 3. Quadrature Mirror Filter Bank (From [1])

B. RELATED WORK

Most of the individual elements required for a real-time implementation of a QMF on the SRC hardware have been explored in previous research and designs, though a complete implementation has not been realized.

1. Filter Design Principals

In general, filter design principals for FPGA implementation have been well discussed in the literature [3,4,5]. The designs in this work are a compilation of several standard filter design practices. Various options for filter element design will be discussed where appropriate. In particular, [3] provides some useful discussion on wavelet decomposition filter trees similar in nature to the target QMF with regards to hardware reutilization.

2. General FPGA Design Practices

The FPGA in the model of SRC-6 available for testing is an older Xilinx Virtex-II FPGA, which is a very well known chip. A large volume of application notes and general design guidelines such as [6,7,8,9,10] have provided useful design elements. Almost all the code used in this work is original and based on standard design practices. The exception is the multiplier element outlined in [6]. Specific code for this multiplier element was downloaded from the Xilinx website along with the associated application note, [6], and modified for this particular application by replacing the multiplier output registers with configurable pipeline delay elements.

3. Previous SRC-6 Hardware Interface Designs

Previous work has been done at the Naval Postgraduate school on a hardware interface system between and ADC and the SRC-6 computer system, as detailed in [11,12]. The previous design proved effective and was completely compliant with the listed specifications of the SRC-6 model available to NPS. The previous design was, however, limited in effective sampling rate and fairly complex due to the necessity of meeting the listed interface specifications of the SRC-6. The signal input specifications listed for the SRC-6 at NPS indicated it was only compatible the Low Voltage TTL (LVTTL) specification and the previous design work was based on that requirement. Since LVTTL outputs are not standard on most commercially available high speed ADC's, a fairly complex design was required to process the ADC output for input into the SRC-6. Also, the previous design was based on clocking the ADC using the SRC-6 clock, which also limited the possible ADC sampling rate to the maximum clock output rate of the FPGAs in the SRC-6 (approximately 400 MHz). This work will attempt to extend the previous work to overcome the speed limitations imposed by the LVTTL standard and sampling rate limitations from the limits of the SRC-6 clock.

Since the initial hardware interface work at NPS, SRC computers has developed an ADC board specifically designed for connection to the newer SRC systems. It uses the Low-Voltage Differential Signaling (LVDS) specification and is not meant to be compatible with Revision C and older SRC systems. As of September 2006 it is not yet available for purchase but is expected to be available in the near future. Future work with the SRC involving external connections may be able to benefit from a commercially designed and tested interface. Limited information on the specifics of the SRC ADC board is available at this time, other than it is capable of a 2GSps sampling rate.

THIS PAGE INTENTIONALLY LEFT BLANK

III. HARDWARE ELEMENTS

A. OVERVIEW

For this work, there are two main hardware elements, the ADC board and the SRC-6 reconfigurable computer system. The physical connection used on the SRC-6 was a direct cable connection to the MAP General Purpose Input Output (I/O) port, which is directly connected to pins on the internal Xilinx Virtex-II FPGA. Much of the effort of this project involves low-level optimizations within the FPGA itself. Certain hardware resources within the FPGA require a detailed description for the design choices made in this work to be fully explained. The Xilinx FPGA contained within the MAP board will be described as a separate component.

B. SRC RECONFIGURABLE COMPUTER

The reconfigurable computer system used for this work is the SRC-6 computer manufactured by SRC Incorporated of Colorado Springs, Colorado. The main elements of the SRC-6 consist of the microprocessor, the MAP board, and the custom software environment developed by SRC for programming the system. The SRC-6 contains all of the support hardware and software necessary to develop FPGA projects. The SRC-6 software environment allowed the development of a QMF prototype with reasonable effort. However, significant development effort was devoted to VHDL code. As a result, this work does not fully take advantage of the capabilities of the SRC-6 software environment. Of additional interest is the General Purpose Input Output port (GPIO), also referred to as the chain port, on the MAP board within the SRC-6, which allows multiple MAP units to be interconnected or interfaces connecting other equipment to be constructed. Figure 4 shows a simplified block diagram of the SRC-6 internal structure. This diagram represents the older model MAP board used for this work and omits components not directly related to this work.

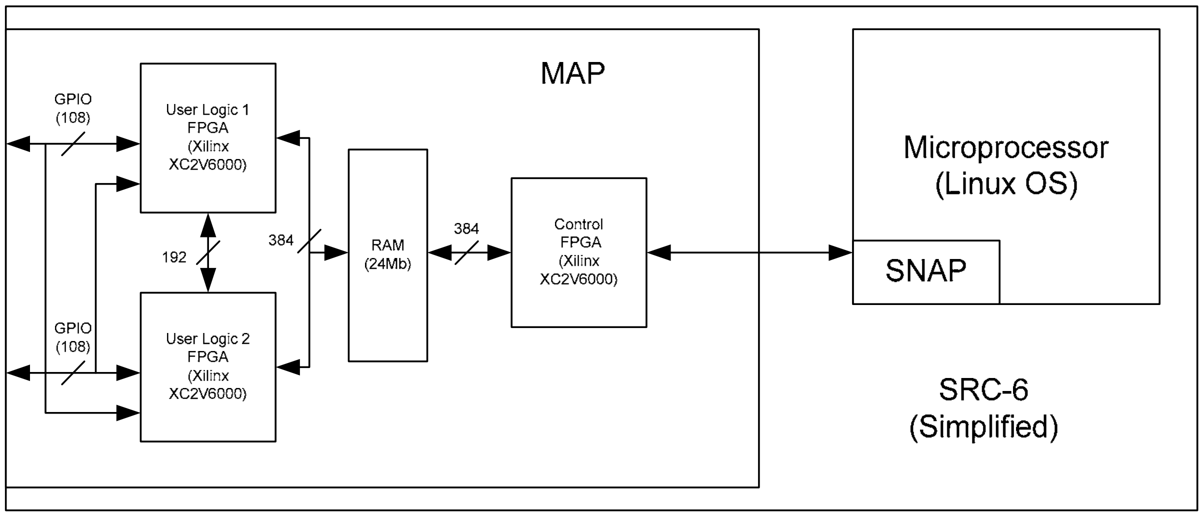

MAP
GPIO (108)
User Logic 1 FPGA (Xilinx XC2V6000)
192
384
RAM (24Mb)
384
Control FPGA (Xilinx XC2V6000)
GPIO (108)
User Logic 2 FPGA (Xilinx XC2V6000)
Microprocessor (Linux OS)
SNAP
SRC-6 (Simplified)

Figure 4. Simplified SRC-6 Architecture

1. Microprocessor

The microprocessor in the SRC-6 used for this work is a standard rack mounted 2U server form factor with dual 2.8 GHz Intel Xeon processors. It runs the Linux Fedora Core 3 operating system. The microprocessor has a SNAP interface board that has been installed in one of the standard memory slots on the microprocessor mother board. It is physically connected to the MAP board using micro-coaxial cables, allowing high rate data transfers between the Microprocessor and the MAP board. The SNAP interface creates a communication path between the Microprocessor and the MAP board by allowing the software on the microprocessor to treat the MAP as if it were a memory location. This avoids the necessity of extensive modifications to the computer architecture of the microprocessor. Currently, the SNAP interface only allows half-duplex communication between the MAP and microprocessor. This is not pertinent to the current work but may be an issue in development of a real-time signal processing system using the SRC-6. Since data can only flow in one direction at a time, and in a real-time signal processing system it can be assumed that the data flowing from the MAP would be continuous, control signaling from the Microprocessor to the MAP is potentially problematic. Future versions of the SRC may remedy this situation, but for the purposes of this work, data was sampled in fixed size blocks of 60,000 samples and transferred to the microprocessor for storage, with the program terminating after the conclusion of the transfer. Larger block transfers are well within the capability of the SRC, however, file sizes for the stored data quickly become unmanageable. The SRC-6 is also capable of continuously streaming data across the interface to the microprocessor, however, that level of functionality is beyond the scope of this work.

2. MAP

Figure 4 details the layout of the pertinent components of the SRC-6 MAP board. This work is based primarily in the User Logic FPGA of the MAP board. The onboard RAM is only used as an intermediary storage location for data samples and is not used for any other purpose in this work. Other SRC-6 applications may use the onboard RAM, and an SRC-6 programmer must normally be cognizant of a number of memory access issues but those are beyond the scope of this work.

Of particular interest at the MAP level is the interconnection between the two User Logic FPGAs, which is a very wide, high-speed data bus. Also of interest are the GPIO port inputs that allow connections with external devices or additional MAP units. One GPIO port is labeled as the GPIO-in port while the other is labeled as the GPIO-out port. While both GPIO ports have connections to both User Logic chips, the actual pin assignments on each FPGA are different. This work is concerned only with the GPIO-in port connected to User Logic 1.

There is one noted operational detail of the MAP board that is not documented by SRC that could prove to be of interest to future expansion of the current work. Once a process has been run on the microprocessor that programs the User Logic chips on the MAP, the logic is not reconfigured until the next time the User Logic is accessed. In a configuration with multiple MAP boards chained together using the GPIO ports, the MAP boards can be configured by activating processes on the microprocessors that instantiate logic on the MAP boards, and then terminate and leave the MAP boards functioning autonomously. In this situation, only one microprocessor connection needs to maintain a running process to interface with multiple MAPs running portions of a program that spans multiple MAP boards. It is unlikely that a practical system would make use of this special case. However, it opens up a number of possibilities for prototype testing of large, FPGA based systems using the SRC interface and support software. Operations that involve interaction between the User Logic portion of the MAP and other MAP elements have not been observed in this special case scenario and further study would be necessary to verify functionality.

3. Software Environment

The software development environment of the SRC-6 is known as CARTE and was developed by SRC for their systems. CARTE includes a C superset programming language, which allows any C programmer to write software that is capable of instantiating dedicated logic in hardware to allow significant improvements in performance for some types of applications. This functionality is the major advantage of the SRC system, which allows for a significant reduction in development time over the use of Hardware Definition Language (HDL) based designs with comparable performance. SRC currently provides a significant number of dedicated functions, called macros, which allow a programmer familiar with the SRC system to produce highly optimized code, which are detailed in [13]. While it is possible to run most C code on the SRC, for code to written for the SRC to be fully optimized and gain maximum performance, an in-depth study of the system or attending a training course [14] is highly recommended.

The CARTE libraries contain many useful functions (referred to as macros) that have been optimized for the SRC. All of the macros designed for off chip communication assume the communication will be between User Logic chips. For access to custom off-chip hardware, such as used in this thesis, user-designed macros written in Verilog HDL or VHDL can be integrated into the C code and are accessed as if they were a C type function from within the code. Since the hardware interface portion of this work was written as a VHDL macro, limited attention was given to the C Code portions of the project. A comparison is made for the hardware utilization of a FIR filter written in C versus one written in VHDL. In addition, benchmarking of the SRC-6 performance has been done in [11,12,15] and is not the focus of this work.

Detailed listings for the file formats and CARTE specific software requirements are contained in [13]. For purposes of this work, the important files consist of the main C file (.c extension), the MAP code file (.mc) extension, and the user macro file (.v extension if written in Verilog or .vhd extension if written in VHDL). Code meant to execute on the Microprocessor is placed by the programmer into the main.c file, which in turn calls the .mc file containing the code meant to execute in dedicated logic on the

MAP. Code in the .mc file can call user defined macros written in HDL, and contained in the user macro file, as if they were a C style functions. There are additional support files necessary to implement user macros which are detailed in [13], but a full description is not necessary for this work.

A programmer attempting to compile software for the SRC has a number of compilation options that can significantly affect compile times. If a program is written only in C, the user can compile the program in the debug mode, which effectively creates an executable that runs the MAP code in an emulation mode. This significantly reduces the compile time since the compiler does not have to synthesize and perform a Place And Route (PAR) operation for all the circuitry destined for MAP execution. For programs utilizing user macros, the macro writer has the option of including a piece of C code that emulates the functionality of the HDL macro when the program is compiled in debug mode. Unfortunately, debugging large user macros becomes very tedious as the macros themselves do not compile in debug mode. Only the C code meant to emulate the functionality of the macro compiles. While there is a simulation mode available as a compile operation, NPS does not have the software licenses for the simulation mode compilation. Hardware mode compilation is always available, but for larger programs, this may require several hours to complete. This is because all the code must be synthesized by the HDL compiler and the resulting hardware design must undergo a multiple pass PAR operation, where each component and net is mapped to a physical location on the FPGA. Full hardware compiles with large designs normally require several hours to complete.

C. XILINX FPGA

While the Xilinx XC2V6000 User Logic FPGA is actually part of the MAP board in the SRC-6, the focus of this work was at a very low hardware level and certain key design elements require a basic understanding of the FPGA functionality. The basic internal structure of the target FPGA is shown in Figure 5, which is from [16]. Table 1 shows the relative component capacities of the XC2V6000 chip used in this thesis and of the XC2VP100 chip that is integrated into newer model SRC systems. The number of available multipliers and block RAM devices is of interest later in this work when capacity planning for practical QMF implementations is discussed.

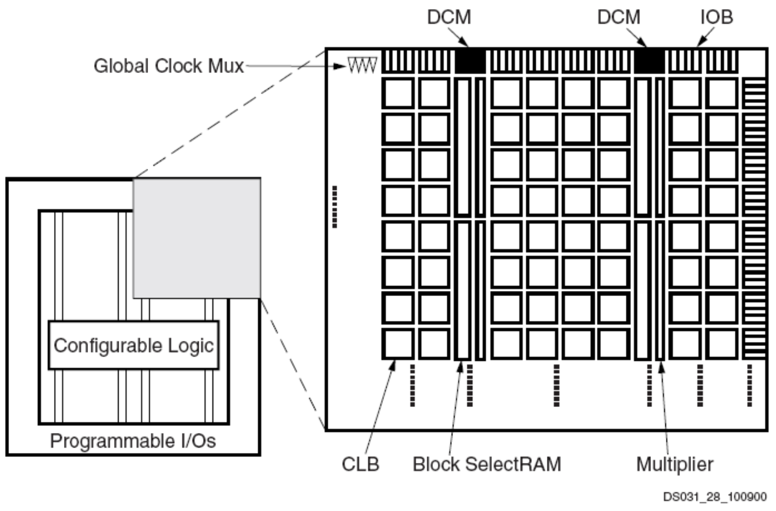


Figure 5. Xilinx Virtex-II Architecture Overview (From [16])

		CLB (1 CLB = 4 slices = Max 128 bits)				SelectRAM Blocks			
Device	System Gates	Array Row x Col.	Slices	Maximum Distributed RAM Kbits	Multiplier Blocks	18 Kbit Blocks	Max RAM (Kbits)	DCMs	Max I/O Pads(1)
XC2V6000	6M	96 x 88	33,792	1,056	144	144	2,592	12	1,104
XC2VP100			44,096	1,378	444	444	7,992	12	1,164

Table 1. Selected Xilinx FPGA Component Capacity (After [16,17])

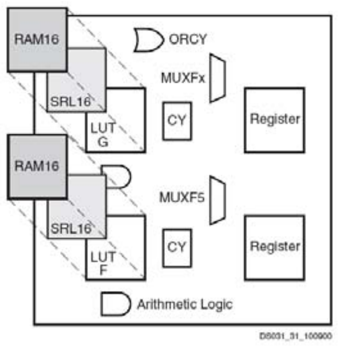


Figure 6. Xilinx FPGA Slice Configuration (From [16])

1. FPGA Structure

The basic structure of the Xilinx Virtex-II FPGA is shown in Figure 5, with component capacities listed in Table 1. Figure 6 shows the basic logic of a single slice, of which four make up a Configurable Logic Block (CLB). Not represented in the figures is the switch matrix and high speed interconnects between components. The switch matrix consists of numerous data lines that surround all of the components shown in Figure 5.

They allow signals to be routed to any location on the chip from any other location on the chip. In addition to the global switch matrix, there are low delay interconnects between all adjacent (horizontally, vertically, and diagonally) CLB elements and between adjacent CLB elements, Multipliers, Block Select RAM, and IOB pads. There are also dedicated low delay interconnect lines running vertically in the CLB columns to connect specialized logic in each slice that is used for arithmetic operations, shift operations, and multiplexing operations. Each slice contains two configurable logic elements that can act as small 16-bit RAM or ROM modules and Look Up Tables (LUT) which can implement any arbitrary 4-input combinational logic function, or function as a 16-bit shift register.

2. Key Elements

The general FPGA layout also includes Digital Clock Managers (DCM) and Global Clock Buffers (physically located with the Global Clock Multiplexer elements in Figure 5). The DCMs are physically located along the top and bottom of the chip while the clock buffers are located at top and bottom center. The physical placement of the DCM modules and the clock buffers is important to this work as several DCM modules are required for the designs described in later chapters. The DCM modules are capable of creating modified clock signals by phase shifting the clock, multiplying the clock, or dividing the clock, or any combination of these modifications, while maintaining a phase lock with a source clock.

Around the edge of the chip are Input Output Block (IOB) elements that contain the IO pads, Electro Static Discharge (ESD) protection, Double Data Rate (DDR) registers to clock in data from external lines, and IO buffers capable of transmitting and receiving a wide variety of signal standards on and off the chip. The IOB elements are grouped into eight IO banks equally spaced around the edge of the chip. Each IO bank has individual power connections that allow different IO banks to be configured for individual IO electrical specifications. The IOB elements are able, with the correct electrical connections, to receive and transmit 33 different specified transmission standards [16] and another 16 with Digitally Controlled Impedance (DCI) termination. In addition, each IOB is connected to one adjacent IOB and the two share differential signal input and output buffers for use with differential signal specifications.

As stated in Chapter II, the model SRC computer at NPS is only designed for the LVTTL IO electrical specification. Jon Huppenthal, of SRC, has stated that this was because the design of the older model SRC systems did not include some of the necessary electrical connections for transmission of differential standards, or the single ended standards that require a different reference voltage than the LVTTL standard. However, there was no reason why the FPGA IOB elements could not be configured to receive the different standards. This work makes use of this distinction in the interconnection of the ADC board to the SRC-6 using the LVDS 3.3V electrical specification with DCI enabled.

Two portions of the CLB slice are key elements used by this work. Figure 6 shows a small block at the bottom labeled arithmetic logic. What this actually represents is a dedicated hardware block that functions as a high speed Carry Look Ahead (CLAH) logic element and is connected by high speed interconnect to other CLAH elements vertically in columns of slices in the chip architecture. These elements allow large, very fast adders to be instantiated in hardware for very low device utilization. The high speed adders are a key element of the designs in this work. The other key element in the slice is the SRL16 configuration of the two configurable logic elements located in each slice. In the SRL16 mode, the element acts as a single bit shift register that can be configured to delay a bit stream from 0 to 16 clock cycles. When in 0 delay mode, the element effectively functions as a logic flip-flop. When combined with the vertical interconnect lines, 18 slices can implement a pipeline delay of up to 16 clock cycles on a 36-bit binary value and 36 slices can implement a 32 clock delay on a 36-bit value. This functionality, along with the high speed adders that can be implemented using the dedicated slice logic, allow for very inexpensive, pipelined arithmetic modules that make up the main elements of the designs in this work.

The final two key elements of the FPGA are the Block SelectRAM elements and the 18x18 Multiplier blocks shown in Figure 6 running vertically between sections of CLB elements. An important issue concerning these elements is that they are grouped with a Block RAM next to a Multiplier element and they share physical input lines. Basically, this means that, if both devices are in use simultaneously, the width of data stored in the Block RAM is limited to 18-bits wide. This is an important consideration that will be discussed in later chapters. While the multiplier blocks are very fast, by the nature of the binary multiply operation, they are not able to complete a full 18x18-bit multiply producing a reliable 36-bit output in a single clock cycle with the addition of normal routing delays. This produced some specific design concerns that are detailed in later chapters.

3. Xilinx Tools

While not directly related to the SRC-6, Xilinx produces some software tools for the creation of FPGA designs that warrant some attention in this section. Xilinx produces a suite of FPGA design tools known as the ISE Foundation suite of tools. The tools include software for editing schematics, Verilog, or VHDL based designs. The software suite includes integrated design verification and simulation tools. Xilinx schematic capture for conversion to VHDL was used for a portion of the macro code design, and the remainder of the code was written directly in VHDL in the Xilinx editor. The Xilinx editor is compatible with LINUX file conventions (non printing character issues) and files modified in the Xilinx editor were immediately transferable from the Windows based workstation on which most development took place to the SRC-6, whereas some other editors (including Windows notepad) produced incompatibilities within the design files.

The Xilinx tools are integrated with a ModelSim simulation tool for design verification and also include all Xilinx FPGA layouts and capacities. The Xilinx tools were invaluable for small scale simulations and component testing but the older versions of the tools licensed by NPS did encounter some difficulties simulating larger portions of the design and proved incapable of fully simulating the design performance. There is some hardware instantiated in the SRC-6 FPGA by the SRC compiler for control and connection to the other devices in the MAP for which the design files are unavailable. Thus, the Xilinx tools could not accurately produce the full chip design for testing. SRC does make simulation files available, but they require the user to simulate the design on the SRC, and as stated in the preceding section, NPS does not currently have a software license available for the third-party simulation software on the SRC.

D. ANALOG TO DIGITAL CONVERTER

The Analog to Digital Converter used for this work is the National Semiconductor ADC08D1500 chip mounted on a custom evaluation board produced by Ballenger Creek

Consulting. The complete data sheet for the ADC chip is available in [18]. Full specifications for the evaluation board are not available.

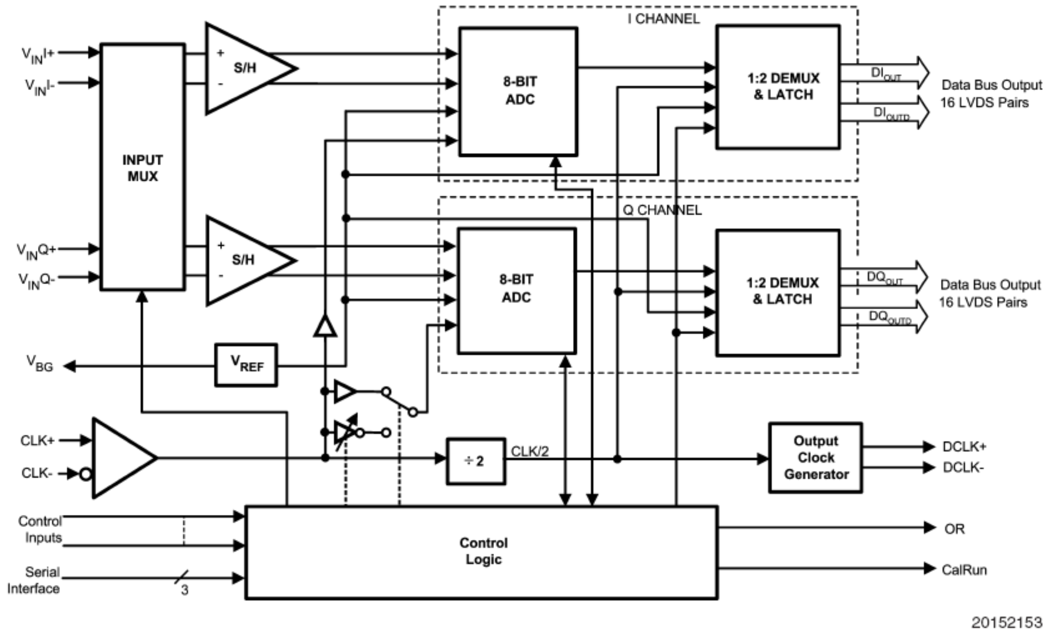


Figure 7. ADC Block Diagram (From [18])

1. ADC Specifications

The ADC08D1500 chip is a single or dual channel ADC capable of sampling two channels at 1.5GSps or a single channel at 3GSps in Dual Edge Sampling (DES) mode. All signal inputs and the clock inputs are differential. All output samples are 8-bit binary values ranging from 0 to 255, with 128 equating to a zero voltage differential on the signal input pins. Complete electrical specifications for the chip are in [18]. The sample outputs on the chip are in the LVDS specification. As shown in Figure 7, the input clock enters the chip on the left and is sent to the two ADC units and a clock divider that divides the clock period by two before sending the clock signal off chip in the LVDS IO specification. Figure 8 shows the timing diagram for the output samples from the ADC when it is operating in Single Data Rate (SDR) mode. Two 8-bit samples are available for the full clock cycle of the ADC output clock signal from each of the two channels when the chip is operating in SDR mode. Depending on a control setting, the output clock from the ADC can transition on either the beginning or middle of the data valid window for the output samples. In DES mode, the input clock is phase shifted 180 degrees for channel 2 and four output samples are available for each cycle with the data samples interleaved between the four 8-bit outputs.

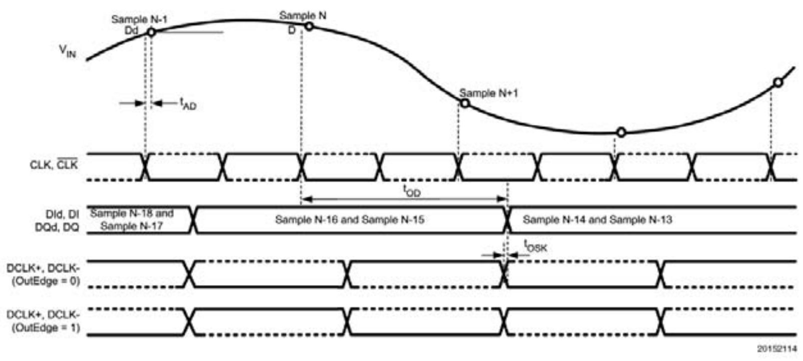


Figure 8. ADC Output Timing in SDR Mode (From [18])

2. Board Specifications

The ADC used for this work is mounted on a custom evaluation board shown in Figure 9. The ADC chip can be seen just to the left of the cables connected on the right of the board and just to the right of the two coaxial cables connected in the center of the board.

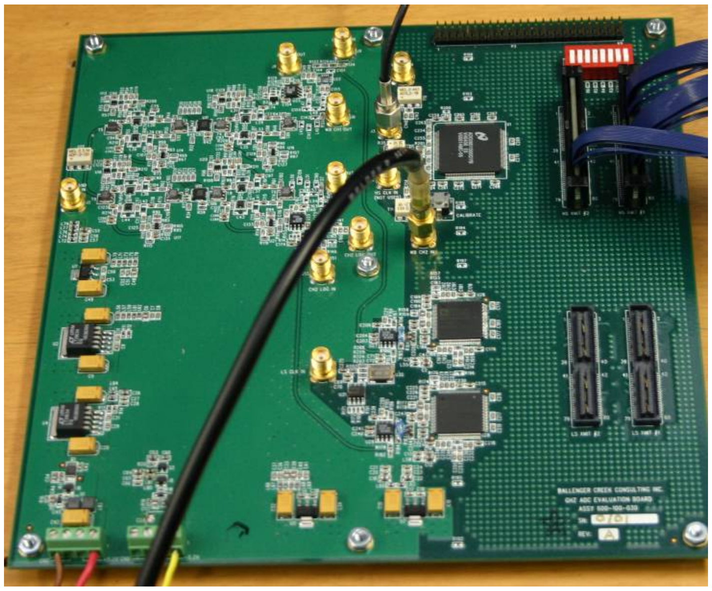


Figure 9. ADC Board

The two cable connections to the right of the board are both 80-pin MICTOR connectors that provide connections for the 32 LVDS pair output lines for the ADC samples and one LVDS pair for the clock output from the ADC. The two coaxial cables connected to the board in Figure 9 are the clock input cable (upper cable) and the input signal cable connected to the ADC channel-2 input connector. Transformers and circuitry mounted on the board conditions single ended input signals into a differential standard suitable for input into the ADC chip. The board was acquired by NPS for work on another project but was of interest to this work due to the LVDS output. Subsequently, complete specifications for the board are not available.

Input clock waveforms of sinusoidal or square forms with a 50% duty cycle and between 1 and 2 Vpp amplitudes were found to be satisfactory for clocking the ADC. However, the ADC board did prove to be very sensitive to variations in the clock input waveform at higher frequencies. The channel input is capable of accepting any waveform ranging from 0 to approximately 800 mVpp.   The board is configurable for two different input waveform voltage ranges by means of the row of switches mounted in the upper right corner of the board that drive several control pins on the ADC chip. The specific switch setting used for this work will be detailed in the hardware interface section. The board requires a +5.2VDC, -5.2VDC, and ground input for power. Onboard voltage regulators provide appropriate power to all board components from the three input wires connected to the power block on the lower left of the board. Finally, there is a small push button switch located adjacent to the channel-2 input connector for activating the self calibration mode on the ADC chip. It is intended that this button be pressed and released after the ADC board has been powered up to initiate self calibration of the ADC.

The configuration of the board does not allow the ADC to enter certain modes of operation. Specifically, the row of switches in the upper right of the board is connected to control lines running to control pins on the ADC chip. The control lines are driven high by pull-up resistors connected to a positive voltage when the switches are open but pulled to ground when the switches are closed. Unfortunately, as specified in [18], for the ADC to enter DES mode, pin 127 on the ADC chip must either be floating or connected to one-half the chip operating voltage. The board design does not allow for this, only ground or high voltage settings are possible, and the specific control lines are not accessible for external modification. In addition, pin 4 on the ADC must be floating or at one-half chip voltage for the ADC to enter the Dual Data Rate (DDR) mode of operation. These limitations prevented some explorations of the maximum attainable data sampling rate of the SRC-6 when connected to this ADC. However, both output channels were tested separately. It is reasonable to assume that by using an identical ADC chip on a different board, effective sampling rates used for this work can be doubled by placing the ADC in DES mode and doubling the number of inputs into the SRC-6.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. HARDWARE INTERFACE

A. INTRODUCTION

The hardware interface between the ADC board and the SRC-6 system consists of two main parts. First, the physical interface consists of the actual cable connections and the electrical specifications of the interface. The physical portion spans from the two 80-pin MICTOR connectors on the ADC board to the actual input pins of the FPGA within the SRC-6. Second, the logical interface includes all of the hardware that needs to be instantiated in the FPGA to capture the signals from the FPGA pins and condition the signals for further processing within the FPGA. Figure 10 shows the signal path from the ADC through the major components of the hardware interface. Once the signal leaves the First-In First-Out (FIFO) buffer, the signal has been converted to the clock domain used by the SRC-6 and is ready for further processing with logic instantiated with an HDL as a user macro, or passed back to the C code instantiated logic. The following sections will detail the individual portions of the design.

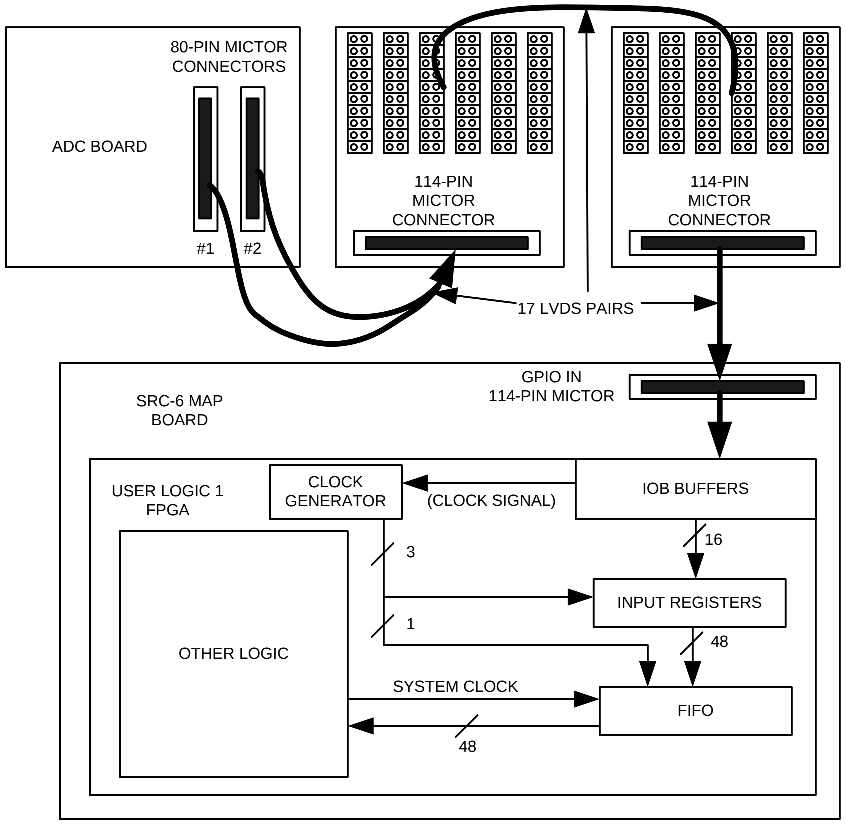


Figure 10. Hardware Interface Block Diagram

B. PHYSICAL INTERFACE

The physical interface portion begins with a custom manufactured cable from Tyco Electronics that combines specified pins on two 80-pin MICTOR connectors into a single 114-pin MICTOR connector, of the type used by the SRC-6. This cable is connected to a board that breaks the 114-pin MICTOR connector out into 114 2-pin pads. Each 2-pin pad consists of a signal pin and a ground pin. High quality 50 ohm coaxial jumpers are used to connect the 2-pin pads carrying signals (66 total, however only 34 are used) to an identical board which is directly connected to the SRC-6 MAP board GPIO-in port via a 114-pin MICTOR cable. Figure 11 shows the setup used for testing. The actual connection to the ADC board is visible in Figure 9. All of the switches on the ADC board (upper right corner of Figure 9) are set to OPEN, except for switches one and two, which are set to CLOSED. These settings configure the input voltage range of the ADC, the output voltage of the ADC, the calibration delay, and set the output clock to transition in the middle of the data valid window for the output data (equivalent to OUTEDGE = 0 in Figure 8). The clock output transition is an arbitrary choice, but the code must be optimized for a particular choice to account for internal delays in the FPGA.

