1-1-2007

DNLMS-based adaptive filters for echo cancellation.

Raymond Lee
University of Windsor

DNLMS-based Adaptive Filters for Echo Cancellation

by

Raymond Lee

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through Electrical and Computer Engineering
in Partial Fulfillment of the Requirements for
the Degree of Master of Applied Science at the
University of Windsor

Windsor, Ontario, Canada
2007

Your file Votre référence
ISBN: 978-0-494-42314-1
Our file Notre référence
ISBN: 978-0-494-42314-1

Abstract

Power consumption is a main issue with echo canceller implementation due to the use of high-order adaptive filters. The normalized least-mean-square (NLMS) algorithm is an adaptive filtering algorithm typically used in echo cancellation, but does not permit pipelining, a technique that facilitates low-power filter architectures. This limitation can be overcome by using the delayed NLMS (DNLMS) algorithm. There are two objectives of this thesis related to the implementing a DNLMS adaptive filter for echo cancellation.

The first objective is to apply computationally-efficient techniques to the DNLMS algorithm to reduce power and/or area consumption. The considered techniques either simplify computationally-intensive operations or schedule less filter coefficient updates.

The second objective is to present the hardware implementation of a pipelined DNLMS adaptive filter. The implemented design is pipelined, modular, low-latency, and portable. The design methodology consists of fixed-point and RTL simulations, physical synthesis for the Altera Stratix FPGA, and hardware verification.

To my family for their everlasting support.

Acknowledgments

I would like express my sincere gratitude for the encouragement and guidance given to me by my advisors, Dr. Esam Abdel-Raheem and Dr. Mohammed A. S. Khalid. I feel very fortunate to have had two supportive and patient advisors. It has been a pleasure working with both of you. I would also like to express my gratitude to the other members of my committee, Dr. Kemal E. Tepe and Dr. Alioune Ngom, for their kindness, flexibility, and helpful feedback. Also, I would like to thank Academy Publisher for allowing me to include material from the Journal of Communications in my thesis.

Of course, I have to thank my fellow graduate students of the ECE department. Their support and advise was critical to my development as graduate student. Their company and great sense of humour was vital to me keeping my sanity. Who knew that a group of such big, gigantic nerds could be so much fun? There are a lot of people I will need to thank, but special recognition goes to Kevin Banović and Harb Abdulhamid. Working along side them in 268 EH wasn't just fun, it was a privilege.

Finally, I would like to dedicate this thesis to my family. I have received a lot of support from different sources, but their daily support in my life is the one that I am most thankful for.

Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Definition
AEC	Acoustic echo cancellation.
ALC	Adaptive linear combiner.
CAD	Computer-aided design.
CMOS	Complementary metal-oxide semiconductor.
CSS	Composite source signal.
DLMS	Delayed least-mean-square.
DNLMS	Delayed normalized least-mean-square.
DSP	Digital signal processor or digital signal processing.
DTD	Double-talk detector.
DUT	Design under test.
ERL	Echo return loss.
ERLE	Echo return loss enhancement.
FIR	Finite-duration impulse response.
FPGA	Field programmable gate array.
FSM	Finite-state machine.
HDL	Hardware description language.
i.i.d.	Independent and identically distributed.
I/O	Input/output.
ITU	International Telecommunications Union.
LAB	Logic array block.
LE	Logic element.
LEM	Loudspeaker-enclosure-microphone.
LMS	Least-mean-square.
LUT	Look-up-table.
MSE	Mean-squared error.

Abbreviation	Definition
NEC	Network echo cancellation.
NLMS	Normalized least-mean-square.
PE	Processing element.
POT	Power-of-two.
PSTN	Public switched telephone network.
QE	Quantized-Error.
QER	Quantized-Error-Regressor-energy.
QR	Quantized-Regressor-energy.
RAM	Random access memory.
ROM	Read-only memory.
RTL	Register transfer level.
SAG	Stop-and-go.
SNR	Signal-to-noise ratio.
VoIP	Voice over Internet protocol.
WGN	White Gaussian noise.

List of Symbols

Symbol	Definition
a	Number of integer bits, excluding the sign bit.
b	Number of fractional bits.
d	Desired response input.
e	Error signal.
n	Time index.
$\mathbf{p}$	Cross correlation vector.
s	Near-end signal.
x	Input signal.
$\mathbf{x}$	Input signal vector or regressor.
w	Weight.
$\mathbf{w}$	Weight or coefficient vector.
y	Output signal.
$\|\mathbf{x}(n)\|$	Regressor energy.
D	Adaptation delay.
M	Parameter of M-Max algorithm; number of weights to be updated.
N	Filter order.
P	Number of weights per PE.
$\mathbf{R}$	Input correlation matrix.
α	Convergence parameter of NLMS algorithm.
β	Parameter of NLMS algorithm.
∇	Gradient vector of performance function.
κ	SAG-threshold.
λ_{max}	Largest eigenvalue of $\mathbf{R}$.
μ	Adaptation step-size.
τ	Smallest magnitude value allowed for POT quantization.

Symbol	Definition
ζ	Mean-square error performance function.
ζ_{min}	Minimum mean-square error.
$E[\cdot]$	Expectation operator.
$\lVert \cdot \rVert$	l_2 norm operation.
$sgn\{\cdot\}$	Sign function.
$\mathrm{tr}[\cdot]$	Trace of a matrix.
$(\cdot)^T$	Transposition.

Chapter 1

Introduction

1.1 Background on Echo Cancellation

Echoes are delayed or distorted versions of a sound or signal that have been reflected back to the source [21]. For small round trip delays, echoes are interpreted as reverberations. In conversations, people usually prefer the presence of reverberations over an anechoic environment. However, when round trip delays are longer than a few tens of milliseconds, echoes become distinct and disruptive [14]. The two types of echoes considered in telecommunications are network echoes and acoustic echoes.

Network echoes appear in telephone calls over the public switched telephone network (PSTN). The link connecting two users is comprised of a two-wire line that connects each phone to its respective local central office and two separate unidirectional lines that make a four-wire inter-office link, as shown in Fig. 1.1. The hybrid transformer is the device that connects the two-wire circuit to the four-wire circuit. Ideally, the hybrid would transfer all energy from the incoming signal on the four-wire

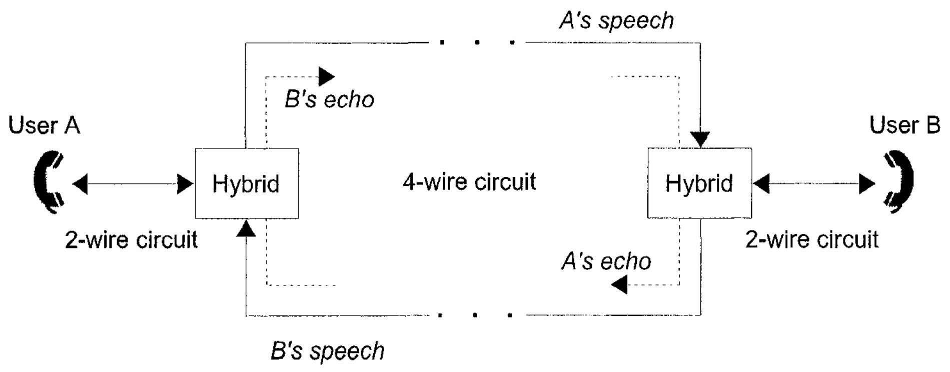


Figure 1.1: Network echoes over the PSTN.

circuit to the two-wire circuit. However, due to imperfect impedance matching, some of the energy is reflected back to its source on the four-wire circuit as an echo [31]. Thus, network echoes arise from hybrid devices. For roundtrip delays less than 100 ms duration, echo suppressors were used to suppress network echoes [33]. However, when the round-trip delay exceeded 100 ms, as was the case with the incorporation of satellite links, network echo cancellation (NEC) was required.

The echo canceller was first introduced in the 1960's by Sondhi [30] and concurrently by Becker and Rudin [8]. The basic principle of echo cancellation is to eliminate the echo from the transmission signal by subtracting a synthesized replica. In order to create the synthetic echo, the unknown time-varying echo path impulse response is modelled using an adaptive filter. Typically, an adaptive filter on the order of hundreds is needed for NEC [21].

Figure 1.2 shows the system model of echo cancellation. When excited by the received signal, the adaptive filter outputs a synthetic echo. By subtracting the synthetic echo, the genuine echo is effectively removed prior to transmission. Usually during adaptation, the near-end signal is assumed to be simply noise. This is a reasonable assumption because a double-talk detector (DTD) is usually implemented to pause the adaptive filter's adaptation, in order to avoid divergence, when both

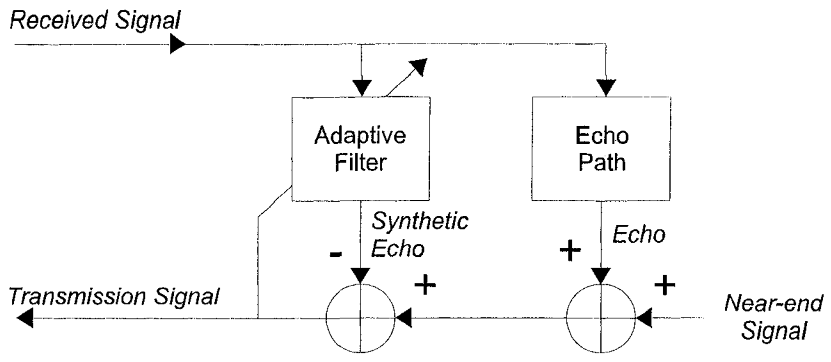


Figure 1.2: Echo cancellation system model.

received and near-end signals are present, i.e. during double talk [14].

Recently, NEC has received renewed attention with the introduction of voice over Internet protocol (VoIP) [19, 29]. In this application, the packet-switched network is connected to the PSTN through a voice gateway. Network echoes are a problem for VoIP because, irrespective of whether the call is local or long distance, the round trip delay is always large as a result of the inherent large delay in the packet-switched network which includes encoding, jitter buffer, and network propagation delays.

An adaptive filter can be similarly applied to eliminate acoustic echoes in acoustic echo cancellation (AEC). Acoustic echoes arise in applications such as teleconferencing and hands-free telephony, where there is a loudspeaker-enclosure-microphone (LEM) system. An electro-acoustic coupling between the loudspeaker and the microphone results in the microphone picking up signals from the loudspeaker as well as signal reflections off surrounding objects and boundaries [9], as illustrated in Fig. 1.3. AEC presents a more challenging problem compared to NEC because acoustic echoes are generally longer, requiring an adaptive filter on the order of thousands [14].

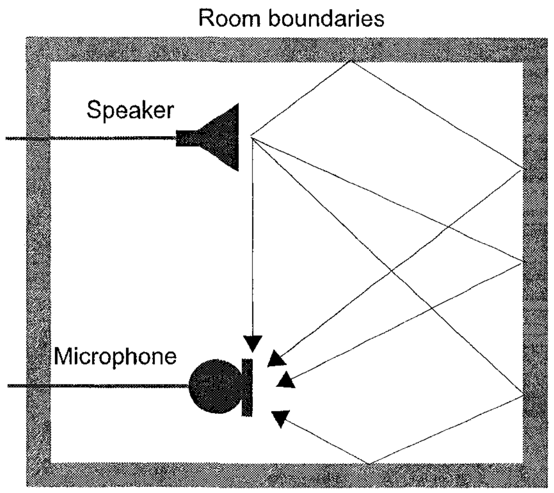


Figure 1.3: Acoustic echoes.

1.2 Thesis Objectives

The normalized least-mean-square (NLMS) algorithm is a commonly used adaptive filtering algorithm for echo cancellation [14]. However, a variant of the NLMS algorithm, the delayed normalized least-mean-square (DNLMS) algorithm [3], is of interest because it allows pipelining, a useful filter design technique suitable for low-power or high-speed applications [22]. With the DNLMS algorithm as the algorithm of choice, this work addresses two issues related to the implementation of an adaptive filter for echo cancellation.

The first issue is the requirement of high-order adaptive filters for echo cancellation. Such large filters have high computational requirements, i.e. a large number of multiplications, divisions, and additions/subtractions occur within one clock period. This translates to large resource demands and high power consumption, thus making implementation challenging. The second issue is that, to this date, there has been little work in presenting architectures for DNLMS adaptive filters [23, 26, 27], none of which present implementation details.

Thus, this work has two objectives.

1. Reducing the computational demand of the DNLMS algorithm through the application of computationally-efficient techniques.

2. Presenting the field programmable gate array (FPGA) implementation of a pipelined DNLMS adaptive filter.

1.3 Thesis Organization

The remainder of the thesis is organized as follows: Chapter 2 provides a review of adaptive filter theory, ending with the introduction of the DNLMS algorithm. Chapter 3 is associated with the first objective of this work: reduction of the amount of computations required by the DNLMS algorithm through application of computationally-efficient techniques. Analysis and simulation results are provided for the modified DNLMS algorithm. Chapter 4 is associated with the second objective of this work: providing details of an FPGA implementation of a pipelined DNLMS adaptive filter. The applied design methodology begins with architecture derivation and ends with hardware verification. Finally, conclusions and future work are described in Chapter 5.

Chapter 2

Review of Adaptive Filtering

2.1 Fundamentals of Adaptive Filtering

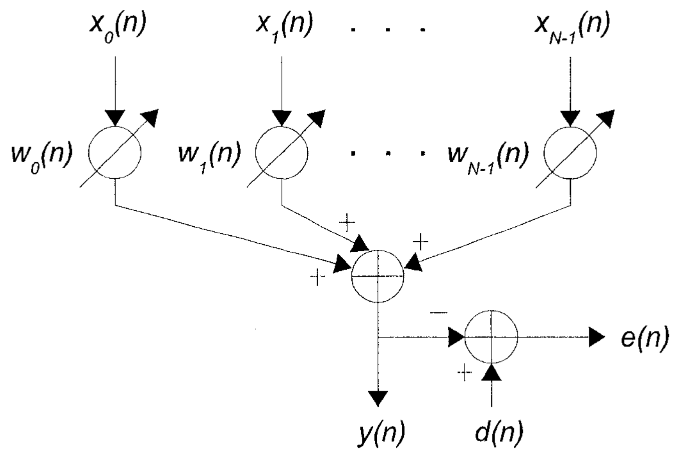
The adaptive linear combiner (ALC), shown in Fig. 2.1(a), is the fundamental building block in most adaptive systems [33]. The output, $y(n)$, is a linear combination of several inputs at time index n and is given by

$$y(n) = \mathbf{x}^T(n)\mathbf{w}(n) = \mathbf{w}^T(n)\mathbf{x}(n) \tag{2.1}$$

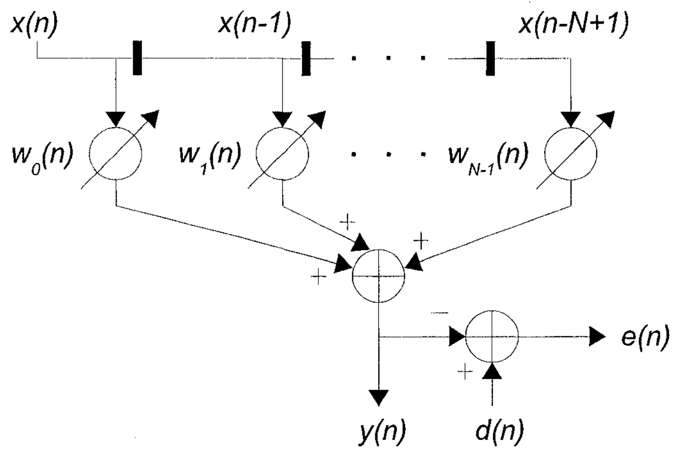
where $\mathbf{x}(n) = [x_0(n)\ x_1(n)\ \ldots\ x_{N-1}(n)]^T$ is the input signal vector comprised of sampled data from N different sources and $\mathbf{w}(n) = [w_0(n)\ w_1(n)\ \ldots\ w_{N-1}(n)]^T$ is the weight vector (also referred to as the coefficient vector). The output is compared to the desired response input, $d(n)$, to produce the error signal, $e(n)$. The error is given by

$$e(n) = d(n) - y(n). \tag{2.2}$$

When $\mathbf{x}(n)$ consists of the N sequential samples of the same signal, i.e. $\mathbf{x}(n) = [x(n)\ x(n-1)\ \ldots\ x(n-N+1)]^T$, the ALC becomes the adaptive finite-duration



(a) General form.



(b) As an FIR filter.

Figure 2.1: Adaptive linear combiner.

impulse response (FIR) filter shown in Fig. 2.1(b). In adaptive filtering, $\mathbf{x}(n)$ is typically referred to as the regressor.

In order for the output to progressively approximate the desired response, the weights are adjusted in a manner that minimizes a cost function. The mean-square error (MSE) is a commonly used cost function given by

$$\begin{aligned} \text{MSE} &\triangleq \zeta \\ &= E[e^2(n)] \\ &= E[(d(n) - y(n))^2] \\ &= E[(d(n) - \mathbf{w}^T(n)\mathbf{x}(n))^2] \\ &= E[d^2(n)] - 2E[d(n)\mathbf{w}^T(n)\mathbf{x}(n)] + E[\mathbf{w}^T(n)\mathbf{x}(n)\mathbf{x}^T(n)\mathbf{w}(n)]. \end{aligned} \tag{2.3}$$

Assuming that the weights are fixed, the MSE cost function is given by

$$\zeta = E[d^2(n)] - 2\mathbf{w}^T E[d(n)\mathbf{x}(n)] + \mathbf{w}^T E[\mathbf{x}(n)\mathbf{x}^T(n)]\mathbf{w}. \tag{2.4}$$

Let $\mathbf{R}$ be defined as the input correlation matrix given by

$$\begin{aligned} \mathbf{R} &= E[\mathbf{x}(n)\mathbf{x}^T(n)] \\ &= E\begin{bmatrix} x_0^2(n) & x_0(n)x_1(n) & \dots & x_0(n)x_{N-1}(n) \\ x_1(n)x_0(n) & x_1^2(n) & \dots & x_1(n)x_{N-1}(n) \\ \vdots & \vdots & \ddots & \vdots \\ x_{N-1}(n)x_0(n) & x_{N-1}(n)x_1(n) & \dots & x_{N-1}^2(n) \end{bmatrix} \end{aligned} \tag{2.5}$$

and let $\mathbf{p}$ be defined as the cross correlation vector given by

$$\mathbf{p} = E[d(n)\mathbf{x}(n)] = E[d(n)x_0(n)\ d(n)x_1(n)\ \dots\ d(n)x_{N-1}(n)]^T. \tag{2.6}$$

By substituting $\mathbf{R}$ and $\mathbf{p}$ into (2.4), the MSE cost function can be simplified to

$$\zeta = E[d^2(n)] - 2\mathbf{w}^T\mathbf{p} + \mathbf{w}^T\mathbf{R}\mathbf{w}. \tag{2.7}$$

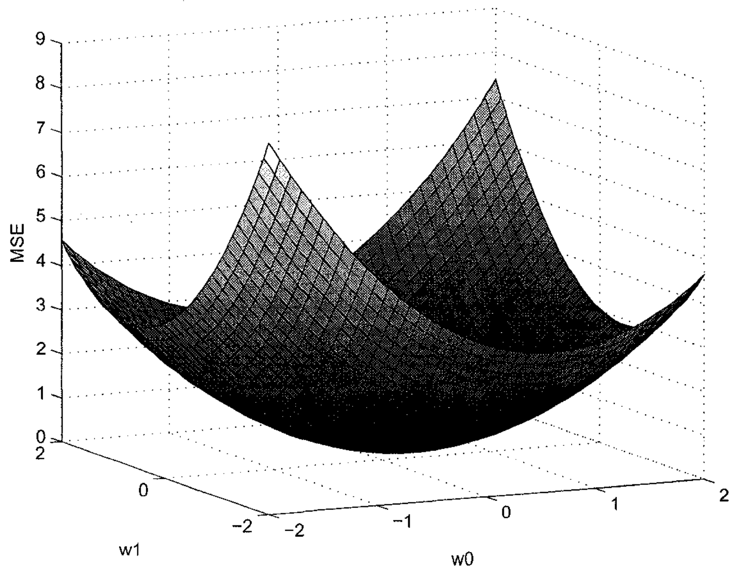


Figure 2.2: Mean-square error surface.

It can be seen from (2.7) that the MSE cost function is a quadratic function of the weights forming a hyperparaboloid surface. Figure 2.2 illustrates the MSE surface as a paraboloid for the case of the weight vector consisting of two weights. The bowl-shaped surface is concave upwards and has only positive values. The bottom of the surface represents the minimum mean-square error, ζ_{min}, which projects to optimal weight vector, $\mathbf{w}^*$. The values of ζ_{min} and $\mathbf{w}^*$ can be found through the gradient of the MSE cost function.

The gradient of the MSE cost function with respect to the weight vector is given by

$$\begin{aligned} \nabla \triangleq \frac{\partial \zeta}{\partial \mathbf{w}} &= \left[\frac{\partial \zeta}{\partial w_0} \, \frac{\partial \zeta}{\partial w_1} \cdots \frac{\partial \zeta}{\partial w_{N-1}} \right]^T \\ &= -2\mathbf{p} + 2\mathbf{R}\mathbf{w}. \end{aligned} \tag{2.8}$$

Equating (2.8) to zero allows for the optimal weight vector to be solved as

$$\mathbf{w}^* = \mathbf{R}^{-1}\mathbf{p}. \tag{2.9}$$

Substitution of $\mathbf{w}^*$ into (2.7) allows for the minimum mean-square error to be solved as

$$\begin{aligned} \zeta_{min} &= E[d^2(n)] - 2\mathbf{w}^{*T}\mathbf{p} + \mathbf{w}^{*T}\mathbf{R}\mathbf{w}^* \\ &= E[d^2(n)] - \mathbf{w}^{*T}\mathbf{p}. \end{aligned} \tag{2.10}$$

The solution for the optimal weight vector is known as the Wiener solution.

2.2 The Method of Steepest-Descent

As discussed in the previous section, the weights of the adaptive filter are adjusted to minimize the MSE cost function. The method of steepest-descent is a well-known weight adaptation procedure that seeks the minimum of the MSE surface. It serves as the basis for several adaptive filtering algorithms [10]. The method iteratively adds to the each weight a term proportional to the instantaneous gradient in order to descend the MSE surface. As a result, the weight vector progressively converges to the Wiener solution or a near-optimal solution.

The weight update equation of the method of steepest-descent is given by

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \mu\nabla(n). \tag{2.11}$$

where μ is the adaptation step-size that controls the stability and convergence rate. The gradient vector measured at $\mathbf{w} = \mathbf{w}(n)$ is denoted as $\nabla(n)$.

Generally, when the step-size is chosen small, the MSE slowly converges to a steady-state value that will be close to the minimum value. On the other hand, when the step-size is chosen large, the MSE quickly converges to a steady-state value that will be larger than the minimum value. However, having a step-size that is too large will result in the weights diverging from the Wiener solution.

