University of Strathclyde

Department of Electronic and Electrical Engineering

FPGA Implementation of Adaptive Filters

by

Michael N. Hanson

MPhil Thesis

2016

This thesis is the result of the author’s original research. It has been composed by the author and contains material that has been previously submitted for examination leading to the award of a degree at The University of Strathclyde in 2011.

Signed: [illegible]

Date: 15/08/2016

Abstract

Adaptive signal processing is an important topic of research covering many application areas such as audio signal processing, radar, wireless communications and control systems. In the context of wireless communications, the impulse response of the channel can vary rapidly with respect to time. Fast adaptive filtering algorithms are required in order to perform equalization of such channels. The two algorithms predominantly used in practice are variants of Least Mean Squares (LMS) and Recursive Least Squares (RLS) algorithms. LMS algorithms are more straightforward to implement however, RLS algorithms offer increased performance at the expense of greater complexity.

For very high throughput operation, dedicated hardware is required to keep up with the incoming sampling rate. Field Programmable Gate Arrays (FPGAs) can provide superior performance both in terms of power utilization and throughput, when compared to Graphic Processor Unit (GPU) or Digital Signal Processor (DSP) implementation. However, despite the potential performance advantage, FPGA implementation progress has been limited by the difficulty of programming such devices. This motivates the development of software allowing the user to program FPGAs in a more straightforward manner than direct low-level programming.

The first part of the following work seeks to alleviate this via means of a high abstraction level Intellectual Property (IP) core for both the classic LMS algorithm, and the Normalized LMS (NLMS) algorithm. High level parameters allow the user to trade resource utilization against throughput, choosing fully parallel, serial or partly-serial architectures. In the second part of this work, a survey is presented on the implementation of the QR Decomposition RLS (QRD-RLS) algorithm. Discussion is given on the numerical performance, cost and throughput of different architectures, with particular detail presented on the Givens based systolic array architecture.

Table of Contents

List of Figures

1 Introduction

1.1 Adaptive Signal Processing

The area of adaptive signal processing is concerned with the design of self-learning filters for a variety of applications, [1],[2],[3]. The generic model of an adaptive filter is given in Figure 1.1. Here, we consider the adaptive filter as being a Finite Impulse Response (FIR) filter with an adaptive algorithm employed to automatically adjust the weight values in order to minimise a specified error metric, e.g Least Square Error (LSE), Mean Square Error (MSE). Note that the sample index is represented by the variable, k.

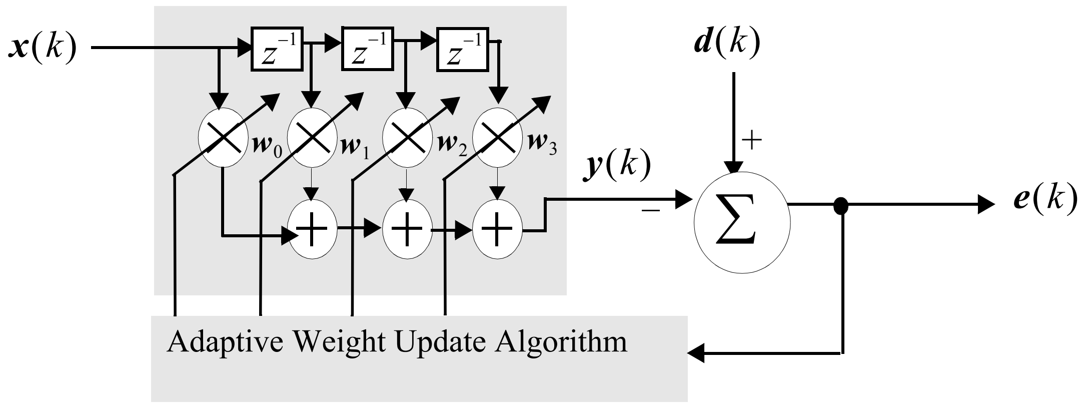
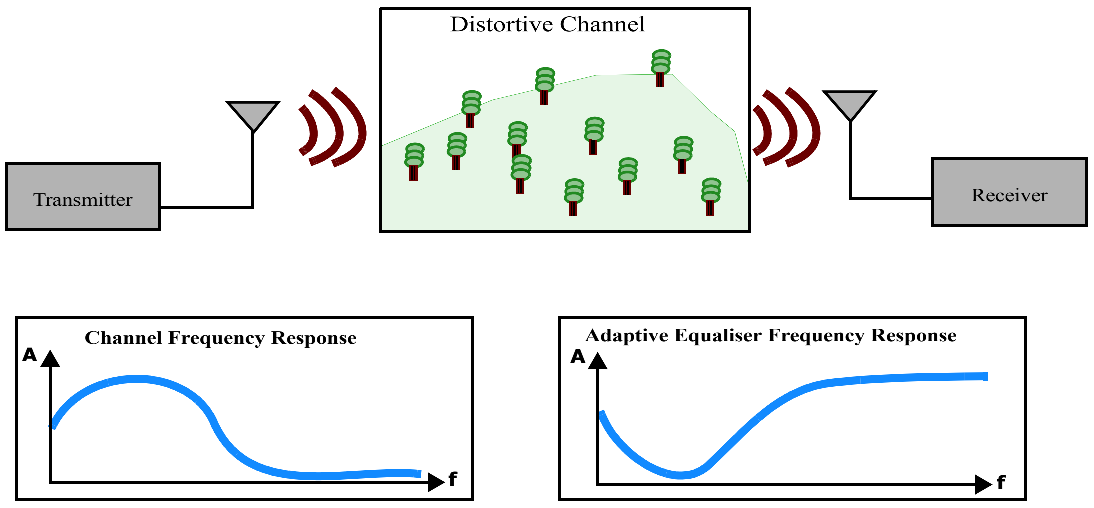


Figure 1.1: The Generic Adaptive FIR Filter

The commonly used notation given in the literature is now defined:

- $x(k)$ - Defines the input signal.
- $y(k)$ - Defines the output signal.
- $d(k)$ - Defines the desired signal to which the output of the filter, $y(k)$, should converge.
- $e(k)$ - Defines the error signal that is used to drive the adaptive filter. It is generated from the sample-by-sample difference between the current output signal, $y(k)$, and the current desired signal, $d(k)$.
- $w(k)$ - Defines the discrete values held in the weight vector of the FIR filter.

There are several general scenarios in which adaptive signal processing may be employed. The most exploited scenario from a communications standpoint is that of adaptive equalisation [3]. Here, the adaptive filter is employed in order to counter the frequency selective nature of the channel by converging towards the inverse of the channel's frequency response (Figure 1.2). Adaptive equalisers are used extensively in modern communications systems, and have been a key driver in improving performance within both wired and wireless scenarios.



Distortive Channel
Transmitter
Receiver
Channel Frequency Response
A
f
Adaptive Equaliser Frequency Response
A
f

Figure 1.2: The Adaptive Equalisation Scenario

1.2 Least Mean Squares Algorithm

One of the most popular and widely used adaptive filtering algorithms in both industry and academia is the Least Mean Squares (LMS) algorithm, which has been applied to a wide variety of applications such as equalisation, echo cancellation and adaptive beamforming. The LMS algorithm has a computational complexity in the order of $2N$ MACs (Multiply-ACcumulates) which is substantially lower than variants of the RLS (Recursive Least Squares) algorithm, which are typically in the order of N^2 MAC operations in complexity. It also has desirable qualities in terms of fixed point implementation. The hardware structure is highly regular and features numerically well conditioned multiplication and addition operations. This is in contrast to the more complex LSE based class of algorithms which feature numerically ill-conditioned operations such as square root, and hence exhibit a larger dynamic range (which translates into higher hardware cost).

The LMS algorithm, given in [1.1], [1.2] and [1.3], aims to find the optimal set of filter weights which shall minimise the MSE.

$$y(k) = w(k)x(k) \qquad [1.1]$$

$$e(k) = d(k) - y(k) \qquad [1.2]$$

$$w(k+1) = w(k) + 2\mu e(k)x(k) \qquad [1.3]$$

where 2μ is a constant commonly referred to as the step size, with bounds (for stable convergence) defined in [1.4].

$$0 < \mu < \frac{1}{N \cdot E[x^2(k)]} \qquad [1.4]$$

N is the order of the LMS filter, *i.e.* the number of coefficients, and *E[ ]* is the expectation function.

An alternative bound on the step size is given by 1.5, where λ_{max} is the greatest eigenvalue of the autocorrelation matrix $\boldsymbol{R}$. This shows that the convergence rate of the LMS algorithm is sensitive to the eigenvalue spread of the input.

$$0 < \mu < \frac{2}{\lambda_{max}} \qquad [1.5]$$

The LMS algorithm may be represented in either the real valued or complex valued form; the inclusion of the complex valued algorithm is a necessity for many practical applications, particularly in the context of communications systems, where the input data is often in the complex form, *e.g.* channel equalisation where complex modulation schemes are employed (Figure 1.3).

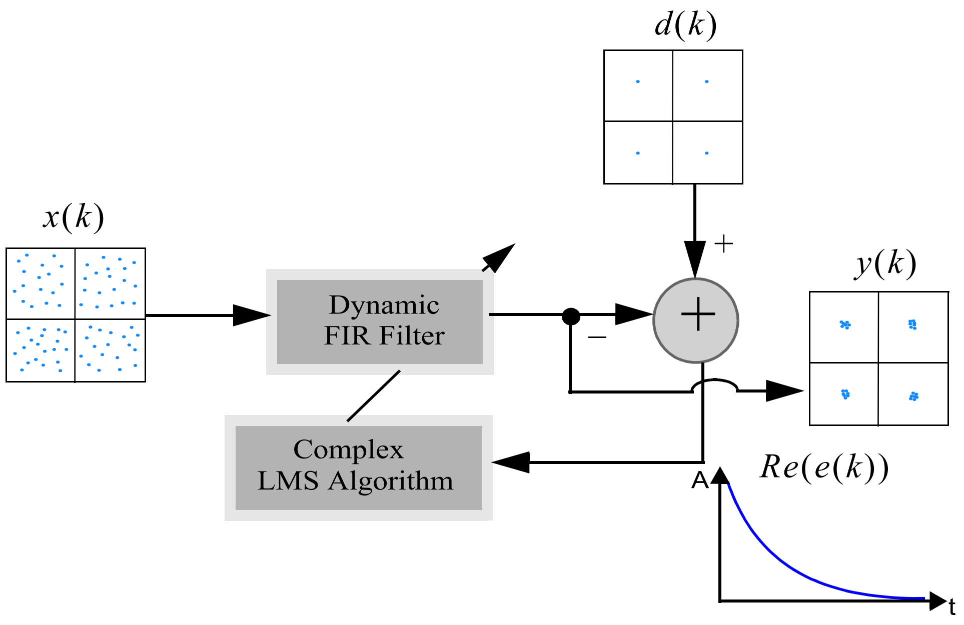


Figure 1.3: Complex LMS Based Adaptive Equalisation of Quadrature Phase Shift Keying Complex Modulation Scheme Input Data

The complex LMS algorithm differs from the real valued LMS algorithm by the use of complex arithmetic operations (Figure 1.4), and a single complex conjugate operation which must be performed [1.6], [4].

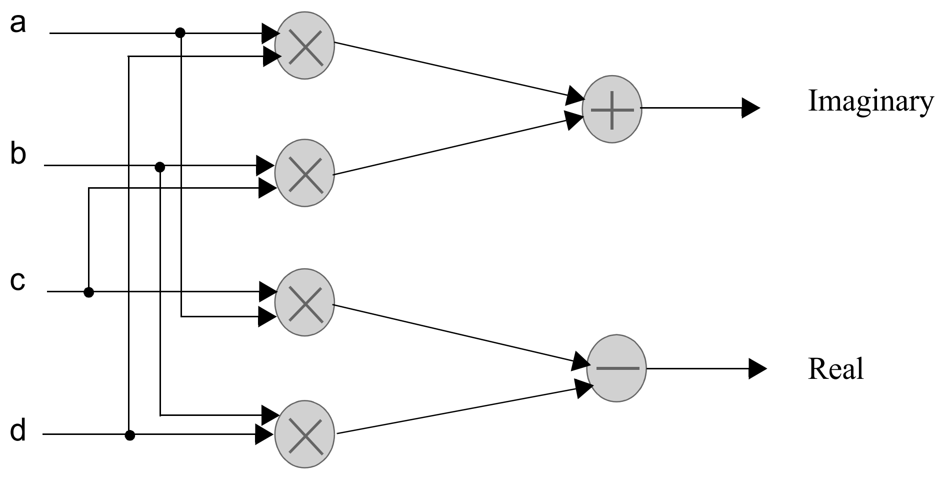
$$\boldsymbol{w}(k+1) = \boldsymbol{w}(k) + 2\mu e(k)\boldsymbol{x}^*(k) \qquad [1.6]$$

where $*$ denotes the complex conjugate operation.

The hardware cost of the complex valued algorithm is significantly greater than that of the real valued case, arising from the computational requirements of complex arithmetic. Complex addition incurs double the cost of real valued addition, whereas complex multiplication incurs a cost that is over four times that of the real valued case (four multiplications, one addition and one subtraction are required).

$$\boxed{(a+ib)(c+id) = (ac-bd)+j(bc+ad)}$$

(a)



$$\boxed{(a+jb)+(c+jd) = (a+c)+j(b+d)}$$

(b)

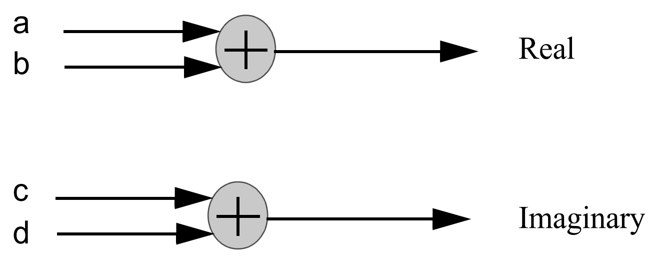


Figure 1.4: Decomposition of Complex Arithmetic Operations to Individual Real Arithmetic Operations: (a) Multiplication and (b) Addition

1.3 QRD- Recursive Least Squares Algorithm

The LMS algorithm uses the mean squared error as the performance criterion with which to update the filter weights. Due to this approximation, the rate at which the error signal converges towards steady state is limited. Where algorithm performance is at a premium, and it is acceptable for higher cost, RLS algorithms may be considered due to the higher rate of convergence.

The cost function of the least squares solution is defined as the total sum of squared errors:

$$v(k) = \boldsymbol{e}_k^T \boldsymbol{e}_k \qquad [1.7]$$

where $\boldsymbol{e}_k = [e_o, e_1, e_2, \ldots e_k]$ is the error signal vector, representing the difference between input and desired signal.

$$v(k) = [\boldsymbol{d}_k - \boldsymbol{X}_k \boldsymbol{w}]^T [\boldsymbol{d}_k - \boldsymbol{X}_k \boldsymbol{w}] \qquad [1.8]$$

where $\boldsymbol{d}_k = [d_0, d_1, d_2, \ldots d_k]$ is the desired signal vector, $\boldsymbol{X}_k$ is the input data matrix, and $\boldsymbol{w}$ is the weight vector.

$$v(k) = \boldsymbol{d}_k^T \boldsymbol{d}_k + \boldsymbol{w}^T \boldsymbol{X}_k^T \boldsymbol{X}_k \boldsymbol{w} - 2\boldsymbol{d}_k^T \boldsymbol{X}_k \boldsymbol{w} \qquad [1.9]$$

The equation is quadratic in $\boldsymbol{w}$, therefore performing partial differentation with respect to $\boldsymbol{w}$ can be used to find the weight vector $\boldsymbol{w}_{LS}$ which minimizes the total squared error.

$$\frac{\delta}{\delta \boldsymbol{w}} v(k) = 2\boldsymbol{X}_k^T \boldsymbol{X}_k \boldsymbol{w} - 2\boldsymbol{X}_k^T \boldsymbol{d}_k = -2\boldsymbol{X}_k^T [\boldsymbol{d}_k - \boldsymbol{X}_k \boldsymbol{w}] \qquad [1.10]$$

$$\boldsymbol{X}_k^T \boldsymbol{X}_k \boldsymbol{w}_{LS} = \boldsymbol{X}_k^T \boldsymbol{d}_k \qquad [1.11]$$

Rearranging the result in [1.11] gives the least squares solution for computing the weight vector, [1.12].

$$\boldsymbol{w}_{LS} = [\boldsymbol{X}_k^T \boldsymbol{X}_k]^{-1} \boldsymbol{X}_k{}^T \boldsymbol{d}_k \qquad [1.12]$$

The recursive least squares solution can be derived from [1.12]. The derivation will not be presented here for brevity, however the reader is referred to [2] for a full proof. The (complex) RLS algorithm is presented in [1.13],[1.14], [1.15], [1.16].

$$\gamma(k) = \frac{1}{(1 + \boldsymbol{x}(k)\boldsymbol{P}(k-1)\boldsymbol{x}(k)^{\boldsymbol{H}})} \qquad [1.13]$$

$$\boldsymbol{g}(k) = \boldsymbol{P}(k-1)\boldsymbol{x}(k)^{\boldsymbol{H}}\gamma(k) \qquad [1.14]$$

$$\boldsymbol{w}(k) = \boldsymbol{w}(k-1) + \boldsymbol{g}(k)[d(k) - \boldsymbol{x}(k)\boldsymbol{w}^{T}(k-1)] \qquad [1.15]$$

$$\boldsymbol{P}(k) = \boldsymbol{P}(k-1) - \frac{\boldsymbol{g}(k)\boldsymbol{g}(k)^{\boldsymbol{H}}}{\gamma(k)} \qquad [1.16]$$

where $\boldsymbol{P}(i-1)$ should be initialized to $\delta \cdot \boldsymbol{I}$. δ is a small constant referred to as the forgetting factor which de-emphasizes older samples. $\boldsymbol{I}$ is the identity matrix. All other parameters are initialized to 0. H represents the Hermitian transpose.

It is well known that the RLS algorithm suffers from numerical stability issues in the presence of finite precision arithmetic. The requirement for division operations results in high dynamic range requirements, leading to floating point arithmetic being preferred over fixed point Very Large Scale Integration (VLSI) implementation. Studies have been performed to quantify the numerical performance, [6], and modifications suggested [7]. QRD-RLS allows for RLS to be performed in a numerically stable fashion. Unlike the conventional RLS algorithm, it also has forms which are well suited for VLSI implementation.

The QR factorization takes an input matrix $\boldsymbol{X}$.and decomposes it into an upper triangular matrix $\boldsymbol{U} = \begin{bmatrix} \boldsymbol{R} \\ 0 \end{bmatrix}$ and orthogonal matrix $\boldsymbol{Q}$, as shown in Figure 1.5.

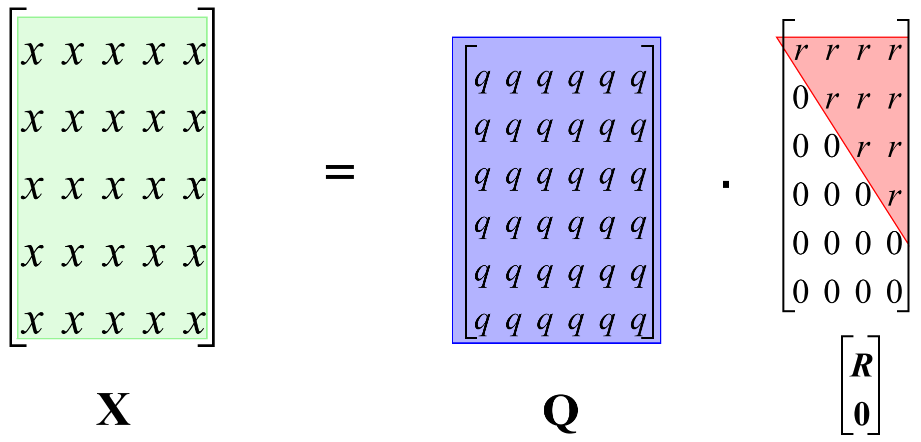


Figure 1.5: QR Decomposition

Now, looking back at the least squares solution given by [1.12], we can substitute $\boldsymbol{X}_k = \boldsymbol{QU}$ and simplify to reach the QR least squares solution, [1.21], where the k indes is dropped for the sake of clarity:

$$[(\boldsymbol{QU})^T(\boldsymbol{QU})]\boldsymbol{w}_{\boldsymbol{LS}} = (\boldsymbol{QU})^T\boldsymbol{d} \qquad [1.17]$$

$$[\boldsymbol{U}^T\boldsymbol{Q}^T\boldsymbol{QU}]\boldsymbol{w}_{\boldsymbol{LS}} = \boldsymbol{U}^T\boldsymbol{Q}^T\boldsymbol{d} \qquad [1.18]$$

$$\boldsymbol{U}^T\boldsymbol{U}\boldsymbol{w}_{\boldsymbol{LS}} = \boldsymbol{U}^T\boldsymbol{Q}^T\boldsymbol{d} \qquad [1.19]$$

$$\boldsymbol{U}\boldsymbol{w}_{\boldsymbol{LS}} = \boldsymbol{Q}^{\boldsymbol{T}}\boldsymbol{d} = \boldsymbol{d} \qquad [1.20]$$

$$\boldsymbol{R}\boldsymbol{w}_{\boldsymbol{LS}} = \boldsymbol{d} \qquad [1.21]$$

Performing the QR decomposition is numerically robust in the presence of finite precision arithmetic. Once the QR decomposition has been performed, the least squares solution weight vector, $\boldsymbol{w}_{\boldsymbol{LS}}$, can be found via straightforward backsubstitution, (Figure 1.6).

$$\begin{array}{ccccc} \boldsymbol{R} & & \boldsymbol{w_{LS}} & & \boldsymbol{d} \\ \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} & r_{15} \\ 0 & r_{22} & r_{23} & r_{24} & r_{25} \\ 0 & 0 & r_{33} & r_{34} & r_{35} \\ 0 & 0 & 0 & r_{44} & r_{45} \\ 0 & 0 & 0 & 0 & r_{55} \end{bmatrix} & \cdot & \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \end{bmatrix} & = & \begin{bmatrix} d_1' \\ d_2' \\ d_3' \\ d_4' \\ d_5' \end{bmatrix} \end{array}$$

$$\begin{tikzcd}
w_5 = \dfrac{d_5'}{r_{55}} \arrow[r] & w_4 = \dfrac{d_4' - r_{45}w_5}{r_{44}} \arrow[r] & w_3 = \dfrac{d_3' - r_{35}w_5 - r_{34}w_4}{r_{33}} \arrow[d] \\
w_1 = \dfrac{d_1' - r_{15}w_5 - r_{14}w_4 - r_{13}w_3 - r_{12}w_2}{r_{11}} & & w_2 = \dfrac{d_2' - r_{25}w_5 - r_{24}w_4 - r_{23}w_3}{r22} \arrow[ll]
\end{tikzcd}$$

Figure 1.6: A 5 Weight Example of a QR Least Squares Solution via Backsubstitution

1.4 FPGA Technology

FPGAs are a type of VLSI technology which offer a compromise between the performance of Application Specific Integrated Circuit (ASIC) implementation and the flexibility of DSP implementation. From a high level of abstraction, an FPGA can be thought of as a collection of digital logic elements connected together by a complex wiring matrix, as shown in Figure 1.7. Hardware designers program FPGAs via a high level description of the digital circuit which is accepted by the FPGA synthesis tool. The synthesis tool is able to translate the functional design of the circuit into a series of interconnections on the FPGA.

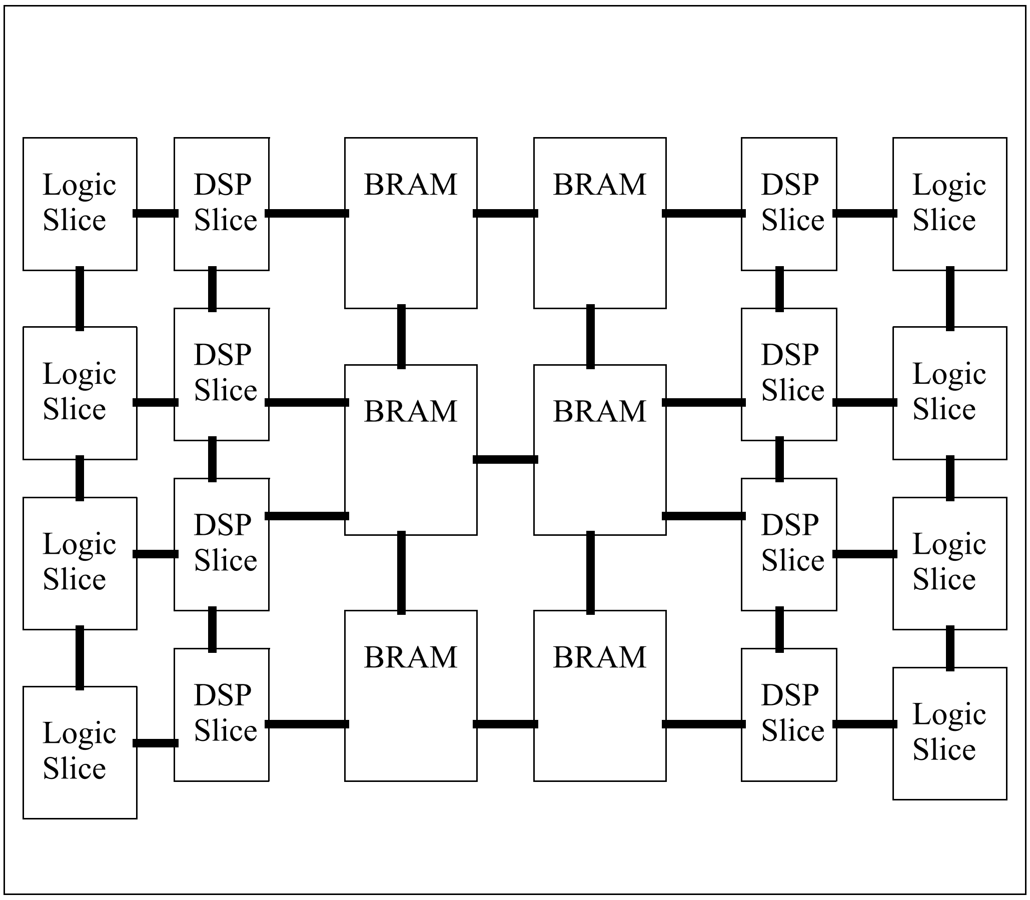


Figure 1.7: FPGA Technology: A Complex Circuit is Formed from the Interconnection of Low Level Operations

In contrast to an ASIC implementation, FPGAs are programmed (and reprogrammed) by the designer as required after manufacture. ASICs on the other hand are manufactured for one specific purpose and are limited in terms of programmability after fabrication. ASIC implementation will result in very high performance for a given application, both in terms of power consumption and throughput. However, ASIC implementations are extremely costly, and are therefore prohibitively expensive for all but the most high volume market areas. FPGA implementation is therefore attractive for designers who require high throughput, low power, designs but cannot absorb the prohibitive costs associated with ASIC design.

DSP/GPUs also offer an implementation platform which can be repeatedly programmed "in the field". However, such devices are general purpose for many applications, and as such there is a performance hit from the associated generality of the hardware. The memory access associated with performing parallel DSP algorithms on a device which is inherently sequential is one such source of performance degradation. Figure 1.8 illustrates this concept. The FIR implementation on DSP platform suffers a performance hit associated with repeated memory access. The FPGA implementation unrolls the algorithm onto parallel hardware allowing the memory access to be performed in place.

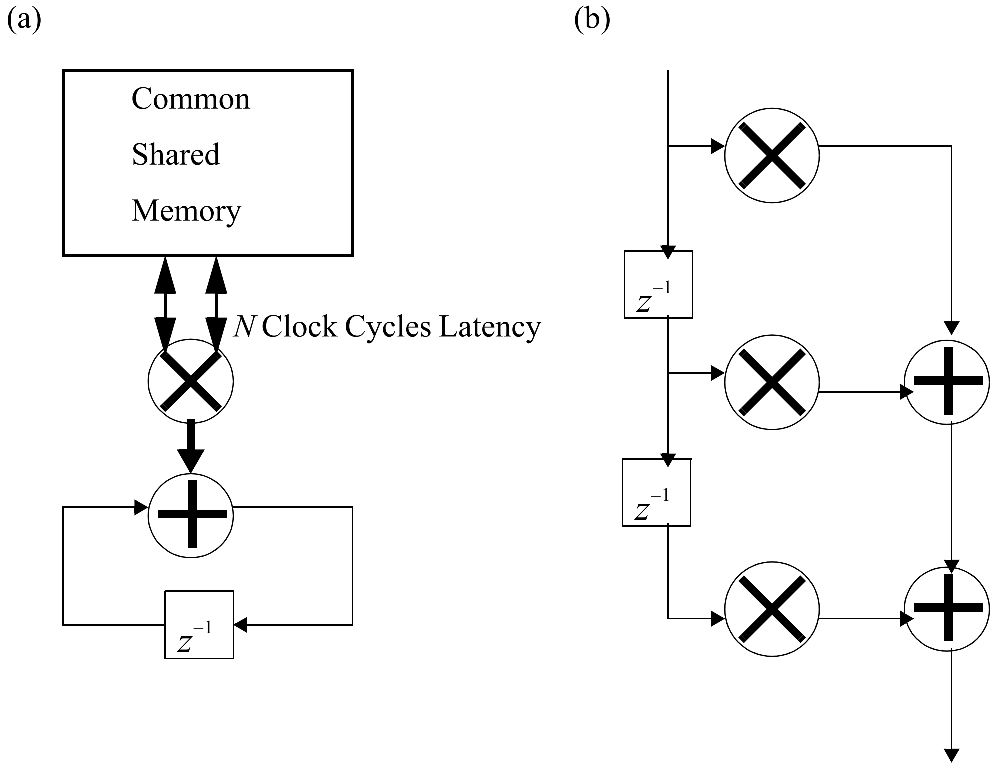


Figure 1.8: Implementation Mapping of a Parallel FIR Filter: (a) Sequential DSP Implementation; (b) Parallel Unrolled FPGA Implementation

While FPGA implementation features the aforementioned advantages, such devices are not straightforward to program for non hardware orientated engineers. In contrast to implementation on floating point DSP, FPGAs in general will support only fixed point, integer arithmetic. It is therefore up to the designer to track the binary point appropriately when performing arithmetic operations. FPGAs must be programmed via code written in a Hardware Description Language (HDL). Such a language captures the parallelism of operations explicitly in the code and is therefore quite a departure from development in C/C++, for example.

The so called “design-gap” when programming FPGAs due to the unfamiliarity of many embedded firmware engineers with hardware-specific concepts is one of the primary blocking factors for more widespread FPGA adoption. In light of this, many FPGA vendors offer high level IP cores in order to abstract complex DSP algorithms into an easy to use package, for which the user only has to configure a few parameters. High level solutions such as this are often referred to as “compilers”, as the scenario is analogous of the compilation of high level functional code into low level assembly language.

1.5 Aims and Objectives

In this project, we seek to close the design gap associated with the FPGA implementation of the LMS adaptive filtering algorithm. A high level compiler is detailed which allows for many implementation styles to be rapidly prototyped. The user is able to make high level design trade-offs such as hardware cost against throughput, via configuration of a small number of high level parameters. A detailed account of the results obtained after synthesizing many different circuits for the LMS algorithm is presented. This forms the first part of the thesis.

The work carried out on the Adaptive LMS Compiler began during the course of the BEng (Hons) final year project (University of Strathclyde, 2011). At the end of this period, fully parallel and fully serial architectures for LMS supporting either real valued or complex valued arithmetic were supported. During the course of the MPhil, this was extended to a parallel-serial architecture, and NLMS support was added. The complete compiler therefore consists of:

- Real or complex valued arithmetic processing.
- LMS or NLMS algorithm.
- Parallel, serial or partly-serial architecture.

In addition to the non-trivial extension of the LMS architecture to support parallel-serial processing, it was during the MPhil programme that the extensive numerical verification and hardware implementation study was carried out. This study represents a large body of results from which analysis is drawn.

The thesis also presents a detailed survey of QRD-RLS implementation from the FPGA design perspective. Discussion is given on the various architectural design optimizations which can be made in order to form an effective, fixed point implementation. This forms the second part of the thesis.

1.6 Project Outline

As previously noted, the project is divided into two separate but related parts. Firstly, practical work is carried out in the design of an "Adaptive LMS Compiler", suitable for automating the generation of RTL for LMS based adaptive filtering. Secondly, a detailed survey is given of the QRD-RLS algorithm, which is well known to be suitable for VLSI implementation. The two parts combined form a comprehensive report of the VLSI implementation of adaptive filters.

