Florida State University Libraries

2004

Adaptive Filter Architectures for FPGA Implementation

Joseph Petrone

THE FLORIDA STATE UNIVERSITY

COLLEGE OF ENGINEERING

ADAPTIVE FILTER ARCHITECTURES FOR FPGA IMPLEMENTATION

By

JOSEPH PETRONE

A Thesis submitted to the
Department of Electrical and Computer Engineering
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Summer Semester, 2004

The members of the Committee approve the thesis of Joseph Petrone defended on 29th of June 2004.

_____
Simon Y Foo
Professor Directing

_____
Uwe Meyer-Baese
Committee Member

_____
Anke Meyer-Baese
Committee Member

Approved:

_____
Reginald Perry, Chair, Department of Electrical and Computer Engineering

_____
Ching-Jen Chen, Dean, FAMU-FSU College of Engineering

The Office of Graduate Studies has verified and approved the above named committee members.

ACKNOWLEDGEMENTS

I would like to thank my major professor Dr. Simon Foo for his guidance and support throughout my graduate study at FSU. I would like to thank the members of my thesis committee, Dr. Uwe Meyer-Baese and Dr. Anke Meyer-Baese, for their valuable advice and guidance. I wish to thank the academic and administrative staff at the Department of Electrical and Computer Engineering for their kind support. I would also like to thank my family and friends for their continuous support and confidence in me.

TABLE OF CONTENTS

LIST OF ACRONYMS

ASIC	Application Specific Integrated Circuit
CLB	Configurable Logic Block
CPLD	Complex Programmable Logic Device
DA	Distributed Arithmetic
DKCM	Dynamic Constant Coefficient Multiplier
DSP	Digital Signal Processing
EDK	Embedded Development Kit
FPGA	Field Programmable Gate Array
FPLD	Field Programmable Logic Device
FPU	Floating-Point Unit
HDL	Hardware Description Language
I/O	Input/Output
IP	Intellectual Property
IPIC	IP Interconnect
IPIF	IP Interface
ISE	Integrated Software Environment
JTAG	Joint Test Action Group
KCM	Constant Coefficient Multiplier
LE	Logic Element
LMS	Least-Mean-Square
LUT	Look-up Table
MAC	Media Access Control
MIPS	Million Instructions per Second
MMI	Monolithic Memories Inc.
MMU	Memory Management Unit
OPB	On-chip Peripheral Bus
PAL	Programmable Array Logic
PLA	Programmable Logic Array
PLB	Processor Local Bus
PLD	Programmable Logic Device
PROM	Programmable Read Only Memory
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RLS	Recursive Least-Squares

ROM	Read Only Memory
RTL	Register Transfer Level
SoC	System-on-Chip
SRAM	Static Random Access Memory
TLB	Translation Look-aside Buffer
UART	Universal Asynchronous Receiver-Transmitter
VCM	Variable Coefficient Multiplier
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Filtering data in real-time requires dedicated hardware to meet demanding time requirements. If the statistics of the signal are not known, then adaptive filtering algorithms can be implemented to estimate the signals statistics iteratively. Modern field programmable gate arrays (FPGAs) include the resources needed to design efficient filtering structures. Furthermore, some manufacturers now include complete microprocessors within the FPGA fabric. This mix of hardware and embedded software on a single chip is ideal for fast filter structures with arithmetic intensive adaptive algorithms.

This thesis aims to combine efficient filter structures with optimized code to create a system-on-chip (SoC) solution for various adaptive filtering problems. Several different adaptive algorithms have been coded in VHDL as well as in C for the PowerPC 405 microprocessor. The designs are evaluated in terms of design time, filter throughput, hardware resources, and power consumption.

CHAPTER 1

Introduction

On systems that perform real-time processing of data, performance is often limited by the processing capability of the system [1]. Therefore, evaluation of different architectures to determine the most efficient architecture is an important task. This chapter discusses the purpose of the thesis, and presents an overview and the direction.

1.1 Purpose

The purpose of this thesis is to explore the use of embedded System-on-Chip (SoC) solutions that modern Field Programmable Gate Arrays (FPGAs) offer. Specifically, it will investigate their use in efficiently implementing adaptive filtering applications. Different architectures for the filter will be compared. In addition, the PowerPC embedded microprocessor will be employed for the various training algorithms. This will be compared to training algorithms implemented in the FPGA fabric only, to determine the optimal system architecture.

1.2 Overview

Digital Signal Processing (DSP) has revolutionized the manner in which we manipulate data. The DSP approach clearly has many advantages over

traditional methods, and furthermore, the devices used are inherently reconfigurable, leading to many possibilities.

1.2.1 Advantages of DSP

Modern computational power has given us the ability to process tremendous amounts of data in real-time. DSP is found in a wide variety of applications, such as: filtering, speech recognition, image enhancement, data compression, neural networks; as well as functions that are unpractical for analog implementation, such as linear-phase filters [2]. Signals from the real world are naturally analog in form, and therefore must first be discretely sampled for a digital computer to understand and manipulate.

The signals are discretely sampled and quantized, and the data is represented in binary format so that the noise margin is overcome. This makes DSP algorithms insensitive to thermal noise. Further, DSP algorithms are predictable and repeatable to the exact bits given the same inputs. This has the advantage of easy simulation and short design time. Additionally, if a prototype is shown to function correctly, then subsequent devices will also.

1.2.2 Reconfigurable Hardware Advantages

There are many advantages to hardware that can be reconfigured with different programming files. Dedicated hardware can provide the highest processing performance, but is inflexible for changes. Reconfigurable hardware devices offer both the flexibility of computer software, and the ability to construct custom high performance computing circuits [1]. The hardware can swap out configurations based on the task at hand, effectively multiplying the amount of physical hardware available.

In space applications, it may be necessary to install new functionality into a system, which may have been unforeseen. For example, satellite applications need to be able to adjust to changing operation requirements [3]. With a reconfigurable chip, functionality that was not predicted at the outset can be uploaded to the satellite when needed.

1.3 Organization of Thesis

Chapter 2 presents a brief history of programmable logic devices. Next, chapter 3 provides an overview of the adaptive filtering problem and the various training algorithms. Chapter 4 details the specifics of FPGA implementation, such as algorithm modification and detailed architectures. Simulation results are presented in chapter 5. Finally, chapter 6 will draw conclusions and future extensions of the work.

CHAPTER 2

Programmable Logic Devices

This chapter details the history of programmable logic devices, from the simple beginnings to their modern complex architectures. Current trends such as embedded DSP blocks are discussed, as well as the hardware description languages and tools that are used to program them.

2.1 History of Programmable Logic

Programmable logic is loosely defined as a device with configurable logic and flip-flops linked together with programmable interconnects. The first programmable device was the programmable array logic (PAL) developed by Monolithic Memories Inc. (MMI) in 1975 [4]. Considering that any Boolean function can be realized as a sum-of-products or equivalently as a product-of-sums by utilizing De Morgan's law, the PAL structure is rather intuitive. It generally consists of inputs with inverters leading into a series of AND gates whose outputs lead into a series of OR gates. This makes the products of any combination of the inputs and their complements available to the OR gates for the sum.

A similar device, the programmable logic array (PLA), reverses the order of the AND and OR gates, which led to greater functionality. The reason is that the product terms can be shared across the OR gates at the outputs, effectively giving the chip more logic width.

The structure in Figure 2.1 is a usual PLA before programming, with all possible connections are pre-wired typically by fuses. To implement a custom design, a programmer is used to blow the fuses with high current and break the unwanted connections.
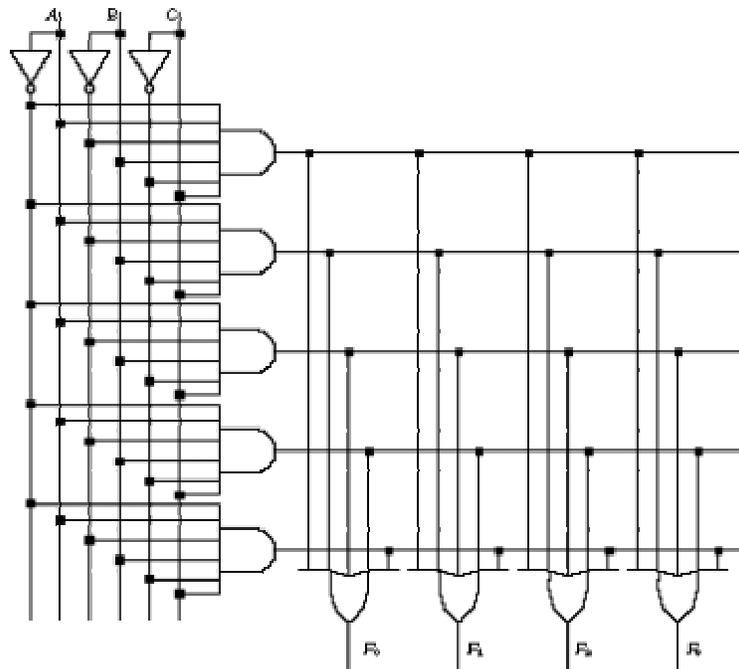
Figure 2.1 PLA structure before programming.

An improvement from PAL and PLAs came with the introduction of the complex programmable logic device (CPLD), which allows for more complex logic circuits. A CPLD consists of multiple PAL-like blocks connected by programmable interconnects. While PALs are programmed with a programmer, a CPLD is programmed in-system with the manufacturers proprietary method or with a JTAG cable connected to a computer. CPLDs are well suited to complex, high-performance state machines.

An alternative type of PLD developed more recently is the field programmable gate array (FPGA). Xilinx introduced the FPGA in 1984. These devices have a more flexible, gate-array-like structure with a hierarchical interconnect arrangement. The fundamental part of the FPGA is the look-up table (LUT), which acts as a function generator, or can alternatively be configured as ROM or RAM. They also include fast carry logic to adjacent cells making them suitable for arithmetic functions and further DSP applications.

2.2   FPGA Architecture

The majority of FPGAs are SRAM-based and can therefore be programmed as easily as standard SRAM. The SRAM bits are coupled to configuration points in the FPGA (Figure 2.2 left) and controls whether or not a connection is made. This is normally accomplished by a passgate structure (Figure 2.2 right) that turns the connection on or off depending on the logic value (True or False) supplied by the SRAM. Because they are SRAM based, FPGAs are volatile. As such, they must be programmed each time power is applied. This is normally accomplished with another part of the circuit that reloads the configuration bitsream, such as a PROM.
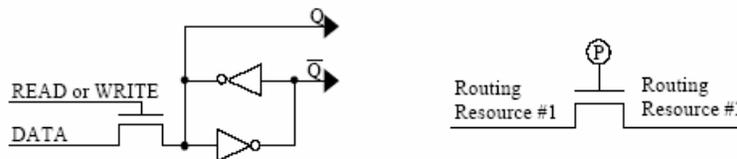
Figure 2.2 SRAM based FPGA configuration.

The configuration bitstream stored in the SRAM controls the connections made and also the data to be stored in the Look-up tables (LUTs). The LUTs are

essentially small memories that can compute arbitrary logic functions. Each manufacturer has a distinct name for their basic block, but the fundamental unit is the LUT. Altera call theirs a Logic Element (LE) while Xilinx's FPGAs have configurable logic blocks (CLBs) organized in an array. The configurable logic blocks of an FPGA are generally placed in an island style arrangement (Figure 2.3). Each logic block in the array is connected to routing resources controlled by a interconnect switch matrix.
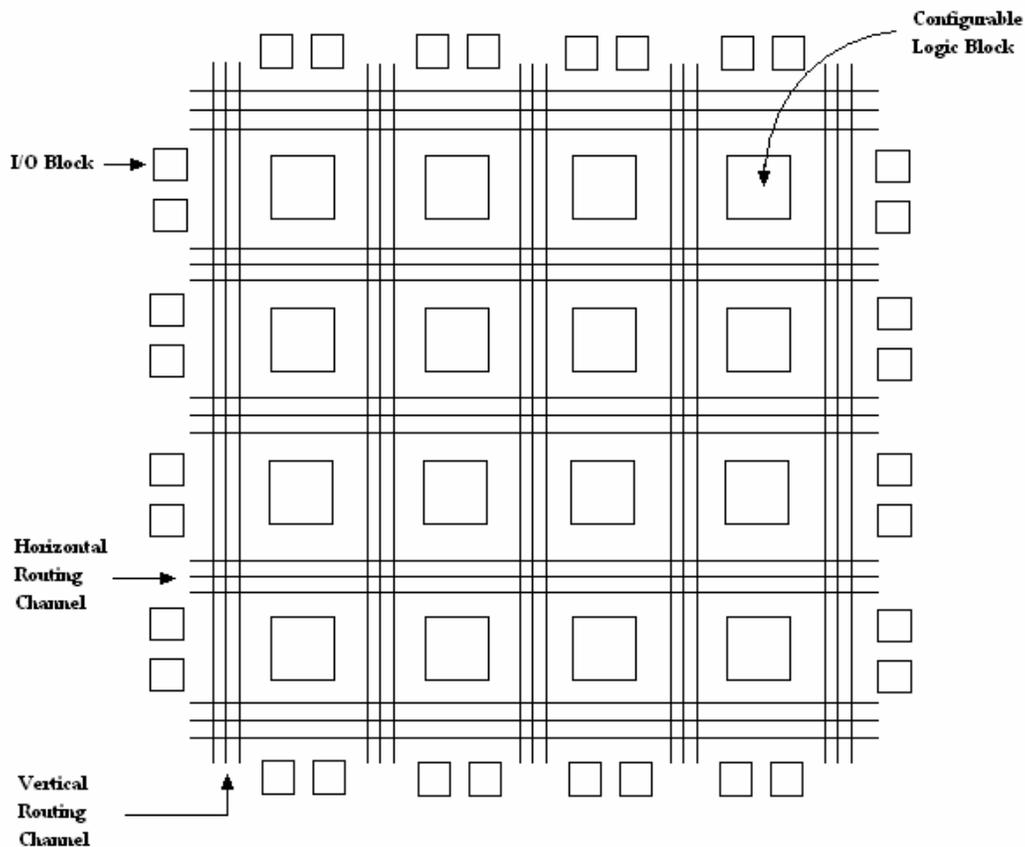
Figure 2.3 Generic Island Style Routing Architecture

With this layout, a very large range of connections can be made between resources. A downside to this flexible routing structure is that unlike the CPLD, signal paths are not fixed beforehand, which can lead to unpredictable timing. However, the tradeoff is the FPGA's increased logic complexity and flexibility.

7

Each CLB in a Xilinx FPGA encompasses four logic slices, which in turn contain two 4-input function generators, carry logic, arithmetic logic gates, wide function multiplexers and two storage elements [5]. The top half of a slice is shown in figure 2.4.
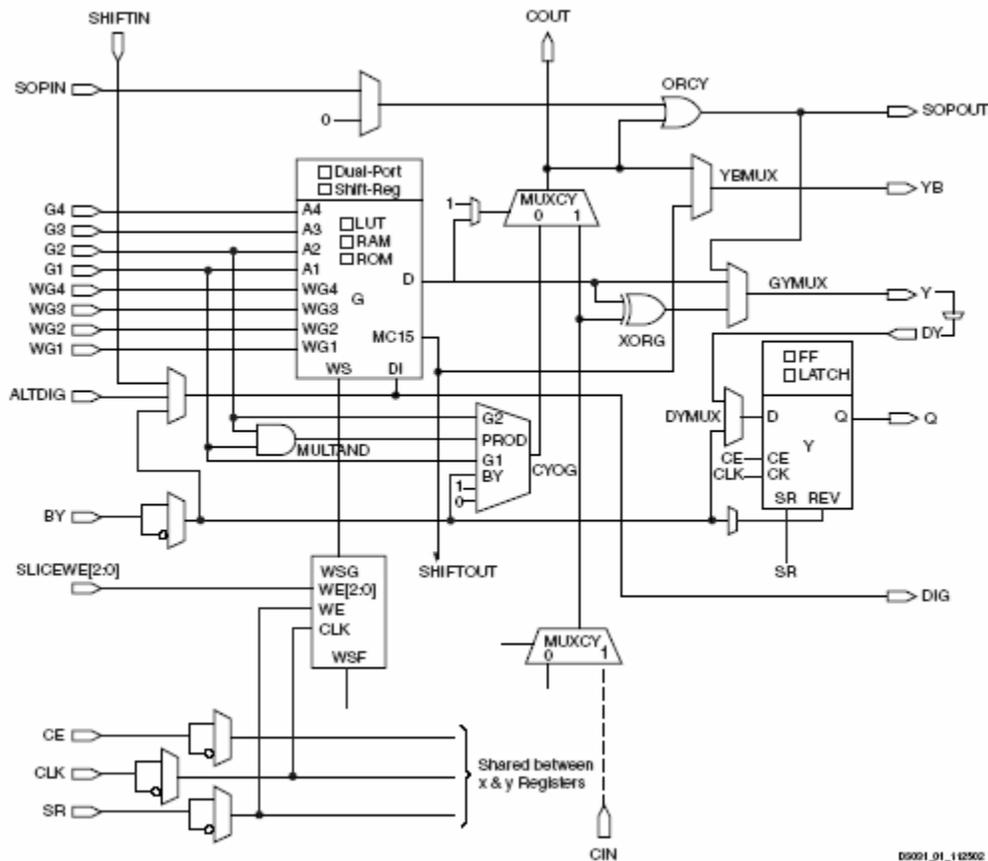
Figure 2.4 Virtex-II Pro Slice (Top Half).

The LUT is capable of implementing any arbitrary defined Boolean function of four inputs and the propagation delay is therefore constant regardless of the function. Each slice also contains flip-flops and a fast carry chain. The dedicated fast carry logic allows the FPGA to realize very fast arithmetic circuits.

8

2.3 Device Configuration

Manually defining the routing connections in a programmable device may have been feasible with the early PALs but is nearly impossible considering the density of modern FPGAs. Configuring these programmable devices can be achieved in several ways, such as schematic design entry, the use of hardware description languages (HDLs), and the use of high-level language compilers. These methods are listed in increasing levels of abstraction, with schematic design entry being the lowest level.