The calculation of the gradient vector requires statistical knowledge of the input and desired signals. In practice, these quantities are usually unknown. To get around this problem, there are methods to estimate the input correlation matrix $\mathbf{R}$ and correlation vector $\mathbf{p}$ [33]. However, several steepest-descent-based algorithms alternatively use an estimate of the gradient.

2.3 Least-Mean-Square Algorithm

The least-mean-square (LMS) algorithm is the most commonly used weight adaptation procedure in adaptive filtering [10, 33]. It is a steepest-descent-based algorithm that estimates the gradient of the MSE, shown in (2.8), with the gradient of the squared error given by

$$\begin{aligned}\hat{\nabla}(n) &= \frac{\partial e^2(n)}{\partial \mathbf{w}(n)} \\ &= 2e(n)\left[\frac{\partial e(n)}{\partial w_0(n)}\ \frac{\partial e(n)}{\partial w_1(n)}\ \cdots\ \frac{\partial e(n)}{\partial w_{N-1}(n)}\right]^T \\ &= -2e(n)\mathbf{x}. \end{aligned} \tag{2.12}$$

Substituting this estimate for the true gradient in (2.11) yields the weight update equation for the LMS algorithm, given by

$$\begin{aligned}\mathbf{w}(n+1) &= \mathbf{w}(n) - \mu\hat{\nabla}(n) \\ &= \mathbf{w}(n) - 2\mu e(n)\mathbf{x}. \end{aligned} \tag{2.13}$$

As seen in (2.13), the weight update equation of the LMS algorithm is very simple and straightforward; it does not require calculations of the input correlation matrix nor the correlation vector. Another advantage of the LMS algorithm is that it has guaranteed stable convergence when the step-size is chosen within the range

$$0 < \mu < \frac{1}{\lambda_{max}} \tag{2.14}$$

where λ_{max} is the largest eigenvalue of $\mathbf{R}$. Since λ_{max} cannot exceed the trace of $\mathbf{R}$, the selectable range of the step-size can also be expressed as

$$0 < \mu < \frac{1}{\mathrm{tr}[\mathbf{R}]}. \tag{2.15}$$

2.4 Normalized Least-Mean-Square Algorithm

Several algorithms have been derived from the standard LMS algorithm. One notable variant commonly used in echo cancellation is the normalized least-mean-square (NLMS) algorithm [12, 14]. Its weight update equation is given by

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu(n)e(n)\mathbf{x}(n). \tag{2.16}$$

The step-size, $\mu(n)$, is time-varying and is given by

$$\mu(n) = \frac{\alpha}{\|\mathbf{x}(n)\|^2 + \beta} \tag{2.17}$$

where α is the convergence parameter, β is a small constant preventing division by zero, and $\|\cdot\|$ is the l_2 norm operation. The quantity $\|\mathbf{x}(n)\|^2$ will be referred to as the regressor energy. By normalizing the convergence parameter by the regressor energy, large values of $\mathbf{x}(n)$ have a minimal affect on the adaptation. In other words, unlike the LMS algorithm, the NLMS algorithm does not suffer from the gradient noise amplification problem [15].

The NLMS algorithm has guaranteed stable convergence when the convergence parameter is chosen within the range

$$0 < \alpha < 2. \tag{2.18}$$

This range is independent of the input signal statistics, thus making selection of α much easier than selection of μ for the LMS algorithm. The other advantage of the NLMS algorithm is that it can potentially converge faster than the LMS algorithm [12].

2.5 Delayed Weight Adaptation

The feedback error of the LMS algorithm limits the speed of adaptation and prohibits pipelining. Pipelining is a technique of breaking up the effective critical path by inserting delays, thereby facilitating either low-power or high-speed architectures [22]. To allow pipelining, (2.13) can be modified by inserting delays of D samples, resulting in the weight update equation for the delayed least-mean-square (DLMS) algorithm given by

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu e(n-D)\mathbf{x}(n-D). \tag{2.19}$$

Since the convergence speed of the DLMS algorithm worsens as D increases, D should be kept as small as possible [18].

Likewise, delaying the weight adaptation can be extended to the NLMS algorithm [3]. The weight update equation of the delayed normalized least-mean-square (DNLMS) algorithm is given by

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu(n-D)e(n-D)\mathbf{x}(n-D). \tag{2.20}$$

Chapter 3

Computationally-efficient DNLMS-based Algorithms

In this chapter, computationally-efficient techniques are applied to the DNLMS algorithm in order to reduce power and/or area consumption. NEC and AEC simulations are provided to show that applying these techniques introduce marginal performance degradation. Please note that the material presented in this chapter has been published in [17].

3.1 Computationally-efficient techniques

As seen in (2.1), calculation of the ALC output requires N multiplications and $N-1$ additions per iteration. Furthermore, (2.20) indicates that the DNLMS weight update requires $N+1$ multiplications and N additions per iteration. Therefore, the number of required multiplications and additions per iteration is proportional to the adaptive filter order. As mentioned in Chapter 1, high-order adaptive filters are typically applied in echo cancellation. The related high demand for large amounts of computations per iteration translates to high power consumption for the hardware implementation. There are several modifications that can be made to an adaptive filtering algorithm to reduce power and/or area consumption. Two general ways considered here are (1) simplifying computationally-intensive operations of the algorithms and (2) reducing the switching activity in the device.

The large number of multipliers is the main reason for the high power consumption. Adders are not as large of a concern because they are considerably simpler than multipliers. For example, a 4-bit ripple carry adder requires 20 logic gates while a 4-bit by 4-bit binary multiplier requires 16 logic gates plus 3 4-bit adders [20]. Simplifying the multiplications to less complicated operations, such as additions or shifts, would reduce power and area consumption. This can be achieved by using power-of-two (POT) quantization [10, 34].

Dynamic power consumption of complementary metal-oxide semiconductor (CMOS) circuits is due to the charging and discharging of the capacitive loads occurring each time a transistor's binary representation switches, i.e. $0 \rightarrow 1$ or $1 \rightarrow 0$ transition [32]. Thus, the higher the amount of switching activity, the higher the dynamic power consumption. Reducing the switching activity would therefore reduce dynamic power consumption. The M-Max algorithm [2] and the stop-and-go algorithm (SAG) [1, 24] both reduce the switching activity by scheduling less weight updates.

The aforementioned computationally-efficient techniques have been applied to the LMS and NLMS algorithms. The remainder of this section shows the application of these techniques to the DNLMS algorithm. Additionally, a new stopping criterion for the SAG algorithm is introduced.

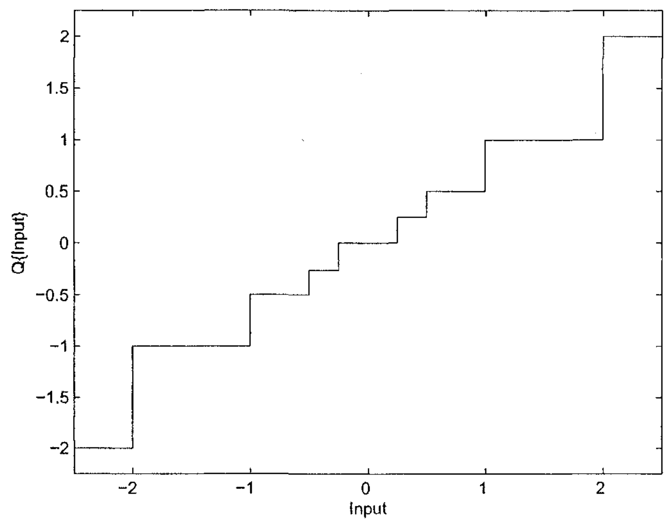


Figure 3.1: Transfer characteristic of POT quantizer for $a = 2, b = 2$, and $\tau = 0$.

3.1.1 Power-of-two Quantization

POT error quantization has been applied to the LMS algorithm in order to simplify N multiplications required for the weight update to shift operations, thereby reducing the computational load [10, 34]. The quantization is a nonlinear operation that results in the error data being represented as a binary word with a single "1" bit. This idea can be extended to the regressor energy, thereby allowing the division operation in (2.17) to be implemented as a shift operation as well. The POT quantization is given as

$$Q\{\cdot\} = \begin{cases} sgn\{\cdot\}2^{a-1}, & |\cdot| \geq 2^{a-1} \\ sgn\{\cdot\}2^{\lfloor log_2(|\cdot|) \rfloor}, & 2^{-b} \leq |\cdot| < 2^{a-1} \\ sgn\{\cdot\}\tau, & |\cdot| < 2^{-b} \end{cases} \tag{3.1}$$

where $a \geq 0$ is the number of integer bits excluding the sign bit, $b \geq 0$ is the number of fractional bits, and τ is set to either 0 or 2^{-b}. Figure 3.1 illustrates the transfer characteristic of the POT quantizer for $a = 2, b = 2$, and $\tau = 0$.

By applying POT quantization to the error and regressor energy, DNLMS is modified to the Quantized-Error-Regressor-energy DNLMS (QER-DNLMS) algorithm, for which the weight update equation is given by

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu(n-D)Q\{e(n-D)\}\mathbf{x}(n-D) \tag{3.2}$$

where

$$\mu(n-D) = \frac{\alpha}{Q\{\|\mathbf{x}(n-D)\|^2 + \beta\}}. \tag{3.3}$$

Note that if α is chosen to be a POT number, then QER-DNLMS weight update equation will consist of $N+1$ shifts plus 2 POT quantizations in place of N multiplications and 1 division per iteration.

3.1.2 M-Max Algorithm

Partial update algorithms update only a portion of the filter weights, effectively reducing the demand of memory resources and computation power when implementing adaptive filtering algorithms on digital signal processors (DSPs) [11]. Since the computational cost of adaptive filtering algorithms is proportional to the filter order, partial update algorithms are most effective in high-order filter applications such as echo cancellation. Partial update algorithms are considered for hardware implementation because updating only a portion of the weights would decrease the switching activity in the device, thereby reducing the dynamic power consumption.

A straightforward selective-partial weight update algorithm is the M-Max algorithm [2]. The M-Max algorithm, which was originally applied to the NLMS algorithm, only updates the weights corresponding to the M largest absolute values of the regressor, where $M < N$. The M-Max-NLMS algorithm saves $N - M$ weight updates per iteration while maintaining close performance to NLMS. Extending this algorithm to DNLMS yields the M-Max-DNLMS algorithm, for which the weight update equation is given by

$$w_i(n+1) = \begin{cases} w_i(n) + \mu(n-D)e(n-D)x(n-i-D), & \text{if } i \text{ corresponds to one} \\ & \text{of the first } M \text{ maxima} \\ & \text{of } |x(n-i-D)| \\ w_i(n), & \text{otherwise} \end{cases} \tag{3.4}$$

where $i = 0, ..., N-1$. Compared to the DNLMS algorithm, the M-Max-DNLMS algorithm has $N - M$ less multiplications and additions per iteration. The overhead cost of this M-Max algorithm includes implementing a sorting algorithm. If the SORTLINE sorting algorithm [25] is used, the amount of additional comparisons per iteration would be at most $\lfloor 2log_2 N \rfloor + 2$.

3.1.3 Stop-and-go Algorithm

A SAG technique was first introduced in [24] to improve the convergence capabilities of decision-aided blind joint equalization and carrier recovery. The idea behind this algorithm is to "stop" adaptation or let it "go" based on the error at the particular sampling time under consideration. In [1], the SAG concept was applied to the NLMS algorithm in order to reduce the amount of computations. In this SAG algorithm, when the magnitude of the error is below a pre-defined threshold, weight adaptation is stopped for that iteration. The weight update equation for the SAG-NLMS algorithm is given by

$$\mathbf{w}(n+1) = \mathbf{w}(n) + f(n)\mu(n)e(n)\mathbf{x}(n) \tag{3.5}$$

where

$$f(n) = \begin{cases} 1, & |e(n)| > \kappa \\ 0, & |e(n)| \leq \kappa \end{cases} \tag{3.6}$$

In (3.6), κ is a positive real number and $f(n)$ is the flag indicating whether or not to update the coefficients. In [1], κ was determined by observing the statistics of $|e(n)|$

over a large number of iterations. Here, the SAG-threshold is related to the regressor energy.

Consider the weight update correction term of the NLMS algorithm given by

$$\begin{aligned}\Delta\mathbf{w}(n) &= \mathbf{w}(n+1) - \mathbf{w}(n) \\ &= \frac{\alpha}{\|\mathbf{x}(n)\|^2} e(n)\mathbf{x}(n)\end{aligned} \tag{3.7}$$

where, for simplicity, the β term has been omitted. The weight update should be stopped when the $|e(n)|$ is small so that $|\Delta\mathbf{w}(n)|$ is significantly small and $\mathbf{w}(n+1) \approx \mathbf{w}(n)$. To ensure that this condition is true for all values in the vector $\Delta\mathbf{w}(n)$, let the stopping criterion be defined in terms of the largest absolute value of $\Delta\mathbf{w}(n)$, which is associated with the largest absolute value of $\mathbf{x}(n)$. The new SAG-stopping criterion is defined as $\max\{|\Delta\mathbf{w}(n)|\} \leq \kappa$, where again κ is a positive real number. Substituting (3.7) into this condition gives

$$|e(n)| \leq \frac{\kappa}{\alpha \max\{|\mathbf{x}(n)|\}} \|\mathbf{x}(n)\|^2. \tag{3.8}$$

To avoid division, (3.8) can be rewritten as

$$\frac{\alpha}{\kappa} \max\{|\mathbf{x}(n)|\}\,|e(n)| \leq \|\mathbf{x}(n)\|^2 \tag{3.9}$$

where the ratio $\frac{\alpha}{\kappa}$ can be implemented as a single constant. Now, applying the SAG algorithm to DNLMS with the new stopping criterion gives SAG-DNLMS, for which the weight update equation is given by

$$\mathbf{w}(n+1) = \mathbf{w}(n) + f(n-D)\mu(n-D)e(n-D)\mathbf{x}(n-D) \tag{3.10}$$

where

$$f(n-D) = \begin{cases} 1, & \|\mathbf{x}(n-D)\|^2 < \frac{\alpha}{\kappa}\max\{|\mathbf{x}(n-D)|\}\,|e(n-D)| \\ 0, & \|\mathbf{x}(n-D)\|^2 \geq \frac{\alpha}{\kappa}\max\{|\mathbf{x}(n-D)|\}\,|e(n-D)| \end{cases} \tag{3.11}$$

Assuming that α and κ are chosen such that the ratio $\frac{\alpha}{\kappa}$ is a POT number, then when the weight adaptation is stopped, there is a savings of N multiplications, N

additions, and 1 shift for the weight update as well as 1 division for the step-size calculation for that iteration. One overhead cost of the SAG algorithm is the calculations of $f(n-D)$, which requires one comparison and two multiplications per iteration. However, if the constants α and κ are power-of-two numbers, then one of the multiplications can be replaced with a shift operation. Another overhead cost is the implementation of a max selection algorithm. A fast algorithm for maximum/minimum calculation across a sliding data window has been proposed in [13] and was labeled the MAXLIST algorithm. This algorithm requires three comparisons and $O(\log N)$ memory locations on average for independent and identically distributed (i.i.d.) input signals. However, if the SAG algorithm is to be used with the M-Max algorithm, then the sorting algorithm can also serve to find the maximum values of the regressor.

3.1.4 Proposed Algorithm

The proposed algorithm is the DNLMS modified with all the techniques previously mentioned in this section. Its coefficient update equation is given by

$$w_i(n+1) = \begin{cases} w_i(n) + f(n-D)\mu(n-D)Q\{e(n-D)\}x(n-i-D), & \text{if } i \text{ corresponds to one of the first } M \text{ maxima of } |x(n-i-D)| \\ w_i(n), & \text{otherwise} \end{cases} \tag{3.12}$$

where $f(n-D)$ is given by

$$f(n-D) = \begin{cases} 1, & \|\mathrm{x}(n-D)\|^2 < \frac{\alpha}{\kappa}\max\{|\mathrm{x}(n-D)|\}\,|Q\{e(n-D)\}| \\ 0, & \|\mathrm{x}(n-D)\|^2 \geq \frac{\alpha}{\kappa}\max\{|\mathrm{x}(n-D)|\}\,|Q\{e(n-D)\}| \end{cases} \tag{3.13}$$

and $\mu(n-D)$ is that in equation (3.3).

Assuming that α and κ are POT numbers, the combined techniques have the effect of: (1) simplifying N multiplications and 1 division to $N+1$ shifts and 2 POT quantizations; (2) saving $N-M$ shifts and additions per "GO" iteration; and (3) saving $M+2$ shifts and M additions per "STOP" iteration.

Table 3.1 summarizes the total number of multiplications, divisions, additions, shifts, and comparisons that execute over m input samples for adaptive filtering when the DNLMS algorithm and its variants previously introduced are used. The amount of computations was derived under the following assumptions: α is a POT number for all algorithms, resulting in at least one shift operation in the coefficient update calculation; the ratio $\frac{\alpha}{\kappa}$ is implemented as a single constant equal to a POT number; the regressor energy is calculated recursively as $\|\mathbf{x}(n)\|^2 = \|\mathbf{x}(n-1)\|^2 + x^2(n) - x^2(n-N)$, requiring 2 multiplications and 2 additions per iteration; and the SAG algorithms have only g out of m samples in the "GO" mode. It can be seen that an adaptive filter using the proposed algorithm has N less multiplications and 1 less division per iteration compared to the DNLMS algorithm at the cost of additional shifts and comparisons. The number of reduced additions is dependent on the choice of M and how often the proposed algorithm is in "STOP" mode.

3.2 Simulation Results

In this section, two simulation examples are presented to compare the performance of all algorithms discussed previous section. See Appendix A for the Matlab source code.

Table 3.1: Number of Operations Executed over m input samples

Algorithm	Multiplications	Divisions	Additions	Shifts	Comparisons
DNLMS	$m(2N+2)$	m	$m(2N+3)$	m	0
QER-DNLMS	$m(N+2)$	0	$m(2N+3)$	$m(N+2)$	0
M-Max-DNLMS	$m(M+N+2)$	m	$m(M+N+3)$	m	$m(2\lfloor log_2 N \rfloor + 2)$
SAG-DNLMS	$gN+m(N+3)$	g	$gN+m(N+3)$	$g+m$	$4m$
Proposed algorithm	$m(N+2)$	0	$gM+m(N+3)$	$g(M+2)+2m$	$m(2\lfloor log_2 N \rfloor + 3)$

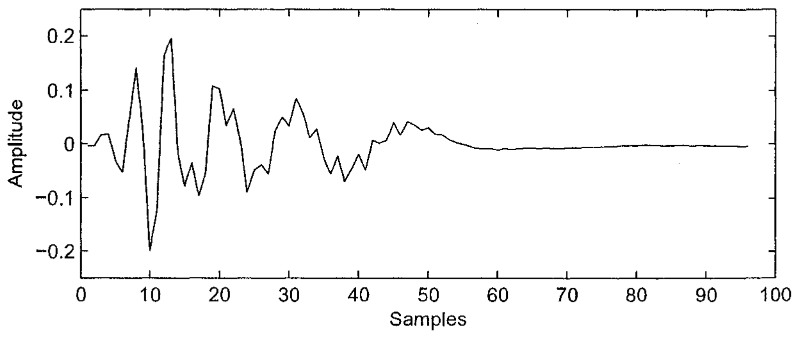
3.2.1 Network Echo Cancellation with White Gaussian Input

In this set of simulations, the performance of each algorithm mentioned in the previous sections is investigated under varying parameters for NEC. Simulations are carried out using an echo path impulse response model from the International Telecommunication Union (ITU) G.168 Recommendation [16], shown in Fig. 3.2(a). The input is white Gaussian noise (WGN) with signal-to-noise ratio (SNR) of 30 dB. The echo return loss (ERL), which is the ratio of the input signal power to the echo signal power, is 6 dB. The filter length is chosen to equal the channel length, i.e. $N = 96$. All simulations have parameters $\alpha = 0.5$, $\beta = 0.008$, and $D = 32$. The MSE is calculated as the average instantaneous squared error over 200 trials.

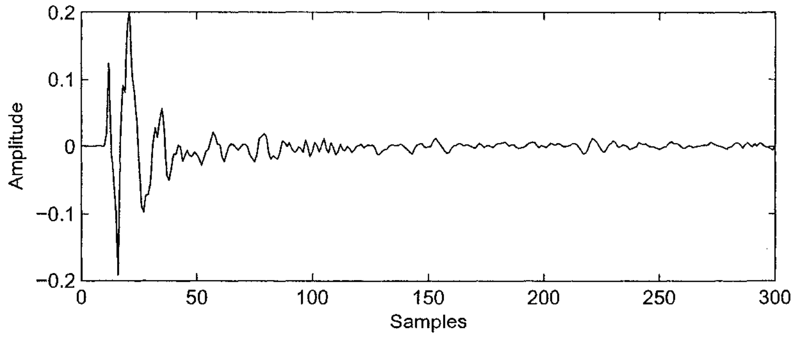
The first simulation results show how DNLMS is affected by POT quantization. Quantized-Error DNLMS (QE-DNLMS) has POT quantization of the delayed error $e(n-D)$ to an 8-bit word ($a = 1, b = 6$). Quantized-Regressor-energy DNLMS (QR-DNLMS) has POT quantization of the delayed regressor energy $\|x(n-D)\|^2$ to an 8-bit word ($a = 7, b = 0$). As mentioned in the previous section, QER-DNLMS has POT quantization of both the delayed error and regressor energy to the same wordlengths used for QE-DNLMS and QR-DNLMS respectively. For QE-DNLMS, $\tau = 0$ and for QR-DNLMS, $\tau = 2^{-b}$ because both achieved better performances for those choices of τ. Figure 3.3 shows that, compared to DNLMS, QE-DNLMS converges slower but achieves a lower steady-state MSE, QR-DNLMS converges slower and achieves a higher steady-state MSE, and QER-DNLMS achieves similar performance.

The next simulation shows the affects of using different values of M for M-Max-DNLMS. Note that for $M = N$ the M-Max-DNLMS is reduced to DNLMS. Figure 3.4 shows that as M decreases, the MSE convergence time decreases.

Next, simulations to investigate how varying κ affects the MSE learning curve of SAG-DNLMS are carried out. Note that $\kappa = 0$ represents DNLMS. It is shown in Fig. 3.5 that as κ increases, convergence time increases. Table 3.2 shows how



(a) A hybrid echo path from ITU G.168.



(b) An acoustic echo path of the inside of a car.

Figure 3.2: Echo path impulse responses.