In Part 1 (pp 19-75) of the thesis, the different hardware architectures used to achieve the various implementation options of the Adaptive LMS Compiler are detailed. Results of an extensive numerical performance analysis are given in order to validate the correct operation of the compiler. Finally, FPGA implementation results are provided and evaluated.

Following the design of the Adaptive LMS Compiler, a survey of the implementation of QRD-RLS adaptive filters was carried out, with particular focus on the Givens based systolic array architecture. This forms Part 2 of the thesis (pp 75- 113).

The two parts of the thesis are brought together in the Conclusion, (pp 114-117), where common topics between both LMS and QRD-RLS implementation are defined. Potential future improvements and gaps in the current literature are also defined.

Part 1: Adaptive LMS Compiler

In the first part of this report, the work carried out on the creation of an Adaptive LMS Compiler is detailed. Chapter 2 gives an overview of the various hardware architecture designs for fully parallel, fully serial and parallel-serial LMS/NLMS architectures for both real and complex valued arithmetic. The results of a numerical performance analysis carried out to validate the correct operation of the compiler are given in Chapter 3. Finally, Chapter 4 presents detailed results of FPGA hardware implementation, investigating metrics such as slice register utilization, LUT utilization, DSP48 utilization, and maximum achievable sampling frequency of input data.

2 Hardware Architecture Overview

The Adaptive LMS Compiler offers the user the option of both the standard LMS algorithm and the NLMS algorithm in both the real valued and complex valued form. Three distinct filter architectures - fully parallel, fully serial and a flexible parallel - serial architecture are available for each algorithm, allowing for an effective trade off to be made between the hardware cost and throughput (maximum achievable sampling frequency) of the design.

In this section, a brief overview of each architecture (parallel, serial, parallel - serial) is given - where the operation of the control hardware needed in order to implement varying degrees of serialisation is discussed. Further information is also given on the hardware implementation of the NMLS based adaptive filter. For the sake of clarity, the discussion is restricted to real valued arithmetic. Before delving further into the details of the architectures however, an introduction to some commonly used FPGA design terminology is provided.

2.1 FPGA Design Introduction

In this section, an overview of the fundamental FPGA design terminology is provided. An important concept to begin with is that of the critical path. When designing for hardware, the maximum rate at which the device can be clocked is limited by the maximum combinatorial path between any two registers in the design. This concept is illustrated in Figure 2.1.

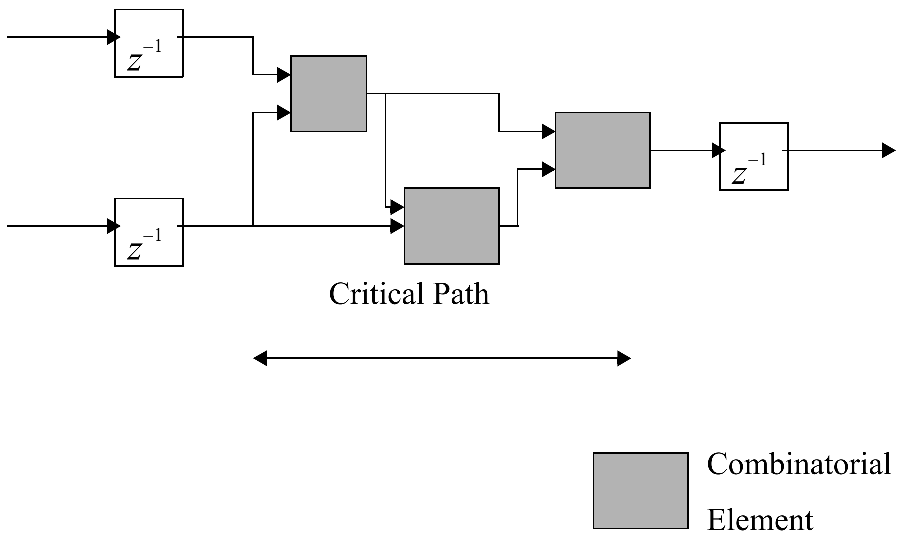


Figure 2.1: FPGA Design: Critical Path Defined as the Maximum Combinatorial Path Between Two Registers

In order to increase the clock frequency, thereby increasing the overall throughput of samples from the input to the output of the circuit, a technique known as pipelining is employed. Pipelining consists of inserting registers throughout the combinatorial logic, thereby reducing the critical path between two registers, as shown in Figure 2.2.

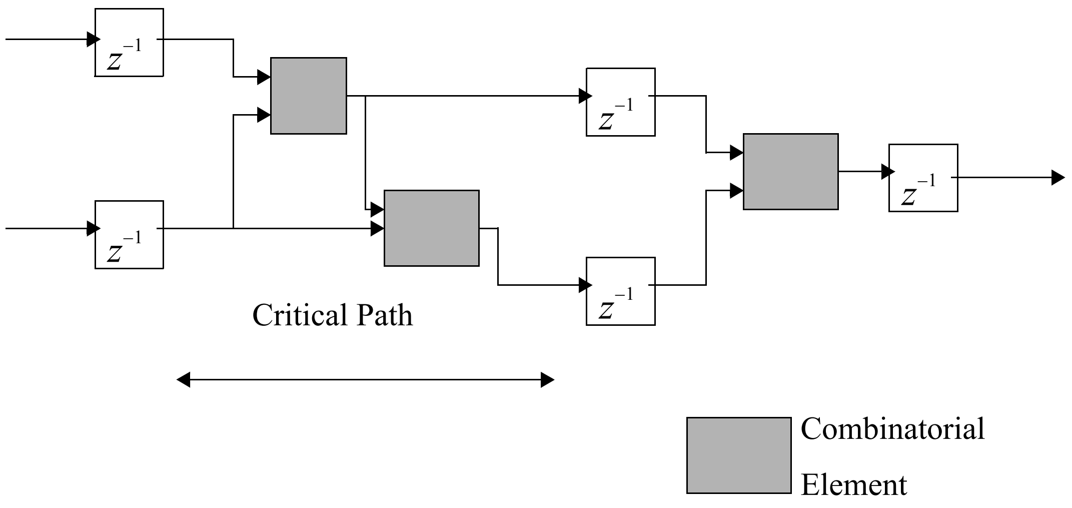


Figure 2.2: Critical Path Reduction via Pipeline Optimization

Another key concept of FPGA design is that of parallelization. When designing for custom hardware, the designer has complete control over the hardware resources used in the architecture. For the example of an FIR filter - in fully parallel mode, a separate hardware MAC unit is used for each cofficient; in fully serial mode a single MAC unit is used to compute all coefficients; and in parallel-serial mode there are several MAC units each computing the result of a number of coefficients. A general design goal is to implement the architecture using as few resources as possible whilst still being able to keep pace with the incoming data.

The following sections describe the hardware architecture for the Adaptive LMS Compiler. The compiler allows the user to have full control over the hardware resources used to implement a filter of given order.

2.2 The Fully Parallel LMS Filter

The most straightforward manner by which the LMS algorithm may be mapped into hardware is that of a fully parallel architecture (Figure 2.3). In this implementation, two Multiply ACcumulate (MAC) components are required for each coefficient of the filter, and hence the total MAC cost is given by the linear relationship, $MACs = 2N$, where M is the total number of coefficients to be computed in the LMS filter.

Although the fully parallel architecture has the highest hardware cost of the various architectures, the advantage of this approach lies in the relationship between the maximum clock frequency and the maximum sampling frequency, given by $f_{clk} = f_{sampling}$, which allows the fully parallel design to accommodate the highest throughput of any of the implemented architectures.

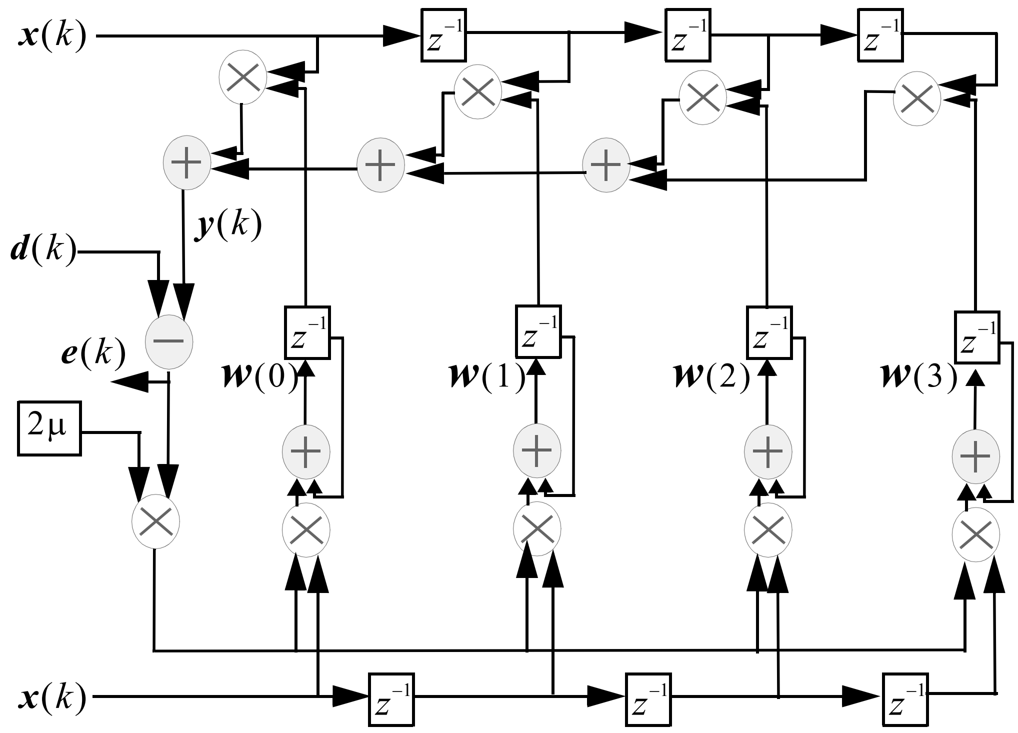


Figure 2.3: The Fully Parallel LMS Filter Architecture: *N*=4

2.3 The Fully Serial LMS Filter

In order to allow for a scalable implementation to be realised, the LMS filter may be serialised, whereby a single MAC unit (which when combined with appropriate control logic is referred to as a Processor Core (PC)) is time-shared in order to calculate the results of several coefficients, hence greatly reducing the hardware cost.

To achieve full serialisation, it is necessary to operate the processor core at a faster rate than the input data, and hence the sampling frequency of the input data is constrained by an integer factor, [2.1].

$$f_{sampling} = \frac{f_{clk}}{(2N) + PIPELINE} \qquad [2.1]$$

where N is the number of coefficients computed by the serialised processor core, *PIPELINE* is an integer delay incurred by pipeline optimisation (architecture and parameter dependent) and f_{clk} is the maximum clock rate of the FPGA - which the processing elements of the design operate at.

Figure 2.2 provides an overview of the fully serial architecture. The Processor Core (PC) is the main computational unit of the design, and computes both the filter output and the updated weight values. The Error Signal / Weight Update, (ES/WU), logic is used to capture the appropriate sample of accumulated filter output $\boldsymbol{y}(k)$ from the PC, and to then compute the error signal $\boldsymbol{e}(k)$ and the resulting value needed for the weight update operation, $2\mu\boldsymbol{e}(k)$. Both the PC and the ES/WU logic contain a single MAC component - the multiplier present in each of these components will be a large contributor towards the critical path in terms of propagation delay and so both of these multipliers are pipelined, incurring a delay of two clock cycles in each multiplication, (a delay of three clock cycles is incurred for the case of complex valued arithmetic).

Also present in the design is an Input Control (IC) unit - this component controls the input data samples $\boldsymbol{x}(k)$ that are passed to the PC, and takes into account the pipeline delay of both the multiplier present in the PC and that present in the ES/WU logic. This is achieved via a combination of a Finite State Machine (FSM), a counter and an Addressable Shift Register (ASR).

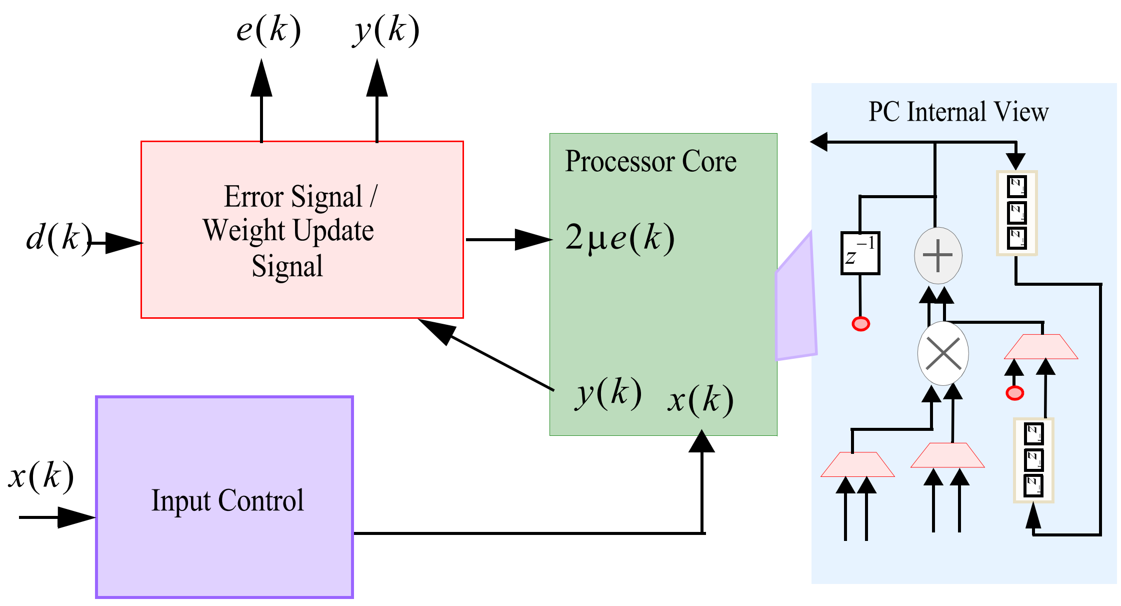


Figure 2.4: The Fully Serial LMS Filter Architecture

Figure 2.5 gives a closer look at the internal operation of the Processor Core. The PC consists of a single MAC component which is time shared via a 2:1 multiplexer for both the filtering operation and the weight update operation. A set of internal shift registers ensure that the weight values for both the filter section and the weight update section are appropriately stored between iterations.

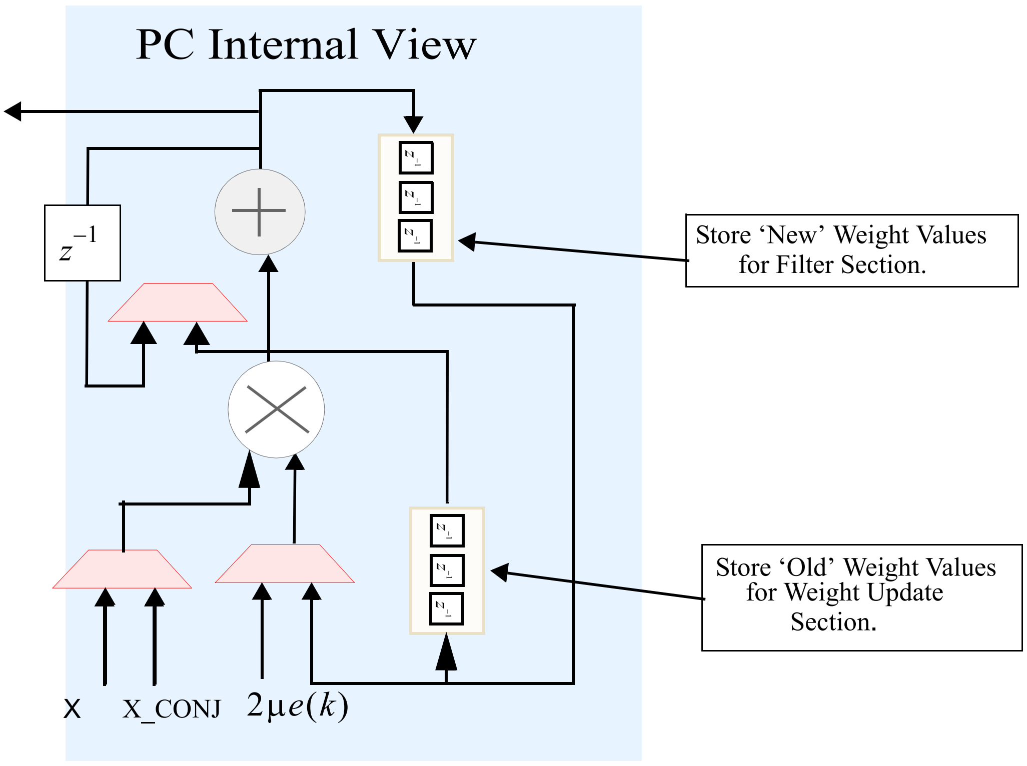

PC Internal View
z^{-1}
Store 'New' Weight Values for Filter Section.
Store 'Old' Weight Values for Weight Update Section.
X
X_CONJ
$2\mu e(k)$

Figure 2.5: A Closer Look at the Processor Core

The PC must receive the correct samples of input data $\boldsymbol{x}(k)$ at the appropriate sample instants in order for the numerical integrity of the algorithm to be maintained. To achieve this, a State Machine Counter (SMC) (i.e a combination of an FSM and a counter) and ASR component are combined to form the Input Control component (Figure 2.4). The ASR allows for samples to be read out using integer addressing values which correspond to internal locations in the memory array. A write operation is performed automatically once every sample period, where a new value of $\boldsymbol{x}(k)$ is accepted on the input port and the current oldest value of $\boldsymbol{x}(k)$ is sent to the output port. The SMC performs the addressing operation to the ASR, taking into account the pipeline delay incurred by each of the multipliers.

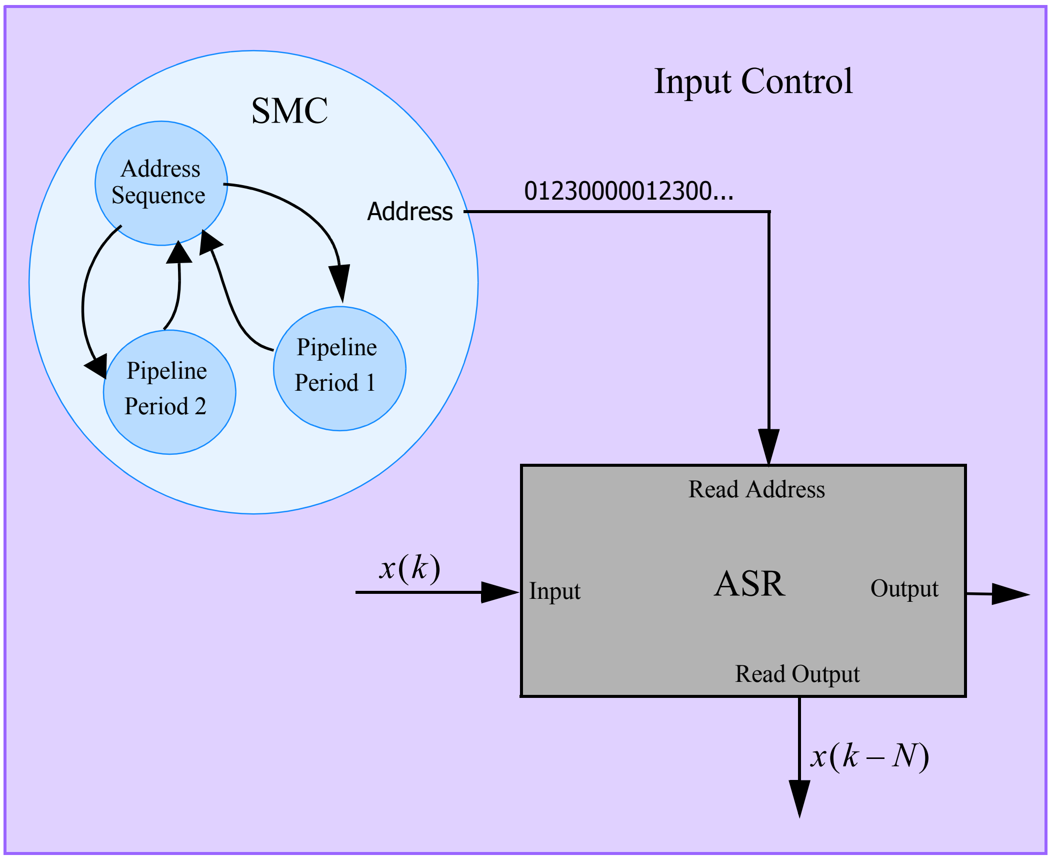


Figure 2.6: Operation of the Input Control Component

An overview of the sample by sample output from the SMC is given in Figure 2.7. Where zeros are present, this indicates that the state machine is taking into account the pipeline stages of the architecture.

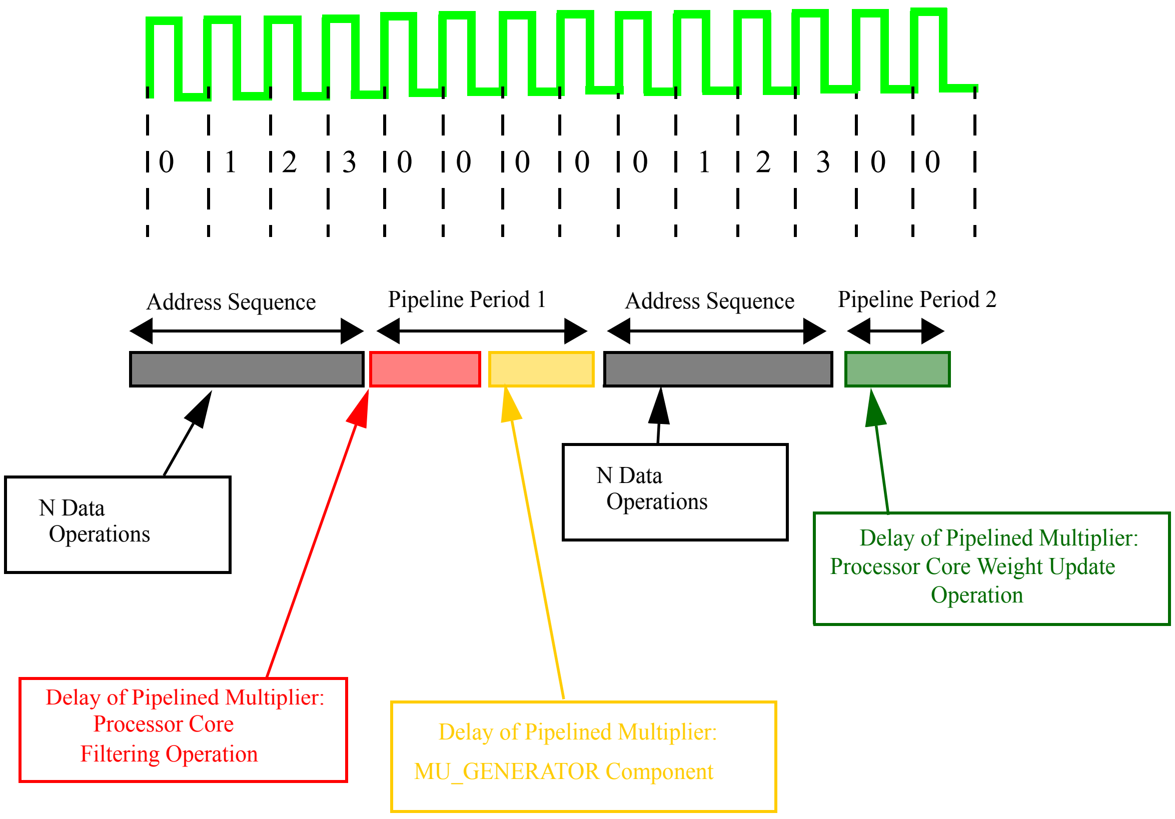

0
1
2
3
0
0
0
0
0
1
2
3
0
0
Address Sequence
Pipeline Period 1
Address Sequence
Pipeline Period 2
N Data Operations
N Data Operations
Delay of Pipelined Multiplier: Processor Core Weight Update Operation
Delay of Pipelined Multiplier: Processor Core Filtering Operation
Delay of Pipelined Multiplier: MU_GENERATOR Component

Figure 2.7: Operation of the State Machine Counter

2.4 The Parallel-Serial LMS Filter

The flexibility that serialisation brings to the design may be extended further with the development of a parallel-serial type architecture. The parallel-serial LMS filter uses a parallel array of PCs, each of which serially computes the results of several coefficients. Like the fully serial architecture, the parallel-serial architecture is deeply pipelined to ensure that the critical path is minimised and hence that the achievable clock frequency is maximised.

The parallel-serial architecture allows for a trade off to be reached between the hardware cost and throughput requirements of the design. As evidenced by [2.2], the hardware cost (number of PC components) may be traded against the number of coefficients computed per PC.

$$f_{sampling} = \frac{f_{clk} \times PC}{2N \times PIPELINE} \qquad [2.2]$$

where N is the total number of coefficients computed in the LMS filter, PC is the total number of Processor Core components employed in the parallel-serial design and $PIPELINE$ is the overhead incurred due to pipeline optimisation (architecture and parameter dependent).

Note that the sampling frequency calculation may also be expressed in the alternate form given by [2.3].

$$f_{sampling} = \frac{f_{clk}}{2 \times N_{PC} + PIPELINE} \qquad [2.3]$$

where N_{PC} is the number of coefficients computed per PC.

An overview of the parallel serial LMS architecture is given in Figure 2.9. The original single ASR of length N featured in the fully serial architecture is partitioned into several ASRs of length N_{PC}, referring to the number of coefficients computed per PC. Each is connected in a linear row, such that at each sample period, the output of the previous ASR serves as the input to the next ASR. The SMC component performs the same role as was the case for the fully serial filter, providing the correct addressing signal for each of the ASR components at the appropriate sampling instants.

As each PC computes only a partial sum of the overall filter output, it is necessary for a parallel addition of each of the PC outputs to be performed. To achieve this, an adder tree type structure is used (as opposed to a linear summation). The adder tree architecture has a lower critical path than the standard linear summation type architecture, and therefore the *PIPELINE* overhead is reduced. This is illustrated in Figure 2.8, where the adder tree architecture has a critical path of three adders, compared to the critical path of eight adders for the linear summation architecture.

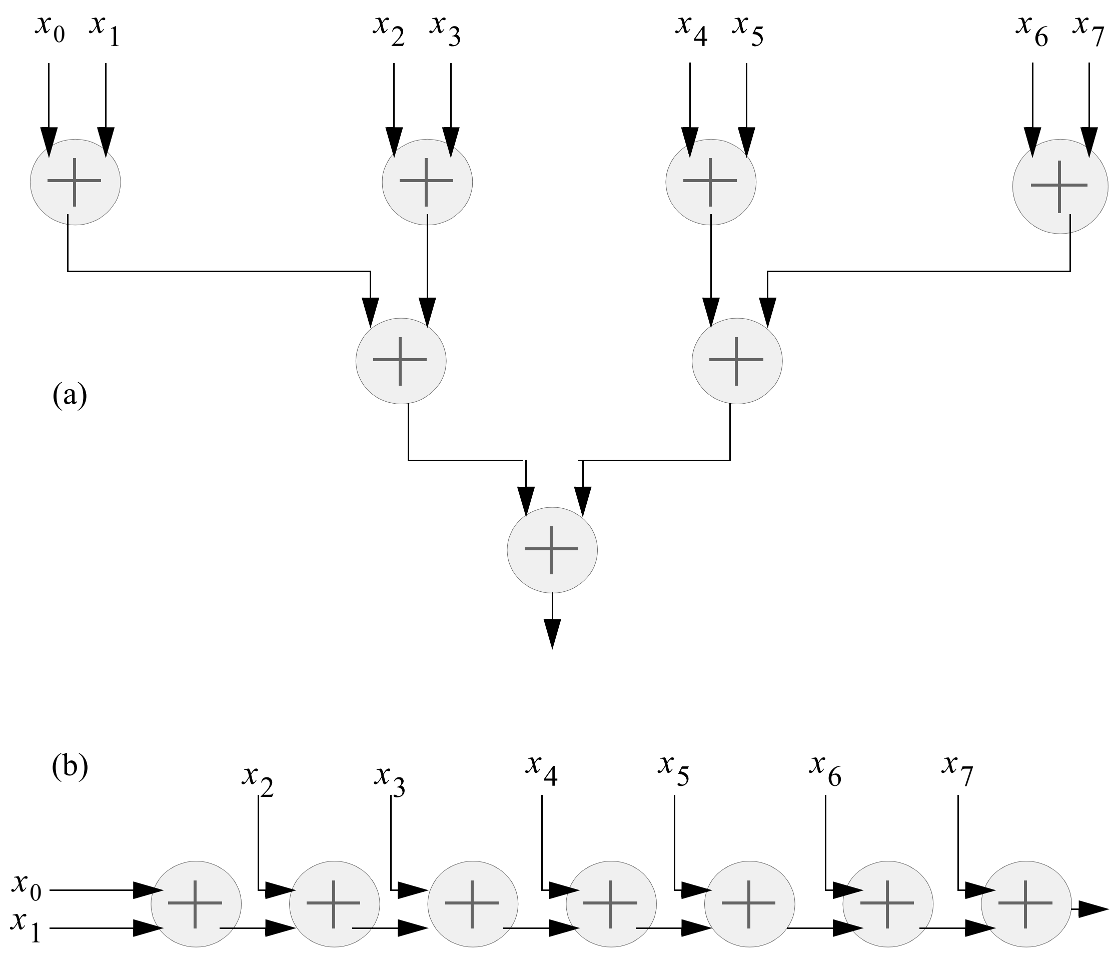


Figure 2.8: Adder Tree (a) vs Linear Summation (b)Architectures

Following the tree summation stage, an accumulator component is present - this is necessary due to the serial nature of the design. Recall that at each iteration, each PC will compute the result of only a single coefficient, and hence the PC summation from the adder tree must be accumulated over *N* iterations until the overall filter output is computed. After *N* iterations have passed, then the output of the accumulator is saved via the CAPTURE REGISTER - this forms the output signal $\boldsymbol{y}(k)$ which is held constant over the sample period duration. The $\boldsymbol{y}(k)$ signal is used in conjunction with the input signal $\boldsymbol{d}(k)$ in order to form the error signal $\boldsymbol{e}(k)$ and subsequently, the $2\mu\boldsymbol{e}(k)$ signal needed for the weight update operation.

It should be noted that the weight update operation is contained within the PCs and follows the same process as was described for the fully serial case.