2.3.1 Schematic Design Entry

Schematic design practices entails selecting standard logic gates from a library to create a graphic description of the circuit to be realized, and manually wiring them together. The schematic design library typically includes standard Boolean logic gates, multiplexers, I/O buffers, and macros for device specific functions, such as clock dividers. Custom components can be constructed from the smaller blocks to create user macros for use in large designs.

As an example, to create a half-adder, whose function is to add to binary bits, requires one to first construct the truth table, as shown in Table 2.1.

Table 2.1 Half-Adder Truth Table.

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

The binary inputs A and B are added to produce the output bit S and a carry bit C. The logic equations to implement can be distinguished from the truth table, and are:

$$S = \overline{A} \cdot B + A \cdot \overline{B}$$
$$C = A \cdot B$$

Once the logic equations are determined the circuit can be easily assembled as shown in figure 2.5. One drawback, however, is that going backward from schematic design to logic function is not so easy. Also, trivial design changes may require heavy schematic modification.
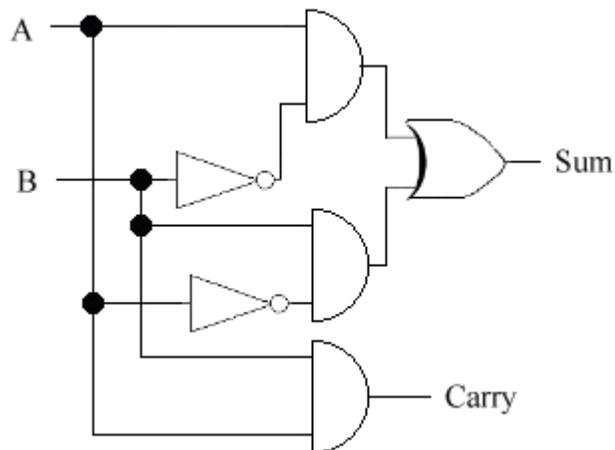
Figure 2.5 Schematic description of a half-adder.

It should be noted that although this is the lowest level of abstraction, the synthesis tool will optimize the design for the specific device structure and the end result may differ significantly in layout from the original design. This is the least popular method of describing hardware designs for several reasons. The most important though, is that reverse engineering a foreign design is very hard to do.

2.3.2 Hardware Description Languages

The most popular hardware description languages are Verilog and VHDL. Both are text-based depictions of the behavior of the digital circuit, and their syntax contains explicit notations for expressing time and concurrency.

Gateway Design Automation Inc. started the Verilog language around 1984 as a proprietary hardware modeling language [6]. The language went public in 1990 and has since been very popular in the semiconductor industry for ASIC and FPGA design.

VHDL is a hardware description language that grew out of the VHSIC program sponsored by the Department of Defense [7] and was first released in 1985. The acronym VHDL, stands for VHSIC Hardware Description Language, with the acronym VHSIC standing for Very High-Speed Integrated Circuit.

2.3.3 High-Level Languages

There is increasing interest in using high-level programming languages for FPGA design. Some, such as Celoxica's DK Design Suite, generate HDL from a C-like language. The Confluence language, based on Python, also takes this approach. The custom language is compiled to generate a VHDL or Verilog circuit description. The AccelFPGA tool from AccelChip similarly produces a register transfer level (RTL) circuit description from a Matlab m-file.

An alternate approach is to generate the device netlist directly form the high-level description. This is what the Lava language, still under research by Xilinx and others, does. Lava is based on the lazy programming language Haskell, but is not yet available for system design.

A shortcoming of the high-level design languages is their inability to instantiate vendor specific functions, such as block RAMs and DSP blocks. With

the move toward incorporating further highly specific blocks, such as microprocessors, this shortcoming will need to be overcome before any of these languages takes hold.

2.4 Current Trends

The current trend in FPGA architectures is a move toward complete embedded systems. FPGA densities have increased to the point that entire RISC microprocessor soft cores can fit comfortably with additional logic on a single chip. Recognizing this trend, FPGA manufacturers are also including embedded block RAM and hard microprocessor cores in several of their new FPGAs. Altera's Excalibur device contains an ARM922T™ processor core whereas Xilinx's Virtex-II Pro contains up to four IBM Power PC microprocessors. This gives engineers the flexibility to mix hardware and software in embedded applications to achieve the maximum performance.

The idea of integrating all the components of a computer system on a single chip is known a System-on-Chip (SoC). This includes the microprocessor, embedded RAM, and output interfaces such as UART or Ethernet MAC. FPGAs are highly attractive for this because the less common components can always be included as a soft core. Standard FPGAs will most likely be produced for a long time, with the dominating trend moving toward those including hard IP cores.

CHAPTER 3

Adaptive Filter Overview

Adaptive filters learn the statistics of their operating environment and continually adjust their parameters accordingly. This chapter presents the theory of the algorithms needed to train the filters.

3.1   Introduction

In practice, signals of interest often become contaminated by noise or other signals occupying the same band of frequency. When the signal of interest and the noise reside in separate frequency bands, conventional linear filters are able to extract the desired signal [2]. However, when there is spectral overlap between the signal and noise, or the signal or interfering signal's statistics change with time, fixed coefficient filters are inappropriate. Figure 3.1 shows an example of a wideband signal whose Fourier spectrum overlaps a narrowband interference signal.
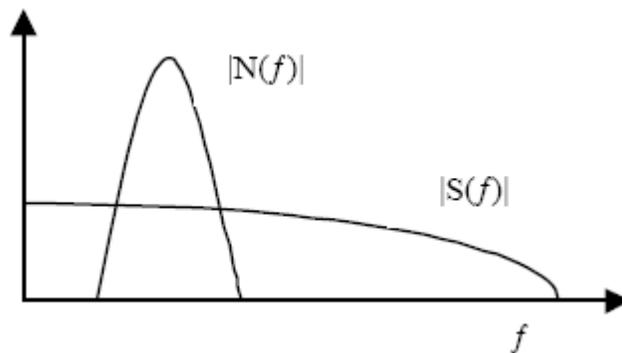
Figure 3.1. A strong narrowband interference N(f) in a wideband signal S(f).

This situation can occur frequently when there are various modulation technologies operating in the same range of frequencies. In fact, in mobile radio systems co-channel interference is often the limiting factor rather than thermal or other noise sources [8]. It may also be the result of intentional signal jamming, a scenario that regularly arises in military operations when competing sides intentionally broadcast signals to disrupt their enemies' communications. Furthermore, if the statistics of the noise are not known a priori, or change over time, the coefficients of the filter cannot be specified in advance. In these situations, adaptive algorithms are needed in order to continuously update the filter coefficients.

3.2 Adaptive Filtering Problem

The goal of any filter is to extract useful information from noisy data. Whereas a normal fixed filter is designed in advance with knowledge of the statistics of both the signal and the unwanted noise, the adaptive filter continuously adjusts to a changing environment through the use of recursive algorithms. This is useful when either the statistics of the signals are not known beforehand of change with time.
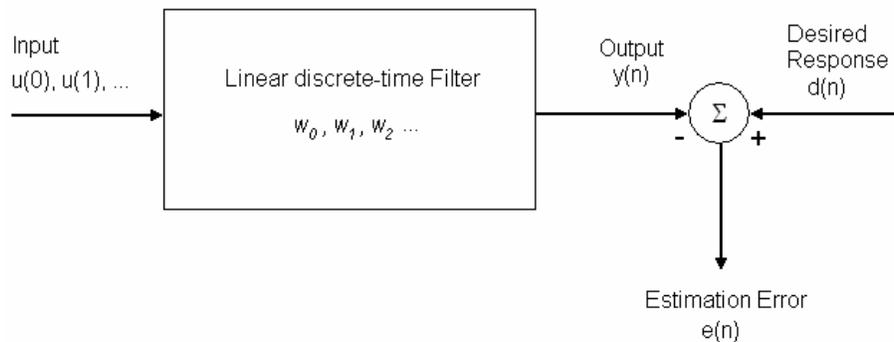
Figure 3.2 Block diagram for the adaptive filter problem.

The discrete adaptive filter (see figure 3.2) accepts an input u(n) and produces an output y(n) by a convolution with the filter's weights, w(k). A desired reference signal, d(n), is compared to the output to obtain an estimation error e(n). This error signal is used to incrementally adjust the filter's weights for the next time instant. Several algorithms exist for the weight adjustment, such as the *Least-Mean-Square* (LMS) and the *Recursive Least-Squares* (RLS) algorithms. The choice of training algorithm is dependent upon needed convergence time and the computational complexity available, as statistics of the operating environment.

3.3 Applications

Because of their ability to perform well in unknown environments and track statistical time-variations, adaptive filters have been employed in a wide range of fields. However, there are essentially four basic classes of applications [9] for adaptive filters. These are: *Identification, inverse modeling, prediction*, and *interference cancellation*, with the main difference between them being the manner in which the desired response is extracted. These are presented in figure 3.3 a, b, c, and d, respectively.

The adjustable parameters that are dependent upon the applications at hand are the number of filter taps, choice of FIR or IIR, choice of training algorithm, and the learning rate. Beyond these, the underlying architecture required for realization is independent of the application. Therefore, this thesis will focus on one particular application, namely noise cancellation, as it is the most likely to require an embedded VLSI implementation. This is because it is sometimes necessary to use adaptive noise cancellation in communication systems such as handheld radios and satellite systems that are contained on a

single silicon chip, where real-time processing is required.   Doing this efficiently is important, because adaptive equalizers are a major component of receivers in modern communications systems and can account for up to 90% ot the total gate count [10].
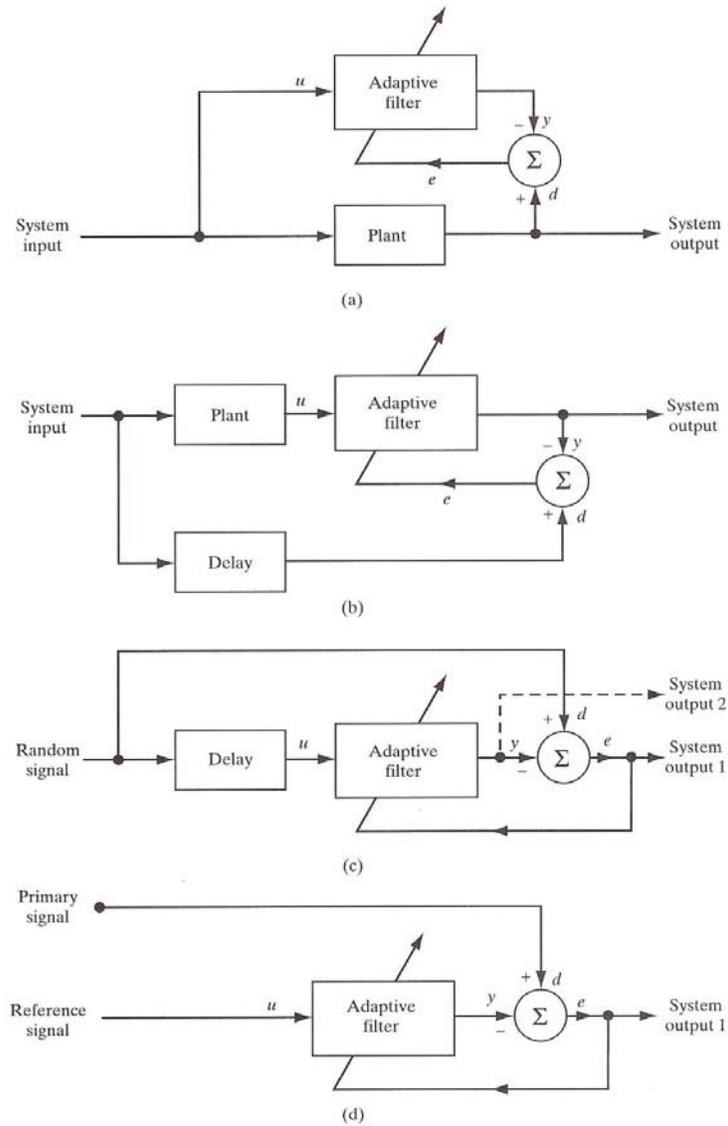
Figure 3.3 Four basic classes of adaptive filtering applications [9].

3.4 Adaptive Algorithms

There are numerous methods for the performing weight update of an adaptive filter. There is the Wiener filter, which is the optimum linear filter in

16

the terms of mean squared error, and several algorithms that attempt to approximate it, such as the method of steepest descent. There is also least-mean-square algorithm, developed by Widrow and Hoff originally for use in artificial neural networks. Finally, there are other techniques such as the recursive-least-squares algorithm and the Kalman filter. The choice of algorithm is highly dependent on the signals of interest and the operating environment, as well as the convergence time required and computation power available.

3.4.1 Wiener Filters

The Wiener filter, so named after its inventor, was developed in 1949. It is the optimum linear filter in the sense that the output signal is as close to the desired signal as possible. Although not often implemented in practice due to computational complexity, the Wiener filter is studied as a frame of reference for the linear filtering of stochastic signals [9] to which other algorithms can be compared.

To formulate the Weiner filter and other adaptive algorithms, the mean squared error (MSE) is used. If the input signal $\mathbf{u}(n)$ to a filter with M taps is given as

$$\mathbf{u}(n) = \left[u(n), u(n-1), \ldots, u(n-M+1)\right]^T,$$

and the coefficients or weight vector is given as

$$\mathbf{w} = \left[w(0), w(1), \ldots, w(M-1)\right]^T,$$

then the square of the output error can be formulated as

$$e_n^2 = d_n^2 - 2d_n\mathbf{u}_n^T\mathbf{w} + \mathbf{w}^T\mathbf{u}_n\mathbf{u}_n^T\mathbf{w}.$$

The mean square error, $\mathbf{J}$, is obtained by taking the expectations of both sides:

$$\begin{aligned}\mathbf{J} &= E[e_n^2] = E[d_n^2] - 2E[d_n\mathbf{u}_n^T\mathbf{w} + \mathbf{w}^T\mathbf{u}_n\mathbf{u}_n^T\mathbf{w}] \\ &= \sigma^2 + 2\mathbf{p}^T\mathbf{w} + \mathbf{w}^T\mathbf{R}\mathbf{w}\end{aligned}.$$

Here, σ is the variance of the desired output, **p** is the cross-correlation vector and **R** is the autocorrelation matrix of **u**. A plot of the MSE against the weights is a non-negative bowl shaped surface with the minimum point being the optimal weights. This is referred to as the error performance surface [2], whose gradient is given by

$$\nabla = \frac{d\mathbf{J}}{d\mathbf{w}} = -2\mathbf{p} + 2\mathbf{R}\mathbf{w} .$$

Figure 3.4 shows an example cross-section of the error performance surface for a two tap filter.
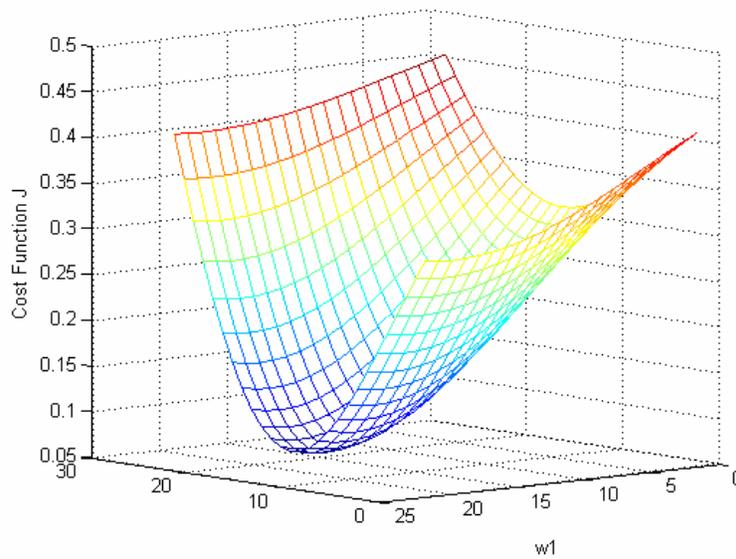
Figure 3.4 Example cross section of an error-performance surface for a two tap filter.

To determine the optimal Wiener filter for a given signal requires solving the Wiener-Hopf equations. First, let the matrix **R** can denote the M-by-M correlation matrix of **u**. That is,

$$\mathbf{R} = E\big[\mathbf{u}(n)\mathbf{u}^H(n)\big],$$

where the superscript H denotes the Hermitian transpose. In expanded form this is

$$\mathbf{R} = \begin{bmatrix} r(0) & r(1) & \cdots & r(M-1) \\ r*(1) & r(0) & \cdots & r(M-2) \\ \vdots & \vdots & \ddots & \vdots \\ r*(M-1) & r*(M-2) & \cdots & r(0) \end{bmatrix}.$$

Also, let $\mathbf{p}$ represent the cross-correlation vector between the tap inputs and the desired response $d(n)$:

$$\mathbf{p} = E\big[\mathbf{u}(n)d*(n)\big],$$

which expanded is:

$$\mathbf{p} = \big[p(0), p(-1),\ldots, p(1-M)\big]^T.$$

Since the lags in the definition of $\mathbf{p}$ are either zero or negative, the Wiener-Hopf equation may be written in compact matrix form:

$$\mathbf{R}\mathbf{w}_o = \mathbf{p},$$

with $\mathbf{w}_o$ stands for the M-by-1 *optimum tap-weight vector* [9], for the transversal filter. That is, the optimum filter's coefficients will be:

$$\mathbf{w}_o = \big[\mathbf{w}_{o0}, \mathbf{w}_{o1},\ldots, \mathbf{w}_{o,M-1}\big]^T.$$

This produces the optimum output in terms of the mean-square-error, however if the signals statistics change with time then the Wiener-Hopf equation must be recalculated. This would require calculating two matrices, inverting one of them and then multiplying them together. This computation cannot be feasibly calculated in real time, so other algorithms that approximate the Wiener filter must be used.