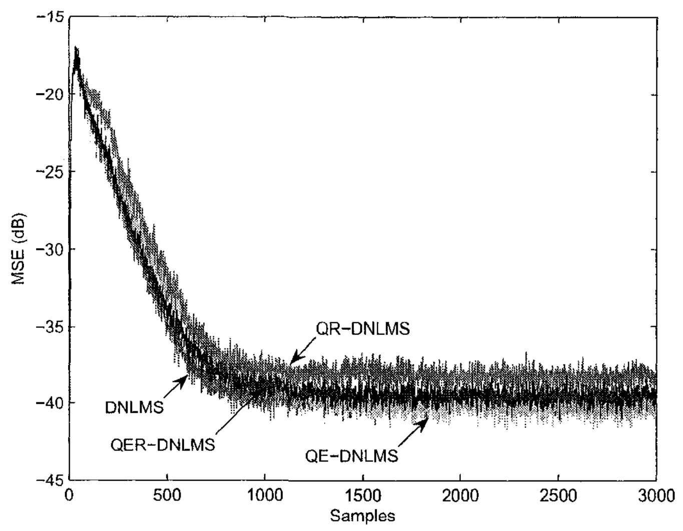


Figure 3.3: MSE curves of DNLMS under different quantization algorithms.

often, on average over 200 trials, the SAG-DNLMS coefficients were updated before and after convergence. This table also includes results for the proposed algorithm, which will be discussed later. For SAG-DNLMS, it can be seen that as κ increases, the percentage of samples in the "GO" mode decreases drastically, especially after convergence.

Finally, the performance of the proposed algorithm is compared to that of the

Table 3.2: Impact of SAG algorithm under WGN input

Algorithm	κ	Percent Samples in "GO" mode	
		Before Convergence	After Convergence
SAG-DNLMS	0.0005	63.24	35.32
SAG-DNLMS	0.0010	31.64	6.75
SAG-DNLMS	0.0015	21.16	1.41
Proposed	2^{-11}	44.97	14.13

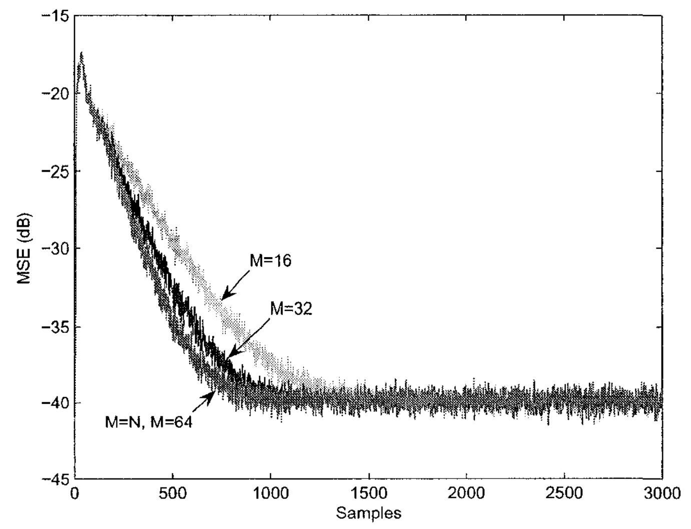


Figure 3.4: MSE curves of M-Max-DNLMS for different M's.

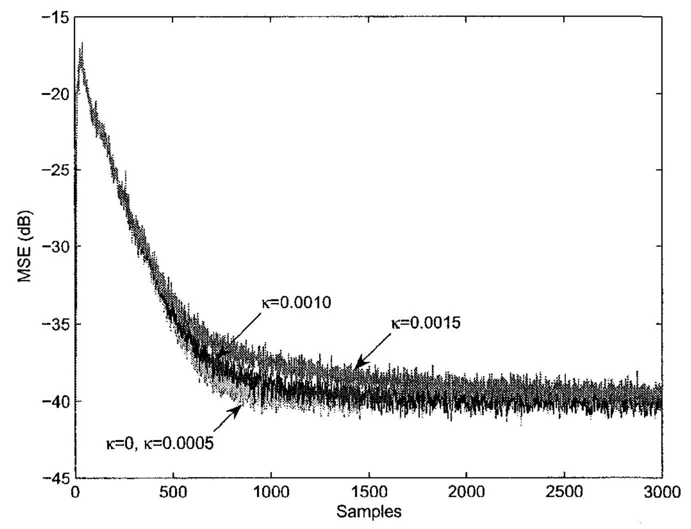


Figure 3.5: MSE curves of SAG-DNLMS for different κ's.

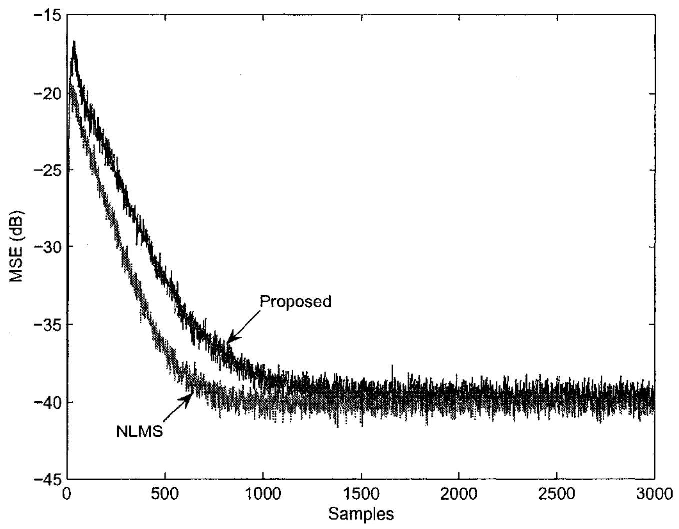


Figure 3.6: MSE curves of NLMS and Proposed algorithm.

standard NLMS algorithm. The parameter chosen include $D = 32$, $M = 32$, $\kappa = 2^{-11}$, quantization of $e(n-D)$ to an 8-bit word $(a = 1, b = 6, \tau = 0)$, and quantization of $\|x(n-D)\|^2$ to an 8-bit word $(a = 7, b = 0, \tau = 2^{-b})$. From Fig. 3.6, it can be seen that the proposed algorithm has moderate performance degradation when compared to NLMS. From Table 3.2, it can be seen that the proposed algorithm experiences significant reductions in computations due to its SAG-related portion alone.

3.2.2 Network and Acoustic Echo Cancellation with Composite Source Signal Input

In this simulation example, NLMS and the proposed algorithm are simulated for both NEC and AEC applications. The input used in this simulation is the composite source signal (CSS) from ITU G.168. The CSS has been downsampled to 8 kHz. It is approximately 350 ms long and consists of a 48.62 ms duration voice signal, a 200 ms duration pseudo-noise signal, and a 101.38 ms duration pause. This sequence is

Table 3.3: Impact of SAG on Proposed algorithm under CSS input

	Percent Samples in "GO" mode		
	Voice	Pseudo Noise	Pause
NEC	32.13	42.23	2.42
AEC	34.33	50.54	6.28

repeated as many times as needed, with an inversion at each repetition, to create a longer signal.

For NEC, the echo path shown in Fig. 3.2(a) is once again used. For AEC, the echo path impulse response model of the inside of a car, shown in Fig. 3.2(b), is used. The SNR is 30 dB. The filter lengths are given as $N = 96$ for NEC and $N = 300$ for AEC. Algorithmic parameters for NLMS and the proposed algorithm in both NEC and AEC simulations include $\alpha = 0.125$ and $\beta = 0.008$. Additionally, the proposed algorithm has the following parameters: $M = 32$ for NEC and $M = 128$ for AEC; $\kappa = 2^{-13}$ for NEC and $\kappa = 2^{-14}$ for AEC; and all remaining parameters are the same as the ones used in the first simulation example.

For this simulation example, echo return loss enhancement (ERLE), a typical measure of echo canceller performance, is calculated and plotted. ERLE is defined as

$$ERLE = 10\log_{10}\frac{E[d^2(n)]}{E[(d(n)-y(n))^2]}dB. \tag{3.14}$$

and can be described as the ratio of the power of the echo versus the power of the residual echo.

Figure 3.7 shows the residual echo and corresponding ERLE of NLMS and the proposed algorithm for NEC simulation. The echo is included with the residual echoes and labelled as the case when there is no echo cancellation. The results show that the echo is effecitively cancelled after the first CSS sequence for both algorithms. Also, the proposed algorithm achieves similar ERLE performance to NLMS.

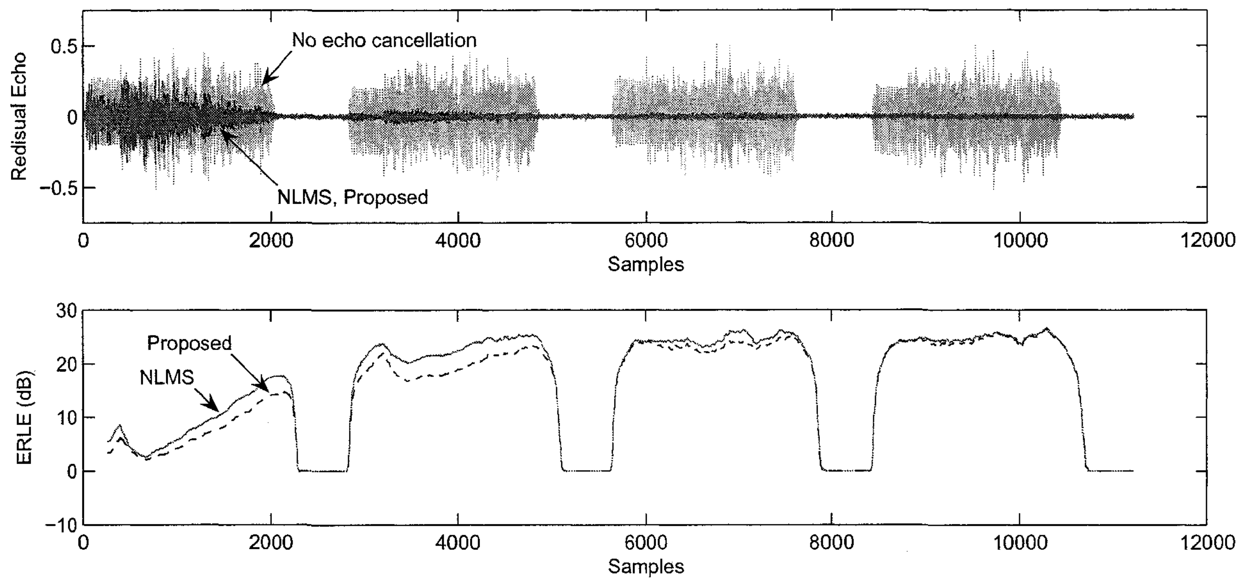


Figure 3.7: Residual echo and ERLE of NLMS and proposed algorithm for NEC simulation.

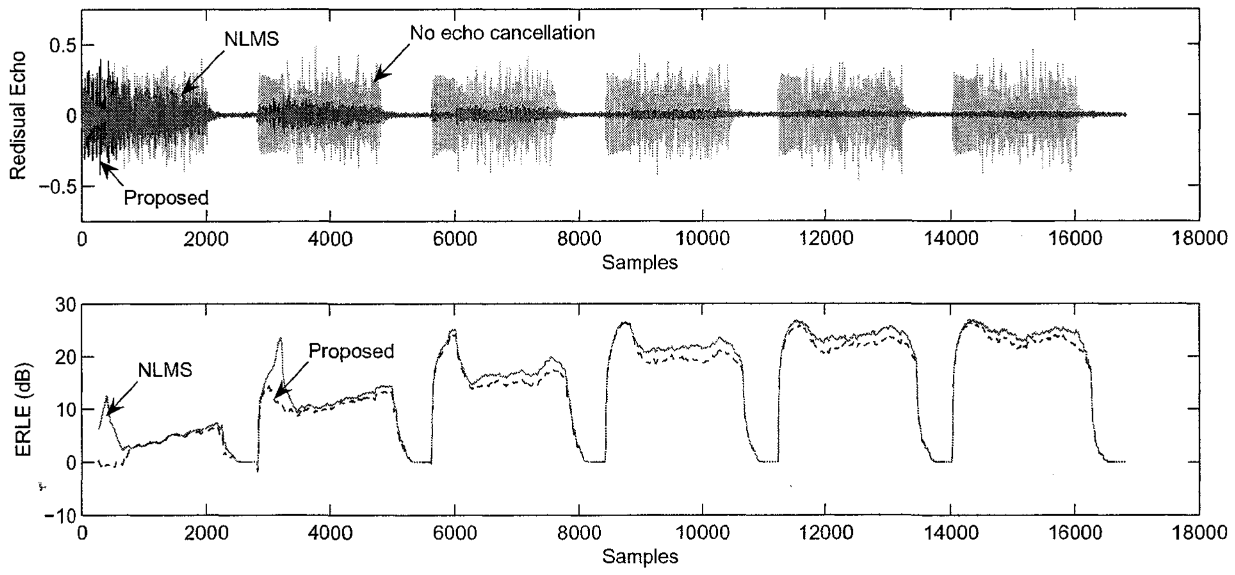


Figure 3.8: Residual echo and ERLE of NLMS and proposed algorithm for AEC simulation.

For AEC simulation, Fig. 3.8 shows that the echo is effectively cancelled after the third CSS sequence. Although the proposed algorithm initially has a lower ERLE performance than NLMS, it achieves similar ERLE performance to NLMS.

Finally, Table 3.3 shows, for the proposed algorithm under NEC and AEC simulations, how often the samples were in the "GO" mode over the voice, pseudo noise, and pause portions of the input. It can be seen that the proposed algorithm provides a significant amount of computational savings, especially during periods of pause.

Chapter 4

FPGA Implementation of Pipelined DNLMS Adaptive Filter

As mentioned in Chapter 2, the DNLMS algorithm is suitable for implementing pipelined adaptive filter architectures. However, to this date there has been little work presenting architectures for DNLMS adaptive filters [23, 26, 28], none having given a detailed implementation. This chapter presents the FPGA implementation of a pipelined, regular, modular, low-latency, portable DNLMS adaptive filter.

This chapter is organized as follows: Section 4.1 shows the derivation of the architecture for the DNLMS adaptive filter. The design is then tested for echo cancellation application. Section 4.2 discusses the design methodology which includes fixed-point and register transfer level (RTL) simulations, physical synthesis for the Altera Stratix EPS140F780C5 FPGA [6], and functional verification results obtained from hardware implementation using the Nios Development Board, Stratix Professional Edition [4].

4.1 Architecture Derivation

A pipelined architecture can be derived using the cutset retiming technique [22]. A cutset is a set of edges in a graph that, when removed, partition the graph into two separate sub-graphs. Cutset retiming is performed by removing k delays from each edge moving in a given direction, from one sub-graph to the other, while adding k delays to the remaining edges moving in the opposite direction, where k is an arbitrary positive integer [22]. This technique does not change the input/output characteristic of the graph. In [7], cutset retiming is used to produce a hybrid FIR form filter architecture, a cross between the direct and transposed FIR forms, for the DLMS adaptive filter. In this section, a pipelined architecture is similarly derived for the DNLMS adaptive filter. Redundant delays elements are then removed for a more efficient implementation. Cutset retiming has been used to derive a pipelined architecture for the DNLMS adaptive filter in [26, 28], however that design was ultimately a folded architecture. Although folded architectures consume less resources, they require operation at higher frequencies than unfolded architectures.

Figure 4.1 shows the direct form DNLMS adaptive FIR filter architecture. Note that D delays are applied to the input $x(n)$ in two places. The delays are applied in this manner to provide a sufficient amount of delays for the retiming technique.

The critical path is defined as the path of the graph not containing a delay element that has the longest computation time. The critical path of the direct form structure is highlighted in Fig. 4.1 and labelled "1^{st} crit. path". Thus, the critical time is $(N+1)t_a + t_m$, where t_a and t_m are the amounts of time required to complete single addition and multiplication operations respectively. Recall from Chapter 1 that echo cancellation requires the use of high-order adaptive filters. In other words, N is large and consequently, so is the critical time of this architecture. Using cutset retiming, delays can be inserted to break up the critical path, thereby decreasing the critical time while preserving the functionality and zero-latency characteristic of the direct form FIR DNLMS adaptive filter architecture.

Cutsets, shown by dashed lines in Fig. 4.1, are applied to the filter such that P weights are grouped together. It is assumed that N is a multiple of P. It will be seen that applying the cutsets in this regular manner results in a regular structure. An additional cutset is applied to the step-size calculation component, seen in the lower-right corner of Fig. 4.1. By doing this, a delay will be inserted in the potential critical path labelled "2^{nd} crit. path", which has a computation time of $3t_a + 3t_m + t_d$, where t_d is the amount of time required to complete a single division operation.

Applying cutset retiming results in the hybrid form DNLMS adaptive FIR filter architecture shown in Fig. 4.2. Identical processing elements (PEs), each containing P weights, are outlined. Two potential critical paths are highlighted. Thus, the critical time for this pipelined architecture is the larger amount between $(P + 1)t_a + t_m$ and $t_a + t_m + t_d$. This is true under the condition that $D > N/P - 1$. Therefore, the minimum value of D is N/P. Note that there is a redundancy of signal values between the regressor, $\mathbf{x}(n)$, and the delayed regressor, $\mathbf{x}(n - D)$. By having one single tap-delay line for the input, the number of delay elements can be reduced by $N - 2N/P + D + 1$.

Figure 4.3 shows the proposed efficient architecture of the hybrid form FIR DNLMS adaptive filter. The contents of i^{th} PE are illustrated in Fig. 4.4, where $i = 0, 1, \ldots,$ $(N/P - 1)$. The values $y_0(n), y_1(n) \ldots, y_{N/P}(n)$ are defined as intermediate values of the output, where $y_0(n) = y(n)$ and $y_{N/P} = 0$. Also, it is shown in Fig. 4.3 that the divider and the delay following it has been replaced with an look-up table (LUT), which is assumed to have an inherent delay of t_d. Since t_d is now equal to the amount of time required to read from memory, the critical time for this design is clearly $(P + 1)t_a + t_m$.

Now the tradeoff introduced by varying the value of P can be discussed. As P decreases, the critical time decreases as well as the degree of data broadcasting of

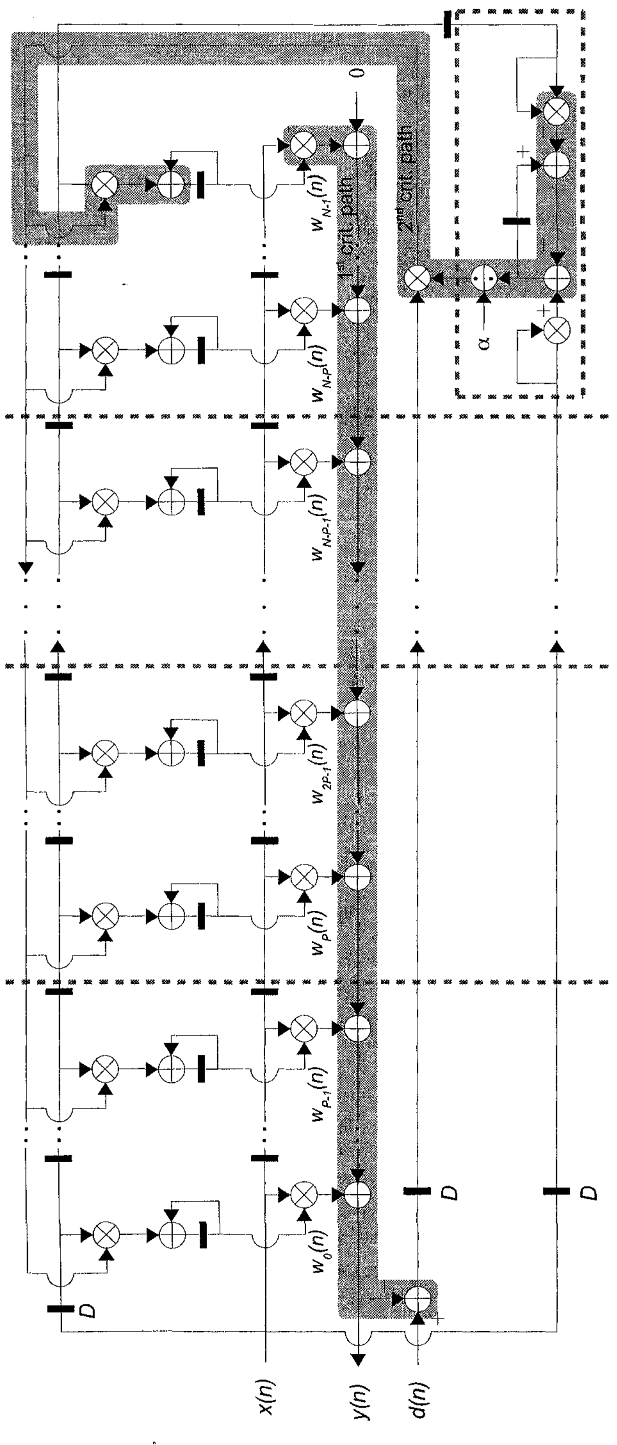


Figure 4.1: Direct form FIR DNLMS adaptive filter architecture with cutsets.

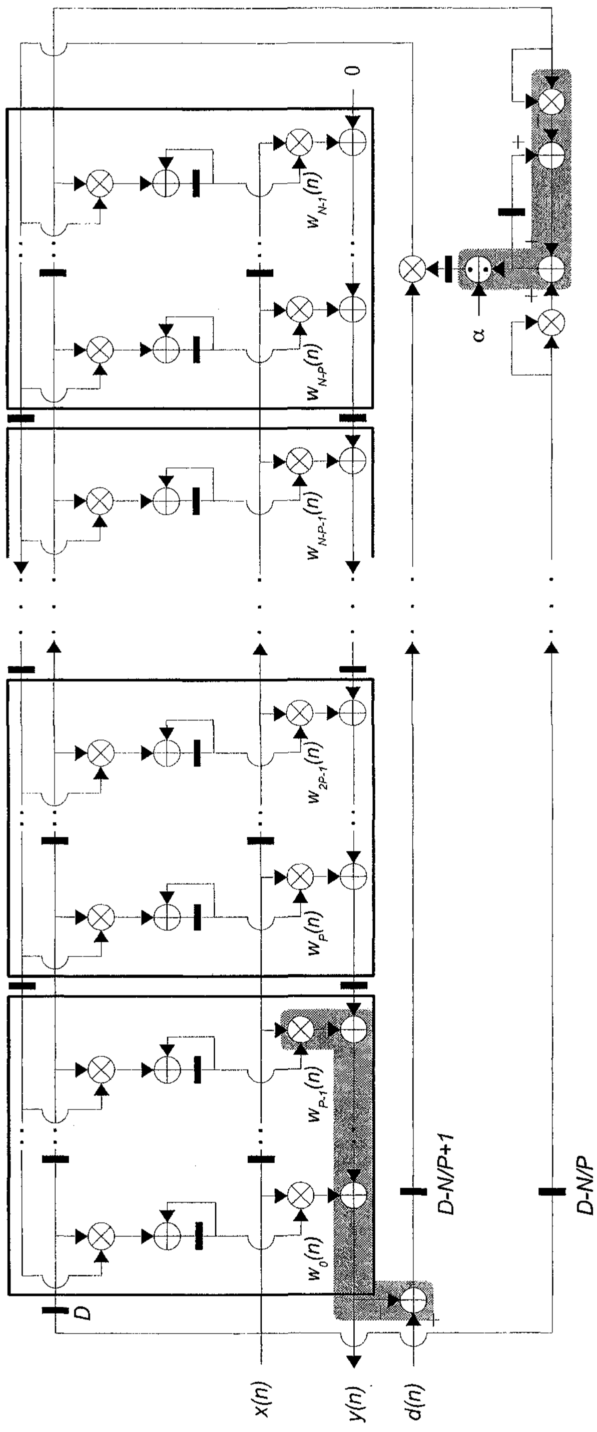


Figure 4.2: Pipelined hybrid form FIR DNLMS adaptive filter architecture.