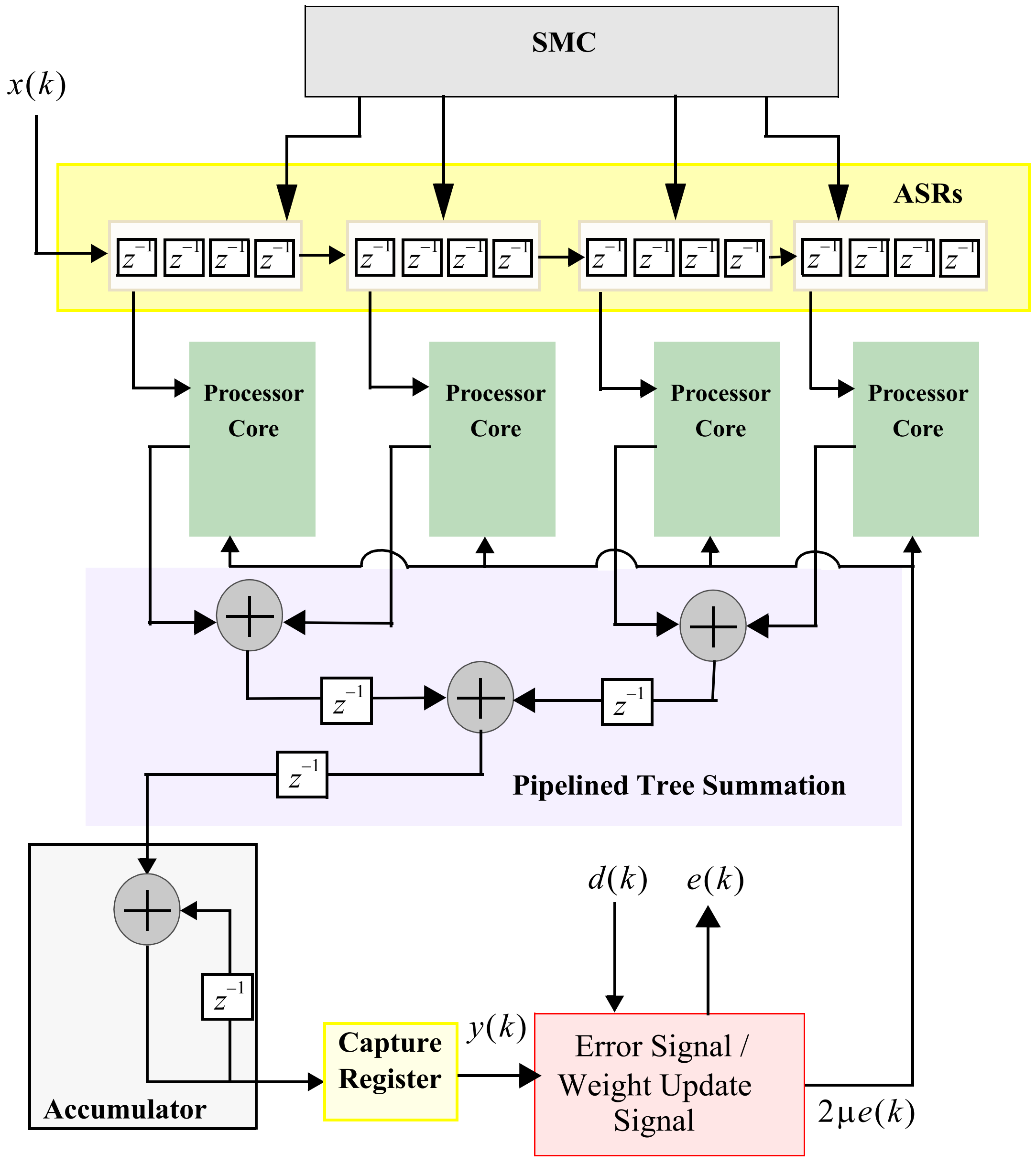
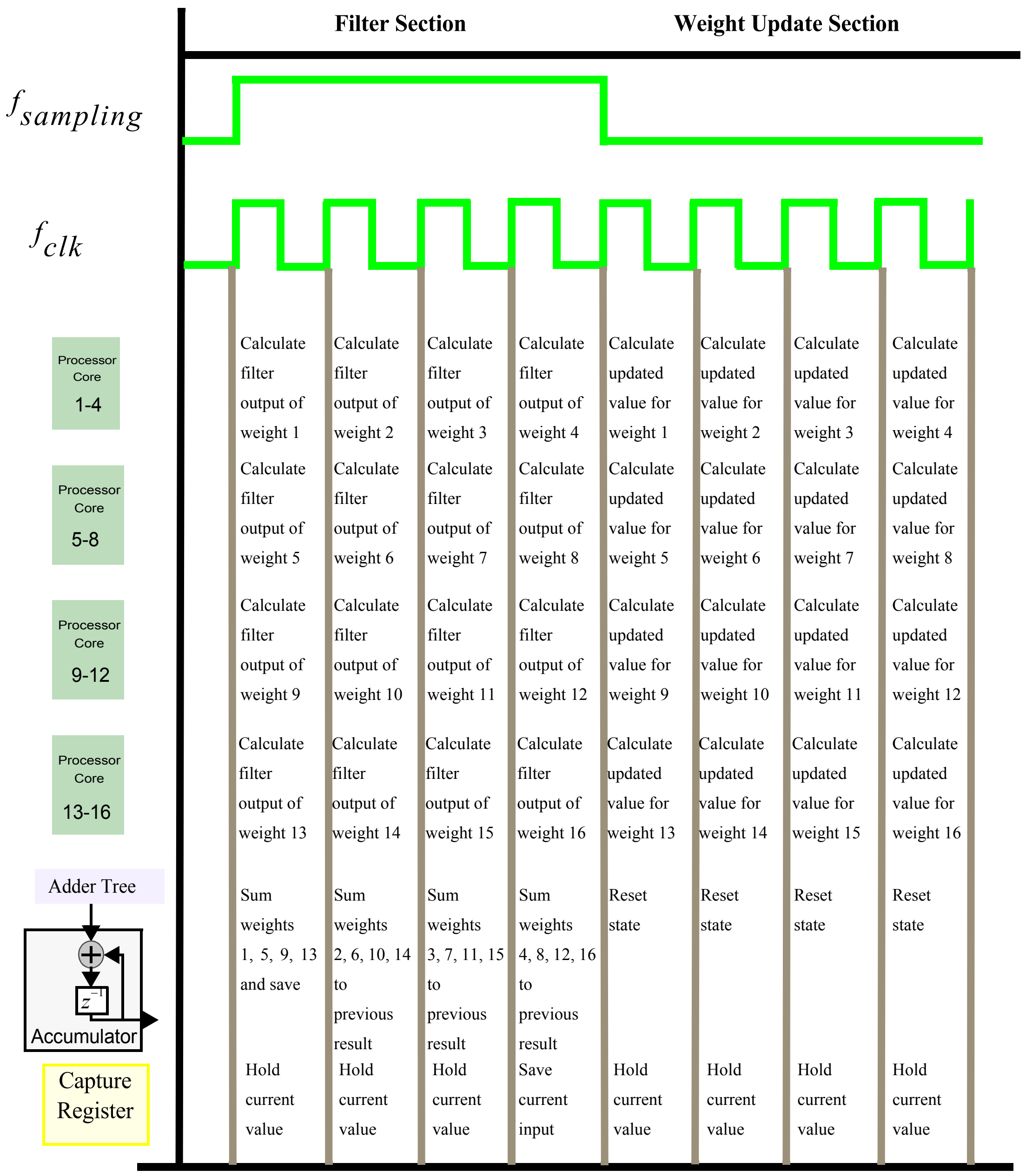


Figure 2.9: The Parallel - Serial LMS Filter Architecture: 4 PCs Computing 4 Coefficients, Forming a 16 Coefficient Filter in Total

The sample by sample operation of the parallel-serial LMS filter design is summarised in Figure 2.10, for the case of a 16 coefficient filter comprising 4 PCs computing 4 coefficients each. In order to be concise, the diagram does not take into account the pipeline stages of the design.



Filter Section
Weight Update Section
$f_{sampling}$
f_{clk}
Processor Core 1-4
Processor Core 5-8
Processor Core 9-12
Processor Core 13-16
Adder Tree
z^{-1}
Accumulator
Capture Register
Calculate filter output of weight 1
Calculate filter output of weight 2
Calculate filter output of weight 3
Calculate filter output of weight 4
Calculate updated value for weight 1
Calculate updated value for weight 2
Calculate updated value for weight 3
Calculate updated value for weight 4
Calculate filter output of weight 5
Calculate filter output of weight 6
Calculate filter output of weight 7
Calculate filter output of weight 8
Calculate updated value for weight 5
Calculate updated value for weight 6
Calculate updated value for weight 7
Calculate updated value for weight 8
Calculate filter output of weight 9
Calculate filter output of weight 10
Calculate filter output of weight 11
Calculate filter output of weight 12
Calculate updated value for weight 9
Calculate updated value for weight 10
Calculate updated value for weight 11
Calculate updated value for weight 12
Calculate filter output of weight 13
Calculate filter output of weight 14
Calculate filter output of weight 15
Calculate filter output of weight 16
Calculate updated value for weight 13
Calculate updated value for weight 14
Calculate updated value for weight 15
Calculate updated value for weight 16
Sum weights 1, 5, 9, 13 and save
Sum weights 2, 6, 10, 14 to previous result
Sum weights 3, 7, 11, 15 to previous result
Sum weights 4, 8, 12, 16 to previous result
Reset state
Reset state
Reset state
Reset state
Hold current value
Hold current value
Hold current value
Save current input
Hold current value
Hold current value
Hold current value
Hold current value

Figure 2.10: Detailed View of the Parallel - Serial LMS Filter Operation: 4 PCs Computing 4 Coefficients Each

2.5 The Normalised LMS Filter and Smith's Algorithm

The Adaptive LMS Compiler also features support for the Normalised LMS (NLMS) algorithm, with a choice of either parallel, serial or parallel-serial architecture and real or complex valued arithmetic. NLMS is perhaps the most popular variant of the LMS algorithm. It solves the practical problem of choosing a step size value which guarantees convergence, regardless of the changing properties of the input data.

As previously discussed, the step size bounds of the conventional LMS algorithm are given by [2.4].

$$\mu < \frac{1}{N \cdot E[\boldsymbol{x}^2(k)]} < \frac{2}{\lambda_{MAX}} \qquad [2.4]$$

where N is the tap length of the LMS filter and $E[\boldsymbol{x}^2(k)]$ is the expectation function performed upon the input data samples.

From this straightforward relationship, it is observed that in order to guarantee convergence using the LMS algorithm, prior knowledge of both (a) the filter length and (b) the magnitude values of the input data samples must be known. While (a) is user controlled, it is obvious that (b) can not be controlled in practical situations.

The NLMS algorithm guarantees convergence, provided that the normalised step size is bounded by $0 < \mu < 1$. The signal flow graph of the NLMS architecture is given in Figure 2.11, where the algorithm has been implemented in a parallel-serial type architecture.

The NLMS algorithm is summarised by [2.5], [2.6] and [2.7], in the complex form. The real valued equivalent is obtained by substituting the Hermitian operations for transpose operations and by removing the conjugation operations.

$$y(k) = \boldsymbol{w}^H(k)\boldsymbol{x}(k) \qquad [2.5]$$

$$e(k) = \boldsymbol{d}(k) - \boldsymbol{w}^H(k)\boldsymbol{x}(k) \qquad [2.6]$$

$$\boldsymbol{w}(k+1) = \boldsymbol{w}(k) + \frac{\mu}{\boldsymbol{x}^H(k)\boldsymbol{x}(k)}\boldsymbol{x}(k)e^*(k) \qquad [2.7]$$

where ^{H} is used to denote the Hermitian transpose (*i.e* the conjugate transpose) and * is used to denote the complex conjugate.

The hardware realisation of the NLMS algorithm in the parallel-serial form is given by Figure 2.11.

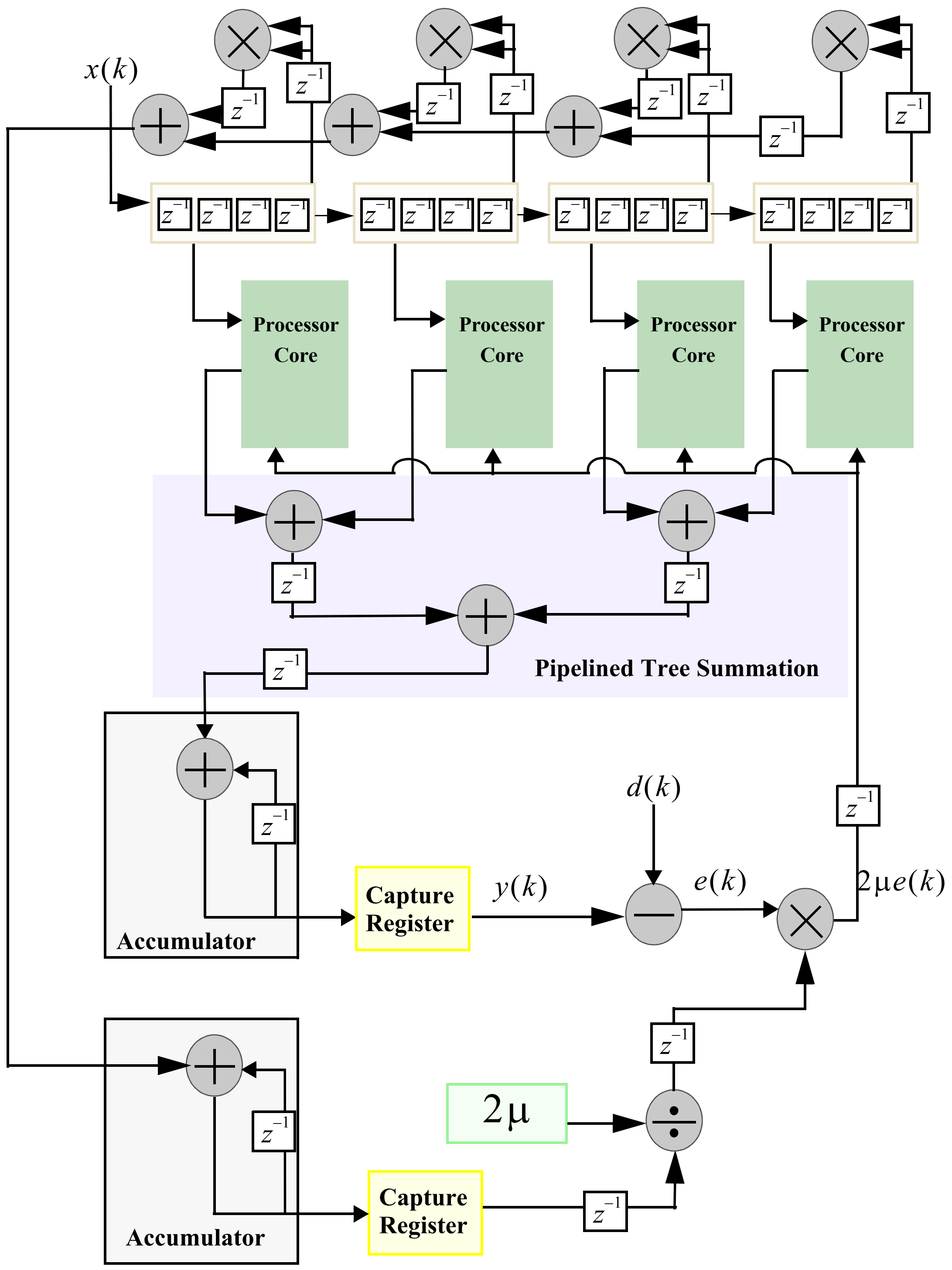


Figure 2.11: Parallel - Serial Implementation of the NLMS Algorithm

A key point regarding the NLMS algorithm is the requirement for not only multiply and add operations, but also for the operation of division. Fixed point division is well known to be numerically ill-conditioned, producing both very small and very large numbers, and hence requiring a very large dynamic range in order for accurate numerical representation to be achieved. If the requirement is not met, then there is the risk of overflow occurring, inducing severe numerical error. This is perhaps even more of an issue for feedback algorithms such as LMS, where the past values will influence the accuracy of future values.

The issue of dynamic range requirements arising from numerical ill conditioning in fixed point implementation is often met with one of two different solutions:

- Increase the dynamic range of the hardware by selecting a larger number of integer and fractional bits, and hence selecting a larger overall wordlength.
- Manipulate the algorithm/arithmetic operations further such that the numerically ill conditioned operation is either removed, or performed in a manner which has good numerical properties.

The first solution is perhaps obvious, however the issue is that the cost of the hardware implementation is somewhat linked to the wordlength that is specified. For this reason, the second solution is employed. For the case of real valued arithmetic, it is not possible to manipulate the algorithm in such a manner that this can be achieved, however as shall be explained, it is only for the case of complex valued arithmetic that the issue of numerical ill - conditioning is a particular problem.

Consider the conventional “pen and paper” formula for complex division, arising from rudimentary algebra, [2.8].

$$z = a + jb = \frac{c + jd}{e + jf} = \frac{ce + df}{e^2 + f^2} + j\frac{de - cf}{e^2 + f^2} \qquad [2.8]$$

With reference to [2.7], consider the denominator of the complex division operation, $\boldsymbol{x}^H(k)\boldsymbol{x}(k)$. Due to the use of Hermitian operations, it is possible to reduce the complex multiplication to a simpler form:

$$\boldsymbol{x}^H(k)\boldsymbol{x}(k) = (a+jb)(c-jd) = a^2+jbc-jad+b^2 \qquad [2.9]$$

noting that $a = c$ and $b = -d$, we then obtain:

$$\boldsymbol{x}^H(k)\boldsymbol{x}(k) = (a+jb)(c-jd) = a^2+b^2 \qquad [2.10]$$

As $\boldsymbol{x}^H(k)\boldsymbol{x}(k)$ is achieved via the sum of individual $\boldsymbol{x}(k)^2$ samples used within each tap of the NLMS filter (see Fig.2.11), the value at the output of the summation will rise as the number of taps of the NLMS filter increases. Hence a greater number of integer bits will be required with an increasing number of taps in the NLMS filter.

In addition to the formula presented in [2.8] for performing complex division, there also exists a method of computation known as Smith's algorithm, [8]. In this method, either e/e or f/f is extracted from the denominator, resulting in an algorithm which does not require e^2 or f^2 to be calculated. By extracting e/e from the denominator, the complex division can be rewritten in the form given by [2.11].

$$z = a+jb = \begin{cases} \dfrac{c+d(f/e)}{e+f(f/e)} + j\dfrac{d-c(f/e)}{e+f(f/e)} \ldots(|c| \geq |d|) \\ \dfrac{c+d(f/e)}{e+f(f/e)} + j\dfrac{d-c(f/e)}{e+f(f/e)} \ldots(|c| < |d|) \end{cases} \qquad [2.11]$$

From a first glance, it might appear that Smith's algorithm has an increased computational complexity. However, by performing appropriate manipulation of the algorithm, an implementation can be achieved which requires fewer dedicated arithmetic components than the original technique.

Firstly, it can be noted that in the separate formulae given in parenthesis in [2.11], the arithmetic operations remain constant, only the values passed to each operation change.

Hence, the equation may be re-written in a more general form, [2.12].

$$z = a + jb = \frac{add1 + mult1((div1)/(div2))}{add2 + mult3((div5)/(div6))} + j\frac{subtract1 - mult2((div3)/(div4))}{add3 + mult4((div7)/(div8))} \qquad [2.12]$$

A 2:1 multiplexer can be applied to each of the generic variables given in [2.12], where the SELECT signal is controlled by a logical test to determine whether $|c| \geq |d|$ or $|c| < |d|$. The values which each variable should take with respect to the SELECT signal are listed in Table 2.11.

Variable	SELECT=0	SELECT=1
ADD1	c	d
MULT1	d	c
DIV1	f	e
DIV2	e	f
SUBTRACT1	d	c
MULT2	c	d
DIV3	f	e
DIV4	e	f
ADD2	e	f
MULT3	f	e
DIV5	f	e
DIV6	e	f
ADD3	e	f
MULT4	f	e
DIV7	f	e
DIV8	e	f

Table 1: List of Variables Against SELECT Input

From Table 2.11, it is apparent that there is a degree of commonality between the values which each variable should take in each case. In fact, the number of variables can be reduced to four, (as opposed to the sixteen listed), as illustrated in Table 2.

Variable	SELECT = 0	SELECT =1
V1	c	d
V2	d	c
V3	f	e
V4	e	f

Table 2: Revised Values Passed in Smith's Algorithm Multiplexer Arrangement

The previously given equation for Smith's algorithm may then be rewritten as a result of the commonality identified in the algorithm, [2.13].

$$z = a + jb = \frac{V1 + V2((V3)/(V4))}{V4 + V3((V3)/(V4))} + j\frac{V2 + V1((V3)/(V4))}{V4 + V3((V3)/(V4))} \qquad [2.13]$$

By writing the equation in this form, we can note further commonality in the arithmetic operations performed, allowing for the complex division operation to be performed using 3 divides, 3 multiplications, 3 additions, four 2:1 multiplexers and a low cost logic test, (Figure 2.12). This is of lower cost than the original algorithm, (which requires 6 multiplications, 3 additions and two divides). Hence, the resulting implementation has better numerical properties than the original calculation, and can be implemented at a lower hardware cost.

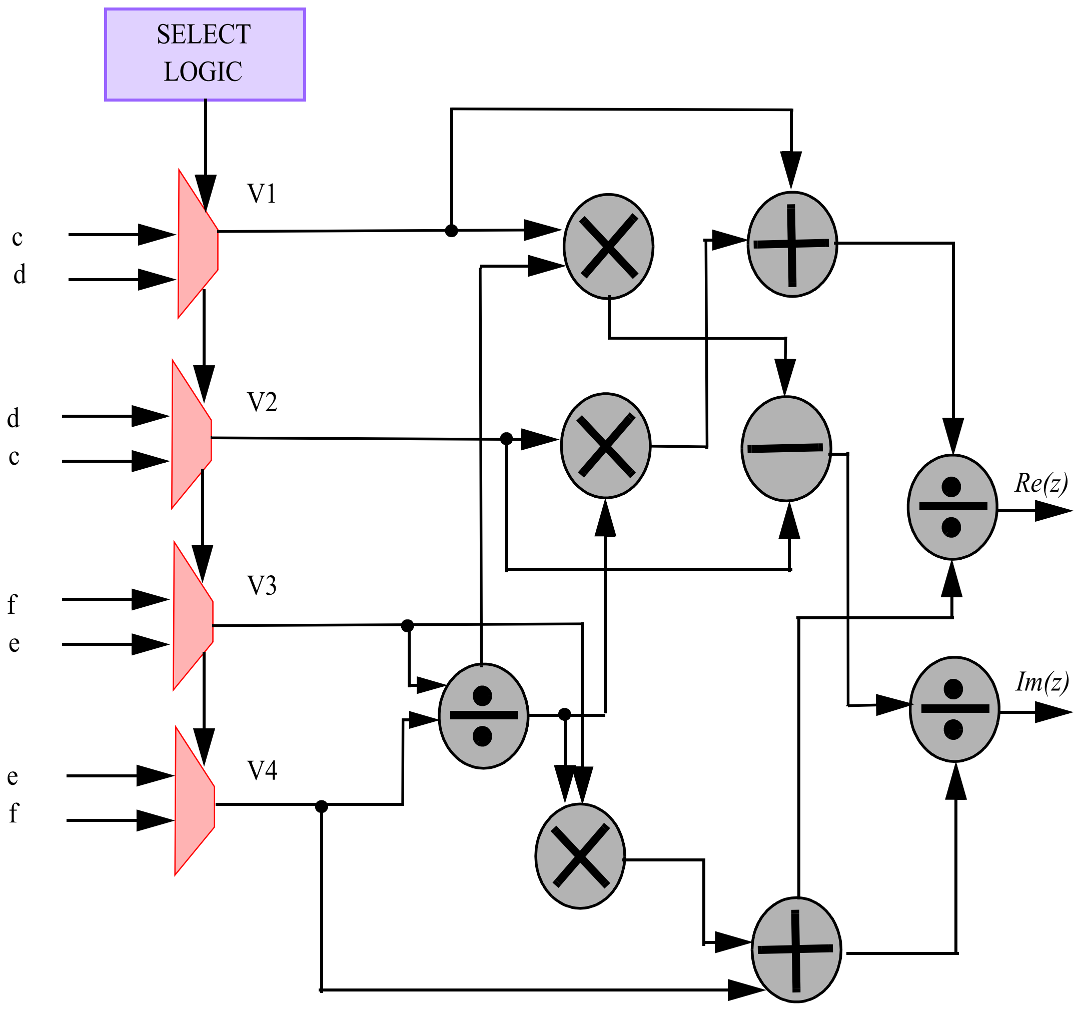


Figure 2.12: Hardware Resulting from Implementation of Smith's Algorithm

2.6 Sample Rate and Hardware Cost Trade Off

In each of the various LMS architectures, there is a fundamental trade off to be made between the achievable sample rate of the design, and the hardware cost. As the folding factor (i.e. the degree of serialisation) is increased, then the work which must be carried out by the hardware is increased, and so the sampling rate must be decreased with respect to the maximum clock rate, in order to allow the serialised hardware a sufficient number of clock cycles to compute the overall result.

Although the sample rate restrictions have been highlighted in previous sections, they have not been given a comprehensive study. In this section, the corresponding sample rate equation for each of the various architectures is given definitively.

In the fully parallel case, for both the LMS and NLMS algorithms (real valued and complex valued), there is no serialisation imparted on the design, and hence the sample rate of the input data can match the rate at which each element of the LMS architecture is clocked at, [2.14].

$$f_{sampling} = f_{clk} \qquad [2.14]$$

For the case of the fully serial LMS/NLMS filter architecture, the number of clock cycles taken to compute the result is a function of the number of coefficients employed in the design, which is multiplied by two seeing as a single PC component is employed for both the filtering and weight update operation. A fixed overhead is added due to the pipelining employed in the design, giving the sample rate equation for the fully serial LMS/NLMS algorithm [2.15].

$$f_{sampling} = f_{clk}/(2.N) + 6 \qquad [2.15]$$

For the case of the complex valued LMS/NLMS algorithm, complex valued multipliers mean that an extra pipeline delay is incurred of three clock cycles, giving the sample rate equations for the fully serial complex LMS/NLMS algorithm by [2.16].

$$f_{sampling} = f_{clk}/(2.N) + 9 \qquad [2.16]$$

When the parallel-serial design is considered, then there is a further clock cycle overhead incurred, due to the pipelined adder tree type structure used to compute the sum of the output of the individual serialised PC units. Hence, the sample rate equation for the real valued parallel-serial LMS/NLMS algorithm is given by [2.17], while the sample rate equation for the complex valued parallel-serial LMS/NLMS algorithm is given by [2.18].

$$f_{sampling} = f_{clk}/(2.N) + ceil(\log 2(PC)) + 6 \qquad [2.17]$$

where *ceil()* is the ceiling function (rounding up to the nearest integer) and *log2* represents the base two logarithm.

$$f_{sampling} = f_{clk}/(2.N) + ceil(\log2(PC)) + 9 \qquad [2.18]$$

3 Numerical Performance Analysis

The compiler has been tested for the numerical accuracy of the results, in order to ensure that the hardware implementation of the design matches the original algorithm given in the literature. To achieve this, the fixed point Adaptive LMS Compiler has been tested against a floating point "Golden Reference" design featured in The MathWorks Simulink tool. Results are given for several different parameter options of the Adaptive LMS Compiler which confirm the correct operation of the implementation.

The "Golden Reference" used is the LMS Filter block found in the DSP System Toolbox blockset. In order to compare the results within Simulink, a Xilinx System Generator "Black Box" block is used, which allows the user to simulate custom HDL within Simulink by invoking Mentor Graphics Modelsim cosimulation.

For each test scenario, a Mean Squared Error (MSE) performance analysis has been carried out. In such an analysis, several simulations of different random input data are carried out, where the parameters of the LMS IP core are fixed. The sample by sample error is squared, and then the mean is taken of each sample point using the data obtained from different simulations. The figures display both the MSE results, and the fixed and floating point samples overlayed upon one another. For each filter architecture, results are given for both the case of a 20 coefficient filter and that of a 50 coefficient filter, in order to give a greater confidence in the validity of the results.

$$MSE = E[(\boldsymbol{e}_{fp} - \boldsymbol{e}_{fxp})^2] \qquad [3.1]$$

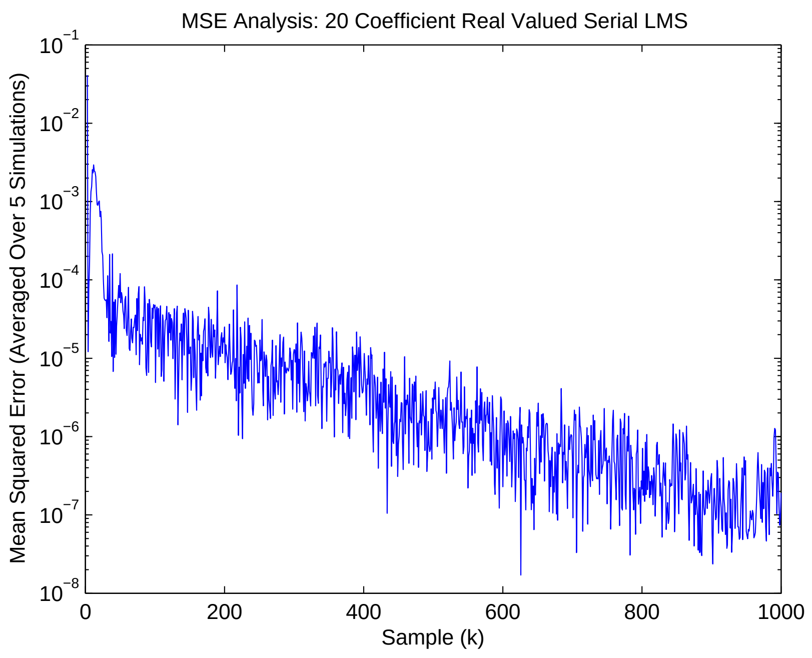
where:

$\boldsymbol{e}_{fp}$ = floating point error signal.

$\boldsymbol{e}_{fxp}$ = fixed point error signal.

3.1 Real Valued Arithmetic Serial LMS Filter Results

Figure 3.1 and Figure 3.2 show the numerical simulation results for the case of a 20 coefficient filter and a 50 coefficient filter. It can be observed that as the relative magnitude of the error signal decreases, there is a closer match between the results from the fixed point adaptive filter compiler and the floating point golden reference. This can be explained due to rounding/saturation noise in the fixed point implementation causing slightly different results to appear. The small error between the fixed point Adaptive LMS Compiler and the floating point golden reference confirms the correct operation of the generated HDL.



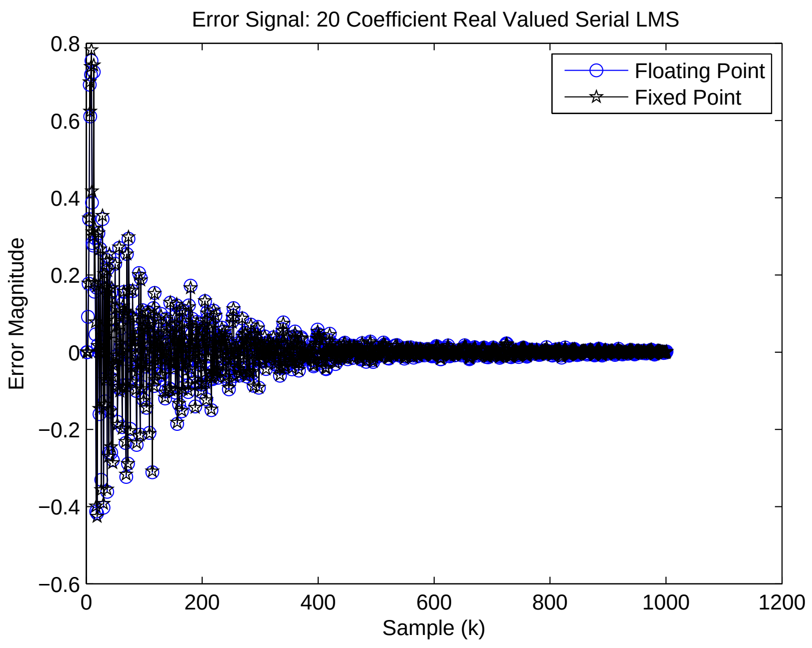
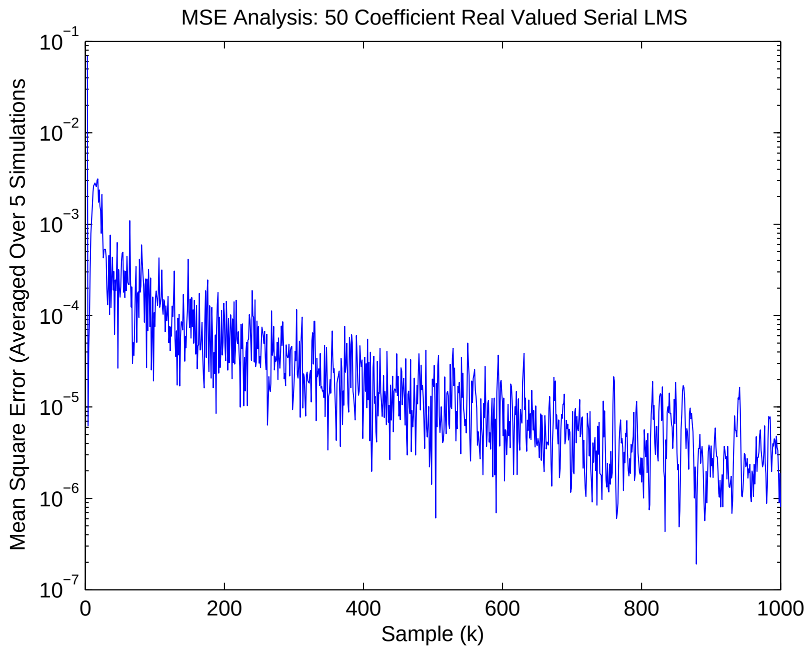


Figure 3.1: Test Case 1: 20 Coefficient Real Valued Serial LMS



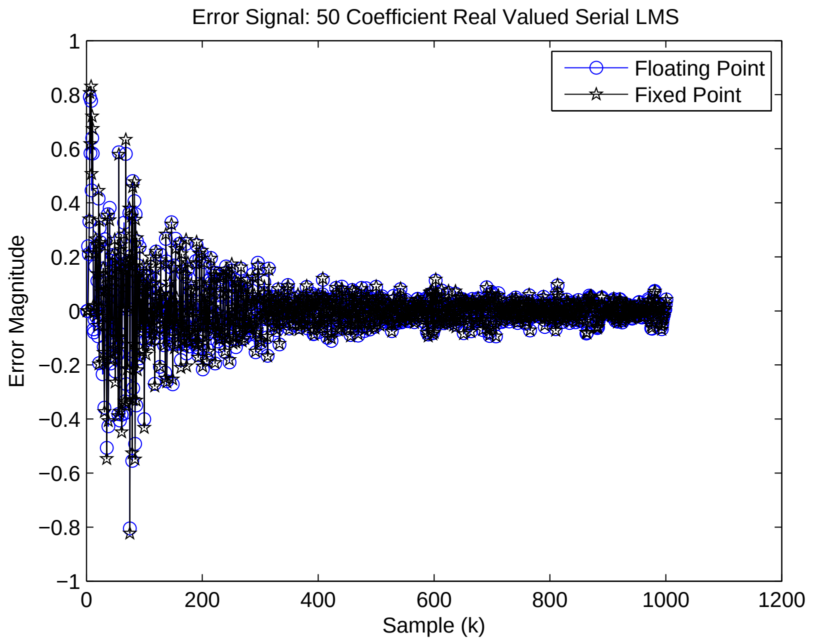


Figure 3.2: Test Case 2: 50 Coefficient Real Valued Serial LMS

The results for the remaining architectures all follow a similar trend. For the purpose of completeness, these results are contained in Section 13: Appendix A.