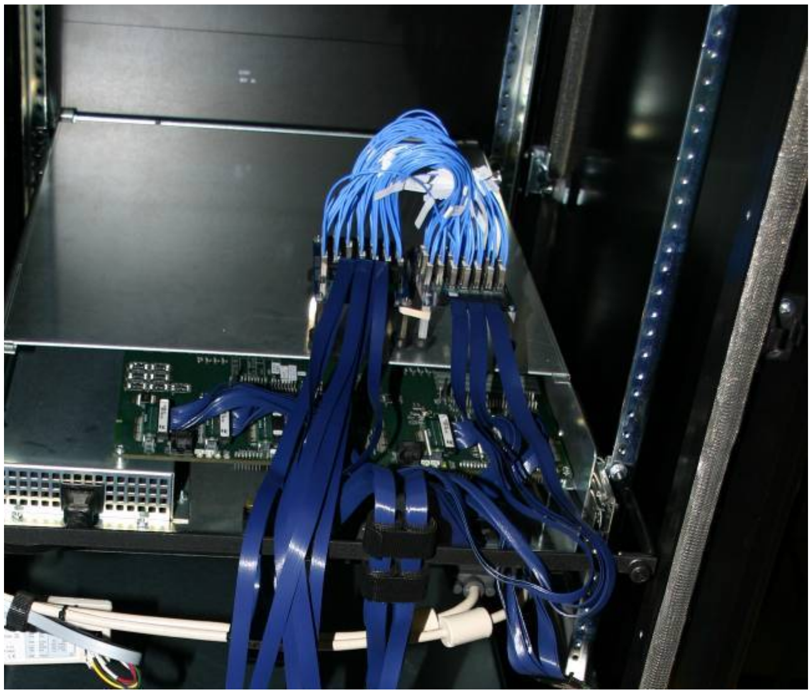


Figure 11. Physical Interface

1. Electrical Specifications

Detailed electrical specifications for the ADC chip are given in [18]. The ADC chip outputs are directly connected to the output pins in the two MICTOR connector blocks located on the ADC board. The ADC output is Low-Voltage Differential Signaling (LVDS), with approximately 700 mVpp difference between the positive and negative signal lines for each data bit. The MICTOR connector is a high quality, low loss connector type. All of the cables are of high quality 50 ohm micro-coaxial cable suitable for transmission of high frequency signals over distances ranging from a few inches to several feet, depending on the signal frequency. The short jumpers connecting the two breakout boards are of high quality 50 ohm coaxial cable. However, the actual two-pin connector is effectively a balanced transmission line at higher frequencies while the jumper coaxial cable is unbalanced. The multiple balanced to unbalanced transitions is non-optimal and proved unsatisfactory for achieving the target sampling rates. However, it proved to be a workable solution for lower frequency transmissions.

It proved impractical to conduct direct electrical testing on the physical connection between the ADC and the SRC. The extremely small size of the individual signal lines and the proprietary nature of the MICTOR connectors limited the options for connecting test equipment without signal distortion caused by the test equipment connections themselves. Subsequently, it was determined by experimentation that the cable setup would reliably transfer data at low error rates up to approximately 340 MHz (seen at ADC sampling rates of 680 MSps). At this frequency, the signal quality seen on the input pins of the SRC-6 begins to degrade significantly and becomes unusable by 345 MHz.

The custom cables fabricated for this project by Tyco Electronics would, ideally, be used to directly connect the ADC board to the SRC-6, reducing the total cable length and removing the two breakout board connections. This would significantly reduce the noise on the signal lines caused by the slight impedance mismatch inherent in any physical connector of this type and the multiple balanced to unbalanced transitions. The connections at each end of the short jumpers used to connect the two breakout boards would be especially desirable to remove, as they provide a much lower quality connection than the MICTOR cable connections. The next section details the reasons for the non-optimal physical connection.

2. Pin to Pin Connection Requirements

Inquiries were made to custom cable vendors concerning the fabrication of the custom cable used to combine the two 80-pin MICTOR cables into a single 114-pin connector. It was determined that a custom cable could be made but the expense for certain configurations might be prohibitive. Fabrication costs could be minimized by connecting contiguous blocks of pins on one connector to contiguous blocks of pins on the other connector. Since the MICTOR connectors have rows of pins down both sides of a center channel (even numbered pins on one side, odd numbered pins on the other side), this would require that pins starting on the even side of one connector all terminate on either the even or the odd side of the opposite connector. Figure 12 shows a diagram of the custom cable fabricated. Even numbered pins are shown on the top half and odd numbered pins are shown on the bottom half. The actual connector places pin 1 directly across a center channel from pin 2 with all subsequent even and odd pins directly across from each other. Appendix A contains a complete listing of all pin assignments with their related signal for the ADC board, both breakout boards, and the User Logic 1 FPGA on the MAP board of the SRC-6. Pin assignments on the user logic chips inside the SRC-6 were taken from [18,19].

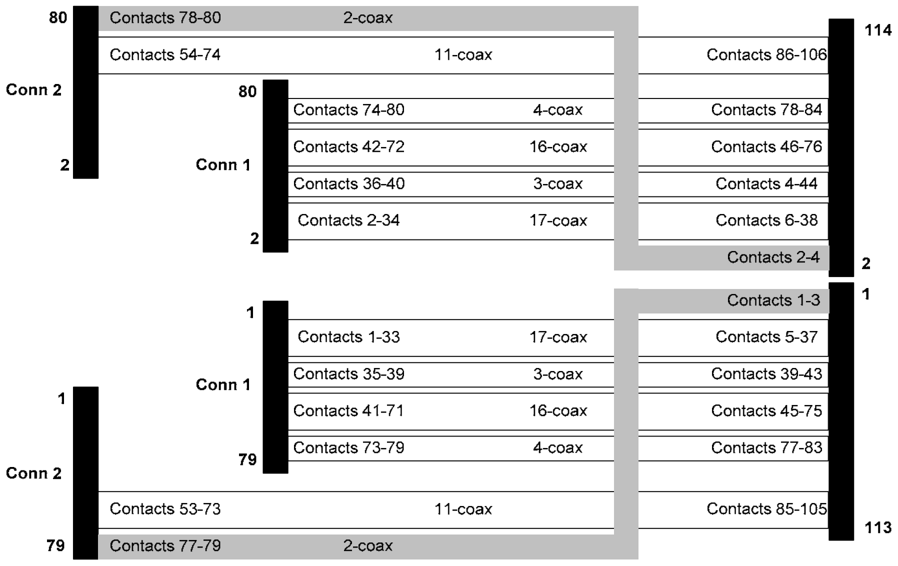


Figure 12. Custom Cable Assembly

Each MICTOR connector is physically attached to a ribbon cable containing micro-coaxial cables by means of a small Printed Circuit Board (PCB). The extremely small size of each individual micro-coaxial cable and the requirement to collectively ground the shield on all of the coaxial cables prohibits direct connections between the connector and the cables. To allow individual pins to be swapped from the even to the odd side, would require a custom PCB to be built that made the necessary pin swaps between the even and odd sides of the connectors. Directly swapping individual micro-coaxial cables would be very difficult. Due to time and expense constraints, the custom PCB option was impractical and not believed to be necessary due to an error interpreting the output of a Xilinx software tool.

Unfortunately, pin swapping was necessary between the even and odd sides of the MICTOR cable connectors. The LVDS pairs are placed on physically adjacent pins on the same side of the ADC MICTOR connectors (see Appendix A for complete pin charts), whereas the LVDS pairs line up on pins directly across from each other on the MICTOR connector for the GPIO port. The solution that met the time and expense restrictions for this project was to use existing breakout boards and coaxial jumpers to make the pin swaps required, aligning each LVDS pair from the ADC to an LVDS pair on the User Logic FPGA within the SRC-6. While this solution proved to be workable at some frequencies, the additional cable length added by a second MICTOR cable and the additional connections added by the breakout boards have restricted the maximum attainable performance and sampling rate for the ADC.

C. LOGICAL INTERFACE

Appendix B contains the full set of schematics for the logical interface portion of the design. The logical interface was coded entirely in the Xilinx schematic capture utility and automatically converted to VHDL by the Xilinx software. Schematic capture was chosen for this portion of the design due to the large number of Xilinx pre-defined components available. The logical interface portion generated just over 1000 lines of VHDL code. The complete VHDL code for this portion of the design is not included but is completely reproducible from the schematics, except for some specific component settings that are noted in the text description of this section.

The physical interface is configured to connect both the channel 1 and channel 2 outputs from the ADC to the SRC-6. However, as stated in Section D of Chapter III, the ADC board is not capable of entering DES mode, which limits the ADC output to two 8-bit samples per clock cycle. This results in a requirement for the logical interface to accept a total of 16 LVDS signal pairs and one LVDS clock input. Logical connections were tested for both channel 1 and channel 2 output from the ADC but channel 2 was chosen for the final design. The lower portion of figure 10 shows the major components of the logical interface.

1. LVDS Input Buffer

The first logical components encountered by the input signals are the differential input buffers. The buffers are instantiated using the Xilinx IBUFDS component. Complete descriptions for the IOB components of the FPGA are contained in [16], along with differential pair assignment specifications between adjacent IOB elements. The IBUFDS components are shared between two adjacent IOB elements. Each is capable of accepting 8 [16] different differential signal specifications, assuming the correct electrical connections are made on the IO bank to which it belongs. DCI is also enabled or disabled for a differential input pair based upon the IBUFDS attributes. Most of the differential signal specifications require some type of termination on the receiver side of a signal path to prevent reflections back along the signal path from an impedance mismatch. Built-in circuitry in the IOB provides this termination when the DCI attribute is active on an input buffer. For each IBUFDS component in the design, the IOSTANDARD attribute was set to LVDS_33_DCI. This attribute sets the input buffer to receive 3.3V LVDS with DCI enabled.

Using the LVDS 2.5V standard would have been preferential based on the output of the ADC. However, each of the IO banks physically occupied by the pins associated with the GPIO ports also contained pins assigned by SRC for onboard communications with other MAP components. These pins were configured for LVTTL, which cannot share an IO bank with 2.5V LVDS. 3.3V LVDS was found to work effectively with low bit error rates up to 340 MHz (which equates to an ADC input clock of 680 MHz). A combination of the input standard mismatch and the problems noted in the physical interface section are most likely to be the factors limiting higher performance.

Each of the input wires connected to the input pins on the IBUFDS components have the LOC attribute set to the identifier of their assigned package pin. Both the LOC and IOSTANDARD attributes can be set in VHDL or in schematic capture mode. In VHDL, the attributes are set using the ‘attribute’ command and in schematic capture they are set by double-clicking the device and adding or modifying the attribute in the attribute list. These two attributes are the only non-visible settings contained within the schematics listed in Appendix B.

After passing through the differential input buffers, the number of signal lines is decreased by one half. The input buffers for the GPIO-in port are physically located in the upper right corner of the User Logic 1 FPGA, with the exception of the clock input pins. Special input pads designed specifically for clock signals are located in the upper and lower center regions of the Xilinx FPGA (Figure 5). The clock input pins contain the global clock buffers and are placed adjacent to blocks containing the DCM elements. Since multiple DCM elements were required for this design, the clock input was placed on a clock input pin instead of in a block with the other input signals. There is only one choice available for a differential clock input from the GPIO-in port, so the actual location was nondiscretionary.

2. Input Registers

After passing through the input buffers, the next logical interface components are the input registers. Figure 13 is the schematic for the input registers for a single bit input.

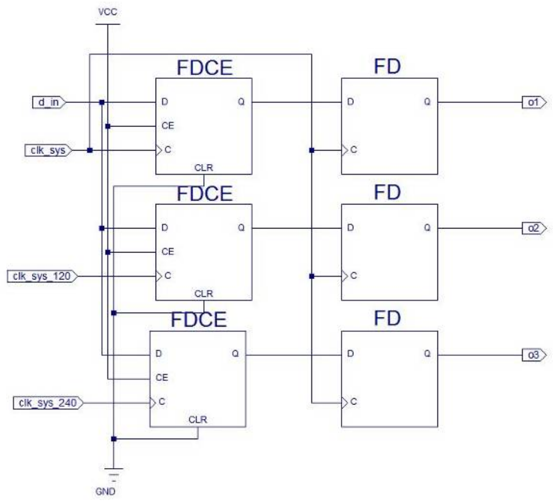


Figure 13. Input Registers

Ideally, the initial input registers for a high speed input signal would be placed in the IOB. Two Double Data Rate (DDR) registers are available in each IOB specifically to clock in high speed signals. These DDR registers are physically located adjacent to the input buffer. There is a very small line delay between the input buffer and the DDR registers, resulting in very good performance at higher clock rates. This presented a design choice for the first stage registers. Either the two DDR input registers contained in the IOB could be used to reduce the 300 MHz input clock to two inputs clocked at 150 MHz each, or the input registers could be moved to the first row of CLB slices adjacent to the IOB and three input registers could be used to down-clock the input directly to three 100 MHz inputs.

The first option of clocking the 300 MHz input down to two 150 MHz signals is incompatible with the rest of the design. This would produce two 150 MHz signals that would have to be down-clocked to 100 MHz. This proves to be inconvenient and produces an uneven number of outputs on every clock (i.e. 2 outputs first clock, 4 outputs next clock). Placing two of the three input registers of the second option in the IOB, while leaving the third in the first row of CLB slices, is inadvisable due to the different line delays between the input buffer and the registers. The different line delays make clock phase alignment with the data valid windows of the input signals much more problematic and the margins are fairly low with a 300 MHz input signal. Figure 14 shows a timing chart for the design optimized for 300 MHz (600 MSps on the ADC).

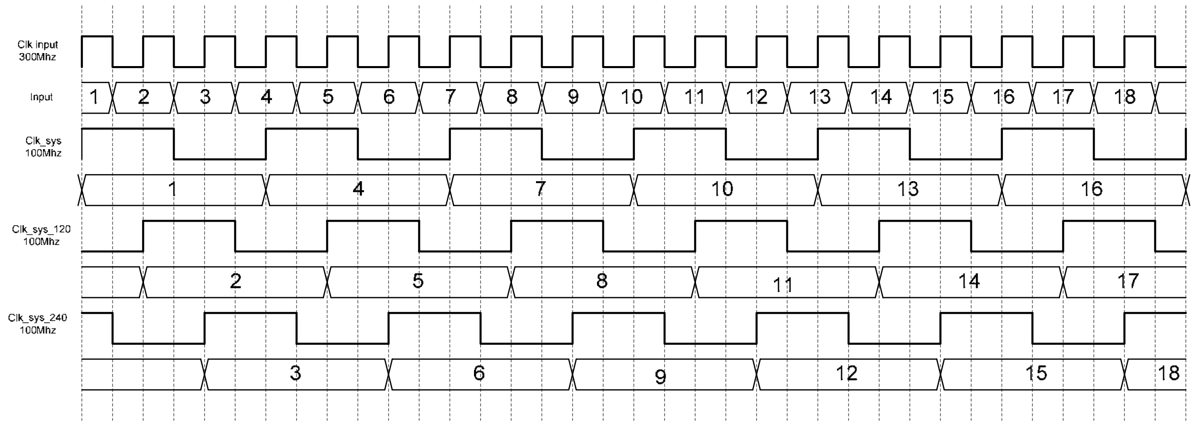


Figure 14. Timing Diagram for 300 MHz Input

The clock inputs to the three initial input registers are listed on the left column of Figure 14. The input clock period is approximately 3.3ns at 300 MHz and the resulting data valid window slightly is shorter. The input data is valid for approximately 1.5ns before and after the input clock pulse transitions from low to high. From [16], the setup time for the registers in a slice is approximately 370ps and the hold time is approximately 90ps. The worst case clock skew between the input registers assigned to channel 2 (16 input bits, for 48 total registers) is approximately 500ps. Clock jitter produced by the DCM elements described in the next section is approximately 450ps. Taken together, this produces a required time block of 1.41ns that must be centered in a data valid window slightly less than 3.3ns wide. There is an effective margin of error of approximately 800ps for line delays caused by routing on the FPGA. Clock delay from the DCM and an average value for line delay from the IOB elements to the input registers located in the first CLB row is removed in the clock generator and will be explained in detail in the following section.

The second column of registers in Figure 13 is not strictly necessary, but they serve to immediately align all three input bits to the clock domain used by the FIFO input. This ensures that any routing delays introduced during the PAR of the FPGA compilation process do not cause timing errors between the input registers and the clock synchronization stage.

The output of the individual bit input registers are combined into a single bus containing three 8-bit words of data from each of the two 8-bit inputs from the ADC channel. The two resulting 24-bit wide data buses are combined into a single 48-bit wide bus for transfer to the clock synchronization stage.

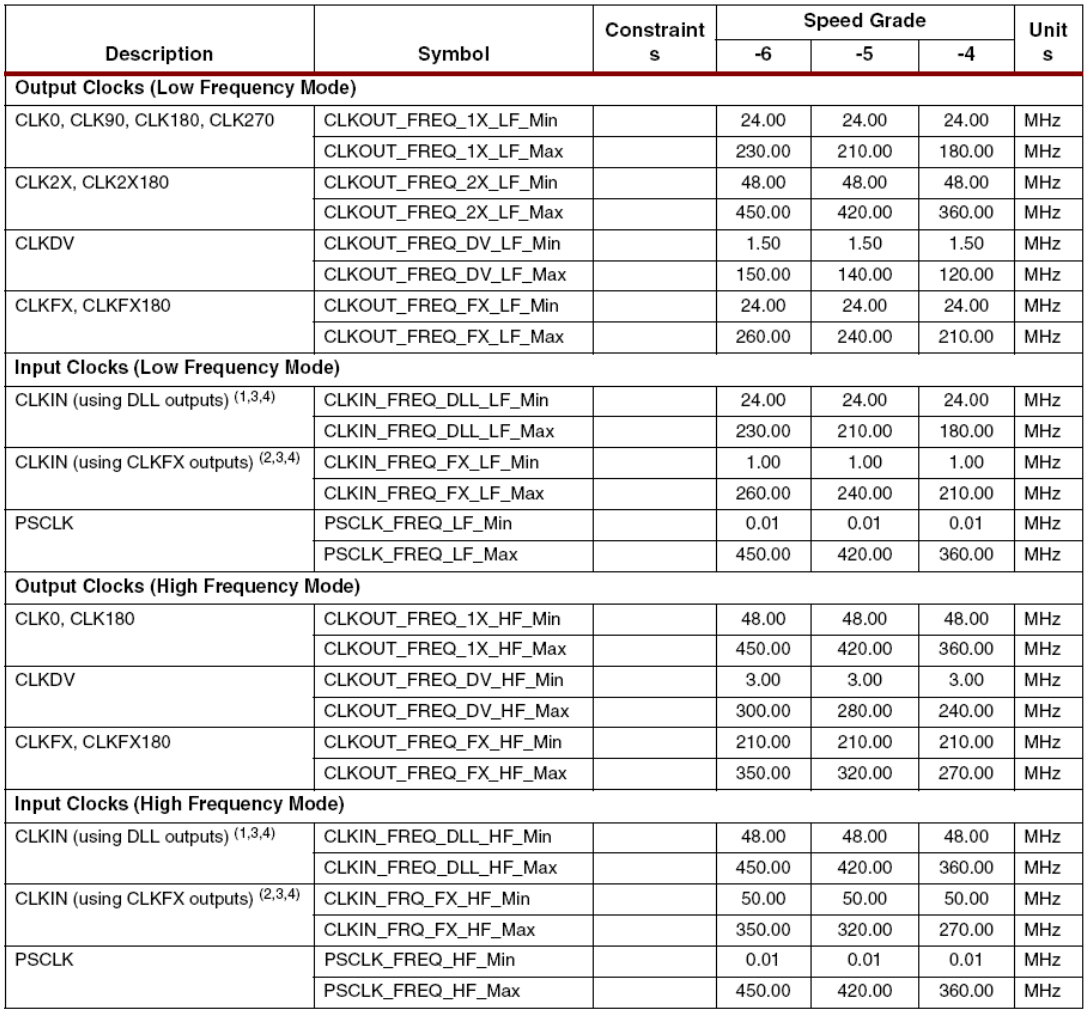
The current design is non-optimal and can be improved by individually setting the LOC attribute for each of the initial input registers to manually assign them a specific physical location on the FPGA. This was not done during initial testing since no timing errors attributable to line delay between the IBUFDS and the input registers were observed. However, in later designs with much higher chip utilizations, timing errors attributable to the input registers not being placed by the PAR process consistent distances (in terms of line delay) from the input buffers have been observed. This has not occurred on every design and can largely be compensated for by modification of the clock phase shift in the DCM elements. The Bit Error Rate (BER) for low utilization designs has been observed to be on the order of 10^{-4} (approximately one error every 10,000 samples), while the BER for higher utilization designs is on the order of 10^{-3}. Due to time constraints, adding individual LOC attributes to 96 separate registers was not considered critical while the BER was still reasonable.

Appendix B also contains the schematic and timing diagram for an input register design optimized for 400 MHz (800 MSps on the ADC). As previously stated, input frequencies over 340 MHz proved unreliable, so the 400 MHz design was not used for the final design. The logic was tested and performed better than the 300 MHz design at lower clock frequencies. Since a 400 MHz design is able to use the two DDR input registers located in the IOB (defined by setting the IOB=TRUE attribute on any register type), the timing margins for the individual components are actually better than the 300 MHz design. A design using the IOB DDR registers should be capable of much higher input speeds than can be tested with the physical interface used for this work. A design for an LVDS receiver operating at 644 MHz is detailed in [6] and speeds up to 720 MHz are theoretically possible (720 MHz is the input limit for the DCM modules in the model of FPGA used in the older SRC-6 systems), assuming a highly optimized design. The XC2VP100 FPGA installed in newer SRC systems is capable of much higher input speeds [17] and contains some additional input circuitry that allows Gigabit range IO transfers.

3. Clock Generation

For this design it was necessary to generate three, 100 MHz clock signals, each phase shifted 120 degrees apart. This allows each of the three input registers running on the phase shifted 100 MHz clocks to sample the 300 MHz input signal and fully recover the data, as shown in Figure 14. This portion of the design used the dedicated logic designed into the Virtex-II series of FPGA specifically for clock modifications. The Virtex-II FPGA in the NPS SRC-6 has a total of 12 Digital Clock Manager (DCM) units located along the top and bottom edges of the chip. These specialized logic devices contain the circuitry to phase shift a clock signal from zero to 360 degrees. They are also capable of multiplying a clock signal, or dividing a clock signal, within the restrictions listed in [16].

For this design, the incoming clock signal from the ADC was routed into a DCM module with the CLKIN_DIVIDE_BY_2 attribute set to true. When set to true, this attribute causes incoming clock signal to be divided to half rate before input to the DCM. With this attribute set, the FPGA model in the NPS SRC-6 is capable of receiving up to 720 MHz incoming clock signals. However, due to the non-optimal signal path from the ADC, incoming clock signals over 340 MHz were unsatisfactory for DCM input. Table 2 lists the complete timing specifications for the DCM modules. The FPGA speed grade in the NPS SRC-6 is the -4 model. The initial DCM was set to the high frequency mode of operation and subsequent DCM modules were set to low frequency modes of operation. The Virtex-II datasheet specifies that when the CLKIN_DIVIDE_BY_2 attribute is set, the frequencies listed in Table 2 for CLKIN can be doubled.



Description	Symbol	Constraints	Speed Grade			Units
			-6	-5	-4	
Output Clocks (Low Frequency Mode)						
CLK0, CLK90, CLK180, CLK270	CLKOUT_FREQ_1X_LF_Min		24.00	24.00	24.00	MHz
	CLKOUT_FREQ_1X_LF_Max		230.00	210.00	180.00	MHz
CLK2X, CLK2X180	CLKOUT_FREQ_2X_LF_Min		48.00	48.00	48.00	MHz
	CLKOUT_FREQ_2X_LF_Max		450.00	420.00	360.00	MHz
CLKDV	CLKOUT_FREQ_DV_LF_Min		1.50	1.50	1.50	MHz
	CLKOUT_FREQ_DV_LF_Max		150.00	140.00	120.00	MHz
CLKFX, CLKFX180	CLKOUT_FREQ_FX_LF_Min		24.00	24.00	24.00	MHz
	CLKOUT_FREQ_FX_LF_Max		260.00	240.00	210.00	MHz
Input Clocks (Low Frequency Mode)						
CLKIN (using DLL outputs) (1,3,4)	CLKIN_FREQ_DLL_LF_Min		24.00	24.00	24.00	MHz
	CLKIN_FREQ_DLL_LF_Max		230.00	210.00	180.00	MHz
CLKIN (using CLKFX outputs) (2,3,4)	CLKIN_FREQ_FX_LF_Min		1.00	1.00	1.00	MHz
	CLKIN_FREQ_FX_LF_Max		260.00	240.00	210.00	MHz
PSCLK	PSCLK_FREQ_LF_Min		0.01	0.01	0.01	MHz
	PSCLK_FREQ_LF_Max		450.00	420.00	360.00	MHz
Output Clocks (High Frequency Mode)						
CLK0, CLK180	CLKOUT_FREQ_1X_HF_Min		48.00	48.00	48.00	MHz
	CLKOUT_FREQ_1X_HF_Max		450.00	420.00	360.00	MHz
CLKDV	CLKOUT_FREQ_DV_HF_Min		3.00	3.00	3.00	MHz
	CLKOUT_FREQ_DV_HF_Max		300.00	280.00	240.00	MHz
CLKFX, CLKFX180	CLKOUT_FREQ_FX_HF_Min		210.00	210.00	210.00	MHz
	CLKOUT_FREQ_FX_HF_Max		350.00	320.00	270.00	MHz
Input Clocks (High Frequency Mode)						
CLKIN (using DLL outputs) (1,3,4)	CLKIN_FREQ_DLL_HF_Min		48.00	48.00	48.00	MHz
	CLKIN_FREQ_DLL_HF_Max		450.00	420.00	360.00	MHz
CLKIN (using CLKFX outputs) (2,3,4)	CLKIN_FRQ_FX_HF_Min		50.00	50.00	50.00	MHz
	CLKIN_FRQ_FX_HF_Max		350.00	320.00	270.00	MHz
PSCLK	PSCLK_FREQ_HF_Min		0.01	0.01	0.01	MHz
	PSCLK_FREQ_HF_Max		450.00	420.00	360.00	MHz

Table 2. DCM Timing Parameters (From [16])

It is within specifications for the initial DCM to operate in the high frequency mode of operation without the CLKIN_DIVIDE_BY_2 attribute set and still receive the 300 MHz signal used for this work. However, this would require the clock signal to be processed through an additional clock buffer element and incur more delay in the signal that would require compensation. The design trade-off chosen is that the clock divide on the initial DCM is set to 1.5 to process the incoming 300 MHz (150 MHz seen by the DCM) down to the required 100 MHz (nominal clock speed of the FPGA). The choice incurs an additional 150ps of jitter on the clock out of the first DCM for a total of 300ps of jitter on the first DCM. Applying the full 300 MHz clock and dividing by 3 would reduce the total jitter out of the first DCM to 150ps. It was determined experimentally that the actual clock signal received from the ADC was more stable when the CLKIN_DIVIDE_BY_2 was set, so the choice was made to accept the extra 150ps of clock jitter. Future designs with access to a lower noise input clock could benefit from configuring the DCM modules to divide by an integer value, thus reducing clock jitter by 150ps.

The clock output of the initial DCM is set to 100 MHz and phase locked to the input clock. The initial output clock is used to drive one of the 3 input registers on each input bit and to drive the FIFO circuitry for synchronization to the normal clock domain of the FPGA. It is also used as the input clock to the two second stage DCM modules. Both second stage modules are configured for low frequency operation and each produces an output clock phase shifted either plus or minus 120 degrees from the input clock. These clock outputs are only used to drive the other two input registers on each input bit. The second stage DCM modules also introduce an additional 150ps of clock jitter. This means that one of the three registers suffers from a maximum of 300ps of clock jitter while the other two suffer from up to 450ps of clock jitter. Complete schematics are available in Appendix B.

To compensate for line delay incurred by the clock signals as they travel from the top center region of the FPGA to the top right corner of the FPGA, an additional phase shift is entered into the initial DCM. The average delay between the DCM modules and the input registers is approximately 2.5ns. Since the clock period of the 300 MHz input signal is 3.3ns, some compensation must occur. This line delay has the effect of shifting the clock pulse approximately 800ps within the data valid window. As stated in the previous section, the error margin for line delay from the input buffer to the initial input registers is also approximately 800ps.

The phase shift attribute value for the DCM is a number between zero and 255, which corresponds to phase shift values between zero and 360 degrees. A phase shift on the initial DCM of between 15 and 60 was found to be satisfactory to compensate for the line delay. Values of 35 or 40 were used for most testing in this work. A calculated value of 64 should be optimal but the lower values produced slightly better performance.

The clock generation design has one major non-optimal element. The coding in VHDL of the phase shift for the initial DCM is unnecessary and fails to provide an optimal solution under all operating conditions. To correct this, future designs could make use of the additional circuitry in the DCM modules specifically designed for dynamically phase shifting the clock. Complete instructions and sample code are available from Xilinx in [10] for actively aligning clock signals using the DCM modules. Time constraints prevented the realization of a complete active clock phase alignment circuit in this work, so the expedient of coding a constant phase shift into the VHDL was substituted. This is acceptable for a laboratory environment where component temperatures are stable and predictable but might be unsatisfactory for any application where the components would undergo temperature variations, as this would cause the line delay values to fluctuate. In addition, an active phase alignment system would be more accurate than coding a phase shift manually.

4. Clock Synchronization

To process data coming into the SRC from an external source, it is necessary at some point to synchronize the externally generated data stream with the internal clock of the FPGA. As noted in Chapter II, some of the previous work at NPS involved collecting data from an external ADC. The previous work partially avoided the issue of clock synchronization by sending the SRC-6 internal clock to the external ADC. This proved a workable solution but limited the maximum achievable sampling rate of the ADC to one reachable by the SRC-6. Table 2 gives the maximum clock output value of the DCM as 270 MHz on the CLKFX (clock multiplier) output pin. Also, in the previous work [11,12], the data valid signal from the ADC was used as an input to a DCM to ensure that data would be properly clocked into the SRC-6. While the SRC-6 provides a mechanism to send the FPGA clock signal to any user macro, any modification of the clock signal (as in [11,12]), or any external clock signal used by a user macro, is only used by the logic to which it is assigned in the user macro and cannot be used by the C code portion of the program. It is the responsibility of the user macro programmer to resynchronize the logic with the clock used by the SRC code before returning the data to the C code portion of the program. Since the C code portion of the program is mandatory for communication with the microprocessor, this must be done for any program that will return data to the microprocessor. The previous work maintained fairly low data rates without returning the data to the SRC-6 clock domain only because it used the SRC clock to drive the hardware external to the SRC-6. This solution is impractical for achieving the higher sampling rates possible by using an externally generated clock.

The simplest solution to the synchronization issue is to instantiate a First-In First-Out (FIFO) buffer in the user logic macro that writes data into the buffer using the macro or external clock and reads data out of the buffer using the system clock passed to the user macro by the C code portion of the program. There are other methods that can be used to achieve the necessary clock synchronization, such as the asynchronous method described in [9], but the FIFO method is both simple and reliable, using the Block SelectRam (BRAM) logic elements built into the Virtex-II FPGA.

The BRAM units can be instantiated in a variety of ways with many different options but, for this work, they were instantiated using the Xilinx component for a dual-ported, 16-bit wide word, 256 location RAM block, with each memory location addressed by an 8-bit address line. Three of these units are combined to input and output the 48-bit wide data bus from the input registers. The 16-bit wide word was chosen to prevent the data lines used by the BRAM blocks from interfering with the multipliers with which they share data lines. Also, increasing the width of the word stored in a BRAM unit decreases the number of memory locations available. A FIFO depth of 256 has proved to be reliable with this design and only consumes three BRAM blocks.

The dual ported design of the BRAM units allows data to be written into port A, running off the external clock, while other data is simultaneously being read from port B, which is running off the clock signal passed from the C code portion of the program. The data read from port B is now synchronized with the internal clock of the SRC and is ready for further processing within the user macro, or can be passed directly back to the C code that called the user macro. In this work, FIFO placement in the design could be anywhere before the data stream is returned to the C code portion. However, since the QMF design relies on very tight timing windows for correct operation, it makes sense to place the FIFO directly after the input is received and before further processing occurs. Should the FIFO be placed after the QMF, the QMF would be required to function using a modified version of the externally generated clock, which is undesirable.

a. Addressing Circuit

The addressing sub-circuit in the FIFO is used to create the memory addresses used in both the port A writes and the port B reads. It accomplishes this using the very simple expedient of two 8 bit counters that continuously cycle sequentially through all 256 possible address locations with each counter clock being driven by the clock appropriate to the operation. If both clocks are running at exactly 100 MHz, the read operation of port B would always occur on the memory location that was just written by port A. In fact, it is very improbable that both clocks would ever be running at exactly the same speed. One clock will always be slightly faster than the other which will cause it to cycle faster than the other clock, eventually causing either a buffer underflow or an overflow, depending on which clock is faster.

It is possible to build FIFO systems that are immune to buffer underflow or overflow but these systems require a mechanism to halt the operation of the writing system when the buffer is full, or to halt the operation of the reading system when the buffer is empty. Since this is a real-time data capture and processing system, halting the data capture or the processing is undesirable. For this purpose the addressing circuit is designed in such a way that upon system initialization, the write buffer starts at address zero and writes continuously. The read addressing circuit remains at zero until the most significant bit of the write address first changes from zero to one. At this point, the read address counter begins counting. This introduces a gap of 128 memory locations between the read and write address counters.

Since one clock will, inevitably, be slightly faster than the other, an underflow or overflow situation will eventually occur. A 1% variation in clock speeds is calculated to produce an underflow or overflow approximately every 12,800 cycles, or every 128ms. While this seems excessive, actual clock variations have been observed to be much lower, with an observed underflow or overflow occurring approximately every 100,000-200,000 clock cycles, or every one to two seconds. This error rate would most likely still be too large for an operational system, but can be further reduced simply by enlarging the FIFO size, at the expense of additional BRAM units. When an underflow or overflow does occur, data stream read out of the FIFO either jumps forward in time 256 samples, or falls back in time 256 samples. While this would be problematic for larger filter designs, the small demonstration filters in this design are not significantly affected by the discontinuity.

The output of the read address counter is also passed out of the FIFO for reuse in timing some of the pipeline elements of the QMF design. There is a requirement within the QMF design for multiplexer elements to switch between two input busses every 1, 2, and 4 clock cycles. The three least significant bits of the read address counter are used to switch the multiplexer elements.