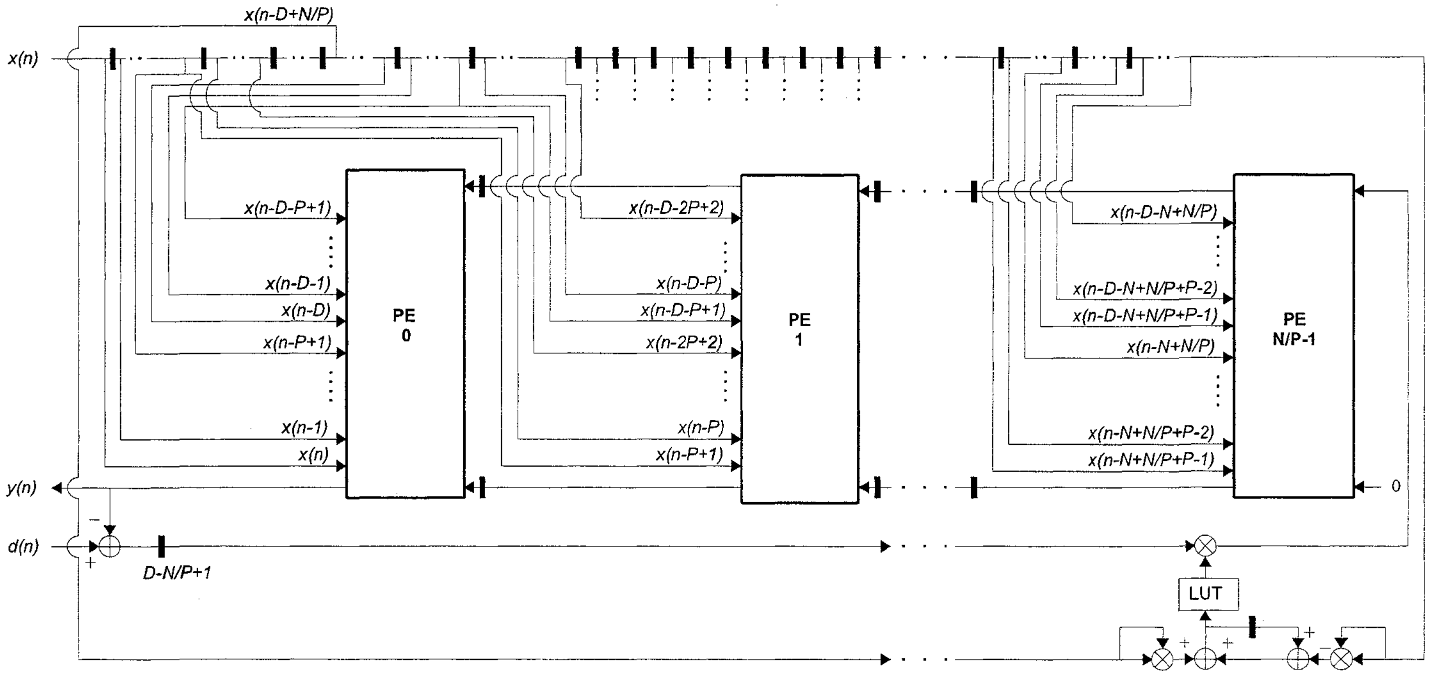


Figure 4.3: Proposed efficient hybrid form FIR DNLMS adaptive filter architecture.

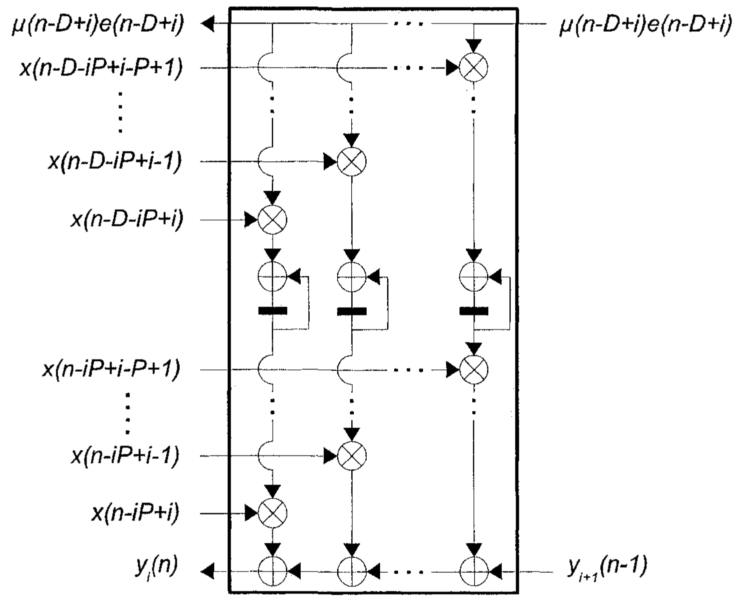


Figure 4.4: The i^{th} PE of the efficient hybrid form FIR DNLMS adaptive filter architecture.

$\mu(n-D+i)e(n-D+i)$ seen in the i^{th} PE. However, the minimum value of D, i.e. N/P, increases. Recall from Section 2.5 that D should be kept as small as possible since a larger value would worsen the convergence behaviour. Therefore, changes in P results in a tradeoff between architectural design characteristics and algorithmic performance.

Table 4.1 summarizes the different critical times and number of delay elements required for each discussed architecture.

Table 4.1: Architecture Comparison

Architecture	Critical Time	No. of Delay Elements
Direct Form	$(N+1)t_a + t_m$	$3N+3D$
Hybrid Form	$\max\{(P+1)t_a + t_m, t_a + t_m + t_d\}$	$3N-2N/P+3D+1$
Proposed	$(P+1)t_a + t_m$	$2N+2D$

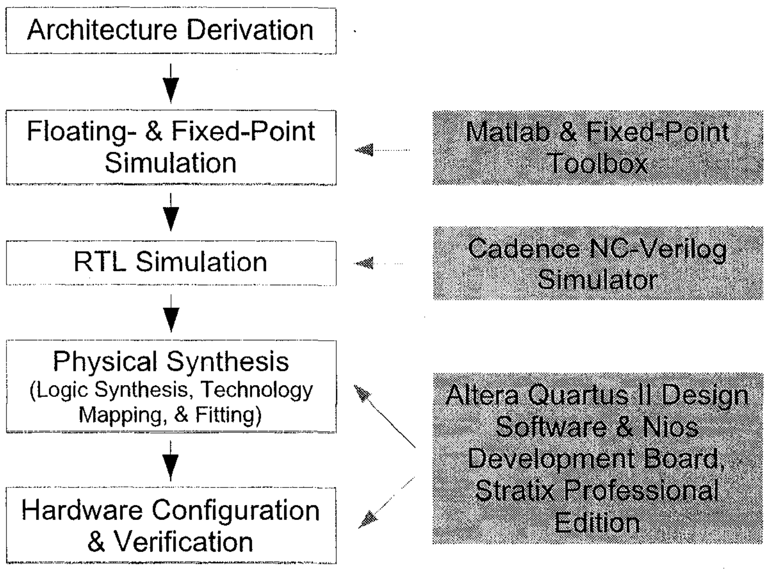

Architecture Derivation
Floating- & Fixed-Point Simulation
RTL Simulation
Physical Synthesis (Logic Synthesis, Technology Mapping, & Fitting)
Hardware Configuration & Verification
Matlab & Fixed-Point Toolbox
Cadence NC-Verilog Simulator
Altera Quartus II Design Software & Nios Development Board, Stratix Professional Edition

Figure 4.5: Design methodology.

4.2 Design Methodology for FPGA Implementation

Figure 4.5 shows the design methodology used to implement the proposed architecture shown in Fig. 4.3. Each step is discussed in further details in the following subsections.

4.2.1 Fixed-Point Simulations

Fixed-point simulations of the pipelined hybrid form FIR DNLMS adaptive filter for echo cancellation are carried out using the fixed-point toolbox in Matlab. See Appendix A for the Matlab fixed-point source code.

The echo path impulse response model illustrated in Fig. 3.2(a) is used to create the desired response signal. The two test input signals used are WGN and the CSS from ITU G.168. The CSS sequence is repeated four times, with an inversion at each repetition, to create a longer signal. The SNR is 30 dB.

The filter length is chosen to equal the channel length, i.e. $N = 96$. Other parameter values were set as $P = 3$ and $\beta = 0.0625$. Recall from Section 4.1 that the design constraint for the adaptation delay was given as $D > N/P - 1$. Since it should also be kept as small as possible, D is chosen to equal 32. The parameter α is equal to 0.5 and 0.125 for WGN input and CSS input respectively. To measure the filter's performance, the instantaneous squared error is calculated for the case of WGN input and ERLE is calculated for the case of CSS input.

Two's complement number system was chosen for the implementation. Overflow is avoided by saturating the sum or product while rounding is handled with truncation. The wordlengths, shown in Table 4.2, were determined by observing the range of all signals in floating-point simulations and by performing fixed-point and RTL simulations to ensure adequate performance.

Figure 4.6(a) shows the instantaneous squared error plotted from fixed-point simulation of the DNLMS adaptive filter when WGN input was applied. It can be seen that the filter exhibits stable behaviour with the squared error having converged to its steady-state value within the first 600 samples. Figure 4.6(b) shows the ERLE performance of the filter when CSS input was applied. It is shown that the echo is effecitively cancelled after the first CSS sequence. The results shown in Fig. 4.6 will be used as a benchmark for checking the equivalence of between the fixed-point

Table 4.2: Signal Wordlengths

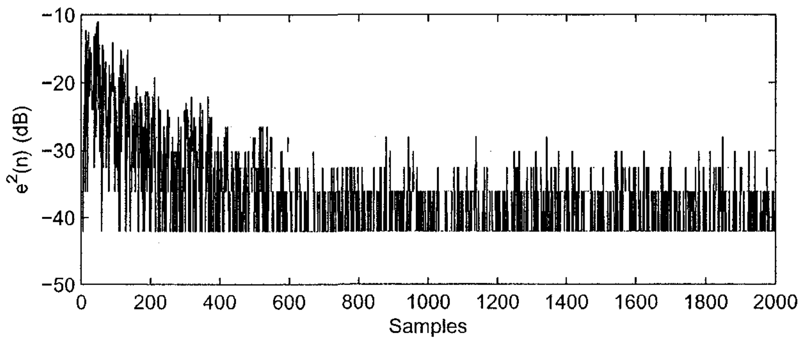
Signal(s)	Wordlengths (Total, Fractional)
$x(n), d(n), e(n), y(n)$	(8, 7)
$\mathbf{w(n)}$	(18, 17)
$\mu(n)$	(11, 7)
$\|\mathbf{x(n)}\|^2$	(12, 7)

design model and the other design models that follow in this section.

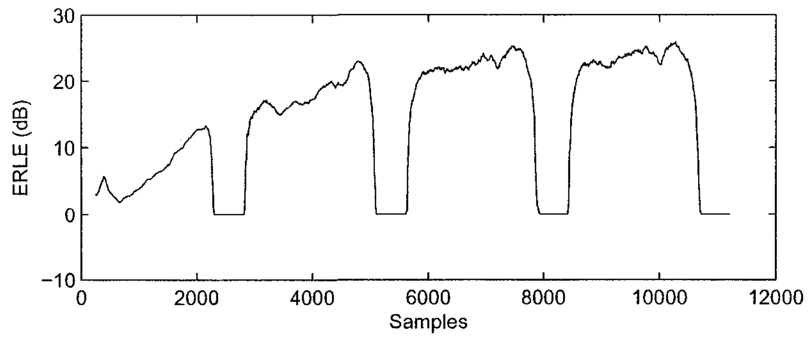
4.2.2 RTL Simulations

The design in Fig. 4.3 was modelled using Verilog Hardware Description Language (HDL). Each Verilog module was written behaviourally and is parametizable, and independent of vendor technology (see Appendix B for Verilog source code). Thus, the design description can be considered as an intellectual property (IP) core. It is flexible and portable, making it suitable for design re-use. These characteristics are important in digital design because it reduces the "time-to-market" for projects using similar architectures.

Cadence NC-Verilog Simulator was used to perform RTL simulations. All parameters and wordlengths have the same values mentioned in Section 4.2.1. Both WGN and CSS inputs are used. Figures 4.7(a) and 4.7(b) show the squared error and ERLE plots from the RTL simulation results, respectively. These plots are nearly identical to the fixed-point simulation plots in Fig. 4.6(a) and Fig. 4.6(b). The only difference is that the error and output from the RTL simulations are delayed by one clock period. This is due to buffering of the input. Thus, the implemented design will have a latency of one. Aside from that difference, the RTL simulations generate the same outputs as fixed-point simulations.

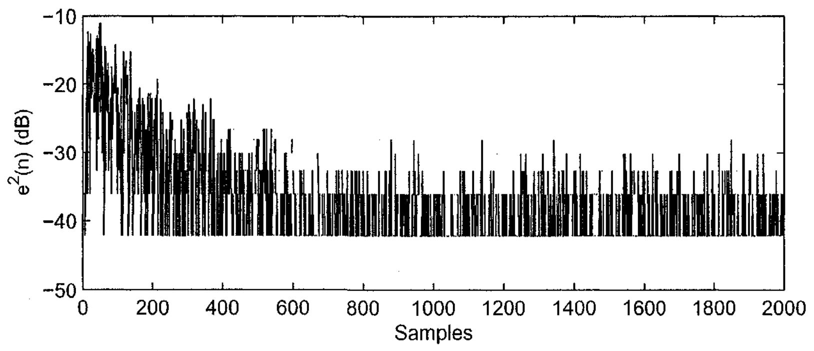


(a) Instantaneous squared error plot from fixed-point simulation for WGN input.

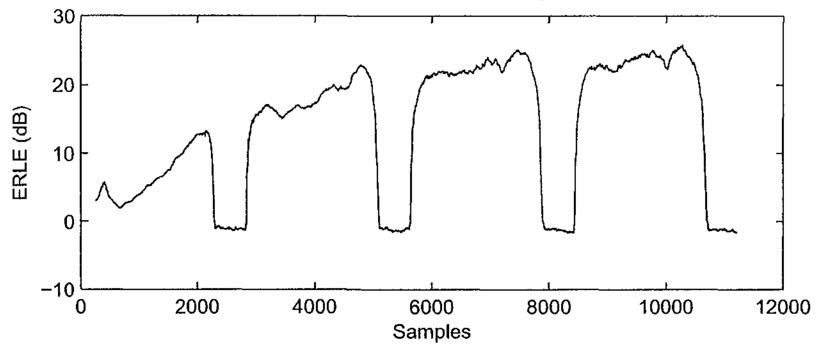


(b) ERLE

Figure 4.6: Results from fixed-point simulations.



(a) Instantaneous squared error



(b) ERLE

Figure 4.7: Results from RTL simulations.

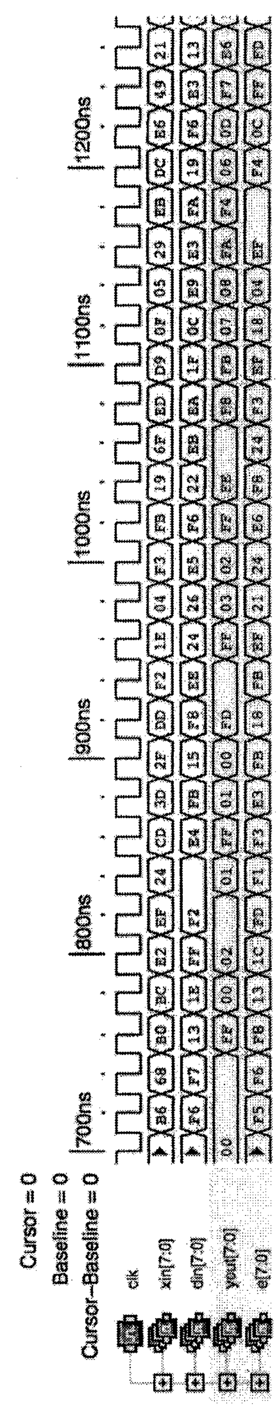
Figure 4.8 shows the captured signal waveforms of $x(n)$, $d(n)$, $y(n)$, and $e(n)$ (which are given alias names $xin, din, yout$, and e, respectively) from the RTL simulations. The chosen time frame is the moment when $y(n)$ begins to have a non-zero output. These results will be used to compare the hardware implementation in Section 4.2.3 to the RTL simulation.

4.2.3 Synthesis and Hardware Verification

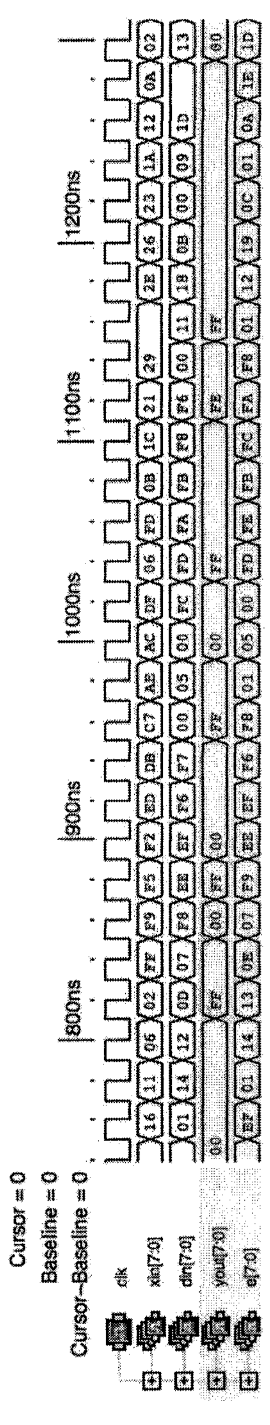
The Verilog behavioural description of the design under test (DUT), i.e. the adaptive filter, was physically synthesized for the Altera Stratix EPS140F780C5 FPGA, then functionally verified using the Altera Nios Development Board, Stratix Professional Edition. The Altera Quartus II 5.0 design software is used to perform logic synthesis, technology mapping, fitting, timing analysis, hardware configuration, and functional verification [5].

To stimulate the DUT, a finite-state machine (FSM) was incorporated into the Quartus project. The FSM was used to reset the DUT, apply inputs, and write outputs to memory. Appendix B contains the Verilog source code for the FSM.

The top-level of the Quartus project is shown in Fig. 4.9. It can be seen that the FSM's I/Os are mapped to an on-board clock pin through a frequency divider module, two on-board push-button pins, and memory modules. The timing analysis performed by the Quartus II Timing Analyzer indicated that the maximum operating frequency of the DUT is 32.27 MHz. Thus, a frequency divider was necessary to reduce the 50 MHz clock supplied on the on-board oscillator. The push-buttons were used to manually reset the FSM to its idle state and to initiate the test. The ROM modules contained the input and desired response signal data as well as the LUT data necessary for the division operation. The RAM modules were used to store the error and output data. Using the In-System Memory Content Editor, the content of the ROM/RAM modules was viewed and manually changed during online operation.



(a) WGN input



(b) CSS input

Figure 4.8: Waveforms from RTL simulation results.

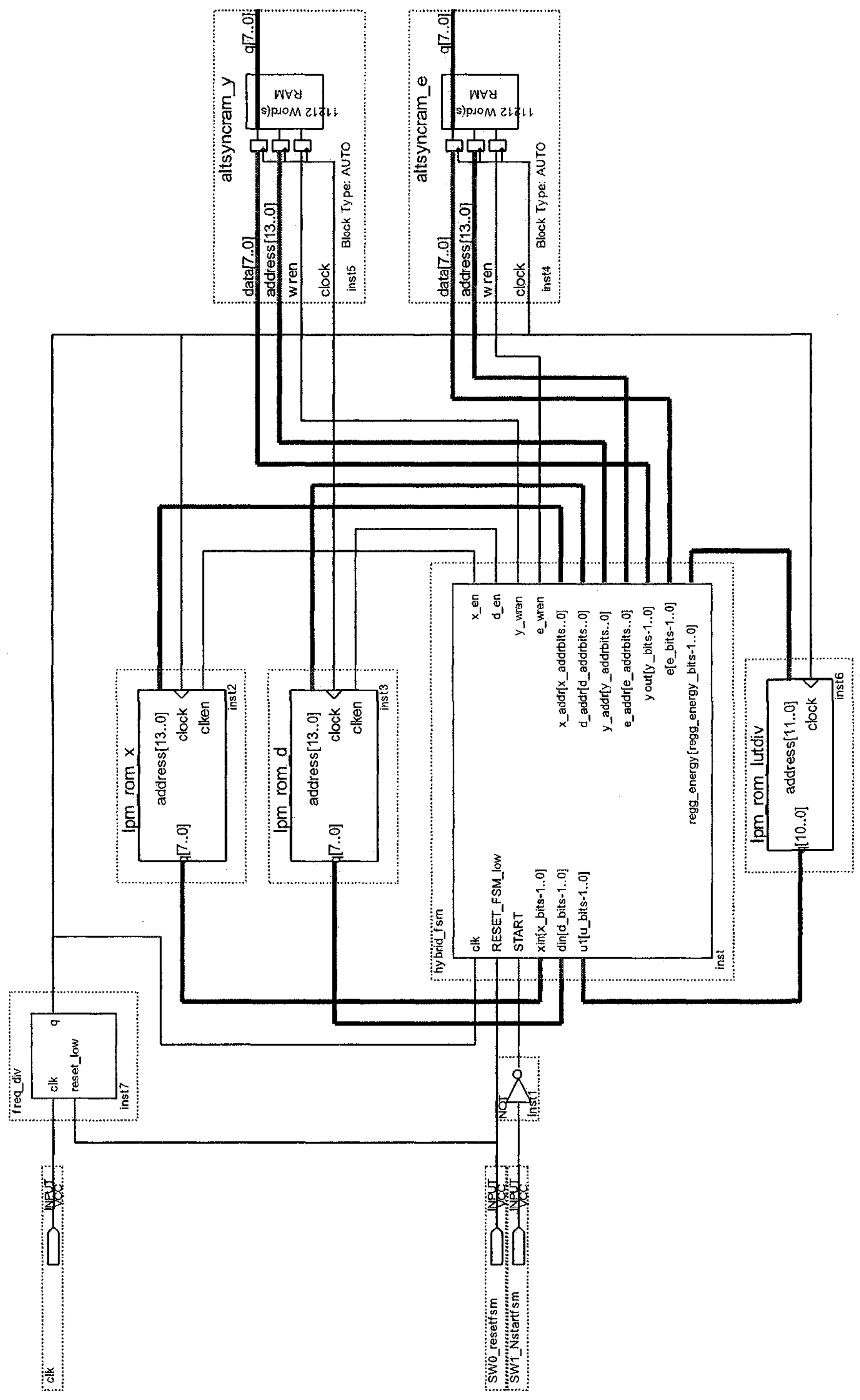


Figure 4.9: Top-level view of Quartus project.