4 Hardware Implementation Results

In this section, results from the implementation of the Adaptive LMS Compiler using the Xilinx ISE synthesis and implementation tool are given. A Xilinx Virtex 6 XCV0605 FPGA is targeted in all cases. The results show the relative trade-offs which can be made when using each of the various architectures. For readers unfamiliar with the various resources targeted on Xilinx FPGAs or FPGAs in general, a short introductory section is presented prior to the results.

4.1 FPGA Technology Mapping

When targeting an HDL design to an FPGA, there are several metrics which can be used to evaluate the overall performance and cost of the design. Firstly, it is common to evaluate the overall performance of the design via the maximum critical path. Taking the reciprocal of this value allows the hardware engineer to work out a value (within tolerance) for the maximum achievable clock frequency of the design.

It is important to realise however that this is not necessarily the same as the maximum sampling frequency of the incoming data. If parts of the design are serialized, then this will limit the sampling frequency at the input. The design will require several clock cycles in order to compute a single sample, therefore a reduction by an integer factor is introduced. All of the results in this section are therefore presented in terms of sampling frequency, although the reader can use the formulae presented in Section 2.6 to compute the given clock frequency.

The Adaptive LMS Compiler allows the user to impart varying degrees of serialization on the design, which allows for trade offs to be made between the resources utilized on the device and the overall sampling frequency. There are several resources in question which can be examined.

In the following analysis, three basic units are examined - LUT utilization, DSP48 utilization and slice register utilization. The LUT is a basic building block of the FPGA fabric. Any digital function can be represented by the interconnection of LUTS. By investigating how the LUT cost increases or decreases for given configurations of the LMS compiler, it is possible to evaluate the cost of the control logic used to determine how data is passed around the filter architecture.

In order to investigate the hardware cost in terms of multiply-add operations, the DSP48 utilization can be investigated. DSP48s are high speed multiply-add resources present on Xilinx FPGA for the purpose of performing high speed arithmetic operations. Targeting these resources is critical to achieving a high speed, low power design. Such devices are however, limited in number on the FPGA. By configuring the various serialization options on the Adaptive LMS Compiler, it is possible to time-share a single DSP48 between multiple coefficients of the design. Examining the actual implementation results verifies that this is the case.

The final basic hardware resource under investigation is the slice register utilization. Slice registers are small memory elements which can be interconnected to form shift registers. For designs where some parts of the architecture are serialized, data needs to be held for several clock cycles. It is therefore important to investigate the slice register utilization in order to quantify the cost associated with serialization.

4.2 Real Valued Arithmetic Results

4.2.1 Sampling Frequency Results

Firstly, the sampling frequency achieved under varying degrees of serialization is quantified. As expected, the highest possible sampling frequency rates are achieved under fully parallel operation. The parallel-serial architecture allows trade-offs to be made between the overall number of PCs and the coefficients per PC, therefore there is a degree of variation in the results that can be obtained. It can be observed that for the fully serial architecture, the sampling frequency continuously decreases as the number of coefficients is increased, due to the increasing number of clock cycles required to compute a single output sample. In the parallel-serial case, we can observe that the sampling frequency is relatively constant with increasing PC, but decreases as more coefficients are computed per core.

It is interesting to note that the sampling frequency results obtained from the NLMS configuration are in general lower than those obtained from the LMS configuration. The NLMS architecture features division operations which incur a high critical path, therefore limiting the maximum sampling frequencies which can be obtained.

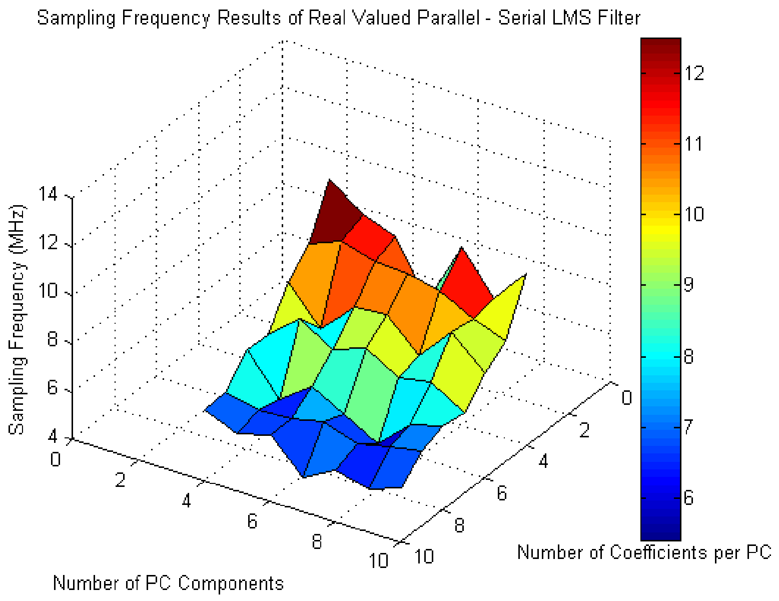


Figure 4.1: Sampling Frequency Results of Real Valued Parallel - Serial LMS Architecture

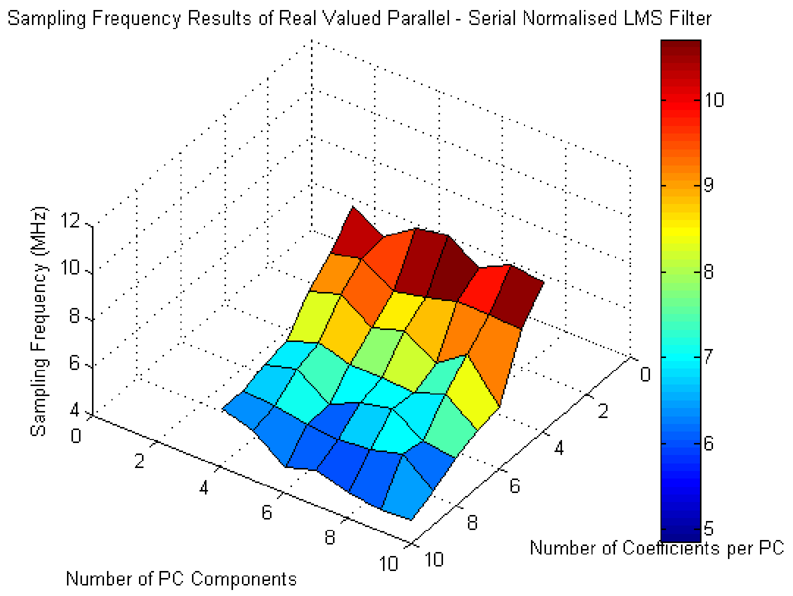


Figure 4.2: Sampling Frequency Results of Real Valued Parallel - Serial NLMS Architecture

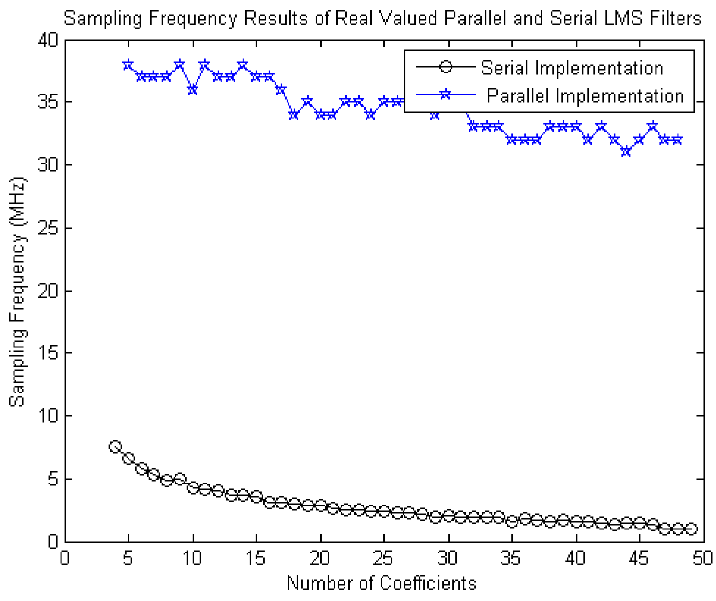


Figure 4.3: Sampling Frequency Results of Real Valued Parallel and Serial LMS Architectures

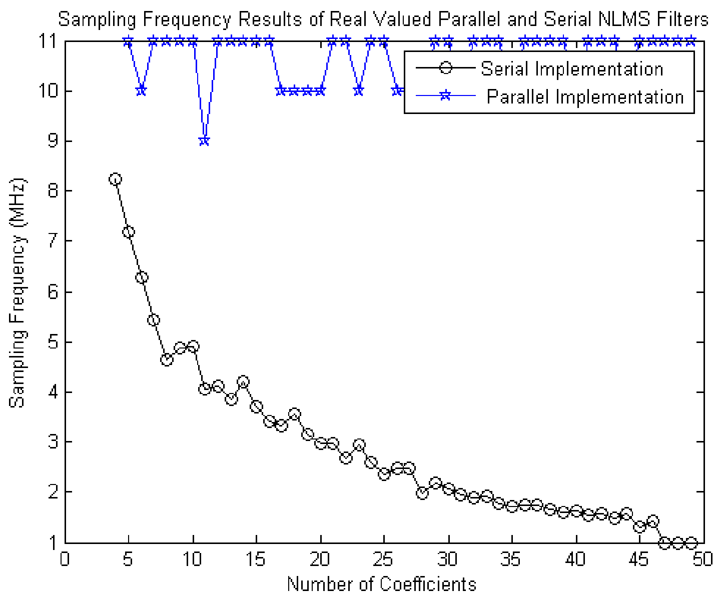


Figure 4.4: Sampling Frequency Results of Real Valued Parallel and Serial NLMS Architectures

4.2.2 DSP48 Utilization

As previously noted, examining the DSP48 utilization allows the correct operation of the Adaptive LMS Compiler to be verified. For the parallel-serial architecture, the multipliers are time-shared and therefore there should be a constant DSP48 cost when the number of PCs is fixed at a set value. This is observed in Figure 4.7 and Figure 4.8, which show a linear increase in DSP48S with increasing PCs. It is also observed that for the parallel filter architecture, there is a linear increase of DSP48 utilization as the number of coefficients increases.

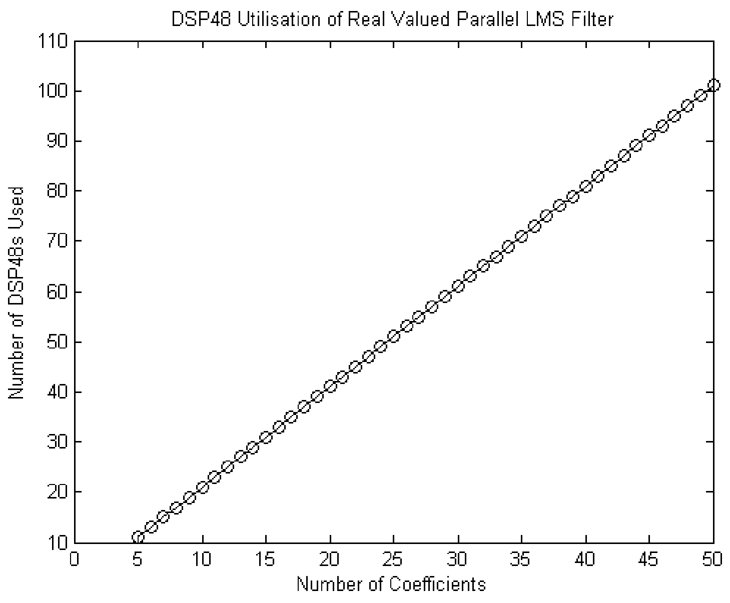


Figure 4.5: DSP48 Utilisation of Real Valued Parallel LMS Architecture

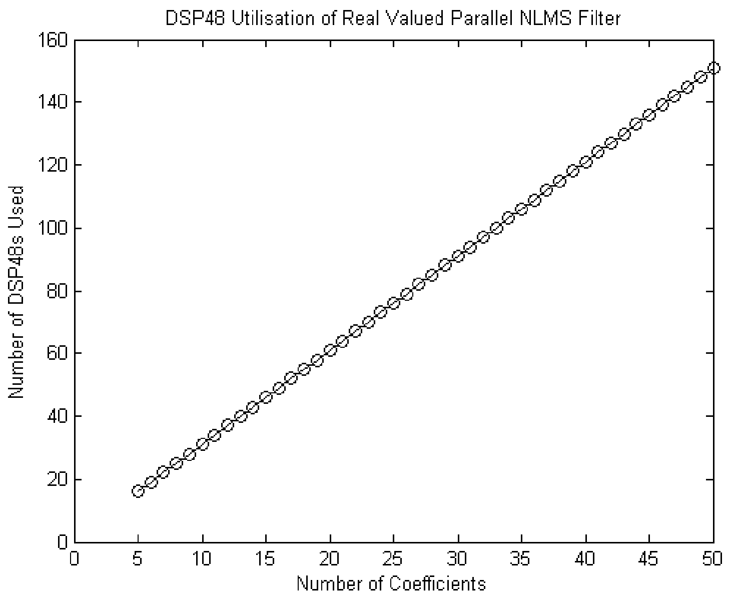


Figure 4.6: DSP48 Utilisation of Real Valued Parallel NLMS Architecture

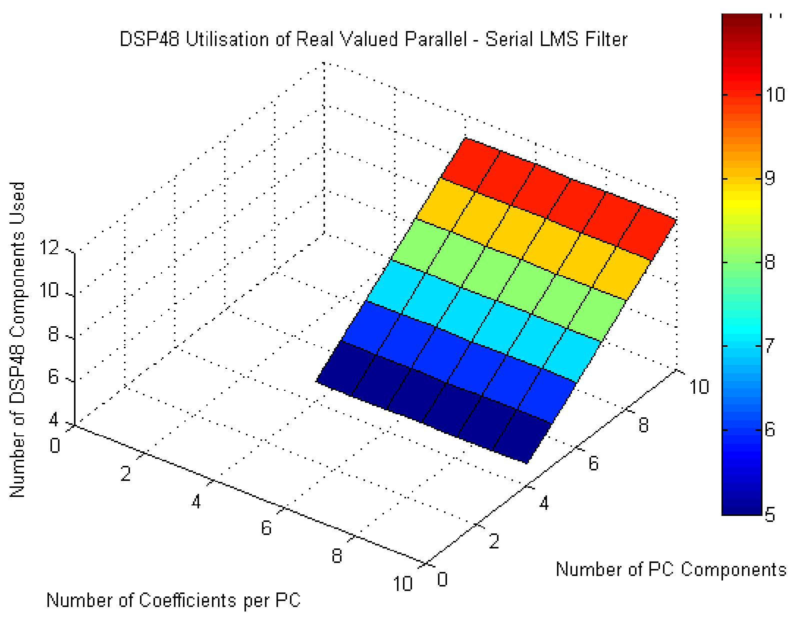


Figure 4.7: DSP48 Utilisation of Real Valued Parallel - Serial LMS Architecture

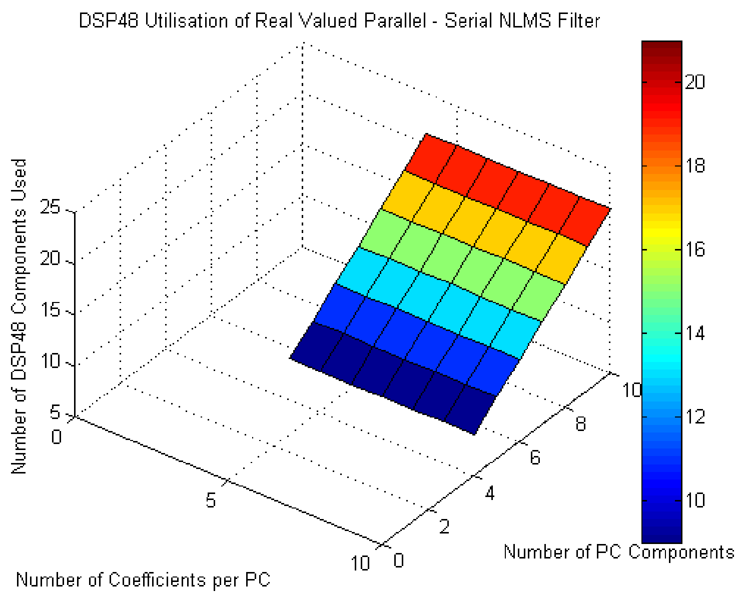


Figure 4.8: DSP48 Utilisation of Real Valued Parallel - Serial NLMS Architecture

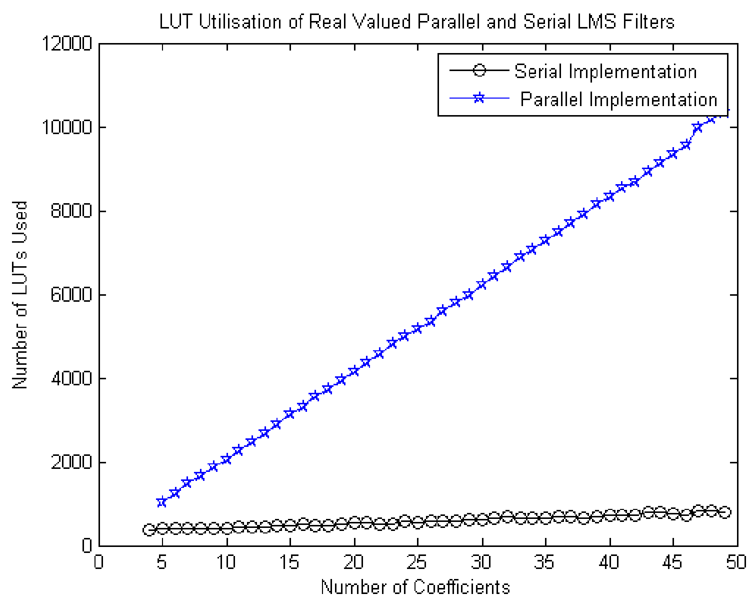


Figure 4.9: LUT Utilisation of Real Valued Parallel and Serial LMS Architectures

4.2.3 LUT Utilization

As LUTS are general purpose logic elements used in all the architectures for various functions, the cost scales up with increasing number of overall coefficients. The LUT cost is far greater in the NLMS architecture when compared to the LMS architecture due to the complex division logic.

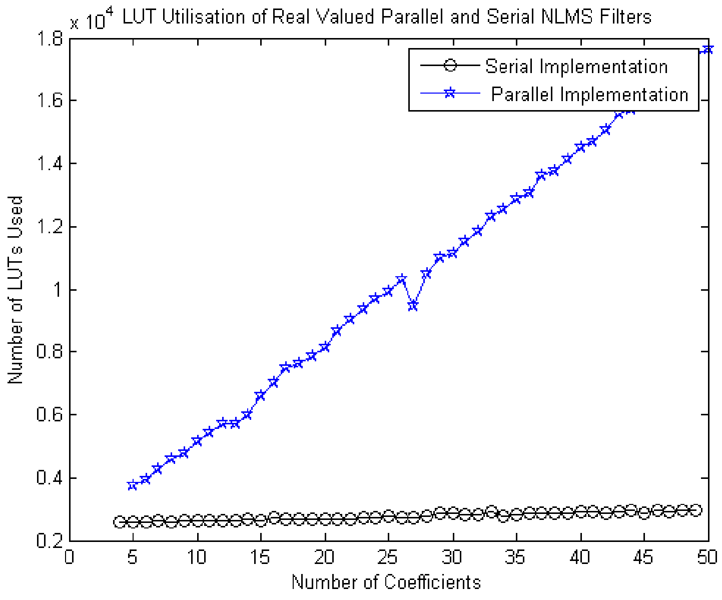


Figure 4.10: LUT Utilisation of Real Valued Parallel and Serial NLMS Architectures

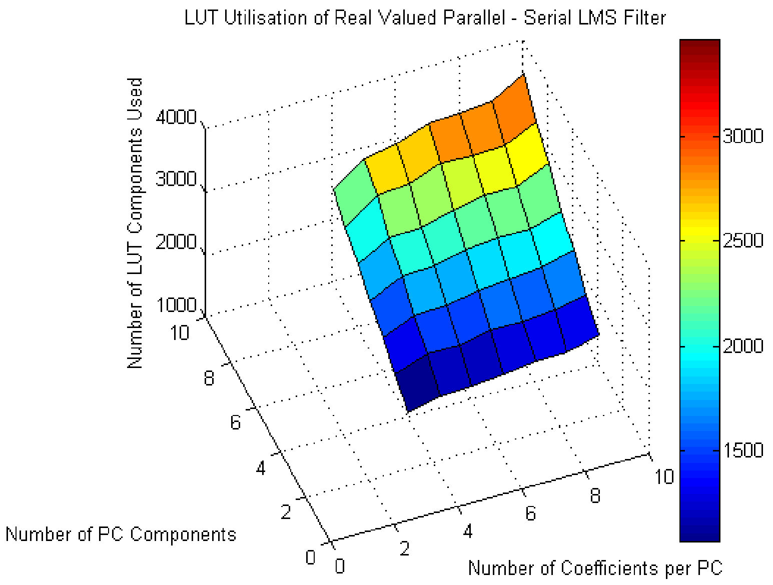


Figure 4.11: LUT Utilisation of Real Valued Parallel - Serial LMS Architecture

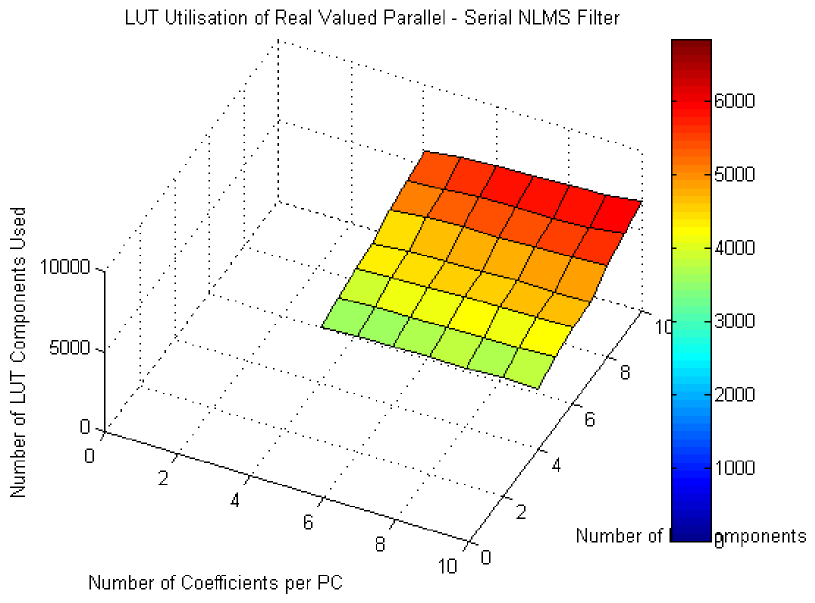


Figure 4.12: LUT Utilisation of Real Valued Parallel - Serial NLMS Architecture

4.2.4 Slice Register Utilization

The slice register utilization increases across all the architectures with an increasing number of coefficients, however it is of particular interest to note this resource cost for the serial and parallel-serial arhitectures. For these architectures, extra registers are required to store samples between computation. It is interesting to observe that the fully serial architecture is actually more costly to implement than the fully parallel architecture up to around 20 coefficients. This is due to the fixed overhead of pipeline registers which are inserted into the serial datapath to minimize the critical path.

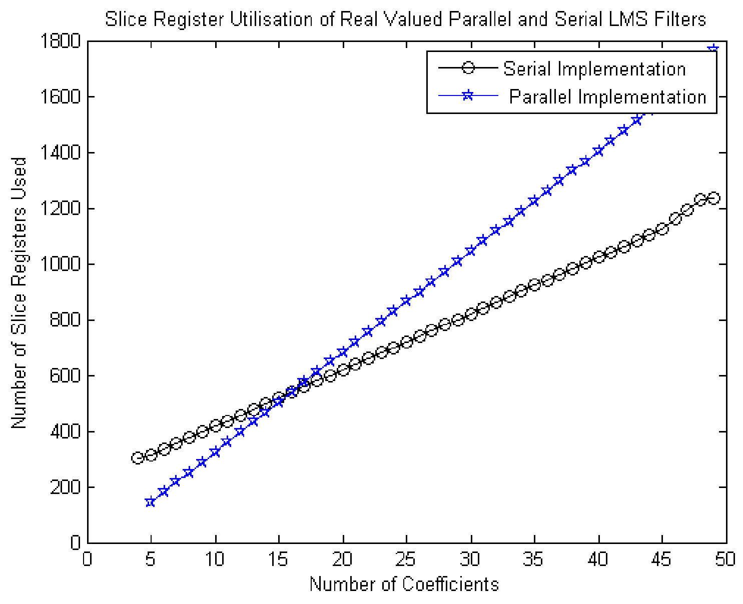


Figure 4.13: Slice Register Utilisation of Real Valued Parallel and Serial LMS Architectures

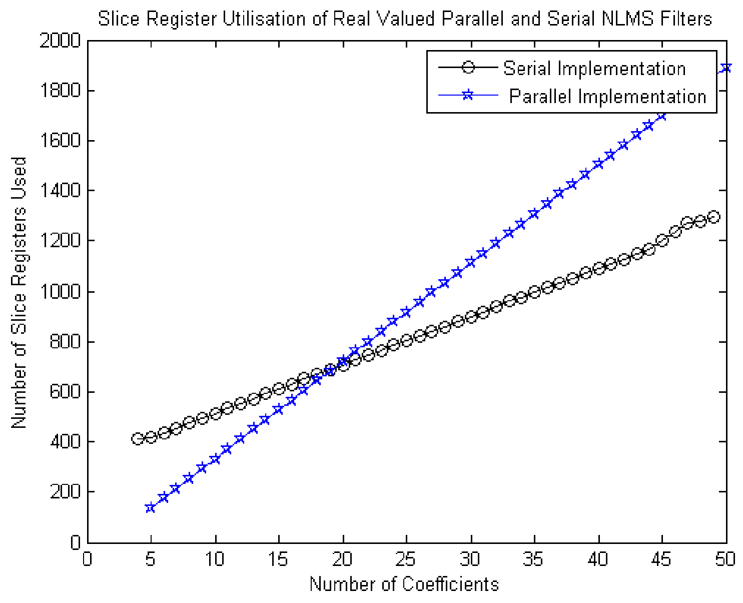


Figure 4.14: Slice Register Utilisation of Real Valued Parallel and Serial NLMS Architectures

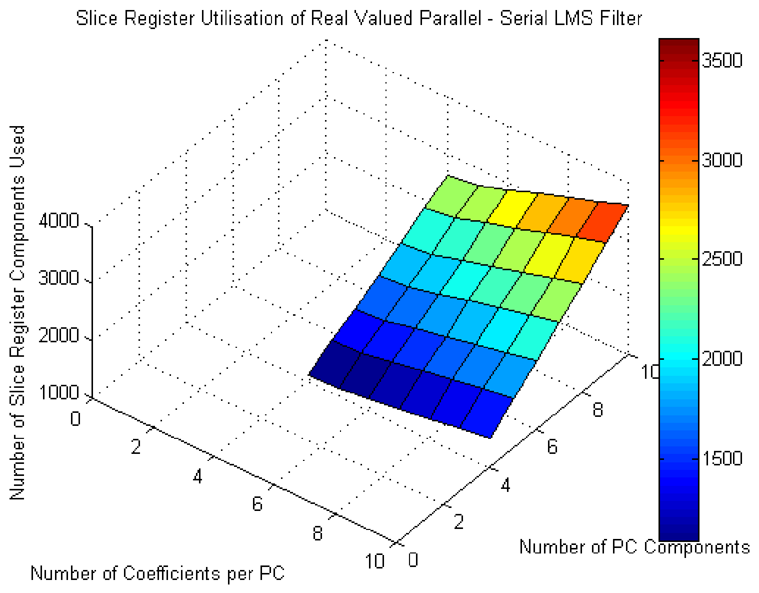


Figure 4.15: Slice Register Utilisation of Real Valued Parallel - Serial LMS Architecture

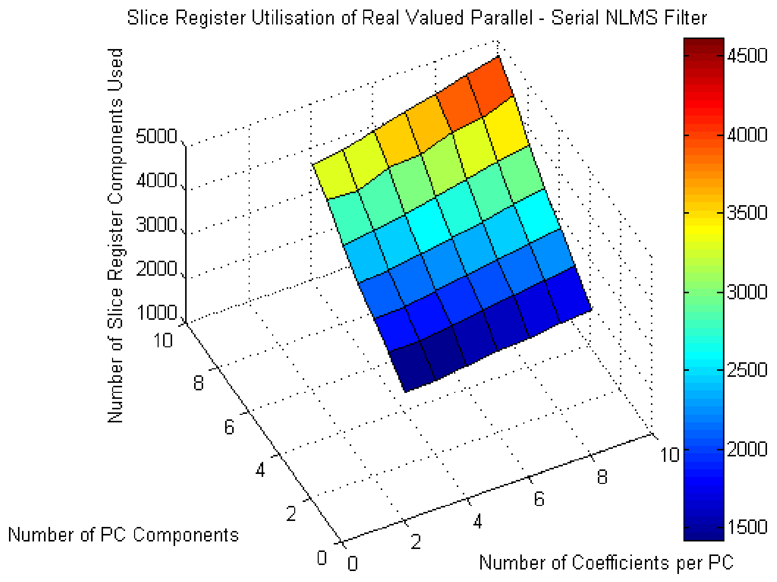


Figure 4.16: Slice Register Utilisation of Real Valued Parallel - Serial NLMS Architecture

4.3 Complex Valued Arithmetic Results

4.3.1 Sampling Frequency Results

In general, the sampling frequencies obtained when complex arithmetic is employed are lower than those obtained for real valued arithmetic. Complex valued arithmetic operations are achieved as a compound of several real valued operations, therefore the critical path is increased. Extra pipeline registers were inserted in order to counter this, however as shown by the results, further pipelining work is required to obtain the throughput achieved in the real-valued arithmetic architecture.