3.4.2 Method of Steepest Descent

With the error-performance surface defined previously, one can use the method of steepest-descent to converge to the optimal filter weights for a given problem. Since the gradient of a surface (or hypersurface) points in the direction of maximum increase, then the direction opposite the gradient ($-\nabla$) will point

towards the minimum point of the surface. One can adaptively reach the minimum by updating the weights at each time step by using the equation

$$\mathbf{w}_{n+1} = \mathbf{w}_n + \mu(-\nabla_n),$$

where the constant μ is the step size parameter. The step size parameter determines how fast the algorithm converges to the optimal weights. A necessary and sufficient condition for the convergence or stability of the steepest-descent algorithm [9] is for μ to satisfy

$$0 < \mu < \frac{2}{\lambda_{max}},$$

where λ_{max} is the largest eigenvalue of the correlation matrix $\mathbf{R}$.

Although it is still less complex than solving the Wiener-Hopf equation, the method of steepest-descent is rarely used in practice because of the high computation needed. Calculating the gradient at each time step would involve calculating $\mathbf{p}$ and $\mathbf{R}$, whereas the least-mean-square algorithm performs similarly using much less calculations.

3.4.3 Least-Mean-Square Algorithm

The least-mean-square (LMS) algorithm is similar to the method of steepest-descent in that it adapts the weights by iteratively approaching the MSE minimum. Widrow and Hoff invented this technique in 1960 for use in training neural networks. The key is that instead of calculating the gradient at every time step, the LMS algorithm uses a rough approximation to the gradient.

The error at the output of the filter can be expressed as

$$e_n = d_n - \mathbf{w}_n^T \mathbf{u}_n,$$

which is simply the desired output minus the actual filter output. Using this definition for the error an approximation of the gradient is found by

$$\hat{\nabla} = -2e_n \mathbf{u}_n .$$

Substituting this expression for the gradient into the weight update equation from the method of steepest-descent gives

$$\mathbf{w}_{n+1} = \mathbf{w}_n + 2\mu \cdot e_n \mathbf{u}_n ,$$

which is the Widrow-Hoff LMS algorithm. As with the steepest-descent algorithm, it can be shown to converge [9] for values of μ less than the reciprocal of λ_{max}, but λ_{max} may be time-varying, and to avoid computing it another criterion can be used. This is

$$0 < \mu < \frac{2}{MS_{max}} ,$$

where M is the number of filter taps and S_{max} is the maximum value of the power spectral density of the tap inputs u.

The relatively good performance of the LMS algorithm given its simplicity has caused it to be the most widely implemented in practice. For an N-tap filter, the number of operations has been reduced to 2*N multiplications and N additions per coefficient update. This is suitable for real-time applications, and is the reason for the popularity of the LMS algorithm.

3.4.4 Recursive Least Squares Algorithm

The recursive-least-squares (RLS) algorithm is based on the well-known least squares method [2]. The least-squares method is a mathematical procedure for finding the best fitting curve to a given set of data points. This is done by minimizing the sum of the squares of the offsets of the points from the curve. The RLS algorithm recursively solves the least squares problem.

In the following equations, the constants λ and δ are parameters set by the user that represent the *forgetting factor* and *regularization parameter* respectively. The forgetting factor is a positive constant less than unity, which is roughly a

measure of the memory of the algorithm; and the regularization parameter's value is determined by the signal-to-noise ratio (SNR) of the signals. The vector $\hat{\mathbf{w}}$ represents the adaptive filter's weight vector and the *M*-by-*M* matrix **P** is referred to as the inverse correlation matrix. The vector $\boldsymbol{\pi}$ is used as an intermediary step to computing the gain vector **k**. This gain vector is multiplied by the a priori estimation error $\xi(n)$ and added to the weight vector to update the weights. Once the weights have been updated the inverse correlation matrix is recalculated, and the training resumes with the new input values. A summary of the RLS algorithm follows [9]:

Initialize the weight vector and the inverse correlation matrix P.

$$\hat{\mathbf{w}}^H(0) = \overline{\mathbf{0}},$$
$$\mathbf{P}(0) = \delta^{-1}\mathbf{I}, \qquad \text{where} \qquad \delta = \begin{cases} \text{Small positive constant for high SNR} \\ \text{Large positive constant for low SNR} \end{cases}$$

For each instance of time n = 1, 2, 3 …, compute:

$$\pi(n) = \mathbf{P}(n-1)\mathbf{u}(n),$$
$$\mathbf{k}(n) = \frac{\pi(n)}{\lambda + \mathbf{u}^H(n)\pi(n)},$$
$$\xi(n) = d(n) - \hat{\mathbf{w}}^H(n-1)\mathbf{u}(n),$$
$$\widehat{\mathbf{w}}(n) = \hat{\mathbf{w}}(n-1) + \mathbf{k}(n)\xi^*(n),$$
and
$$\mathbf{P}(n) = \lambda^{-1}\mathbf{P}(n-1) - \lambda^{-1}\mathbf{k}(n)\mathbf{u}^H(n)\mathbf{P}(n-1).$$

An adaptive filter trained with the RLS algorithm can converge up to an order of magnitude faster than the LMS filter at the expense of increased computational complexity.

CHAPTER 4

FPGA Implementation

The efficient realization of complex algorithms on FPGAs requires a familiarity with their specific architectures. This chapter discusses the modifications needed to implement an algorithm on an FPGA and also the specific architectures for adaptive filtering and their advantages.

4.1 FPGA Realization Issues

Field programmable gate arrays are ideally suited for the implementation of adaptive filters. However, there are several issues that need to be addressed. When performing software simulations of adaptive filters, calculations are normally carried out with floating point precision. Unfortunately, the resources required of an FPGA to perform floating point arithmetic is normally too large to be justified, and measures must be taken to account for this. Another concern is the filter tap itself. Numerous techniques have been devised to efficiently calculate the convolution operation when the filter's coefficients are fixed in advance. For an adaptive filter whose coefficients change over time, these methods will not work or need to be modified significantly.

First, the issues involved in transitioning to a fixed-point algorithm will be detailed. Next, the design of the filter tap will be considered. The reconfigurable filter tap is the most important issue for a high performance adaptive filter architecture, and as such it will be discussed at length. Finally, the integration of the embedded processor for the coefficient update will be discussed.

4.2 Finite Precision Effects

Although computing floating point arithmetic in an FPGA is possible [11], it is usually accomplished with the inclusion of a custom floating point unit. These units are very costly in terms of logic resources. Because of this, only a small number of floating point units can be used in an entire design, and must be shared between processes. This does not take full advantage of the parallelization that is possible with FPGAs and is therefore not the most efficient method. All calculation should therefore be mapped to fixed point only, but this can introduce some errors. The main errors in DSP are [2]:

1) ADC quantization error – this results from representing the input data by a limited number of bits.
2) Coefficient quantization error – this is caused by representing the coefficients of DSP parameters by a finite number of bits.
3) Overflow error – this is caused by the addition of two large numbers of the same sign which produces a result that exceeds permissible word length.
4) Round off error – this is caused when the result of a multiplication is rounded (or truncated) to the nearest discrete value or permissible word length.

The first issue is not applicable here, but the others three must be addresses and handled properly.

4.2.1 Scale Factor Adjustment

A suitable compromise for dealing with the loss of precision when transitioning from a floating point to a fixed-point representation is to keep a limited number of decimal digits. Normally, two to three decimal places is adequate, but the number required for a given algorithm to converge must be found through experimentation. When performing software simulations of a

digital filter for example, it is determined that two decimal places is sufficient for accurate data processing. This can easily be obtained by multiplying the filter's coefficients by 100 and truncating to an integer value. Dividing the output by 100 recovers the anticipated value. Since multiplying and dividing be powers of two can be done easily in hardware by shifting bits, a power of two can be used to simplify the process. In this case, one would multiply by 128, which would require seven extra bits in hardware. If it is determined that three decimal digits are needed, then ten extra bits would be needed in hardware, while one decimal digit requires only four bits.

For simple convolution, multiplying by a preset scale and then dividing the output by the same scale has no effect on the calculation. For a more complex algorithm, there are several modifications that are required for this scheme to work. These are given in Algorithm 4.1. The first change needed to maintain the original algorithm's consistency requires dividing by the scale constant any time two previously scaled values are multiplied together. Consider, for example, the values a and b and the scale constant s. The scaled integer values are represented by s·a and s·b. To multiply theses values requires dividing by s to correct for the s^2 term that would be introduced and recover the scaled product a·b:

$$(s \cdot a \times s \cdot b)/s = s \cdot ab.$$

Likewise, divisions must be corrected with a subsequent multiplication. It should now be evident why a power of two is chosen for the scale constant, since multiplication and division by powers of two result in simple bit shifting. Addition and subtraction require no additional adjustment.

The aforementioned procedure must be applied with caution, however, and does not work in all circumstances. While it is perfectly legal to apply to the convolution operation of a filter, it may need to be tailored for certain aspects of a

given algorithm. Consider the tap-weight adaptation equation for the LMS algorithm:

$$\hat{\mathbf{w}}(n+1) = \hat{\mathbf{w}}(n) + \mu \cdot \mathbf{u}(n)e^*(n),$$

where μ is the learning rate parameter; its purpose is to control the speed of the adaptation process. The LMS algorithm is convergent in the mean square provided that

$$0 < \mu < \frac{2}{\lambda_{max}}$$

where λ_{max} is the largest eigenvalue of the correlation matrix $\mathbf{R_x}$ [9] of the filter input. Typically this is a fraction value and its product with the error term has the effect of keeping the algorithm from diverging. If μ is blindly multiplied by some scale factor and truncated to a fixed-point integer, it will take on a value greater than one. The affect will be to make the LMS algorithm diverge, as its inclusion will now amplify the added error term. The heuristic adopted in this case is to divide by the inverse value, which will be greater than 1. Similarly, division by values smaller than 1 should be replaced by multiplication with its

---

Algorithm 4.1: Fixed Point Conversion

Determine Scale
      Through simulations, find the needed accuracy (# decimal places).
      Scale = accuracy rounded up to a power of two.
      Multiply all constants by scale
        - Divide by scale when two scaled values are multiplied.
        - Multiply by scale when two scaled values are divided.
Replace
      For multiplication by values les than 1
          → Replace with division by the reciprocal value.
      Likewise, for division by values less than 1
          → Replace with multiplication by the reciprocal value.

inverse. The outputs of the algorithm will then need to be divided by the scale to obtain the true output.

4.2.2 Training Algorithm Modification

The training algorithms for the adaptive filter need some minor modifications in order to converge for a fixed-point implementation. Changes to the LMS weight update equation were discussed in the previous section. Specifically, the learning rate μ and all other constants should be multiplied by the scale factor. First, μ is adjusted

$$\hat{\mu} = \frac{1}{\mu} \cdot scale.$$

The weight update equation then becomes:

$$\hat{\mathbf{w}}(n+1) = \hat{\mathbf{w}}(n) + \frac{\mathbf{u}(n)e^*(n)}{\hat{\mu}}.$$

This describes the changes made for the direct from FIR filter, and further changes may be needed depending on the filter architecture at hand.

The direct form FIR structure has a delay that is determined by the depth of the output adder tree, which is dependent on the filter's order. The transposed form FIR, on the other hand, has a delay of only one multiplier and one adder regardless of filter length. It is therefore advantageous to use the transposed form for FPGA implementation to achieve maximum bandwidth. Figure 4.1 shows the direct and transposed form structures for a three tap filter. The relevant nodes have been labeled A, B, and C for a data flow analysis. The filters each have thee coefficients, and are labeled $h_0[n]$, $h_1[n]$, and $h_2[n]$. The coefficients' subscript denotes the relevant filter tap, and the n subscript represents the time index, which is required since adaptive filters adjust their coefficients at every time instance.
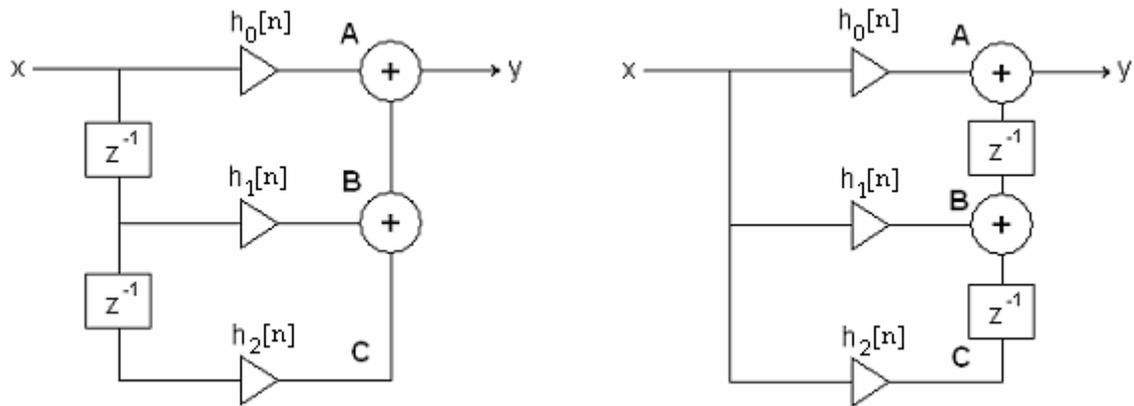
Figure 4.1 Direct form FIR structure (left) and transposed form FIR structure (right).

For the direct FIR structure (Fig. 4.1 left), the output y at time n is given by

$$y[n] = A[n] = x[n] \cdot h_0[n] + B[n],$$

where the node B is equal to

$$B[n] = x[n-1] \cdot h_1[n] + C[n],$$

and for the node C

$$C[n] = x[n-2] \cdot h_2[n].$$

The output y of the direct from FIR is therefore equal to

$$y[n] = x[n] \cdot h_0[n] + x[n-1] \cdot h_1[n] + x[n-2] \cdot h_2[n],$$

or more generally

$$y[n] = \sum_{k=0}^{N-1} x[n-k] \cdot h_k[n].$$

Now for the transposed form FIR structure (Fig. 4.1 Right), the output y is given by

$$y[n] = x[n] \cdot h_0[n] + B[n-1],$$

with the nodes B and C equal to

$$B[n] = x[n] \cdot h_1[n] + C[n-1]$$
$$C[n] = x[n] \cdot h_2[n]$$

and

$$C[n-1] = x[n-1] \cdot h_2[n-1]$$
$$B[n-1] = x[n-1] \cdot h_1[n-1] + x[n-2] \cdot h_2[n-2]$$

The output y at time n is then

$$y[n] = x[n] \cdot h_0[n] + x[n-1] \cdot h_1[n-1] + x[n-2] \cdot h_2[n-2],$$

or more generally

$$y[n] = \sum_{k=0}^{N-1} x[n-k] \cdot h_k[n-k].$$

Compared to the general equation for the direct form FIR output

$$y[n] = \sum_{k=0}^{N-1} x[n-k] \cdot h_k[n],$$

with the difference being the [n-k] index of the coefficient; meaning that the filters produce equivalent output only when the coefficients don't change with time. This means that, if the transposed FIR architecture is used, the LMS algorithm will not converge differently than when the direct implementation is used.

The first change needed is to account for the weight (or coefficient) reversal:

$$\hat{\mathbf{w}}(M-n+1) = \hat{\mathbf{w}}(M-n) + \frac{\mathbf{u}(n)e^*(n)}{\mu \cdot scale}$$

This would still converge slowly, however, because the error at the output is due to multiple past inputs and coefficients and not only one coefficient, as the direct form is. A suitable approximation is to update the weights at most every N inputs, where N is the length of the filter. This obviously will converge N times slower, though simulations show that it never actually converges with as good results as the traditional LMS algorithm. It may be acceptable still though, due to

the increased bandwidth of the transposed form FIR, when high convergence rates are not required [18].

For the RLS algorithm, there are also several modification needed. The constants are all multiplied by the scale factor. Similar to the learning rate constant of the LMS algorithm, the values of the gain vector $\mathbf{k}(n)$ of the RLS algorithm are less than unity. However, the inverse of a vector is undefined, so in this case we take the heuristic of multiplying it by an additional scale factor and dividing by an additional scale factor and accounting for this anywhere $\mathbf{k}$ is used. A summary of the modified algorithm follows.

$$\hat{\delta} = \delta \cdot scale,$$

$$\hat{\lambda} = \lambda \cdot scale,$$

$$\pi(n) = \mathbf{P}(n-1)\mathbf{u}(n),$$

$$\mathbf{k}(n) = \frac{scale^2 \cdot \pi(n)}{\hat{\lambda} + \mathbf{u}^H(n)\pi(n)},$$

$$\xi(n) = d(n) - \frac{\widehat{\mathbf{w}}^H(n-1)\mathbf{u}(n)}{scale},$$

$$\widehat{\mathbf{w}}(n) = \hat{\mathbf{w}}(n-1) + \frac{\mathbf{k}(n)\xi^*(n)}{scale},$$

and

$$\mathbf{P}(n) = \frac{\mathbf{P}(n-1) \cdot scale}{\hat{\lambda}} - \frac{\mathbf{k}(n)\mathbf{u}^H(n)\mathbf{P}(n-1)}{\hat{\lambda} \cdot scale^2}.$$

The equations here are those as described in Section 3.4.4 (pages 21-22) with the fixed-point modifications. Examining of the variables of the algorithm at random time steps and then applying algorithm 4.1 determined these changes. This was verified through software simulations.

4.3 Loadable Coefficient Filter Taps

The heart of any digital filter is the filter tap. This is where the multiplications take place and is therefore the main bottleneck in implementation. Many different schemes for fast multiplication in FPGAs have been devised, such as distributed arithmetic, serial-parallel multiplication, and Wallace trees [12], to name a few. Some, such as the distributed arithmetic technique, are optimized for situations where one of the multiplicands is to remain a constant value, and are referred to as constant coefficient multipliers (KCM)[13]. Though this is true for standard digital filters, it is not the case for an adaptive filter whose coefficients are updated with each discrete time sample. Consequently, an efficient digital adaptive filter demands taps with a fast variable coefficient multiplier (VCM).