V. QUADRATURE MIRROR FILTER BANK IMPLEMENTATION

A. OVERVIEW

Chapter II introduced a Quadrature Mirror Filter (QMF) as a form of wavelet decomposition filter made up primarily of a tree structure containing high pass filters, low pass filters, and down samplers. This chapter will introduce some options for construction of a QMF on the SRC-6 written in both C code and implemented entirely in VHDL. A fully working demonstration model was completed in VHDL, with a working FIR filter (the main element of a QMF) completed in C, for comparison to the VHDL FIR filter elements contained in the full VHDL based QMF. The following two sections detail the design requirements and some of the discretionary options that were chosen to fulfill the design requirements.

1. Design Requirements

The first main requirement of the design is that it be real-time. For purposes of this work, real-time is defined as a code block being able to accept input values on each clock cycle equal in size to those values delivered from the hardware interface on each clock. The design must be able to complete processing of the inputs in a fashion that returns output values at a rate determined by the size of the QMF. Specifically, for a three stage QMF, six simultaneous 8-bit input values from the hardware interface must produce 48 8-bit output values, once every eight clock cycles. This may be accomplished by a pipeline architecture, with pipeline depth not limited by the requirement.

The next main requirement is that the QMF is as large as practical. Since the input size is fixed by the data flow from the hardware interface, the size of the filter is determined by the number of taps in each FIR filter and the total number of filter stages. Taken further, this requirement means that each filter design uses the absolute minimal hardware so that more total filters can be instantiated.

The demonstration filter design must also be extensible into a practical design. This means that individual elements should be constructed in such a fashion that the majority of the components could be re-used in a practical filter design. A practical filter design is defined to mean a filter with 30 or more filter taps in each high pass and low pass filter element and contain at least four total QMF stages. For this to be feasible, the filter design must take into account that one user logic FPGA will not be large enough for a practical design. Thus, the design should be easily extensible across multiple user logic chips and across multiple SRC-6 MAP boards.

The design must maintain as much numerical precision as possible. As stated in Chapter II, it is intended for the data output of the QMF to be passed on to another stage in a larger design for processing. Thus, it is desirable that any precision loss through the arithmetic operations and number format changes in the QMF be less than or equal to the quantization error that will result when the calculated output values are returned to an 8-bit format.

While not a requirement, it is assumed for the purpose of this work, that all high pass and low pass FIR filters used in the design will be identical to each other with identical constant coefficients used in all filters of the same type.

2. Design Element Options

Assuming that the construction of the high pass and low pass filters are identical, with only the coefficients being different, there are only two main components of the QMF. They are the filter element, configured as high pass or low pass by the coefficients, and the down sampler. The only purpose of the down sampler is to discard every second sample. Through careful design, the down sampler can be removed entirely by the simple expedient of not calculating the values that would be discarded by the down sampler. This leaves the filter element as the only required element and reduces by one half the total number of multiplications that must occur in each filter.

The filter element is made up of some form and number of multipliers, and some form and number of adders that sum the results of the multipliers. In addition, since the full operation cannot be completed in one clock cycle, some type of register or delay element must be used in a practical design, resulting in a pipelined data flow through the filter.

a. Multiplier Instantiation

There are four main methods of instantiating a multiplier in the Xilinx FPGA used by the SRC-6. The first method is to simply multiply a binary number by any power of two by right or left shifting the bits in the number. Actually, building a multiplier of this type is trivial and requires effectively no actual hardware be instantiated. This type of multiplier will be discussed in Chapter VI but will not be used in the demonstration design. The next type of multiplier can be instantiated in the CLB matrix. An array type binary multiplier can be constructed that only utilizes CLB resources. However it is not possible to build a multiplier of this type that can execute large multiplies in a single clock cycle. The array-type multiplier will be discussed in Chapter VI but was not considered for the demonstration design.

Another interesting method for multiplier instantiation is to simply use a look up table based in either BRAM modules configured as ROM units, or to directly instantiate ROM units in the CLB matrix. There are some definite limitations to this method of multiplication, but it is potentially a useful method in cases such as a QMF design where all multiplications use constant coefficient values. Due to some scaling issues, ROM based look up tables are not appropriate as the primary multiplier type for the demonstration QMF design, but their utility is discussed in Chapter VI. A future design using ROM based look up tables could easily be used for the first stage of the QMF, where the 8-bit values from the ADC are used to address two BRAM modules producing a 32-bit product. Since the BRAM modules are dual ported, two BRAM modules could be used to provide two parallel results from two 8-bit inputs using separate ports.

The fourth type of multiplier exists as a dedicated hardware device within the Virtex-II. The specialized multiplier circuit in the Virtex-II is capable of one clock multiplies of two 18-bit binary values, returning a 36-bit binary value, in a single clock cycle. This hardware multiplier was chosen as the multiplier element for the demonstration design. The use of mixed multiplier types will be discussed in Chapter VI but was not used for the demonstration QMF design.

b. Bit Truncation

Another choice made in the design was to utilize bit truncation instead of rounding whenever the bit length of a number had to be reduced. Truncation is generally undesirable since truncation of signed binary numbers effectively causes signal attenuation much greater than a rounding operation. When a positive signed binary number has a portion of its least significant bits removed during truncation, it can only become less positive. Inversely, negative signed binary numbers can only become more positive during truncation. Truncation was chosen for the demonstration design because implementation is simpler than a binary rounding system and can be accomplished with wired connections that do not include combinational logic. A rounding system would most likely (depending on complexity and accuracy) require an extra pipeline stage to be added to the system. Also, the VHDL compiler will remove any logic whose output values are not referenced by other logic. This allows standard components to be used in all locations by the designer but still allows a savings in hardware utilization when the excess hardware elements are removed when the design is compiled.

c. Filter Coefficient Selection

A Six tap FIR filter design was chosen. The filter coefficients were chosen using the FIR function in MATLAB. An odd number of filter coefficients are required to produce an even order high pass filter, which is required for a symmetric FIR design of this type. As a result, the filter coefficients were chosen in such a manner that the least significant filter coefficient was of the same order as the precision limit of the system. By omitting the final filter coefficient, filter performance is degraded but the effect is limited because of the small magnitude value of the omitted coefficient. Table 3 shows the listing of the chosen coefficients.

Coefficient #	1	2	3	4	5	6	7
High Pass	-0.0013	-0.0052	-0.0128	0.982	-0.0128	-0.0052	-0.0013
Low Pass	0.0287	0.143	0.3282	0.3282	0.143	0.0287	

Table 3. Filter Coefficients

Figure 15 shows the normalized frequency response of the high pass filter with the chosen coefficients and Figure 16 shows the normalized frequency response of the low pass filter. At 600MSps, both filters were designed to have a cutoff frequency of approximately 5 GHz. The figures demonstrate the poor performance of these filters with only six taps available. Performance is particularly bad for the high pass filter. A practical filter design would require a significant increase in the total number of filter taps to achieve acceptable performance.

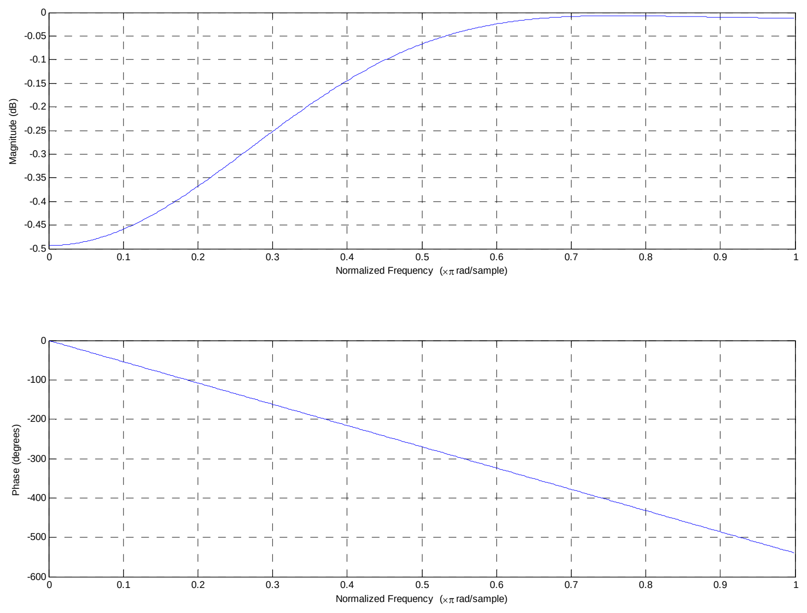


Figure 15. High Pass Filter Frequency Response

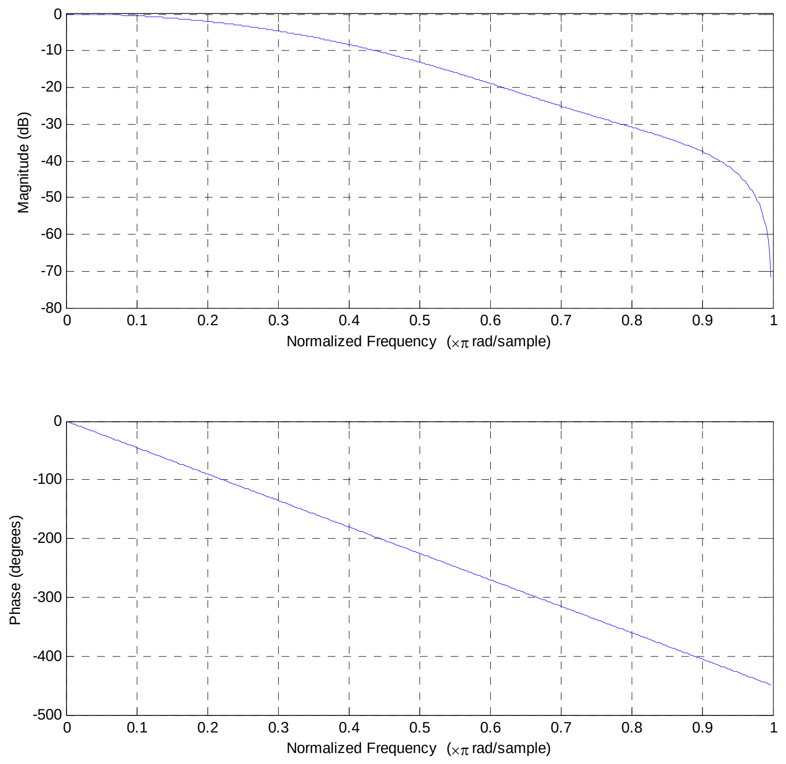


Figure 16. Low Pass Filter Frequency Response

B. DATA INTERLEAVING TO REDUCE HARDWARE REQUIREMENTS

A QMF tree is shown in Figure 17 with the outputs of each stage numbered.

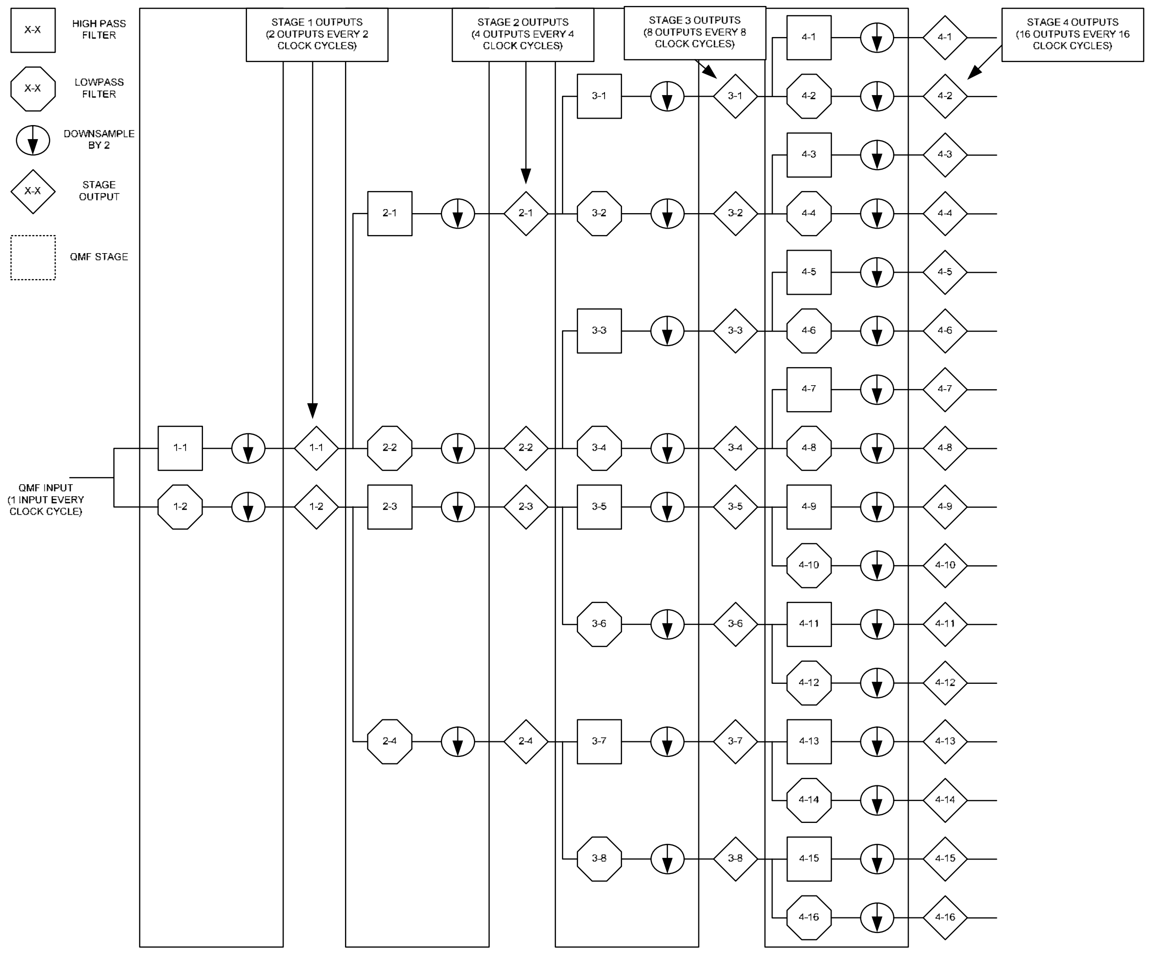

X-X HIGH PASS FILTER
X-X LOWPASS FILTER
DOWNSAMPLE BY 2
X-X STAGE OUTPUT
QMF STAGE
STAGE 1 OUTPUTS (2 OUTPUTS EVERY 2 CLOCK CYCLES)
STAGE 2 OUTPUTS (4 OUTPUTS EVERY 4 CLOCK CYCLES)
STAGE 3 OUTPUTS (8 OUTPUTS EVERY 8 CLOCK CYCLES)
STAGE 4 OUTPUTS (16 OUTPUTS EVERY 16 CLOCK CYCLES)
QMF INPUT (1 INPUT EVERY CLOCK CYCLE)

Figure 17. Annotated QMF Tree

As shown in Figure 17, each filter stage produces 2^S outputs every 2^S clock cycles, where S is the stage number. Designing a QMF identical to Figure 17 for hardware implementation is possible but very impractical. Every stage after stage 1 would have to be halted for every clock cycle where the down sampled output from the previous stage left a gap. Also, enough hardware elements would need to be committed to the design for each stage to allow simultaneous processing on 2^S inputs. This would be a very inefficient design in terms of hardware, both because of the amount of hardware necessary for simultaneous processing, and for the complicated control circuitry required to halt and restart a pipelined architecture. These issues can be mitigated by hardware reuse through data interleaving. Through data interleaving the 2^S filters (both high pass and low pass) in each stage can be reduced to a single filter of each type that operates continuously. This also removes the requirement for control circuitry to halt and restart the data path due to the gaps caused by down sampling.

1. Single Input Data Interleaving

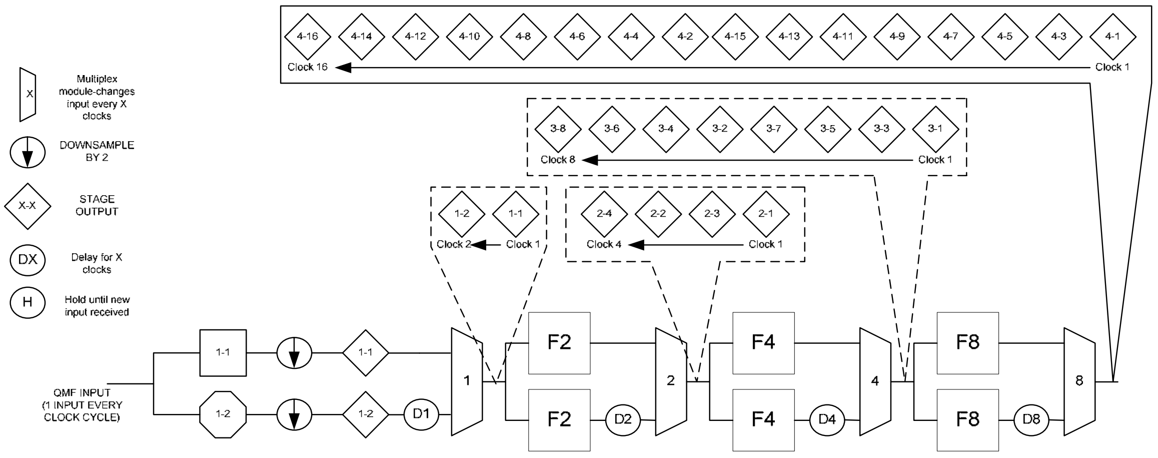


Figure 18. Single Input Data Interleaving QMF Design

To fully illustrate the data interleaving concept of design for a QMF filter, Figure 18 shows a single input QMF block diagram. The stage outputs, as numbered in Figure 16, are listed in Figure 18 in the order that they appear on the output multiplexer from the previous stage. In this representation, there is a single input value entering the QMF on every clock. The first stage operates by passing the data stream through a high pass and low pass FIR filter, then down sampling the resulting data streams by removing every other value, producing outputs 1-1 and 1-2 on every second clock cycle. The delay element and multiplexer return the data stream to one value on every clock cycle. With the outputs of filter 1-1 and 1-2 alternating, as shown in the figure, by switching the selected input for the multiplexer on every clock cycle, a single data path is produced in which two separate data streams are interleaved.

The second stage of the QMF then receives the interleaved data stream and passes it to two identical filter elements, one a high pass filter, and the other a low pass filter. We will ignore the inner workings of the filter blocks labeled F2, F4, and F8. They are pipelined filter elements, designed to handle the interleaved data stream, and produce a continuous flow of filtered outputs with the same interleaved data stream as their input. Down sampling is not required in these filter elements, as every other value from each previous stage filter output will not be selected by the multiplexer to move to the next stage. The multiplexers in Figure 18 switch inputs every N clock cycles, where N is the number labeled on the multiplexer.

The resulting outputs for each stage are labeled with the final number of outputs being equal to 2^S samples every 2^S clock cycles. This format produces an equivalent design to Figure 17 but with a substantial reduction in hardware requirements since only two filter elements are required for each stage.

2. Parallel Input Data Interleaving

Figure 18 shows a practical design strategy, although for simplicity it omits some details of filter construction. However, it is only designed to handle a single input on each clock, and it still produces and discards the values that are down sampled. Since most ADC units are designed to operate at higher sampling rates than can be handled by FPGAs. ADC units are usually designed to allow for parallel data outputs at lower clock rates. Any practical QMF design capable of handing high ADC sampling rates must be able to handle multiple parallel inputs on every clock cycle. Also, since every second value in a particular data stream (there may be multiple streams occupying a single data path) leaving a filter is discarded, additional hardware savings can be generated by not calculating those values. For a FIR filter, this can be accomplished by only multiplying odd numbered inputs with the odd numbered filter coefficients and even number inputs with even numbered filter coefficients.

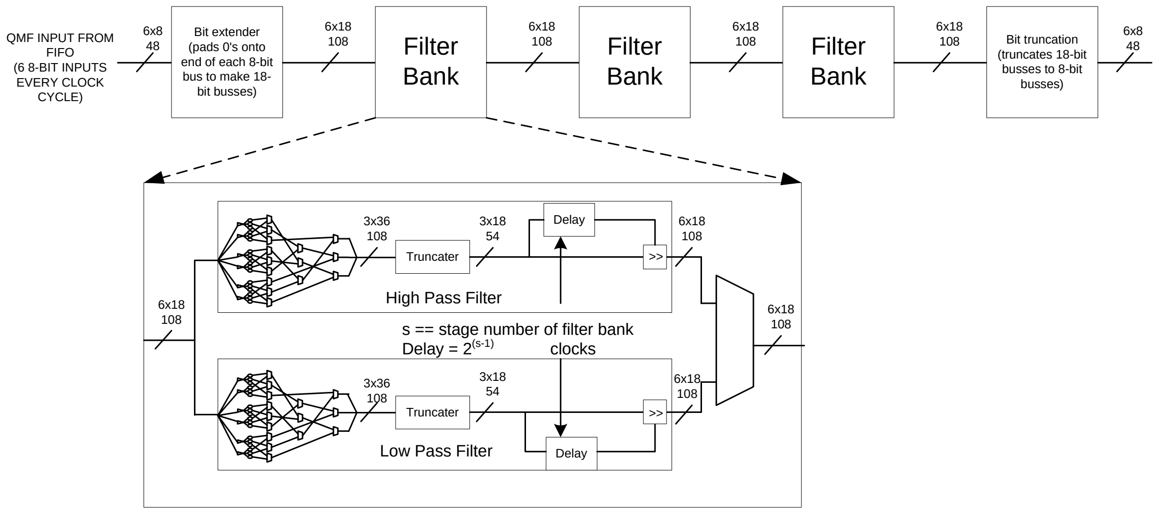

QMF INPUT FROM FIFO (6 8-BIT INPUTS EVERY CLOCK CYCLE)
6x8 48
Bit extender (pads 0's onto end of each 8-bit bus to make 18-bit busses)
6x18 108
Filter Bank
6x18 108
Filter Bank
6x18 108
Filter Bank
6x18 108
Bit truncation (truncates 18-bit busses to 8-bit busses)
6x8 48
6x18 108
3x36 108
Truncater
3x18 54
Delay
>>
6x18 108
High Pass Filter
s == stage number of filter bank
Delay = $2^{(s-1)}$ clocks
6x18 108
3x36 108
Truncater
3x18 54
>>
6x18 108
Low Pass Filter
Delay

Figure 19. 6 Parallel Input QMF

Figure 19 shows the block diagram of the actual demonstration QMF as it was implemented. The design accepts a 48-bit bus containing 6 8-bit values on every clock cycle directly from the FIFO stage of the hardware interface. Each input is padded with trailing zeros to generate 6 18-bit values, which is the common input for all the filter bank stages. All filter bank stages are identical hardware instantiations which take as inputs, values representing the appropriate element delays for their stage in the QMF. After passing through the three stages of the demonstration QMF, the data outputs are truncated back to 6 8-bit values. Details of the filter bank implementation and pipeline timing are explained in the next section.

C. IMPLEMENTATION IN VHDL

All VHDL code for the complete implementation described in this section is contained in Appendix C.

The first consideration for an implementation of a QMF in hardware is the number system and number format that will be used. Next, an actual filter design must be created that produces the required number of outputs for the required number of inputs. Components for the filter must then be designed and constructed. Finally, for a fully pipelined design capable of correctly processing interleaved data streams, a timing structure and control must be implemented to ensure that individual values in each interleaved data stream are correctly calculated in the correct order and calculations are made with values that are members of the same data stream.

1. Number System

The number system chosen for this design is a variable length fixed point numbering scheme using signed binary arithmetic operations on signed binary values. Figure 20 shows the bit position values and decimal point placement for the different number formats used in this design. The current design accounts for overflow conditions in the multiplication operations but not in the addition operations. All addition operations occur using the format given in Figure 20 for the multiplier output, two sign bits, followed by seven integer bits, followed by 27 fractional bits.

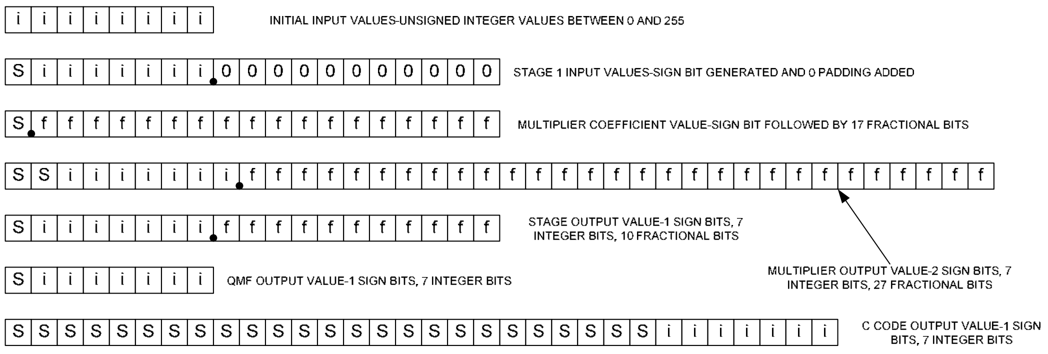

INITIAL INPUT VALUES-UNSIGNED INTEGER VALUES BETWEEN 0 AND 255
STAGE 1 INPUT VALUES-SIGN BIT GENERATED AND 0 PADDING ADDED
MULTIPLIER COEFFICIENT VALUE-SIGN BIT FOLLOWED BY 17 FRACTIONAL BITS
STAGE OUTPUT VALUE-1 SIGN BITS, 7 INTEGER BITS, 10 FRACTIONAL BITS
MULTIPLIER OUTPUT VALUE-2 SIGN BITS, 7 INTEGER BITS, 27 FRACTIONAL BITS
QMF OUTPUT VALUE-1 SIGN BITS, 7 INTEGER BITS
C CODE OUTPUT VALUE-1 SIGN BITS, 7 INTEGER BITS

Figure 20. Binary Number System

The initial sign bit in the stage one input is generated by inverting the most significant bit of the ADC input value. This creates a signed binary value between -128 and 127. Trailing zeros are then added before the value is passed to the stage 1 filter bank so that all filter banks have the same input format. The use of look-up table based multiplication for the first stage of a large design would not require the zero padding and would exist as a unique stage design. The hardware multipliers in the FPGA produce the format given by the multiplier output format in Figure 20, which is also used by the adder elements.

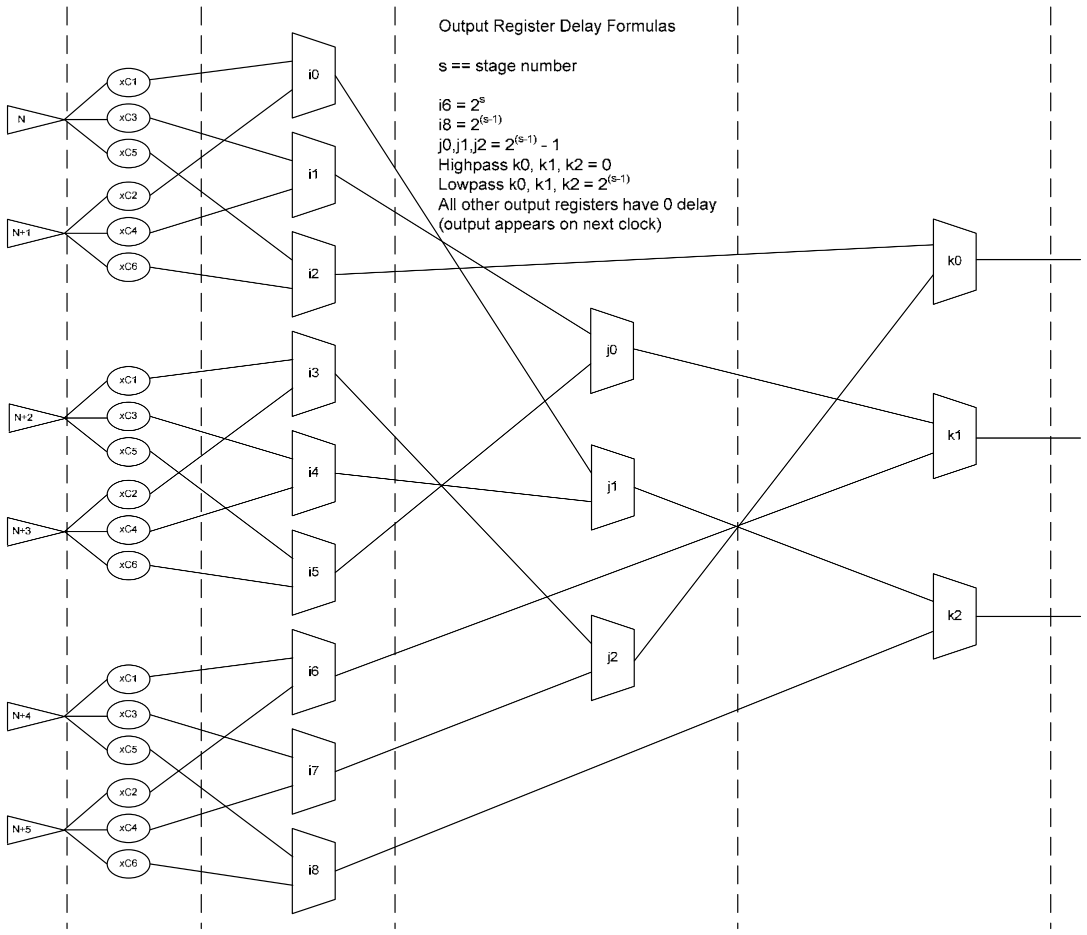
The number system and the adder hardware can ignore the possibility of overflow during addition operations because of coefficient scaling in the multiplication operations. This assumption is true for most cases in FIR filter design where the sum of the coefficients in the filter is less than one. As long as the filter coefficients are chosen so that when all the filter inputs are -128 or 127 simultaneously (minimum and maximum ADC output values), the sum of all the multiplications must add to less than or equal to -128 or 127. The stage output format is generated by truncation of the redundant sign bit and the 10 least significant fractional bits.

This number scheme allows the maximum precision to be retained during the arithmetic operations but reduces the length of values to more manageable lengths between stages. The inter-stage format retains 10 fractional bits which equates to approximately three decimal places of precision in base 10 numbers. As shown in Figure 19, the inter-stage data bus is 108 bits wide. While large, this size is still small enough to allow a single clock transfer between user logic chips on the SRC-6 MAP board utilizing the 192 line bridge port between them. It is also the exact size of the GPIO port, which could be used to transfer data to a separate MAP board for the next stage of processing. Using the GPIO port would require clock synchronization between the chips, so one of the 108 bits in the inter-stage bus would most likely need to be dropped in favor of a clock output. These factors meet the design requirements of precision and extensibility across multiple user logic chips.

2. Filter Stage Structure

The structure for an individual FIR filter is shown in Figure 21. The N values indicate 18-bit inputs. The circles indicate multipliers with the constant 18-bit coefficient index indicated by the number. The rhomboids indicate 36-bit by 36-bit adder units. All multiplier and adder components are constructed with a variable length delay element on the component output. The equations for the delay values that must be applied to each component for each QMF stage are listed in Figure 21. In addition, each component is designed to complete its operation in one clock cycle.

Since the intent is to only calculate every other filter output value, it is not necessary to multiply every input by every coefficient as it would be in a traditional transposed FIR filter like those presented in [5]. The limitation this imposes is that an even number of filter coefficients is required. This creates a problem for using the design as a high pass FIR filter, since high pass filters must normally be of an even order, which means an odd number of filter coefficients. To avoid this issue, the implemented high pass filter was chosen with seven coefficients. The least significant coefficient was omitted (numerical value of -.0013). This introduces inaccuracies into the design. However, since the omitted value is approximately equal to the degree of precision carried over from stage to stage, and significantly smaller than the quantization error at the end of the QMF, the loss of the coefficient has a minimal impact on total system accuracies.



Output Register Delay Formulas
s == stage number
i6 = 2^{s}
i8 = $2^{(s-1)}$
j0,j1,j2 = $2^{(s-1)} - 1$
Highpass k0, k1, k2 = 0
Lowpass k0, k1, k2 = $2^{(s-1)}$
All other output registers have 0 delay
(output appears on next clock)
N
N+1
N+2
N+3
N+4
N+5
xC1
xC3
xC5
xC2
xC4
xC6
i0
i1
i2
i3
i4
i5
i6
i7
i8
j0
j1
j2
k0
k1
k2

Figure 21. 6 Tap, 6 Input, Pipelined FIR filter

The output register delay equations are listed in Figure 21. The wiring pattern between adders is designed to allow values calculated during the previous and subsequent six-value input from a particular data stream to arrive simultaneously with the sums from the current six-value input at the k-level adders. The specifics of the timing used to accomplish this are covered in a later section. The components shown as part of the filter bank in Figure 19 are described in detail in the next sections. It should be noted that the actual ordering of the outputs from the k level adders in time is k1, k2, then k0. The VHDL code correctly orders the outputs for delivery to the next stage.

a. Multipliers

As previously stated, the multipliers in this design are modified from [6]. The timing requirements for a single clock multiply are very strict and great care must be taken in the design of a single clock multiplier unit of this type. The multiplier elements consist of input registers individually assigned a physical location relative to the multiplier block by the use of the RLOC attribute in VHDL. Since the assigned locations are relative to the multiplier block, multiple instantiations of the multiplier code are handled properly by the VHDL compiler. Each input bit is physically aligned adjacent to the input line that will be used to input the bit into the multiplier. The multiplier itself is a dedicated logic device physically located between CLB columns in the FPGA as shown in Figure 5. The output lines on the multiplier were originally connected to D-registers located directly adjacent to the output lines for each respective bit. Modifications to the code from [6] connected the multiplier output lines to SRL16 logic blocks (Figure 6) contained in the same slice as the original D-registers. This change allows the multiplier component to be configured with a variable length output pipeline delay from zero to 16 with zero delay being the equivalent of a D-register.