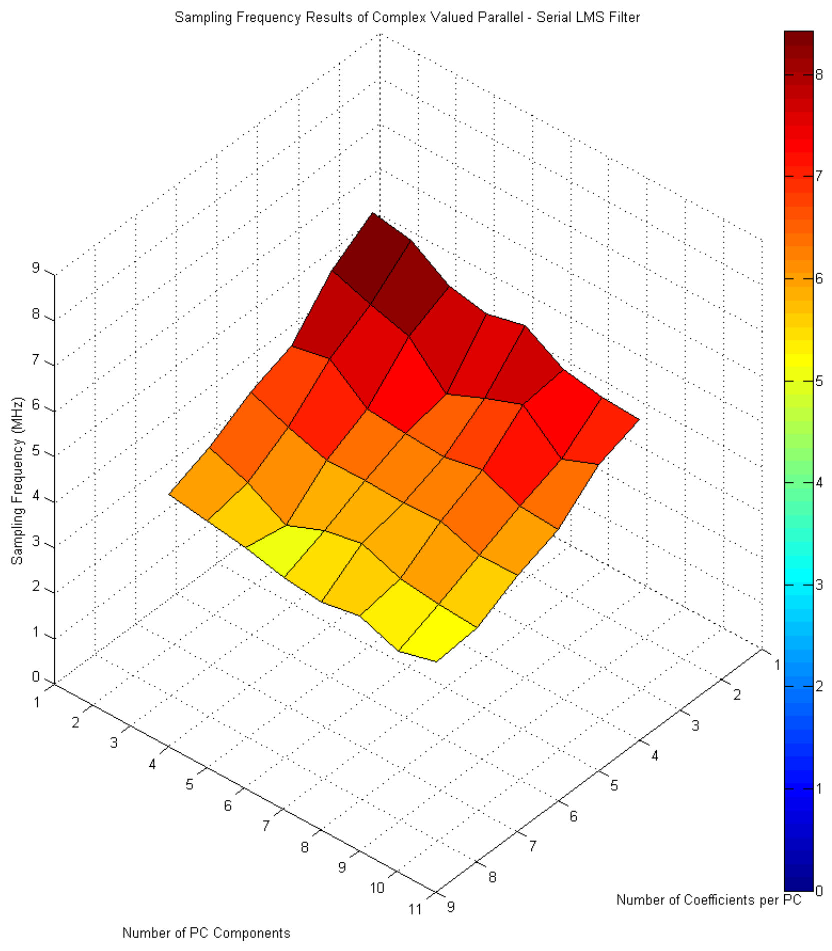


Figure 4.17: Sampling Frequency Results of Complex Valued Parallel - Serial LMS Architecture

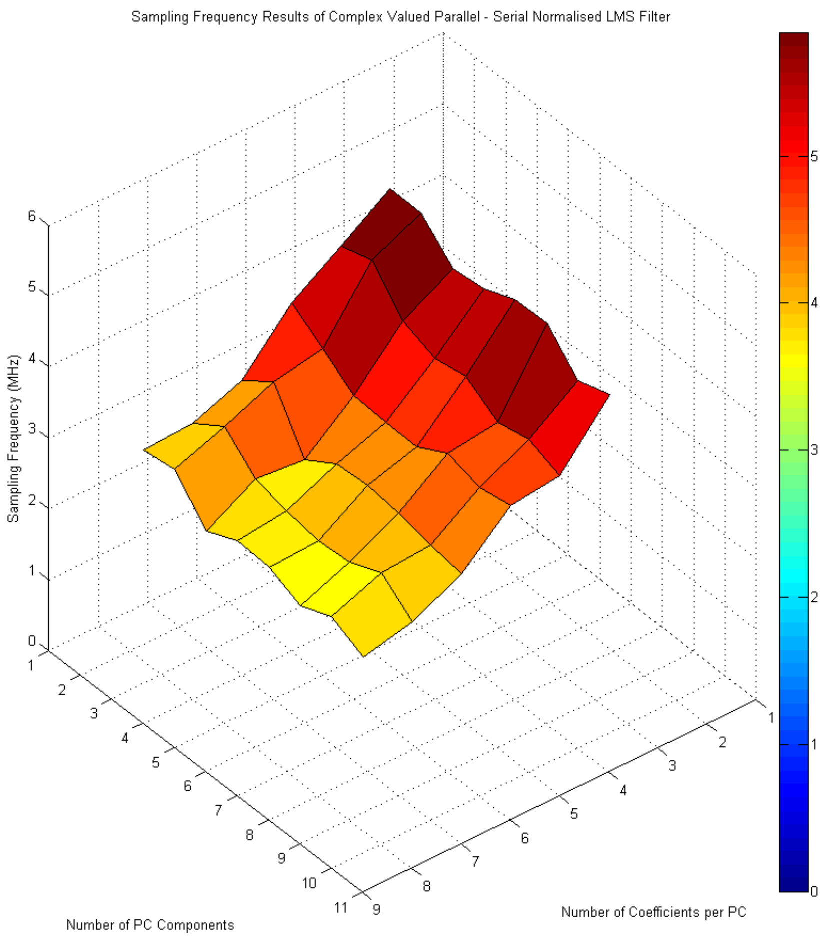


Figure 4.18: Sampling Frequency Results of Complex Valued Parallel - Serial NLMS Architecture

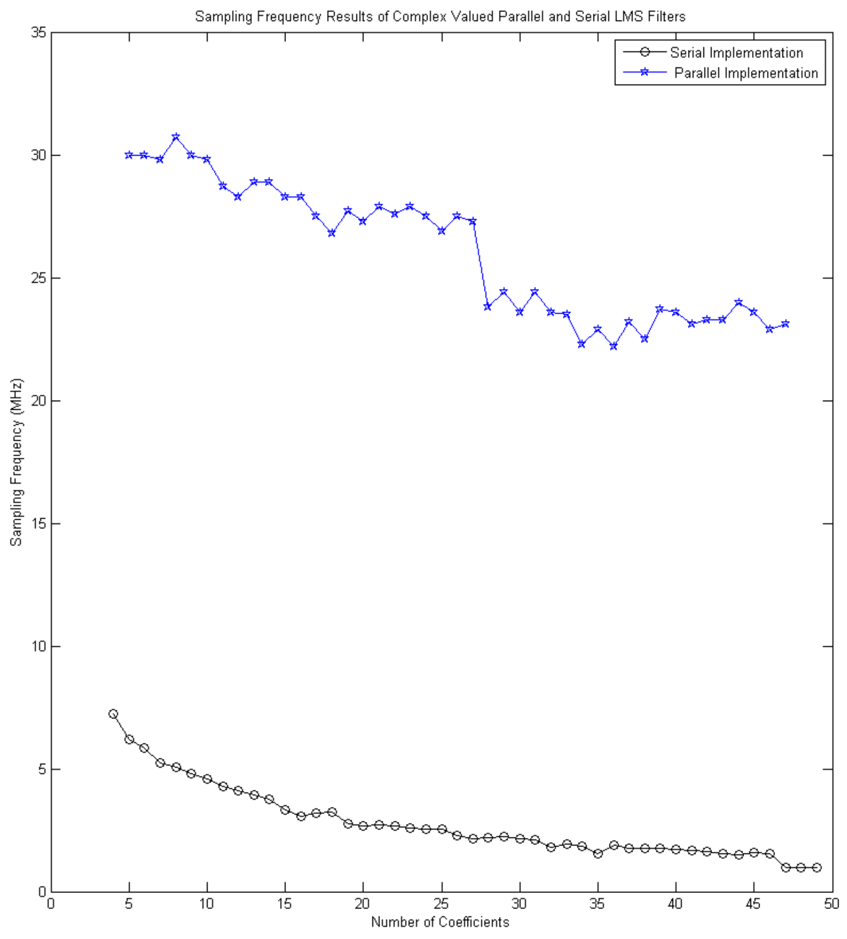


Figure 4.19: Sampling Frequency Results of Complex Valued Parallel and Serial LMS Architectures

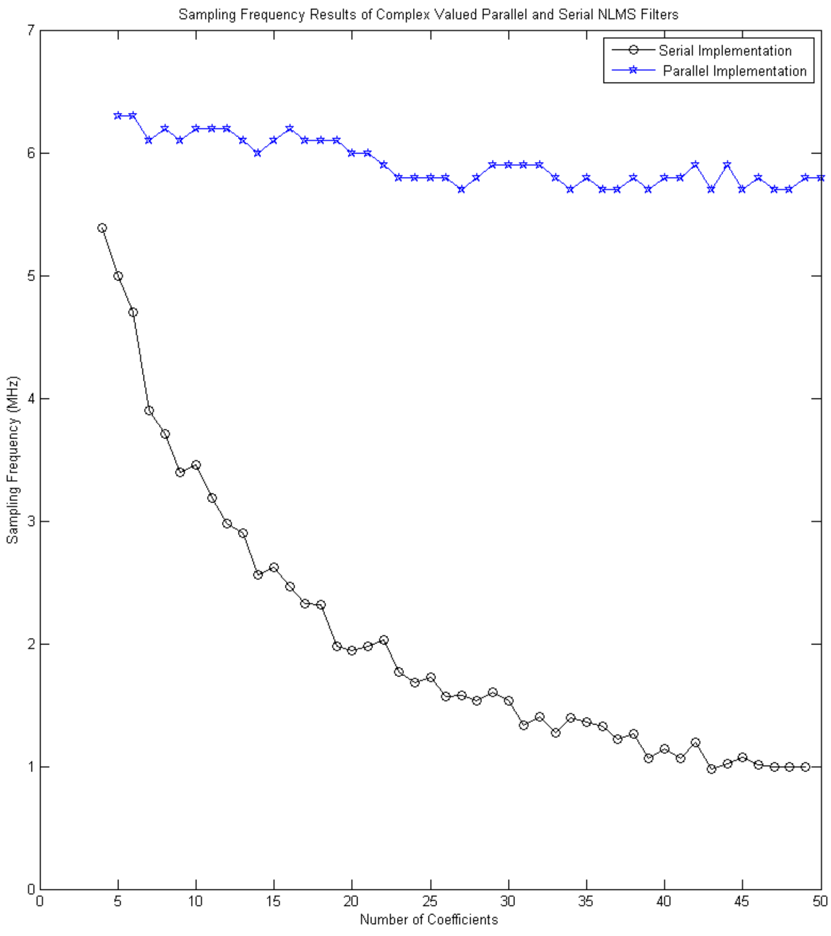


Figure 4.20: Sampling Frequency Results of Complex Valued Parallel and Serial NLMS Architectures

4.3.2 DSP48 Utilization

Due to complex valued arithemetic being formed as a compound of several real valued operations, the DSP48 utilization is in general greater for the complex valued case when compared to the real valued arithmetic equivalent. The general relationship of linear scaling with increasing parallelization is maintained however.

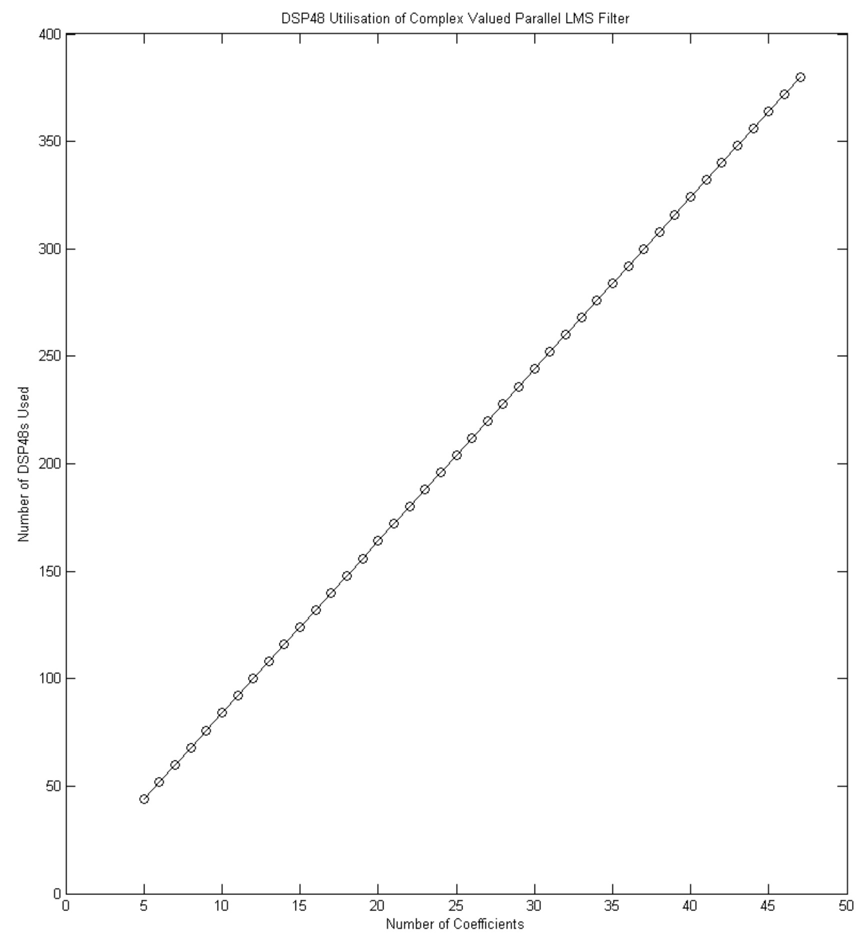


Figure 4.21: DSP48 Utilisation of Complex Valued Parallel LMS Architecture

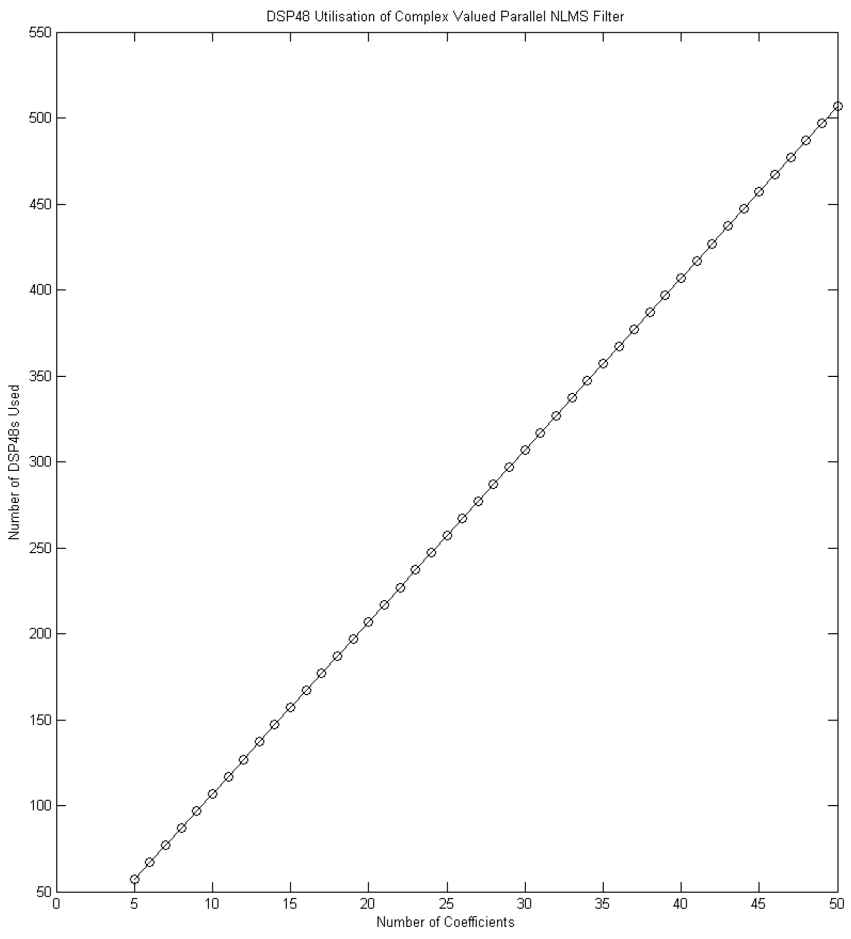


Figure 4.22: DSP48 Utilisation of Complex Valued Parallel NLMS Architecture

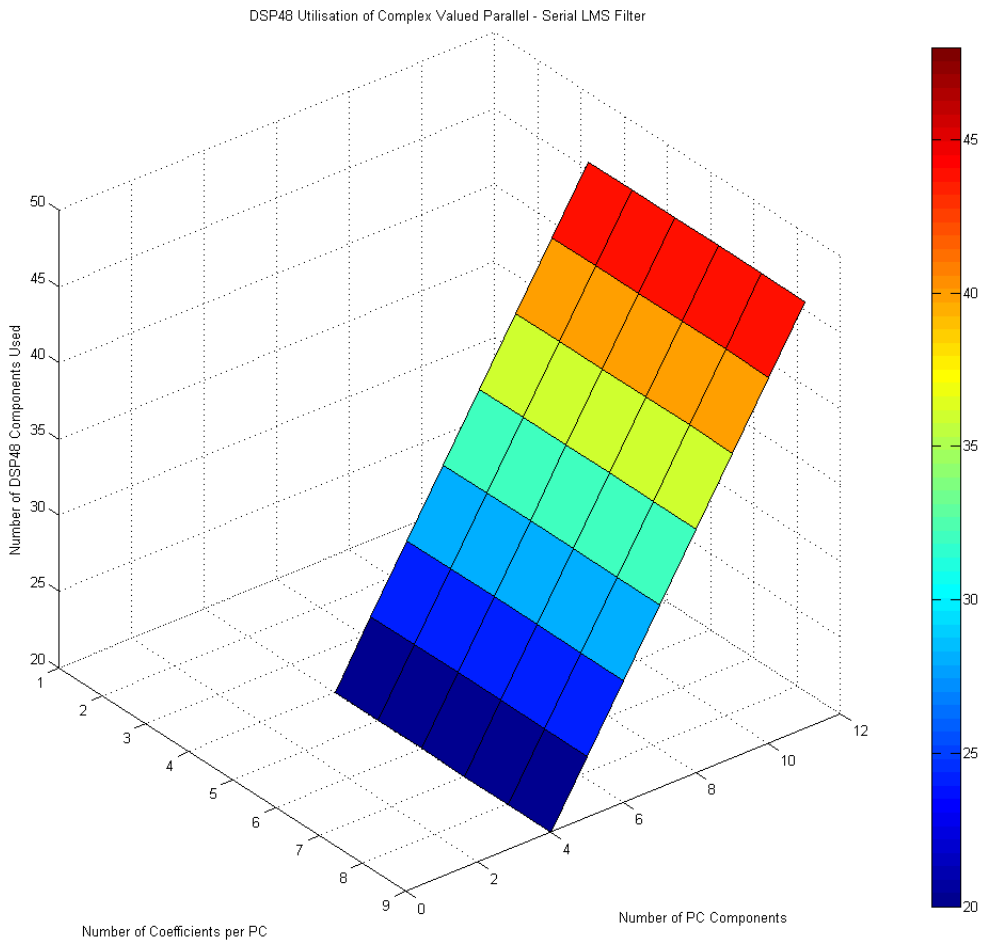


Figure 4.23: DSP48 Utilisation of Complex Valued Parallel Serial LMS Architecture

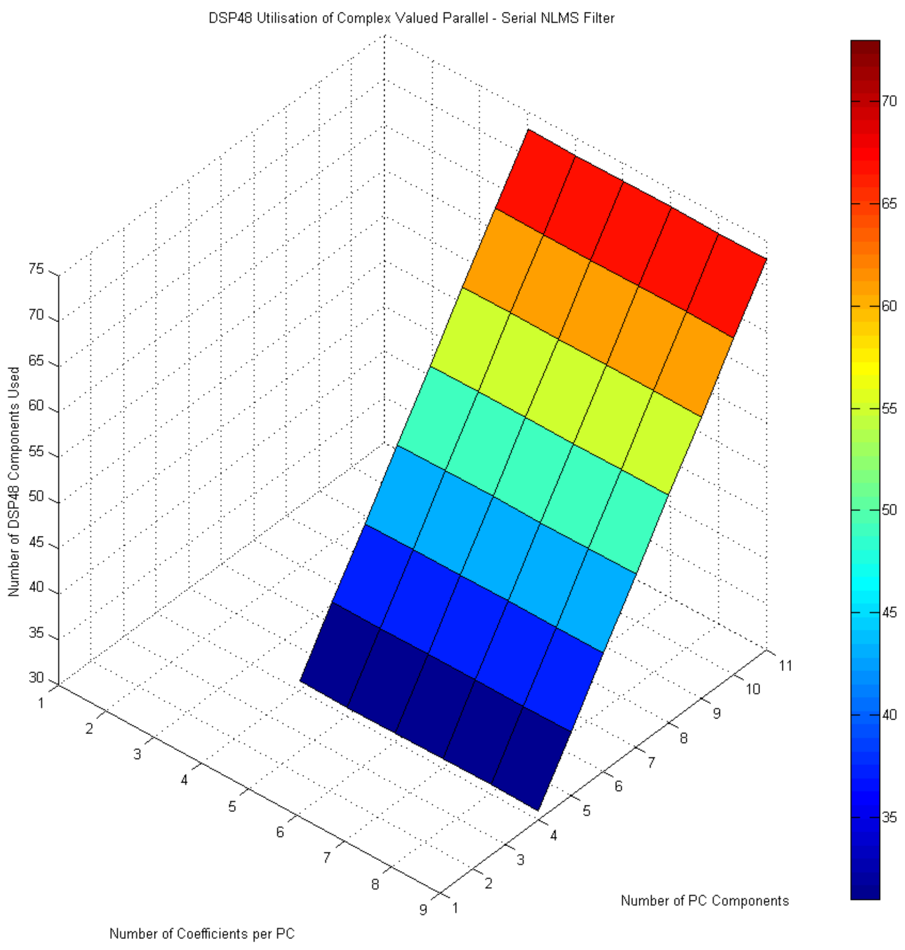


Figure 4.24: DSP48 Utilisation of Complex Valued Parallel - Serial NLMS Architecture

4.3.3 LUT Utilization

The overall LUT utilization is greater when complex valued arithmetic is employed as opposed to real valued arithmetic, due to the increased operation count incurred.

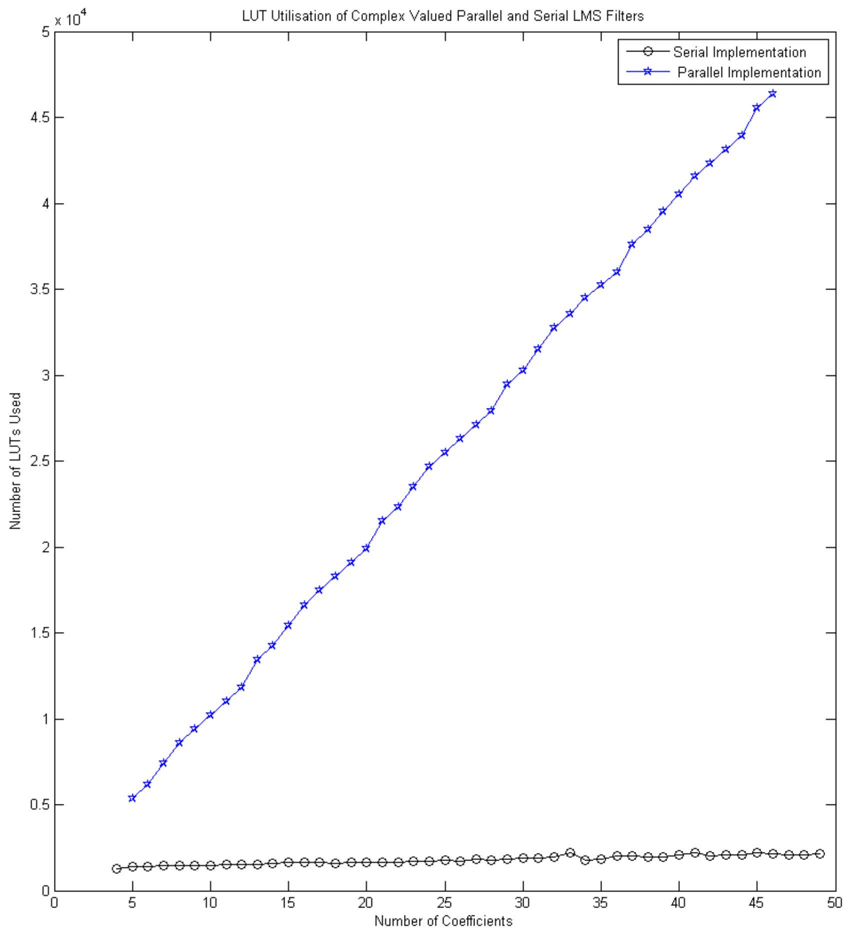


Figure 4.25: LUT Utilisation of Complex Valued Parallel and Serial LMS Architectures

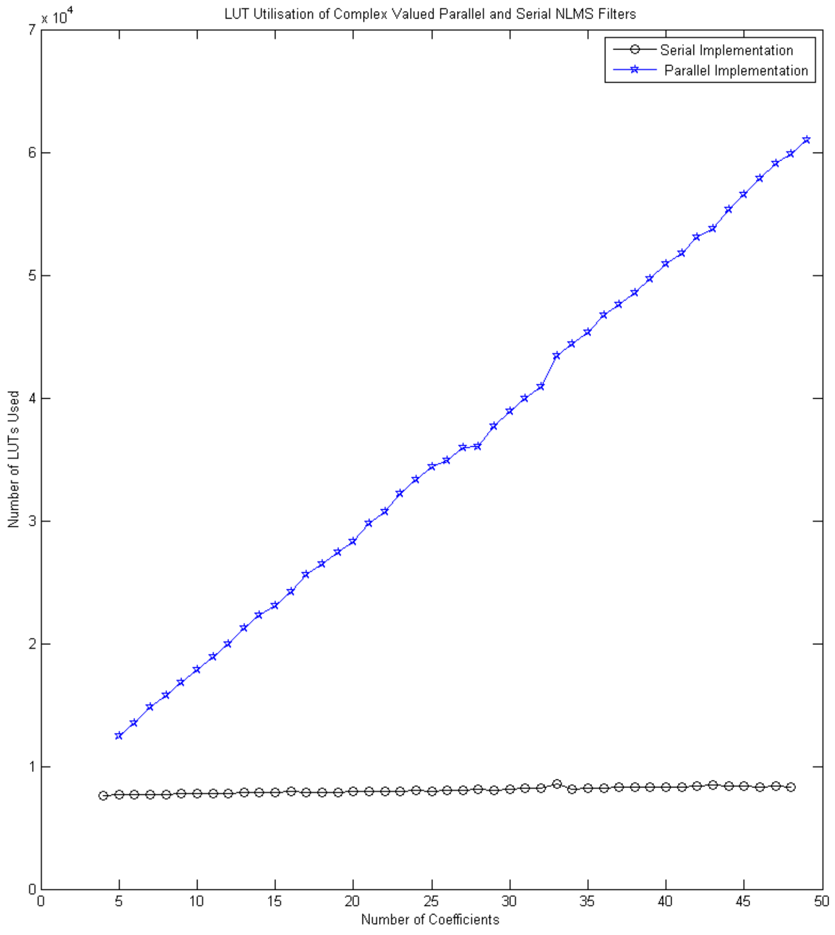


Figure 4.26: LUT Utilisation of Complex Valued Parallel and Serial NLMS Architectures

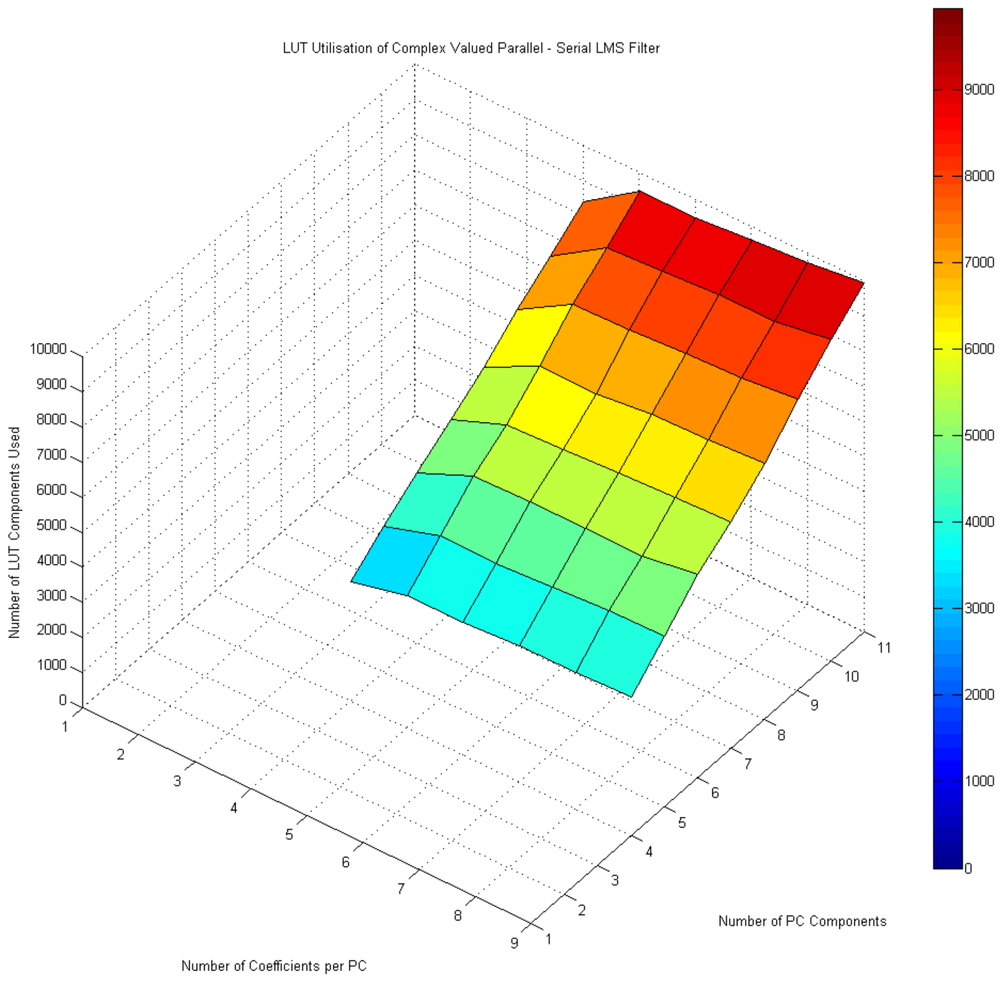


Figure 4.27: LUT Utilisation of Complex Valued Parallel - Serial LMS Architecture

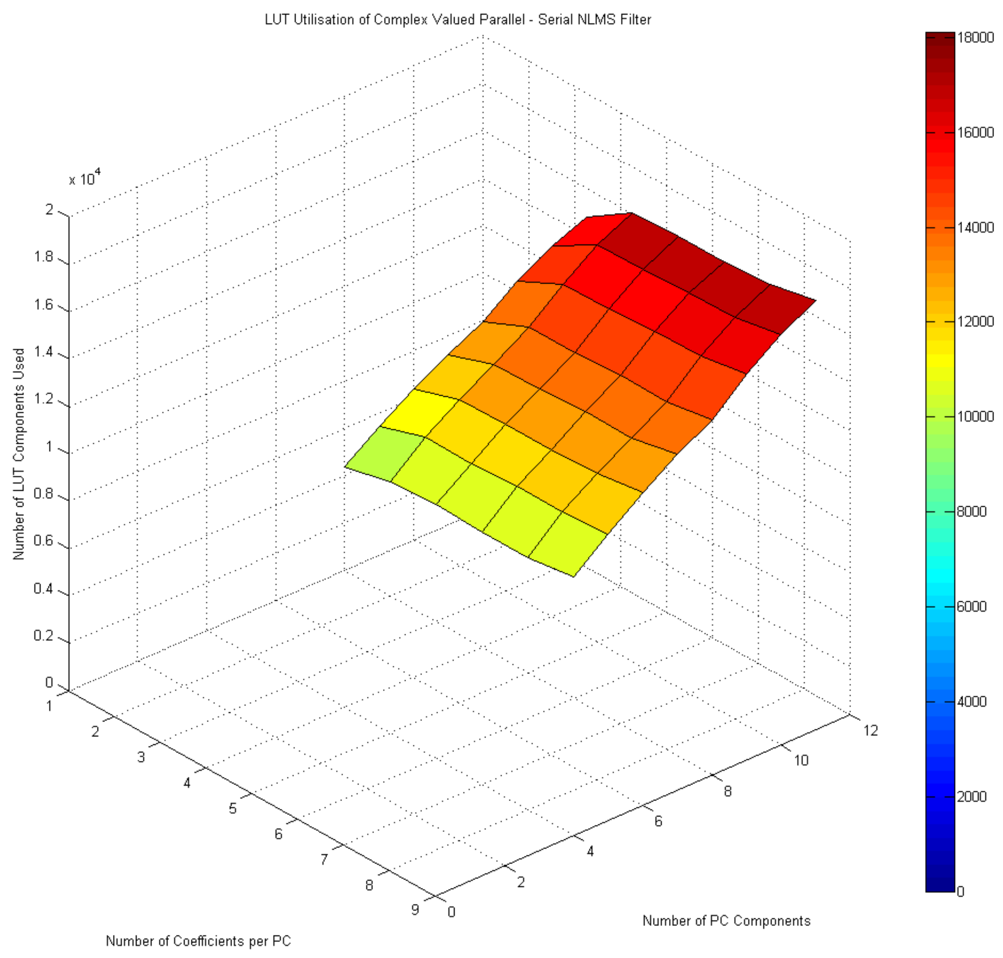


Figure 4.28: LUT Utilisation of Complex Valued Parallel - Serial NLMS Architecture

4.3.4 Slice Register Utilization

Again, the general relationship between increasing number of coefficients and increasing resource consumption is observed for slice register utilization under complex valued arithmetic. The overall cost is increased by a factor due to the extra operation count of complex valued arithmetic.