A VCM can however obtain some of the benefits of a KCM by essentially being designed as a KCM that can reconfigure itself. In this case it is known as a dynamic constant coefficient multiplier (DKCM) and is a middle-way between KCMs and VCMs [13]. A DKCM offers the speed of a KCM and the reconfiguration of a DCM although utilizes more logic than either. This is a necessary price to pay however, for an adaptive filter.

4.3.1 Computed Partial Products Multiplication

An approach to multiplication that uses the LUTs or block RAM in an FPGA similarly to distributed arithmetic is partial products multiplication. Any efficient DKCM implementation in essence produces a matrix containing as rows, partial products or modified partial products [15]. Partial products multiplication is similar to conventional longhand multiplication for decimal numbers.

The method works with any radix and it directly affects the size of the ROM needed. A control bit is required for the most significant address so that the sign of the result will be correct. This is demonstrated with the stored coefficient 5 and the input multiplicand is -182. First, the partial products table (given in table 4.1) is generated by multiplying the coefficient by successive values. The negative values are included for the signed arithmetic to work correctly.

Table 4.1 Partial Products Table for Coefficient 5.

Address	Data	Value	Signed Binary
0000	0*C	0	0000 0000
0001	1*C	5	0000 0101
0010	2*C	10	0000 1010
0011	3*C	15	0000 1111
0100	4*C	20	0001 0100
0101	5*C	25	0001 1001
0110	6*C	30	0001 1110
0111	7*C	35	0010 0011
1000	0*C	0	0000 0000
1001	1*C	5	0000 0101
1010	2*C	10	0000 1010
1011	3*C	15	0000 1111
1100	-4*C	-20	1110 1100
1101	-5*C	-25	1111 0001
1110	-6*C	-30	1111 0110
1111	-7*C	-35	1111 1011

For example, to calculate 5 x (-182), the following is performed:

First, -182 is broken into signed binary octets: 101   001   010

Next, a control bit is added to the MSB: 1101 0001 0010

These address are given to the ROM and the values returned are 0000 1010, 0000 0101 and 1111 0001. These numbers are added but each octet is 3 bits more significant than the one below it, so the results must be shifted before addition. Sign extension for negative numbers is also required. The addition would therefore be:

$$00001010 \qquad \text{LSB Register:} 010$$
$$+ \underline{00000101}$$
$$00000110$$

$$00000110 \qquad \text{LSB Register:} 110$$
$$+ \underline{11110001}$$
$$11110001$$

The result is 11110001 and concatenated with the stored LSBs is 1100 0111 0010, which is −910 in decimal. If more precision is required the LUT contents can be increased and if the input bit width is larger the LUT can simply be accessed more times. The additions required can take place in serial or in parallel at the expense of more LUTs or higher latency.

This technique is better than distributed arithmetic because each look-up table is dependent only on one coefficient, not all coefficients. Further, the size of the look-up table can be manipulated by means of different radices. The size of the LUT required is important for two reasons. Obviously, a larger LUT will require more FPGA resources, but more importantly, it will take longer to reload with a new coefficient. The goal of an adoptive filter tap is to be able to reload quickly. In the example presented it would take only 16 clock cycles to reload the LUT contents.

4.3.2 Embedded Multipliers

As a final word on multiplication in FPGAs, it should be noted that many device manufacturers have been working on the problem. Because it is so important to most all DSP operations, and can be a major bottleneck, they are now offering dedicated embedded multipliers in some of the newer devices. The Virtex-II device by Xilinx can include up to 556 eighteen-bit (18x18) embedded multipliers. Altera has gone a step further by including up to 96 hard DSP blocks in its Stratix-II device. Each embedded DSP block includes four multipliers, adders, subtractors, accumulators and a summation unit. However, the success of the FPGA is due to its versatility, and the more specific the components inside them become, the less flexible the FPGA is. Since a Virtex-II Pro device was available for this research, the embedded multipliers were tested along with the implementations described earlier.

4.3.3 Tap Implementation Results

Of the DKCM architectures described, several were chosen and coded in VHDL to test their performance. Namely, the serial-parallel, partial products multiplication, and embedded multiplier are compared to ordinary CLB based multiplication inferred by the synthesis tool. All were designed for 12-bit inputs and 24-bit outputs. The synthesis results relevant to the number of slices flip-flops, 4 input LUTs, BRAMs, and embedded multipliers instantiated is offered in Appendix A. A comparison of the speed in Megahertz and resources used in terms of configurable logic blocks for the different implementations is presented in figure 4.2.
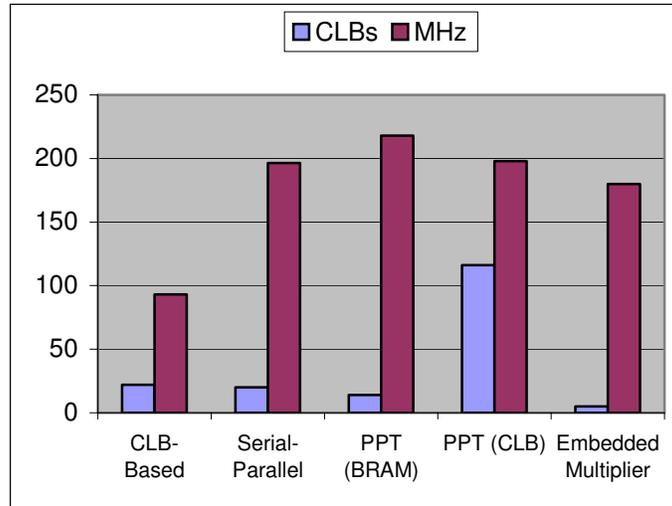
Figure 4.2 CLB Resources and Speed of Selected Tap Implementations

It would seem that the inclusion of embedded multipliers would make the previous discussion insignificant. However, they did not have the highest clock speed of the group. The fastest multiplier architecture was the partial products multiplier using embedded block RAM, followed by the partial products multiplier using CLBs only. The serial-parallel multiplier was the third fastest, but it takes 13 clock cycles to compute the result for 12 bit input data. The latency of the serial-parallel multiplier is directly dependent on the input bit width, and it does not pipeline well. In contrast, the partial products multiplier can add its results sequentially for a high latency and low cost or take as little as one clock cycle to complete at the cost of additional look-up tables.

Since the filter is adaptive and updates its coefficients at regular intervals, the time required to configure the tap for a new coefficient is important. The reconfiguration times for the various multipliers are listed in table 4.2. For the partial-products multiplier, the look-up table can be stored in the Configurable Logic Blocks (CLBs), or within the on-chip block RAM (BRAM). Using the BRAM, the partial products multiplier is 10% faster than with the CLBs.

Table 4.2 Reconfiguration Time and Speed for Different Multipliers

Architecture	Reconfiguration Time (clks)	Speed (MHz)
CLB-Based	1	93.075
Embedded Multiplier	1	179.988
Serial-Parallel	1	196.425
Partial Product (CLB)	16	197.902
Partial Product (BRAM)	16	217.96

The only shortcoming of the partial products multiplier is the higher reconfiguration time, but this can be overcome with the use of two separate look-up tables per tap. With this configuration, one LUT is used for computing results while the other is receiving new values. This configuration has the ultimate performance in terms of size and speed. A block diagram of this arrangement is shown in Figure 4.3 below.

Figure 4.3 Loadable Coefficient Partial Product Multiplier.

The signal Tap_Sel is asserted high when a new coefficient is ready. The Bank_Sel signal determines which look-up table the new data is written to and also which is being read from. The three multiplexers choose the correct signal to pass to the LUTs and also to output. For 24 bits of output precision, four of the units would be needed in parallel, or the four inputs could be presented serially and accumulated as described previously. Little additional logic is required to properly shift and add the outputs for the final result. The VHDL code describing this configuration is provided in Appendix B.

4.4 Embedded Microprocessor Utilization

The current trend in programmable logic is the inclusion of embedded DSP blocks and microprocessors. The Virtex-II Pro FPGA form Xilinx contains an embedded PowerPC 405 microprocessor, and numerous soft IP cores. To design for this environment the Embedded Development Kit must be used.

4.4.1 IBM PowerPC 405

The IBM PowerPC 405 is a 32-bit RISC microprocessor embedded in Xilinx's Virtex-II Pro FPGA. The core occupies a small die area and consumes minimal power making it ideal for system-on-chip (SoC) embedded applications. It can run at a clock speed of over 400 MHz to produce over 600 Dhrystone MIPS. A memory management unit (MMU), a 64-entry unified Translation Look-aside Buffers (TLB), debug support, and watchdog timers enable an embedded operating system to function for no additional logic cost.

4.4.2 Embedded Development Kit

To utilize the embedded PowerPC the Embedded Development Kit (EDK) from Xilinx must be used. EDK includes the tools necessary to instantiate the embedded microprocessor, as well as numerous soft IP cores, and an integrated C compiler. The engineer defines the system architecture in EDK and generates the netlists and HDL wrappers, and then writes the embedded software. EDK can then be used to generate the bitstream and download it to the FPGA, or alternatively, the netlists and HDL wrappers can be exported to an ISE project for place and route and bitstream generation.

4.4.3 Xilinx Processor Soft IP Cores

Soft Intellectual Property (IP) is a pre-designed netlist that can implement a variety of tasks. These connect to the microprocessor but are soft in the sense that they are instantiated in the FPGA fabric, i.e. look-up tables at synthesis time, and are not hard-wired.

A variety of soft IP cores are included for free with the Xilinx software, and other more complex or obscure functions can be bought from third parties or may be custom designed. Included IP are busses, memory interfaces, and peripherals, which together enable complete SoC designs. Example busses are the Processor Local Bus (PLB) and the On-chip Peripheral Bus (OPB). The IP cores attach to these busses to communicate with the PowerPC. Figure 4.4 shows an example PowerPC based SoC embedded design using these soft IP components.
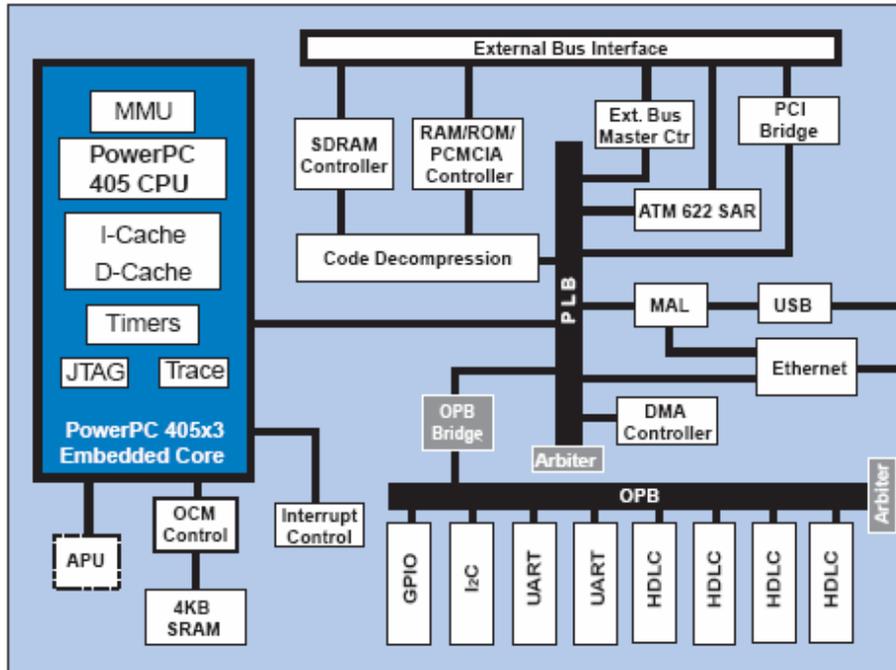
Figure 4.4 Example SoC embedded design with PPC405 core, and soft IP such as busses, memory interfaces, and peripherals [16].

4.4.3.1 User IP Cores

If a soft IP core doesn't exist to meet the design specifications, then a custom user core may be created. This is necessary as it is very difficult to communicate with the PowerPC from external FPGA logic unless the function is extremely simple (such as a clock divider). Xilinx uses the IP Interface (IPIF) to connect a core to the bus. The IPIF presents an interface called the IP Interconnect (IPIC) to the user logic while taking care of the bus interface signals, bus protocol, and other interface issues. Templates exist for OPB and PLB bus attachments, but due to bugs in the immature software, accomplishing this is not so straightforward. A block of the Xilinx IP Interface is shown in Figure 4.5.
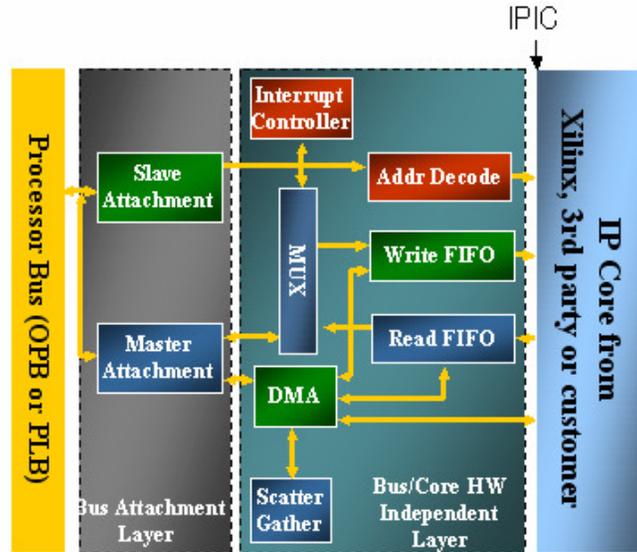
Figure 4.5 IPIF Block Diagram

To add a user core to an EDK project, one must first be create it by editing the provided reference design. The procedure for a PLB core is as follows:

1. Copy the plb_core_ssp0_ref_v1_00_a folder from C:\EDK\hw\XilinxReferenceDesigns\pcores\ to the local \pcores directory. Alternatively you can use the opb_core reference design.
2. Rename folder to the name of new core leaving "_v1_00_a" ie. user_core_v1_00_a
3. Rename pcores\user_core_v1_00_a\hdl\vhdl\plb_core_ssp0_ref.vhd to user_core.vhd
   - change library statement in vhdl
   - change entity and architecture declarations
4. Rename mpd and pao file in \data directory
   - change library statements at end of pao file
   - change BEGIN statement in mpd file
5. Add in Project->Add/Edit Cores
   - assign address range
   - add bus connection
   - add clock
   - override c_mir parameters in 'Parameters' section
6. Read/Write data to core:
   - XIo_Out32($ADDR, data);   // write
   - Input = XIo_In32($ADDR);   // read

40

The core IPIF drops address bits 30 and 31 and this must be dealt with in the user core. Data is passed to and from the core as if it were a memory location that the processor can read from and write to.

4.4.4 Adaptive Filter IP Core

The technique outlined in the previous section was used to create an adaptive filter IP core for use in an embedded PowerPC system. In this hybrid design, the microprocessor is used to handle memory transfers and give inputs to the FIR filter. The filter core performs the convolution and returns the result. The microprocessor runs the training algorithm in software and updates the filter core when new coefficients are available. Figure 4.6 shows a block diagram of the hybrid adaptive filter system.
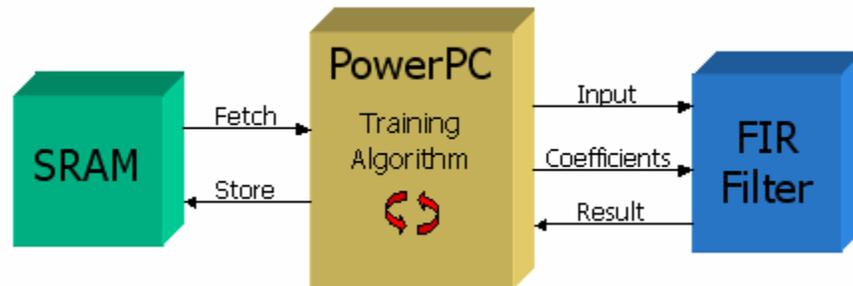
Figure 4.6 Hybrid Adaptive Filter Design.

In this arrangement, the filter acts similarly to a custom instruction for filtering data. The benefits of this design are that a complete system prototype can be built quickly by utilizing the high-level software for mundane tasks such as I/O, and also that the training algorithm used can be easily interchanged to evaluate their effectiveness. Of course, performance is slower but this can be overcome by training only at specific intervals (when signal statistics change), or by moving parallelizable sections of the training algorithm into the core as needed.

CHAPTER 5

Results

Several different implementations were tested, including hardware only designs as well as combined hardware/software embedded systems. This chapter gives an overview of the hardware verification method, presents the implementation results, and compares them to the results from Matlab trials.

5.1 Methods Used

Due to their inherent complexity, DSP algorithms are typically written in high-level languages and software packages such as Matlab. There is usually no emphasis on the hardware implementation until the algorithm is fully developed. This can lead to problems when coding the algorithm in a fixed-point format for hardware. The approach that has been taken is to verify the algorithm's output in the high level language with a fixed-point representation before hardware implementation. This was done according to the method outlined in Chapter 4, and ensures that the transition to a VHDL representation will be as easy as possible and that hardware results will be "bit-true" to the software simulation. This design practice requires more time in initial algorithm development but it is made up for in the implementation phase. Matlab Version 6.5 was used for the initial investigation, and also utilized in the final verification phase for its superb plotting capabilities.

Several different hardware configurations were designed all having in common a filter length of 16 and 16-bit precision. The length of 16 for the filter

was chosen, as it was able to support a direct-form FIR implementation at a frequency of over 100 MHz on the Virtex-II Pro, allowing the PowerPC to run at the maximum speed on the Avnet development board. The required 16-bit precision was determined through Matlab simulations.

A hybrid adaptive filter was designed with a direct-form FIR filter coded in VHDL and with the LMS algorithm written in C code executing on the PowerPC for training as well as the with the LMS algorithm designed in VHDL only. The transposed-form FIR structure was coded in VHDL, with the transposed-form LMS algorithm in C code and VHDL. Finally, an FPGA direct-form FIR was trained with the RLS algorithm coded for the PowerPC. A VHDL only RLS design was investigated, but some algorithm components were too demanding to meet timing requirements without significant reworking to include pipelining or more efficient structures. A summary of the filter designs implemented is given in table 5.1. The third column (PowerPC) indicates if the PowerPC was utilized for training algorithm. Even when the PowerPC is used for training and data passing, the filters were instantiated in the FPGA fabric. In this case, the PowerPC passes the data and coefficients to the VHDL core to do the filtering and return the results.