The current design does not take advantage of the output delay on the multipliers. A design using a mixed configuration of dedicated multiplier elements and array multipliers instantiated in the CLB logic of the FPGA would require the dedicated multipliers to generate a pipeline delay on their output equivalent to the pipeline delay of the array multiplier. The ramifications of this are discussed more fully in Chapter VI.

b. Adders

Each of the adder elements are 36-bit by 36-bit adders with a variable length delay element of the same type as used by the multiplier on their output. The adders are inferred in VHDL by a single line of code: S = A + B. While this method of instantiation seems simplistic, the VHDL compiler on the SRC-6 will infer an optimized adder vertically in a column of slices and utilize the CLAH dedicated logic of the slices to generate a very efficient single clock adder. This functionality was verified with both the Xilinx software tools as well as actual implementation on the SRC-6.

This simple solution works very well for the implemented design but has some potential, though correctable, flaws if used with a larger design. First, the input of the adder does not have a direct, mandatory connection to a register immediately adjacent to the adder, nor does the adder have a guarantee that the delay element will be immediately adjacent to it. This means that any routing delays from the previous component are added to the combinational logic delays of the adder and to the line delay between the adder and the subsequent delay element. If the combined delay exceeds the length of a clock cycle a timing failure will occur. Normally this would not be an issue since the PAR routine automatically attempts to remove any routing delays that would cause this to occur. The second major flaw is that one clock additions are not guaranteed in the current design. If the PAR routine is not able to place the adder contiguously in a vertical column of slices, the low delay interconnect lines and dedicated CLAH logic may not be fully utilized. Both of these flaws will not normally occur unless chip utilization is very close to 100%. Both flaws can also be corrected by the addition of user constraints to the VHDL code. The first flaw can be corrected by placing maximum delay constraints on all of the incoming lines from the previous element. The second flaw can be corrected simply by adding a user constraint that defines the adder and delay element as an object that must occupy a block of 36 vertically contiguous slices tall and two horizontally contiguous slices wide.

c. Delay elements

The stand-alone delay elements are identical to the delay elements that are part of the adder and multiplier elements. They consist entirely of SRL16 configured logic units within a CLB slice. One slice can contain the delay elements for two bits of data to be delayed from 0 to 16 clock cycles, or a single bit to be delayed up to 32 clock cycles. No constraints are placed on the compiler or PAR routing for placement of the delay elements, allowing the automated systems flexibility in their placement. Since the stand-alone delay elements are isolated from combinational logic by dedicated delay elements, the current delay element design should be extensible into any size design.

d. Truncation

The truncation elements are small VHDL modules that accept a 36-bit input and wire connect bits 17 to 35 to an 18-bit output. As previously mentioned, this truncation will cause the VHDL compiler to remove any logic whose only output is one of the truncated bits from the design before it is instantiated and the PAR routine places the components. This produces a savings in hardware utilization without the designer being required to audit every possible path to verify which can be deleted safely. Since the truncation elements are simple wired connections, they are fully extensible into any size design, though it may be preferable to replace them with a rounding system.

e. Output Multiplexer

The output multiplexer from each stage is implemented by a VHDL process statement that switches between the two 108-bit inputs from the two filters within the stage based on a single control line passed to each stage. The output multiplexers must change inputs every 2^{S-1} clock cycles with S being the stage number. The three least significant bits of the output address counter of the FIFO element are used to drive the three output multiplexers at this rate for this design. Since each slice in the CLB matrix contains two dedicated 2-to-1 single bit multiplexers (Figure 6), the hardware implementation will result in the partial utilization of 54 slice elements. Since the inputs to the multiplexers are the outputs of SRL16 based delay registers, the VHDL compiler and PAR routine will attempt to place the actual multiplexers into slices already occupied by the preceding logic. This design is both simple and robust and should be easily extensible into a larger design. However, the system designer needs to be aware of the alignment of the multiplexer transition with respect to transitions in the preceding stages. Since the multiplexers control the interleave order of the data streams, the output order of the individual streams within the data path are determined by multiplexer alignment between the various stages.

3. Pipeline Timing

Derivation of the pipeline timing for the filter element is the most complex portion of the design. The pipeline delays and component connections must be organized in such a way that the only data values that exist in the same data stream be used in calculations, and that the data values in each data stream are correctly ordered. In addition, the filter elements for each stage must be able to handle 2^{S-1} data streams interleaved together in one continuous data path.

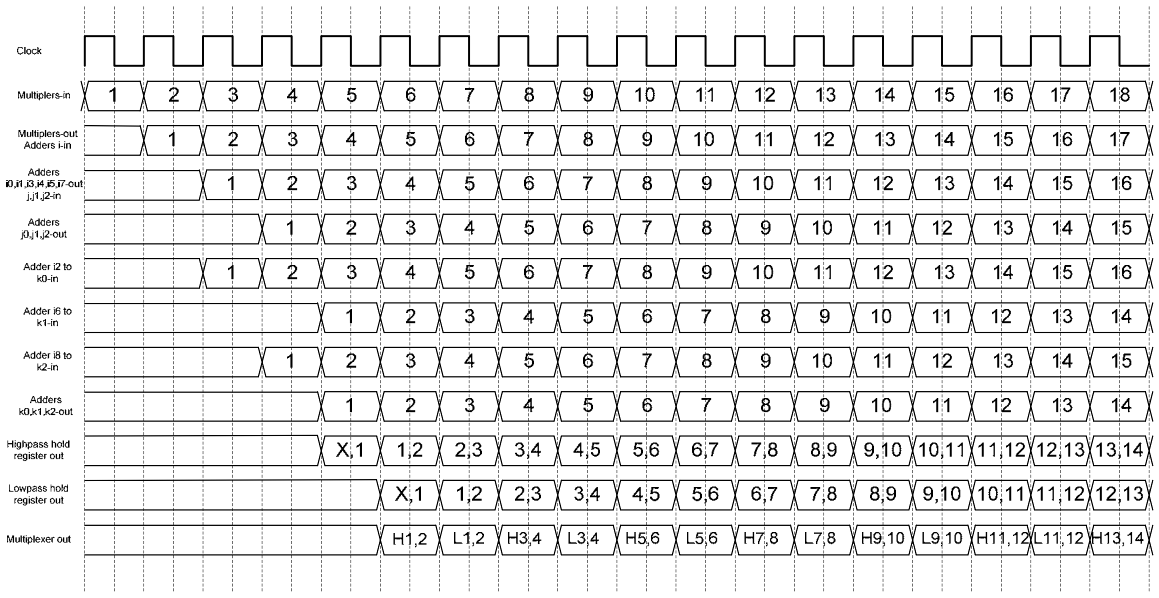

Clock
Multiplers-in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
Multiplers-out Adders i-in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
Adders i0,i1,i3,i4,i5,i7-out j,j1,j2-in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Adders j0,j1,j2-out 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Adder i2 to k0-in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Adder i6 to k1-in 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Adder i8 to k2-in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Adders k0,k1,k2-out 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Highpass hold register out X,1 1,2 2,3 3,4 4,5 5,6 6,7 7,8 8,9 9,10 10,11 11,12 12,13 13,14
Lowpass hold register out X,1 1,2 2,3 3,4 4,5 5,6 6,7 7,8 8,9 9,10 10,11 11,12 12,13
Multiplexer out H1,2 L1,2 H3,4 L3,4 H5,6 L5,6 H7,8 L7,8 H9,10 L9,10 H11,12 L11,12 H13,14

Figure 22. First Stage QMF Pipeline Timing Diagram

Figure 22 shows the timing diagram for the first stage pipeline of the demonstration QMF design. The input sets are numbered to illustrate a continuous input stream of sequential values in six value sets. In the first stage, the input sets enter the successive elements shown in Figure 21 in the relative time slots shown in Figure 22. It should be noted that the diagram assumes that the pipeline is full and that a value from the equivalent of input set 0 would be available at the output of adder i6 for the calculation of the first output set. Since each six input data set outputs three values, the values of two successive sets are combined using the delay element shown just before the multiplexer in Figure 19. The low pass filters in the design were arbitrarily chosen to have additional delay on the output of the k-level adders so that they are aligned properly when the output multiplexer switches to the low pass filter output. The notation for the first value in the multiplexer out line of Figure 22 indicates that the value is the result of the high pass filter calculations on input sets one and two combined into a single output set of six values. The next element is the low pass filter calculation on input sets one and two. Because the stage 1 output multiplexer switches every clock cycle, every other output set from each filter and delay element is discarded, although the individual three value components of each discarded set exist as part of the previous or following six value set.

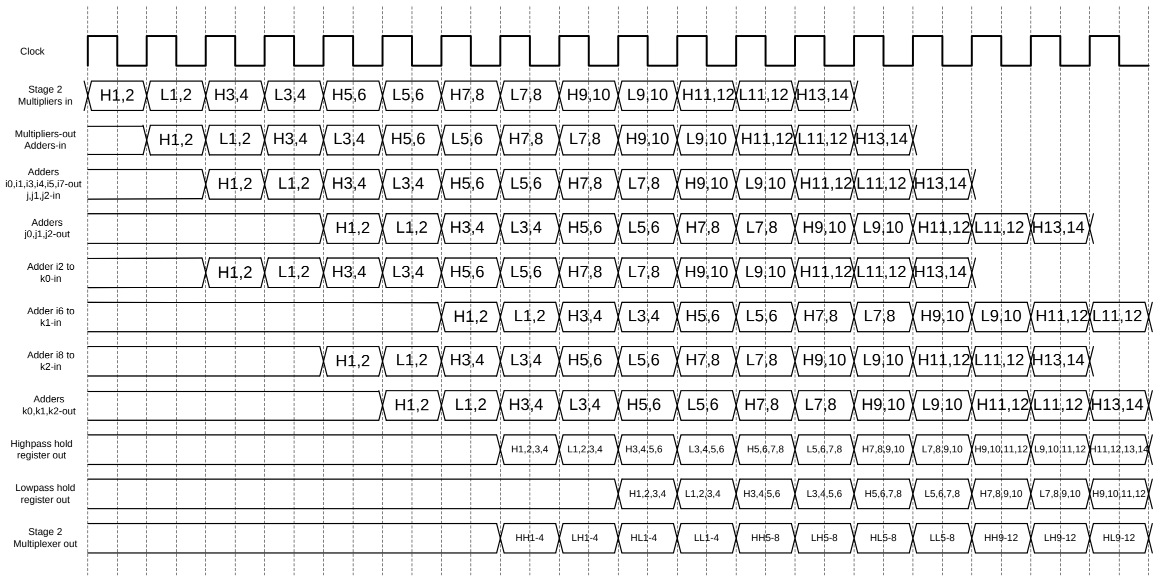


Figure 23. Second Stage QMF Pipeline Timing Diagram

Figure 23 shows the pipeline timing diagram for the second stage filter of the QMF design. The second stage filter bank is instantiated with the same VHDL module as stage 1 and stage 3, but different delay values for the various components are specified. Again, the diagram assumes the pipeline is full and a preceding value is available at the i6 output for the calculation of the first output set shown. Delay values on the various elements must be increased to ensure the proper data streams are aligned at each stage of the calculation. The labeling of the first output on the multiplexer out line indicates it has passed through two successive high pass filters and is the composite six-value data set derived from data sets 1-4 by combinations and down sampling. The second output is the low pass output from the first stage that has subsequently passed through the high pass filter in the second stage and is also the composite of sets 1-4. Successive data sets are labeled accordingly.

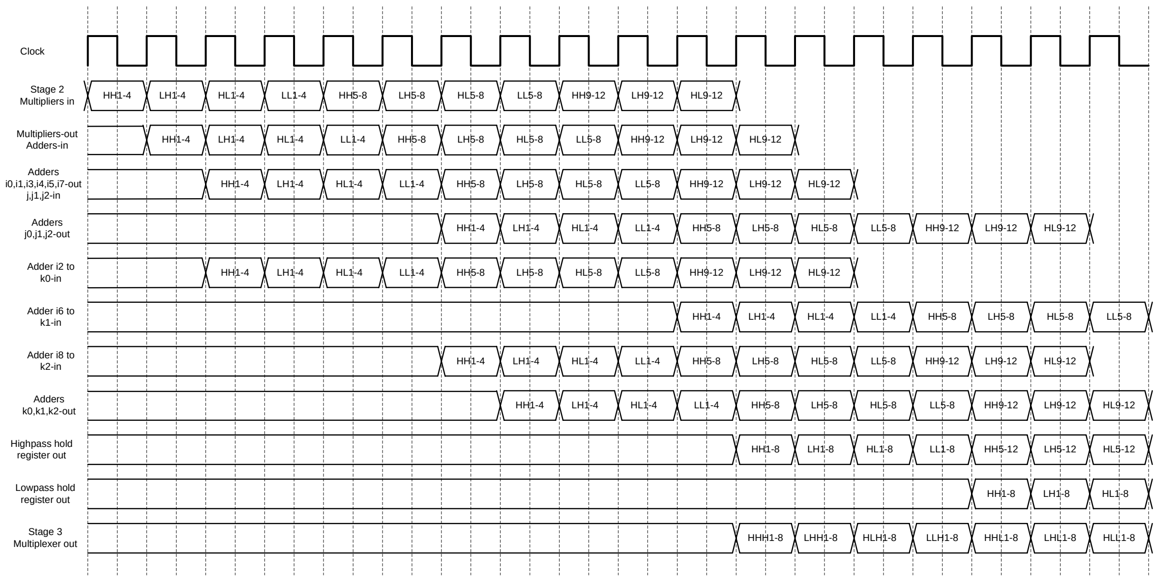


Figure 24. Third Stage QMF Pipeline Timing

Component delays are increased for the third QMF stage as given by the equations in Figure 21 and shown in Figure 24. The labeling on the first output data set indicates that it has passed through three successive high pass filters and is the composite data set derived from input data sets 1-8. The second output set first passed through the low pass filter in stage 1, then the high pass filters in stage 2 and stage 3. Subsequent data sets are labeled accordingly.

The data sets belonging to specific data streams representing a particular path through the filter bank are output in a consistent order which allows the data sets to be de-interleaved before being passed to another stage for further processing. Currently, the design outputs the data to the C code portion of the program while it is still interleaved. For purposes of this work, the de-interleaving is done by using MATLAB to separate the data streams.

D. IMPLEMENTATION IN SRC C CODE

An implementation of a single low pass filter element in C code is included in Appendix D. In Chapter VI, the hardware utilization and accuracy of this implementation is compared against a single low pass filter element from the QMF design in terms of hardware utilization and precision. A speed comparison between a C code based FIR and a VHDL based FIR is irrelevant since they both accept inputs on each clock cycle and produce outputs on each clock cycle. The pipeline depth is different but not relevant since the output is continuous once the pipeline is full of data and any pipeline depth meets the real-time requirement. The C code version of the FIR filter was written in a form optimized for the execution on the SRC-6 MAP. A fixed point number scheme similar to the VHDL code is used for data storage between calculations. However, number widths are slightly smaller than the VHDL code, with 18-bit VHDL values residing in 16-bit variables in C and 36-bit VHDL values residing in 32-bit values in the C code version. Calculations are ordered in a non-intuitive manner in the C code. The ordering is designed to avoid any intermediate storage of values in either the MAP onboard memory or BRAM modules instantiated by the C code, both of which could cause significant increases in pipeline depth or potentially cause the code not to execute in real-time, as required.

E. IMPLEMENTATION IN MATLAB

A MATLAB version of single filter elements, as well as a complete QMF design, is included in Appendix E along with the MATLAB code used to de-interleave and process the output data from the SRC into a format for comparison to the MATLAB QMF results.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. PERFORMANCE RESULTS AND COMPARISONS

A. HARDWARE INTERFACE RESULTS

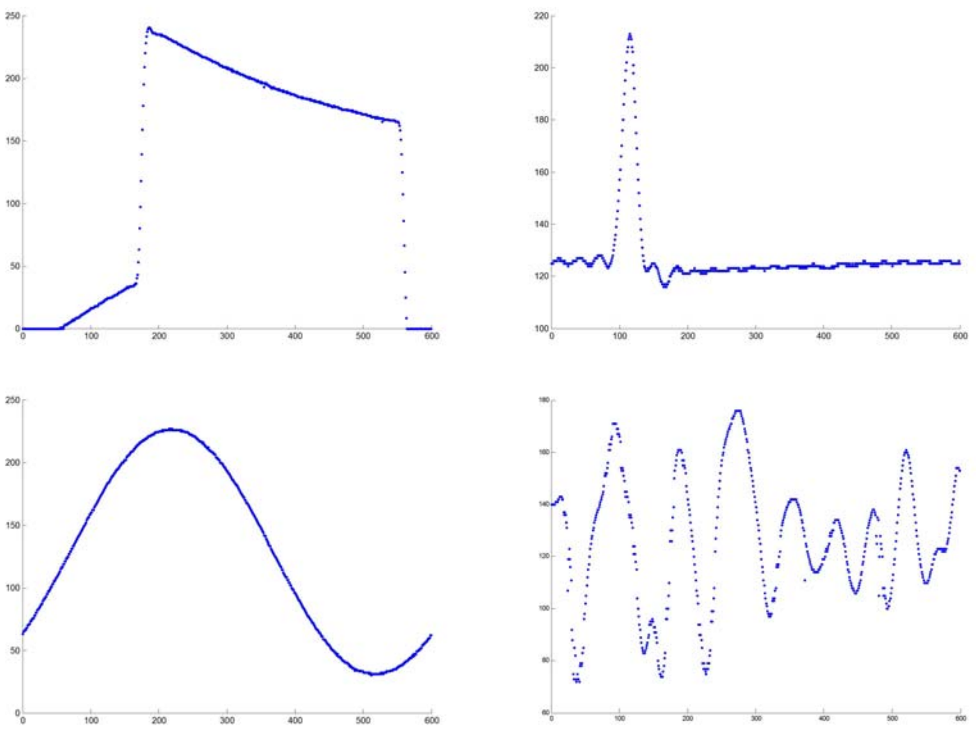


Figure 25. Selected Input Waveforms Sampled at 600 MSps

Figure 25 shows some selected waveforms sampled at 600 MSps on the ADC and read into the SRC-6. These waveforms were collected utilizing a user macro containing only the hardware interface portion of the design code. The performance of the hardware interface portion of the design was determined to be sufficient to collect ADC data for use in testing the QMF designs. Numerical error rates were not determined for the hardware interface since the purpose of the interface was to enable the testing of the QMF design, not as a stand-alone design element. However, error rates for the hardware interface are seen to be fairly low when large samples groups are viewed. Figure 26 shows a group of 6000 samples with no visible errors. When the hardware interface code block is instantiated by itself, results as shown in Figure 26 are typical. However, when combined with larger logic blocks such as the VHDL filters or the full QMF, higher error rates are visible in the unfiltered data output from the SRC-6. As previously mentioned, the most likely cause for the rise in error rates is lack of location constraints placed on the input registers within the hardware interface section of code. With the current code, the PAR routine does not recognize the tight timing tolerances for the input registers since it has no knowledge of the input signal clock rates. Without additional constraints, the PAR routine will route signals from the input buffers so that they will arrive at the input registers within tolerance for 100MHz signals. Since the incoming signal is at a higher frequency, this can be a problem if the delays are not consistent. Consistent delays can be removed by a phase shift on the incoming clock signal.

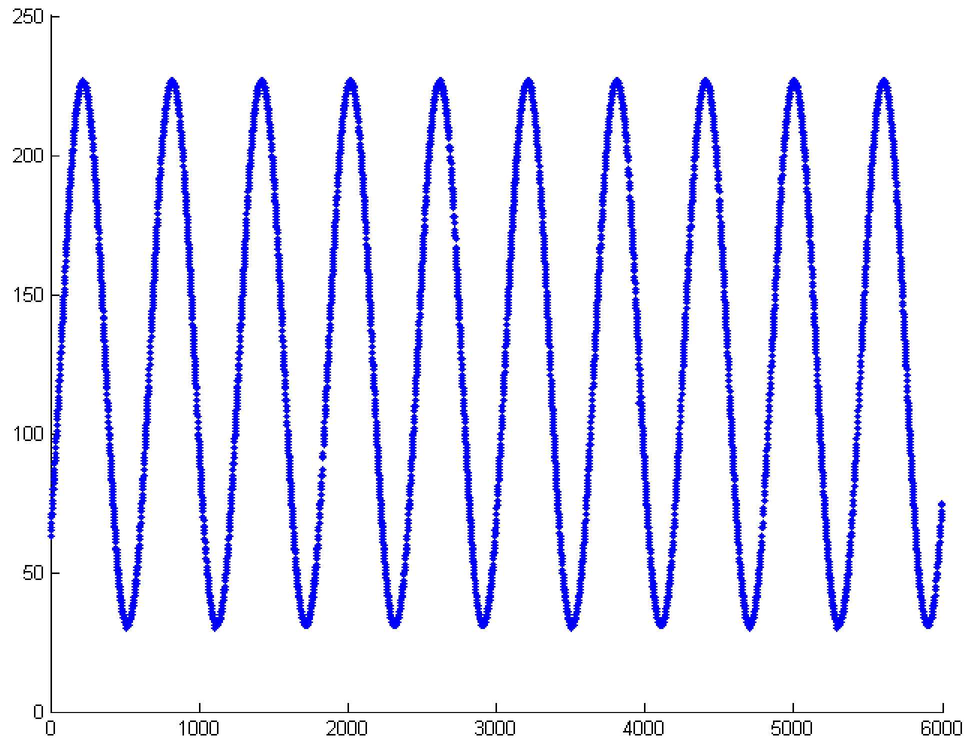


Figure 26. 1 MHz Sine Wave Sampled at 600 MSps

One additional improvement that can be made to the hardware interface, besides the location constraints on the components, involves the cable interface. By improving the cable interface, much faster transfer rates should be possible using the same design techniques described in this work. Data input at up to 644 MHz is documented in [7] and would translate to an ADC sampling rate over 1.2GSps. In addition, by connecting to the same ADC chip model that is mounted on a board that would allow the chip to enter DES mode, the effective sampling rate could be doubled to over 2GSps.

B. HARDWARE UTILIZATION

	Flip-Flop	FF %	LLUT	LLUT %	SRL	SRL %	Total LUT	TLUT %	SLICES	SLICE %	18x18 Multipliers	MULT %
Full QMF	2827	4.2%	3281	4.9%	6829	10.1%	10110	15.0%	8526	25.2%	108	75.0%
VHDL Filter	471	0.7%	547	0.8%	1099	1.6%	1646	2.4%	1423	4.2%	18	12.5%
C Code Filter	5406	8.0%	3550	5.3%	137	0.2%	3687	5.5%	3157	9.3%	0	0.0%
C Code Filter 2	4526	6.7%	2726	4.0%	50	0.1%	2776	4.1%	4740	14.0%	36	25.0%
Overhead	3661	5.4%	1307	1.9%	0	0.0%	1307	1.9%	2610	7.7%	0	0.0%

Table 4. Hardware Utilization Comparison

Table 4 is a detailed listing of the hardware utilization of the FPGA for the full VHDL QMF design, a VHDL high pass filter, and two versions of the C code high pass filter. Also listed is the overhead that is common to all four designs. The overhead consists of the hardware interface portion of the design and the logic instantiated by the SRC compiler for MAP communications and control. The table shows the total number of flip-flops used, the number of LUTs configured as 4-input logic, the number of LUTs configured as SRL16s, the total number of LUTs used, the number of slices containing logic, and the number of 18x18 multipliers used. Each quantity also has the percentage of the total chip resources of that type that are used. The numbers are derived from the compilation reports produced by the SRC compiler. The utilization for the overhead logic has been subtracted from the actual generated report values to produce the values shown in the table.

1. Full design

Note that the full QMF uses the largest fraction of the 18x18 multipliers, 75%. An upper bound on the size of the QMF is given by Equation 6.1,

$$I \cdot F \cdot S \leq M \tag{6.1}$$

where I is the number inputs, F is the number of filter taps, S is the number of QMF stages, and M is the number of available multipliers. The demonstration QMF design has nearly reached the maximum size using the 18x18 multipliers with 108 multipliers in use out of 144 available. With the current design, one more stage containing two filter elements with 36 multipliers could be added or the tap size of each filter could be enlarged to eight taps. Either option would increase the multiplier module utilization to 100 percent. It can also be observed that the utilization of the other FPGA components scale linearly from the VHDL filter design. The VHDL filter is copied six times in the full design, and component usage for the full design, is approximately six times that of the filter only design. This indicates that without considering multiplier usage, the QMF design could grow to approximately four times its current size and still fit on a single chip. The addition of the overhead would appear to push this percentage over 100 percent. However, most of the instantiated logic is not fully packed into the available space on the chip.

The percentage of slice utilization is approximately five percent higher than the sum of the total LUT utilization and total flip-flop utilization. This indicates that logic which could be packed into a single slice is instead residing in separate slices. As chip utilization increases, more time will be spent by the PAR routine to more efficiently pack logic into slices. When chip utilization is low, as in all these designs, less effort is spent on combining logic elements into tighter physical locations.

2. VHDL Filter Element

The chip utilization shown for the VHDL high pass filter is within the expected bounds for the design. There is a slight reduction in component utilization from what is explicitly defined within the VHDL. This is due to portions of logic being removed by the compiler because the output dependencies of the logic are located only within the bits that are truncated at the end of the calculations. Replacing the bit truncation with a rounder would reduce the savings in hardware utilization.

3. C Code Filter Element

The two different versions of the C code high pass filter shown indicate two different methods for instantiating the same function from within the C code. The process, operations conducted, and operation ordering are identical between the two versions. The only difference in the code that produced these two utilization reports is the size of some of the variables. One of the intermediate variable types involved in the multiplication operations was changed from an 8-bit variable to a 16-bit variable data type. The first version, using the 8-bit data type, created multipliers in the CLB matrix instead of utilizing the dedicated multiplier units. The second version does use the dedicated multiplier blocks. It used twice as many as the VHDL version, for slightly lower precision variables. Both versions of the C code filter used between two and three times the total utilization seen in the VHDL module.

Neither version of the C code utilizes LUT elements configured as SRL16 shift registers. This seems to indicate that all pipeline delays within the C code filters utilize flip-flops as delay registers. This is supported by the total flip-flop utilization seen for the C code variants. It would be interesting to observe the behavior of the C code version with the much longer delays in the data path that occur in later stages of the full QMF design. Due to time constraints, completion and testing of a full QMF design in C code was impractical.

C. EXPANSION OPTIONS

As previously noted in Chapter V, there are several methods of creating multiplier elements within the FPGA. The main limiting factor of the current design is the number of multipliers available. By augmenting the dedicated multipliers with alternate multiplier instantiations methods, significant increases in total filter size are possible. By utilizing wired multiplies (bit shifts) where filter coefficients can be approximated by negative powers of two, effectively no additional hardware is instantiated for the multiply.

Utilization of BRAM modules as look-up tables will increase the number of multipliers available. This method would be most effective for the first stage of the QMF. Since the first stage can accept the 8-bit value output of the hardware interface, two BRAM modules can be used as look up tables to produce two 32 bit wide values from two inputs. If it is assumed a FIR filter with satisfactory performance can be made with 40 filter taps, and each of the six inputs is multiplied by one half of the filter taps, each stage of the QMF would require 240 total multiplies to implement both a high pass and low pass filter. In terms of multipliers, this is well within the 288 bound of 144 dedicated multipliers plus 144 BRAM modules (minus BRAMs used in the FIFO). The adder trees associated with each of the two filter elements in the stage will each require 117 adders to complete their operations for a total of 234 adders. A very rough calculation of chip utilization based on the utilization for the VHDL high pass filter, would place the utilization percentage for the chip at around 50 percent. This is satisfactory for the first QMF stage where the inputs are 8-bit inputs.

Subsequent stages, which are assumed to exist on a separate FPGA, can also make use of the dedicated multipliers and the BRAM look up tables. However, subsequent stages incur an additional penalty for BRAM usage. Without decreasing the precision of the 18-bit inputs for each stage after the first, two BRAM modules will no longer be sufficient to produce two 16-bit values. However, by reducing the inputs to 16-bit values, four BRAM modules can produce two 32-bit partial products that will result in the correct product when added together. This results in four BRAM look up tables supporting two inputs, decreasing the available multipliers to 221 as well as requiring a two clock multiply. The two clock multiply can be handled by the multiplier units designed for the demonstration filter simply by modification of the delay value passed to their integral output delay modules. An additional 13 multipliers are still necessary to fulfill the requirements for later stages. These can be created as array type multipliers within the CLB matrix. The array type multipliers will require additional pipeline delay and are also much more resource intensive, as demonstrated by the C code variant which instantiated multipliers within the CLB matrix. A design of this size is practical and could be realized by placing one QMF stage on each available FPGA.

Newer model SRC-6 systems contain an upgraded Xilinx FPGA. The component specifications for this chip are listed in Table 1 under the XC2VP100 model. It contains a total of 444 dedicated multipliers and 444 BRAM modules. However, it only has an increase of about 30 percent in total CLB capacity. With this chip, large QMF designs would be possible. By using the techniques described in the preceding paragraphs, multipliers are no longer the only limiting factor with this chip. CLB logic would likely be exhausted before full utilization of the dedicated multipliers and BRAM look up tables is reached.

D. QMF PERFORMANCE

The exact performance of the SRC based QMF filter is difficult to quantize numerically. Uncertainties in the system startup timing make it difficult to determine the exact starting point of the filter. While it is possible to de-interleave the output path correctly by means of an additional output bit that identifies the state of the final output multiplexer, the current design does not allow the determination of the first discarded sample. This means that calculations on the exact same sample set are not possible from MATLAB, using the current design. Visual comparisons of the QMF calculations in the SRC-6 agree with the calculations in MATLAB. In many cases, individual data points are within the quantization error between the high precision MATLAB calculations and the integer values returned by the SRC-6 QMF. By averaging the difference between the VHDL QMF data point and MATLAB data point occupying each time slice, an average difference value can be generated for the sample set. This value is not an exact representation of the error between the sample sets, since the VHDL data points will be shifted by some unknown amount. The average difference does provide an upper bound on the actual error. The MATLAB correlation function was also used to produce a linear correlation coefficient for the two waveforms. This value is useful for comparison but does not properly represent magnitude errors between the waveforms. Correlation function calculations are generally consistent across the eight waveforms produced by the three stage QMF design.