Table 4.3: FPGA Resource Utilization

Resource	Used	Available	Utilization (%)
Logic elements	38,037	41,250	92
Pins	3	616	< 1
Memory bits	407,936	3,423,744	11
DSP block 9-bit elements	112	112	100

In this manner, the error and output data were exported to files.

Table 4.3 summarizes the resource utilization after fitting. It is shown that the implementation requires nearly all of the available logic elements (LEs). An LE is the Altera Stratix FPGA's smallest unit of logic [6]. The digital signal processing (DSP) blocks, which contain dedicated hardware suitable for DSP applications, were completely utilized. The implementation requires few I/O pins and memory resources. To further convey the resource utilization, Fig. 4.10 shows a high-level view of the FPGA after fitting. Unused resources are given light colours. The logic array blocks (LABs), each of which contain 10 LEs, have a colour gradient corresponding to their usage. The darker a LAB is coloured, the more LEs within it are utilized.

After completing physical synthesis and timing analysis, programming files are generated and used to configure the FPGA. After programming the FPGA, WGN and CSS input were applied to the DUT. During online operation, SignalTap II Analyzer was used to capture signal waveforms. Figure 4.11 shows the captured signal waveforms of $x(n)$, $d(n)$, $y(n)$, and $e(n)$ from the hardware tests. The trigger moment used for capturing the data is the instance when $y(n)$ begins to have a non-zero output. It can be seen that the waveforms in this figure identically match the RTL simulation waveforms in Fig. 4.8.

Finally, the error data from the hardware implementation was acquired through the In-System Memory Content Editor and plotted in Fig. 4.12. These plots are

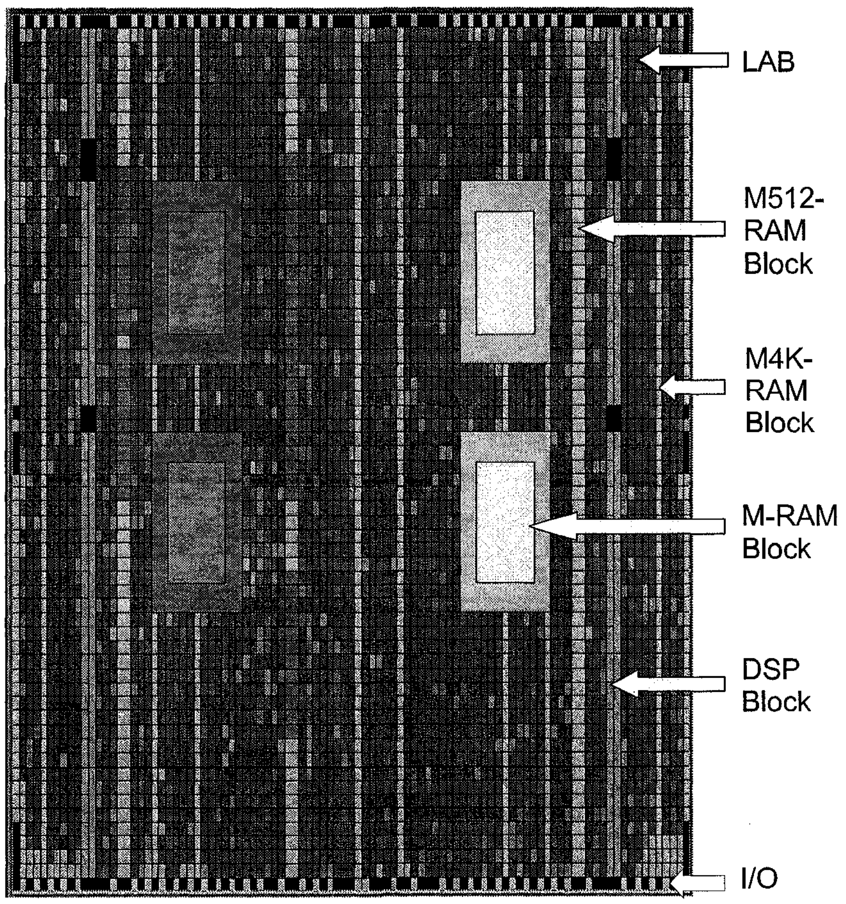
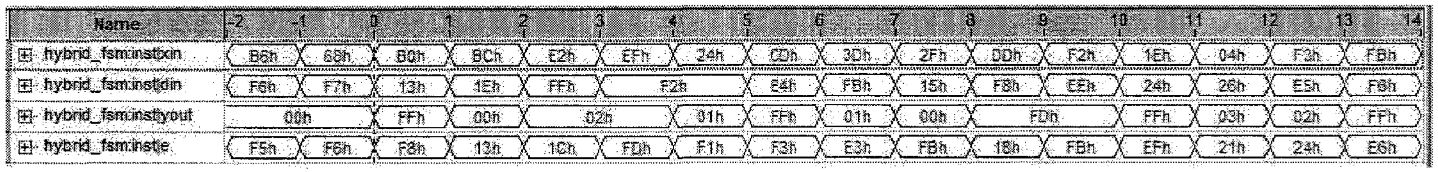
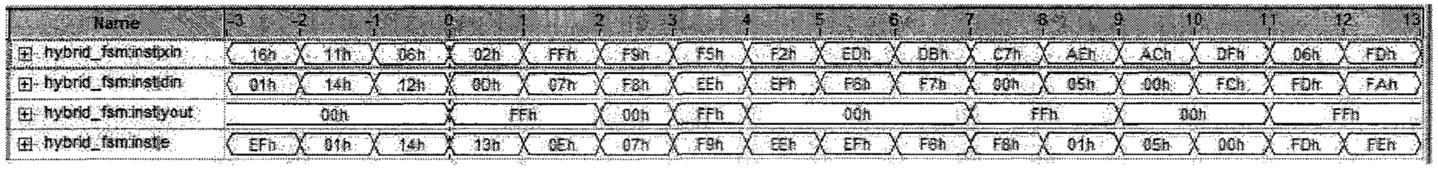


Figure 4.10: Post-fitting chip view illustrating resource utilization.



```
Name                     -2   -1    0    1    2    3    4    5    6    7    8    9   10   11   12   13   14
hybrid_fsm:inst|xin      B6h  68h  B0h  BCh  E2h  EFh  24h  CDh  3Dh  2Fh  DDh  F2h  1Eh  04h  F3h  FBh
hybrid_fsm:inst|din      F6h  F7h  13h  1Eh  FFh  F2h       E4h  FBh  15h  F8h  EEh  24h  26h  E5h  F8h
hybrid_fsm:inst|yout     00h       FFh  00h  02h       01h  FFh  01h  00h  FDh       FFh  03h  02h  FFh
hybrid_fsm:inst|e        F5h  F6h  F8h  13h  1Ch  FDh  F1h  F3h  E3h  FBh  18h  FBh  EFh  21h  24h  E6h
```

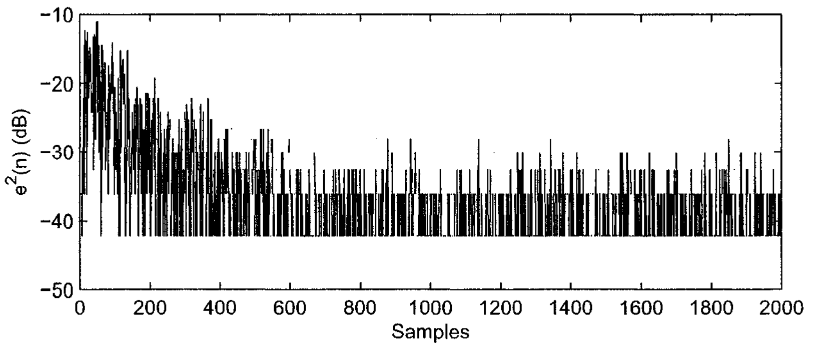
(a) WGN input



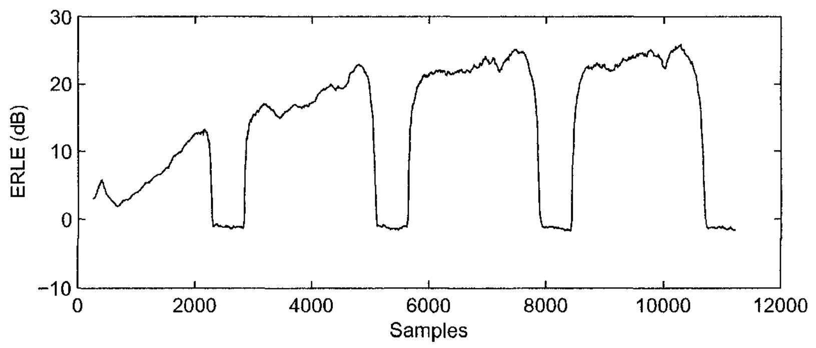
```
Name                     -3   -2   -1    0    1    2    3    4    5    6    7    8    9   10   11   12   13
hybrid_fsm:inst|xin      16h  11h  06h  02h  FFh  F9h  F5h  F2h  EDh  DBh  C7h  AEh  ACh  DFh  06h  FDh
hybrid_fsm:inst|din      01h  14h  12h  00h  07h  F8h  EEh  EFh  F6h  F7h  00h  05h  00h  FCh  FDh  FAh
hybrid_fsm:inst|yout     00h            FFh       00h  FFh  00h            FFh       00h       FFh
hybrid_fsm:inst|e        EFh  01h  14h  13h  0Eh  07h  F9h  EEh  EFh  F6h  F8h  01h  05h  00h  FDh  FEh
```

(b) CSS input

Figure 4.11: Captured waveforms from hardware implementation.



(a) Instantaneous squared error



(b) ERLE

Figure 4.12: Results from hardware simulations.

identical to the RTL simulation plots in Fig. 4.7 and differ the fixed-point simulation plots in Fig. 4.6(a) and Fig. 4.6(b) by a delay of one clock period. Again, this is due to buffering of the input.

Chapter 5

Conclusions and Future Work

This thesis has focused on the implementation of the DNLMS adaptive filter for echo cancellation. The DNLMS algorithm is of interest because unlike the NLMS algorithm, DNLMS allows pipelining, which in turn facilitates low-power or high-speed architectures. Two issues related to the implementation were addressed.

The first issue is that echo cancellation typically requires high-order adaptive filters which induce high power and area consumption. Application of computationally-efficient techniques to the DNLMS algorithm has been considered. POT quantization was applied to the DNLMS algorithm, which has simplified multiplication/division to a single shift. Simplifying such computationally-intensive operations has the effect of reducing power and area consumption. The DNLMS algorithm was also modified by using the M-Max algorithm and a SAG algorithm. Both reduce the amount of computations by scheduling less weight updates. Having less computations results in decreased switching activity in the device, which reduces the dynamic power consumption. For the SAG algorithm, a new and effective stopping criterion related to the regressor energy has been introduced. NEC and AEC simulations have shown that, compared to the standard NLMS algorithm, the proposed algorithm experienced only moderate performance degradation when using either WGN input or ITU G.168 CSS input.

The second issue is that, to this date, there has been no work presenting a detailed implementation of a pipelined DNLMS adaptive filter. Thus, the design and implementation of a DNLMS adaptive filter was presented. The design methodology consisted of architectural derivation, fixed-point and RTL simulations, physical synthesis for the Altera Stratix EPS140F780C5 FPGA, and real-time hardware verification using the Nios Development Boad, Stratix Professional Edition. The pipelined, modular architecture was derived from the direct form by applying cutset retiming. Then, to eliminate redundancy of $N - 2N/P + D + 1$ delay elements between two tap-delay lines, the architecture was modified to use a single tap-delay line. The proposed architecture thus achieved the smallest amount of delay elements as well as the least critical time equal to $(P+1)t_a+t_m$. The minimum adaptation delay for this architecture was determined to be N/P. It was also seen that the varying the parameter P resulted in a tradeoff between design characteristics, namely critical time and degree of local signal broadcasting, and algorithmic performance. The implemented design is pipelined and offers modularity, low-latency, and portablility. Fixed-point, RTL, and hardware models of the adaptive filter were tested for NEC using both WGN and CSS as test signals. It was shown that results from fixed-point, RTL, and hardware simulations produce nearly identical outputs, thus verifying the functionality of the implementation.

Future work that could follow this thesis is the implementation of a DNLMS adaptive filter with computationally-efficient techniques. A sorting algorithm and POT quantizer would be included in the implementation. The impact of the techniques, namely savings in power and area, can be measured when comparing this proposed implementation to the DNLMS adaptive filter implementation presented in this work.

References

[1] E. Abdel-Raheem. On computationally-efficient NLMS-based algorithms for echo cancellation. In *Proc. of the 5th IEEE Int. Symp. on Signal Process. and Inform. Technology*, pages 680–684, Athens, Greece, Dec. 2005.

[2] T. Aboulnasr and K. Mayyas. Complexity reduction of the NLMS algorithm via selective coefficient update. *IEEE Trans. on Signal Process.*, 47(5):1421–1424, May 1999.

[3] S. Ahn and P. J. Voltz. Convergence of the delayed normalized LMS algorithm with decreasing step size. *IEEE Trans. on Signal Process.*, 44(12):3008–3016, Dec. 1996.

[4] Altera Corporation. *Nios Development Board Reference Manual, Stratix Professional Edition*, Oct. 2004.

[5] Altera Corporation. *Quartus II Handbook*, 2005.

[6] Altera Corporation. *Stratix II Device Handbook Section I. Stratix Device Family Data Sheet*, July 2006.

[7] K. Azadet and C. Nicole. Low-power equalizer architectures for high-speed modems. *IEEE Comm. Mag.*, 36(10):118–126, Oct. 1998.

[8] F. K. Becker and H. R. Rudin. Application of automatic transversal filters to the problem of echo suppression. *Bell Syst. Tech. Journal*, 45:1847–1850, 1966.

[9] C. Breining, P. Dreiscitel, E. Hansler, A. Mader, B. Nitsch, H. Puder, T. Schertler, G. Schmidt, and J. Tilp. Acoustic echo control. An application of very-high-order adaptive filters. *IEEE Signal Process. Mag.*, 16(4):42–69, Jul. 1999.

[10] P. S. R. Diniz. *Adaptive Filtering, Algorithms and Practical Application.* Kluwer Academic Publishers, Norwell, MA, 2nd. edition, 2002.

[11] K. Doğançy and O. Tanrikulu. Adaptive filtering algorithms with selective partial updates. *IEEE Trans. on Circuits and Syst. II: Analog and Digital Signal Process.*, 48(8):762–769, Aug. 2001.

[12] S. C. Douglas. A family of normalized LMS algorithms. *IEEE Signal Process. Letters*, 1(3):49–51, Mar. 1994.

[13] S. C. Douglas. Running max/min calculation using a pruned ordered list. *IEEE Signal Trans. on Signal Process.*, 44(11):2872 – 2877, Nov. 1996.

[14] S. L. Gay and J. Benesty. *Acoustic Signal Processing for Telecommunication.* Kluwer Academic Publishers, Norwell, MA, 2000.

[15] S. Haykin. *Adaptive Filtering Theory.* Prentice Hall, Englewood cliffs, NJ, 3rd. edition, 1996.

[16] ITU-T. G.168 digital network echo cancellers. Recommendation, 2004.

[17] R. Lee, E. Abdel-Raheem, and M. A. S. Khalid. Computationally-efficient DNLMS-based adaptive algorithms for echo cancellation. *Journal of Comm.*, 7(1):1–8, Nov./Dec. 2006.

[18] G. Long, F. Ling, and J. G. Proakis. The LMS algorithm with delayed weight adaptation. *IEEE Trans. on Acoustic, Speech, and Signal Process.*, 37(9):1397–1405, Sept. 1989.

[19] Y. Lu, R. Fowler, W. Tian, and L. Thompson. Enhancing echo cancellation via estimation of delay. *IEEE Trans. on Signal Process.*, 53(11):4159–4168, Nov. 2005.

[20] M. M. Mano and C. R. Kime. *Logic and Computer Design Fundamentals.* Prentice Hall, Upper Saddle River, NJ, 2nd updated edition, 2001.

[21] K. Murano, S. Unagami, and F. Amano. Echo cancellation and applications. *IEEE Comm. Mag.*, 28(1):49–55, Jan. 1990.

[22] K. K. Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation.* John Wiley & Sons, New York, NY, 1999.

[23] R. Perry, D. R. Bull, and A. Nix. Pipelined DFE architectures using delayed coefficient adaptation. *IEEE Trans. on Circuits and Syst. II: Analog and Digital Signal Process.*, 45(7):868–873, Jul. 1998.

[24] G. Picchi and G. Prati. Blind equalization and carrier recovery using a "stop-and-go" decision-directed algorithm. *IEEE Trans. on Comm.*, 35(9):877–887, Sept. 1987.

[25] I. Pitas. Fast algorithms for running ordering and max/min calculation. *IEEE Trans. on Circuits and Syst.*, 36(6):795–804, Jun. 1989.

[26] S. Ramanathan, S. K. Nandy, and V. Visvanathan. Reconfigurable filter coprocessor architecture for DSP applications. *Journal of VLSI Signal Process.*, 26(3):333–359, Nov. 2000.

[27] S. Ramanathan and V. Visvanathan. Low-power pipelined LMS adaptive filter architectures with minimal adaptation delay. *Integration, the VLSI Journal*, 27(1):1–32, Jan. 1999.

[28] S. Ramanathan, V. Visvanathan, and S. K. Nandy. Architectural synthesis of computational engines for subband adaptive filtering. *Journal of VLSI Signal Process.*, 22(3):173–195, Sept. 1999.

[29] R. J. B. Reynolds and A. W. Rix. Quality VoIP – An engineering challenge. *BT Technology Journal*, 19(2):23–32, Apr. 2001.

[30] M. M. Sondhi. An adaptive echo canceler. *Bell Syst. Tech. Journal*, 46:1851–1854, 1966.

[31] M. M. Sondhi and D. A. Berkley. Silencing echoes on the telephone network. *Proc. of the IEEE*, 68(8):948–963, Aug. 1980.

[32] J. P. Uyemura. *Introduction to VLSI Circuits and Systems.* John Wiley & Sons, New York, NY, 2002.

[33] B. Widrow and S. D. Stearn. *Adaptive Signal Processing.* Prentice Hall, Englewood Cliffs, NJ, 1985.

[34] P. Xue and B. Liu. Adaptive equalizer using finite-bit power-of-two quantizer. *IEEE Trans. on Acoustic, Speech, and Signal Process.*, ASSP-34(6):1603–1611, Dec. 1986.

Appendix A

Matlab Source Code

f_nlms.m

The following source code describes the floating-point function describing the NLMS algorithm.

```
% This function performs NLMS adaptive filtering
% Last updated: Aug 7, 2006
% By Raymond Lee

function [e, y, c] = f_nlms(x, d, alpha, beta, Lf);
% x - input to the filter
% d - desired signal
% alpha - constant in range (0,2)
% beta - small constant greater than 0
% e- error equal to d - y
% y - output of filter
% c - coefficients of the filter
% regg - regressor vector containing Lf last last samples of the input.
% Lf - length of the filter

% Initialize vectors
n = length(x);
```

```
regg = zeros(1, Lf);
y= zeros(1,n);
e= zeros(1,n);
c= zeros(1,Lf);

% perform filtering
for i = 1 : n
      regg = [x(i) regg(1:Lf-1)];
      y(i) = c*(regg.');
      e(i) = d(i) - y(i);
      mu= alpha/(beta + (regg)*(regg.'));
      c= c+ mu*(regg)*e(i);   %update equation
end
```

f_dnlms.m

The following source code describes the floating-point function describing the DNLMS algorithm.

```
% This function performs DNLMS adaptive filtering
% Last updated: Aug 7, 2006
% By Raymond Lee

function [e, y, c] = f_dnlms(x, d, alpha, beta, Lf, Da);
% x - input to the filter
% d - desired signal
% alpha - constant in range (0,2]
% beta - small constant greater than 0
% e- error equal to d - y
% y - output of filter
% c - coefficients of the filter
% regg - regressor vector containing Lf last last samples of the input.
% Lf - length of the filter
% Da - adaptation delay

% Initialize vectors
n = length(x);
regg = zeros(1, Lf);
regg_delayed = zeros(1, Lf);
y= zeros(1,n);
e= zeros(1,n);
c= zeros(1,Lf);

% perform filtering
```

```
for i = 1 : n
      regg = [x(i) regg(1:Lf-1)];
      y(i) = c*(regg.');
      e(i) = d(i) - y(i);

      if ((i-Da)>0)
            regg_delayed = [x(i-Da) regg_delayed(1:Lf-1)];
            regg_energy = regg_delayed*regg_delayed.';
            mu = alpha/(beta + regg_energy);
            c = c + mu*(regg_delayed)*e(i-Da);    %update equation
      end
end
```

f_qspotdnlms.m

The following source code describes the floating-point function describing the QER-DNLMS algorithm.

```
% This function performs POT Quantization on the error and regressor
% engergy of DNLMS adaptive filtering.
% Last updated: Aug 27, 2006
% By Raymond Lee

function [e, y, c] = f_qspotdnlms(x, d, alpha, beta, Lf, Da, qe_ibits,
    qe_fbits, qre_ibits, qre_fbits);
% x - input to the filter
% d - desired signal
% alpha - constant in range (0,2]
% beta - small constant greater than 0
% e- error equal to d - y
% y - output of filter
% c - coefficients of the filter
% regg - regressor vector containing Lf last last samples of the input.
% Lf - length of the filter
% Da - adaptation delay
% qe_ibits, qe_fbit - number of integer and fractional bits (including
    sign
% bit) that the error is quantized to.
% qre_ibits, qre_fbits - number of integer and fractional bits (
    including
% sign bit) that the regressor energy is quantized to.

% Initialize vectors
n = length(x);
```

```
regg = zeros(1, Lf);
regg_delayed = zeros(1, Lf);
y= zeros(1,n);
e= zeros(1,n);
c= zeros(1,Lf);
qe = zeros(1,n);        % quantized error signal

% perform filtering
for i = 1 : n
      regg = [x(i) regg(1:Lf-1)];
      y(i) = c*(regg.');
      e(i) = d(i) - y(i);
      qe(i) = f_powoftwo_anywl2(e(i), qe_ibits, qe_fbits,0);

      if ((i-Da)>0)
            regg_delayed = [x(i-Da) regg_delayed(1:Lf-1)];
            regg_energy = f_powoftwo_anywl2(beta + regg_delayed*
               regg_delayed.', qre_ibits, qre_fbits,1);
            mu = alpha/(regg_energy);
            c = c + mu*(regg_delayed)*qe(i-Da);    %update equation
      end
end
```