Table 5.1 Filter forms and training algorithms implemented.

Filter Form	Training Algorithm	PowerPC
FIR-Direct	LMS	N
FIR-Direct	LMS	Y
FIR-Transposed	Transposed-LMS	N
FIR-Transposed	Transposed-LMS	Y
FIR-Direct	RLS	Y

5.2 Algorithm Analyses

The algorithms for adaptive filtering were coded in Matlab and experimented to determine optimal parameters such as the learning rate for the LMS algorithm and the regularization parameter of the RLS algorithm. Next, the algorithms were converted to a fixed-point representation, and finally, coded for the Virtex-II Pro.

5.2.1 Full Precision Analysis

The application tested was adaptive noise cancellation, for reasons discussed in a previous chapter. This corresponds to figure 3.3 (d) on page 16. In the example presented a sine wave is the desired signal, but is corrupted by a higher frequency sinusoid and random Gaussian noise with a signal to noise ratio of 5.865 dB. A direct form FIR filter of length 16 is used to filter the input signal. The adaptive filter is trained with the LMS algorithm with a learning rate $\mu = 0.05$. The filter is also trained with the RLS algorithm with the parameters $\delta = 1$ and $\lambda = 0.99$.

The floating-point precision results are presented in figure 5.2. It appears that the filter trained with the LMS algorithm has learned the signals statistics and is filtering acceptable within 200 – 250 iterations. When trained with the RLS algorithm, the filters weights are near optimal within 50 training iterations, almost an order of magnitude faster, as expected.
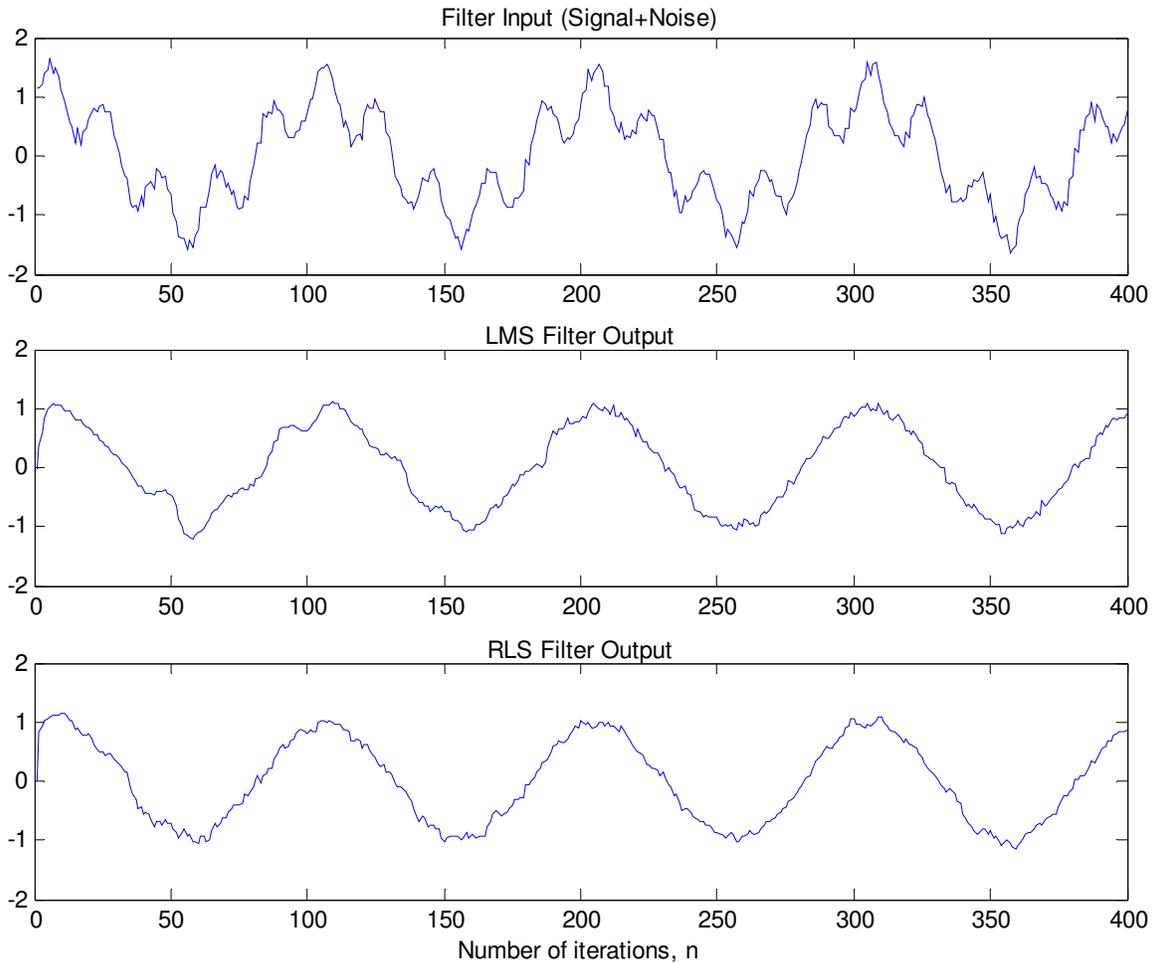
Figure 5.2 Input signal (top), and output of LMS (middle) and RLS (bottom).

Figure 5.3 displays the ensemble-averaged error of the LMS and RLS algorithm over 50 independent trials. Although the RLS algorithm converges to produce acceptable output rather quickly, a look at the error reveals that in this case, it converges to it minimum mean in approximately the same time as the LMS does, which is around 250 training iterations. After this point, both filters produce an output error of roughly 0.01. Consequently, if the convergence time is acceptable, then the LMS is preferred for its simplicity.
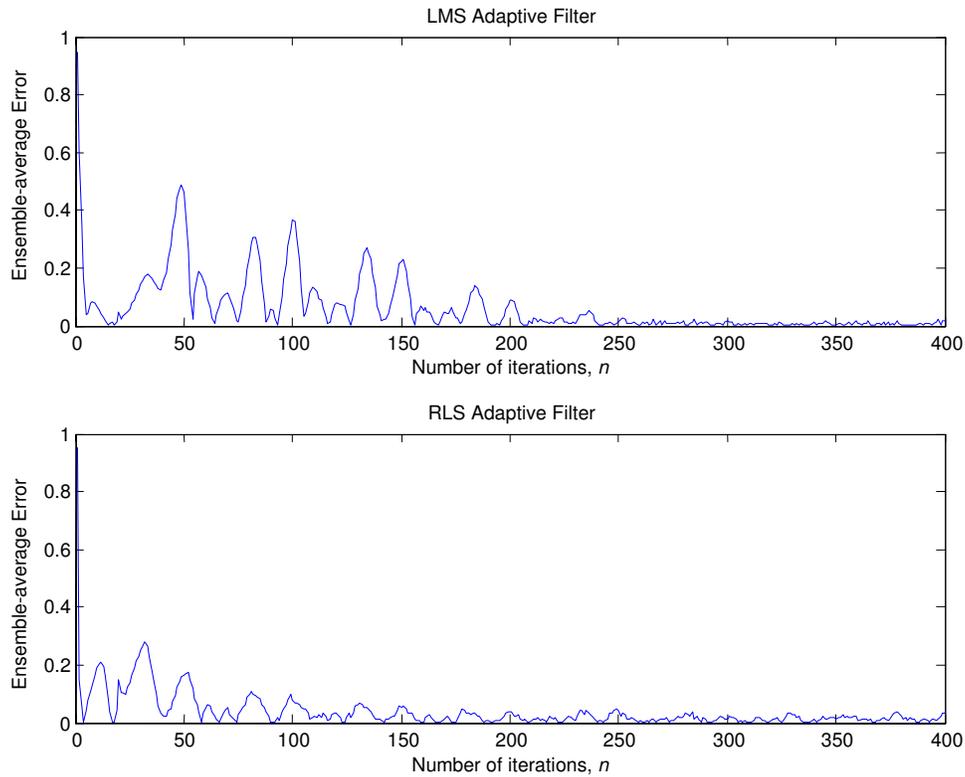
Figure 5.3 Plot of absolute error of the LMS (top) and RLS (bottom) algorithms.

5.2.2 Fixed-Point Analysis

The above algorithms were converted so that all internal calculations would be done with a fixed-point number representation. This is necessary, as the embedded PowerPC has no floating-point unit (FPU), and FPGA's don't natively support floating-point either. Although a FPU could be designed in an FPGA, they are resource intensive, and therefore can feasibly only support sequential operations. Doing so however would fail to take full advantage of the FPGA's major strength, which is high parallelization.

The LMS and RLS algorithms were modified as detailed in Chapter 4, and a transposed representation of the LMS was also implemented. A scale of 256 with 16-bit precision was found to be suitable. The results of the fixed-point LMS algorithm were comparable to the full precision representation of the same

algorithm. The RLS though, was considerably worse. The 16-bit fixed-point results are presented in Figure 5.4.
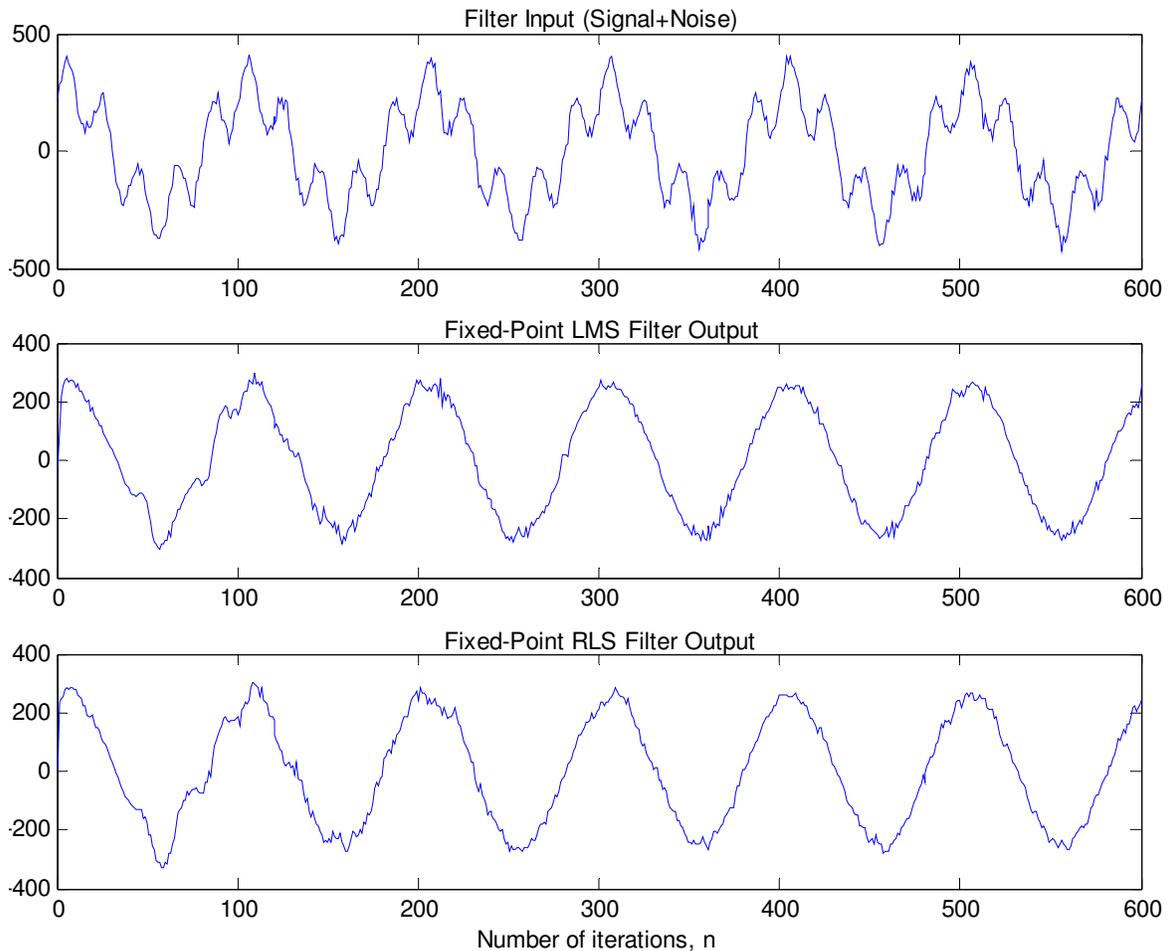
Figure 5.4 Fixed-Point Algorithm Results

For a 16-bit fixed representation, the RLS algorithm displayed a significant degradation as compared to the algorithm represented with floating-point accuracy. It appears that the RLS algorithm is very sensitive to the internal precision used, and therefore its superiority over the LMS algorithm is diminished when a fixed representation is needed. In fact, considering the extra computation needed, the RLS algorithm is barely better, yet requires significantly more development time due to its complexity.

Because the error at the output of a transposed-form FIR filter is due to the accumulation of past inputs and weight, it converges much differently than the direct form FIR. The transposed LMS algorithm takes much longer to converge and never converges close enough. However, it may be suitable when absolute data throughput is necessary. Figure 5.5 shows the output of a fixed-point transposed form LMS algorithm after approximately 2500 iterations. The corrupting sinusoid is completely removed, yet there is still considerable noise.
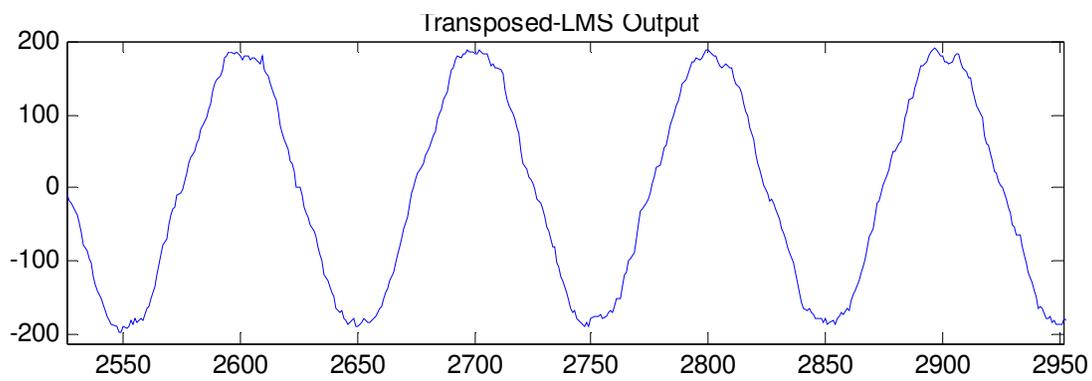
Figure 5.5 Transposed-form LMS output.

5.3 Hardware Verification

All the designs were thoroughly tested on the FPGA. The VHDL design using the FPGA fabric only was test as well as the hybrid designs using the FPGA fabric for filtering and utilizing the PowerPC for the training algorithm. To test the validity of the hardware results, an Avnet Virtex-II Pro Development kit with a Xilinx XC2VP20 FPGA was used. This board can plug directly into the PCI slot of a host computer for fast data transfer over the PCI bus. The included PCIUtility enabled this as well as quick FPGA reconfiguration over the PCI. The SRAM contents can be read back and stored in a .hex file, which could then be parsed to gather meaningful data. Although there are faster ways to do this, Matlab was again used, to maintain a continuous design flow, and take advantage of Matlab's plotting capabilities. The steps were as follows:

1. Generate input signals in Matlab.
2. Use Matlab to convert signals to hexadecimal format and write to .hex file.
3. Download .hex file to Avnet Board SRAM with PCIUtility.
4. Start filtering with command issued through HyperTerminal.
5. Read back results from SRAM with PCIUtility.
6. Parse .hex file with Matlab and check results.

This process is also represented graphically in Figure 5.6.
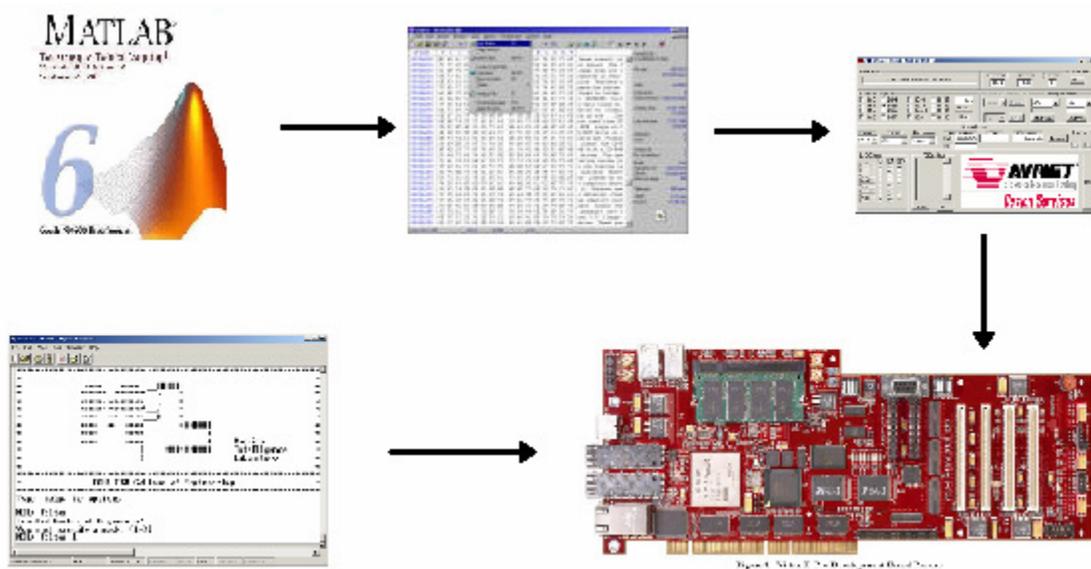
Figure 5.6 Data flow from Matlab to Avnet PCI board.