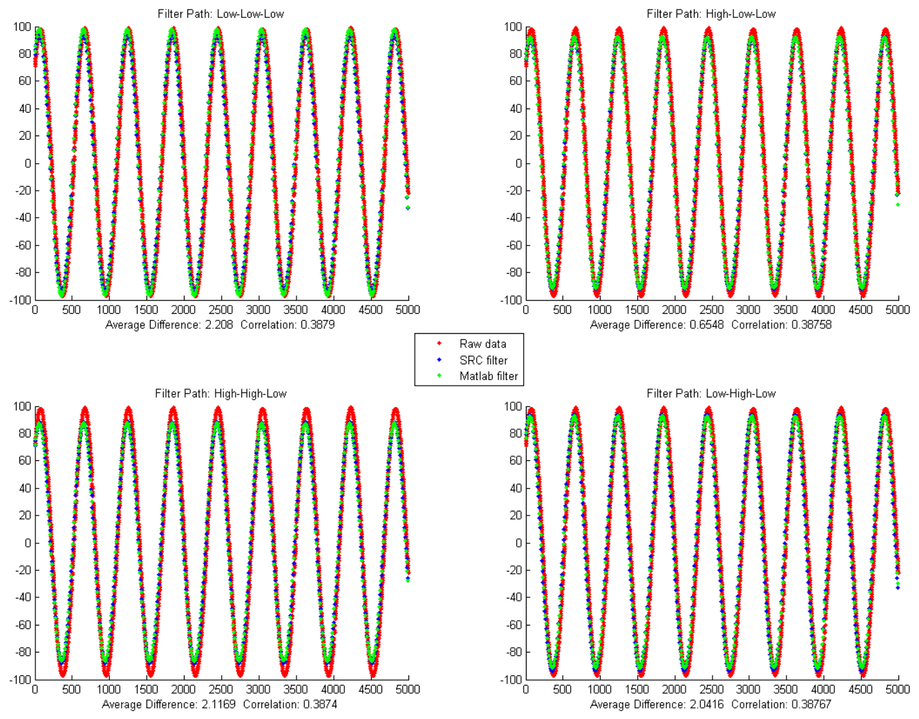


Figure 27. Comparison of 1Mhz Sine Wave Through the QMF Path

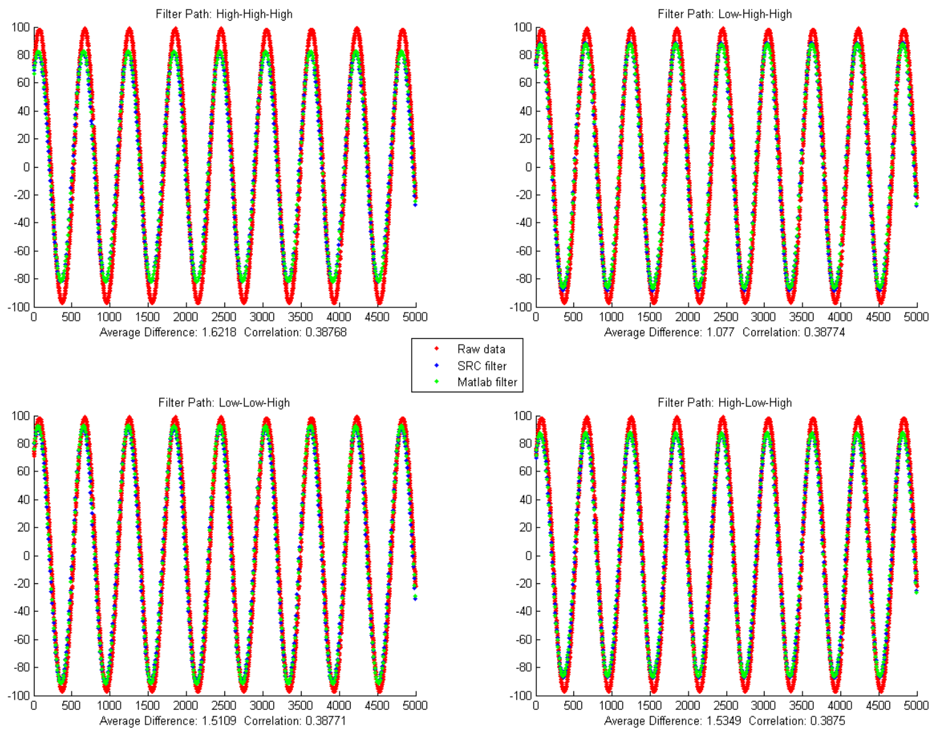


Figure 28. Comparison of 1Mhz Sine Wave Through the QMF Path

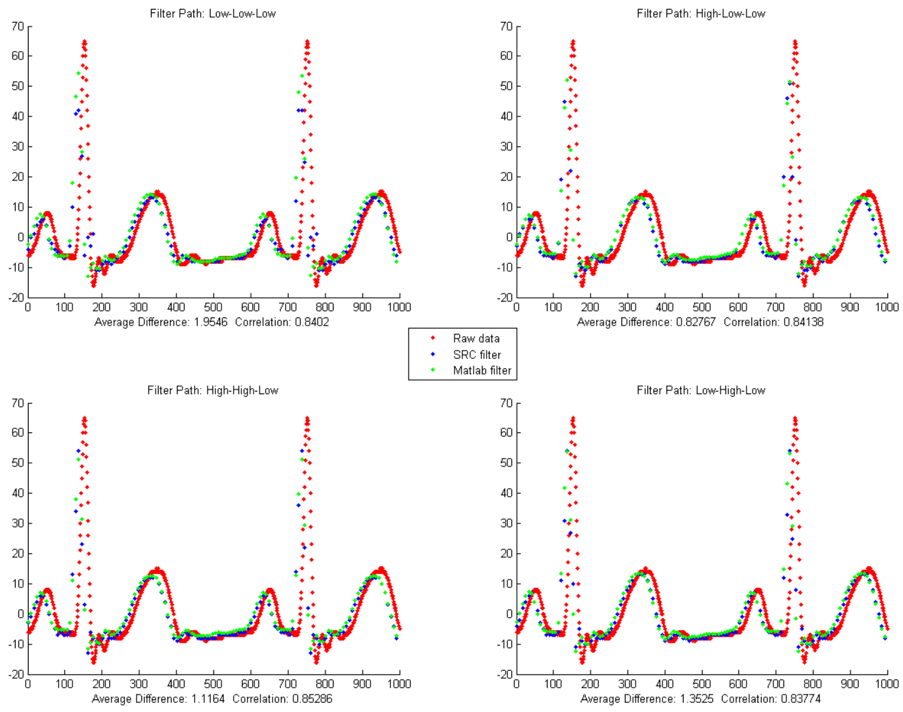


Figure 29. Comparison of 1Mhz Cardiac Wave Through the QMF Path

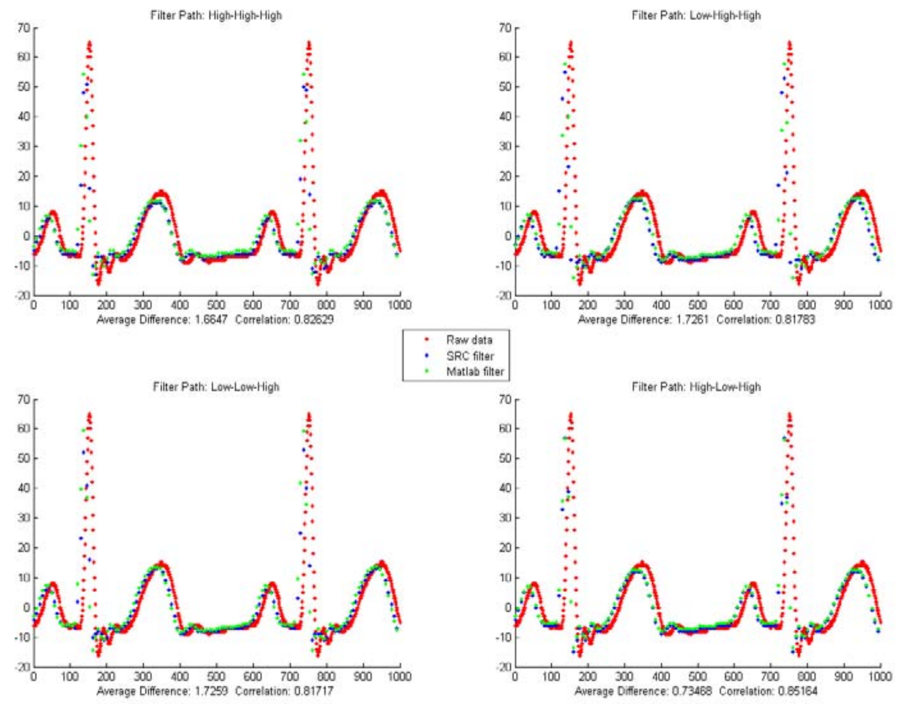


Figure 30. Comparison of 1Mhz Cardiac Wave Through the QMF Path

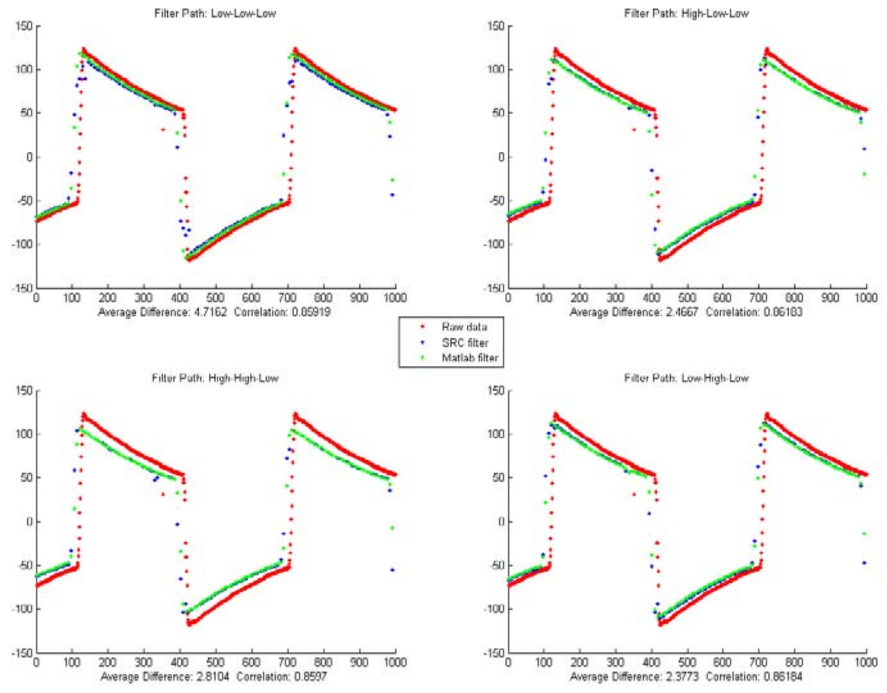


Figure 31. Comparison of 1Mhz Sine Wave Through the QMF Path

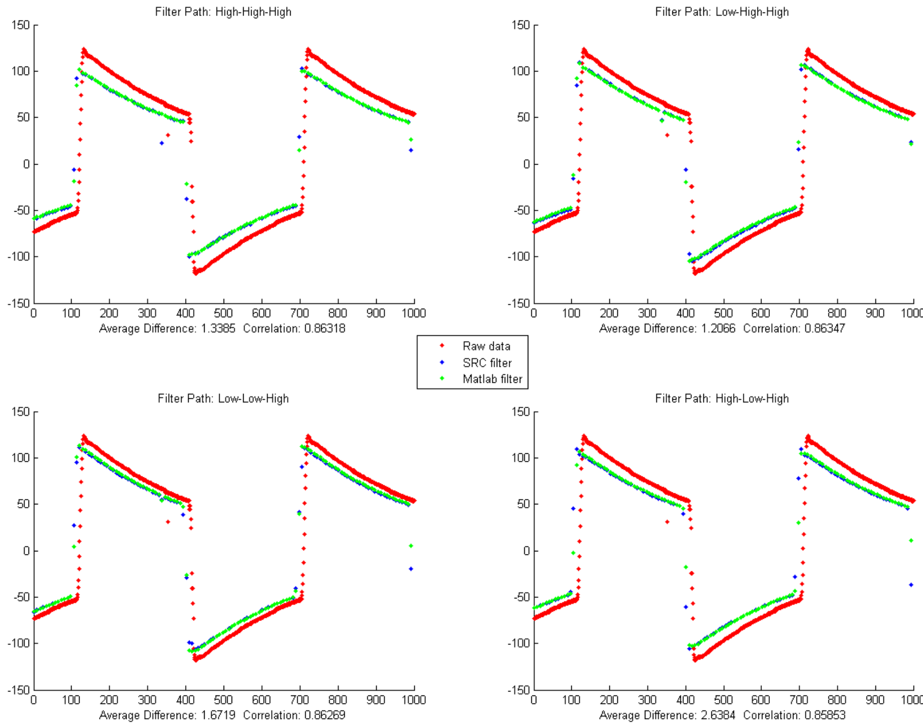


Figure 32. Comparison of 1Mhz Cardiac Wave Through the QMF Path

Figures 27-32 show the output of both the VHDL based QMF and the MATLAB based QMF through all of the possible QMF paths. To produce these figures, the offset caused by the pipeline delay in the VHDL version has been removed. Additional Figures are included in Appendix F.

The signal path that passes through all low-pass filters consistently produces the largest average difference. There is a small but visible attenuation of the signal on this path. This is most likely due to truncation errors adding up through the QMF path. The particular coefficients of this path seem to be especially sensitive to the bit truncation errors. This can be partially corrected by replacing the truncation system with a rounding system. As previously stated, this will affect the hardware utilization since the VHDL compiler will no longer delete logic when outputs are not used, as well as requiring additional logic and pipeline delays.

There is a very high degree of congruence between the MATLAB calculated waveforms and the waveforms produced by the real-time VHDL filter. The average difference value is fairly low in most cases. The cardiac waveforms provide an especially clear display of the time shifting between the VHDL and MATLAB QMF implementations. From observation of the these waveforms, it can be observed that the individual data points appear to fall along the same waveform path but with a small time shift introduced by the uncertainty of which samples were discarded by the SRC-6 based code. This uncertainty could be removed by adding additional circuitry to report the interleave status of the data streams through the entire QMF path in the same fashion that the final multiplexer control bit is passed out to allow the de-interleaving of the data streams. This would not be required for normal operation of the program as the intent is to pass on only processed data.

The QMF output interleave pattern for this particular design is different from the generic timing charts shown in Figures 22-24. The interleave multiplexers for the demonstration design were controlled by the least significant bits of the hardware interface FIFO address counter. The pipeline depth for the implementation would require the multiplexer control signals to be on a slightly different timing than the FIFO address circuit to produce the exact output order shown in the timing diagrams. The actual output order of the data streams for demonstration QMF with respect to the output multiplexer control line transitioning from high to low is: high-low-low, low-high-low, low-low-low, high-high-low, high-low-high, low-high-high, low-low-high, and high-high-high. With this design, the sequential order of the data streams will remain the same. The starting point for the pattern occurs when bit 48 of the output bus transitions from high to low.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. CONCLUSION

A. SUMMARY

This work has provided an overview of the hardware and software concerns relevant to the construction of a Quadrature Mirror Filter bank on the SRC-6 reconfigurable computer system. A discussion is given for a possible practical use of a QMF for the detection and classification of Low Probability of Intercept signals, as part of a larger system. Specific elements of the SRC-6 reconfigurable computer are discussed with respect to their relevance to the designs contained in this work, and a background of related and similar work is given.

The main design project of this work was focused on two main goals. The first main section consists of the hardware interface between a National Semiconductor ADC08D1500 Analog to Digital Converter and the SRC-6 reconfigurable computer. The hardware interface was detailed in terms of the physical interface and the logical interface. The physical interface was defined as the specifications of the ADC board, the cabling and physical connections, and the electrical specifications of the signal path. The logical interface was defined to include hardware instantiated in the reconfigurable logic on the SRC-6 in order to receive the incoming data stream from the ADC and prepare it for further processing or storage within the SRC-6 MAP board. Several of the important design choices associated with the design were discussed and compared to alternative approaches. Deficiencies in the design were identified and proposals were made for correcting the design deficiencies.

The second main section of the work involved the actual construction and testing of a QMF design in VHDL for implementation on the SRC-6. After a detailed listing of the design requirements for a demonstration size QMF design, a discussion was given for some of the critical design choices made in the design along with the advantages and limitations of the design choices. An explanation for reducing hardware utilization, through the re-use of hardware elements by interleaving different streams of data along the same data path, was provided. Details of the actual implementation were given with respect to the binary number formats and the actual hardware component design in

VHDL. Several timing diagrams were provided to illustrate the process used by the design for implementation of a data stream interleaving scheme to allow maximum utilization for the minimum amount of hardware. Brief descriptions were given for versions of a single high pass filter created using the SRC CARTE programming language for comparison against the VHDL version of a high pass filter element.

Finally, performance results for the hardware interface and QMF design were given. The goal of a working hardware interface with acceptable error rates was achieved for that portion of the design. Hardware utilization comparisons were made between the full QMF design in VHDL, the high pass filter design in VHDL, and two versions of the C Code high pass filter. Options available for expanding the size of the demonstration design into a larger practical design were given along with a discussion of relevant issues in expanding the design. A display of the actual performance of the real-time, VHDL QMF design were made against the unfiltered data and QMF calculations made in MATLAB using a similar filter design.

1. Practical Limitations

This work has discussed a number of practical limitations in designing a full scale QMF for implementation on a particular FPGA architecture. The main limitation of the design is set by the number of simultaneous multiplications that can be accomplished within the target FPGA. The number of simultaneous multiplies is determined by multiplying the number of filter taps with the number of simultaneous inputs to the system and the total number of QMF stages.

As the number of filter taps increase, the complexity of the filter design increases significantly. The design methodology described in this work will always result in filter output sets one half the size of the input sets (6 inputs equal 3 outputs), however the number of components and simultaneous operations are approximately equal to the number of filter taps multiplied by the number of simultaneous inputs. Developing the pipeline timing and wiring structure for larger designs is extremely complex and will require significant development time.

2. Data Interleaving

The methods described in this work for interleaving multiple data streams along a single data path have proven to be effective with the demonstration QMF design. While the complexity of the filter design will increase with larger designs, the complexity of the data interleaving system does not increase. The width of the data path may vary depending on the number of simultaneous system inputs but the interleave system will not. Larger designs will have to account for changes in pipeline depth to properly de-interleave the data. However, once the interleave pattern for a particular pipeline depth is known, it remains constant.

3. C Code versus VHDL on the SRC-6

Some useful conclusions can be drawn involving the benchmarking the SRC-6 system. The VHDL code filter design was two to three times smaller than the C Code versions, while maintaining slightly better precision. Development time for the C code filter was less than one hour, while the development time for the VHDL version was well over five hours for the comparable code block. Including the research required for the background to complete the VHDL version, the VHDL version required approximately ten times the total time to produce.

The conclusion that can be drawn from the disparity in development times is that the C code environment provides an outstanding development platform for FPGA based code design. Even for designs that will eventually be ported to VHDL for improved performance, the rapid development times possible using the C code environment may significantly enhance a design project.

One area where the SRC-6 software environment currently lacks features is when dealing with connections to external hardware. VHDL user macros are currently required for the interface, as well as for synchronizing external data to the internal clock domain. A process for synchronizing external data collected from a custom interface would reduce the requirements for the interface designer developing a clock synchronization system. Such a system must already exist within the standard SRC macro libraries for use by the SRC macros for GPIO port access but are not available as separately callable functions.

B. SUGGESTED FUTURE WORK

There are a number of areas related to this work that can be extended with future work. With respect to the SRC code libraries, the addition of standard DSP filter elements to the macro libraries would be beneficial for any signal processing application. Construction of optimized FIR type filters, which could be passed a data stream and a set of coefficients, would greatly enhance the utility of the SRC-6 for prototype and developmental testing of signal processing systems. Also, with respect to the SRC macro library, some type of synchronization routine, such as the FIFO used in this work, could be written to simplify data synchronization for a user macro when there is a requirement to deal with data in a different clock domain.

With respect to the QMF filter design, future work could involve extending the size of the filter elements to a practical size. As previously stated, the size and complexity of the filter elements will increase significantly as the number of simultaneous inputs and filter taps are increased. Also, with respect to the QMF design, support code and structures could be developed to extend the QMF across multiple user logic chips or MAP boards. Spanning of multiple FPGA devices would be a requirement for a practical QMF design, and while the data path size in the current design will support spanning multiple devices, there must also be control and synchronization systems that have not been developed.

With respect to the SRC general environment, future real-time signal processing systems require an interface capable of continuously transferring data from the MAP board to the microprocessor. While limitations in the current architecture of the interface prevent simultaneous bi-directional communications, a software system could be developed that samples data in a near-continuous stream from the MAP, yet is still capable of passing control signals back to the MAP on some interval. Such a system would allow for active control of a real-time system and continuous display of data on the microprocessor.

APPENDIX A. PHYSICAL INTERFACE MAPPINGS

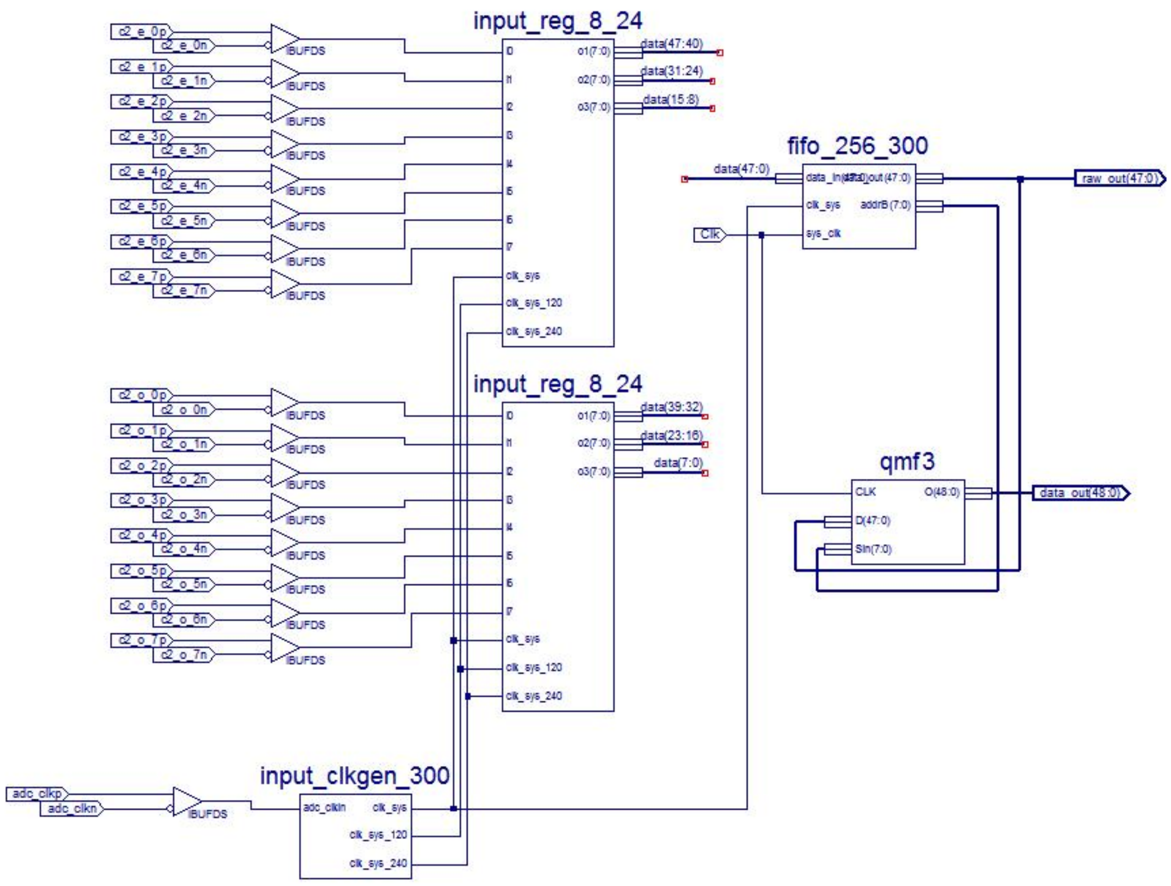
ADC CONN 1 80-PIN CABLE PIN	ADC CONN 2 80-PIN CABLE PIN	ADC 114-pin cable #	Breakout Board Pad # ADC side	Breakout Board Pad # SRC side	SRC 114-pin cable #	Signal Name	User Logic 1 Pad #
77		1	D00	D03	4	c1_e_7n	N2
78		2	D01	D01	2	c1_e_6n	T12
79		3	D02	D02	3	c1_e_7p	M2
80		4	D03	D00	1	c1_e_6p	U12
	3	5	D04	D38	39	clk_p	F19
	1	7	D06	NC1	79	clk_n	F20
	7	11	D10	D13	14	c1_e_5n	P10
	8	12	D11	D11	12	c1_e_4n	N5
	9	13	D12	D12	13	c1_e_5p	R10
	10	14	D13	D10	11	c1_e_4p	M5
	13	17	D16	D19	20	c1_e_3n	N9
	14	18	D17	D17	18	c1_e_2n	M4
	15	19	D18	D18	19	c1_e_3p	P9
	16	20	D19	D16	17	c1_e_2p	L4
	19	23	D22	D22	23	c1_e_1n	M6
	20	24	D23	D26	27	c1_e_0n	K2
	21	25	D24	D21	22	c1_e_1p	L6
	22	26	D25	D25	26	c1_e_0p	J2
	37	41	D40	D42	43	c2_o_7n	K6
	38	42	D41	D44	45	c2_o_6n	M10
	39	43	D42	D41	42	c2_o_7p	J6
	40	44	D43	D43	44	c2_o_6p	N10
	41	45	D44	D46	47	c2_o_5n	J3
	42	46	D45	D48	49	c2_o_4n	H4
	43	47	D46	D45	46	c2_o_5p	H3
	44	48	D47	D47	48	c2_o_4p	G4
	47	51	D50	D50	51	c2_o_3n	L9
	48	52	D51	D54	55	c2_o_2n	K7
	49	53	D52	D49	50	c2_o_3p	M9
	50	54	D53	D53	54	c2_o_2p	J7
	53	57	D56	D56	57	c2_o_1n	K9
	54	58	D57	D60	61	c2_o_0n	H5
	55	59	D58	D55	56	c2_o_1p	L10
	56	60	D59	D59	60	c2_o_0p	G5
	59	63	D62	D62	63	c2_e_7n	J8
	60	64	D63	D66	67	c2_e_6n	H6
	61	65	D64	D61	62	c2_e_7p	K8
	62	66	D65	D65	66	c2_e_6p	H7
	65	69	D68	D68	69	c2_e_5n	M12

	66	70	D69	VALID0	73	c2_e_4n	G6
	67	71	D70	D67	68	c2_e_5p	N12
	68	72	D71	D71	72	c2_e_4p	F5
	71	75	VALID1	VALID1	75	c2_e_3n	L12
	72	76	VALID2	VALID3	77	c2_e_2n	F4
	73	77	VALID3	FULL	74	c2_e_3p	M11
	74	78	VALID4	VALID2	76	c2_e_2p	E4
	77	81	SPARE1	SPARE4	84	c2_e_1n	C4
	78	82	SPARE2	SPARE2	82	c2_e_0n	E3
	79	83	SPARE3	SPARE3	83	c2_e_1p	B4
	80	84	SPARE4	SPARE1	81	c2_e_0p	D2
53		85	SPARE5	SPARE8	88	c1_o_7n	F8
54		86	SPARE6	SPARE6	86	c1_o_6n	C5
55		87	SPARE7	SPARE7	87	c1_o_7p	E8
56		88	SPARE8	SPARE5	85	c1_o_6p	C6
59		91	INT DCLK	PROC RST	94	c1_o_5n	H12
60		92	INT D0	INT D0	92	c1_o_4n	B5
61		93	INT D1	INT D1	93	c1_o_5p	H11
62		94	PROC RST	INT DCLK	91	c1_o_4p	B6
65		97	UID P1	RESERVED	100	c1_o_3n	E9
66		98	UID P2	UID P2	98	c1_o_2n	D8
67		99	UID P3	UID P3	99	c1_o_3p	E10
68		100	RESERVED	UID P1	97	c1_o_2p	E7
71		103	BK2	SEG2	107	c1_o_1n	G9
72		104	BK3	BK3	104	c1_o_0n	H10
73		105	SEG0	SEG1	106	c1_o_1p	G10
74		106	SEG1	BK2	103	c1_o_0p	J10