f_mmaxdnlms.m

The following source code describes the floating-point function describing the M-Max-DNLMS algorithm.

```
% This function performs M-Max DNLMS adaptive filtering
% Last updated: Aug 7, 2006
% By Raymond Lee

function [e, y, c] = f_mmaxdnlms(x, d, alpha, beta, Lf, Da, M);
% x - input to the filter
% d - desired signal
% alpha - constant in range (0,2]
% beta - small constant greater than 0
% e- error equal to d - y
% y - output of filter
% c - coefficients of the filter
% regg - regressor vector containing Lf last last samples of the input.
% Lf - length of the filter
% Da - adaptation delay
% M- the number of taps that will be updated
% Initialize vectors
n = length(x);
regg = zeros(1, Lf);
regg_delayed = zeros(1, Lf);
y= zeros(1,n);
e= zeros(1,n);
c= zeros(1,Lf);

% perform filtering
for i = 1 : n
      regg = [x(i) regg(1:Lf-1)];
      y(i) = c*(regg.');
      e(i) = d(i) - y(i);

      if ((i-Da)>0)
            regg_delayed = [x(i-Da) regg_delayed(1:Lf-1)];

            % create a vector that flags the largest P values of the
                regressor
            [mm tempii] = sort(abs(regg_delayed));
            tempii= tempii(Lf:-1:1);
            index= zeros(1,Lf);
            for iii = 1 : M
                index(tempii(iii)) = 1;
            end

            regg_energy = regg_delayed*regg_delayed.';
            mu = alpha/(beta + regg_energy);
            c = c + mu*(regg_delayed)*diag(index)*e(i-Da);    %update
                equation
      end
end
```

f_sagdnlms.m

The following source code describes the floating-point function describing the SAG-DNLMS algorithm.

```
% This function performs SAG-DNLMS adaptive filtering
% Last updated: Aug 27, 2006
% By Raymond Lee

function [e, y, c, flag] = f_sagnlms(x, d, alpha, beta, Lf, Da, kappa);
```

```
% x - input to the filter
% d - desired signal
% alpha - constant in range (0,2]
% beta - small constant greater than 0
% e- error equal to d - y
% y - output of filter
% c - coefficients of the filter
% regg - regressor vector containing Lf last last samples of the input.
% Lf - length of the filter
% Da - adaptation delay
% kappa - parameter for SAG

% Initialize vectors
n = length(x);
regg = zeros(1, Lf);
regg_delayed = zeros(1, Lf);
y= zeros(1,n);
e= zeros(1,n);
c= zeros(1,Lf);

% perform filtering
for i = 1 : n
      regg = [x(i) regg(1:Lf-1)];
      y(i) = c*(regg.');
      e(i) = d(i) - y(i);

      if ((i-Da)>0)
            regg_delayed = [x(i-Da) regg_delayed(1:Lf-1)];
            regg_energy = regg_delayed*regg_delayed.';
            mu = alpha/(beta + regg_energy);

            % determine if there will be an update of not
            if (alpha/kappa*abs(e(i-Da))*max(abs(regg_delayed)) >
                regg_energy)
                  f = 1;
                  flag(i) = 1;
            else
                  f = 0;
                  flag(i) = 0;
            end
            c = c + f*mu*(regg_delayed)*e(i-Da);    %update equation
      end
end
```

f_qsmdnlms.m

The following source code describes the floating-point function describing the proposed algorithm.

```
% This function performs proposed (QER SAG M-Max) DNLMS adaptive
    filtering
% Last updated: Aug 27, 2006
% By Raymond Lee

function [e, y, c, flag] = f_qsmdnlms(x, d, alpha, beta, Lf, Da, M,
    kappa, qe_ibits, qe_fbits, qre_ibits, qre_fbits);
% x - input to the filter
% d - desired signal
% alpha - constant in range (0,2]
% beta - small constant greater than 0
% M - the number of taps that will be updated
% kappa - constant value greater than 0; used for SSAG
% qe_ibits, qe_fbit - number of integer and fractional bits (including
    sign
% bit) that the error is quantized to.
% qre_ibits, qre_fbits - number of integer and fractional bits (
    including
% sign bit) that the regressor energy is quantized to.
% e- error equal to d - y
% y - output of filter
% c - coefficients of the filter
% flag - vector that keeps track of number of updates
% regg - regressor vector containing Lf last last samples of the input.
% Lf - length of the filter
% Da - adaptation delay
% f - flag = 0 or 1

% Initialize vectors
n = length(x);
regg = zeros(1, Lf);
regg_delayed = zeros(1, Lf);
y= zeros(1,n);
e= zeros(1,n);
c= zeros(1,Lf);
qe = zeros(1,n);      % quantized error signal

% perform filtering
for i = 1 : n
     regg = [x(i) regg(1:Lf-1)];
     y(i) = c*(regg.');
```

```
        e(i) = d(i) - y(i);
        qe(i) =f_powoftwo_anywl2(e(i), qe_ibits , qe_fbits ,0);

        if ((i-Da)>0)
            regg_delayed = [x(i-Da) regg_delayed(1:Lf-1)];

            % create a vector that flags the largest P values of the
                regressor
            [mm tempii] = sort(abs(regg_delayed), 'descend');
            index= zeros(1,Lf);
            for iii = 1 : M
                index(tempii(iii)) = 1;
            end

            regg_energy =f_powoftwo_anywl2(beta + regg_delayed*
                regg_delayed.', qre_ibits , qre_fbits ,1);
            mu = alpha/(regg_energy);

            % determine if there will be an update of not
            if (alpha/kappa*abs(qe(i-Da))*max(abs(regg_delayed)) >
                regg_energy)
                f = 1;
                flag(i) = 1;
            else
                f = 0;
                flag(i) = 0;
            end

            c = c + f*mu*(regg_delayed)*diag(index)*qe(i-Da);    %update
                equation
        end
end
```

f_powoftwo_anywl2.m

The following source code describes the floating-point function describing POT quantization.

```
% This function returns an error quantized to a power of two. Inputs to
% this function are the error, the number of integer bits (EXCLUDING the
% sign bit), the number of fractional bits , and tau = 0 for 0 or 1 for
% 2^(-fracbits). Note that error is real valued.
% Last Modified Aug. 27, 2006
% Raymond Lee
```

```
function e = f_powoftwo_anywl2(error, intbits, fracbits, tau)

abs_error = abs(error);

% set the value of tau
if tau == 0
      tau = 0;
elseif tau == 1
      tau = 2^(-fracbits);
else
      error('tau should be 0 or 1')
end

% quantize
if abs_error >= 2^(intbits-1)
      e = 2^(intbits-1);
elseif abs_error >= 2^(-fracbits)
      e =  2^(floor(log2(abs_error)));
else
      e = tau;
end

e = sign(error)*e;
```

echo_qsmdnlms_comparison.m

The following source code describes the floating-point NEC simulation of NLMS and the proposed algorithm for WGN input. It specifically generates the results seen in Fig. 3.6 and the last row of Table 3.2. Other results provided in Section 3.2.1 are similarly simulated, but call on different functions previously mentioned in this section to simulate different algorithms.

```
% This program simulates echo cancellation with NLMS and proposed using
% floating-point arithmetic. The channels used here are the hybrid echo
% paths from G.168, a model of a car room, and a model of a room.
% The test signals include white gaussian noise only.
% We will assume an 8 kHz sampling frequency.
% Last modified Aug 27, 2006
% Raymond Lee
```

```
clear all
% close all

%%%%%%%% LIST OF VARIABLES %%%%%%%%%%%%%%%%%%%%
% Lf - filter length
% n - # of samples
% mu - adaptation step-size
% mse - mean squared error
% num_realizations - number of realizations over which the squared error
% is averaged
% d - genuine echo that the filter is trying to match (desired signal)
% y - filter output or echo replica
% error - error signal = d - y
% mse - mean square error
% erle - echo return loss enhancement
% chan - echo path impulse response
% w - weights of the adaptive filter
% regg - regressor which holds current input and Lf-1 past input signals
% x - the input to the adaptive filter (Far-end speaker's signal). x is
% either wgn or CSS.
% noise - Near-end speaker's signal (part of the desired signal)
% alpha - parameter for NLMS - (0,2)
% beta - parameter for NLMS - small and positive
% SNR - signal to noise ratio = x_power/noise_power
% SNRdb - 10log10(SNR)
% Da - amount by which DNLMS adaptation is delayed

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%% PARAMETERS and INITIALIZATION %%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

chan_num_flag = input('Enter 1 for chan g1; 2 for chan g2; .... ; 8 for 
   chan g8; 9 for car_room; 10 for room1: ');
if (~(chan_num_flag == 1 || chan_num_flag == 2 || chan_num_flag == 3 ||
    chan_num_flag == 4 || chan_num_flag == 5 || chan_num_flag == 6 ||
    chan_num_flag == 7 || chan_num_flag == 8 || chan_num_flag == 9 ||
    chan_num_flag ==10))
     error('You were supposed to enter a number from 1 to 10');
end

if (chan_num_flag == 9) % car_room
       n = 5000;
elseif (chan_num_flag == 10)  % car_room
       n = 5000;
else
       n = 3000;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% LOADING CHANNEL %%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%% Echo Path Channels
% Refer to generate_channels.mat for the generation of these channels.
% ERL is set to 6 dB. Channels g1 -> g4 are models from network hybrid
% simulator while channels g5 -> g8 are models from  measured from
% telephone networks in North America.
if (chan_num_flag == 9)
      load mat_dat_files/room_car.mat;
      chan = 0.2*car_room;
elseif (chan_num_flag == 10)
      load mat_dat_files/room1.mat;
      chan = room1;
else
      load mat_dat_files/gecho_paths.mat;
      if (chan_num_flag == 1)
            chan = g1;
      elseif (chan_num_flag == 2)
            chan = g2;
      elseif (chan_num_flag == 3)
            chan = g3;
      elseif (chan_num_flag == 4)
            chan = g4;
      elseif (chan_num_flag == 5)
            chan = g5;
      elseif (chan_num_flag == 6)
            chan = g6;
      elseif (chan_num_flag == 7)
            chan = g7;
      else
            chan = g8;
      end
end

Lf = length(chan);

% SNR and power calculations
x_power = 1;
SNRdb = 30;
SNR = 10^(SNRdb/10);
noise_power = x_power/SNR;

% Parameters
% Note that we use beta = 0 for QSM-DNLMS
if (chan_num_flag == 10)        % room1
      alpha = 0.5;
      beta = 0.008;
      kappa = 2^-13;
      M = 64;
elseif (chan_num_flag == 9)     % car_room
```

```
        alpha = 0.5;
        beta = 0.008;
        kappa = 2^-13;
        M = 64;
else                        % g.168 hybrids
        alpha = 0.5;
        beta = 0.008;
        kappa = 2^-11;
        M = 32;
end

Da = 32;
qe_ibits = 1;
%NOTE: ibits EXCLUDES the sign bit, i.e. total WL = 1 + *_ibits + *
    _fbits
qe_fbits = 6;
qre_ibits = 7;
qre_fbits = 0;

num_realizations = 20;  %WGN input

esq_nlms = zeros(1,n);
esq_qsmdnlms = zeros(1,n);
dsq = zeros(1,n);
flag_qsm = zeros(1,n);

for nr = 1: num_realizations
        nr
        tic

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %%%%%%%%%%%% RESET VECTORS TO ZERO %%%%%%%%%%%%%%%%
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        %%%%%%% NLMS vectors %%%%%%%%
        y_nlms = zeros(1,n);          % output of the filter
        error_nlms = zeros(1,n);      % error signal
        c_nlms = zeros(1,Lf);       % coefficient INITIALIZATION

        %%%%%%% QSM DNLMS vectors %%%%%%%%
        y_qsmdnlms = zeros(1,n);          % output of the filter
        error_qsmdnlms = zeros(1,n);      % error signal
        c_qsmdnlms = zeros(1,Lf);       % coefficient INITIALIZATION
        flag_qsm_temp = zeros(1,n);

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %%%%%%% CREATE DESIRED SIGNAL (ECHO) %%%%%%%%%%%%%%%
        %%%%% WGN input
        x = randn(1,n);       % 1xn vector (var = 1)
        Max_x = max(abs(x));
        x = x./Max_x;
        noise = randn(1,n);
        noise = sqrt(noise_power).*(noise./Max_x);
        d = filter(chan,1,x) + noise;

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %%%%%%%%%%%%% ADAPTIVE FILTERING %%%%%%%%%%%%%%
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        [error_nlms, y_nlms, c_nlms] = f_nlms(x, d, alpha, beta, Lf);
        [error_qsmdnlms, y_qsmdnlms, c_qsmdnlms, flag_qsm_temp] =
            f_qsmdnlms(x, d, alpha, beta, Lf, Da, M, kappa, qe_ibits,
            qe_fbits, qre_ibits, qre_fbits);

        esq_nlms = esq_nlms + error_nlms.^2;
        esq_qsmdnlms = esq_qsmdnlms + error_qsmdnlms.^2;
        flag_qsm= flag_qsm + flag_qsm_temp;
        dsq = dsq + (d.^2);
        toc
end

iter = 1:n;
iterms = iter./8;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% FILTER PERFORMANCE %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
mse_nlms = esq_nlms./num_realizations;
mse_qsmdnlms = esq_qsmdnlms./num_realizations;

mse_nlms = 10*log10(mse_nlms);
mse_qsmdnlms = 10*log10(mse_qsmdnlms);

% Quantitative Analysis returning Steady-state MSE, Misadjustment, and
    Time to convergence (TTC)
[ssmse_nlms, M_nlms, TTC_nlms] = f_qanalysis(mse_nlms, iter, Lf, SNRdb,
    chan);
[ssmse_qsmdnlms, M_qsmdnlms, TTC_qsmdnlms] = f_qanalysis(mse_qsmdnlms,
    iter, Lf, SNRdb, chan);

% Calculate the percent updates before and after convergence
flag_qsm = flag_qsm./num_realizations;
percent_update_qsm = sum(flag_qsm)/n;
percent_update_qsm_bc = sum(flag_qsm(1:TTC_qsmdnlms))/(TTC_qsmdnlms);
percent_update_qsm_ac = sum(flag_qsm(TTC_qsmdnlms:n))/(n-TTC_qsmdnlms);

display('Steady-State_MSE: ') %last 500 samples
ssmse_nlms
```

```
ssmse_qsmdnlms

display('TTC: ')
TTC_nlms
TTC_qsmdnlms
display('Percent updates')
percent_update_qsm
percent_update_qsm_bc
percent_update_qsm_ac

% plots
figure
plot(iter, mse_nlms,'r');
hold on;
plot(iter, mse_qsmdnlms,'b');
xlabel('Samples');
ylabel('MSE (dB)');
legend('NLMS', 'Proposed');
```

echo_qsmdnlms_erle.m

The following source code describes the floating-point NEC and AEC simulation of NLMS and the proposed algorithm for CSS input. It generates the results seen in Section 3.2.2.

```
% This program simulates echo cancellation with NLMS and proposed
% (QER SAG M-Max) DNLMS using floating-point arithmetic. The channels
% used here are the hybrid echo paths from G.168, a model of a car
% room, and a model of a room.  The test signals include CSS from
% G.168 and a voice signal. We will assume an 8 kHz sampling frequency.
% Last modified Aug 28, 2006
% Raymond Lee

clear all
% close all
j = sqrt(-1);

%%%%%%%%%% LIST OF VARIABLES %%%%%%%%%%%%%%%%%%%%%
% Lf - filter length
% n - # of samples
% mu - adaptation step-size
% mse - mean squared error
% num_realizations - number of realizations over which the squared error
    is averaged
% d - genuine echo that the filter is trying to match (desired signal)
% y - filter output or echo replica
% error - error signal = d - y
% mse - mean square error
% erle - echo return loss enhancement
% chan - echo path impulse response
% w - weights of the adaptive filter
% regg - regressor which holds current input and Lf-1 past input signals
% x - the input to the adaptive filter (Far-end speaker's signal). x is
% either wgn or CSS.
% noise - Near-end speaker's signal (part of the desired signal)
% alpha - parameter for NLMS - (0,2)
% beta - parameter for NLMS - small and positive
% SNR - signal to noise ratio = x_power/noise_power
% SNRdb - 10log10(SNR)
% Da - amount by which DNLMS adaptation is delayed

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%% PARAMETERS and INITIALIZATION %%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
input_flag = input('Enter 1 for CSS input or 2 for voice signal: ');
if (~(input_flag == 1 || input_flag == 2))
    error('You were supposed to enter 1 or 2');
end
chan_num_flag = input('Enter 1 for chan g1; 2 for chan g2; ... ; 8 for
    chan g8; 9 for car_room; 10 for room1: ');
if (~(chan_num_flag == 1 || chan_num_flag == 2 || chan_num_flag == 3 ||
    chan_num_flag == 4 || chan_num_flag == 5 || chan_num_flag == 6 ||
    chan_num_flag == 7 || chan_num_flag == 8 || chan_num_flag == 9 ||
    chan_num_flag ==10))
    error('You were supposed to enter a number from 1 to 10');
end

if (input_flag == 1)
    %%%%%%% Loading the Composite Source Signal (CSS)
    % This signal has been normalized, i.e. it's power equals 1
    if (chan_num_flag == 9 || chan_num_flag == 10)
        load mat_dat_files/css8kHz6.mat;
        x = css8kHz6;
    else
        load mat_dat_files/css8kHz4.mat;
        x = css8kHz4;
    end
else % (input_flag == 2)
    %%%% Loading voice signal
    load mat_dat_files/tape_8kHz_8bits.mat;
    x = tape_8kHz_8bits;
```

```
end
n = length(x);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% LOADING CHANNEL %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%% Echo Path Channels
% Refer to generate_channels.mat for the generation of these channels.
% ERL is set to 6 dB. Channels g1 -> g4 are models from network hybrid
% simulator while channels g5 -> g8 are models from  measured from
% telephone networks in North America.
if (chan_num_flag == 9)
      load mat_dat_files/room_car.mat;
      chan = 0.2*car_room;
elseif (chan_num_flag == 10)
      load mat_dat_files/room1.mat;
      chan = room1;
else
      load mat_dat_files/gecho_paths.mat;
      if (chan_num_flag == 1)
          chan = g1;
      elseif (chan_num_flag == 2)
          chan = g2;
      elseif (chan_num_flag == 3)
          chan = g3;
      elseif (chan_num_flag == 4)
          chan = g4;
      elseif (chan_num_flag == 5)
          chan = g5;
      elseif (chan_num_flag == 6)
          chan = g6;
      elseif (chan_num_flag == 7)
          chan = g7;
      else
          chan = g8;
      end
end

Lf = length(chan);

% SNR and power calculations
x_power = 1;
SNRdb = 30;
SNR = 10^(SNRdb/10);
noise_power = x_power/SNR;

% Parameters
%NOTE: ibits EXCLUDES the sign bit, i.e. total WL = 1 + *_ibits + *
    _fbits
```

```
if (input_flag == 1) % CSS input
    if (chan_num_flag == 10)        % room1
        alpha = 2^-3;
        beta = 0.008;

        M = 64;
        qe_ibits = 1;
        qe_fbits = 6;
        qre_ibits = 8;
        qre_fbits = 0;
        kappa = 2^-13;
    elseif (chan_num_flag == 9)     % car_room
        alpha = 2^-3;
        beta = 0.008;

        M = 128;
        qe_ibits = 1;
        qe_fbits = 6;
        qre_ibits = 7;
        qre_fbits = 0;
        kappa = 2^-14;
    else                        % g.168 hybrids
        alpha = 2^-3
        beta = 0.008;

        M = 32;
        qe_ibits = 1;
        qe_fbits = 6;
        qre_ibits = 7;
        qre_fbits = 0;
        kappa = 2^-13;
    end
else % voice input
    if (chan_num_flag == 10)        % room1
        alpha = 2^-4;
        beta = 0.008;

        M = 32;
        qe_ibits = 1;
        qe_fbits = 6;
        qre_ibits = 8;
        qre_fbits = 0;
        kappa = 2^-13;
    elseif (chan_num_flag == 9)     % car_room
        alpha = 2^-4;
        beta = 0.008;

        M = 32;
        qe_ibits = 1;
```

```
          qe_fbits = 6;
          qre_ibits = 8;
          qre_fbits = 0;
          kappa = 2^-13;
    else                              % g.168 hybrids
          alpha = 2^-4
          beta = 0.008;

          M = 32;
          qe_ibits = 1;
          qe_fbits = 6;
          qre_ibits = 7;
          qre_fbits = 0;
          kappa = 2^-13;
    end
end

Da = 32;

esq_nlms = zeros(1,n);
esq_qsmdnlms = zeros(1,n);
dsq = zeros(1,n);
flag_qsm = zeros(1,n);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%% RESET VECTORS TO ZERO %%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%% NLMS vectors %%%%%%%%%
y_nlms = zeros(1,n);            % output of the filter
error_nlms = zeros(1,n);        % error signal
c_nlms = zeros(1,Lf);         % coefficient INITIALIZATION

%%%%%%% QSM DNLMS vectors %%%%%%%%%
y_qsmdnlms = zeros(1,n);            % output of the filter
error_qsmdnlms = zeros(1,n);        % error signal
c_qsmdnlms = zeros(1,Lf);         % coefficient INITIALIZATION
flag_qsm_temp = zeros(1,n);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%% CREATE DESIRED SIGNAL (ECHO) %%%%%%%%%%%%%%%%%%%%%
Max_x = max(abs(x));
x = x./Max_x;
noise = randn(1,n);
noise = sqrt(noise_power).*(noise./Max_x);
d = filter(chan,1,x) + noise;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%% ADAPTIVE FILTERING %%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[error_nlms, y_nlms, c_nlms] = f_nlms(x, d, alpha, beta, Lf);
[error_qsmdnlms, y_qsmdnlms, c_qsmdnlms, flag_qsm_temp] = f_qsmdnlms(x,
    d, alpha, beta, Lf, Da, M, kappa, qe_ibits, qe_fbits, qre_ibits,
    qre_fbits);

esq_nlms = esq_nlms + error_nlms.^2;
esq_qsmdnlms = esq_qsmdnlms + error_qsmdnlms.^2;
flag_qsm= flag_qsm + flag_qsm_temp;
dsq = dsq + (d.^2);

iter = 1:n;
iterms = iter./8;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%% FILTER PERFORMANCE %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
block = 256;
for (kk = block:n)
      dsqav(kk) = sum(dsq(kk-(block-1):kk))/(block);
      esqav_nlms(kk) = sum(esq_nlms(kk-(block-1):kk))/(block);
      esqav_qsmdnlms(kk) = sum(esq_qsmdnlms(kk-(block-1):kk))/(block);

      erle_nlms(kk) = dsqav(kk)./esqav_nlms(kk);
      erle_qsmdnlms(kk) = dsqav(kk)./esqav_qsmdnlms(kk);
end

erle_nlms = 10*log10(erle_nlms);
erle_qsmdnlms = 10*log10(erle_qsmdnlms);

% Calculation of % updates
percent_update_qsm = sum(flag_qsm)/n;
percent_update_voice = 0;
percent_update_pseudo = 0;
percent_update_pause = 0;
if (chan_num_flag == 9 || chan_num_flag == 10)
      css_lim = 6-1;
else
      css_lim = 4-1;
end
for u=0:css_lim    %for css8kHz6
      percent_update_voice = percent_update_voice + sum(flag_qsm(1+2803*
          u:388+2803*u));
      percent_update_pseudo = percent_update_pseudo + sum(flag_qsm
          (389+2803*u:1994+2803*u));
      percent_update_pause = percent_update_pause + sum(flag_qsm
          (1995+2803*u:2803+2803*u));
end
```

```
percent_update_voice = percent_update_voice/css_lim/388;
percent_update_pseudo = percent_update_pseudo/css_lim/1603;
percent_update_pause = percent_update_pause/css_lim/812;

display('Percent_updates')
percent_update_qsm
percent_update_voice
percent_update_pseudo
percent_update_pause

% plots
figure
subplot(2,1,1)
plot(iter,d,'k');
hold on;
plot(iter,error_qsmdnlms,'b');
plot(iter,error_nlms,'r');
xlabel('Samples');
ylabel('Redisual_Echo');

subplot(2,1,2)
hold on;
plot(iter, erle_nlms,'r');
plot(iter, erle_qsmdnlms,'b');
xlabel('Samples');
ylabel('ERLE_(dB)');
legend('NLMS', 'Proposed');
```