Since extensive effort was put forth ensuring the validity of the adaptive algorithms in a 16-bit fixed-point environment in advance, all hardware results were able to match precisely with software simulations.

5.4 Power Consumption

When choosing between different designs for embedded applications, the power consumed by the device is an important issue. The power consumed by the different architectures is displayed in Figure 5.7. Designs utilizing the PowerPC microprocessor averaged 920 mW of power use, while designs using

FPGA logic only averaged 757 mw. On average, the inclusion of the PowerPC uses 164 more mW than design without it.
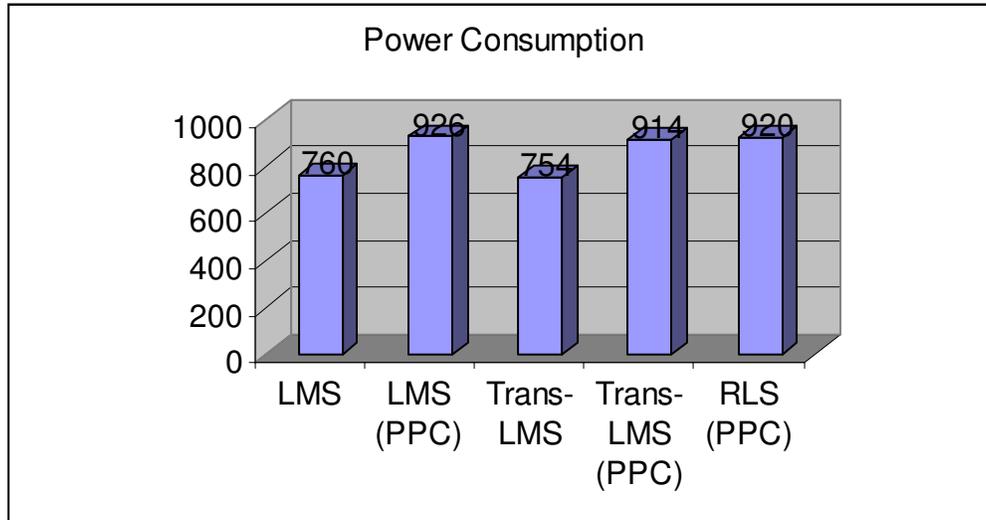
Figure 5.7 Power Consumption (mW) of different architectures.

The Xilinx website reports that the PowerPC 405 microprocessor consumes 0.9 mW/MHz, but this appears to be a conservative estimate. The actual power consumed averaged 1.6 mW/MHz for a PowerPC system, but the discrepancy is most likely due to the power consumed by the PLB (Processor Local Bus), OCM (On Chip Memory) bus for both the data and instruction sides, and the DCR (Device Control Register) bus.

5.5. Bandwidth Considerations

An important consideration is the available throughput of the filter architecture. For training algorithms executed on the serial microprocessor, this is limited by the number of operations required between weight updates. For the LMS algorithm, the equation is one line:

$$\hat{\mathbf{w}}(n+1) = \hat{\mathbf{w}}(n) + \frac{\mathbf{u}(n)e^*(n)}{\hat{\mu}}.$$

This requires N additions, N multiplications, and N divisions. The PowerPC does not have a single cycle multiply-accumulate instruction, so this would take 3*N cycles before the next input can be processed. For a 16-tap filter operating at 200 MHz, the effective throughput would be 4.16 MHz.

The RLS algorithm has many more calculations per weight update. The number of operations per function is given in table 5.2. In the count, additions and subtractions are considered equivalent, as are multiplications and divisions. The number of operations required for an N-tap filter is $5 \times N^2$ mults and $2 \times N^2$ adds, $5 \times N$ mults and $4 \times N - 1$ adds. Since multiplications and additions each take one clock cycle to complete, this can be simplified to $7 \times N^2 + 9 \times N$ operations per coefficient update, with N-1 adds approximating to N operations for simplicity. For a 16-tap filter operating at 200 MHz, the effective throughput would be 101.6 kHz.

Table 5.2 Number of Operations for RLS Algorithm

Equation	Operations
$\pi(n) = \mathbf{P}(n-1)\mathbf{u}(n)$	$N \times (N \text{ mults and } N - 1 \text{ adds})$
$\mathbf{k}(n) = \dfrac{scale^2 \cdot \pi(n)}{\hat{\lambda} + \mathbf{u}^H(n)\pi(n)}$	$3 \times N$ mults and $2 \times N - 1$ adds
$\widehat{\mathbf{w}}(n) = \hat{\mathbf{w}}(n-1) + \dfrac{\mathbf{k}(n)\xi^*(n)}{scale}$	N mults and N adds
$\mathbf{P}(n) = \dfrac{\mathbf{P}(n-1) \cdot scale}{\hat{\lambda}} - \dfrac{\mathbf{k}(n)\mathbf{u}^H(n)\mathbf{P}(n-1)}{\hat{\lambda} \cdot scale^2}$	$4 \times N^2$ mults and $2 \times N^2$ adds N, N mults & N - 1 adds.

CHAPTER 6

Conclusions

Many lesson were learned undertaking this research, and new ideas were found. This final chapter will discuss the conclusions and also the future work of this thesis.

6.1 Conclusions

A significant amount of design time was spent working out minor bugs and intricacies within the EDK software. In fact, the extra time spent debugging cancelled out the amount of time saved by coding in a high-level language. Hopefully, these bugs will be addressed in future software releases. In addition, reference designs provided by Avnet were used for the memory transfer to the SRAM through the PCI bus. Since the PCI bus and the PPC405 share the SRAM, a control bit and machine interrupt are used. This bit needed to be set manually during testing. It is unclear whether this is due to software or hardware errors.

Until an appropriate fixed-point structure is found for the Recursive Least-Squares algorithm, the Least Mean-Square algorithm was found to be the most efficient training algorithm for FPGA based adaptive filters. The issue of whether to train in hardware or software is based on bandwidth needed and power specifications, and is dependent on the complete system being designed.

While the extra power consumed would make the PowerPC seem unattractive, as part of a larger embedded system this could be practical. If

many processes can share the PowerPC then the extra power would be mitigated by the creation of extra hardware that it has avoided. Furthermore, an area greatly simplified by the inclusion of a microprocessor is memory transfers. These are sequential in nature and within the EDK there are many memory IP modules. With no microprocessor, a finite state machine for timing, as well as a memory interface is needed, and these will consume more power, although still less than the PowerPC. Lastly, the microprocessor can be used to easily swap out software training algorithms for application testing and evaluation.

Embedded microprocessors within FPGA's are opening up many new possibilities for hardware engineers, although it requires a new design process. The future of embedded Systems-on-Chip design will involve more precisely determining the optimal hardware and software tradeoffs for the functionality needed.

6.2 Future Work

The fixed-point RLS algorithm implementations performed very poorly, and therefore a variant of the algorithm that converges acceptable with fixed-point precision is needed. Papaodysseus, et al, have proposed such an algorithm [19], which is said to be less sensitive to numerical precision. Furthermore, the authors claim it parallelizes well. It would be interesting to explore this further.

High-level design languages for programmable logic could help to ease the design flow of difficult algorithms. However, many are still in early stages and have trouble inferring device specific components, such as microprocessors. Evaluating these new languages, and their integration with embedded microprocessors, would be useful.

Many times the decision to use an embedded microprocessor focuses on the power consumed and logic cells saved. Therefore, a framework for determining the amount of equivalent logic cells recovered for a given function on the microprocessor is needed.

APPENDIX A

Matlab Code

LMS Adaptive Noise Cancellation

```
% Description:
%   Adaptive filter trained with the LMS algorithm. A sinusoidal
%   signal corrupted with an unwanted frequency and random noise
%   is filtered.  The desired signal is used to adapt the weights.

%--- Filter Parameters ---
M  = 20;        % number of taps
mu = 0.05;      % step-size parameter
e_max = 400;  % maximum # of epochs

%--- Constants ---
pi = 3.14;
Fs = 0.01;        % signal frequency
Fn = 0.05;        % noise frequency

%--- Initialize ---
w = (randn(1,M)-randn(1,M))/100;
d = zeros(1,M);
u = zeros(1,M);
u_out = zeros(1,e_max-M);
f_out = zeros(1,e_max-M);

% Generate desired signal and input(signal+noise)
for t = 1:M-1
    d(t) = sin(2*pi*Fs*t);
    u(t) = d(t)+0.5*sin(2*pi*Fn*t)+0.09*randn;
end
t = M;
epoch = 0;

while epoch < e_max
    % Generate new input
    input = sin(2*pi*Fs*t);
```

```
  % Shift new input into array
  for i = 2:M
     d(M-i+2) = d(M-i+1);
     u(M-i+2) = u(M-i+1);
  end
  d(1) = input;
  % Add undesired freq & random noise
  u(1) = input+0.5*sin(2*pi*Fn*t)+0.09*randn;
  u_out(t-M+1) = u(1);

  % Compute filter output
  output = dot(w,u);
  f_out(t-M+1) = output;

  %--------------------- LMS Algorithm ---------------------
  % Compute error
  e = d(1) - output;

  % Update weights
  for n = 1:M
     w(n) = w(n)+mu*u(n)*e;
  end
  %-----------------------------------------------------------

  in(t-M+1)  = u(1);
  out(t-M+1) = output;
  err(t-M+1) = e;

  t = t+1;
  epoch = epoch +1;

  %--- Plot noise and filtered signal ---
  subplot(211), plot(t,u(1)), axis([0 e_max -2.5 2.5]), title('Filter Input (Signal + Noise)'),
drawnow, hold on
  subplot(212), plot(t,output), axis([0 e_max -2.5 2.5]), title('Filtered Signal'), drawnow,
hold on

end
```

RLS Adaptive Noise Cancellation

```
% Description:
%   Adaptive filter trained with the RLS algorithm. A sinusoidal
%   signal corrupted with an unwanted frequency and random noise
%   is filtered.  The desired signal is used to adapt the weights.
```

```matlab
%--- Filter Parameters ---
M = 20;          % number of taps
delta = 1;
lamda = 0.99;
e_max = 400;     % maximum # of epochs

%--- Constants ---
pi = 3.14;
Fs = 0.01;       % signal frequency
Fn = 0.05;       % noise frequency

%--- Initialize ---
w = zeros(M,1);
d = zeros(M,1);
u = zeros(M,1);
P = eye(M)/delta;

% Generate desired signal and input(signal+noise)
for t = 1:M-1
   d(t) = sin(2*pi*Fs*t);
   u(t) = d(t)+0.5*sin(2*pi*Fn*t)+0.09*randn;
end
t = M;
epoch = 0;

while epoch < e_max
   % Generate new input
   input = sin(2*pi*Fs*t);

   % Shift new input into array
   for i = 2:M
      d(M-i+2) = d(M-i+1);
      u(M-i+2) = u(M-i+1);
   end
   d(1) = input;
   % Add undesired freq & random noise
   u(1) = input+0.5*sin(2*pi*Fn*t)+0.09*randn;

   % Compute filter output
   output = w'*u;

   %----------------- RLS Algorithm --------------------
   k = ( P*u )/(lamda+u'*P*u);

   E = d(1) - w'*u;
```

```matlab
   w = w + k*E;

   P = (P/lamda) - (k*u'*P/lamda);
   %----------------------------------------------------

   in(t-M+1)  = u(1);
   out(t-M+1) = output;
   err(t-M+1) = E;

   t = t+1;
   epoch = epoch +1;

   %--- Plot noise and filtered signal ---
   subplot(211), plot(t,u(1)), axis([0 e_max -2.5 2.5]), title('Filter Input (Signal + Noise)'),
drawnow, hold on
   subplot(212), plot(t,output), axis([0 e_max -2.5 2.5]), title('Filtered Signal'), drawnow,
hold on
end
```

APPENDIX B

VHDL Code

Partial Products Multiplier Tap

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity tap is
    Port ( clk          : in std_logic;
           rst          : in std_logic;
           Write_Data   : in std_logic_vector(14 downto 0);
           Bank_Sel     : in std_logic;
           Tap_Sel      : in std_logic;
           Write_Addr   : in std_logic_vector(3 downto 0);
           Read_Data    : in std_logic_vector(3 downto 0);
           Data_Valid   : out std_logic;
           Tap_out      : out std_logic_vector(23 downto 0);
           NT_tsb       : out std_logic_vector(3 downto 0)     );
end tap;
---------------------------------------------------------------------
architecture Structural of tap is
component tsb is
    Port ( clk          : in std_logic;
           data_in      : in std_logic_vector(3 downto 0);
           data_out     : out std_logic_vector(3 downto 0);
           first_oct     : out std_logic  );
end component;

component tap_lut is
    Port ( clk          : in std_logic;
           write_en     : in std_logic;
           Write_Data   : in std_logic_vector(14 downto 0);
           lut_input    : in std_logic_vector(3 downto 0);
           lut_output   : out std_logic_vector(14 downto 0) );
end component;

component mux_reg15 is
```

```vhdl
    Port ( clk          : in std_logic;
           rst          : in std_logic;
           sel          : in std_logic;
           mux_in15a    : in std_logic_vector(14 downto 0);
           mux_in15b    : in std_logic_vector(14 downto 0);
           mux_out15    : out std_logic_vector(14 downto 0) );
end component;

component pp_adder is
    Port ( clk   : in std_logic;
           rst   : in std_logic;
           f_oct : in std_logic;
           valid : out std_logic;
           input : in std_logic_vector(14 downto 0);
           output : out std_logic_vector(23 downto 0) );
end component;

signal tsb_out        : std_logic_vector(3 downto 0);
signal mux_1_out      : std_logic_vector(3 downto 0);
signal mux_2_out      : std_logic_vector(3 downto 0);
signal mux_3_out      : std_logic_vector(14 downto 0);

signal First_oct      : std_logic;
signal lut1_we        : std_logic;
signal lut2_we        : std_logic;
signal lut_1_out      : std_logic_vector(14 downto 0);
signal lut_2_out      : std_logic_vector(14 downto 0);

--------------------------------------------------------------------------------
begin

TSB_1 : tsb port map(clk, Read_Data, tsb_out, First_oct);

--------------------- Partial Product Multiplier Section ---------------------
MUX_1 :
        with Bank_Sel select
               mux_1_out <= Write_Addr when '0',
                                tsb_out when others;


MUX_2 :
        with Bank_Sel select
               mux_2_out <= Write_Addr when '1',
                                tsb_out when others;

lut1_we <= not Bank_sel and Tap_Sel;
lut2_we <= Bank_sel and Tap_Sel;
```

```vhdl
LUT_1 : tap_lut port map(clk, lut1_we, Write_Data, mux_1_out, lut_1_out);

LUT_2 : tap_lut port map(clk, lut2_we, Write_Data, mux_2_out, lut_2_out);

------------------------ Scaling Accumulator Section -------------------------

MUX_3 : mux_reg15 port map(clk, rst, Bank_Sel, lut_1_out, lut_2_out, mux_3_out);

ADDER : pp_adder port map(clk,rst,First_oct,Data_Valid,mux_3_out,Tap_out);

NT_tsb  <= tsb_out;

end Structural;

-- tsb component
architecture Behavioral of tsb is
signal fo1, fo2        : std_logic;
signal data0           : std_logic_vector(3 downto 0);
signal data1           : std_logic_vector(3 downto 0);
signal data2           : std_logic_vector(3 downto 0);
signal data3           : std_logic_vector(3 downto 0);
begin
process(clk)
  begin
        if clk'event and clk = '1' then
                data0   <= data_in;
                data1   <= data0;
                data2   <= data1;
                data3   <= data2;
                data_out  <= data3;
                -- Assert control bit for first octect
                -- delay for 2 clk's
                fo1             <= data3(3);
                fo2             <= fo1;
                first_oct       <= fo2;
        end if;
end process;
end Behavioral;


-- tap_lut Component
architecture Behavioral of tap_lut is
type lut_array is array (0 to 15) of std_logic_vector(14 downto 0);
signal lut_contents : lut_array;
begin
process(clk, write_en)
 variable addr : integer range 0 to 15;
```

```vhdl
begin
  if clk'event and clk = '1' then
        addr := CONV_INTEGER(lut_input);
        if write_en = '1' then
                lut_contents(addr) <= Write_Data;
                lut_output      <= (others => '0');
        else
                lut_output <= lut_contents(addr);
        end if;
  end if;
end process;
end Behavioral;


-- mux_reg15 Component
architecture Behavioral of mux_reg15 is
signal mux_out15_sig : std_logic_vector(14 downto 0);
begin
with sel select
        mux_out15_sig <= mux_in15a when '1',
                                mux_in15b when others;
REG: process(clk,rst)
begin
        if rst = '1' then
                mux_out15     <= (others => '0');
        elsif clk'event and clk = '1' then
                mux_out15 <= mux_out15_sig;
        end if;
end process;
end Behavioral;


--pp_adder Component
architecture Behavioral of pp_adder is
signal f_o_delay       : std_logic;
signal Adder_sig       : std_logic_vector(17 downto 0);
signal LSB_0           : std_logic_vector(2 downto 0);
signal LSB_1           : std_logic_vector(2 downto 0);
signal LSB_2           : std_logic_vector(2 downto 0);
begin
process(clk, input, rst)
        variable s_extend, a_extend : std_logic_vector(2 downto 0);
begin
        if rst = '1' then
                Adder_sig     <= (others => '0');
                LSB_0                 <= (others => '0');
                LSB_1                 <= (others => '0');
                LSB_2                 <= (others => '0');
```

```vhdl
        elsif clk'event and clk = '1' then
                -- Sign extend for input and Adder_sig
                for i in 2 downto 0 loop
                        s_extend(i)     := input(14);
                        a_extend(i) := Adder_sig(17);
                end loop;
                f_o_delay <= f_oct;
                if f_o_delay = '1' then
                        Adder_sig <= ( s_extend & input );
                        LSB_0                   <= (others => '0');
                        LSB_1                   <= (others => '0');
                        LSB_2                   <= input(2 downto 0);
                        valid                   <= '1';
                else
        Adder_sig       <= ( a_extend & Adder_sig(17 downto 3) ) + ( s_extend & input );
                        LSB_0                   <= LSB_1;
                        LSB_1                   <= LSB_2;
                        LSB_2                   <= Adder_sig(2 downto 0);
                        valid                   <= '0';
                end if;
                output  <= Adder_sig(14 downto 0) & LSB_2 & LSB_1 & LSB_0;
        end if;
end process;
end Behavioral;
```

LMS Adaptive Filter