CHANNEL NAME FORMAT: c1_e_0p = ADC channel 1, even sample, bit 0, positive LVDS

Table 5. Physical Interface Pin Map

APPENDIX B. LOGICAL INTERFACE SCHEMATICS



Logical Interface Top Level Schematic

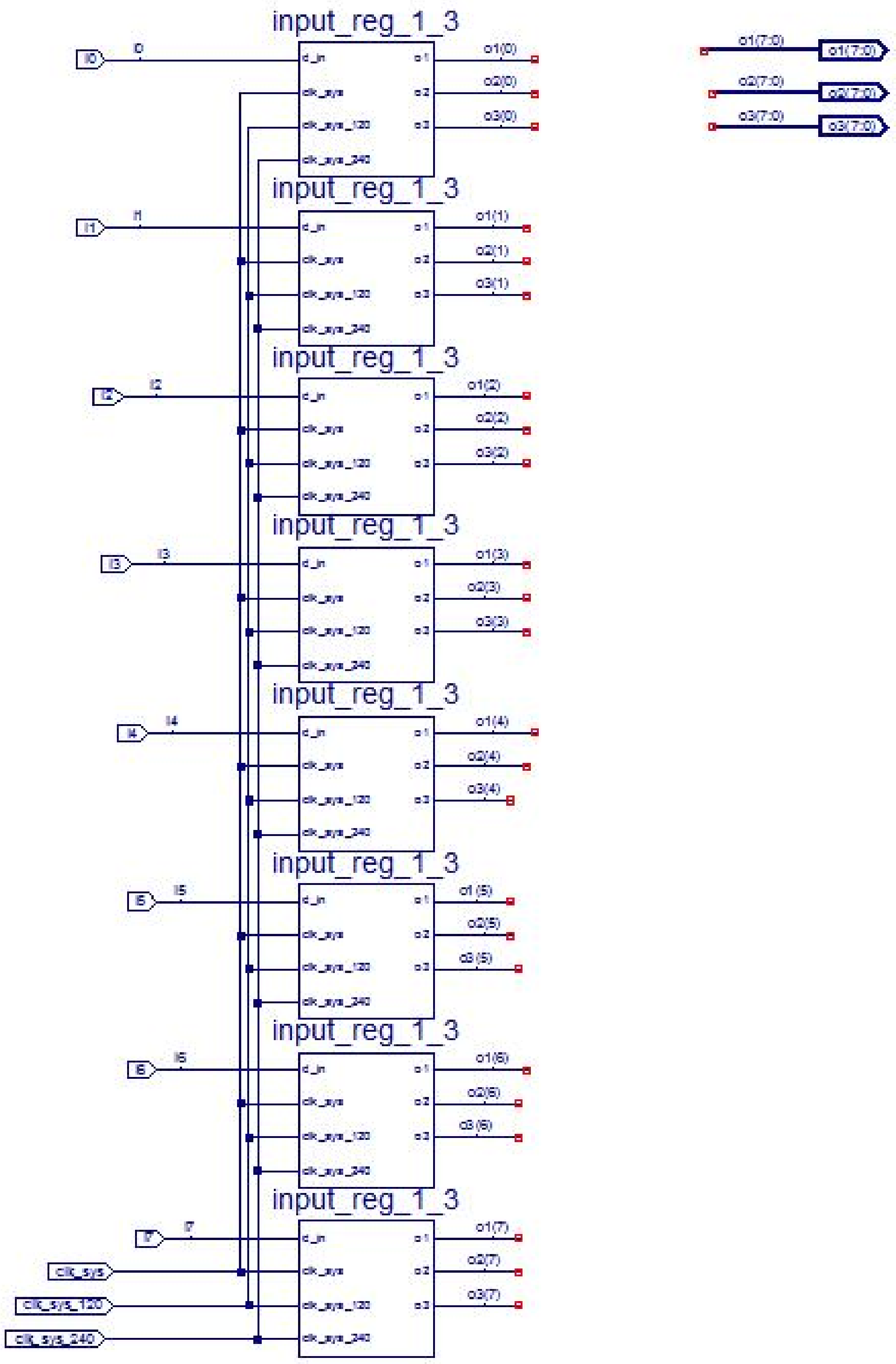


Figure 33. Logical Interface Input Register Block Schematic

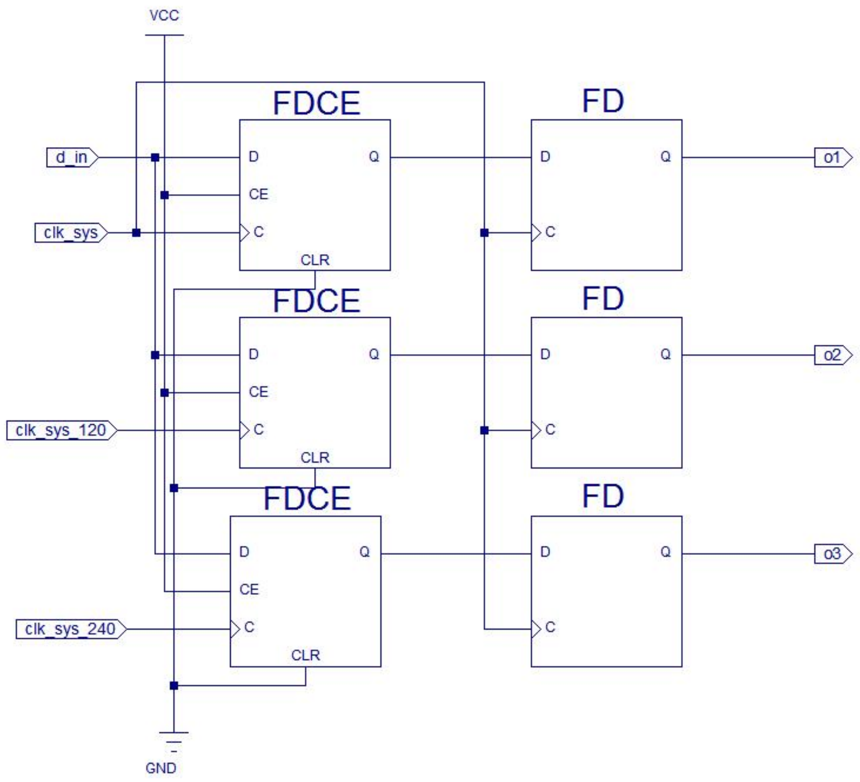


Figure 34. Logical Interface Input Register Schematic

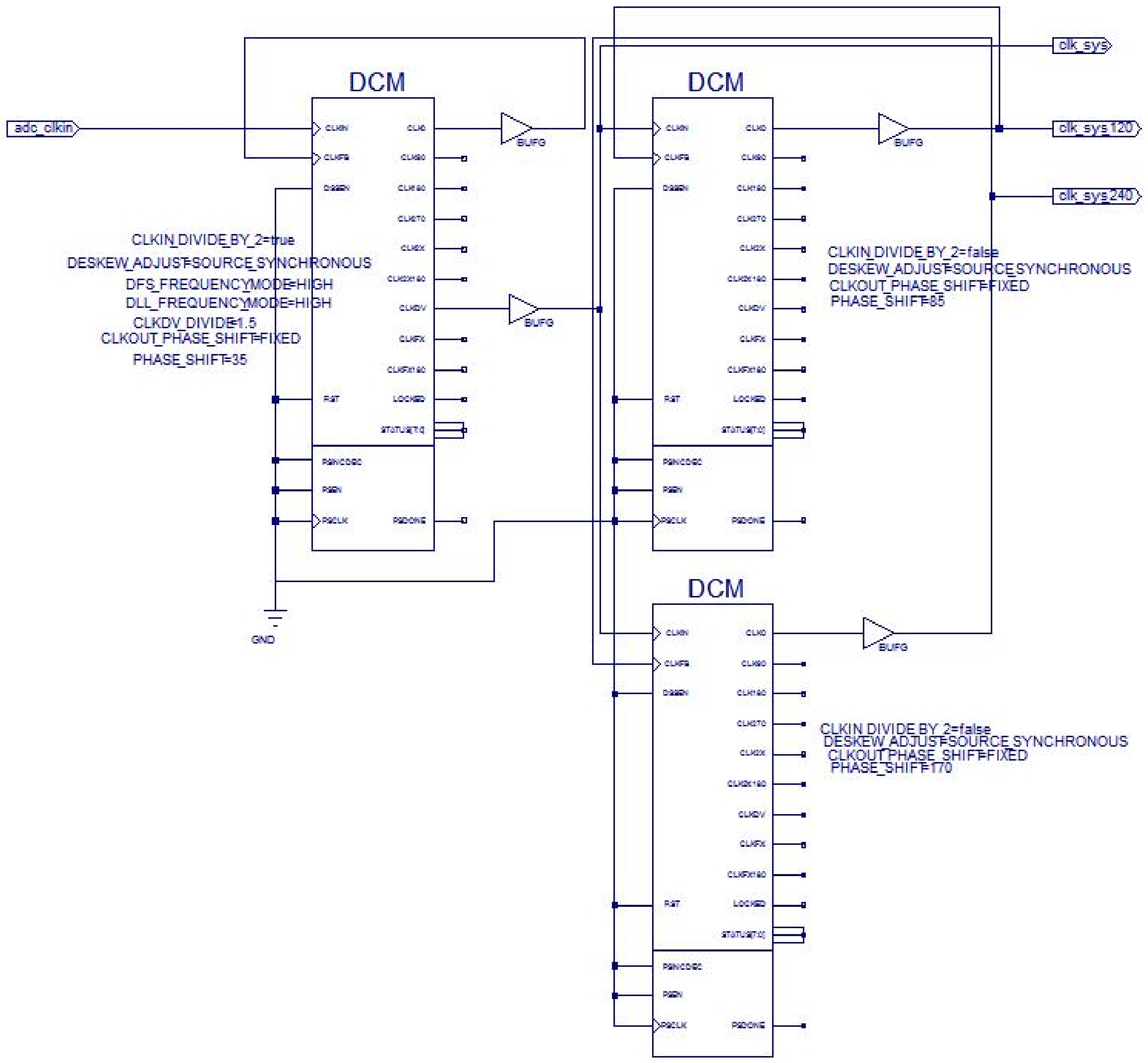


Figure 35. Logical Interface Clock Generator Schematic

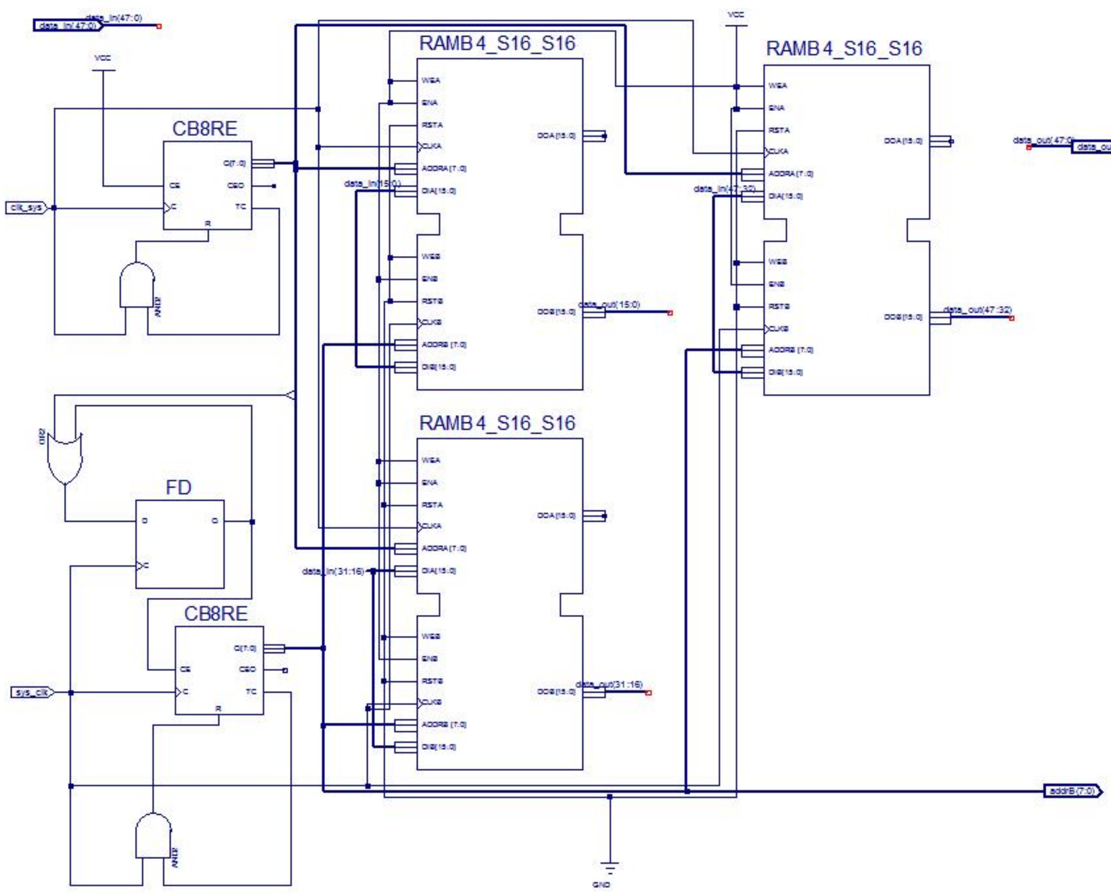


Figure 36. Logical Interface FIFO Schematic

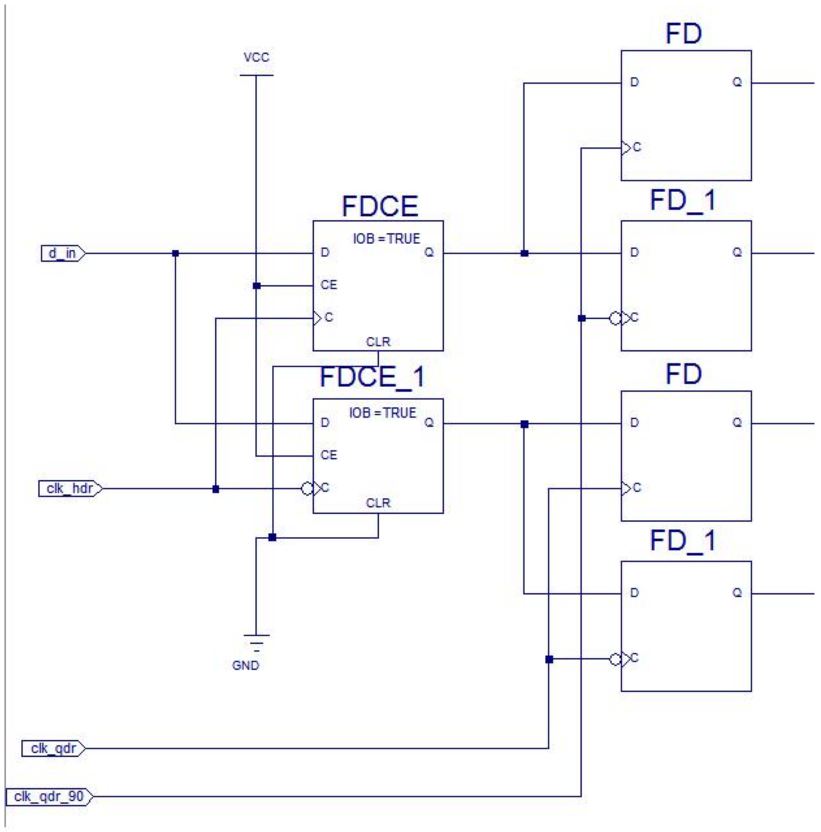


Figure 37. 400 MHz Input Register Schematic

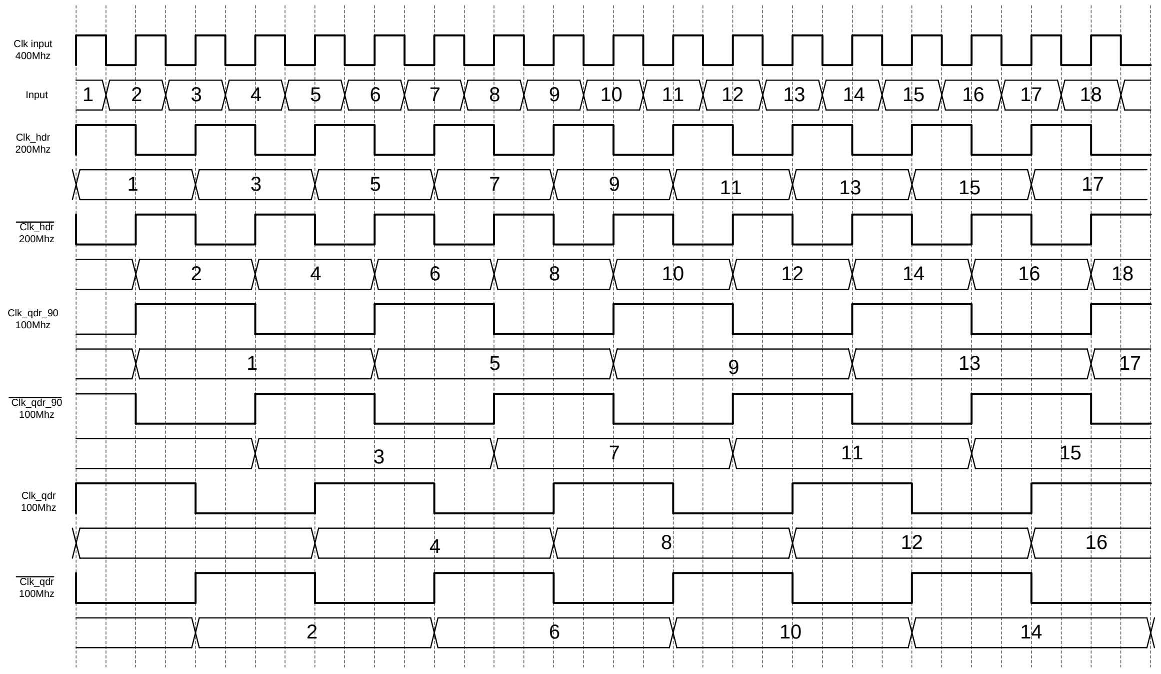


Figure 38. 400 MHz Input Register Timing diagram

APPENDIX C. QMF VHDL CODE

The following VHDL code has been reformatted to remove extra spaces and header files to reduce the size of the code. The filter element entity for a high pass filter is not included, but is identical to the low pass element with different filter coefficients.

```
entity qmf3 is
   Port ( D : in std_logic_vector(47 downto 0);
         Sin : in std_logic_vector(7 downto 0);
         O : out std_logic_vector(48 downto 0);
         CLK : in std_logic);
end qmf3;
architecture Behavioral of qmf3 is
component filter6x6 is
   Port (  D :            in std_logic_vector(107 downto 0);
         i6dly :          in std_logic_vector(3 downto 0);
         i8dly :          in std_logic_vector(3 downto 0);
                 jdly :   in std_logic_vector(3 downto 0);
                          S :                  in std_logic;
         O :  out std_logic_vector(107 downto 0);
         CLK :            in std_logic);
end component;
component bit_drop_18x8 is
   Port ( D : in  STD_LOGIC_VECTOR (17 downto 0);
         Q : out  STD_LOGIC_VECTOR (7 downto 0));
end component;
component bit_ext_8x18 is
   Port ( D : in  STD_LOGIC_VECTOR (7 downto 0);
         Q : out  STD_LOGIC_VECTOR (17 downto 0));
end component;
              signal d0:  STD_LOGIC_VECTOR (7 downto 0);
              signal d1:  STD_LOGIC_VECTOR (7 downto 0);
              signal d2:  STD_LOGIC_VECTOR (7 downto 0);
              signal d3:  STD_LOGIC_VECTOR (7 downto 0);
              signal d4:  STD_LOGIC_VECTOR (7 downto 0);
              signal d5:  STD_LOGIC_VECTOR (7 downto 0);
              signal o0:  STD_LOGIC_VECTOR (7 downto 0);
              signal o1:  STD_LOGIC_VECTOR (7 downto 0);
              signal o2:  STD_LOGIC_VECTOR (7 downto 0);
              signal o3:  STD_LOGIC_VECTOR (7 downto 0);
              signal o4:  STD_LOGIC_VECTOR (7 downto 0);
              signal o5:  STD_LOGIC_VECTOR (7 downto 0);
              signal s1in:          STD_LOGIC_VECTOR (107 downto 0);
              signal s1out:         STD_LOGIC_VECTOR (107 downto 0);
              signal s2out:         STD_LOGIC_VECTOR (107 downto 0);
              signal s3out:         STD_LOGIC_VECTOR (107 downto 0);
              signal s1mux:         STD_LOGIC;
              signal s2mux:         STD_LOGIC;
              signal s3mux:         STD_LOGIC;
begin
              s1mux <= Sin(0);
              s2mux <= Sin(1);
              s3mux <= Sin(2);
              d0 <= D(47 downto 40);
              d1 <= D(39 downto 32);
              d2 <= D(31 downto 24);
              d3 <= D(23 downto 16);
              d4 <= D(15 downto 8);
              d5 <= D(7 downto 0);
              samp0ext : bit_ext_8x18 port map( D => d0, Q => s1in(107 downto 90));
              samp1ext : bit_ext_8x18 port map( D => d1, Q => s1in(89 downto 72));
              samp2ext : bit_ext_8x18 port map( D => d2, Q => s1in(71 downto 54));
              samp3ext : bit_ext_8x18 port map( D => d3, Q => s1in(53 downto 36));
```

```
            samp4ext : bit_ext_8x18 port map( D => d4, Q => s1in(35 downto 18));
            samp5ext : bit_ext_8x18 port map( D => d5, Q => s1in(17 downto 0));
            stage1 : filter6x6 port map(D => s1in, i6dly => "0010", i8dly => "0001",jdly => "0000",S => s1mux , O => s1out, CLK =>
            CLK);
            stage2 : filter6x6 port map(D => s1out, i6dly => "0100", i8dly => "0010",jdly => "0001",S => s2mux , O => s2out, CLK
            => CLK);
            stage3 : filter6x6 port map(D => s2out, i6dly => "1000", i8dly => "0100",jdly => "0011",S => s3mux , O => s3out, CLK
            => CLK);
            samp0drop : bit_drop_18x8 port map( Q => o0, D => s3out(107 downto 90));
            samp1drop : bit_drop_18x8 port map( Q => o1, D => s3out(89 downto 72));
            samp2drop : bit_drop_18x8 port map( Q => o2, D => s3out(71 downto 54));
            samp3drop : bit_drop_18x8 port map( Q => o3, D => s3out(53 downto 36));
            samp4drop : bit_drop_18x8 port map( Q => o4, D => s3out(35 downto 18));
            samp5drop : bit_drop_18x8 port map( Q => o5, D => s3out(17 downto 0));
        O(48) <= s3mux;
            O(47 downto 40) <= o0;
            O(39 downto 32) <= o1;
            O(31 downto 24) <= o2;
            O(23 downto 16) <= o3;
            O(15 downto 8) <= o4;
            O(7 downto 0) <= o5;
end Behavioral;

entity filter6x6 is
   Port (  D :              in std_logic_vector(107 downto 0);
         i6dly :            in std_logic_vector(3 downto 0);
         i8dly :            in std_logic_vector(3 downto 0);
                jdly :      in std_logic_vector(3 downto 0);
                            S :                   in std_logic;
         O :  out std_logic_vector(107 downto 0);
         CLK :              in std_logic);
end filter6x6;
architecture Behavioral of filter6x6 is
 component filter6x3_h is
    Port ( n0 :             in std_logic_vector(17 downto 0);
         n1 :               in std_logic_vector(17 downto 0);
         n2 :               in std_logic_vector(17 downto 0);
         n3 :               in std_logic_vector(17 downto 0);
         n4 :               in std_logic_vector(17 downto 0);
         n5 :               in std_logic_vector(17 downto 0);
         i6dly :            in std_logic_vector(3 downto 0);
         i8dly :            in std_logic_vector(3 downto 0);
         jdly :             in std_logic_vector(3 downto 0);
         o0 :               out std_logic_vector(17 downto 0);
         o1 :               out std_logic_vector(17 downto 0);
         o2 :               out std_logic_vector(17 downto 0);
                            o3 :      out std_logic_vector(17 downto 0);
                            o4 :      out std_logic_vector(17 downto 0);
                            o5 :      out std_logic_vector(17 downto 0);
                            CLK :     in std_logic);
end component;
component filter6x3_l is
    Port ( n0 :             in std_logic_vector(17 downto 0);
         n1 :               in std_logic_vector(17 downto 0);
         n2 :               in std_logic_vector(17 downto 0);
         n3 :               in std_logic_vector(17 downto 0);
         n4 :               in std_logic_vector(17 downto 0);
         n5 :               in std_logic_vector(17 downto 0);
         i6dly :            in std_logic_vector(3 downto 0);
         i8dly :            in std_logic_vector(3 downto 0);
         jdly :             in std_logic_vector(3 downto 0);
         o0 :               out std_logic_vector(17 downto 0);
         o1 :               out std_logic_vector(17 downto 0);
         o2 :               out std_logic_vector(17 downto 0);
                            o3 :      out std_logic_vector(17 downto 0);
                            o4 :      out std_logic_vector(17 downto 0);
                            o5 :      out std_logic_vector(17 downto 0);
                            CLK :     in std_logic);
end component;
component mux108x2 is
```

```
   Port ( D0 :              in std_logic_vector(107 downto 0);
          D1 :              in std_logic_vector(107 downto 0);
          O :  out std_logic_vector(107 downto 0);
          S : in std_logic);
end component;
                signal hout0:           STD_LOGIC_VECTOR (17 downto 0);
                signal hout1:           STD_LOGIC_VECTOR (17 downto 0);
                signal hout2:           STD_LOGIC_VECTOR (17 downto 0);
                signal hout3:           STD_LOGIC_VECTOR (17 downto 0);
                signal hout4:           STD_LOGIC_VECTOR (17 downto 0);
                signal hout5:           STD_LOGIC_VECTOR (17 downto 0);
                signal lout0:           STD_LOGIC_VECTOR (17 downto 0);
                signal lout1:           STD_LOGIC_VECTOR (17 downto 0);
                signal lout2:           STD_LOGIC_VECTOR (17 downto 0);
                signal lout3:           STD_LOGIC_VECTOR (17 downto 0);
                signal lout4:           STD_LOGIC_VECTOR (17 downto 0);
                signal lout5:           STD_LOGIC_VECTOR (17 downto 0);
                signal input0:          STD_LOGIC_VECTOR (17 downto 0);
                signal input1:          STD_LOGIC_VECTOR (17 downto 0);
                signal input2:          STD_LOGIC_VECTOR (17 downto 0);
                signal input3:          STD_LOGIC_VECTOR (17 downto 0);
                signal input4:          STD_LOGIC_VECTOR (17 downto 0);
                signal input5:          STD_LOGIC_VECTOR (17 downto 0);
                signal muxin0:          STD_LOGIC_VECTOR (107 downto 0);
                signal muxin1:          STD_LOGIC_VECTOR (107 downto 0);
begin
                input0 <= D(107 downto 90);
                input1 <= D(89 downto 72);
                input2 <= D(71 downto 54);
                input3 <= D(53 downto 36);
                input4 <= D(35 downto 18);
                input5 <= D(17 downto 0);
                highpass : filter6x3_h port map(
                                n0 => input0,n1 => input1,n2 => input2,n3 => input3,n4 =>  input4,n5 => input5,
                                i6dly => i6dly,i8dly => i8dly,jdly =>jdly,
                                o0 => hout0,o1 => hout1,o2 => hout2,o3 => hout3,o4 => hout4,o5 => hout5,
                                CLK => CLK);
                lowpass : filter6x3_l port map(
                                n0 => input0,n1 => input1,n2 => input2,n3 => input3,n4 =>  input4,n5 => input5,
                                i6dly => i6dly,i8dly => i8dly,jdly => jdly,
                                o0 => lout0,o1 => lout1,o2 => lout2,o3 => lout3,o4 => lout4,o5 => lout5,
                                CLK => CLK);
                muxin0(107 downto 90) <= hout0;
                muxin0(89 downto 72) <= hout1;
                muxin0(71 downto 54) <= hout2;
                muxin0(53 downto 36) <= hout3;
                muxin0(35 downto 18) <= hout4;
                muxin0(17 downto 0) <= hout5;
                muxin1(107 downto 90) <= lout0;
                muxin1(89 downto 72) <= lout1;
                muxin1(71 downto 54) <= lout2;
                muxin1(53 downto 36) <= lout3;
                muxin1(35 downto 18) <= lout4;
                muxin1(17 downto 0) <= lout5;
                outmux  : mux108x2 port map ( D0 => muxin0, D1 =>muxin1,O => O,S => S);
end Behavioral;

entity filter6x3_l is
   Port ( n0 :              in std_logic_vector(17 downto 0);
          n1 :              in std_logic_vector(17 downto 0);
          n2 :              in std_logic_vector(17 downto 0);
          n3 :              in std_logic_vector(17 downto 0);
          n4 :              in std_logic_vector(17 downto 0);
          n5 :              in std_logic_vector(17 downto 0);
          i6dly :           in std_logic_vector(3 downto 0);
          i8dly :           in std_logic_vector(3 downto 0);
          jdly :            in std_logic_vector(3 downto 0);
          o0 :              out std_logic_vector(17 downto 0);
          o1 :              out std_logic_vector(17 downto 0);
          o2 :              out std_logic_vector(17 downto 0);
```

```
				o3 :		out std_logic_vector(17 downto 0);
				o4 :		out std_logic_vector(17 downto 0);
				o5 :		out std_logic_vector(17 downto 0);
				CLK :		in std_logic);
end filter6x3_l;
architecture Behavioral of filter6x3_l is
component Mult18x18_var_hld is
				port(	A, B:			in std_logic_vector(17 downto 0);
						Dly:				in std_logic_vector(3 downto 0);
						CLK, RST, CE:		in std_logic;
						P:				out std_logic_vector(35 downto 0));
end component;
component add_36x36_var_hld is
   Port (	A :			in  STD_LOGIC_VECTOR (35 downto 0);
		B :			in  STD_LOGIC_VECTOR (35 downto 0);
		S :			out  STD_LOGIC_VECTOR (35 downto 0);
		Dly :	in  STD_LOGIC_VECTOR (3 downto 0);
		CLK :	in  STD_LOGIC);
end component;
component var_dly18 is
   Port (	Din : in  STD_LOGIC_VECTOR (17 downto 0);
				Dly : in  STD_LOGIC_VECTOR (3 downto 0);
				Clk : in  STD_LOGIC;
		Qout : out  STD_LOGIC_VECTOR (17 downto 0));
end component;
		signal xn0c1:		STD_LOGIC_VECTOR (35 downto 0);
		signal xn0c3:		STD_LOGIC_VECTOR (35 downto 0);
		signal xn0c5:		STD_LOGIC_VECTOR (35 downto 0);
		signal xn1c2:		STD_LOGIC_VECTOR (35 downto 0);
		signal xn1c4:		STD_LOGIC_VECTOR (35 downto 0);
		signal xn1c6:		STD_LOGIC_VECTOR (35 downto 0);
		signal xn2c1:		STD_LOGIC_VECTOR (35 downto 0);
		signal xn2c3:		STD_LOGIC_VECTOR (35 downto 0);
		signal xn2c5:		STD_LOGIC_VECTOR (35 downto 0);
		signal xn3c2:		STD_LOGIC_VECTOR (35 downto 0);
		signal xn3c4:		STD_LOGIC_VECTOR (35 downto 0);
		signal xn3c6:		STD_LOGIC_VECTOR (35 downto 0);
		signal xn4c1:		STD_LOGIC_VECTOR (35 downto 0);
		signal xn4c3:		STD_LOGIC_VECTOR (35 downto 0);
		signal xn4c5:		STD_LOGIC_VECTOR (35 downto 0);
		signal xn5c2:		STD_LOGIC_VECTOR (35 downto 0);
		signal xn5c4:		STD_LOGIC_VECTOR (35 downto 0);
		signal xn5c6:		STD_LOGIC_VECTOR (35 downto 0);
		signal i0:	STD_LOGIC_VECTOR (35 downto 0);
		signal i1:	STD_LOGIC_VECTOR (35 downto 0);
		signal i2:	STD_LOGIC_VECTOR (35 downto 0);
		signal i3:	STD_LOGIC_VECTOR (35 downto 0);
		signal i4:	STD_LOGIC_VECTOR (35 downto 0);
		signal i5:	STD_LOGIC_VECTOR (35 downto 0);
		signal i6:	STD_LOGIC_VECTOR (35 downto 0);
		signal i7:	STD_LOGIC_VECTOR (35 downto 0);
		signal i8:	STD_LOGIC_VECTOR (35 downto 0);
		signal j0:	STD_LOGIC_VECTOR (35 downto 0);
		signal j1:	STD_LOGIC_VECTOR (35 downto 0);
		signal j2:	STD_LOGIC_VECTOR (35 downto 0);
		signal k0:	STD_LOGIC_VECTOR (35 downto 0);
		signal k1:	STD_LOGIC_VECTOR (35 downto 0);
		signal k2:	STD_LOGIC_VECTOR (35 downto 0);
begin
		mult_n0_c1 : Mult18x18_var_hld port map( P => xn0c1, A => n0, B => "111111111101010110", Dly => "0000", CLK =>
		CLK, CE => '1', RST => '0');
		mult_n0_c3 : Mult18x18_var_hld port map( P => xn0c3, A => n0, B => "111111100101110010", Dly => "0000", CLK =>
		CLK, CE => '1', RST => '0');
		mult_n0_c5 : Mult18x18_var_hld port map( P => xn0c5, A => n0, B => "111111100101110010", Dly => "0000", CLK =>
		CLK, CE => '1', RST => '0');
		mult_n1_c2 : Mult18x18_var_hld port map( P => xn1c2, A => n1, B => "111111110101010110", Dly => "0000", CLK =>
		CLK, CE => '1', RST => '0');
		mult_n1_c4 : Mult18x18_var_hld port map( P => xn1c4, A => n1, B => "011111011010001110", Dly => "0000", CLK =>
		CLK, CE => '1', RST => '0');
		mult_n1_c6 : Mult18x18_var_hld port map( P => xn1c6, A => n1, B => "111111110101010110", Dly => "0000", CLK =>
		CLK, CE => '1', RST => '0');
		mult_n2_c1 : Mult18x18_var_hld port map( P => xn2c1, A => n2, B => "111111111101010110", Dly => "0000", CLK =>
		CLK, CE => '1', RST => '0');
		mult_n2_c3 : Mult18x18_var_hld port map( P => xn2c3, A => n2, B => "111111100101110010", Dly => "0000", CLK =>
		CLK, CE => '1', RST => '0');
		mult_n2_c5 : Mult18x18_var_hld port map( P => xn2c5, A => n2, B => "111111100101110010", Dly => "0000", CLK =>
		CLK, CE => '1', RST => '0');
		mult_n3_c2 : Mult18x18_var_hld port map( P => xn3c2, A => n3, B => "111111110101010110", Dly => "0000", CLK =>
		CLK, CE => '1', RST => '0');
		mult_n3_c4 : Mult18x18_var_hld port map( P => xn3c4, A => n3, B => "011111011010001110", Dly => "0000", CLK =>
		CLK, CE => '1', RST => '0');
		mult_n3_c6 : Mult18x18_var_hld port map( P => xn3c6, A => n3, B => "111111110101010110", Dly => "0000", CLK =>
		CLK, CE => '1', RST => '0');
		mult_n4_c1 : Mult18x18_var_hld port map( P => xn4c1, A => n4, B => "111111111101010110", Dly => "0000", CLK =>
		CLK, CE => '1', RST => '0');
		mult_n4_c3 : Mult18x18_var_hld port map( P => xn4c3, A => n4, B => "111111100101110010", Dly => "0000", CLK =>
		CLK, CE => '1', RST => '0');
		mult_n4_c5 : Mult18x18_var_hld port map( P => xn4c5, A => n4, B => "111111100101110010", Dly => "0000", CLK =>
		CLK, CE => '1', RST => '0');
		mult_n5_c2 : Mult18x18_var_hld port map( P => xn5c2, A => n5, B => "111111110101010110", Dly => "0000", CLK =>
		CLK, CE => '1', RST => '0');
		mult_n5_c4 : Mult18x18_var_hld port map( P => xn5c4, A => n5, B => "011111011010001110", Dly => "0000", CLK =>
		CLK, CE => '1', RST => '0');
		mult_n5_c6 : Mult18x18_var_hld port map( P => xn5c6, A => n5, B => "111111110101010110", Dly => "0000", CLK =>
		CLK, CE => '1', RST => '0');
		add_i0	: add_36x36_var_hld port map( S => i0, A => xn0c1, B => xn1c2, Dly => "0000", CLK => CLK);
		add_i1	: add_36x36_var_hld port map( S => i1, A => xn0c3, B => xn1c4, Dly => "0000", CLK => CLK);
		add_i2	: add_36x36_var_hld port map( S => i2, A => xn0c5, B => xn1c6, Dly => "0000", CLK => CLK);
		add_i3	: add_36x36_var_hld port map( S => i3, A => xn2c1, B => xn3c2, Dly => "0000", CLK => CLK);
		add_i4	: add_36x36_var_hld port map( S => i4, A => xn2c3, B => xn3c4, Dly => "0000", CLK => CLK);
		add_i5	: add_36x36_var_hld port map( S => i5, A => xn2c5, B => xn3c6, Dly => "0000", CLK => CLK);
		add_i6	: add_36x36_var_hld port map( S => i6, A => xn4c1, B => xn5c2, Dly => i6dly, CLK => CLK);
		add_i7	: add_36x36_var_hld port map( S => i7, A => xn4c3, B => xn5c4, Dly => "0000", CLK => CLK);
		add_i8	: add_36x36_var_hld port map( S => i8, A => xn4c5, B => xn5c6, Dly => i8dly, CLK => CLK);
		add_j0	: add_36x36_var_hld port map( S => j0, A => i1, B => i5, Dly => jdly, CLK => CLK);
		add_j1	: add_36x36_var_hld port map( S => j1, A => i0, B => i4, Dly => jdly, CLK => CLK);
		add_j2	: add_36x36_var_hld port map( S => j2, A => i3, B => i7, Dly => jdly, CLK => CLK);
		add_k0	: add_36x36_var_hld port map( S => k0, A => i2, B => j2, Dly => i8dly, CLK => CLK);
		add_k1	: add_36x36_var_hld port map( S => k1, A => j0, B => i6, Dly => i8dly, CLK => CLK);
		add_k2	: add_36x36_var_hld port map( S => k2, A => j1, B => i8, Dly => i8dly, CLK => CLK);
		o_hold0	: var_dly18 port map( Qout => o0, Din => k1(34 downto 17), Dly => jdly, CLK => CLK);
		o_hold1	: var_dly18 port map( Qout => o1, Din => k2(34 downto 17), Dly => jdly, CLK => CLK);
		o_hold2	: var_dly18 port map( Qout => o2, Din => k0(34 downto 17), Dly => jdly, CLK => CLK);
		o3 <= k1(34 downto 17);
		o4 <= k2(34 downto 17);
		o5 <= k0(34 downto 17);
end Behavioral;

entity bit_drop_18x8 is
    Port ( D : in  STD_LOGIC_VECTOR (17 downto 0);
           Q : out  STD_LOGIC_VECTOR (7 downto 0));
end bit_drop_18x8;
architecture Behavioral of bit_drop_18x8 is
begin
		Q(7 downto 0) <= D(17 downto 10);
end Behavioral;

entity bit_ext_8x18 is
    Port ( D : in  STD_LOGIC_VECTOR (7 downto 0);
           Q : out  STD_LOGIC_VECTOR (17 downto 0));
end bit_ext_8x18;
architecture Behavioral of bit_ext_8x18 is
begin
   Q(17) <= not D(7);
   Q(16 downto 10) <= D(6 downto 0);
   Q(9) <= '0';
   Q(8) <= '0';
   Q(7) <= '0';
   Q(6) <= '0';
  Q(5) <= '0';
  Q(4) <= '0';
  Q(3) <= '0';
  Q(2) <= '0';
  Q(1) <= '0';
  Q(0) <= '0';
end Behavioral;
entity mux108x2 is
   Port ( D0 :        in std_logic_vector(107 downto 0);
        D1 :          in std_logic_vector(107 downto 0);
        O :  out std_logic_vector(107 downto 0);
        S : in std_logic);
end mux108x2;
architecture Behavioral of mux108x2 is
begin
                  process (D0, D1, S)
                  begin
                           case S is
                                    when '0' => O <= D0;
                                    when '1' => O <= D1;
                                    when others => NULL;
                           end case;
                  end process;
end Behavioral;

entity add_36x36_var_hld is
   Port ( A : in  STD_LOGIC_VECTOR (35 downto 0);
        B : in  STD_LOGIC_VECTOR (35 downto 0);
        S : out  STD_LOGIC_VECTOR (35 downto 0);
        Dly : in  STD_LOGIC_VECTOR (3 downto 0);
        CLK : in  STD_LOGIC);
end add_36x36_var_hld;
architecture Behavioral of add_36x36_var_hld is
component add_36x36
  port(
                                    Ain : in  STD_LOGIC_VECTOR (35 downto 0);
                                    Bin : in  STD_LOGIC_VECTOR (35 downto 0);
                                    Sout : out  STD_LOGIC_VECTOR (35 downto 0));
end component;
component var_dly36 is
   Port (  Din : in  STD_LOGIC_VECTOR (35 downto 0);
                                    Dly : in  STD_LOGIC_VECTOR (3 downto 0);
                                    Clk : in  STD_LOGIC;
         Qout : out  STD_LOGIC_VECTOR (35 downto 0));

end component;
signal s_wire: std_logic_vector (35 downto 0);
begin
inst_add_36x36 : add_36x36 port map( Ain => A, Bin => B, Sout => s_wire);
inst_var_dly36 : var_dly36 port map( Din => s_wire, Dly => Dly, Clk => Clk, Qout => S);
end Behavioral;

entity add_36x36 is
   Port ( Ain : in  STD_LOGIC_VECTOR (35 downto 0);
        Bin : in  STD_LOGIC_VECTOR (35 downto 0);
        Sout : out  STD_LOGIC_VECTOR (35 downto 0));
end add_36x36;
architecture Behavioral of add_36x36 is
begin
         Sout <= Ain + Bin;
end Behavioral;
entity var_dly36 is
   Port (  Din : in  STD_LOGIC_VECTOR (35 downto 0);
                                    Dly : in  STD_LOGIC_VECTOR (3 downto 0);
                                    Clk : in  STD_LOGIC;
         Qout : out  STD_LOGIC_VECTOR (35 downto 0));
end var_dly36;
architecture Behavioral of var_dly36 is
component SRL16
  port(
    Q                        :            out   STD_ULOGIC;
    D                        :            in    STD_ULOGIC;
    CLK                        :          in    STD_ULOGIC;
                         A0                       : in    STD_ULOGIC;
                         A1                       : in    STD_ULOGIC;
                         A2                       : in    STD_ULOGIC;
    A3                        :           in    STD_ULOGIC);
end component;
begin
REG_S0  : SRL16 port map(Q => Qout(0)  , CLK => CLK, D => Din(0)   , A0 => Dly(0)              , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S1  : SRL16 port map(Q => Qout(1)  , CLK => CLK, D => Din(1)   , A0 => Dly(0)              , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S2  : SRL16 port map(Q => Qout(2)  , CLK => CLK, D => Din(2)   , A0 => Dly(0)              , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S3  : SRL16 port map(Q => Qout(3)  , CLK => CLK, D => Din(3)   , A0 => Dly(0)              , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S4  : SRL16 port map(Q => Qout(4)  , CLK => CLK, D => Din(4)   , A0 => Dly(0)              , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S5  : SRL16 port map(Q => Qout(5)  , CLK => CLK, D => Din(5)   , A0 => Dly(0)              , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S6  : SRL16 port map(Q => Qout(6)  , CLK => CLK, D => Din(6)   , A0 => Dly(0)              , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S7  : SRL16 port map(Q => Qout(7)  , CLK => CLK, D => Din(7)   , A0 => Dly(0)              , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S8  : SRL16 port map(Q => Qout(8)  , CLK => CLK, D => Din(8)   , A0 => Dly(0)              , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S9  : SRL16 port map(Q => Qout(9)  , CLK => CLK, D => Din(9)   , A0 => Dly(0)              , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S10 : SRL16 port map(Q => Qout(10) , CLK => CLK, D => Din(10)  , A0 => Dly(0)           , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S11 : SRL16 port map(Q => Qout(11) , CLK => CLK, D => Din(11)  , A0 => Dly(0)           , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S12 : SRL16 port map(Q => Qout(12) , CLK => CLK, D => Din(12)  , A0 => Dly(0)           , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S13 : SRL16 port map(Q => Qout(13) , CLK => CLK, D => Din(13)  , A0 => Dly(0)           , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S14 : SRL16 port map(Q => Qout(14) , CLK => CLK, D => Din(14)  , A0 => Dly(0)           , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S15 : SRL16 port map(Q => Qout(15) , CLK => CLK, D => Din(15)  , A0 => Dly(0)           , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S16 : SRL16 port map(Q => Qout(16) , CLK => CLK, D => Din(16)  , A0 => Dly(0)           , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S17 : SRL16 port map(Q => Qout(17) , CLK => CLK, D => Din(17)  , A0 => Dly(0)           , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S18 : SRL16 port map(Q => Qout(18) , CLK => CLK, D => Din(18)  , A0 => Dly(0)           , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S19 : SRL16 port map(Q => Qout(19) , CLK => CLK, D => Din(19)  , A0 => Dly(0)           , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S20 : SRL16 port map(Q => Qout(20) , CLK => CLK, D => Din(20)  , A0 => Dly(0)           , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S21 : SRL16 port map(Q => Qout(21) , CLK => CLK, D => Din(21)  , A0 => Dly(0)           , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S22 : SRL16 port map(Q => Qout(22) , CLK => CLK, D => Din(22)  , A0 => Dly(0)           , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S23 : SRL16 port map(Q => Qout(23) , CLK => CLK, D => Din(23)  , A0 => Dly(0)           , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S24 : SRL16 port map(Q => Qout(24) , CLK => CLK, D => Din(24)  , A0 => Dly(0)           , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S25 : SRL16 port map(Q => Qout(25) , CLK => CLK, D => Din(25)  , A0 => Dly(0)           , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S26 : SRL16 port map(Q => Qout(26) , CLK => CLK, D => Din(26)  , A0 => Dly(0)           , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S27 : SRL16 port map(Q => Qout(27) , CLK => CLK, D => Din(27)  , A0 => Dly(0)           , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S28 : SRL16 port map(Q => Qout(28) , CLK => CLK, D => Din(28)  , A0 => Dly(0)           , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S29 : SRL16 port map(Q => Qout(29) , CLK => CLK, D => Din(29)  , A0 => Dly(0)           , A1 => Dly(1)           ,    A2    =>
Dly(2)       , A3 => Dly(3));
```

```
REG_S30 : SRL16 port map(Q => Qout(30) , CLK => CLK, D => Din(30)  , A0 => Dly(0)            , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S31 : SRL16 port map(Q => Qout(31) , CLK => CLK, D => Din(31)  , A0 => Dly(0)            , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S32 : SRL16 port map(Q => Qout(32) , CLK => CLK, D => Din(32)  , A0 => Dly(0)            , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S33 : SRL16 port map(Q => Qout(33) , CLK => CLK, D => Din(33)  , A0 => Dly(0)            , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S34 : SRL16 port map(Q => Qout(34) , CLK => CLK, D => Din(34)  , A0 => Dly(0)            , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S35 : SRL16 port map(Q => Qout(35) , CLK => CLK, D => Din(35)  , A0 => Dly(0)            , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