dnlms_hybrid_fix.m

The following source code describes the proposed architecture of Chapter 4 using fixed-point wordlengths and arithmetic. This source code generates the results seen in Section 4.2.1

```
% This program simulates echo cancellation with DNLMS using fixed-
% point arithmetic. The channels used here are the ones from G.168.
% The test signals include white gaussian noise and a normalized CSS
% from G.168 sampled at 8 kHz. We will assume an 8 kHz sampling
% frequency. Here, the data flow is manipulated to mimic the data
% flow in the hybrid architecture.
%
% Last modified Nov 26, 2006
% Raymond Lee

clear all
close all
j = sqrt(-1);

%%%%%%%% LIST OF VARIABLES %%%%%%%%%%%%%%%%%%%%
% N - filter length
% n - # of samples
% mu - adaptation step-size
% num_realizations - number of realizations over which the
%       squared error is averaged
% d - genuine echo that the filter is trying to match (desired signal)
% y - filter output or echo replica
% error - error signal = d - y
% mse - mean square error
% erle - echo return loss enhancement
% chan - echo path impulse response
% w - weights of the adaptive filter
% regg - regressor which holds current input and N-1 past input signals
% x - the input to the adaptive filter (Far-end speaker's signal). x is
%       either wgn or CSS.
% noise - Near-end speaker's signal (part of the desired signal)
% alpha - parameter for NLMS - (0,2]
% beta - parameter for NLMS - small and positive
% SNR - signal to noise ratio = x_power/noise_power
% SNRdb - 10log10(SNR)
% D - amount by which DNLMS adaptation is delayed
% P - number of taps per module
% flag - vector to keep track up updates or choose options.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%% PARAMETERS and INITIALIZATION %%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Prompt user
input_flag = input('Enter 0 for WGN input, 1 for CSS input: ');
if (~(input_flag == 0 || input_flag == 1))
    error('You were supposed to enter 0 or 1');
end
chan_num_flag = 3;

%-------------------- Fixed-point parameters --------------------
t1 = numerictype('WordLength',8,'Fractionlength',7,'Signed',1);
t3 = numerictype('WordLength',8,'Fractionlength',6,'Signed',1);
t4 = numerictype('WordLength',16,'Fractionlength',15,'Signed',1);
t5 = numerictype('WordLength',11,'Fractionlength',7,'Signed',1);
t6 = numerictype('WordLength',12,'Fractionlength',7,'Signed',1);
t7 = numerictype('WordLength',13,'Fractionlength',12,'Signed',1);
```

```
t8 = numerictype('WordLength',18,'Fractionlength',17,'Signed',1);

t_c_hybdnlms = t8;
t_d = t1;
t_error_hybdnlms = t1;
t_pf = t4;
t_pw = t8;
t_mu_e = t7;
t_x = t1;
t_y_hybdnlms = t1;
t_mu_hybdnlms = t5;
t_regg_energy = t6;
t_xsq = t1;
t_alpha =t3;

% c_hybdnlms = c_hybdnlms + pw
% pf0 = c_hybdlms(ff)*regg(ff)
F1=fimath;
F1.CastBeforeSum = false;
F1.SumMode = 'SpecifyPrecision';
F1.SumWordLength = t_c_hybdnlms.wordlength;
F1.SumFractionLength = t_c_hybdnlms.fractionlength;
F1.ProductMode = 'SpecifyPrecision';
F1.ProductWordLength = t_pf.wordlength;
F1.ProductFractionLength = t_pf.fractionlength;
F1.RoundMode = 'floor';
F1.OverflowMode = 'saturate';

% error_hybdnlms(i) = d(i) - y_hybdnlms(i);
% mu_e = mu_hybdnlms(i-error_delay) * error_hybdnlms(i-error_delay)
F2=F1;
F2.SumWordLength = t_error_hybdnlms.wordlength;
F2.SumFractionLength = t_error_hybdnlms.fractionlength;
F2.ProductWordLength = t_mu_e.wordlength;
F2.ProductFractionLength = t_mu_e.fractionlength;

% regg_energy = regg_energy + x0sq - xLsq
% regg_energy = regg_energy + x0sq;
% x0sq = x(i-error_delay)*x(i-error_delay)
% xNsq = last_regg_delayed*last_regg_delayed
F3=F1;
F3.SumWordLength = t_regg_energy.wordlength;
F3.SumFractionLength = t_regg_energy.fractionlength;
F3.ProductWordLength = t_xsq.wordlength;
F3.ProductFractionLength = t_xsq.fractionlength;

% pw = regg_delayed.* mue_regg_delayed
% y_addertemp = y_adderline(ff/2) + pf0 + pf1;
F4=F1;
```

```
F4.SumWordLength = t_pf.wordlength;
F4.SumFractionLength = t_pf.fractionlength;
F4.ProductWordLength = t_c_hybdnlms.wordlength;
F4.ProductFractionLength = t_c_hybdnlms.fractionlength;

%--------------------------------------------------------------------

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% LOADING INPUT AND CHANNEL %%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%% Echo Path Channels
% Refer to generate_channels.mat for the generation of these channels.
% ERL is set to 6 dB. Channels g1 -> g4 are models from network hybrid
% simulator while channels g5 -> g8 are models from  measured from
% telephone networks in North America.
%%%%%%% Input x and desired d
% The input signal has been normalized, i.e. it's power equals 1. The
% desired signal was created by passing the input x through the channel.
% White gaussian noise was added the d such that SNR = 30 dB.

if (input_flag == 1)
    %%%%%%% Loading the Composite Source Signal (CSS)
    if (chan_num_flag == 1)
        load mat_dat_files/x_d_css_g1.mat;
    elseif (chan_num_flag == 2)
        load mat_dat_files/x_d_css_g2.mat;
    elseif (chan_num_flag == 3)
        load mat_dat_files/x_d_css_g3.mat;
    elseif (chan_num_flag == 4)
        load mat_dat_files/x_d_css_g4.mat;
    elseif (chan_num_flag == 5)
        load mat_dat_files/x_d_css_g5.mat;
    elseif (chan_num_flag == 6)
        load mat_dat_files/x_d_css_g6.mat;
    elseif (chan_num_flag == 7)
        load mat_dat_files/x_d_css_g7.mat;
    else
        load mat_dat_files/x_d_css_g8.mat;
    end
else
    %%%%%%% Loading the White Gaussian Noise (WGN)
    if (chan_num_flag == 1)
        load mat_dat_files/x_d_wgn_g1.mat;
    elseif (chan_num_flag == 2)
        load mat_dat_files/x_d_wgn_g2.mat;
    elseif (chan_num_flag == 3)
        load mat_dat_files/x_d_wgn_g3.mat;
    elseif (chan_num_flag == 4)
        load mat_dat_files/x_d_wgn_g4.mat;
    elseif (chan_num_flag == 5)
        load mat_dat_files/x_d_wgn_g5.mat;
    elseif (chan_num_flag == 6)
        load mat_dat_files/x_d_wgn_g6.mat;
    elseif (chan_num_flag == 7)
        load mat_dat_files/x_d_wgn_g7.mat;
    else
        load mat_dat_files/x_d_wgn_g8.mat;
    end
end
n = length(x);
x = fi(x ,t_x ,F3);
d = fi(d ,t_d ,F2);

N = length(chan);

% Parameters
P = 3; % the number of taps per module
if (input_flag == 0)
    alpha0 = 2^-1;
    alpha = fi(alpha0 ,t_alpha ,F2);
    % WGN alpha - parameter for NLMS - (0,2); fi properties are
        arbitraty ,
    % but we'll set it to F2, same as mu_hybdnlms
else
    alpha0 = 2^-3;
    alpha = fi(alpha0 ,t_alpha ,F2); % CSS alpha - parameter for NLMS -
        (0,2)
end
beta0 = 2^-4;
beta = fi(beta0 ,t_regg_energy ,F3);
% beta - small and positive; same fi properties as regg_energy
D = floor(N/P); % D >= N/P    % ASSUME N is divisible by P
error_delay = D-N/P+1;
num_realizations = 1;

esq_hybdnlms = zeros(1,n);
dsq = zeros(1,n);

for nr = 1: num_realizations
    nr
    tic

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%% RESET VECTORS TO ZERO %%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    regg = fi(zeros(1, N), t_x ,F1);
    regg_delayed = fi(zeros(1, N), t_x ,F4);
    mue_regg_delayed = fi(zeros(1, N), t_mu_e ,F4);
```

```
y_hybdnlms = fi(zeros(1,n), t_y_hybdnlms,F2);
error_hybdnlms = fi(zeros(1,n), t_error_hybdnlms,F2);
c_hybdnlms = fi(zeros(1,N), t_c_hybdnlms ,F1);
mu_hybdnlms = fi(alpha0/beta0, t_mu_hybdnlms ,F2);
y_adderline = fi(zeros(1, N/P), t_pf,F4);
y_hybdnlms = fi(0, t_y_hybdnlms ,F2);
regg_energy = fi(beta, t_regg_energy ,F3);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%% ADAPTIVE FILTERING %%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for i = 1 : n
    % Update the regressor
  newvalue = x(i);
  for ii = 1:N
        if (ii>1 && mod(ii-1,P)==0)
              regg(ii) = regg(ii-1);
        else
              nextnewvalue = regg(ii);
              regg(ii) = newvalue;
              newvalue = nextnewvalue;
        end
  end
    regg = fi(regg, t_x,F1);

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%% DNLMS %%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % use 2 taps at a time and an adder-delay line to
    % find the output y(n-1)
    y_addertemplast = fi(0, t_pf,F4);
    for ff = N:-P:1
        pf0 = fi(c_hybdnlms(ff)*regg(ff), t_pf ,F4);
        pf1 = fi(c_hybdnlms(ff-1)*regg(ff-1), t_pf , F4);
        pf2 = fi(c_hybdnlms(ff-2)*regg(ff-2), t_pf , F4);

        y_addertempnext = fi(y_adderline(ff/P) + pf0 + pf1 + pf2,
            t_pf,F4);
        y_adderline(ff/P) = fi(y_addertemplast, t_pf,F4);
        y_addertemplast = fi(y_addertempnext, t_pf,F4);
    end
    y_hybdnlms(i) = fi(y_addertempnext, t_y_hybdnlms,F2);
    error_hybdnlms(i) = fi(d(i) - y_hybdnlms(i), t_error_hybdnlms,F2
        );

    % Update the delayed regressor
  if ((i-D)>0)
        newvalue = x(i-D);
        for ii = 1:N
```

```
                    if (ii>1 && mod(ii-1,P)==0)
                        regg_delayed(ii) = regg_delayed(ii-1);
                    else
                        nextnewvalue = regg_delayed(ii);
                        regg_delayed(ii) = newvalue;
                        newvalue = nextnewvalue;
                    end
                end
                regg_delayed = fi(regg_delayed, t_x,F4);
            end

            if ((i-(error_delay-1))>0)
                x0sq = fi(double(x(i-(error_delay-1)))*double(x(i-(
                    error_delay-1))), t_xsq,F3);
                last_regg_delayed = double(regg_delayed(N));
                xNsq = fi(last_regg_delayed*last_regg_delayed, t_xsq,F3);
                regg_energyd = fi(regg_energy, t_regg_energy,F3);
                regg_energytemp = fi(regg_energyd - xNsq, t_regg_energy,F3);
                regg_energy = fi(regg_energytemp + x0sq, t_regg_energy,F3);
            end

            % create the delay line for mu*e
            if ((i-error_delay)>0)
                mu_hybdnlms1 = fi(mu_hybdnlms, t_mu_hybdnlms,F2);
                mu_hybdnlms = t_mu_hybdnlms.divide(alpha, regg_energy);
                mu_e = fi(mu_hybdnlms1 * error_hybdnlms(i-error_delay),
                    t_mu_e,F1);
                mue_regg_delayed = fi([mue_regg_delayed(4:N) mu_e mu_e mu_e
                    ], t_mu_e,F4);

                % update the coefficients
                pw = fi(regg_delayed.*mue_regg_delayed, t_pw, F1);
                c_hybdnlms = fi(c_hybdnlms + pw, t_c_hybdnlms,F1); %update
                    equation

           end
        end

        esq_hybdnlms = esq_hybdnlms + double(error_hybdnlms).^2;
        dsq = dsq + (double(d).^2);
        toc

end

iter = 1:n;
iterms = iter./8;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%% EQUALIZER PERFORMANCE %%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if (input_flag == 0)
        % Calculate MSE
        mse_hybdnlms = esq_hybdnlms./num_realizations;
        mse_hybdnlms = 10*log10(mse_hybdnlms);
else
        block = 256;
        for (kk = block:n)
                dsqav(kk) = sum(dsq(kk-(block-1):kk))/(block);
                esqav_hybdnlms(kk) = sum(esq_hybdnlms(kk-(block-1):kk))/(
                    block);
                erle_hybdnlms(kk) = dsqav(kk)./esqav_hybdnlms(kk);
        end
        erle_hybdnlms = 10*log10(erle_hybdnlms);
end

% Plots
if (input_flag == 0)
        figure
        plot(iter, mse_hybdnlms(1:length(mse_hybdnlms)),'b');
        title(['Mean Squared Error ({\alpha} = ' num2str(double(alpha)),',
            {\beta} = ', num2str(double(beta)),', and SNR = ' num2str(
            SNRdb),' dB')]);
        legend ('DNLMS');
        xlabel('Samples');
        ylabel('MSE (dB)');
else
        figure
        plot(iter,d,'k');
        hold on;
        plot(iter,error_hybdnlms,'b');
        xlabel('Samples');
        ylabel('Redisual Echo');

        figure
        plot(iter, erle_hybdnlms,'b');
        xlabel('Samples');
        ylabel('ERLE (dB)');
        legend('DNLMS');
end
```

Appendix B

Verilog Source Code

This Appendix gives the Verilog source code used for the DNLMS adaptive filter in Chapter 4. The code is written behaviourally, parametzable, and independent of vendor technology. Figure B.1 shows the hierarchy of the Verilog modules.

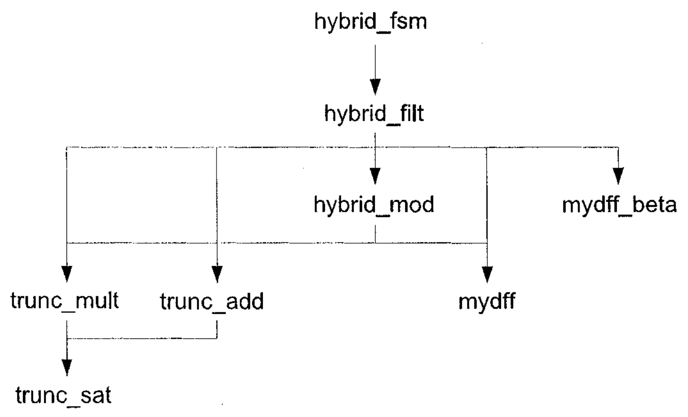


Figure B.1: Hierarchy of Verilog modules

hybrid_fsm.v

The following source code describes the FSM applied to the DUT.

```
/*************************************
This module is intended to act as a finite state machine for
hybrid_filter.v. It uses hybrid_filter.v, hybrid_mod.v, trunc_ams2.v,
and mydff.v

When START is high, the machine exits out of the IDLE state and
        1. Resets all dffs.
        2. Loads parameter beta and initial coefficients from memory.
        3. Applies the input vector to the filter and saves the output.
        4. Returns to the IDLE state when complete.

RESET_FSM_low is active low and initializes the FSM to the IDLE state

Last modified: Nov 26, 2006

By Raymond Lee
***************************************/

module hybrid_fsm(
        clk,            // clk is the clock
        RESET_FSM_low,      // reset for FSM. Changes state to IDLE
        START,              // Control signal to start process
        xin,            // Value read from xROM. Input to filter
        din,            // Value read from dROM. Input to filter
        u1,             // u1 is the delayed step-size which comes from an LUT
        regg_energy,        // regg_energy is the delayed reggressor energy
           and address to the LUT
        x_en,               // Enable signal for xROM
        x_addr,         // Address for xROM
        d_en,               // Enable signal for dROM
        d_addr,         // Address for dROM
        yout,               // Output of filter written to yRAM
        y_wren,             // Enable signal for yRAM
        y_addr,         // Address for yRAM
        e,              // Output of filter written to eRAM
        e_wren,         // Enable signal for eRAM
        e_addr              // Address for eRAM
        );

parameter
        num_inputs = 11212, // CSS input

        N = 96,  // NOTE: N is assumed to by divisible by P
        P = 3,
        D = 32,       // NOTE: D > N/P-1
        NdivP = D, // = N/P

        // *_ibits includes the sign bit
        x_ibits = 0+1,           x_fbits = 7,
        d_ibits = 0+1,           d_fbits = 7,
        regg_energy_ibits = 4+1,regg_energy_fbits = 7,
        w_ibits = 0+1,           w_fbits = 17,
        y_ibits = 0+1,           y_fbits = 7,
        e_ibits = 0+1,           e_fbits = 7,
        u_ibits = 3+1,           u_fbits = 7,
        ue_ibits = 0+1,          ue_fbits = 12,
        xsq_ibits = 0+1,  xsq_fbits = 7,
        a_ibits = 0+1,           a_fbits = 15,

        x_bits = x_ibits+x_fbits ,
        d_bits = d_ibits+d_fbits ,
        regg_energy_bits = regg_energy_ibits+regg_energy_fbits ,
        w_bits = w_ibits+w_fbits ,
        y_bits = y_ibits+y_fbits ,
        e_bits = e_ibits+e_fbits ,
        u_bits = u_ibits+u_fbits ,
        ue_bits = ue_ibits+ue_fbits ,
        xsq_bits = xsq_ibits+xsq_fbits ,
        a_bits = a_ibits+a_fbits ,

        x_maxbits = (D+N-NdivP+1)*x_bits ,
        ue_maxbits = NdivP*ue_bits ,
        a_maxbits = NdivP*a_bits ,
        ed_maxbits = (D-NdivP+1+1)*e_bits ,
        w_maxbits = N*w_bits ,

        x_addrbits = 14-1, // for 11212inputs
        d_addrbits = x_addrbits ,
        y_addrbits = x_addrbits ,
        e_addrbits = x_addrbits
        ;

input clk , RESET_FSM_low, START;
input [x_bits -1:0] xin;
input [d_bits -1:0] din;
input signed [u_bits -1:0] u1;

output x_en , d_en , y_wren , e_wren;
output [x_addrbits :0] x_addr;
output [d_addrbits :0] d_addr;
output [y_addrbits :0] y_addr;
output [e_addrbits :0] e_addr;
output [y_bits -1:0] yout;
output [e_bits -1:0] e;
output signed [regg_energy_bits -1:0] regg_energy;

reg reset, y_wren, e_wren;
reg [x_addrbits:0] x_addr;
reg [d_addrbits:0] d_addr;
reg [y_addrbits:0] y_addr;
reg [e_addrbits:0] e_addr;
reg current_state;
wire[regg_energy_bits -1:0] betain = 12'b000000001000;

hybrid_filt #(
        .N(N),
        .P(P),
        .D(D),
        .NdivP(NdivP),
        .x_ibits(x_ibits), .x_fbits(x_fbits),
        .d_ibits(d_ibits), .d_fbits(d_fbits),
        .regg_energy_ibits(regg_energy_ibits),
        .regg_energy_fbits(regg_energy_fbits),
        .w_ibits(w_ibits), .w_fbits(w_fbits),
        .y_ibits(y_ibits), .y_fbits(y_fbits),
        .e_ibits(e_ibits), .e_fbits(e_fbits),
        .u_ibits(u_ibits), .u_fbits(u_fbits),
        .ue_ibits(ue_ibits), .ue_fbits(ue_fbits),
        .xsq_ibits(xsq_ibits), .xsq_fbits(xsq_fbits),
        .a_ibits(a_ibits), .a_fbits(a_fbits)
        )
        Filt1 (
                clk,
                reset,
                xin,
                din,
                betain,
                ul,
                regg_energy,
                yout,
                e
                );

// Declaring symbolic names for states
parameter IDLE = 1'b0, FILT = 1'b1;

always @ (posedge clk)
begin
        if (RESET_FSM_low == 0)
                current_state <= IDLE;
        else
        begin
                // Next State logic
                case (current_state)
                        IDLE:
                                if (START == 1)
                                        current_state <= FILT;
                                else
                                        current_state <= IDLE;
                        FILT:
                                if (x_addr >= num_inputs)
                                        current_state <= IDLE;
                                else
                                        current_state <= FILT;
                        default:
                                current_state <= IDLE;
                endcase

        // Output Logic (Delayed Control Signals)
        reset <= ~(current_state);
        y_wren <= current_state;
        e_wren <= current_state;

        // Output Logic (Memory Addresses)
        if (x_en == 1) // Increment address
                x_addr <= x_addr + 1;
        else
                x_addr <= 0;
        if (d_en == 1) // Increment address
                d_addr <= d_addr + 1;
        else
                d_addr <= 0;

        if (y_wren == 1) // Increment address
                y_addr <= y_addr + 1;
        else
                y_addr <= 0;

        if (e_wren == 1) // Increment address
                e_addr <= e_addr + 1;
        else
                e_addr <= 0;

        end

end

// Output Logic   (Control signals)
assign x_en = current_state;
assign d_en = current_state;
```

```
endmodule
```