```vhdl
--      Adaptive Filter Package
library IEEE;
use IEEE.STD_LOGIC_1164.all;


package ad_filt is


-- Declare constants
  constant NUM_TAPS         : integer := 16:
  constant mu               : integer := 16;
   type sh_reg is array (0 to NUM_TAPS-1) of std_logic_vector(15 downto 0);
  type tmp_reg is array (0 to NUM_TAPS-1) of std_logic_vector(31 downto 0);
 end ad_filt;



--  LMS adaptive Filter
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
library work;
```

```vhdl
use work.ad_filt.ALL;

entity lms_ad_filt is
    Port ( clk          : in std_logic;
           rst          : in std_logic;
           train        : in std_logic;  -- 1 for yes, 0 for no
           data_in      : in std_logic_vector(15 downto 0);
           new_data     : in std_logic_vector(15 downto 0);
           desired      : in std_logic_vector(15 downto 0);
           data_out     : out std_logic_vector(15 downto 0)  );
end lms_ad_filt;

architecture Behavioral of lms_ad_filt is
 type states is (s0,s1,s2,s3,s4);
 signal state : states;

 signal coeffs   : sh_reg;
 signal inputs   : sh_reg;
 signal tmp      : tmp_reg;
 signal tmp1     : tmp_reg;
 signal add_0         : std_logic_vector(31 downto 0);
 signal add_1         : std_logic_vector(31 downto 0);
 signal desired_0r    : std_logic_vector(15 downto 0);
 signal desired_1r    : std_logic_vector(15 downto 0);
 signal error         : std_logic_vector(15 downto 0);
 signal nd_0, nd_1    : std_logic_vector(15 downto 0);

begin

 process(clk, rst, new_data)
   variable output : std_logic_vector(31 downto 0);
  begin
   if rst = '1' then
state <= s0;
        for i in 0 to NUM_TAPS-1 loop
                inputs(i)       <= (others=>'0');
                tmp(i)          <= (others=>'0');
                coeffs(i)       <= x"0001";
        end loop;
        data_out        <= (others=>'0');
        error           <= (others=>'0');
        desired_0r      <= (others=>'0');
   elsif rising_edge(clk) then
nd_0 <= new_data;
        nd_1 <= nd_0;
        case state is
```

```
    when s0 =>
       if (nd_0 /= nd_1) then  -- new data available
           state <= s1;
           -- register desired output
   desired_0r <= desired;
           desired_1r <= desired_0r;
           -- shift down input array
           for i in NUM_TAPS-1 downto 1 loop
               inputs(i) <= inputs(i-1);
           end loop;

           -- put new value in array
           inputs(0) <= data_in;
           -- compute direct form FIR
           for i in 0 to NUM_TAPS-1 loop
               tmp(i) <= inputs(i) * coeffs(i);
           end loop;
       else
           -- wait for new data
           state <= s0;
       end if;
    when s1 =>
       state <= s2;
       -- first adder stage
       add_0 <= tmp(0)+tmp(1)+tmp(2)+tmp(3);
       add_1 <= tmp(4)+tmp(5)+tmp(6)+tmp(7);
    when s2 =>
       -- second adder stage
       output   := add_0 + add_1;
       data_out <= output(22 downto 7);
       if train = '1' then
           -- compute error
           error <= desired_1r - output(22 downto 7);        --divide by scale
           state <= s3;
       else
           state <= s0;
       end if;
    when s3 =>
       state <= s4;
       -- update coefficients        - first stage
       for i in 0 to NUM_TAPS-1 loop
           tmp1(i) <= inputs(i) * error;
       end loop;
   when s4 =>
       state <= s0;
       -- update coefficients
```

```vhdl
          for i in 0 to NUM_TAPS-1 loop
                coeffs(i) <= coeffs(i) + tmp1(i)(26 downto 11);--divide by scale & mu
          end loop;
      when others => state <= s0;
    end case;
  end if;
end process;
end Behavioral;
```

Direct Form FIR Filter Core

```vhdl
entity user_logic is
 port (
   Bus2IP_Addr  : in  std_logic_vector(0 to 31);
   Bus2IP_Clk   : in  std_logic;
   Bus2IP_CS    : in  std_logic;
   Bus2IP_Data  : in  std_logic_vector(0 to 31);
   Bus2IP_RdCE  : in  std_logic;
   Bus2IP_Reset : in  std_logic;
   Bus2IP_WrCE  : in  std_logic;
   IP2Bus_Data  : out std_logic_vector(0 to 31) );
end entity user_logic;
-------------------------------------------------------------------------------
-- Architecture section
-------------------------------------------------------------------------------
architecture IMP of user_logic is
-------------------------------------------------------------------------------
-- Component Declaration
-------------------------------------------------------------------------------
component filter is
   Port ( clk : in std_logic;
          rst      : in std_logic;
          new_data : in std_logic_vector(15 downto 0);
          valid  : in std_logic_vector(15 downto 0);
        coef_0 : in std_logic_vector(15 downto 0);
        coef_1 : in std_logic_vector(15 downto 0);
        coef_2 : in std_logic_vector(15 downto 0);
        coef_3 : in std_logic_vector(15 downto 0);
        coef_4 : in std_logic_vector(15 downto 0);
        coef_5 : in std_logic_vector(15 downto 0);
        coef_6 : in std_logic_vector(15 downto 0);
        coef_7 : in std_logic_vector(15 downto 0);
        coef_8 : in std_logic_vector(15 downto 0);
        coef_9 : in std_logic_vector(15 downto 0);
        coef_10 : in std_logic_vector(15 downto 0);
        coef_11 : in std_logic_vector(15 downto 0);
        coef_12 : in std_logic_vector(15 downto 0);
```

```vhdl
        coef_13 : in std_logic_vector(15 downto 0);
        coef_14 : in std_logic_vector(15 downto 0);
        coef_15 : in std_logic_vector(15 downto 0);
        result : out std_logic_vector(31 downto 0)  );
end component;
-------------------------------------------------------------------------------
-- Signal declarations
-------------------------------------------------------------------------------
signal addr    : std_logic_vector(0 to 4);
signal reg_0   : std_logic_vector(0 to 31);
signal reg_1   : std_logic_vector(0 to 31);
signal reg_2   : std_logic_vector(0 to 31);
signal reg_3   : std_logic_vector(0 to 31);
signal reg_4   : std_logic_vector(0 to 31);
signal reg_5   : std_logic_vector(0 to 31);
signal reg_6   : std_logic_vector(0 to 31);
signal reg_7   : std_logic_vector(0 to 31);
signal reg_8   : std_logic_vector(0 to 31);
signal reg_9   : std_logic_vector(0 to 31);
signal reg_10  : std_logic_vector(0 to 31);
signal reg_11  : std_logic_vector(0 to 31);
signal reg_12  : std_logic_vector(0 to 31);
signal reg_13  : std_logic_vector(0 to 31);
signal reg_14  : std_logic_vector(0 to 31);
signal reg_15  : std_logic_vector(0 to 31);
signal reg_16  : std_logic_vector(0 to 31);
signal reg_17  : std_logic_vector(0 to 31);
signal reg_18  : std_logic_vector(0 to 31);
signal reg_19  : std_logic_vector(0 to 31);
signal reg_20  : std_logic_vector(0 to 31);
signal reg_21  : std_logic_vector(0 to 31);
signal reg_22  : std_logic_vector(0 to 31);
signal reg_23  : std_logic_vector(0 to 31);
signal reg_24  : std_logic_vector(0 to 31);
signal reg_25  : std_logic_vector(0 to 31);
signal reg_26  : std_logic_vector(0 to 31);
signal reg_27  : std_logic_vector(0 to 31);
signal reg_28  : std_logic_vector(0 to 31);
signal reg_29  : std_logic_vector(0 to 31);
signal reg_30  : std_logic_vector(0 to 31);
signal reg_31  : std_logic_vector(0 to 31);
-------------------------------------------------------------------------------
-- Begin architecture
-------------------------------------------------------------------------------

begin -- architecture IMP
```

```vhdl
   addr <= Bus2IP_Addr(25 to 29);    -- bits 30 & 31 are dropped
REGS_PROC: process(Bus2IP_Clk) is
begin
  if Bus2IP_Clk'event and Bus2IP_Clk='1' then
   if Bus2IP_Reset = '1' then
                reg_0 <= (others => '0');
                reg_1 <= (others => '0');
                reg_2 <= (others => '0');
                reg_3 <= (others => '0');
                reg_4 <= (others => '0');
                reg_5 <= (others => '0');
                reg_6 <= (others => '0');
                reg_7 <= (others => '0');
                reg_8 <= (others => '0');
                reg_9 <= (others => '0');
                reg_10 <= (others => '0');
                reg_11 <= (others => '0');
                reg_12 <= (others => '0');
                reg_13 <= (others => '0');
                reg_14 <= (others => '0');
                reg_15 <= (others => '0');
                reg_16 <= (others => '0');
                reg_17 <= (others => '0');
--              reg_18 <= (others => '0');  result
                reg_19 <= (others => '0');
                reg_20 <= (others => '0');
                reg_21 <= (others => '0');
                reg_22 <= (others => '0');
                reg_23 <= (others => '0');
                reg_24 <= (others => '0');
                reg_25 <= (others => '0');
                reg_26 <= (others => '0');
                reg_27 <= (others => '0');
                reg_28 <= (others => '0');
                reg_29 <= (others => '0');
                reg_30 <= (others => '0');
                reg_31 <= (others => '0');
    else
     if Bus2IP_CS = '1' and Bus2IP_WrCE = '1' then
       case addr is
           -- Address in EDK
           when "00000" => reg_0 <= Bus2IP_Data;          -- 00
           when "00001" => reg_1 <= Bus2IP_Data;          -- 04
           when "00010" => reg_2 <= Bus2IP_Data;          -- 08
           when "00011" => reg_3 <= Bus2IP_Data;          -- 0C
           when "00100" => reg_4 <= Bus2IP_Data;          -- 10
```

```vhdl
          when "00101" => reg_5 <= Bus2IP_Data;           -- 14
          when "00110" => reg_6 <= Bus2IP_Data;           -- 18
          when "00111" => reg_7 <= Bus2IP_Data;           -- 1C
          when "01000" => reg_8 <= Bus2IP_Data;           -- 20
          when "01001" => reg_9 <= Bus2IP_Data;           -- 24
          when "01010" => reg_10 <= Bus2IP_Data;          -- 28
          when "01011" => reg_11 <= Bus2IP_Data;          -- 2C
          when "01100" => reg_12 <= Bus2IP_Data;          -- 30
          when "01101" => reg_13 <= Bus2IP_Data;          -- 34
          when "01110" => reg_14 <= Bus2IP_Data;          -- 38
          when "01111" => reg_15 <= Bus2IP_Data;          -- 3C
          when "10000" => reg_16 <= Bus2IP_Data;          -- 40
          when "10001" => reg_17 <= Bus2IP_Data;          -- 44
--        when "10010" => reg_18 <= Bus2IP_Data;          -- 48
          when "10011" => reg_19 <= Bus2IP_Data;          -- 4C
          when "10100" => reg_20 <= Bus2IP_Data;          -- 50
          when "10101" => reg_21 <= Bus2IP_Data;          -- 54
          when "10110" => reg_22 <= Bus2IP_Data;          -- 58
          when "10111" => reg_23 <= Bus2IP_Data;          -- 5C
          when "11000" => reg_24 <= Bus2IP_Data;          -- 60
          when "11001" => reg_25 <= Bus2IP_Data;          -- 64
          when "11010" => reg_26 <= Bus2IP_Data;          -- 68
          when "11011" => reg_27 <= Bus2IP_Data;          -- 6C
          when "11100" => reg_28 <= Bus2IP_Data;          -- 70
          when "11101" => reg_29 <= Bus2IP_Data;          -- 74
          when "11110" => reg_30 <= Bus2IP_Data;          -- 78
          when "11111" => reg_31 <= Bus2IP_Data;          -- 7C
          when others => null;
      end case;
    end if;
   end if;
  end if;
end process REGS_PROC;

OUT_MUX: process(addr) is
begin
   case addr is
        when "00000" => IP2Bus_Data <= reg_0;
        when "00001" => IP2Bus_Data <= reg_1;
        when "00010" => IP2Bus_Data <= reg_2;
        when "00011" => IP2Bus_Data <= reg_3;
        when "00100" => IP2Bus_Data <= reg_4;
        when "00101" => IP2Bus_Data <= reg_5;
        when "00110" => IP2Bus_Data <= reg_6;
        when "00111" => IP2Bus_Data <= reg_7;
        when "01000" => IP2Bus_Data <= reg_8;
```

```vhdl
        when "01001" => IP2Bus_Data <= reg_9;
        when "01010" => IP2Bus_Data <= reg_10;
        when "01011" => IP2Bus_Data <= reg_11;
        when "01100" => IP2Bus_Data <= reg_12;
        when "01101" => IP2Bus_Data <= reg_13;
        when "01110" => IP2Bus_Data <= reg_14;
        when "01111" => IP2Bus_Data <= reg_15;
        when "10000" => IP2Bus_Data <= reg_16;
        when "10001" => IP2Bus_Data <= reg_17;
        when "10010" => IP2Bus_Data <= reg_18;
        when "10011" => IP2Bus_Data <= reg_19;
        when "10100" => IP2Bus_Data <= reg_20;
        when "10101" => IP2Bus_Data <= reg_21;
        when "10110" => IP2Bus_Data <= reg_22;
        when "10111" => IP2Bus_Data <= reg_23;
        when "11000" => IP2Bus_Data <= reg_24;
        when "11001" => IP2Bus_Data <= reg_25;
        when "11010" => IP2Bus_Data <= reg_26;
        when "11011" => IP2Bus_Data <= reg_27;
        when "11100" => IP2Bus_Data <= reg_28;
        when "11101" => IP2Bus_Data <= reg_29;
        when "11110" => IP2Bus_Data <= reg_30;
        when "11111" => IP2Bus_Data <= reg_31;
        when others => IP2Bus_Data <= (others => '0');
    end case;
  end process OUT_MUX;
-----------------------------------------------------------------------------
-- Component Instantiation
-----------------------------------------------------------------------------
FILTER_0 : filter
  port map(
        clk             => Bus2IP_Clk,
        rst             => Bus2IP_Reset,
        new_data        => reg_16(16 to 31),
        valid           => reg_17(16 to 31),
        coef_0          => reg_0(16 to 31),
        coef_1          => reg_1(16 to 31),
        coef_2          => reg_2(16 to 31),
        coef_3          => reg_3(16 to 31),
        coef_4          => reg_4(16 to 31),
        coef_5          => reg_5(16 to 31),
        coef_6          => reg_6(16 to 31),
        coef_7          => reg_7(16 to 31),
        coef_8          => reg_8(16 to 31),
        coef_9          => reg_9(16 to 31),
        coef_10         => reg_10(16 to 31),
```

```vhdl
        coef_11        => reg_11(16 to 31),
        coef_12        => reg_12(16 to 31),
        coef_13        => reg_13(16 to 31),
        coef_14        => reg_14(16 to 31),
        coef_15        => reg_15(16 to 31),
        result   => reg_18   );
end architecture IMP;

entity filter is
   Port ( clk : in std_logic;
        rst     : in std_logic;
        new_data : in std_logic_vector(15 downto 0);
        valid  : in std_logic_vector(15 downto 0);
      coef_0 : in std_logic_vector(15 downto 0);
      coef_1 : in std_logic_vector(15 downto 0);
      coef_2 : in std_logic_vector(15 downto 0);
      coef_3 : in std_logic_vector(15 downto 0);
      coef_4 : in std_logic_vector(15 downto 0);
      coef_5 : in std_logic_vector(15 downto 0);
      coef_6 : in std_logic_vector(15 downto 0);
      coef_7 : in std_logic_vector(15 downto 0);
      coef_8 : in std_logic_vector(15 downto 0);
      coef_9 : in std_logic_vector(15 downto 0);
      coef_10 : in std_logic_vector(15 downto 0);
      coef_11 : in std_logic_vector(15 downto 0);
      coef_12 : in std_logic_vector(15 downto 0);
      coef_13 : in std_logic_vector(15 downto 0);
      coef_14 : in std_logic_vector(15 downto 0);
      coef_15 : in std_logic_vector(15 downto 0);
      result : out std_logic_vector(31 downto 0)  );
end filter;

architecture Behavioral of filter is
constant NUM_TAPS : integer := 15;          -- one less
constant scale  : integer := 256;
type data_reg0 is array (0 to NUM_TAPS) of std_logic_vector(15 downto 0);
type data_reg1 is array (0 to NUM_TAPS) of std_logic_vector(31 downto 0);
signal coeffs : data_reg0;        -- array for holding coefficients
signal inputs : data_reg0;  -- shift register for inputs
signal tmp    : data_reg1;
signal val_0 : std_logic_vector(15 downto 0);
signal val_1 : std_logic_vector(15 downto 0);
signal add_0, add_1 : std_logic_vector(31 downto 0);
begin
 process(clk, rst, new_data, valid) is
  variable tmp0 : std_logic_vector(31 downto 0);
```