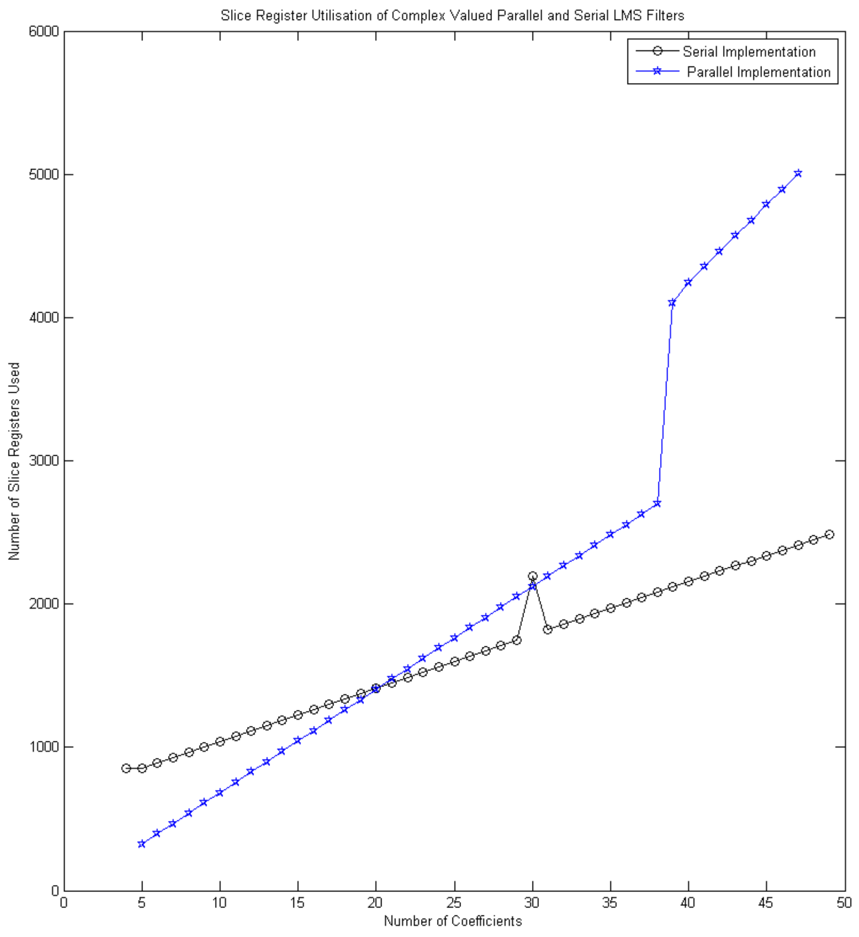


Figure 4.29: Slice Register Utilisation of Complex Valued Parallel and Serial LMS Architectures

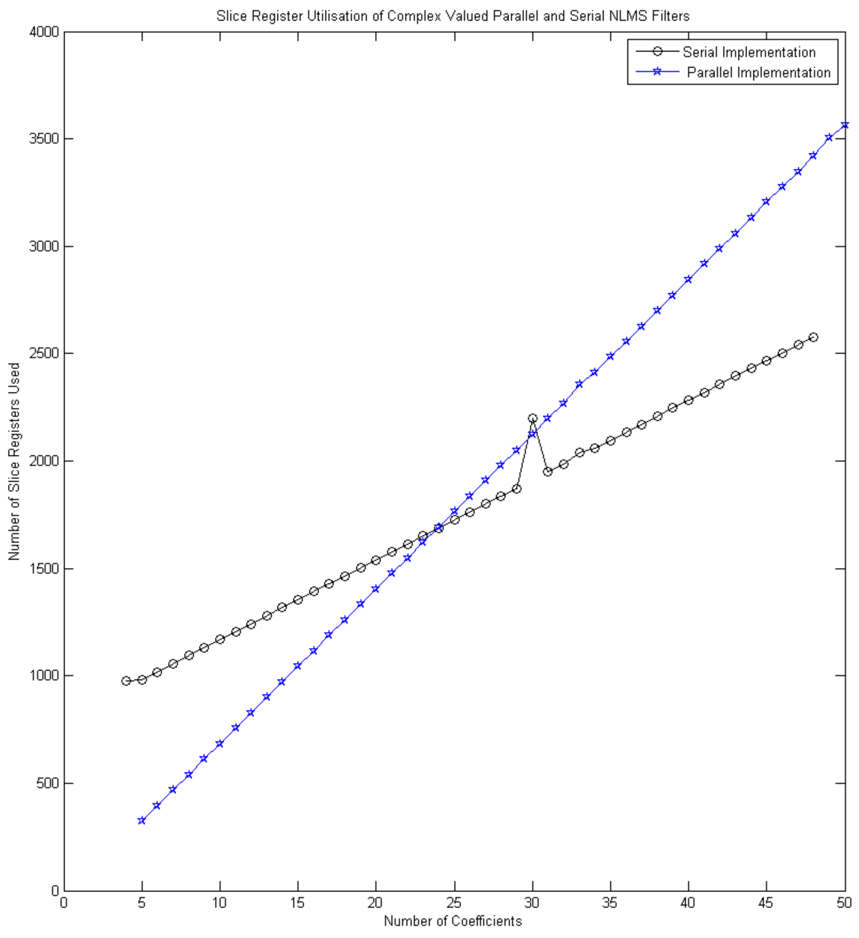


Figure 4.30: Slice Register Utilisation of Complex Valued Parallel and Serial NLMS Architectures

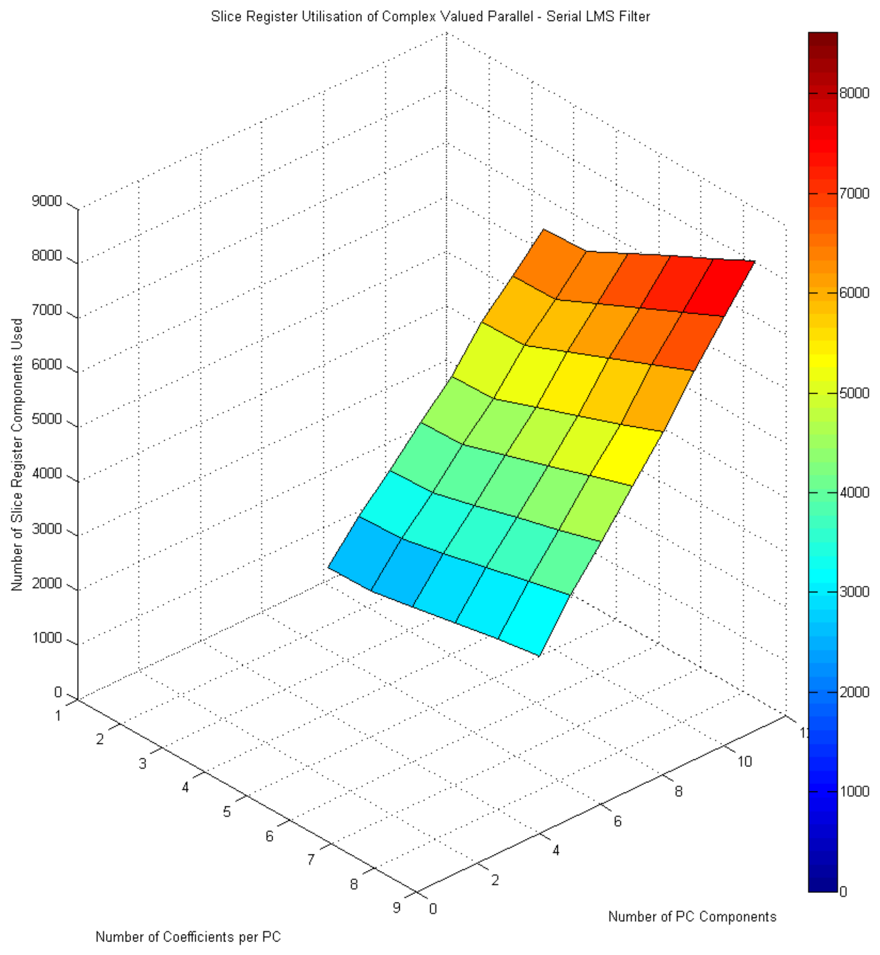


Figure 4.31: Slice Register Utilisation of Complex Valued Parallel - Serial LMS Architecture

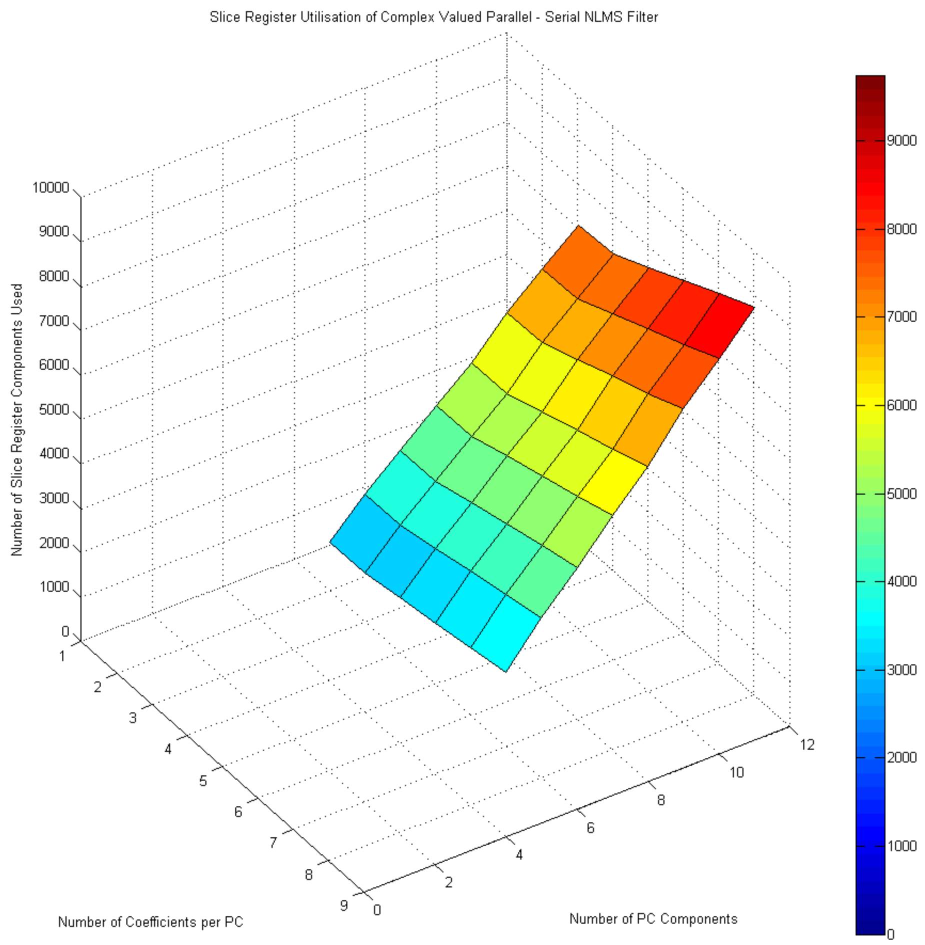


Figure 4.32: Slice Register Utilisation of Complex Valued Parallel - Serial NLMS Architecture

4.4 Discussion

There are a number of interesting observations to be made from the data regarding differing sample rates, LUT and slice register utilization between the different architectures. In this section, a concise summary of the most important results is given.

Firstly, it can be observed from the plots of sampling frequency against number of processor cores/ coefficients per core, there is a clear trend towards decreasing sampling frequency with increasing coefficients per core/ number of processor cores. Note that sampling frequency in this context is used to refer to the maximum rate at which data can be clocked into the LMS filter. Hence, it can be reasoned that as the number of cores/ coefficients per core increases, the resulting clock rate decreases due to the increasing complexity of the control logic used to orchestrate the shuttling of data amongst the filter cores. With further pipeline stages added to the control logic, a more flat distribution with higher overall sampling rate might be expected. In general, lower sampling rates are achieved for the cases where complex valued arithmetic is employed as opposed to real valued arithmetic. This again points to a potential pipelining issue. Further registers should be added to the complex arithmetic cores in order to mitigate against the increased critical path resulting from complex arithmetic.

Looking at the results of DSP48 utilization amongst the various architectures, a linear increase in the number of DSP48s occupied can be observed as the number of filter coefficients increases. This is the expected result - the number of coefficents/ processor cores should be linearly related to the number of DSP48s utilized. It can be observed that for the parallel-serial filter architectures, the number of DSP48s used varies only with the number of processor cores used - increasing the number of coefficients per processor core has no effect on the utilization which is the expected behaviour.

A general trend of utilization remaining flat can be observed for all the plots concerning parallel-serial implementation. While the number of DSP48s only varies with processor core number, there is a slight increase in the slice register utilization/ LUT utilization as the number of coefficients varies for a fixed number of processor cores. This slight variation in LUT/ slice register utilization can be attributed to the extra control logic required when the number of coefficients per core increases. Clearly the variation for fixed processors/increasing coefficients is much less than for the case of increasing processor cores/fixed coefficients.

Regarding the slice register utilization results for the parallel/serial cases, it is interesting to note that initially the fully serial implementation is more expensive to implement than the fully parallel implementation in terms of the number of registers used. This offset is due to the fixed number of pipelining stages inserted into the fully serial architecture. Recall that the extra pipelining stages cannot be inserted into the fully parallel architecture and therefore initially this implementation uses fewer registers than fully serial.

The plots of LUT utilization for fully parallel and fully serial architectures illustrate a linear increase in the resources used for the fully parallel case against a relatively constant LUT utilization for the fully serial implementation. This illustrates the efficiency of the fully serial implementation in terms of resource cost when implementing very large filters.

In general, the plots show the expected trend in terms of trade off between hardware cost and data throughput - as the filter is serialized to a greater degree, the overall throughput of the filter is decreased. This is a fundamental engineering trade off when designing filters for HDL implementation. In general the goal is to serialize the filter as far as possible while still meeting the throughput requirement given by the application in question.

Part 2 - A Survey of QRD-RLS FPGA Implementation

The LMS algorithm is a robust and relatively cheap algorithm to implement, however it can be slow to converge and is sensitive to the eigenvalue spread of the autocorellation matrix, (see 1.5). As the demands for increasing data rates continue, there is the need to consider solutions which converge much faster than LMS. QRD-RLS algorithms provide such a solution. There are many different ways with which to implement such an algorithm.

This part of the thesis seeks to give a detailed overview of the various methods available for performing QRD-RLS, with particular focus on the Givens based systolic array architectures which are highly suited to VLSI implementation. Section 5 provides a general introduction to the method of QRD, looking at the well known Gram Schmidt, Householder and Givens methods. The relative merits of each are discussed from the perspective of VLSI implementation. Following this, in Section 6, a general overview of the Givens based QRD-RLS systolic array architecture is given. Then, in Section 7 various methods for obtaining the least squares solution from the systolic array architecture are given. Section 8 looks at how the previously discussed systolic array can be optimized either for speed or to lower the overall resource consumption. Finally, Section 9 gives an overview of a Gram Schmidt VLSI implementation, such that a comparison can be made.

5 QR Decomposition Methods

The QR Decomposition is a process for which an input matrix, $\boldsymbol{A}$ is transformed into the product of an upper triangular matrix $\boldsymbol{R}$ and orthogonal matrix $\boldsymbol{Q}$. The three well known methods for performing such a procedure are:

- Gram Schmidt

- Householder Reflections

- Givens Rotations

When studying the relative merits of an algorithm, attention should be paid to the numerical stability, memory requirements, and operation count. For FPGA/ASIC implementation, the potential for parallelization also plays a key role. If the algorithm is highly sequential in nature, there will be potentially no speed-up achieved when running on an FPGA. In this section we will review each of the three algorithms and discuss the potential advantages/disadvantages of each.

5.1 Gram Schmidt

Gram Schmidt, [10], provides a procedure by which a set of *k* linearly dependent vectors $\boldsymbol{S} = \{\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3, \ldots, \boldsymbol{a}_k\}$ can be transformed to form an orthonormal basis spanning the sub-space *k*. A property of the orthogonal matrix $\boldsymbol{Q}$, is that it forms an orthonormal basis of the space spanned by $\boldsymbol{A}$**.** Therefore, performing Gram-Schmidt process on the input matrix $\boldsymbol{A}$, treating the columns as linearly independent vectors in the subspace, *k*, we can achieve the orthogonal matrix $\boldsymbol{Q} = \{\boldsymbol{q}_1, \boldsymbol{q}_2, \boldsymbol{q}_3, \ldots, \boldsymbol{q}_k\}$.

The procedure comprises subtracting from $\boldsymbol{a_k}$ its projection, $\boldsymbol{z_k}$ onto the subspace currently constructed. The projection operation is defined by [5.1].

$$proj_z = \frac{\langle \boldsymbol{z}, \boldsymbol{a} \rangle}{\langle \boldsymbol{z}, \boldsymbol{z} \rangle}\boldsymbol{a} \tag{5.1}$$

The procedure then follows as:

$$k = 1, \boldsymbol{z_1} = \boldsymbol{a_1} \quad \boldsymbol{q_1} = \frac{\boldsymbol{z_1}}{\|\boldsymbol{z_1}\|_2} \tag{5.2}$$

$$k = 2, \boldsymbol{z_2} = \boldsymbol{a_2} - proj_{z_1}(\boldsymbol{a_2}) \quad \boldsymbol{q_2} = \frac{\boldsymbol{z_2}}{\|\boldsymbol{z_2}\|_2} \tag{5.3}$$

$$k = N, \boldsymbol{z_N} = \boldsymbol{a_N} - \sum_{j=1}^{N-1} proj_{z_j}\boldsymbol{a_N} \quad \boldsymbol{q_N} = \frac{\boldsymbol{q_N}}{\|\boldsymbol{q_N}\|_2} \tag{5.4}$$

The classic Gram-Schmidt method presented above can produce vectors, $\boldsymbol{q}$, that are non-orthogonal in the presence of finite precision arithmetic due to small rounding errors etc. A simple modification which incurs no extra operation cost can be used to take into account the loss of orthogonality; this is known as Modified Gram-Schmidt method (MGS), [12]. In order to ensure orthogonality at each step, the intermediate result of the projection subtraction is used in the calculation, allowing for any round off errors introduced at each stage to be corrected for. Consider the computation of the third orthogonal vector, $\boldsymbol{z_3}$.

$$\boldsymbol{z_3} = \boldsymbol{a_3} - proj_{z_1}(\boldsymbol{a_3}) - proj_{z_2}(\boldsymbol{a_3}) \qquad [5.5]$$

Instead of carrying out the calculation in one instruction, firstly calculate:

$$\boldsymbol{z_3}' = \boldsymbol{a_3} - proj_{z_1}(\boldsymbol{a_3}) \qquad [5.6]$$

then

$$\boldsymbol{z_3} = \boldsymbol{z_3}' - proj_{z_2}(\boldsymbol{z_3}') \qquad [5.7]$$

Therefore the intermediate results are re-orthogonalized against one another, resulting in numerical stability in the presence of non-exact arithmetic. Note that, although the operation count is not increased in the modified routine, the memory access pattern becomes more irregular, which may degrade performance in software implementation. Using the modified routine gives results which are numerically equivalent to the Householder method [12], (to be discussed in next section). The Householder method is well known to be numerically stable, therefore by extension we can confirm MGS as a numerically stable routine.

In order to use Gram Schmidt to compute the QR decomposition, the method is applied, treating the *k* columns of the input matrix as the vectors to be orthonormalized. Then,

$$\boldsymbol{Q} = [\boldsymbol{q_1}, \boldsymbol{q_2}, \ldots, \boldsymbol{q_k}] \qquad [5.8]$$

and

$$\boldsymbol{R} = \begin{bmatrix} \langle \boldsymbol{q_1}, \boldsymbol{a_1} \rangle & \langle \boldsymbol{q_1}, \boldsymbol{a_2} \rangle & \langle \boldsymbol{q_1}, \boldsymbol{a_3} \rangle & \ldots \\ \boldsymbol{0} & \langle \boldsymbol{q_2}, \boldsymbol{a_2} \rangle & \langle \boldsymbol{q_2}, \boldsymbol{a_3} \rangle & \ldots \\ \boldsymbol{0} & \boldsymbol{0} & \langle \boldsymbol{q_3}, \boldsymbol{q_3} \rangle & \ldots \\ \ldots & \ldots & \ldots & \ldots \end{bmatrix} \qquad [5.9]$$

5.2 Householder Reflections

Householder reflections, [13], provide a numerically robust method to compute QR. It is often the preferred method for computation on sequential machines, used in many numerical computing libraries, e.g LAPACK [14]. Although Gram Schmidt can produce results of similar accuracy with reorthogonalization in place, the memory access of Householder is more suited to software implementation.

Householder transformations are a linear, norm preserving procedure. The process is illustrated graphically in Figure 5.1, below. For ease of exposition, a 2-dimensional subspace is shown, however the method can be readily extended to higher dimensional problems. The goal is to perform a transformation on vector **x**, such that all of the information is represented in one element. This represents an annihilation of elements in the vector, which we can extend later to perform a full QR decomposition of an input matrix. To achieve such a transformation, **x** is reflected with respect to the hyperplane, which is perpendicular to $\boldsymbol{y}$, $\boldsymbol{y}^{\perp}$.

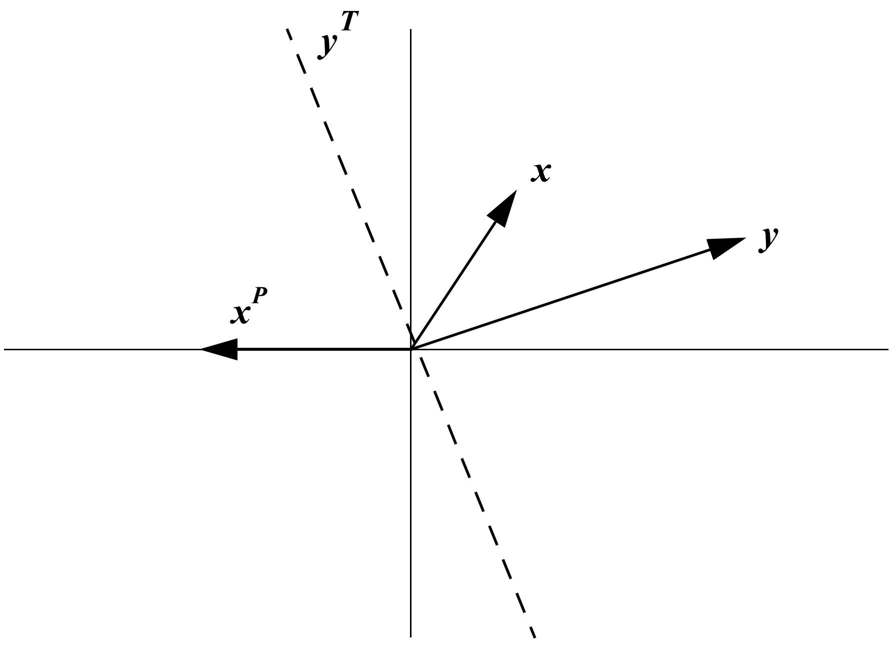


Figure 5.1: Householder Reflections

The vector $\boldsymbol{y}$ can be constructed from:

$$\boldsymbol{y} = \boldsymbol{x} + \|\boldsymbol{x}\|_2 \boldsymbol{a}_i \qquad [5.10]$$

where $\boldsymbol{a}_i$ represents a column of the identity matrix with column index *i*.

The Householder matrix $\boldsymbol{H}$ is then constructed using:

$$\boldsymbol{H} = \boldsymbol{I} - 2\boldsymbol{y}\boldsymbol{y}^T \qquad [5.11]$$

where $\boldsymbol{I}$ is the identity matrix.

Householder reflections can therefore be used to form a series of $\boldsymbol{Q}_i$ matrices, where:

$$\boldsymbol{Q}_i = \begin{bmatrix} \boldsymbol{I}_i & 0 \\ 0 & \boldsymbol{H}_i \end{bmatrix} \qquad [5.12]$$

Therefore, in order to perform a full QR decomposition of the input matrix, a series of $N-1$ reflections must be performed (where N is the number of columns), (Figure 5.2).

$$\begin{bmatrix} x & x & x \\ x & x & x \\ x & x & x \end{bmatrix} \xrightarrow{\boldsymbol{Q}_1} \begin{bmatrix} x' & x' & x' \\ 0 & x' & x' \\ 0 & x' & x' \end{bmatrix} \xrightarrow{\boldsymbol{Q}_2} \begin{bmatrix} x' & x' & x' \\ 0 & x'' & x'' \\ 0 & 0 & x'' \end{bmatrix}$$

Figure 5.2: QR Decomposition via Householder Reflections

Every non-zero element of the matrix is altered when each orthogonal matrix $\boldsymbol{Q}_i$ is applied. The calculation of each Householder matrix, $\boldsymbol{H}$, can only be performed after the orthogonalization matrix has ben applied. Therefore, it is not possible to parallelize the orthogonalization and the processing bound is limited by the ($N-1$) Householder reflections.

Householder is an excellent method for floating point processor implementation. The regularity of the memory access pattern, coupled with the use of common floating point operations makes the algorithm very suited to modern processor architectures. The highly sequential nature, along with requirement for non-fixed point friendly operations such as square root (from the L2 norm used to calculate $\boldsymbol{y}$), mean that it is not the method of choice for FPGA/ASIC implementation. The Givens method discussed next provides a more VLSI friendly choice of implementation.

5.3 Givens Rotations

The method of Givens rotations, [15], like Householders reflections is a norm-preserving, orthogonal transformation. Unlike Householder reflections, however, Givens rotations can be more readily parallelized. The basic operation is that of a [2x2] matrix:

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \qquad [5.13]$$

Such a matrix will rotate the [2x1] element vector by the specified angle. From straightforward geometry, we can compute the angles required in order to rotate the vector onto the axis:

$$\cos\theta = \frac{x}{\sqrt{x^2+y^2}} \; ; \; \sin\theta = -\frac{-y^2}{\sqrt{x^2+y^2}} \qquad [5.14]$$

In matrix notation, we can represent the Givens Matrix, **G,** as:

$$\boldsymbol{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \cos\theta & 0 & \sin\theta & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -\sin\theta & 0 & \cos\theta & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad [5.15]$$

For a Givens rotation performed on the row i, column j, the c, s terms will appear on the intersection points of row j, column i.

Unlike Householder reflections, for each element that is annihilated from the input matrix, only a subset of the elements of the matrix are altered, with each unitary rotation matrix that is applied (see Figure 5.3). This opens up potential for Givens rotations to be applied in parallel, unlike Householder where each orthogonal matrix $\boldsymbol{Q}_i$, must be applied sequentially in turn.

$$\boldsymbol{Q}_1 \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & -7 & 11 \end{bmatrix} = \begin{bmatrix} 2.24 & 7.60 & 12.97 \\ 0 & -1.79 & -3.58 \\ 3 & -7 & 11 \end{bmatrix}$$

$$\boldsymbol{Q}_2 \begin{bmatrix} 2.24 & 7.60 & 12.97 \\ 0 & -1.79 & -3.58 \\ 3 & -7 & 11 \end{bmatrix} = \begin{bmatrix} 3.74 & -1.07 & 16.57 \\ 0 & -1.79 & -3.58 \\ 0 & -10.28 & -3.82 \end{bmatrix}$$

$$\boldsymbol{Q}_3 \begin{bmatrix} 3.74 & -1.07 & 16.57 \\ 0 & -1.79 & -3.58 \\ 0 & -10.28 & -3.82 \end{bmatrix} = \begin{bmatrix} 3.74 & -1.07 & 16.57 \\ 0 & -10.43 & -4.38 \\ 0 & 0 & 2.87 \end{bmatrix}$$

Figure 5.3: Givens Rotations Example

To ease the computational requirements we desire an algorithm which can recursively update the matrix without having to perform a full matrix-by-matrix multiplication at each time step. Consider the example illustrated in Figure 5.4, where a full QR decomposition has been performed on the 5x5 data matrix $\boldsymbol{A}$, and a new row has arrived. Instead of performing a full QR decomposition of the matrix formed with the new row, we can exploit the fact that the matrix is almost in the upper triangular form. Treating the new row and previous row as a recursion, we can form the update of each row iteratively, for each new row that arrives.

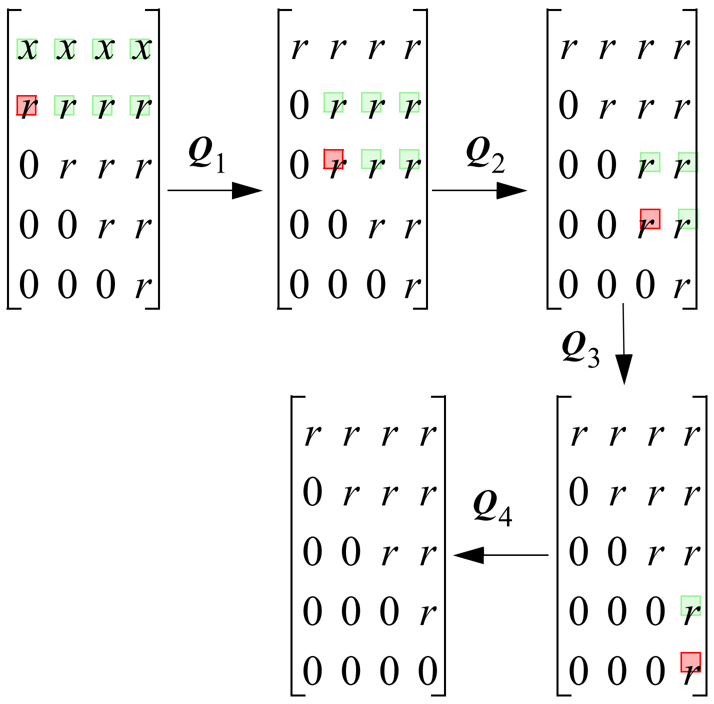


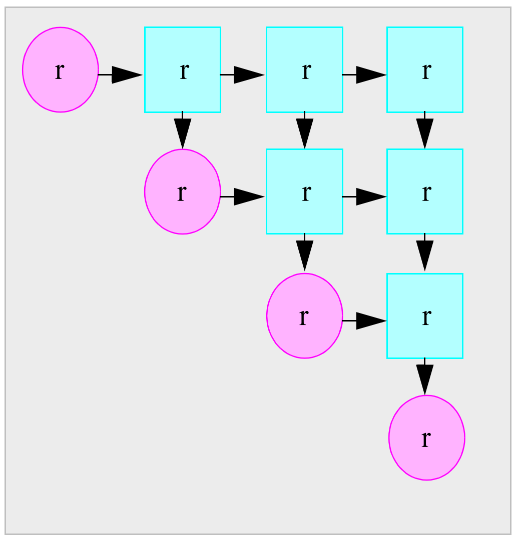
Figure 5.4: Recursive QR Decomposition

6 Givens Based QRD-RLS Implementation

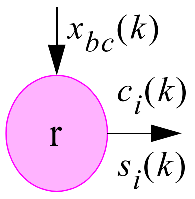
In this section, the basic concepts behind Givens rotation based systolic array architectures are developed. We look at how an array of common processing elements can be interconnected to form a recursive QRD processor. Highly efficient CORDIC arithmetic is explored as a suitable candidate for FPGA implementation of both real and complex valued QRD systolic arrays.

6.1 QRD-RLS Systolic Array Architecture

We previously examined how the QRD could be formed recursively with each new row of data that arrives. This is the essential idea behind the well known systolic array architecture for QR decomposition (Figure 6.1), first proposed by Gentleman and Kung [17]. The systolic array architecture comprises two different types of processing element, named boundary cell, and internal cell. The boundary cell takes the input formed by the current (k), and previous $(k-1)$ input data, forming a vector which is rotated onto the axes, annihilating the lower element. In doing so, the angle by which the vector was rotated is stored, and passed along to the internal cells, where the vector is rotated. The variable λ, is known as the forgetting factor. It is a small constant (in the order of 0.9999999...) which ensures that the systolic array will gradually de-emphasize older samples. Without such a constant in place, the systolic array will attempt to find the solution to all data, both past and present, and potentially may never converge.