end Behavioral;

entity var_dly18 is
   Port (  Din : in  STD_LOGIC_VECTOR (17 downto 0);
                                                          Dly : in  STD_LOGIC_VECTOR (3 downto 0);
                                                          Clk : in  STD_LOGIC;
          Qout : out  STD_LOGIC_VECTOR (17 downto 0));
end var_dly18;
architecture Behavioral of var_dly18 is
component SRL16
  port(
            Q                    :  out   STD_ULOGIC;
            D                    :  in    STD_ULOGIC;
            CLK                    :          in    STD_ULOGIC;
                    A0                    : in    STD_ULOGIC;
                    A1                    : in    STD_ULOGIC;
                    A2                    : in    STD_ULOGIC;
            A3                    : in    STD_ULOGIC);
end component;
begin
REG_S0  : SRL16 port map(Q => Qout(0)  , CLK => CLK, D => Din(0)   , A0 => Dly(0)              , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S1  : SRL16 port map(Q => Qout(1)  , CLK => CLK, D => Din(1)   , A0 => Dly(0)              , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S2  : SRL16 port map(Q => Qout(2)  , CLK => CLK, D => Din(2)   , A0 => Dly(0)              , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S3  : SRL16 port map(Q => Qout(3)  , CLK => CLK, D => Din(3)   , A0 => Dly(0)              , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S4  : SRL16 port map(Q => Qout(4)  , CLK => CLK, D => Din(4)   , A0 => Dly(0)              , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S5  : SRL16 port map(Q => Qout(5)  , CLK => CLK, D => Din(5)   , A0 => Dly(0)              , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S6  : SRL16 port map(Q => Qout(6)  , CLK => CLK, D => Din(6)   , A0 => Dly(0)              , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S7  : SRL16 port map(Q => Qout(7)  , CLK => CLK, D => Din(7)   , A0 => Dly(0)              , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S8  : SRL16 port map(Q => Qout(8)  , CLK => CLK, D => Din(8)   , A0 => Dly(0)              , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S9  : SRL16 port map(Q => Qout(9)  , CLK => CLK, D => Din(9)   , A0 => Dly(0)              , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S10 : SRL16 port map(Q => Qout(10) , CLK => CLK, D => Din(10)  , A0 => Dly(0)            , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S11 : SRL16 port map(Q => Qout(11) , CLK => CLK, D => Din(11)  , A0 => Dly(0)            , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S12 : SRL16 port map(Q => Qout(12) , CLK => CLK, D => Din(12)  , A0 => Dly(0)            , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S13 : SRL16 port map(Q => Qout(13) , CLK => CLK, D => Din(13)  , A0 => Dly(0)            , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S14 : SRL16 port map(Q => Qout(14) , CLK => CLK, D => Din(14)  , A0 => Dly(0)            , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S15 : SRL16 port map(Q => Qout(15) , CLK => CLK, D => Din(15)  , A0 => Dly(0)            , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S16 : SRL16 port map(Q => Qout(16) , CLK => CLK, D => Din(16)  , A0 => Dly(0)            , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
REG_S17 : SRL16 port map(Q => Qout(17) , CLK => CLK, D => Din(17)  , A0 => Dly(0)            , A1 => Dly(1)          ,    A2    =>
Dly(2)       , A3 => Dly(3));
end Behavioral;
```

```
entity Mult18x18_var_hld is
        port(     A, B:                                             in std_logic_vector(17 downto 0);
                      Dly:                                                  in std_logic_vector(3 downto 0);
                      CLK, RST, CE:                  in std_logic;
                      P:                                                    out std_logic_vector(35 downto 0));
end Mult18x18_var_hld;
architecture Behavioral of Mult18x18_var_hld is
component FDR
  port(
    Q              :        out   STD_ULOGIC;
    D              :        in    STD_ULOGIC;
    C              :        in    STD_ULOGIC;
    R              :        in    STD_ULOGIC);
end component;
component SRL16
  port(
    Q              :        out   STD_ULOGIC;
    D              :        in    STD_ULOGIC;
    CLK             :       in    STD_ULOGIC;
               A0            : in    STD_ULOGIC;
               A1            : in    STD_ULOGIC;
               A2            : in    STD_ULOGIC;
    A3              :       in    STD_ULOGIC);
end component;
component MULT18X18S
   port (A        : in      STD_LOGIC_VECTOR (17 downto 0);
        B         : in      STD_LOGIC_VECTOR (17 downto 0);
        C         : in      STD_ULOGIC ;
        CE   : in   STD_ULOGIC ;
                      P              : out STD_LOGIC_VECTOR (35 downto 0);
        R    : in   STD_ULOGIC );
end component;
signal a_wire0: STD_ULOGIC ;
signal a_wire1: STD_ULOGIC ;
signal a_wire2: STD_ULOGIC ;
signal a_wire3: STD_ULOGIC ;
signal a_wire4: STD_ULOGIC ;
signal a_wire5: STD_ULOGIC ;
signal a_wire6: STD_ULOGIC ;
signal a_wire7: STD_ULOGIC ;
signal a_wire8: STD_ULOGIC ;
signal a_wire9: STD_ULOGIC ;
signal a_wire10: STD_ULOGIC ;
signal a_wire11: STD_ULOGIC ;
signal a_wire12: STD_ULOGIC ;
signal a_wire13: STD_ULOGIC ;
signal a_wire14: STD_ULOGIC ;
signal a_wire15: STD_ULOGIC ;
signal a_wire16: STD_ULOGIC ;
signal a_wire17: STD_ULOGIC ;
signal b_wire0: STD_ULOGIC ;
signal b_wire1: STD_ULOGIC ;
signal b_wire2: STD_ULOGIC ;
signal b_wire3: STD_ULOGIC ;
signal b_wire4: STD_ULOGIC ;
signal b_wire5: STD_ULOGIC ;
signal b_wire6: STD_ULOGIC ;
signal b_wire7: STD_ULOGIC ;
signal b_wire8: STD_ULOGIC ;
signal b_wire9: STD_ULOGIC ;
signal b_wire10: STD_ULOGIC ;
signal b_wire11: STD_ULOGIC ;
signal b_wire12: STD_ULOGIC ;
signal b_wire13: STD_ULOGIC ;
signal b_wire14: STD_ULOGIC ;
signal b_wire15: STD_ULOGIC ;
signal b_wire16: STD_ULOGIC ;
signal b_wire17: STD_ULOGIC ;
signal p_wire0: STD_ULOGIC ;
signal p_wire1: STD_ULOGIC ;
signal p_wire2: STD_ULOGIC ;
signal p_wire3: STD_ULOGIC ;
signal p_wire4: STD_ULOGIC ;
signal p_wire5: STD_ULOGIC ;
signal p_wire6: STD_ULOGIC ;
signal p_wire7: STD_ULOGIC ;
signal p_wire8: STD_ULOGIC ;
signal p_wire9: STD_ULOGIC ;
signal p_wire10: STD_ULOGIC ;
signal p_wire11: STD_ULOGIC ;
signal p_wire12: STD_ULOGIC ;
signal p_wire13: STD_ULOGIC ;
signal p_wire14: STD_ULOGIC ;
signal p_wire15: STD_ULOGIC ;
signal p_wire16: STD_ULOGIC ;
signal p_wire17: STD_ULOGIC ;
signal p_wire18: STD_ULOGIC ;
signal p_wire19: STD_ULOGIC ;
signal p_wire20: STD_ULOGIC ;
signal p_wire21: STD_ULOGIC ;
signal p_wire22: STD_ULOGIC ;
signal p_wire23: STD_ULOGIC ;
signal p_wire24: STD_ULOGIC ;
signal p_wire25: STD_ULOGIC ;
signal p_wire26: STD_ULOGIC ;
signal p_wire27: STD_ULOGIC ;
signal p_wire28: STD_ULOGIC ;
signal p_wire29: STD_ULOGIC ;
signal p_wire30: STD_ULOGIC ;
signal p_wire31: STD_ULOGIC ;
signal p_wire32: STD_ULOGIC ;
signal p_wire33: STD_ULOGIC ;
signal p_wire34: STD_ULOGIC ;
signal p_wire35: STD_ULOGIC ;
attribute RLOC : string;
attribute RLOC of REG_A0 : label is "X0Y0" ;
attribute RLOC of REG_A1 : label is "X0Y0" ;
attribute RLOC of REG_A2 : label is "X0Y1" ;
attribute RLOC of REG_A3 : label is "X0Y1" ;
attribute RLOC of REG_A4 : label is "X0Y2" ;
attribute RLOC of REG_A5 : label is "X0Y2" ;
attribute RLOC of REG_A6 : label is "X0Y3" ;
attribute RLOC of REG_A7 : label is "X0Y3" ;
attribute RLOC of REG_A8 : label is "X0Y4" ;
attribute RLOC of REG_A9 : label is "X0Y4" ;
attribute RLOC of REG_A10: label is "X0Y5" ;
attribute RLOC of REG_A11: label is "X0Y5" ;
attribute RLOC of REG_A12: label is "X0Y6" ;
attribute RLOC of REG_A13: label is "X0Y6" ;
attribute RLOC of REG_A14: label is "X0Y7" ;
attribute RLOC of REG_A15: label is "X0Y7" ;
attribute RLOC of REG_A16: label is "X-1Y7";
attribute RLOC of REG_A17: label is "X-1Y7";
attribute RLOC of REG_B0 : label is "X2Y0" ;
attribute RLOC of REG_B1 : label is "X2Y0" ;
attribute RLOC of REG_B2 : label is "X2Y1" ;
attribute RLOC of REG_B3 : label is "X2Y1" ;
attribute RLOC of REG_B4 : label is "X2Y2" ;
attribute RLOC of REG_B5 : label is "X2Y2" ;
attribute RLOC of REG_B6 : label is "X2Y3" ;
attribute RLOC of REG_B7 : label is "X2Y3" ;
attribute RLOC of REG_B8 : label is "X2Y4" ;
attribute RLOC of REG_B9 : label is "X2Y4" ;
attribute RLOC of REG_B10: label is "X2Y5" ;
attribute RLOC of REG_B11: label is "X2Y5" ;
attribute RLOC of REG_B12: label is "X2Y6" ;
attribute RLOC of REG_B13: label is "X2Y6" ;
attribute RLOC of REG_B14: label is "X2Y7" ;
attribute RLOC of REG_B15: label is "X2Y7" ;
attribute RLOC of REG_B16: label is "X-1Y6";
attribute RLOC of REG_B17: label is "X-1Y6";
attribute RLOC of REG_P0 : label is "X-2Y0";
attribute RLOC of REG_P1 : label is "X1Y0" ;
attribute RLOC of REG_P2 : label is "X1Y0" ;
attribute RLOC of REG_P3 : label is "X1Y1" ;
attribute RLOC of REG_P4 : label is "X1Y1" ;
attribute RLOC of REG_P5 : label is "X3Y0" ;
attribute RLOC of REG_P6 : label is "X3Y0" ;
attribute RLOC of REG_P7 : label is "X3Y1" ;
attribute RLOC of REG_P8 : label is "X-2Y2";
attribute RLOC of REG_P9 : label is "X1Y2" ;
attribute RLOC of REG_P10: label is "X1Y2" ;
attribute RLOC of REG_P11: label is "X1Y3" ;
attribute RLOC of REG_P12: label is "X1Y3" ;
attribute RLOC of REG_P13: label is "X3Y2" ;
attribute RLOC of REG_P14: label is "X3Y2" ;
attribute RLOC of REG_P15: label is "X3Y3" ;
attribute RLOC of REG_P16: label is "X-2Y4";
attribute RLOC of REG_P17: label is "X1Y4" ;
attribute RLOC of REG_P18: label is "X1Y4" ;
attribute RLOC of REG_P19: label is "X1Y5" ;
attribute RLOC of REG_P20: label is "X1Y5" ;
attribute RLOC of REG_P21: label is "X3Y4" ;
attribute RLOC of REG_P22: label is "X3Y4" ;
attribute RLOC of REG_P23: label is "X3Y5" ;
attribute RLOC of REG_P24: label is "X-2Y6";
attribute RLOC of REG_P25: label is "X1Y6" ;
attribute RLOC of REG_P26: label is "X1Y6" ;
attribute RLOC of REG_P27: label is "X1Y7" ;
attribute RLOC of REG_P28: label is "X1Y7" ;
attribute RLOC of REG_P29: label is "X3Y6" ;
attribute RLOC of REG_P30: label is "X3Y6" ;
attribute RLOC of REG_P31: label is "X3Y7" ;
attribute RLOC of REG_P32: label is "X3Y1" ;
attribute RLOC of REG_P33: label is "X3Y3" ;
attribute RLOC of REG_P34: label is "X3Y5" ;
attribute RLOC of REG_P35: label is "X3Y7" ;
attribute BEL : string;
attribute BEL of REG_A0 : label is "FFX" ;
attribute BEL of REG_A1 : label is "FFY" ;
attribute BEL of REG_A2 : label is "FFX" ;
attribute BEL of REG_A3 : label is "FFY" ;
attribute BEL of REG_A4 : label is "FFX" ;
attribute BEL of REG_A5 : label is "FFY" ;
attribute BEL of REG_A6 : label is "FFX" ;
attribute BEL of REG_A7 : label is "FFY" ;
attribute BEL of REG_A8 : label is "FFX" ;
attribute BEL of REG_A9 : label is "FFY" ;
attribute BEL of REG_A10: label is "FFX" ;
attribute BEL of REG_A11: label is "FFY" ;
attribute BEL of REG_A12: label is "FFX" ;
attribute BEL of REG_A13: label is "FFY" ;
attribute BEL of REG_A14: label is "FFX" ;
attribute BEL of REG_A15: label is "FFY" ;
attribute BEL of REG_A16: label is "FFX" ;
attribute BEL of REG_A17: label is "FFY" ;
attribute BEL of REG_B0 : label is "FFX" ;
attribute BEL of REG_B1 : label is "FFY" ;
attribute BEL of REG_B2 : label is "FFX" ;
attribute BEL of REG_B3 : label is "FFY" ;
attribute BEL of REG_B4 : label is "FFX" ;
attribute BEL of REG_B5 : label is "FFY" ;
attribute BEL of REG_B6 : label is "FFX" ;
attribute BEL of REG_B7 : label is "FFY" ;
attribute BEL of REG_B8 : label is "FFX" ;
attribute BEL of REG_B9 : label is "FFY" ;
attribute BEL of REG_B10: label is "FFX" ;
attribute BEL of REG_B11: label is "FFY" ;
attribute BEL of REG_B12: label is "FFX" ;
attribute BEL of REG_B13: label is "FFY" ;
attribute BEL of REG_B14: label is "FFX" ;
attribute BEL of REG_B15: label is "FFY" ;
attribute BEL of REG_B16: label is "FFX" ;
attribute BEL of REG_B17: label is "FFY" ;
attribute BEL of REG_P0 : label is "G" ;
attribute BEL of REG_P1 : label is "F" ;
attribute BEL of REG_P2 : label is "G" ;
attribute BEL of REG_P3 : label is "F" ;
attribute BEL of REG_P4 : label is "G" ;
attribute BEL of REG_P5 : label is "F" ;
attribute BEL of REG_P6 : label is "G" ;
attribute BEL of REG_P7 : label is "F" ;
attribute BEL of REG_P8 : label is "G" ;
attribute BEL of REG_P9 : label is "F" ;
attribute BEL of REG_P10: label is "G" ;
attribute BEL of REG_P11: label is "F" ;
attribute BEL of REG_P12: label is "G" ;
attribute BEL of REG_P13: label is "F" ;
attribute BEL of REG_P14: label is "G" ;
attribute BEL of REG_P15: label is "F" ;
attribute BEL of REG_P16: label is "G" ;
attribute BEL of REG_P17: label is "F" ;
attribute BEL of REG_P18: label is "G" ;
attribute BEL of REG_P19: label is "F" ;
attribute BEL of REG_P20: label is "G" ;
attribute BEL of REG_P21: label is "F" ;
attribute BEL of REG_P22: label is "G" ;
attribute BEL of REG_P23: label is "F" ;
attribute BEL of REG_P24: label is "G" ;
attribute BEL of REG_P25: label is "F" ;
attribute BEL of REG_P26: label is "G" ;
attribute BEL of REG_P27: label is "F" ;
attribute BEL of REG_P28: label is "G" ;
attribute BEL of REG_P29: label is "F" ;
attribute BEL of REG_P30: label is "G" ;
attribute BEL of REG_P31: label is "G" ;
attribute BEL of REG_P32: label is "G" ;
attribute BEL of REG_P33: label is "G" ;
attribute BEL of REG_P34: label is "G" ;
attribute BEL of REG_P35: label is "F" ;
attribute MAXDELAY : string;
attribute MAXDELAY of a_wire0: signal is "500 ps";
attribute MAXDELAY of a_wire1: signal is "500 ps";
attribute MAXDELAY of a_wire2: signal is "500 ps";
attribute MAXDELAY of a_wire3: signal is "500 ps";
attribute MAXDELAY of a_wire4: signal is "500 ps";
attribute MAXDELAY of a_wire5: signal is "500 ps";
attribute MAXDELAY of a_wire6: signal is "500 ps";
attribute MAXDELAY of a_wire7: signal is "500 ps";
attribute MAXDELAY of a_wire8: signal is "500 ps";
attribute MAXDELAY of a_wire9: signal is "500 ps";
attribute MAXDELAY of a_wire10: signal is "500 ps";
attribute MAXDELAY of a_wire11: signal is "500 ps";
attribute MAXDELAY of a_wire12: signal is "500 ps";
attribute MAXDELAY of a_wire13: signal is "500 ps";
attribute MAXDELAY of a_wire14: signal is "500 ps";
attribute MAXDELAY of a_wire15: signal is "500 ps";
attribute MAXDELAY of a_wire16: signal is "500 ps";
attribute MAXDELAY of a_wire17: signal is "500 ps";
attribute MAXDELAY of b_wire0: signal is "500 ps";
attribute MAXDELAY of b_wire1: signal is "500 ps";
attribute MAXDELAY of b_wire2: signal is "500 ps";
attribute MAXDELAY of b_wire3: signal is "500 ps";
attribute MAXDELAY of b_wire4: signal is "500 ps";
attribute MAXDELAY of b_wire5: signal is "500 ps";
attribute MAXDELAY of b_wire6: signal is "500 ps";
attribute MAXDELAY of b_wire7: signal is "500 ps";
attribute MAXDELAY of b_wire8: signal is "500 ps";
attribute MAXDELAY of b_wire9: signal is "500 ps";
attribute MAXDELAY of b_wire10: signal is "500 ps";
attribute MAXDELAY of b_wire11: signal is "500 ps";
attribute MAXDELAY of b_wire12: signal is "500 ps";
attribute MAXDELAY of b_wire13: signal is "500 ps";
attribute MAXDELAY of b_wire14: signal is "500 ps";
attribute MAXDELAY of b_wire15: signal is "500 ps";
attribute MAXDELAY of b_wire16: signal is "500 ps";
attribute MAXDELAY of b_wire17: signal is "500 ps";
attribute MAXDELAY of p_wire0: signal is "500 ps";
attribute MAXDELAY of p_wire1: signal is "500 ps";
attribute MAXDELAY of p_wire2: signal is "500 ps";
attribute MAXDELAY of p_wire3: signal is "500 ps";
attribute MAXDELAY of p_wire4: signal is "500 ps";
attribute MAXDELAY of p_wire5: signal is "500 ps";
attribute MAXDELAY of p_wire6: signal is "500 ps";
attribute MAXDELAY of p_wire7: signal is "500 ps";
attribute MAXDELAY of p_wire8: signal is "500 ps";
attribute MAXDELAY of p_wire9: signal is "500 ps";
attribute MAXDELAY of p_wire10: signal is "500 ps";
attribute MAXDELAY of p_wire11: signal is "500 ps";
attribute MAXDELAY of p_wire12: signal is "500 ps";
attribute MAXDELAY of p_wire13: signal is "500 ps";
attribute MAXDELAY of p_wire14: signal is "500 ps";
attribute MAXDELAY of p_wire15: signal is "500 ps";
attribute MAXDELAY of p_wire16: signal is "500 ps";
attribute MAXDELAY of p_wire17: signal is "500 ps";
attribute MAXDELAY of p_wire18: signal is "500 ps";
attribute MAXDELAY of p_wire19: signal is "500 ps";
attribute MAXDELAY of p_wire20: signal is "500 ps";
attribute MAXDELAY of p_wire21: signal is "500 ps";
attribute MAXDELAY of p_wire22: signal is "500 ps";
attribute MAXDELAY of p_wire23: signal is "500 ps";
attribute MAXDELAY of p_wire24: signal is "500 ps";
attribute MAXDELAY of p_wire25: signal is "500 ps";
attribute MAXDELAY of p_wire26: signal is "500 ps";
attribute MAXDELAY of p_wire27: signal is "500 ps";
attribute MAXDELAY of p_wire28: signal is "500 ps";
attribute MAXDELAY of p_wire29: signal is "500 ps";
attribute MAXDELAY of p_wire30: signal is "500 ps";
attribute MAXDELAY of p_wire31: signal is "500 ps";
attribute MAXDELAY of p_wire32: signal is "500 ps";
attribute MAXDELAY of p_wire33: signal is "500 ps";
attribute MAXDELAY of p_wire34: signal is "500 ps";
attribute MAXDELAY of p_wire35: signal is "500 ps";
begin
REG_A0  : FDR port map(Q => a_wire0   , C => CLK, D => A(0)  , R => RST);
REG_A1  : FDR port map(Q => a_wire1   , C => CLK, D => A(1)  , R => RST);
REG_A2  : FDR port map(Q => a_wire2   , C => CLK, D => A(2)  , R => RST);
REG_A3  : FDR port map(Q => a_wire3   , C => CLK, D => A(3)  , R => RST);
REG_A4  : FDR port map(Q => a_wire4   , C => CLK, D => A(4)  , R => RST);
REG_A5  : FDR port map(Q => a_wire5   , C => CLK, D => A(5)  , R => RST);
REG_A6  : FDR port map(Q => a_wire6   , C => CLK, D => A(6)  , R => RST);
REG_A7  : FDR port map(Q => a_wire7   , C => CLK, D => A(7)  , R => RST);
REG_A8  : FDR port map(Q => a_wire8   , C => CLK, D => A(8)  , R => RST);
REG_A9  : FDR port map(Q => a_wire9   , C => CLK, D => A(9)  , R => RST);
REG_A10 : FDR port map(Q => a_wire10  , C => CLK, D => A(10) , R => RST);
REG_A11 : FDR port map(Q => a_wire11  , C => CLK, D => A(11) , R => RST);
REG_A12 : FDR port map(Q => a_wire12  , C => CLK, D => A(12) , R => RST);
REG_A13 : FDR port map(Q => a_wire13  , C => CLK, D => A(13) , R => RST);
REG_A14 : FDR port map(Q => a_wire14  , C => CLK, D => A(14) , R => RST);
REG_A15 : FDR port map(Q => a_wire15  , C => CLK, D => A(15) , R => RST);
REG_A16 : FDR port map(Q => a_wire16  , C => CLK, D => A(16) , R => RST);
REG_A17 : FDR port map(Q => a_wire17  , C => CLK, D => A(17) , R => RST);
REG_B0  : FDR port map(Q => b_wire0   , C => CLK, D => B(0)  , R => RST);
REG_B1  : FDR port map(Q => b_wire1   , C => CLK, D => B(1)  , R => RST);
REG_B2  : FDR port map(Q => b_wire2   , C => CLK, D => B(2)  , R => RST);
REG_B3  : FDR port map(Q => b_wire3   , C => CLK, D => B(3)  , R => RST);
REG_B4  : FDR port map(Q => b_wire4   , C => CLK, D => B(4)  , R => RST);
REG_B5  : FDR port map(Q => b_wire5   , C => CLK, D => B(5)  , R => RST);
REG_B6  : FDR port map(Q => b_wire6   , C => CLK, D => B(6)  , R => RST);
REG_B7  : FDR port map(Q => b_wire7   , C => CLK, D => B(7)  , R => RST);
```

```
REG_B8  : FDR port map(Q => b_wire8   , C => CLK, D => B(8)  , R => RST);
REG_B9  : FDR port map(Q => b_wire9   , C => CLK, D => B(9)  , R => RST);
REG_B10 : FDR port map(Q => b_wire10  , C => CLK, D => B(10) , R => RST);
REG_B11 : FDR port map(Q => b_wire11  , C => CLK, D => B(11) , R => RST);
REG_B12 : FDR port map(Q => b_wire12  , C => CLK, D => B(12) , R => RST);
REG_B13 : FDR port map(Q => b_wire13  , C => CLK, D => B(13) , R => RST);
REG_B14 : FDR port map(Q => b_wire14  , C => CLK, D => B(14) , R => RST);
REG_B15 : FDR port map(Q => b_wire15  , C => CLK, D => B(15) , R => RST);
REG_B16 : FDR port map(Q => b_wire16  , C => CLK, D => B(16) , R => RST);
REG_B17 : FDR port map(Q => b_wire17  , C => CLK, D => B(17) , R => RST);
inst_mult18x18s : MULT18X18S port map(
        P(0) => p_wire0,   P(1) => p_wire1,   P(2) => p_wire2,   P(3) => p_wire3,   P(4) => p_wire4,   P(5) => p_wire5,
        P(6) => p_wire6,   P(7) => p_wire7,   P(8) => p_wire8,   P(9) => p_wire9,   P(10) => p_wire10, P(11) => p_wire11,
        P(12) => p_wire12, P(13) => p_wire13, P(14) => p_wire14, P(15) => p_wire15, P(16) => p_wire16, P(17) => p_wire17,
        P(18) => p_wire18, P(19) => p_wire19, P(20) => p_wire20, P(21) => p_wire21, P(22) => p_wire22, P(23) => p_wire23,
        P(24) => p_wire24, P(25) => p_wire25, P(26) => p_wire26, P(27) => p_wire27, P(28) => p_wire28, P(29) => p_wire29,
        P(30) => p_wire30, P(31) => p_wire31, P(32) => p_wire32, P(33) => p_wire33, P(34) => p_wire34, P(35) => p_wire35,
        (0) => a_wire0,   A(1) => a_wire1,   A(2) => a_wire2,   A(3) => a_wire3,   A(4) => a_wire4,   A(5) => a_wire5,
        A(6) => a_wire6,   A(7) => a_wire7,   A(8) => a_wire8,   A(9) => a_wire9,   A(10) => a_wire10, A(11) => a_wire11,
        A(12) => a_wire12, A(13) => a_wire13, A(14) => a_wire14, A(15) => a_wire15, A(16) => a_wire16, A(17) => a_wire17,
        B(0) => b_wire0,   B(1) => b_wire1,   B(2) => b_wire2,   B(3) => b_wire3,   B(4) => b_wire4,   B(5) => b_wire5,
        B(6) => b_wire6,   B(7) => b_wire7,   B(8) => b_wire8,   B(9) => b_wire9,   B(10) => b_wire10, B(11) => b_wire11,
        B(12) => b_wire12, B(13) => b_wire13, B(14) => b_wire14, B(15) => b_wire15, B(16) => b_wire16, B(17) => b_wire17,
        C => CLK, CE => CE, R => RST);
REG_P0  : SRL16 port map(Q => P(0)  , CLK => CLK, D => p_wire0   , A0 => Dly(0)   ,  A1  =>  Dly(1),  A2  =>  Dly(2),  A3  =>
Dly(3));
REG_P1  : SRL16 port map(Q => P(1)  , CLK => CLK, D => p_wire1   , A0 => Dly(0)   ,  A1  =>  Dly(1),  A2  =>  Dly(2),  A3  =>
Dly(3));
REG_P2  : SRL16 port map(Q => P(2)  , CLK => CLK, D => p_wire2   , A0 => Dly(0)   ,  A1  =>  Dly(1),  A2  =>  Dly(2),  A3  =>
Dly(3));
REG_P3  : SRL16 port map(Q => P(3)  , CLK => CLK, D => p_wire3   , A0 => Dly(0)   ,  A1  =>  Dly(1),  A2  =>  Dly(2),  A3  =>
Dly(3));
REG_P4  : SRL16 port map(Q => P(4)  , CLK => CLK, D => p_wire4   , A0 => Dly(0)   ,  A1  =>  Dly(1),  A2  =>  Dly(2),  A3  =>
Dly(3));
REG_P5  : SRL16 port map(Q => P(5)  , CLK => CLK, D => p_wire5   , A0 => Dly(0)   ,  A1  =>  Dly(1),  A2  =>  Dly(2),  A3  =>
Dly(3));
REG_P6  : SRL16 port map(Q => P(6)  , CLK => CLK, D => p_wire6   , A0 => Dly(0)   ,  A1  =>  Dly(1),  A2  =>  Dly(2),  A3  =>
Dly(3));
REG_P7  : SRL16 port map(Q => P(7)  , CLK => CLK, D => p_wire7   , A0 => Dly(0)   ,  A1  =>  Dly(1),  A2  =>  Dly(2),  A3  =>
Dly(3));
REG_P8  : SRL16 port map(Q => P(8)  , CLK => CLK, D => p_wire8   , A0 => Dly(0)   ,  A1  =>  Dly(1),  A2  =>  Dly(2),  A3  =>
Dly(3));
REG_P9  : SRL16 port map(Q => P(9)  , CLK => CLK, D => p_wire9   , A0 => Dly(0)   ,  A1  =>  Dly(1),  A2  =>  Dly(2),  A3  =>
Dly(3));
REG_P10 : SRL16 port map(Q => P(10) , CLK => CLK, D => p_wire10   , A0 => Dly(0), A1 => Dly(1), A2 => Dly(2), A3 =>
Dly(3));
REG_P11 : SRL16 port map(Q => P(11) , CLK => CLK, D => p_wire11   , A0 => Dly(0), A1 => Dly(1), A2 => Dly(2), A3 =>
Dly(3));
REG_P12 : SRL16 port map(Q => P(12) , CLK => CLK, D => p_wire12   , A0 => Dly(0), A1 => Dly(1), A2 => Dly(2), A3 =>
Dly(3));
REG_P13 : SRL16 port map(Q => P(13) , CLK => CLK, D => p_wire13   , A0 => Dly(0), A1 => Dly(1), A2 => Dly(2), A3 =>
Dly(3));
REG_P14 : SRL16 port map(Q => P(14) , CLK => CLK, D => p_wire14   , A0 => Dly(0), A1 => Dly(1), A2 => Dly(2), A3 =>
Dly(3));
REG_P15 : SRL16 port map(Q => P(15) , CLK => CLK, D => p_wire15   , A0 => Dly(0), A1 => Dly(1), A2 => Dly(2), A3 =>
Dly(3));
REG_P16 : SRL16 port map(Q => P(16) , CLK => CLK, D => p_wire16   , A0 => Dly(0), A1 => Dly(1), A2 => Dly(2), A3 =>
Dly(3));
REG_P17 : SRL16 port map(Q => P(17) , CLK => CLK, D => p_wire17   , A0 => Dly(0), A1 => Dly(1), A2 => Dly(2), A3 =>
Dly(3));
REG_P18 : SRL16 port map(Q => P(18) , CLK => CLK, D => p_wire18   , A0 => Dly(0), A1 => Dly(1), A2 => Dly(2), A3 =>
Dly(3));
REG_P19 : SRL16 port map(Q => P(19) , CLK => CLK, D => p_wire19   , A0 => Dly(0), A1 => Dly(1), A2 => Dly(2), A3 =>
Dly(3));
REG_P20 : SRL16 port map(Q => P(20) , CLK => CLK, D => p_wire20   , A0 => Dly(0), A1 => Dly(1), A2 => Dly(2), A3 =>
Dly(3));
REG_P21 : SRL16 port map(Q => P(21) , CLK => CLK, D => p_wire21  , A0 => Dly(0),  A1  =>  Dly(1),  A2  =>  Dly(2),  A3  =>
Dly(3));
REG_P22 : SRL16 port map(Q => P(22) , CLK => CLK, D => p_wire22  , A0 => Dly(0),  A1  =>  Dly(1),  A2  =>  Dly(2),  A3  =>
Dly(3));
```

```
REG_P23 : SRL16 port map(Q => P(23) , CLK => CLK, D => p_wire23  , A0 => Dly(0) ,   A1   =>   Dly(1),   A2   =>   Dly(2),   A3   =>
Dly(3));
REG_P24 : SRL16 port map(Q => P(24) , CLK => CLK, D => p_wire24  , A0 => Dly(0) ,   A1   =>   Dly(1),   A2   =>   Dly(2),   A3   =>
Dly(3));
REG_P25 : SRL16 port map(Q => P(25) , CLK => CLK, D => p_wire25  , A0 => Dly(0) ,   A1   =>   Dly(1),   A2   =>   Dly(2),   A3   =>
Dly(3));
REG_P26 : SRL16 port map(Q => P(26) , CLK => CLK, D => p_wire26  , A0 => Dly(0) ,   A1   =>   Dly(1),   A2   =>   Dly(2),   A3   =>
Dly(3));
REG_P27 : SRL16 port map(Q => P(27) , CLK => CLK, D => p_wire27  , A0 => Dly(0) ,   A1   =>   Dly(1),   A2   =>   Dly(2),   A3   =>
Dly(3));
REG_P28 : SRL16 port map(Q => P(28) , CLK => CLK, D => p_wire28  , A0 => Dly(0) ,   A1   =>   Dly(1),   A2   =>   Dly(2),   A3   =>
Dly(3));
REG_P29 : SRL16 port map(Q => P(29) , CLK => CLK, D => p_wire29  , A0 => Dly(0) ,   A1   =>   Dly(1),   A2   =>   Dly(2),   A3   =>
Dly(3));
REG_P30 : SRL16 port map(Q => P(30) , CLK => CLK, D => p_wire30  , A0 => Dly(0) ,   A1   =>   Dly(1),   A2   =>   Dly(2),   A3   =>
Dly(3));
REG_P31 : SRL16 port map(Q => P(31) , CLK => CLK, D => p_wire31  , A0 => Dly(0) ,   A1   =>   Dly(1),   A2   =>   Dly(2),   A3   =>
Dly(3));
REG_P32 : SRL16 port map(Q => P(32) , CLK => CLK, D => p_wire32  , A0 => Dly(0) ,   A1   =>   Dly(1),   A2   =>   Dly(2),   A3   =>
Dly(3));
REG_P33 : SRL16 port map(Q => P(33) , CLK => CLK, D => p_wire33  , A0 => Dly(0) ,   A1   =>   Dly(1),   A2   =>   Dly(2),   A3   =>
Dly(3));
REG_P34 : SRL16 port map(Q => P(34) , CLK => CLK, D => p_wire34  , A0 => Dly(0) ,   A1   =>   Dly(1),   A2   =>   Dly(2),   A3   =>
Dly(3));
REG_P35 : SRL16 port map(Q => P(35) , CLK => CLK, D => p_wire35  , A0 => Dly(0) ,   A1   =>   Dly(1),   A2   =>   Dly(2),   A3   =>
Dly(3));

end Behavioral;
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D.   C CODE FILTER

The code in this section is representative of all the code used for various portions of the project. The main.c code returns output files containing either filtered or unfiltered data collected by the SRC-6. The unfiltered data is the same data that was processed by the internal filter to produce the filtered data. Data is returned to the main function packed into 64-bit words, which are then unpacked into individual variable that are printed in the output files with six values on each line separated by commas. This format is easily imported into any spreadsheet or math program. Since the filtered data is a signed value, the code determines the sign bit and extends it if required. The function t6c in the first loop of the t6d.mc code block is the call to the VHDL macro that captures the data, and in this case, returns both the unfiltered data and data filtered by the VHDL high pass filter. The C code version of the high pass filter is applied to the unfiltered output and a total of three outputs are returned to the main function.

************************MAIN.C CODE BLOCK*************************