```vhdl
begin
  if (rst = '1') then
    for i in 0 to NUM_TAPS loop
          tmp(i)    <= (others => '0');
          inputs(i) <= (others => '0');
    end loop;
          add_0 <= (others => '0');
          add_1 <= (others => '0');
  elsif clk'event and clk = '1' then
          val_0 <= valid;  -- shift down to
          val_1 <= val_0;        -- catch event
          if (val_0 /= val_1) then -- new data
            -- shift in new input
                for i in NUM_TAPS downto 1 loop
                        inputs(i) <= inputs(i-1);
                end loop;
                inputs(0) <= new_data;
                -- multiply inputs and coefficients
                for i in 0 to NUM_TAPS loop
                  --tmp(i) <= inputs(i) * coeffs(i);
                  tmp0 := inputs(i) * coeffs(i);
                  tmp(i) <= tmp0(31)&tmp0(31)&tmp0(31)&tmp0(31)&tmp0(31)
                            &tmp0(31)&tmp0(31)&tmp0(31)&tmp0(31 downto 8);
                end loop;
            -- pipeline 1 stage
                add_0  <= tmp(0)+tmp(1)+tmp(2)+tmp(3)+tmp(4)
                                +tmp(5)+tmp(6)+tmp(7);
                add_1  <= tmp(8)+tmp(9)+tmp(10)+tmp(11)+tmp(12)
                                +tmp(13)+tmp(14)+tmp(15);
                result <= add_0 + add_1;
    end if;
  end if;
end process;

-- put coefficients into array
coeffs(0) <= coef_0;
coeffs(1) <= coef_1;
coeffs(2) <= coef_2;
coeffs(3) <= coef_3;
coeffs(4) <= coef_4;
coeffs(5) <= coef_5;
coeffs(6) <= coef_6;
coeffs(7) <= coef_7;
coeffs(8) <= coef_8;
coeffs(9) <= coef_9;
coeffs(10) <= coef_10;
```

```
coeffs(11) <= coef_11;
coeffs(12) <= coef_12;
coeffs(13) <= coef_13;
coeffs(14) <= coef_14;
coeffs(15) <= coef_15;

end Behavioral;
```

Transposed Form FIR Filter Core

```
entity filter is
    Port ( clk : in std_logic;
          rst     : in std_logic;
          new_data : in std_logic_vector(15 downto 0);
          valid  : in std_logic_vector(15 downto 0);
          coef_0 : in std_logic_vector(15 downto 0);
          coef_1 : in std_logic_vector(15 downto 0);
          coef_2 : in std_logic_vector(15 downto 0);
          coef_3 : in std_logic_vector(15 downto 0);
          coef_4 : in std_logic_vector(15 downto 0);
          coef_5 : in std_logic_vector(15 downto 0);
          coef_6 : in std_logic_vector(15 downto 0);
          coef_7 : in std_logic_vector(15 downto 0);
          coef_8 : in std_logic_vector(15 downto 0);
          coef_9 : in std_logic_vector(15 downto 0);
          coef_10 : in std_logic_vector(15 downto 0);
          coef_11 : in std_logic_vector(15 downto 0);
          coef_12 : in std_logic_vector(15 downto 0);
          coef_13 : in std_logic_vector(15 downto 0);
          coef_14 : in std_logic_vector(15 downto 0);
          coef_15 : in std_logic_vector(15 downto 0);
          result : out std_logic_vector(31 downto 0)      );
end filter;

architecture Behavioral of filter is
constant NUM_TAPS : integer := 15;            -- one less
type data_reg0 is array (0 to NUM_TAPS) of std_logic_vector(15 downto 0);
type data_reg1 is array (0 to NUM_TAPS) of std_logic_vector(31 downto 0);
signal coeffs : data_reg0;        -- array for holding coefficients
signal tmp    : data_reg1;
signal atmp   : data_reg1;
signal val_0 : std_logic_vector(15 downto 0);
signal val_1 : std_logic_vector(15 downto 0);
begin
process(clk, rst, new_data, valid) is
begin
  if (rst = '1') then
```

```vhdl
    for i in 0 to NUM_TAPS loop
          tmp(i)  <= (others => '0');
          atmp(i) <= (others => '0');
    end loop;
  elsif clk'event and clk = '1' then
    val_0 <= valid;  -- shift down to
    val_1 <= val_0;     -- catch event
      if (val_0 /= val_1) then -- new data
          -- compute transposed FIR
          --  * multiplication with coefficients
          for i in 0 to NUM_TAPS loop
              tmp(i) <= new_data * coeffs(i);
          end loop;
          --  * additions and shifts
          for i in NUM_TAPS downto 1 loop
             atmp(i) <= atmp(i-1) + tmp(i);
          end loop;
          atmp(0) <= tmp(0);
          --  * assign outputs
          result <= atmp(NUM_TAPS);
      end if;
  end if;
end process;

-- put transposed coefficients into array
coeffs(0) <= coef_15;
coeffs(1) <= coef_14;
coeffs(2) <= coef_13;
coeffs(3) <= coef_12;
coeffs(4) <= coef_11;
coeffs(5) <= coef_10;
coeffs(6) <= coef_9;
coeffs(7) <= coef_8;
coeffs(8) <= coef_7;
coeffs(9) <= coef_6;
coeffs(10) <= coef_5;
coeffs(11) <= coef_4;
coeffs(12) <= coef_3;
coeffs(13) <= coef_2;
coeffs(14) <= coef_1;
coeffs(15) <= coef_0;

end Behavioral;
```

APPENDIX C

C Code

Software LMS Training Algorithm

```c
// Adaptive filter core base address
#define FILT_ADDR XPAR_FIR_FILTER_PLB_CORE_0_BASEADDR

static void filter()    // filter data from mem
{
        int i, j, k, m;
        int32u wr_addr;
        unsigned start_addr = 0x80000000;
        Xint16 mu    = 0xC;
        Xint16 scale = 0x100;   // 256;
        Xuint16 offset[16] = {0x0,0x4,0x8,0xC,0x10,0x14,0x18,0x1C,
                              0x20,0x24,0x28,0x2C,0x30,0x34,0x38,0x3C};
        Xint16 weight[16] = {-3,-4,0,2,-3,6,1,-4,2,-1,-3,5,2,4,0,-1};
        Xint16 u[19]       = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
        Xint16 desired[3]  = {0,0,0};
        Xint16 input   = 0x0000;
        Xint16 desir   = 0x0000;
        Xint16 valid   = 0x0000;
        Xint32 data    = 0x00000000;
        Xint16 output  = 0x0000;
        Xint16 error   = 0x0000;

        // Assign random coefficients
        print("Assigning Coefficients...\r\n");
        for (k=0; k<16; k++){
                XIo_Out32( FILT_ADDR + offset[k], weight[k]);
        }

        // filter data from memrory
        for (i=0; i < 400; i++){
                // get data from memory
                MEM_ADDR = start_addr + 4*i;
                data = MEM_REG;
```

```
// split into input & desired
input = data & 0xFFFF;
desir = (data & 0xFFFF0000) >> 16;
// store inputs  -- need to store 3 extra for pipeline delay
for (j=1; j<19; j++) {
        u[19-j] = u[19-j-1];
}
u[0] = input;

//send input to filter core
XIo_Out32( FILT_ADDR + 0x40, input);
// tell core data is valid
valid = valid + 1;
XIo_Out32( FILT_ADDR + 0x44, valid);

// check output
output = XIo_In32((FILT_ADDR + 0x48));
error  = desired[2] - output;

// store result in sdram
wr_addr = 0x81000000 + 4*i;
mem_write(wr_addr, output);

// store desired for 4 cycles
for (j=1; j<3; j++) {
        desired[3-j] = desired[3-j-1];
}
desired[0] = desir;

//------------------------------------------------------
// LMS Algorithm
if( i % 3 == 0 ){
        for (m=0; m<16; m++){
                // update weight
                weight[m] = weight[m] + (u[m+3]*error)/(mu*scale);
                // send new weight to core
                XIo_Out32( FILT_ADDR + offset[m], weight[m]);
        }
}
//------------------------------------------------------
        }

}  // end filter fcn
```

Software RLS Training Algorithm

```
// Adaptive filter core base address
        int i,j,t, tmp0;
        Xint16 tmp1[16][16],

        //--- Filter Parameters ---
        int     scale = 300;
        Xint16  lamda = 297;
        Xint16 delta    = 300;

        //intepoch = 0;
        //int    e_max = 600;        //% maximum # of epochs
        Xint16            input;
        Xint16            output;
        Xint16            E;

        //--- Initialize ---
        Xint16 w[16];
        Xint16 d[16];
        Xint16 u[16];
        Xint16 y[16];

        Xint16 P[16][16];
        Xint16 PIv[16];
        Xint16 k[16];

        for (i=0; i<16; i++){
                w[i]    = 0;
                d[i]    = 0;
                u[i]    = 0;
                y[i]    = 0;
                PIv[i]  = 0;
        }
        //initialize P
        for( i=0; i<16; i++){
                for( j=0; j<16; j++ ){
                        if (i == j)
                                P[i][j] = scale*delta;
                        else
                                P[i][j] = 0;
                }
        }
```

```
// filter data from memrory
for (i=0; i < 400; i++){
        // get data from memory
        MEM_ADDR = start_addr + 4*i;
        data = MEM_REG;

        // split into input & desired
        input = data & 0xFFFF;
        desir = (data & 0xFFFF0000) >> 16;
        // store inputs  -- need to store 3 extra for pipeline delay
        for (j=1; j<19; j++) {
                u[19-j] = u[19-j-1];
        }
        u[0] = input;

        //send input to filter core
        XIo_Out32( FILT_ADDR + 0x40, input);
        // tell core data is valid
        valid = valid + 1;
        XIo_Out32( FILT_ADDR + 0x44, valid);

        // check output
        output = XIo_In32((FILT_ADDR + 0x48));
        error  = desired[2] - output;

        // store result in sdram
        wr_addr = 0x81000000 + 4*i;
        mem_write(wr_addr, output);

        //-------------------------- RLS Algorithm ------------------------
        // Compute PIv
        for( i=0; i<16; i++){
                for( j=0; j<16; j++){
                        PIv[i] = PIv[i] + (u[j]*P[i][j])*scale;
                }
        }

        // Compute k
        tmp0 = 0;
        for( i=0; i<16; i++){
                tmp0 = tmp0 + u[i]*PIv[i];
        }
        tmp0 = lamda + tmp0;

        for( i=0; i<16; i++){
                k[i] = scale*PIv[i]/tmp0;
```

```
        }

        // Compute Error
        E = d[0] - output;

        // Update weights
        for (m=0; m<16; m++){
                // update weight
                w[m] = w[m] + (E*k[i])/scale;
                // send new weight to core
                XIo_Out32( FILT_ADDR + offset[m], w[m]);
        }

        // Update P
        //  compute k*u'*P
        for( i=0; i<16; i++){
                for( j=0; j<16; j++){
                        tmp1[i][j] = (k[i]*u[j]*P[i][j])/(scale*scale*lamda);
                }
        }
        for( i=0; i<16; i++){
                for( j=0; j<16; j++){
                        P[i][j] = ((P[I][j]*scale)/lamda) – tmp1[i][j];
                }
        }
        //-----------------------------------------------------------------------
}
```

APPENDIX D

Device Synthesis Results

Selected Device: 2vp20ff896-5

Filter Tap Synthesis Results

LUT-Based Multiplier

Number of Slices:	86	out of	9280	0%
Number of Slice Flip Flops:	36	out of	18560	0%
Number of 4 input LUTs:	147	out of	18560	0%

Minimum period: 10.744ns (Maximum Frequency: 93.075MHz)

Serial-Parallel Multiplier

Number of Slices:	79	out of	9280	0%
Number of Slice Flip Flops:	91	out of	18560	0%
Number of 4 input LUTs:	146	out of	18560	0%

Minimum period: 5.091ns (Maximum Frequency: 196.425MHz)
Pipeline depth = 15

Partial Product Multiplier with LUTs

Number of Slices:	461	out of	9280	4%
Number of Slice Flip Flops:	587	out of	18560	3%
Number of 4 input LUTs:	376	out of	18560	2%

Minimum period: 5.053ns (Maximum Frequency: 197.902MHz)
Pipeline depth = 4

Partial Product Multiplier with BRAM

Number of Slices:	53	out of 9280	0%
Number of Slice Flip Flops:	77	out of 18560	0%
Number of 4 input LUTs:	76	out of 18560	0%
Number of BRAMs:	2	out of 88	2%

Minimum period: 4.588ns (Maximum Frequency: 217.960MHz)
Pipeline depth = 4

Virtex-II Embedded Multiplier

Number of Slices:	21	out of 9280	0%
Number of Slice Flip Flops:	36	out of 18560	0%
Number of MULT18X18s:	1	out of 88	1%

Minimum period: 5.556ns (Maximum Frequency: 179.986MHz)

Adaptive Filter Synthesis Results

Software Trained Filter

Number of External IOBs	85 out of 556	15%
Number of LOCed External IOBs	85 out of 85	100%

Number of MULT18X18s	16 out of 88	18%
Number of PPC405s	2 out of 2	100%
Number of RAMB16s	24 out of 88	27%
Number of SLICEs	3531 out of 9280	38%

Number of BUFGMUXs	2 out of 16	12%
Number of DCMs	1 out of 8	12%
Number of JTAGPPCs	1 out of 1	100%
Number of TBUFs	128 out of 4640	2%

Hardware Only Filter

Number of Slices:	556	out of 9280	5%
Number of Slice Flip Flops:	843	out of 18560	4%

Number of 4 input LUTs: 474 out of 18560   2%
Number of bonded IOBs: 357 out of   556   64%
Number of MULT18X18s: 24 out of   88   27%

REFERENCES

[1] K.A. Vinger, J. Torresen, "Implementing evolution of FIR-filters efficiently in an FPGA." *Proceeding,. NASA/DoD Conference on Evolvable Hardware*, 9-11 July 2003. Pages: 26 – 29

[2] E. C. Ifeachor and B. W. Jervis, *Digital Signal Procesing, A Practical Approach*, Prentice Hall, 2002.

[3] S.J. Visser, A.S. Dawood, J.A. Williams, "FPGA based real-time adaptive filtering for space applications," *Proceedings, IEEE International Conference on Field-Programmable Technology*, 16-18 Dec. 2002. Pages: 322–326.

[4] "Hardware Description Language." *Wikepedia: The Free Encyclopedia*. 18 May 2004. http://en.wikipedia.org/wiki/Hardware_description_language.

[5] Xilinx Inc., "Virtex-II Pro™ Platform FPGAs: Functional Description," DS083-2 (v3.0), December 10, 2003.

[6] Project Veripage, retrieved from: *http://www.angelfire.com/ca/verilog/history.html*.

[7] Sudhakar Yalamanchili, *Introductory VHDL, From Simulation to Synthesis*, Prentice Hall, 2001.

[8] S. W. Wales, "Co-channel interference suppression for TDMA mobile radio systems," *IEEE Colloquium on Mobile Communications Towards the Year 2000*, Oct 17th 1994. Pages: 9/1 - 9/6

[9] S. Haykin, *Adaptive Filter Theory*, Prentice Hall, Upper Saddle River, NJ, 2002.

[10] Naresh R. Shanbhag, Manish Goel, "Low-Power Adaptive Filter Architecture and Their application to 51.84 Mb/s ATM-LAN", *IEEE Transactions on Signal Processing*, Vol. 45, NO. 5, May 1997. 70
http://citeseer.nj.nec.com/shanbhag97lowpower.html

[11] J. Liang, R. Tessier, O. Mencer, "Floating point unit generation and evaluation for FPGAs," *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003. FCCM 2003. 11th, 9-11 April 2003 Pages:185 – 194

[12] U. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays,* Springer, 2001.

[13] K. Wiatr, "Implementation of multipliers in FPGA structures," *2001 International Symposium on Quality Electronic Design*, 26-28 March 2001 Pages: 415 – 420

[14] M. Karlsson, M. Vesterbacka, L. Wanhammar, "Design and implementation of a complex multiplier using distributed arithmetic,**"** *Workshop on Signal Processing Systems, 1997. SIPS 97 - Design and Implementation*, 1997 IEEE, 3-5 Nov. 1997 Pages: 222 – 231

[15] S. Vassiliadis, E.M. Schwarz, B.M. Sung, *"*Hard-wired multipliers with encoded partial products" *IEEE Transactions on Computers*, Volume: 40, Issue: 11, Nov. 1991 Pages: 1181 – 1197

[16] IBM Microelectronics Division, "The PowerPC 405™ Core," White Paper, November 11, 1998.

[17] Xilinx Inc., "*Block Adaptive Filter*," Application Note XAPP 055, Xilinx, San Jose, CA, January 9, 1997.

[18] D.L. Jones, "Learning characteristics of transpose-form LMS adaptive filters," IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, [see also Circuits and Systems II: Express Briefs, IEEE Transactions on], Volume: 39 , Issue: 10 , Oct. 1992 Pages:745 – 749

[19] C. Papaodysseus, D. Gorgoyannis, E. Koukoutsis, P. Roussopoulos, "A very robust, fast, parallelizable adaptive least squares algorithm with excellent tracking abilities," *IEEE International Conference on Acoustics, Speech, and Signal Processing,* 1994. ICASSP-94., 1994, Volume: iii , 19-22 April 1994 Pages:III/385 - III/388 vol.3

[20] Xilinx, Inc., *The Programmable Logic Data Book 2000*, Xilinx, Inc, CA: 2002.

[21] R.J. Andraka and A. Berkun, "FPGAs Make Radar Signal Processor on a Chip a Reality," *Proceedings, 33rd Asilomar Conference on Signals, Systems and Computers*, October 24-27, 1999, Monterey, CA

[22] T. Rissa, R. Uusikartano, J. Niittylahti, "Adaptive FIR filter architectures for run-time reconfigurable FPGAs," *Proceedings, IEEE International Conference on Field-Programmable Technology*, 16-18 Dec. 2002. Pages: 52-59.

[23] G. Welch, G. Bishop, "An introduction to the Kalman Filter," Course Notes, *ACM SIGGRAPH 2001*, August 12-17 2001.

BIOGRAPHICAL SKETCH

Joe Petrone was born on August 6, 1980 in West Palm Beach, Florida. He completed his high school education in Cranston, Rhode Island. He received dual bachelor degrees in Electrical and Computer Engineering from the Department of Electrical and Computer Engineering at the Florida State University in December 2002.

He was admitted into the Masters program in the Department of Electrical and Computer Engineering at the Florida State University in the spring of 2003. His research specializes in the fields of Digital Signal Processing and ASIC Design.