hybrid_filter.v

The following source code describes the proposed architecture seen in Fig. 4.3.

```
/*************************************
This module uses hybrid_mod.v, trunc_ams.v, and mydff.v to make an
adaptive filter DNLMS FIR filter.

Last modified: Nov 26, 2006

By Raymond Lee
***************************************/

module hybrid_filt(
        clk,    // clk is the clock
        reset,          // reset is the synchronous reset signal
        xin0,   // xin0 is the input to the filter
        d,      // d is the desired signal
        beta,   // beta is a parameter of the DNLMS alg equal to a small
            value
        u1,     // u1 is the delayed step-size which comes from an LUT
        regg_energy,        // regg_energy is the delayed reggressor energy
            and address to the LUT
        yout,   // yout is the output of the filter
        e       // e is the error signal
        );

parameter
        N = 96,        // N is the length of the filter (NOTE: N is assumed
            to be divisible by P)
        P = 3, // P is the number of taps per PE
        D = 32,        // D is the adaptation delay (NOTE: D > N/P-1)
        NdivP = D, // = N/P

        // *_ibits includes the sign bit
        x_ibits = 0+1,          x_fbits = 7,
        d_ibits = 0+1,          d_fbits = 7,
        regg_energy_ibits = 4+1,regg_energy_fbits = 7,
        w_ibits = 0+1,          w_fbits = 17,
        y_ibits = 0+1,          y_fbits = 7,
        e_ibits = 0+1,          e_fbits = 7,
        u_ibits = 3+1,          u_fbits = 7,
        ue_ibits = 0+1,          ue_fbits = 12,
        xsq_ibits = 0+1,   xsq_fbits = 7,
        a_ibits = 0+1,           a_fbits = 15,

        x_bits = x_ibits+x_fbits ,
        d_bits = d_ibits+d_fbits ,
        regg_energy_bits = regg_energy_ibits+regg_energy_fbits ,
        w_bits = w_ibits+w_fbits ,
        y_bits = y_ibits+y_fbits ,
        e_bits = e_ibits+e_fbits ,
        u_bits = u_ibits+u_fbits ,
        ue_bits = ue_ibits+ue_fbits ,
        xsq_bits = xsq_ibits+xsq_fbits ,
        a_bits = a_ibits+a_fbits ,

        x_maxbits = (D+N-NdivP+1)*x_bits ,
        ue_maxbits = NdivP*ue_bits ,
        a_maxbits = NdivP*a_bits ,
        ed_maxbits = (D-NdivP+1+1)*e_bits ,
        w_maxbits = N*w_bits
        ;

input clk;
input reset;
input signed [x_bits -1:0] xin0;
input signed [d_bits -1:0] d;
input signed [regg_energy_bits -1:0] beta;
input signed [u_bits -1:0] u1;

output signed [regg_energy_bits -1:0] regg_energy;
output signed [y_bits -1:0] yout;
output signed [e_bits -1:0] e;

wire signed [x_maxbits -1:0] x; // regressor containing values x(n), x(n
    -1), ... , x(n-D-N+N/P)
wire signed [d_bits -1:0] d1; // the inputs, xin0 and d, are buffered, i.
    e. they are delayed by 1
wire signed [y_bits -1:0] yout1, youtneg;
// yout1 is the output yout is buffered, i.e. they are delayed by 1
// youtneg = -yout
wire signed [xsq_bits -1:0] xsq0, xsqN;       // x(n-D+N/P)^2 and x(n-D-N+N/
    P)^2 respectively
wire signed [regg_energy_bits -1:0] regg_energyd, regg_energy_temp;
// regg_energy = regg_energy_temp + x(n-D+N/P)^2
// regg_energy_temp = regg_energyd + x(n-D-N+N/P)^2
// regg_energyd <= regg_energy
wire signed  [ue_maxbits -1:0] ue; // is a delay-line containing u(n-D+N/
    P-1)e(n-D+N/P-1), ..., u(n-D)e(n-D)
```

```
wire signed  [a_maxbits-1:0] ain, aout; // is a collection of the inputs
    and outputs for the serial addition of each PE. The input of PE i
    is the delayed output of PE i+1
wire signed  [ed_maxbits-1:0] ed; // is a delay-line containing e(n),
    ..., e(n-D+N/P)
reg signed [u_bits-1:0] u1muxout; // = 0 on reset. else = u1

assign ain[a_bits-1:0] =0;      // the input to PE N/P-1 is set to zero
assign yout = $signed(aout[a_maxbits-1:a_maxbits-y_bits]);
// the filter output yout is the serial addition output of PE 0,
// truncated to yout's wordlength
assign ed[ed_maxbits-1:ed_maxbits-e_bits] = e;
assign youtneg = -yout;

// e = d1 - yout
trunc_add #(
      .a_ibits(y_ibits), .a_fbits(y_fbits),
      .b_ibits(d_ibits), .b_fbits(d_fbits), // the # of 'a' and 'b'
         fractional bits need to be the same.
      .c_ibits(e_ibits), .c_fbits(e_fbits)
      )
       Add_Error (
            youtneg,
            d1,    // d1 is just the buffered input. It has zeros padded
                to the end of the word because, for the addition to
                occur, the number of fractional bits of both operands
                need to be the same.
            e
            );

// xsq0 = x(n-D+N/P)*x(n-D+N/P) = x(n)*x(n) iff D=N/P
trunc_mult #(
      .a_ibits(x_ibits), .a_fbits(x_fbits),
      .b_ibits(x_ibits), .b_fbits(x_fbits),
      .c_ibits(xsq_ibits), .c_fbits(xsq_fbits)
      )
      Mult_Xsq0 (
            x[x_maxbits-1-(D-NdivP)*x_bits : x_maxbits-(D-NdivP+1)*
                x_bits],
            x[x_maxbits-1-(D-NdivP)*x_bits : x_maxbits-(D-NdivP+1)*
                x_bits],
            xsq0
            );

// xsqN = x(n-D-N+N/P)*x(n-D-N+N/P)
trunc_mult #(
      .a_ibits(x_ibits), .a_fbits(x_fbits),
      .b_ibits(x_ibits), .b_fbits(x_fbits),
      .c_ibits(xsq_ibits), .c_fbits(xsq_fbits)
```

```
        )
        Mult_XsqN (
              x[x_bits -1:0],
              x[x_bits -1:0],
              xsqN
              );

// regg_energy = regg_energy_temp + x(n-D+N/P)^2
trunc_add #(
        .a_ibits(regg_energy_ibits), .a_fbits(regg_energy_fbits),
        .b_ibits(xsq_ibits), .b_fbits(xsq_fbits),
        .c_ibits(regg_energy_ibits), .c_fbits(regg_energy_fbits)
        )
        Add_ReggEnergy (
              regg_energy_temp,
              xsq0,
              regg_energy
              );

// regg_energy_temp = regg_energyd - x(n-D-N+N/P)^2
trunc_add #(
        .a_ibits(regg_energy_ibits), .a_fbits(regg_energy_fbits),
        .b_ibits(xsq_ibits), .b_fbits(xsq_fbits),
        .c_ibits(regg_energy_ibits), .c_fbits(regg_energy_fbits)
        )
        Add_ReggEnergyTemp (
              regg_energyd,
              -xsqN,
              regg_energy_temp
              );

// regg_energyd <= regg_energy
mydff_beta #(.dff_bits(regg_energy_bits))
        Dff_ReggEnergyd (
              clk,
              reset,
              regg_energy,
              beta,
              regg_energyd
              );

// ue_{mod(N/P-1)} = u(n-D+N/P-1)*e(n-D+N/P-1)
trunc_mult #(
        .a_ibits(u_ibits), .a_fbits(u_fbits),
        .b_ibits(e_ibits), .b_fbits(e_fbits),
        .c_ibits(ue_ibits), .c_fbits(ue_fbits)
        )
        Mult_UE (
              u1muxout,
              ed[e_bits -1:0],
              ue[ue_bits -1:0]
              );

// Buffering the inputs, xin0 and d, is necessary for the FSM
mydff #(.dff_bits(x_bits))
        Dff_BuffX (
              clk,
              reset,
              xin0,
              x[x_maxbits-1:x_maxbits-x_bits]
              );

mydff #(.dff_bits(d_bits))
        Dff_BuffD (
              clk,
              reset,
              d,
              d1
              );

// Buffer the output yout too.
mydff #(.dff_bits(y_bits))
        Dff_BuffY (
              clk,
              reset,
              yout,
              yout1
              );

genvar ii;
generate
        for (ii=0; ii<=D+N-NdivP-1; ii=ii+1)
        begin : reggressor
              // tap-delay line containing x(n), x(n-1), ..., x(n-D-N+N/P)
              mydff #(.dff_bits(x_bits))
                    Dff_Regressor (
                          clk,
                          reset,
                          x[x_maxbits-1-ii*x_bits : x_maxbits-(ii+1)*
                             x_bits],
                          x[x_maxbits-1-(ii+1)*x_bits : x_maxbits-(ii+2)*
                             x_bits]
                          );
        end
endgenerate

generate
        for (ii=0; ii<=NdivP-1; ii=ii+1)
```

```
        begin : hybridmodules
              hybrid_mod #(
                    .x_in_ibits(x_ibits), .x_in_fbits(x_fbits),
                    .w_ibits(w_ibits), .w_fbits(w_fbits),
                    .ue_ibits(ue_ibits), .ue_fbits(ue_fbits),
                    .a_ibits(a_ibits), .a_fbits(a_fbits)
                    )
                    HybridMod (
                          clk,
                          reset,
                          x[x_maxbits-1-ii*(P-1)*x_bits : x_maxbits-ii*(P
                              -1) *x_bits-P*x_bits],
                          x[x_maxbits-1-D*x_bits-ii*(P-1)*x_bits :
                              x_maxbits-D*x_bits-ii*(P-1)*x_bits-P*x_bits
                              ],
                          ain[a_maxbits-1-ii*a_bits : a_maxbits-(ii+1)*
                              a_bits],
                          ue[ue_maxbits-1-ii*ue_bits : ue_maxbits-(ii+1)*
                              ue_bits],
                          aout[a_maxbits-1-ii*a_bits : a_maxbits-(ii+1)*
                              a_bits]
                          );
        end
endgenerate

generate
        for (ii=1; ii<=NdivP-1; ii=ii+1)
        begin : serialadder
              // ain_{ii-1}(n) = aout_{ii}(n-1)
              mydff #(.dff_bits(a_bits))
                    Dff_SerialAdd (
                          clk,
                          reset,
                          aout[a_maxbits-1-ii*a_bits : a_maxbits-(ii+1)*
                              a_bits],
                          ain[a_maxbits-1-(ii-1)*a_bits : a_maxbits-ii*
                              a_bits]
                          );
        end
endgenerate

generate
        for (ii=1; ii<=NdivP-1; ii=ii+1)
        begin : ue_delay_line
              // ue_{ii-1}(n) = ue_{ii}(n-1)
              mydff #(.dff_bits(ue_bits))
                    Dff_UE (
                          clk,
                          reset,
                        ue[ue_maxbits-1-ii*ue_bits : ue_maxbits-(ii+1)*
                           ue_bits],
                        ue[ue_maxbits-1-(ii-1)*ue_bits : ue_maxbits-ii*
                           ue_bits]
                        );
      end
endgenerate

generate
      for (ii=0; ii<=D-NdivP; ii=ii+1)
      begin : error_delay_line
            mydff #(.dff_bits(e_bits))
                  Dff_Error (
                        clk,
                        reset,
                        ed[ed_maxbits-1-ii*e_bits : ed_maxbits-(ii+1)*
                           e_bits],
                        ed[ed_maxbits-1-(ii+1)*e_bits : ed_maxbits-(ii
                           +2)*e_bits]
                        );
      end
endgenerate

always @(reset or u1)
begin
      if (reset == 1)
            u1muxout=0;
      else
            u1muxout=u1;
end

endmodule
```

hybrid_mod.v

The following source code describes the PE used in the proposed architecture seen in Fig. 4.4.

```
/*************************************
This module is the PE containing P taps of the hybrid adaptive FIR
form filter.

Last modified: Nov 25, 2006
```

```
By Raymond Lee
****************************************/

module hybrid_mod(
      clk,  // clk is the clock
      reset,       // reset is the synchronous reset signal
      x_in,        // x_in is a collection of P values of the regressor.
      xd_in,       // xd_in is a collection of P values of the delayed
         regressor.
      a_in,        // a_in is the input of the serial adder.
      ue_in,       // ue_in is an input u(n-D+i)*e(n-D+i).
      a_out        // a_out is the output of the serial adder.
      );

parameter
      P = 3,
      x_in_ibits = 0+1, x_in_fbits = 7,
      w_ibits = 0+1,          w_fbits = 7,
      a_ibits = 0+1,          a_fbits = 15,
      ue_ibits = 0+1,         ue_fbits = 12,

      a_bits = a_ibits + a_fbits,
      x_in_bits = x_in_ibits+x_in_fbits,
      w_bits = w_ibits+w_fbits,
      ue_bits = ue_ibits+ue_fbits,

      x_maxbits = P*x_in_bits,
      a_maxbits = (P+1)*a_bits,
      pf_maxbits = P*a_bits,
      w_maxbits = P*w_bits
      ;

input clk;
input reset;
input signed [x_maxbits-1:0] x_in, xd_in;
input signed [a_bits-1:0] a_in;
input signed [ue_bits-1:0] ue_in;

output signed [a_bits-1:0] a_out;

wire signed [w_maxbits-1:0] w, w_next; //w and w_next are a set of P
    weights for this iteration and the next iteration respectively.
wire signed [pf_maxbits-1:0] pf; //product x_in.*w for filtering
wire signed [a_maxbits-1:0] a; //intermediate values of the serial
    addition
wire signed [w_maxbits-1:0] pw;      //product xd_in*mue_in for weight-
    update

assign a_out = a[a_maxbits-1:a_maxbits-a_bits];
assign a[a_bits -1:0] = a_in;

genvar ii;
generate
      for (ii=0; ii<=P-1; ii=ii+1)
      begin : hybridmod
            // pf = x_in.*w
            trunc_mult #(
                  .a_ibits(x_in_ibits), .a_fbits(x_in_fbits),
                  .b_ibits(w_ibits), .b_fbits(w_fbits),
                  .c_ibits(a_ibits), .c_fbits(a_fbits)
                  )
                  Mult_PF (
                        x_in[x_maxbits-1-x_in_bits*ii : x_maxbits-
                            x_in_bits*(ii+1)],
                        w[w_maxbits-1-w_bits*ii : w_maxbits-w_bits*(ii
                            +1)],
                        pf[pf_maxbits-1-a_bits*ii : pf_maxbits-a_bits*(
                            ii+1)]
                        );

            // pw = xd_in * ue_in
            trunc_mult #(
                  .a_ibits(x_in_ibits), .a_fbits(x_in_fbits),
                  .b_ibits(ue_ibits), .b_fbits(ue_fbits),
                  .c_ibits(w_ibits), .c_fbits(w_fbits)
                  )
                  Mult_PW (
                        xd_in[x_maxbits-1-x_in_bits*ii : x_maxbits-
                            x_in_bits*(ii+1)],
                        ue_in,
                        pw[w_maxbits-1-w_bits*ii : w_maxbits-w_bits*(ii
                            +1)]
                        );

            //w_next = w + pw
            trunc_add #(
                  .a_ibits(w_ibits), .a_fbits(w_fbits),
                  .b_ibits(w_ibits), .b_fbits(w_fbits),
                  .c_ibits(w_ibits), .c_fbits(w_fbits)
                  )
                  Add_W (
                        w[w_maxbits-1-w_bits*ii : w_maxbits-w_bits*(ii
                            +1)],
                        pw[w_maxbits-1-w_bits*ii : w_maxbits-w_bits*(ii
                            +1)],
                        w_next[w_maxbits-1-w_bits*ii : w_maxbits-w_bits
                            *(ii+1)]
                        );
```

```
                //serial addition a_ii=a_{ii-1}+pf_ii
                trunc_add #(
                      .a_ibits(a_ibits), .a_fbits(a_fbits),
                      .b_ibits(a_ibits), .b_fbits(a_fbits),
                      .c_ibits(a_ibits), .c_fbits(a_fbits)
                      )
                      Add_SerialA (
                            a[a_maxbits-1-a_bits*(ii+1) : a_maxbits-a_bits*(
                               ii+2)],
                            pf[pf_maxbits-1-a_bits*ii : pf_maxbits-a_bits*(
                               ii+1)],
                            a[a_maxbits-1-a_bits*(ii) : a_maxbits-a_bits*(ii
                               +1)]
                            );

                mydff #(.dff_bits(w_bits))
                      DFF_W (
                            clk,
                            reset,
                            w_next[w_maxbits-1-w_bits*ii : w_maxbits-w_bits
                               *(ii+1)],
                            w[w_maxbits-1-w_bits*ii : w_maxbits-w_bits*(ii
                               +1)]
                            );
      end
endgenerate

endmodule
```

trunc_ams.v

The following source code describes two modules, namely *trunc_add* and *trunc_mult*, for handling addition and multiplication arithmetic respectively. Also included is the *trunc_sat* module, which handles truncation and saturation.

```
/*****************************************
This module takes 2 numbers, adds them, then truncates them. If
overflow occurs, the result is saturated. Note that the operands
and sum are signed numbers.

Last modified: Nov 20, 2006
```

```
By Raymond Lee
*******************************************/

module trunc_add(
       a,     // a and b are inputs
       b,
       c      // c is the final output
       );

parameter

/*NOTE: 1. *ibits includes the sign bit
       2. the wordlength of ctemp is calculated under the assumption that
             a has a larger wordlength than b.
       3. a and b should have the same fractional wordlength */

       a_ibits = 1, a_fbits = 19,
       b_ibits = 1, b_fbits = 19,
       c_ibits = 1, c_fbits = 19,
       ctemp_ibits = a_ibits + 1,
       ctemp_fbits = a_fbits ,

       a_bits = a_ibits + a_fbits ,
       b_bits = b_ibits + b_fbits ,
       c_bits = c_ibits + c_fbits ,
       ctemp_bits = ctemp_ibits + ctemp_fbits;

input signed [a_bits -1:0] a;
input signed [b_bits -1:0] b;
output signed [c_bits -1:0] c;

reg signed [ctemp_bits -1:0] ctemp; // ctemp is the full precision sum of
     a and b

trunc_sat #(
       .a_ibits(a_ibits), .a_fbits(a_fbits),
       .b_ibits(b_ibits), .b_fbits(b_fbits),
       .c_ibits(c_ibits), .c_fbits(c_fbits),
       .ctemp_ibits(ctemp_ibits), .ctemp_fbits(ctemp_fbits)
       )
       TRUNC_SAT_ADDER (
                   ctemp,
                   c
                   );

always @(a or b)
begin
       ctemp = a + b;
end
```

```
endmodule


/*****************************************
This module takes 2 numbers, multiplies them, then truncates them.
If overflow occurs, the result is saturated.

Last modified: Nov 19, 2006

By Raymond Lee
******************************************/

module trunc_mult(
        a,      // a and b are inputs
        b,
        c       // c is the final output
        );

parameter
/* NOTE: *ibits includes the sign bit */
        a_ibits = 2, a_fbits = 6,
        b_ibits = 1, b_fbits = 15,
        c_ibits = 2, c_fbits = 16,
        ctemp_ibits = a_ibits-1 + b_ibits-1 + 1+1,
        ctemp_fbits = a_fbits + b_fbits ,

        a_bits = a_ibits + a_fbits ,
        b_bits = b_ibits + b_fbits ,
        c_bits = c_ibits + c_fbits ,
        ctemp_bits = ctemp_ibits + ctemp_fbits;
        //ctrunc_msb = ctemp_bits-a_ibits-b_ibits+c_ibits -1;

input signed [a_bits -1:0] a;
input signed [b_bits -1:0] b;
output signed [c_bits -1:0] c;

reg signed [ctemp_bits -1:0] ctemp; // ctemp is the full precision
    product of a and b

trunc_sat #(
        .a_ibits(a_ibits), .a_fbits(a_fbits),
        .b_ibits(b_ibits), .b_fbits(b_fbits),
        .c_ibits(c_ibits), .c_fbits(c_fbits),
        .ctemp_ibits(ctemp_ibits), .ctemp_fbits(ctemp_fbits)
        )
        TRUNC_SAT_MULT (
                        ctemp,
                        c
```

```
                    );

always @(a or b)
begin
      ctemp = a * b;
end

endmodule


/*****************************************
This module truncates the input to a specified wordlength. If the input
is larger than the truncated result , then the output is saturated.

Last modified: Nov 20, 2006

By Raymond Lee
*****************************************/

module trunc_sat(
      ctemp,      // ctemp is the full precision input
      c     // c is the truncated/saturated output
      );

parameter
      a_ibits = 2, a_fbits = 3,
      b_ibits = 2, b_fbits = 3,
      c_ibits = 2, c_fbits = 3,
      ctemp_ibits = 3, ctemp_fbits = 4,

      c_bits = c_ibits + c_fbits ,
      ctemp_bits = ctemp_ibits + ctemp_fbits;

input signed [ctemp_bits -1:0] ctemp;
output signed [c_bits -1:0] c;

reg signed [c_bits -1:0] c, ctrunc, csat;
// csat is the saturated output derived from ctemp
// ctrunc is the truncated output derived from ctemp
reg sel;  // sel = 1 means overflow will occur after truncation =>
    saturate the ouput instead

always @(ctemp)
begin
      ctrunc = $signed(ctemp[ctemp_bits-1-ctemp_ibits+c_ibits: 0+
          ctemp_fbits-c_fbits]);
      csat = $signed({ctemp[ctemp_bits -1], {(c_bits -1) {~ctemp[
          ctemp_bits -1]} }});
```

```
        sel = ( ~(&ctemp[ctemp_bits-2:ctemp_bits-1-ctemp_ibits+c_ibits])&(
            ctemp[ctemp_bits-1]) ) | ( (|ctemp[ctemp_bits-2:ctemp_bits-1-
            ctemp_ibits+c_ibits])&(~ctemp[ctemp_bits-1]) );

        // MUX operations
        c = $signed((({c_bits{sel}})&csat)| (({c_bits{~sel}})&ctrunc));
end

endmodule
```

mydff.v

The following source code describes two D flip-flops: the first has a synchronous reset to zero and the second has synchronous reset to *beta.*

```
/*************************************
This module is a D flip-flop with a reset.

Last modified: Nov 25, 2006

By Raymond Lee
*****************************************/

module mydff(
        clk,
        reset,
        D,
        Q
        );

parameter    dff_bits = 8;

input clk;
input reset;
input [dff_bits-1:0] D;
output [dff_bits-1:0] Q;

reg [dff_bits-1:0] Q;

always @(posedge clk)
begin
        if (reset == 1)
                Q<=0;
        else
```

```
                Q<=D;
end

endmodule

/***********************************
This module is a D flip-flop that sets value to beta upon reset.

Last modified: Nov 25, 2006

By Raymond Lee
***************************************/
module mydff_beta(
        clk,
        reset,
        D,
        beta,
        Q
        );

parameter    dff_bits = 8;

input clk;
input reset;
input [dff_bits-1:0] D, beta;
output [dff_bits-1:0] Q;

reg [dff_bits-1:0] Q;

always @(posedge clk)
begin
        if (reset == 1)
                Q<=beta;
        else
                Q<=D;
end

endmodule
```

VITA AUCTORIS

Raymond Lee was born in Windsor, Ontario, Canada, on October 13, 1981. He graduated from Assumption High School in 2000. He then went on to earn a B.A.Sc. degree in Electrical and Computer Engineering in 2004 from the University of Windsor. Having chosen to continue his studies, he is currently a candidate in the Electrical and Computer Engineering M.A.Sc. program at the University of Windsor and will graduate in Spring 2007.