Boundary Cell (Givens Generation)



$if\ \boldsymbol{x}_{bc}(k) = 0\ then$

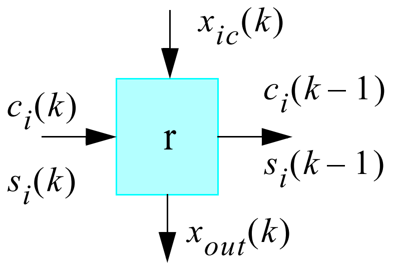
$\boldsymbol{c}_i(k) = 1\ ;\ \boldsymbol{s}_i(k) = 0$

else

$c_i(k) = r(k)(r^2(k) + x^2{}_{bc}(k))^{-1/2}$

$s_i(k) = x_{bc}(k)(r^2(k) + x^2{}_{bc}(k))^{-1/2}$
$r(k+1) = \lambda^{1/2}(r^2(k) + x^2{}_{bc}(k))^{1/2}$

Internal Cell (Givens Rotation)



$r(k) = s_i(k)x_{ic}(k) + \lambda^{1/2}c_i(k)r(k-1)$

$x_{out}(k) = c_i(k)x_{ic}(k) - \lambda^{1/2}s_i(k)r(k)$

Figure 6.1: Systolic Array Architecture for QR Decomposition

In order to form the least squares solution, Gentlemen and Kung also showed in [17] how an “on the fly” least squares computation can be performed concurrently with the triangularization of the input matrix. Consider that the essential problem is to solve the following:

$$Rw_{ls} = Qd = p \qquad [6.1]$$

This can be achieved by appending an additional column to the systolic array, and orthogonalizing in the same manner, (Figure 6.2).

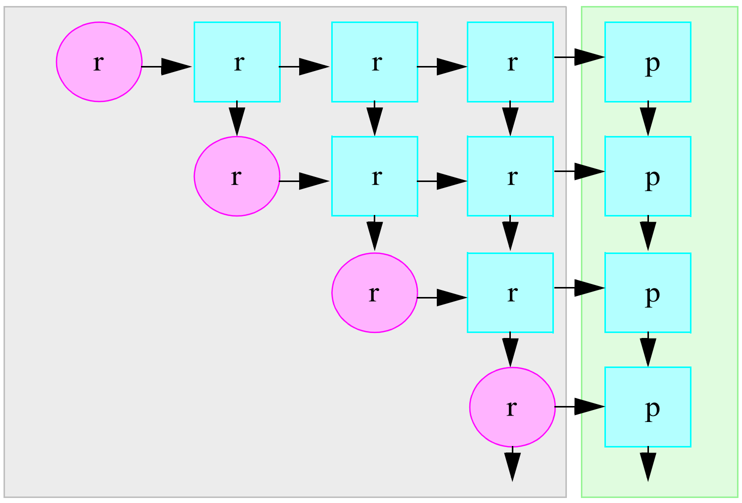


Figure 6.2: Systolic Array with Additional Column Appended

The least squares solution can then be obtained via backsubstitution, as will be discussed in Section 7.1. Before moving on however, it is important to note the potential for CORDIC arithmetic to be used in the architecture.

6.2 CORDIC Arithmetic for Boundary/Internal Cells

The Coordinate Rotation DIgital Computer (CORDIC) algorithm, [18],is a numerically robust, multiplier free algorithm suitable for performing a variety of trigonometric functions using only shift and add arithmetic. It arises from a simplification of the Givens rotations previously discussed.

The classic Givens rotations are given by:

$$x' = x\cos\theta - y\sin\theta \qquad [6.2]$$

$$y' = y\sin\theta - x\cos\theta \qquad [6.3]$$

after simple rearrangement:

$$x' = \cos\theta(x - y\tan\theta) \qquad [6.4]$$

$$y' = \cos\theta(y + x\tan\theta) \qquad [6.5]$$

Now, consider when the $\tan\theta$ terms are restricted such that $\tan\theta = 2^{-i}$, where i is a real valued integer number. That is to say, only rotations which equate to powers of two terms are permitted. Then, [6.4], [6.5] become:

$$x' = \cos\theta(x - yd2^{-i}) \qquad [6.6]$$

$$y' = \cos\theta(y + xd2^{-i}) \qquad [6.7]$$

The decision factor $d = \pm 1$ in [6.6], [6.7] takes into account both clockwise and anti-clockwise rotations.

Now in order to reduce the algorithm to a shift-add form, the $\cos\theta$ term is dropped. Therefore, each rotation that is performed incurs a gain of $1/(\cos\theta)$. If we perform a rotation as a fixed succession of clockwise and anticlockwise power of two micro-rotations, then as $\cos\theta = -\cos\theta$, the accumulated processing gain from each micro rotation can be precomputed and applied to the output, see [6.8]. As this value is a constant, it can be applied using multiplier free Canonic Signed Digit arithmetic or other similar methods at the input or output of the CORDIC unit.

$$\boldsymbol{K}_n = \prod \cos \operatorname{atan} 2^{-i} \qquad [6.8]$$

In order to perform a rotation by an arbitrary angle, we require decision logic at each micro-rotation to determine whether to rotate clockwise or anti-clockwise. This can be performed by introducing a third, z_i accumulator, which keeps track of the current angle rotated:

$$z' = z + d\theta \qquad [6.9]$$

By determining whether the current angle held in the accumulator is greater than or less than the desired angle of rotation, the decision factor for the next stage in the CORDIC pipeline can be generated:

$$d = \begin{cases} 1 if(z > 0) \\ -1 otherwise \end{cases} \qquad [6.10]$$

In the example shown in Figure 6.3 below, it is desired to rotate the input vector by $30°$. After the first rotation, the angle stored in the angle accumulator is $30° - 45° = -15°$. The second rotation therefore performs an anti-clockwise rotation to bring the accumulated angle to $11.6°$. A third clockwise rotation brings the angle held in the accumulator to $-3.4°$. The total angle accumulated relative to the input is therefore $33.4°$. Further rotations will bring the error in rotation down further. Note how the magnitude of the vector grows with each iteration performed. This can be corrected at the output as previously discussed.

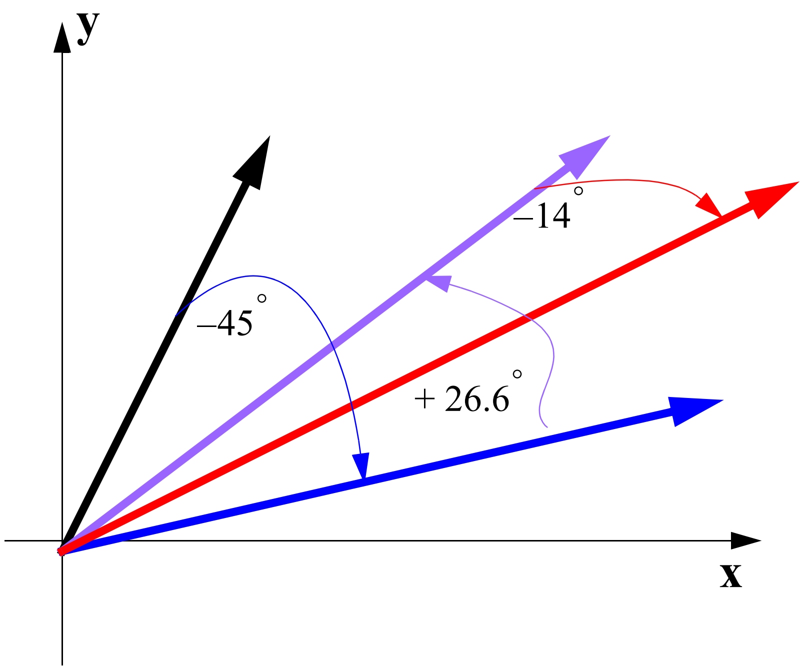


Figure 6.3: CORDIC Rotation Mode

In addition to being able to rotate an input vector by a given angle, CORDIC can also be used in the so-called “Vectoring Mode”. In this mode, the input vector is iteratively rotated onto the x axis. In order to achieve this the decision factor is now generated by looking at the sign of the y input, [6.11].

$$d_i = \begin{cases} 1 if(sign(y) < 0) \\ -1 otherwise \end{cases} \qquad [6.11]$$

An example of CORDIC operating in the vectoring mode is given in Figure 6.4 below. The vector has an initial angle of $30°$. The first rotation pushes the vector over the x axis by $15°$. Therefore, the second rotation is performed in the anticlockwise direction. The third rotation can then be made in the clockwise direction. At the end of the series of rotations, the angle held in the accumulator is $-45° + 26.6° - 14° = -32.4°$. Further iterations will bring the vector closer to the axis, and will therefore improve the accuracy of the angle estimate.

:

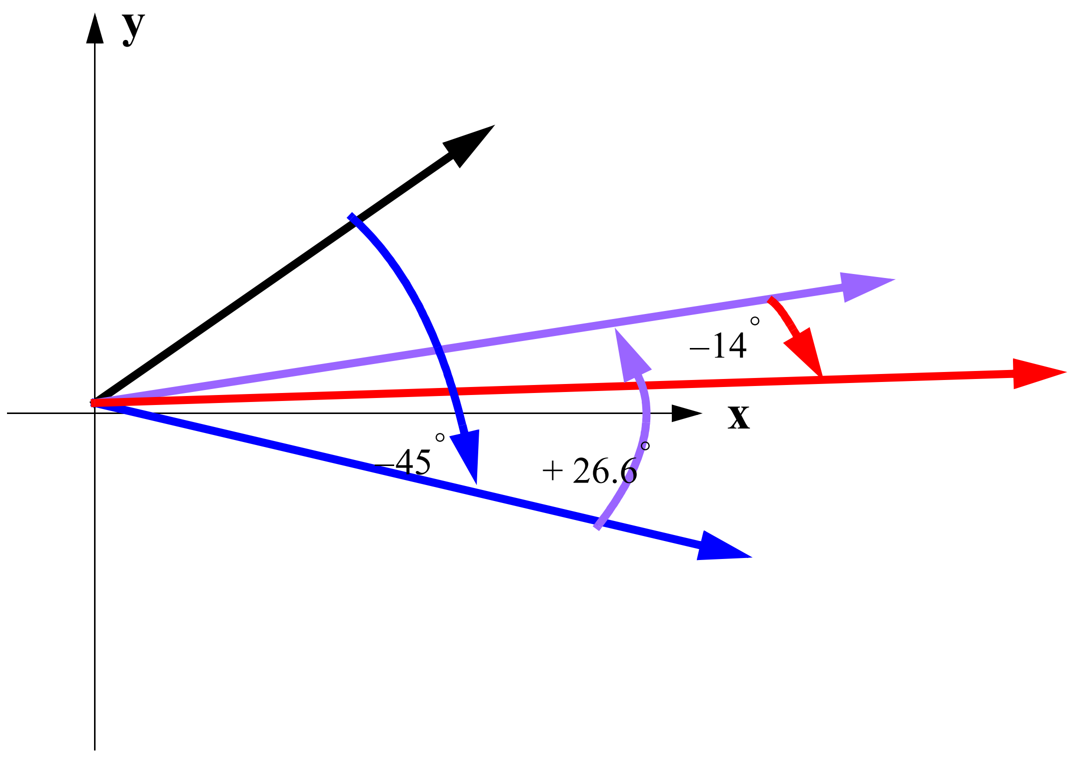


Figure 6.4: CORDIC Vectoring Mode

CORDIC arithmetic can be used to perform the processing tasks required by the boundary and internal cells of the systolic array architecture previously discussed. Using CORDIC in vectoring mode will annihilate an element of the input vector by forcing it onto the axis; this is the operation of the boundary cell. The resulting angle can be passed along to the internal cells, for which CORDIC in rotation mode can be used. This results in a highly efficient architecture, whereby the previously required square root and multiplications are removed.

6.3 Complex Systolic Array using CORDIC Arithmetic

Up until now, we have only considered the processing of real valued input data in the systolic array. The equations for the boundary and internal cell shown in Figure 6.1 can be readily extended to the complex domain.

The extension of CORDIC arithmetic into the complex domain for QRD-RLS filtering has been investigated independently in both [21] and [22]. To achieve the boundary cell operation, the circuit shown in Figure 6.5 can be used. The circuit takes in the real and imaginary components of the complex valued input, and performs a CORDIC vectoring operation, such that the imaginary part of the input is annihilated. Then, the basic recursion of the boundary cell is performed, whereby a vector is formed from the (k), $(k-1)$ values, and the previous element is annihilated. The angles Φ and θ used to null the imaginary component and perform the recursion are passed along to the complex rotation cells.

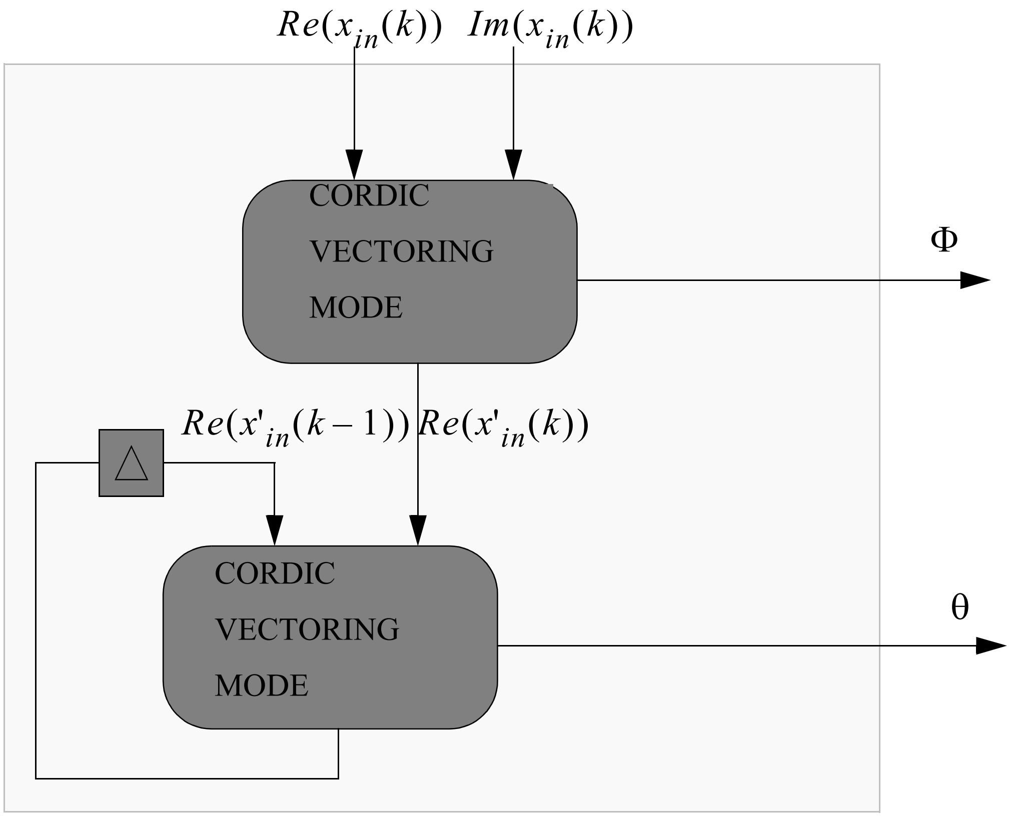


Figure 6.5: Complex Boundary Cell Operation using CORDIC Arithmetic

To compute the complex Givens rotations, the circuit shown in Figure 6.6 is employed. The components labelled Givens rotation differ slightly in [21] and [22]. In the former, a standard CORDIC rotation cell is employed, while in the latter, the Givens rotation presented in [5.13] is directly computed using multiply and add logic. The advantage of using CORDIC for the Givens rotation is that it increases the regularity of the combined circuit. Indeed, in [21] a "CORDIC Super Cell" (CSC) which can perform either complex boundary cell, or complex internal cell operations is used for each element in the array. Also, as previously discussed, using CORDIC exclusively results in a multiplier free design. Using Multiply ACCumulate (MAC) logic to compute the Givens rotations is advantageous in [22] as they are specifically targeting a Xilinx FPGA. Such devices feature embedded MACs, known as DSP48s, [23], which would have been otherwise left idle.

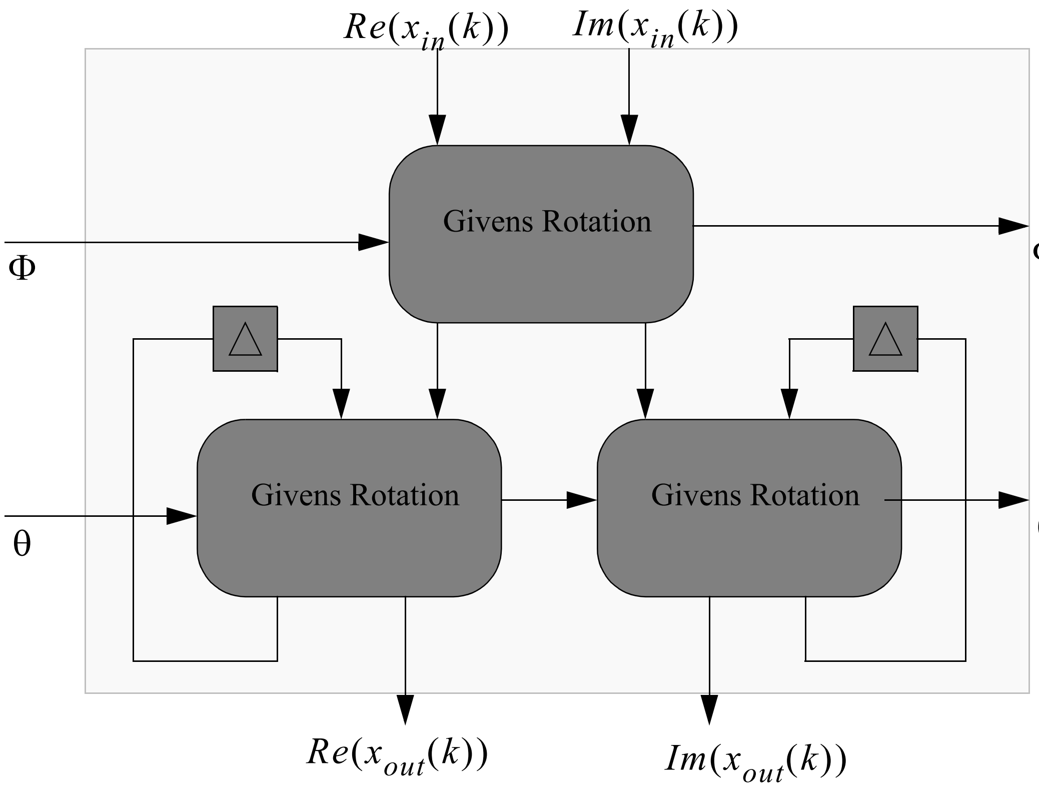


Figure 6.6: Complex Internal Cell Operation

7 Least Squares Solution Computation

In previous sections we have examined how a systolic array with CORDIC arithmetic results in an efficient architecture for performing the QRD and forming the equation $\boldsymbol{R}\boldsymbol{w}_{\boldsymbol{ls}} = \boldsymbol{Q}\boldsymbol{d}$. Now we will explore several ways in which the least squares solution can be obtained.

7.1 Back-Substitution

An obvious method for extracting the weights is to directly solve the system of equations formed by the upper triangular matrix, using Gaussian elimination, also known as back-substitution. Indeed, this is the method proposed by Gentleman and Kung in their original paper detailing the systolic array [17]. The basic idea is shown in Figure 7.1 below.

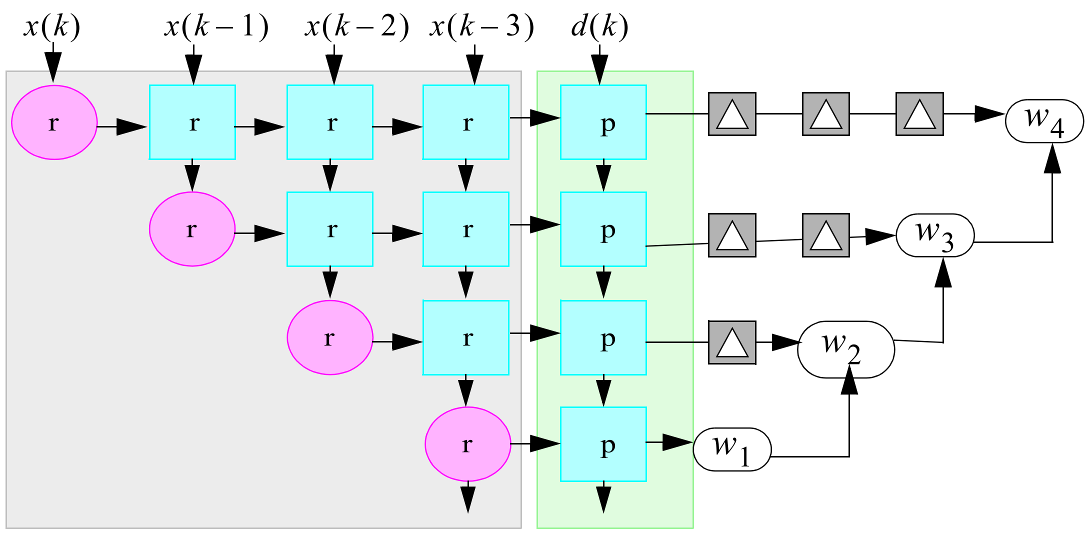


Figure 7.1: Back-Substitution Approach for Least Squares Solution

There are two immediate problems with this approach however:

- Gaussian elimination is known to be an approach that can be numerically unstable. High precision arithmetic and row pivoting are generally required to ensure correct computation for any input.
- During the computation of the weight vector, the QRD must be halted, stalling new data from the input until the procedure is complete.

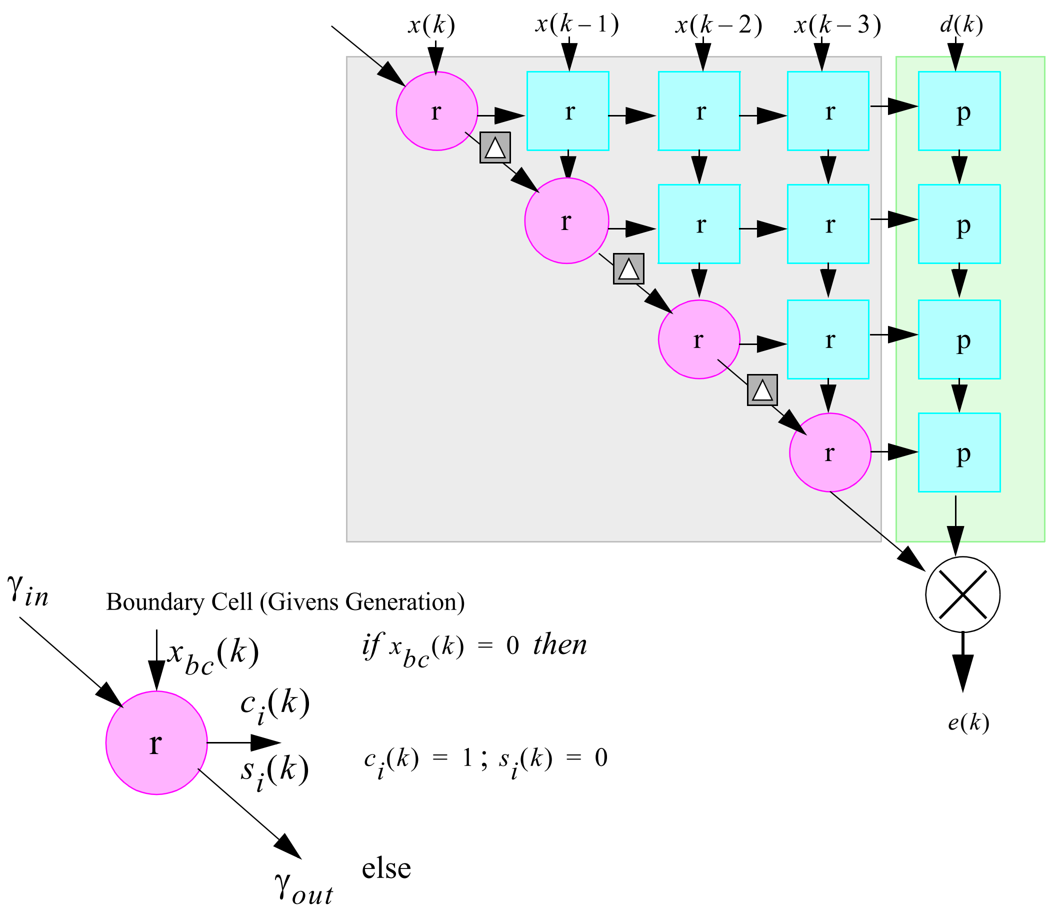
For these reasons, it can be preferable to perform the Gaussian Elimination using a separate floating point processor in tandem with the FPGA logic. An interesting approach is taken in [24], where an embedded soft processor is instantiated in the FPGA fabric. The processor features single floating point arithmetic, and custom divide operations using hardware acceleration, allowing the Gaussian elimination to be computed in a fast and numerically stable fashion without having to couple an FPGA and external processor together.

7.2 Implicit Weight Extraction

The previously discussed backsubstitution approach has several limitations which prevent a full systolic array implementation being realized which can keep pace with the input data. McWhirter, [25], was the first to propose a systolic architecture which can compute the least squares solution in real-time without needing to stall the input data for weight extraction. In fact, the proposed systolic array does not explicitly compute the weights and perform the well known equation, [7.1], to compute the residual. The residual is instead computed on the fly, at the same time as the QRD is performed.

$$e(k) = \boldsymbol{R}\boldsymbol{w}_{\boldsymbol{ls}} - d(k) \qquad [7.1]$$

McWhirter introduces an additional processing requirement to the boundary cells of the systolic array. Now, the boundary cells are also required to compute the product of cosines, produced by the diagonal interconnect of boundary cells, (shown in Figure 7.2). The product of cosines is then multiplied by the output of the column which produces the orthogonally rotated desired vector, $\boldsymbol{d}(k)$, directly producing the residual. The reader can refer to [26] for a full proof.



if $x_{bc}(k) = 0$ *then*

$$c_i(k) = 1\,;\ s_i(k) = 0$$

else

$$c_i(k) = r(k)(r^2(k) + x^2{}_{bc}(k))^{-1/2}$$

$$s_i(k) = x_{bc}(k)(r^2(k) + x^2{}_{bc}(k))^{-1/2}$$

$$r(k+1) = \lambda^{1/2}(r^2(k) + x^2{}_{bc}(k))^{1/2}$$

$$\gamma_{out} = \gamma_{in} c_i(k)$$

Figure 7.2: McWhirter Systolic Array with Direct Residual Extraction

As an alternative to computing the product of cosines in the boundary cells, an additional column of internal cells can be appended onto the systolic array where the input is unity (Figure 7.3). This can result in a more regular architecture which can be more easily folded, in order to share resources.

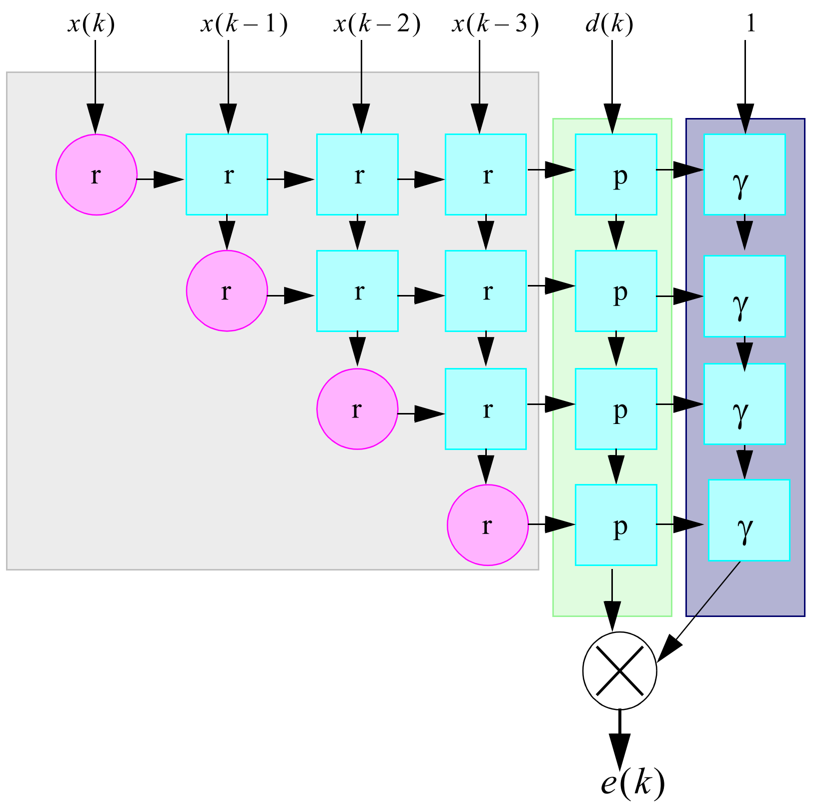


Figure 7.3: Modified McWhirter Systolic Array with Direct Residual Extraction

There are many applications where only the error signal $e(k)$ is required. This is common in the adaptive beamforming scenario, where the antenna array is steered using the residual obtained from the least squares solution.

7.3 Weight Flushing

A trivial extension to the architecture mentioned in the previous section allows the filter weights to be extracted directly. The method known as "weight flushing", [27], consists of freezing the systolic array once the QRD has been performed, and then providing the identity matrix, $\boldsymbol{I}$, as input. This corresponds to measuring the impulse response of the system.

7.4 Downdating Method

Building upon the McWhirter Systolic array with direct residual extraction previously presented, we can create an architecture that can extract the filter weights at the same rate as the QRD is computed. The QRD-RLS downdating architecture, discussed in [28], [29], [30], allows for the "on the fly" computation of the weight vector, while both the QRD is performed and the residual extracted.

With the downdating method, we apply the same orthogonal rotation matrix, $\boldsymbol{Q}$, (used to create $\boldsymbol{R}$), to a block matrix consisting of $(\boldsymbol{R}^{-T}(k-1))/(\sqrt{\lambda})$ and $\underline{0}^T$, [7.2].

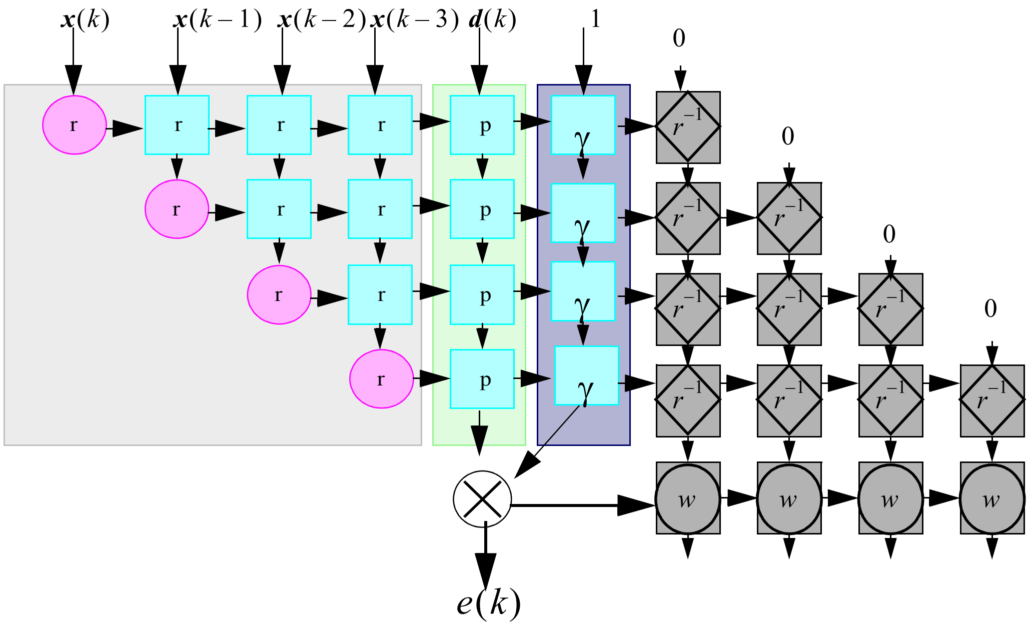
$$\boldsymbol{Q}(k)\begin{bmatrix}(\boldsymbol{R}^{-T}(k-1))/(\sqrt{\lambda})\\ \underline{0}^T\end{bmatrix} = \begin{bmatrix}\boldsymbol{R}^{-T}(k)\\ \boldsymbol{g}^T(k)\end{bmatrix} \tag{7.2}$$

After some simplification, the recursion shown in [7.3] is obtained.