```
static char const cvsid[] = "$Id: main.c,v 2.1 2005/06/14 22:16:48 jls Exp $";
#include <libmap.h>
#include <stdlib.h>
void t6d (int64_t A[], int64_t B[], int64_t C[], int64_t *time, int mapnum);
int main (int argc, char *argv[]) {
   FILE *raw, *vhd, *ccode;
    if (argc < 4) {
                                        fprintf (stderr, "USAGE: t6dexe <raw data ouput file> <vhdl ouput file> <c code ouput file>\n");
                                        exit (1);
        }

    if ((raw = fopen (argv[1], "w")) == NULL) {
        fprintf (stderr, "failed to open raw data ouput file\n");
        exit (1);
        }
    if ((vhd = fopen (argv[2], "w")) == NULL) {
        fprintf (stderr, "failed to open vhdl data ouput file\n");
        exit (1);
        }
    if ((ccode = fopen (argv[3], "w")) == NULL) {
        fprintf (stderr, "failed to open c code data ouput file\n");
        exit (1);
        }
    int i;
    int64_t *A, *B, *C;
    int64_t tm, tm2;
    int mapnum = 0;
    int o1[10000];
    int o2[10000];
    int o3[10000];
    int o4[10000];
    int o5[10000];
    int o6[10000];
    int o7[10000];
    int o8[10000];
    int r1[10000];
    int r2[10000];
    int r3[10000];
    int r4[10000];
    int r5[10000];
    int r6[10000];
    int r7[10000];
    int r8[10000];
    int c1[10000];
    int c2[10000];
    int c3[10000];
    int c4[10000];
    int c5[10000];
    int c6[10000];
    int c7[10000];
    int c8[10000];

    A = (int64_t*) malloc (MAX_OBM_SIZE * sizeof (int64_t));
    B = (int64_t*) malloc (MAX_OBM_SIZE * sizeof (int64_t));
    C = (int64_t*) malloc (MAX_OBM_SIZE * sizeof (int64_t));
    map_allocate (1);

    t6d (A, B, C, &tm, mapnum);

    printf ("%lld clocks\n", tm);

    for (i=0; i<10000; i++) {
                        o3[i]=(A[i] >> 40) & 0x00000000000000ff;
                            if (o3[i]>128) o3[i]=o3[i] | 0xffffffffffffff00;
                        o4[i]=(A[i] >> 32) & 0x00000000000000ff;
                            if (o4[i]>128) o4[i]=o4[i] | 0xffffffffffffff00;
                        o5[i]=(A[i] >> 24) & 0x00000000000000ff;
                            if (o5[i]>128) o5[i]=o5[i] | 0xffffffffffffff00;
                        o6[i]=(A[i] >> 16) & 0x00000000000000ff;
                            if (o6[i]>128) o6[i]=o6[i] | 0xffffffffffffff00;
                        o7[i]=(A[i] >> 8) & 0x00000000000000ff;
                            if (o7[i]>128) o7[i]=o7[i] | 0xffffffffffffff00;
                        o8[i]=A[i] & 0x00000000000000ff;
                            if (o8[i]>128) o8[i]=o8[i] | 0xffffffffffffff00;


                        r3[i]=(B[i] >> 40) & 0x00000000000000ff;
                        r4[i]=(B[i] >> 32) & 0x00000000000000ff;
                        r5[i]=(B[i] >> 24) & 0x00000000000000ff;
                        r6[i]=(B[i] >> 16) & 0x00000000000000ff;
                        r7[i]=(B[i] >> 8) & 0x00000000000000ff;
                        r8[i]=B[i] & 0x00000000000000ff;

                        c3[i]=(C[i] >> 40) & 0x00000000000000ff;
                        c4[i]=(C[i] >> 32) & 0x00000000000000ff;
                        c5[i]=(C[i] >> 24) & 0x00000000000000ff;
                        c6[i]=(C[i] >> 16) & 0x00000000000000ff;
                        c7[i]=(C[i] >> 8) & 0x00000000000000ff;
                        c8[i]=C[i] & 0x00000000000000ff;
                }
                for (i=0; i<10000; i++) {
                         fprintf (vhd, "%d,%d,%d,%d,%d,%d,%d,%d\n", i,o3[i],o4[i],o5[i],o6[i],o7[i],o8[i],o2[i]);
                }
                for (i=0; i<10000; i++) {
                        fprintf (raw, "%d,%d,%d,%d,%d,%d,%d\n", i,r3[i],r4[i],r5[i],r6[i],r7[i],r8[i]);
                }
                for (i=0; i<10000; i++) {
                          fprintf (ccode, "%d,%d,%d,%d,%d,%d,%d\n", i,c3[i],c4[i],c5[i],c6[i],c7[i],c8[i]);
                }
    map_free (1);
    exit(0);
    }

        *************************T6D.MC CODE BLOCK*************************
#include <libmap.h>
```

```
void t6d (int64_t A[], int64_t B[], int64_t C[],int64_t *time, int mapnum) {
   OBM_BANK_A (AL, int64_t, MAX_OBM_SIZE)
   OBM_BANK_B (BL, int64_t, MAX_OBM_SIZE)
   OBM_BANK_C (CL, int64_t, MAX_OBM_SIZE)
   int64_t t0, t1, out, outraw;
   int i, out2;
   int8_t s0,s1,s2,s3,s4,s5,s6,s7;
   int8_t pad=0;
   int16_t n0=0;
   int16_t n1=0;
   int16_t n2=0;
   int16_t n3=0;
   int16_t n4=0;
   int16_t n5=0;
   int n0c1=0;
   int n0c3=0;
   int n0c5=0;
   int n1c2=0;
   int n1c4=0;
   int n1c6=0;
   int n2c1=0;
   int n2c3=0;
   int n2c5=0;
   int n3c2=0;
   int n3c4=0;
   int n3c6=0;
   int n4c1=0;
   int n4c3=0;
   int n4c5=0;
   int n5c2=0;
   int n5c4=0;
   int n5c6=0;
   int i0=0;
   int i1=0;
   int i2=0;
   int i3=0;
   int i4=0;
   int i5=0;
   int i6=0;
   int i6t1=0;
   int i6t2=0;
   int i7=0;
   int i8=0;
   int i8t1=0;
   int j0=0;
   int j1=0;
   int j2=0;
   int k0=0;
   int k1=0;
   int k2=0;
   int k3=0;
   int k4=0;
   int k5=0;
   int8_t k0t=0;
   int8_t k1t=0;
   int8_t k2t=0;
   int8_t k3t=0;
   int8_t k4t=0;
   int8_t k5t=0;
   int kd01,kd02,kd03,kd11,kd12,kd13,kd21,kd22,kd23,kd31,kd32,kd33,kd41,kd42,kd43,kd51,kd52,kd53;

   read_timer (&t0);

   for (i=0; i<MAX_OBM_SIZE; i++)
   {
                              t6c(&out, &outraw);
                              AL[i] =  out;
                              BL[i] =  outraw;
   }
   for (i=0; i<MAX_OBM_SIZE; i++)
   {
                              split_64to8(BL[i], &s6, &s7, &s0, &s1, &s2, &s3, &s4, &s5);

     k3 = k0;
     k4 = k1;
     k5 = k2;

     k0 = i2 + j2;
     k1 = i6t2 + j0;
     k2 = i8t1 + j1;

     split_32to8(k0, &k0t, &kd01, &kd02, &kd03);
     split_32to8(k1, &k1t, &kd11, &kd12, &kd13);
     split_32to8(k2, &k2t, &kd21, &kd22, &kd23);
     split_32to8(k3, &k3t, &kd31, &kd32, &kd33);
     split_32to8(k4, &k4t, &kd41, &kd42, &kd43);
     split_32to8(k5, &k5t, &kd51, &kd52, &kd53);

     comb_8to64(pad, pad, k1t,k2t,k0t,k4t,k5t,k3t, &CL[i]);

     j0 = i1 + i5;
     j1 = i0 + i4;
     j2 = i3 + i7;

     i8t1=i8;
     i6t2=i6t1;
     i6t1=i6;

     i0=n0c1+n1c2;
     i1=n0c3+n1c4;
     i2=n0c5+n1c6;
     i3=n2c1+n3c2;
     i4=n2c3+n3c4;
     i5=n2c5+n3c6;
     i6=n4c1+n5c2;
     i7=n4c3+n5c4;
     i8=n4c5+n5c6;

     n0c1=n0*1881;
     n0c3=n0*21509;
     n0c5=n0*9372;
     n1c2=n1*9372;
     n1c4=n1*21509;
     n1c6=n1*1881;
     n2c1=n2*1881;
     n2c3=n2*21509;
     n2c5=n2*9372;
     n3c2=n3*9372;
     n3c4=n3*21509;
     n3c6=n3*1881;
     n4c1=n4*1881;
     n4c3=n4*21509;
     n4c5=n4*9372;
     n5c2=n5*9372;
     n5c4=n5*21509;
     n5c6=n5*1881;
    comb_8to16(s0, pad, &n0);
    comb_8to16(s1, pad, &n1);
    comb_8to16(s2, pad, &n2);
    comb_8to16(s3, pad, &n3);
    comb_8to16(s4, pad, &n4);
    comb_8to16(s5, pad, &n5);
   }
   read_timer (&t1);
   *time = t1 - t0;
   DMA_CPU (OBM2CM, AL, MAP_OBM_stripe(1,"A"), A, 1, MAX_OBM_SIZE*sizeof(int64_t), 0);
   DMA_CPU (OBM2CM, BL, MAP_OBM_stripe(1,"B"), B, 1, MAX_OBM_SIZE*sizeof(int64_t), 0);
   DMA_CPU (OBM2CM, CL, MAP_OBM_stripe(1,"C"), C, 1, MAX_OBM_SIZE*sizeof(int64_t), 0);
   wait_DMA (0);
}
```

APPENDIX E. MATLAB CODE

This section contains the MATLAB code used to format that output data from the SRC-6 and to generate reference calculations with which to compare the QMF data generated on the SRC-6 . The splitter function is used to de-interleave the processed output data from the SRC-6 QMF implementations. The lfilter and hfilter functions are called by the qmf3 function or individually to generate comparison data from the unprocessed samples from the ADC. The lfilter function is not included but is identical to the hfilter function except for the filter coefficients.

```
function [out] = seri(in,length)
for m = 1:length
   for n = 1:6
      out(6*(m-1)+n) = in(m,n);
   end
end
********************************************************************************
function [out] = splitter(in)
for i = 1:8
   for m = (i+8):8:8008
      for n = 1:6
         out(i,6*(((m-i)/8)-1)+n) = in(m-8,n);
      end
   end
end
********************************************************************************
function [lout] = hfilter(in,stage)
hc1 = .0287;
hc2 = .1430;
hc3 = .3282;
hc4 = .3282;
hc5 = .1430;
hc6 = .0287;
in(stage+1)=0;
in(stage+2)=0;
in(stage+3)=0;
in(stage+4)=0;
in(stage+5)=0;
in(stage+6)=0;
for m = 1:2:stage
   lout(m) = hc1*in(m+5)+hc2*in(m+4)+hc3*in(m+3)+hc4*in(m+2)+hc5*in(m+1)+hc6*in(m);
end
lout=lout(1:2:stage);
********************************************************************************
function [lout] = qmf3(in)

lout(9,:) = in;

hhh=hfilter(in,6000);
hhh=hfilter(hhh,3000);
hhh=hfilter(hhh,1500);

lhh=lfilter(in,6000);
lhh=hfilter(lhh,3000);
lhh=hfilter(lhh,1500);

hlh=hfilter(in,6000);
hlh=lfilter(hlh,3000);
```

```
hlh=hfilter(hlh,1500);

llh=lfilter(in,6000);
llh=lfilter(llh,3000);
llh=hfilter(llh,1500);

hhl=hfilter(in,6000);
hhl=hfilter(hhl,3000);
hhl=lfilter(hhl,1500);

lhl=lfilter(in,6000);
lhl=hfilter(lhl,3000);
lhl=lfilter(lhl,1500);

hll=hfilter(in,6000);
hll=lfilter(hll,3000);
hll=lfilter(hll,1500);

lll=lfilter(in,6000);
lll=lfilter(lll,3000);
lll=lfilter(lll,1500);

lout(1,1:750)=hhh;
lout(2,1:750)=lhh;
lout(3,1:750)=hlh;
lout(4,1:750)=llh;
lout(5,1:750)=hhl;
lout(6,1:750)=lhl;
lout(7,1:750)=hll;
lout(8,1:750)=lll;
```

APPENDIX F. SAMPLE DATA

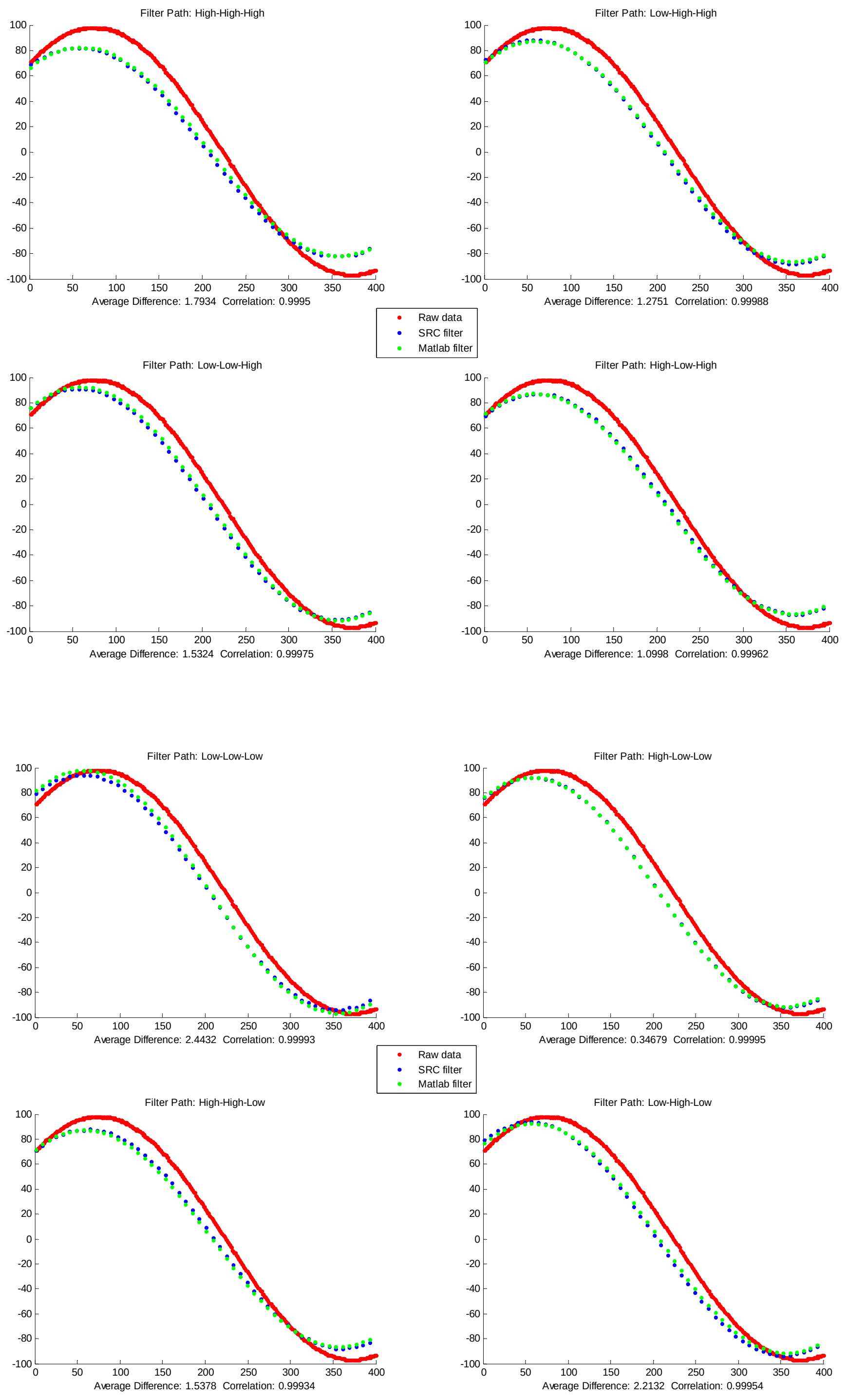


Figure 39. High Resolution View of 1MHz Sine Wave

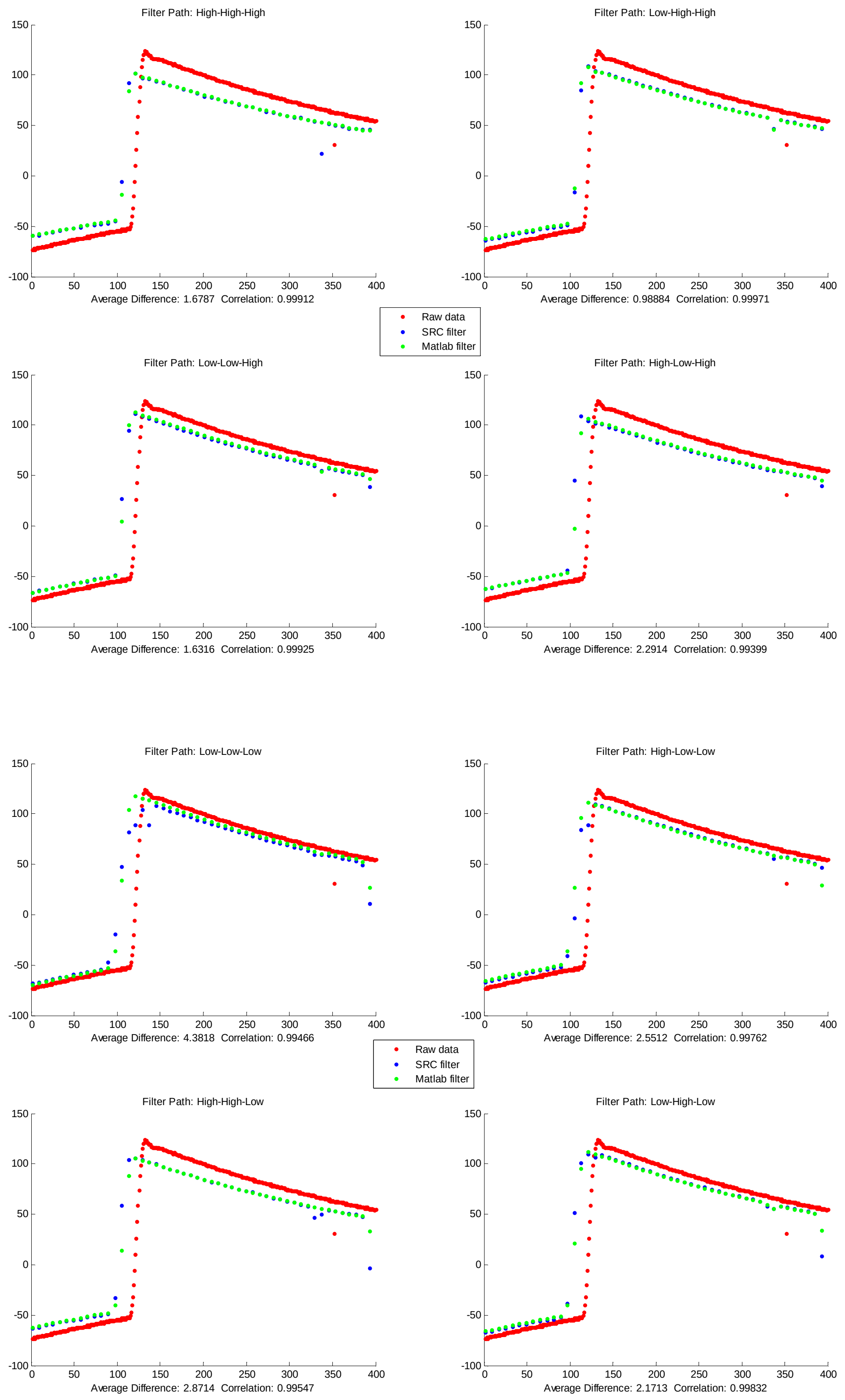


Figure 40. High Resolution View of 1MHz Square Wave

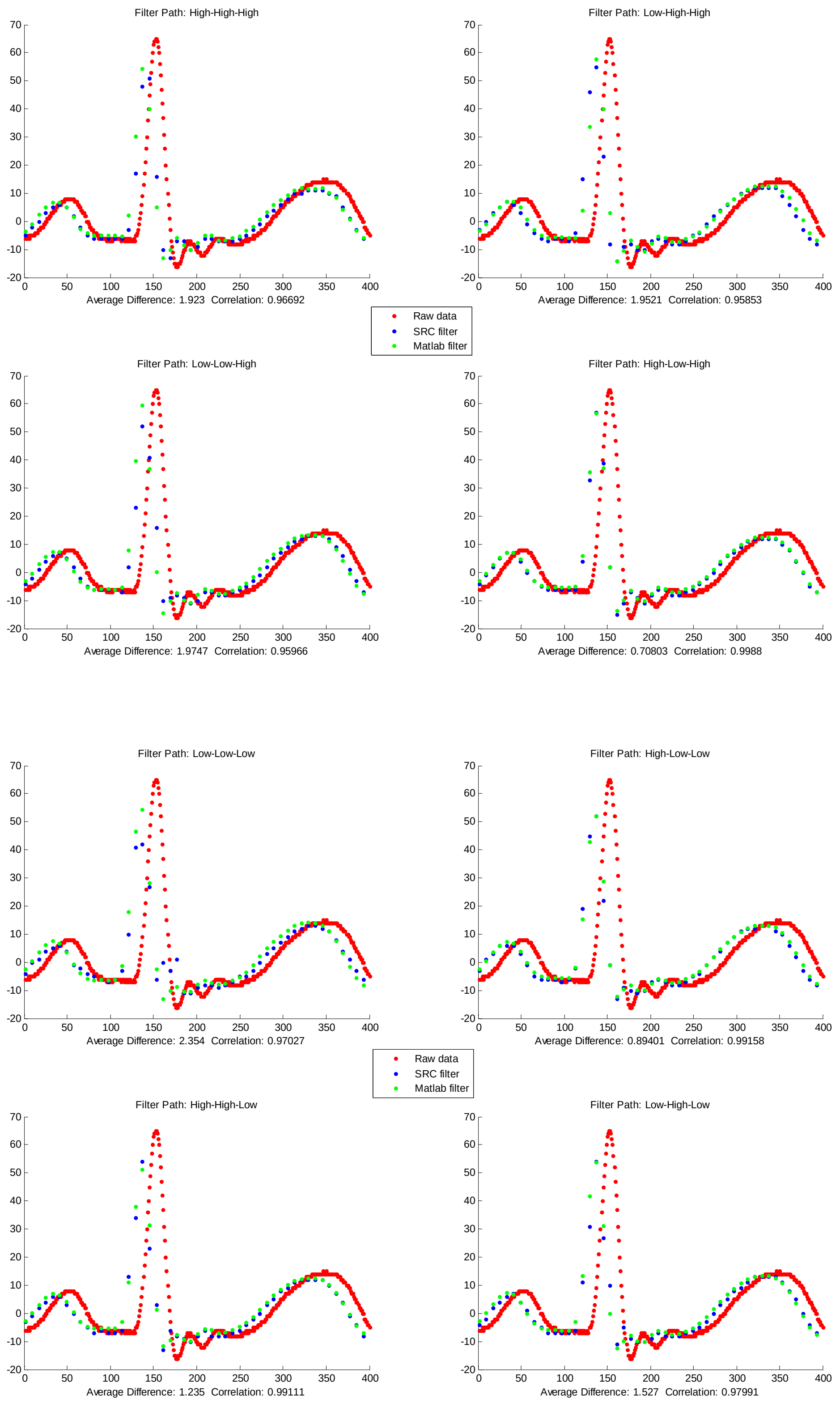


Figure 41. High Resolution View of 1MHz Cardiac Wave

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] P. E. Pace, *Detecting and Classifying Low Probability of Intercept Radar*, Boston Massachusetts: Artech House Inc., 2004, pp 263-302.

[2] D.A. Brown, “ELINT Signal Processing on Reconfigurable Computers for Detection and Classification of LPI Emitters,” M.S. thesis, Naval Postgraduate School, Monterey, California, 2006.

[3] R. Turney, C. Dick, A. Reza, “Multirate Filters and Wavelets: From Theory to Implementation,” Xilinx Inc., Internet: http://www.xilinx.com/products/logicore/dsp/wavelet_final.pdf, November 2, 1999 [September 5, 2006].

[4] J. B. Evans. *Efficient FIR Filter Architectures Suitable for FPGA Implementation*. IEEE Transactions on Circuits and Systems, 41(7):490-493, July 1994.

[5] V. Pasham, A. Miller, K. Chapman, “Transposed Form FIR Filters,” Xilinx XAPP219, Internet: http://direct.xilinx.com/bvdocs/appnotes/xapp219.pdf, October 25, 2001 [September 5, 2006].

[6] M. Adhiwiyogo, “Optimal Pipelining of I/O Ports of the Virtex-II Multiplier,” Internet: http://direct.xilinx.com/bvdocs/appnotes/xapp636.pdf, March 23, 2005, [September 5, 2006].

[7] Xilinx, “644-MHz LVDS Transmitter/Receiver,” Xilinx XAPP622 Internet: http://direct.xilinx.com/bvdocs/appnotes/xapp622.pdf, April 27, 2004, [September 5, 2006].

[8] N. Sawyer, “Data to Clock Phase Alignment,” Xilinx XAPP225, Internet: http://direct.xilinx.com/bvdocs/appnotes/xapp225.pdf, April 4, 2002 [September 5, 2006].

[9] N. Sawyer, “High Speed Data Serialization and Deserialization,” Xilinx XAPP265, Internet: http://direct.xilinx.com/bvdocs/appnotes/xapp265.pdf, June 19, 2002 [September 5, 2006].

[10] N. Sawyer, “Active Phase Alignment,” Xilinx XAPP268, Internet: http://direct.xilinx.com/bvdocs/appnotes/xapp268.pdf, December 9, 2002 [September 5, 2006].

[11] T.L. King, “Hardware Interface to Connect and AN/SPS-65 Radar to an SRC-6E Reconfigurable Computer,” M.S. Thesis, Naval Postgraduate School, Monterey, California, 2005.

[12] T.G. Guthrie, “Design, Implementation, and Testing of a Software Interface between the AN/SPS-65(V)1 Radar and the SRC-6E Reconfigurable Computer,” M.S. Thesis, Naval Postgraduate School, Monterey, California, 2005.

[13] SRC Computers Inc., “SRC-6 C Programming Environment V2.1 Guide,” SRC-007-16, SRC Computers Inc., Colorado Springs, Colorado, August 31, 2005.

[14] David Caliga, Private training session, “SRC Carte Training Course,” Colorado Springs, Colorado, November 2005.

[15] K.R. Macklin, “Benchmarking and Analysis of the SRC-6E Reconfigurable Computing System,” M.S. Thesis, Naval Postgraduate School, Monterey, California, 2003.

[16] Xilinx, “Virtex-II Platform FPGAs: Complete Data Sheet,” Internet: http://direct.xilinx.com/bvdocs/publications/ds031.pdf, March 1, 2005, [September 5, 2006].

[17] Xilinx, “Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet,” Internet: http://direct.xilinx.com/bvdocs/publications/ds083.pdf, October 10, 2005, [September 5, 2006].

[18] National Semiconductor, “ADC08D1500 High Performance, Low Power, Dual 8-Bit, 1.5GSPS A/D Converter,” Internet: http://www.national.com/ds.cgi/DC/ADC08D1500.pdf, October 2005 [September 5, 2006].

[19] SRC Computers Inc., “SRC-6 MAP© Hardware Guide,” SRC-005-05,” Colorado Springs, Colorado, May 26, 2004.

[20] C. Orth, “User Hardware Interfacing to the Chain Port of the SRC MAP© Revision D Board,” SRC-009-00, SRC Computers Inc., Colorado Springs, Colorado, July 17, 2003.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Marine Corps Represenative
   Naval Postgraduate School
   Monterey, California

4. Director, Training and Education, MCCDC, Code C46
   Quantico, Virginia

5. Director, Marine Corps Research Center, MCCDC, Code C40RC
   Quantico, Virginia

6. Marine Corps Tactical Systems Support Activity (Attn: Operations Officer)
   Camp Pendleton, California

7. Chairman, Code EC
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

8. Alan Hunsberger
   National Security Agency
   Ft. Meade Maryland

9. Douglas J. Fouts
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

10. Jon Butler
    Department of Electrical and Computer Engineering
    Naval Postgraduate School
    Monterey, California

11. Phillip E. Pace
    Department of Electrical and Computer Engineering
    Naval Postgraduate School
    Monterey, California

12. Jon Huppenthal
SRC Computers, Inc
Colorado Springs, Colorado