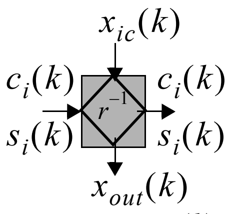
$$\boldsymbol{w}(k) = \boldsymbol{w}(k-1) - \boldsymbol{g}(k)e(k) \tag{7.3}$$

We can therefore form a recursive, parallel weight extraction QRD-RLS systolic array architecture by appending an additional lower triangular array to compute $\boldsymbol{R}^{-T}(k)$ and a row of weight extraction cells which satisfy the recursion given in [7.3], as shown in Figure 7.4.

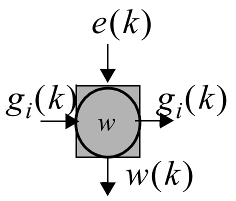
Two new cells are therefore introduced into the systolic array architecture, namely the downdating cell and weight extraction cell. The downdating cell performs almost the same operation as the internal cell in the upper triangular (left-side) QRD array. The only difference is that while the forgetting factor for the internal cell is λ, the forgetting factor for the downdating cell is λ^{-1}. As noted in [29], this is potentially an issue for numerical stability. The forgetting factor in the downdating section is greater than unity, and so any small errors in orthogonalization computed by the left-side, lower triangular array will be amplified over time, and eventually the algorithm may diverge.



Downdating Cell



Weight Extraction Cell



$$\boldsymbol{w}(k) = \boldsymbol{w}(k-1) - \boldsymbol{g}(k)e(k)$$

$$r(k) = \boldsymbol{s}_i(k)x_{ic}(k) + \lambda^{(-1/2)}\boldsymbol{c}_i(k)r^{-1}(k-1)$$

$$x_{out}(k) = \boldsymbol{c}_i(k)x_{ic}(k) - \lambda^{(-1/2)}\boldsymbol{s}_i(k)r^{-1}(k)$$

Figure 7.4: QRD-RLS with Downdating

8 Optimization for Throughput Increase or Resource Minimization

In this section several techniques are examined in order to maximize the throughput of the systolic array architecture, or for the conflicting goal of minimizing the resources consumed.

8.1 Fine Grain Pipelining

The previously discussed systolic array architectures are fully pipelined at the cell level, commonly referred to in the literature as coarse-grain pipelining. The iteration bound is therefore limited by the minimum time taken to compute the result of an individual cell. However, the internal circuitry of an individual cell in the systolic array can not be directly pipelined due to the feedback loop created by the recursive operation. For applications requiring very high throughput, this lower limit may prove to become a problem. In this section we examine how the Annihilation Reordering Look Ahead technique can be used to pipeline the QRD-RLS systolic array without altering the numerical behaviour of the algorithm.

8.1.1 Processing Element Iteration Bound

Before discussing the potential strategies to mitigate against the lower iteration bound presented at the cell level, we will first look at the problem in greater detail. Consider the boundary cell in the systolic array, which must rotate the vector formed by the incoming sample $\boldsymbol{x}_{bc}(k)$ and the previous upper triangular element $\boldsymbol{r}(k-1)$, where CORDIC arithmetic is used. Figure 8.1, illustrates the macro-level CORDIC rotator is composed of a connection of several CORDIC micro-rotations, each of which rotate the input by a fixed angle, 2^{-i}.

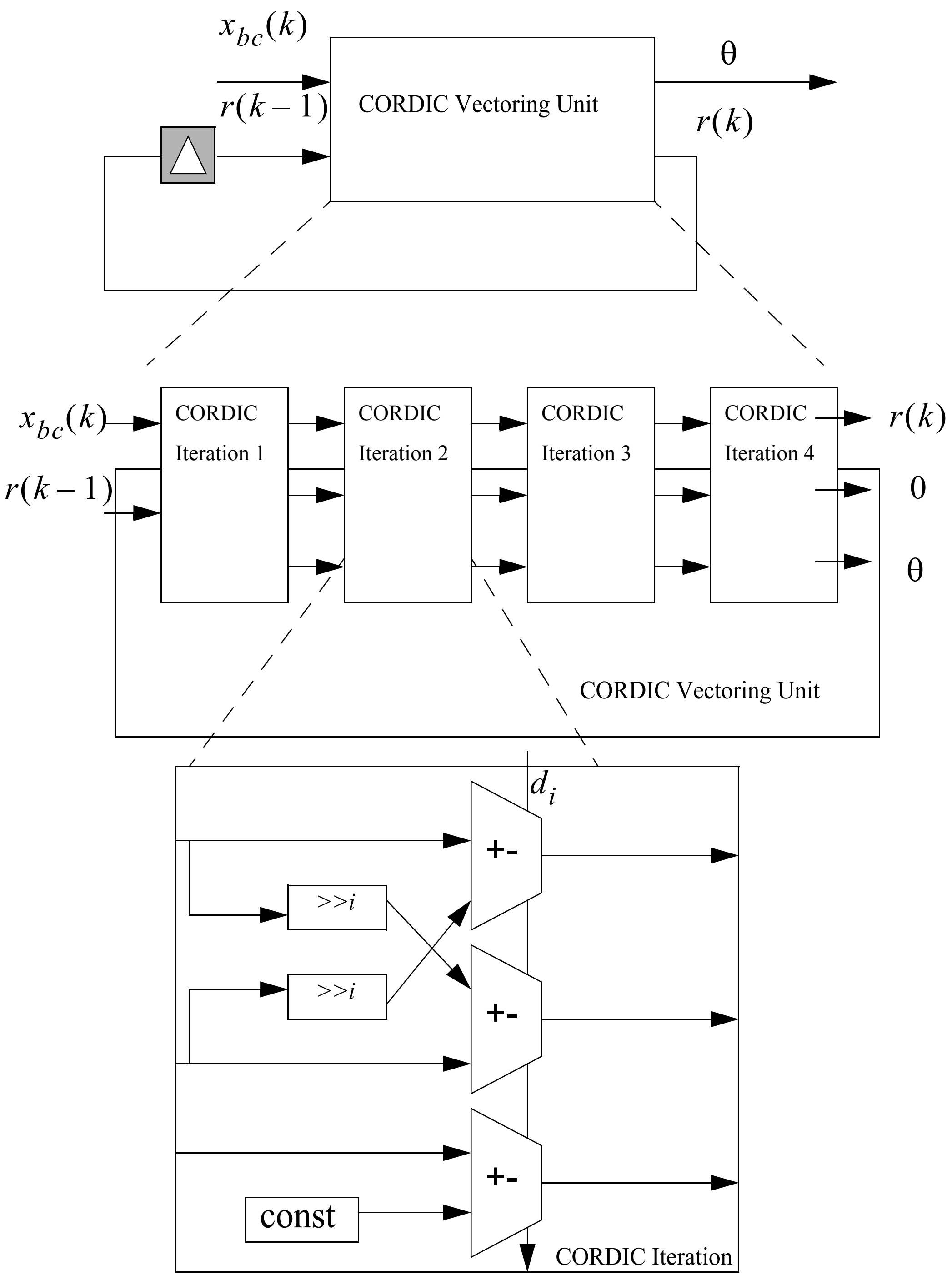


Figure 8.1: A Detailed Look at CORDIC Based Boundary Cell (Givens Generation)

Due to the presence of the feedback loop, it is not possible to pipeline inside the CORDIC rotation block without affecting the numerical behaviour of the algorithm. It is also commonplace to select the number of iterations of the CORDIC algorithm as being equal to one less than the input wordlength. It is unfortunate then, that if the working precision at the input is increased by N bits, then the number of CORDIC iterations must also increase by $N-1$, and therefore the critical path will increase. Without pipelining being performed inside the QRD cells, it is not possible to increase the precision of the computation without affecting the rate at which it is executed.

8.1.2 Look Ahead Technique

The problem of inserting pipeline registers where a feedback loop is present is clearly not isolated to QRD-RLS. Pipelining a system where recursion is present is a well studied problem. The look ahead technique [31] is a well known method for allowing pipeline delays to be inserted where feedback is present. In order to reduce the iteration bound of the algorithm, additional concurrency is created in the feedforward section of the algorithm, which allows for registers to be inserted in the feedback section of the algorithm.

The look ahead technique has been applied to the problem of pipelining IIR filter structures in [32]. A simple example can be used to explain the concept. Consider a first order IIR digital filter which can be described by [8.1].

$$\boldsymbol{y}(k+1) = a \cdot \boldsymbol{y}(k) + b \cdot \boldsymbol{x}(k) \qquad [8.1]$$

The recursion when looking ahead one sample is represented by [8.2].

$$\boldsymbol{y}(k+2) = a[a\boldsymbol{y}(k) + b\boldsymbol{x}(k)] + b\boldsymbol{x}(k+1) \qquad [8.2]$$

We can observe that [8.1] and [8.2] have the same iteration bound. If however, we recast [8.2] into the form given in [8.3], we can pipeline the multiplier in the feedback path, achieving a speed up of two.

$$\boldsymbol{y}(k+2) = a^2\boldsymbol{y}(k) + ab\boldsymbol{x}(k) + b\boldsymbol{x}(k+1) \qquad [8.3]$$

In the z-domain, such a transformation corresponds to inserting extra poles and zeros in the unit circle which cancel one another out. Extra concurrency is created in the system without affecting the numeric behaviour. Figure 8.2 shows the look ahead technique applied in [8.1], [8.2] and [8.3] in terms of the corresponding Signal Flow Graph at each stage.

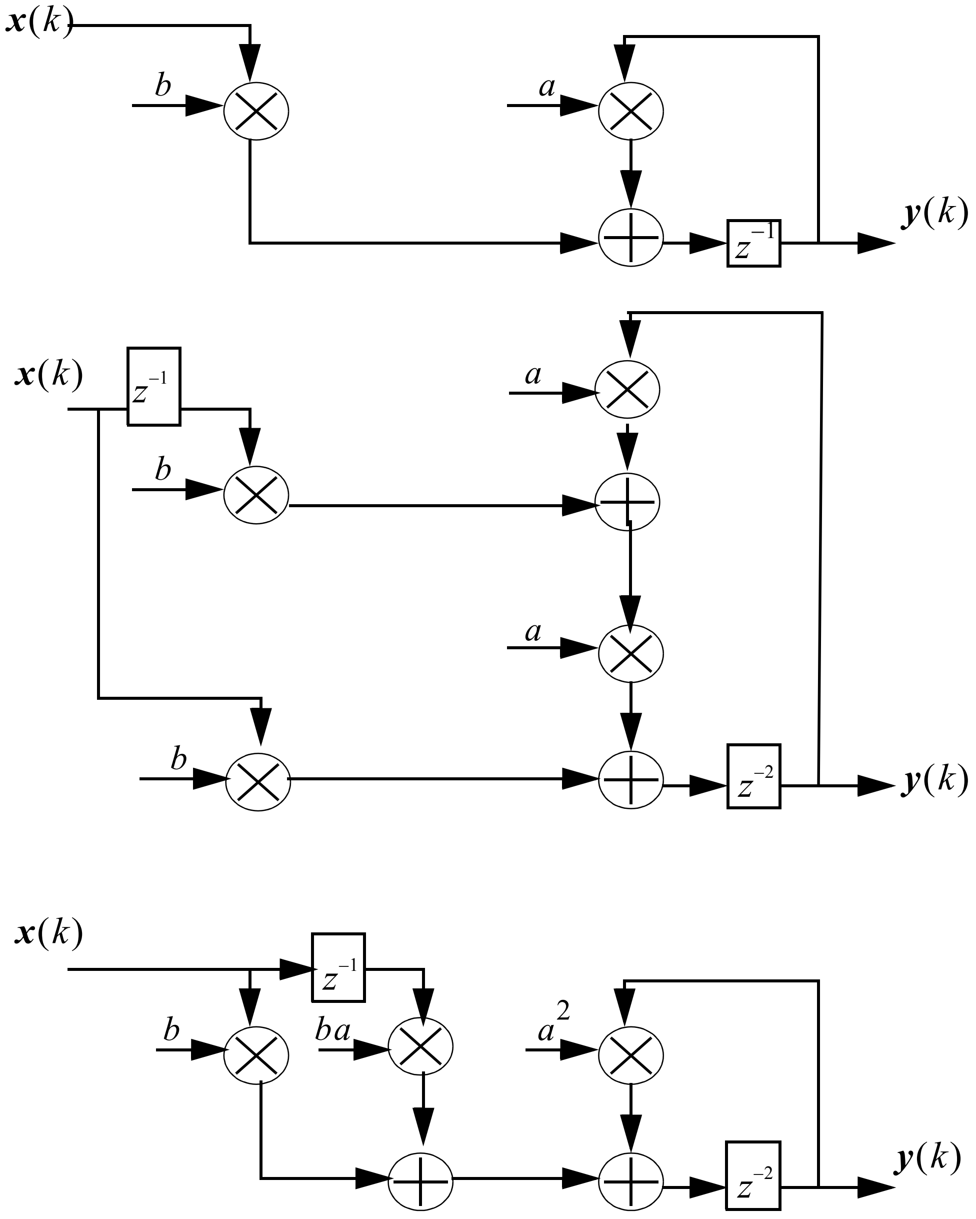


Figure 8.2: Look Ahead Applied to 1st Order IIR Digital Filter

8.1.3 Annihilation-Reordering Look Ahead QRD-RLS

The look ahead technique has been applied to the problem of QRD-RLS systolic array implementation in [34], to produce a QRD-RLS systolic array which can be arbitrarily pipelined without loss of orthogonality. It is assumed that CORDIC arithmetic is used in both the boundary and internal cells.

In order to introduce additional concurrency to the QRD update process, each upper triangular element, r, is formed as the update of a block of input data, size M, where M is the desired speedup. The operations of the boundary cells in a [4x4] QR decomposition, where a speedup of 3 is desired are shown in Figure 8.3.

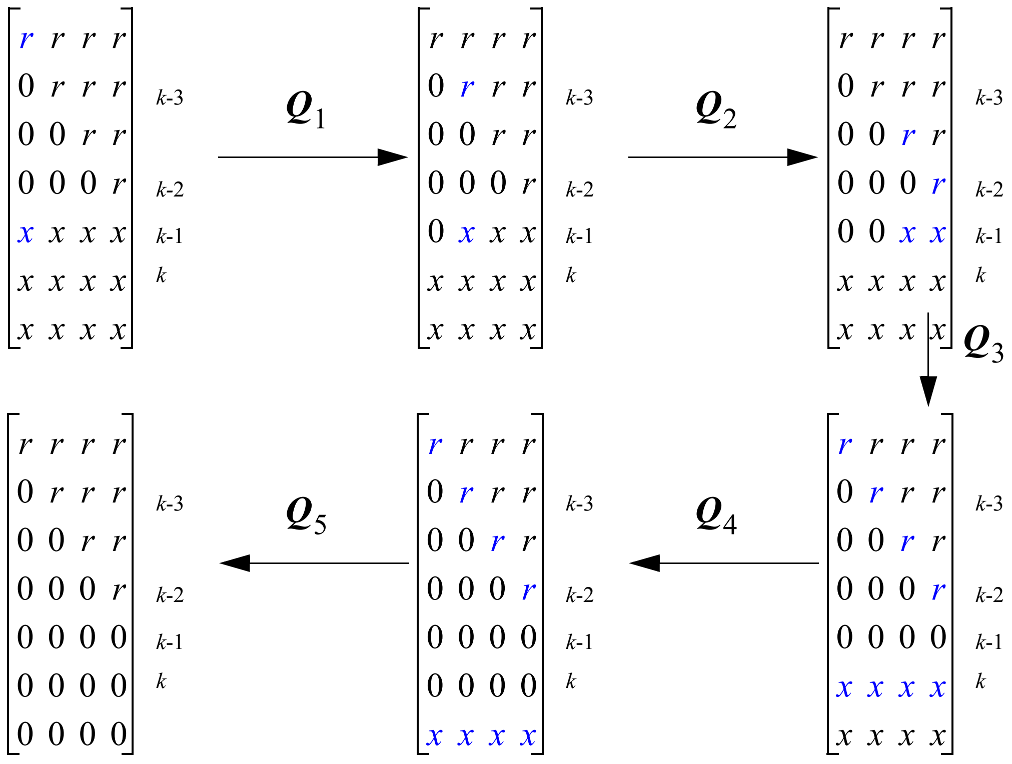


Figure 8.3: Block Update QR Decomposition

The recursive update of the r elements are computed with each time step, and so pipeline registers cannot be placed across the feedback section in the current form. The annihilation reordering transformation instead annihilates the block input data in a column by column fashion, meaning that the diagonal r elements are only updated at the last time step (Figure 8.4).

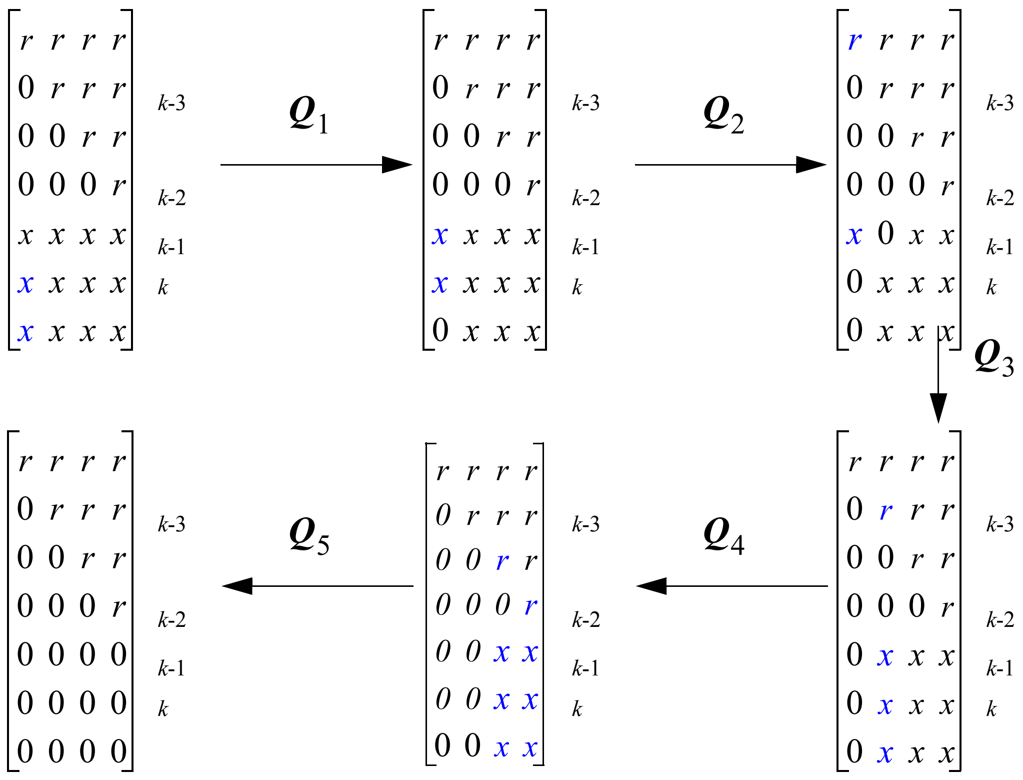


Figure 8.4: Annihilation Reordering Look Ahead QRD-RLS

The resulting hardware architecture of the processing element, shown in Figure 8.5, allows the extra latency created by the transformation to be applied across the CORDIC cell contained in the feedback loop, [44].

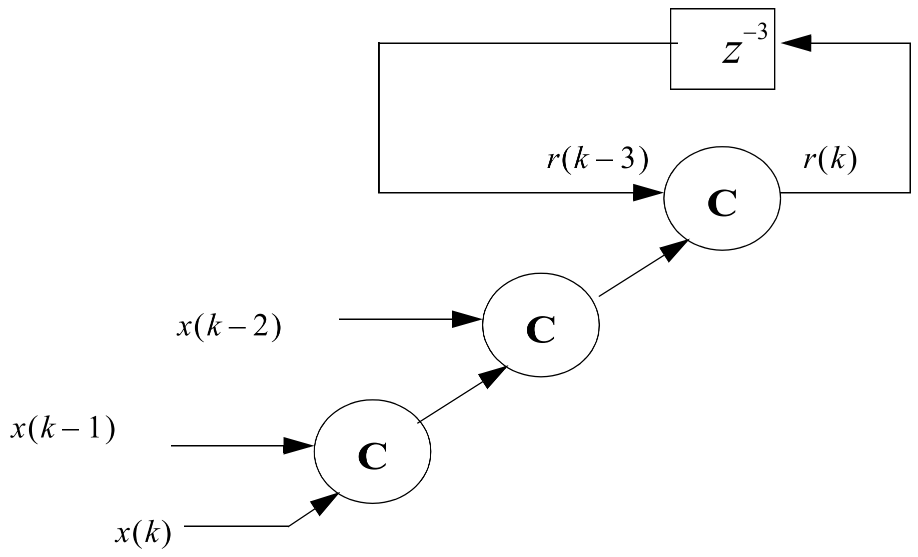
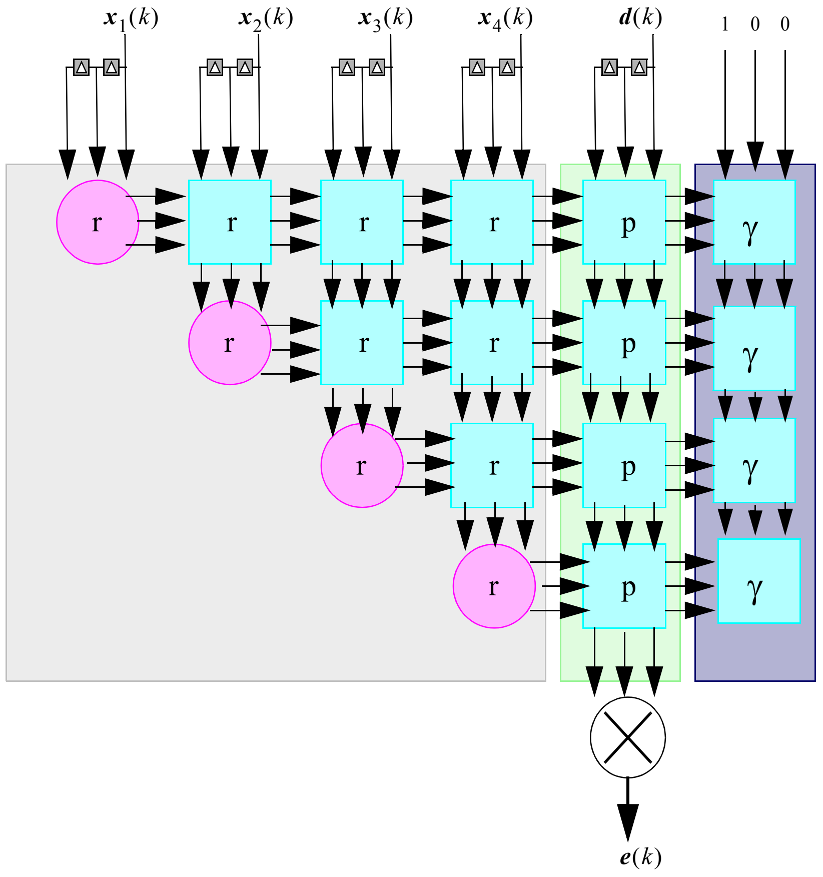


Figure 8.5: Pipelined CORDIC Processing Element

The resulting [4x4] systolic array obtained after annihilation reordering look ahead is applied with a desired speedup of 3 is shown in Figure 8.6. The increased performance resulting from additional pipeline registers results in a much greater hardware overhead. This is the trade-off of the annihilation reordering look ahead transformation, increased throughput is gained at the expense of greater hardware complexity.



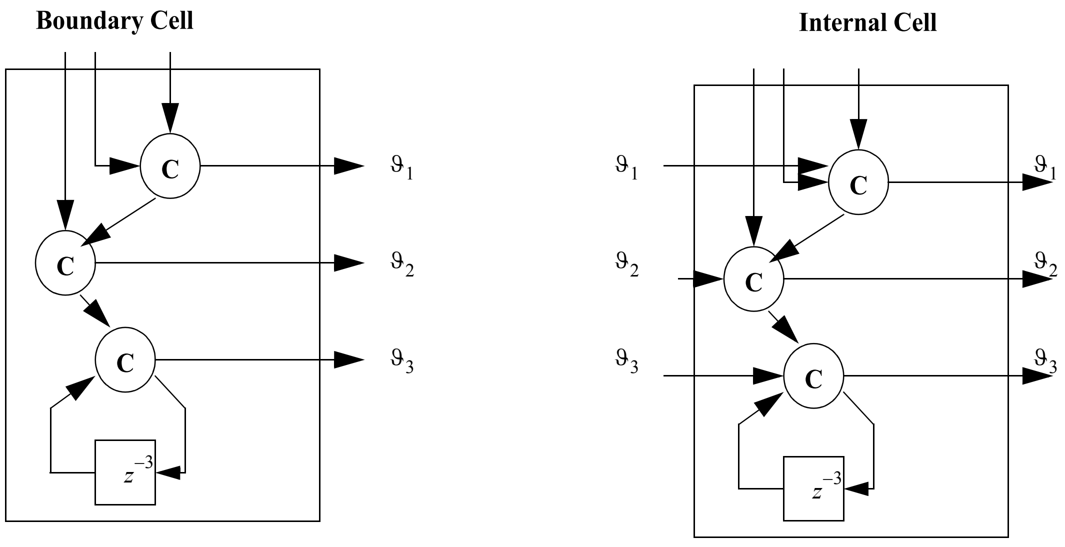


Figure 8.6: Annihalation Reordering Look Ahead Pipelined Systolic Array

8.2 Resource Sharing Folded Systolic Array

While the previous section considered techniques to allow for the maximum possible throughput from a QRD-RLS systolic array architecture, there are also scenarios where the required throughput is much lower than that provided by the fully parallel systolic array architecture. In such instances, individual processing elements may be sitting idle for a large percentage of the time.

When folding the QRD-RLS systolic array architecture, either a linear array or processing element style architecture can be chosen, as shown in Figure 8.7. In the linear array style of architecture, the fully parallel QRD-RLS systolic array architecture is folded onto a single row (the largest of the rows in the systolic array). The data rate is then reduced to f_s/N, where N is the number of rows, [36]. Further folding results in the internal and boundary cells being folded onto a reduced subset of cells, which together comprise a single QR processor.

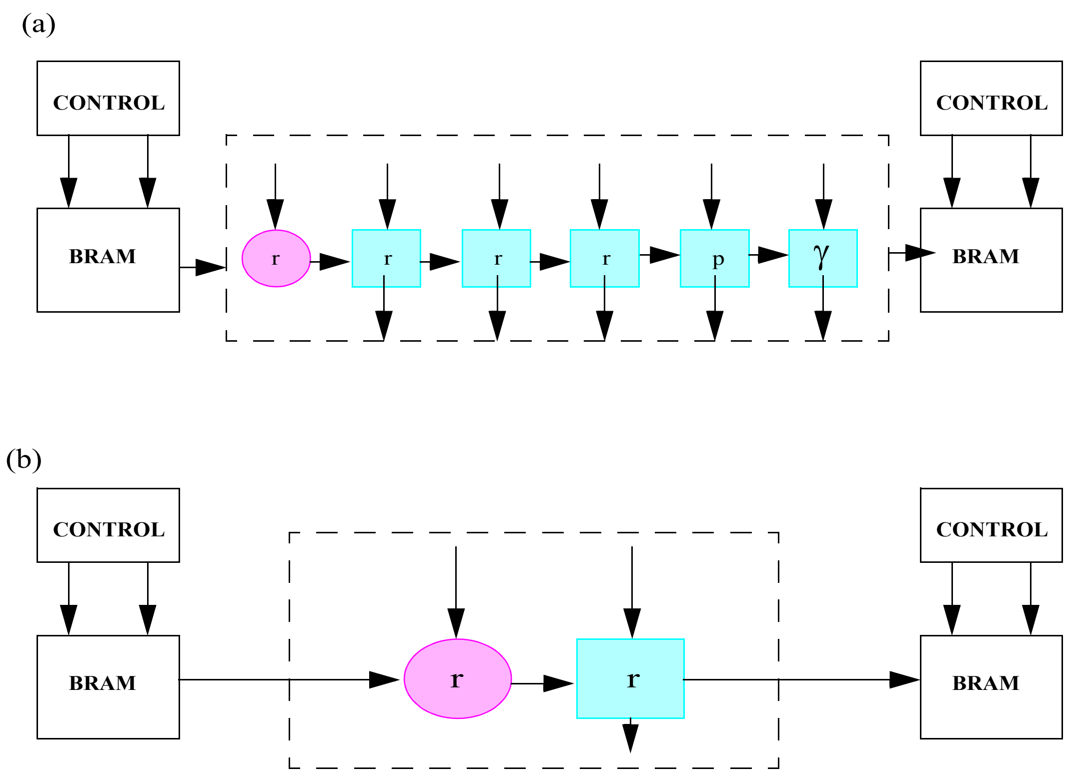


Figure 8.7: Folded QRD RLS Systolic Array: (a) Linear Array (b) Processing Element

Mapping the fully parallel QRD-RLS systolic array onto a single QRD-RLS processor is an interesting problem. Due to the different functionality of the internal and boundary cells, in [24], two separate CORDIC blocks are used for internal and boundary cells, i.e. in vectoring and rotation modes. The implementation in [22] is similar, however a MAC is used instead of CORDIC rotation for the internal cell.

The work in [36] seeks to alleviate the requirement for two distinct components to compute boundary cell and internal cell functions. In this implementation, a modified version of the CORDIC algorithm, referred to as Coarse-Angle Rotation Mode CORDIC is introduced. The modified version of the CORDIC algorithm regularizes the function of the CORDIC processing element by removing the angle data path. Instead of calculating the total angle rotated in order to annihilate an element of the input vector, the decision, d_i, taken at each iteration is stored. Instead of the decision factor being directly computed in the internal cell, it can therefore be applied at each stage, retrieving the stored decision factors previously calculated in vectoring mode.

9 Comparison to Gram Schmidt Implementation

Although the Givens rotation based systolic array is immediately favourable for FPGA/ ASIC implementation, there are scenarios where Gram-Schmidt/Householder methods can be considered. To allow a balanced conclusion to be drawn between the different methods, in this section we review an implementation using Gram Schmidt for a complex MIMO receiver.

The implementation of Gram Schmidt for complex valued QR Decomposition is considered in [38]. The task is to perform QR decomposition targeted at a 3G LTE system. Note that the paper was published in 2008, prior to the LTE standard being formalized. Deployed LTE systems use a pre-coding matrix based on Householder transformation, which allows for an MMSE equaliser to be used in practice, see [39] for an overview.
The requirements of the LTE standard are considered in order to influence the hardware architecture. The authors assume a [4x4] matrix is to be decomposed, from the format specified in [40].The authors relate the period over which this can be performed whilst keeping pace with the input data to the coherence time of the channel, i.e. the period in which the impulse response is stationary, defined by [9.1].

$$t_{coh} = c/(vf_c) \qquad [9.1]$$

With $c = 3 \times 10^8$ m/s being the speed of light, $v = 250$ km/h being the speed of the receiver and $f_c = 2.4$ GHz being the carrier frequency, the coherence time is calculated as 1.8ms. For LTE, Orthogonal Frequency Division Multiplexing (OFDM) is used, which gives rise to a maximum of 2048 subcarriers, as defined in [41]. Therefore the QR decomposition of the [4x4] input matrix should be computed in $(1.8ms)/2048 = 8.7\mu s$. This is a relatively relaxed requirement.

In addition to the requirement in terms of computation time, the reusability of the hardware for other high level functions required in LTE is considered. In addition to MIMO decoding, the Fast Fourier Transform (FFT) is another high level function required, for ODFM modulation/demodulation. To compute the FFT, complex valued arithmetic is required; the authors therefore propose that highly specialized systolic array processing cells in the form of CORDIC vectoring/rotation units are not suitable as they have limited potential for re-use.

With the aforementioned requirements in mind, a general purpose processing architecture based on Transport Triggered Architecture (TTA) is proposed [42]. This is a form of computing architecture which allows for parallel computing resources, such as complex mutlipliers, adders, sqrt etc. to be effectively shared for multiple functions. Each resource is denoted as a Functional Unit (FU), and are interconnected by a shared bus, as shown in Figure 9.1.

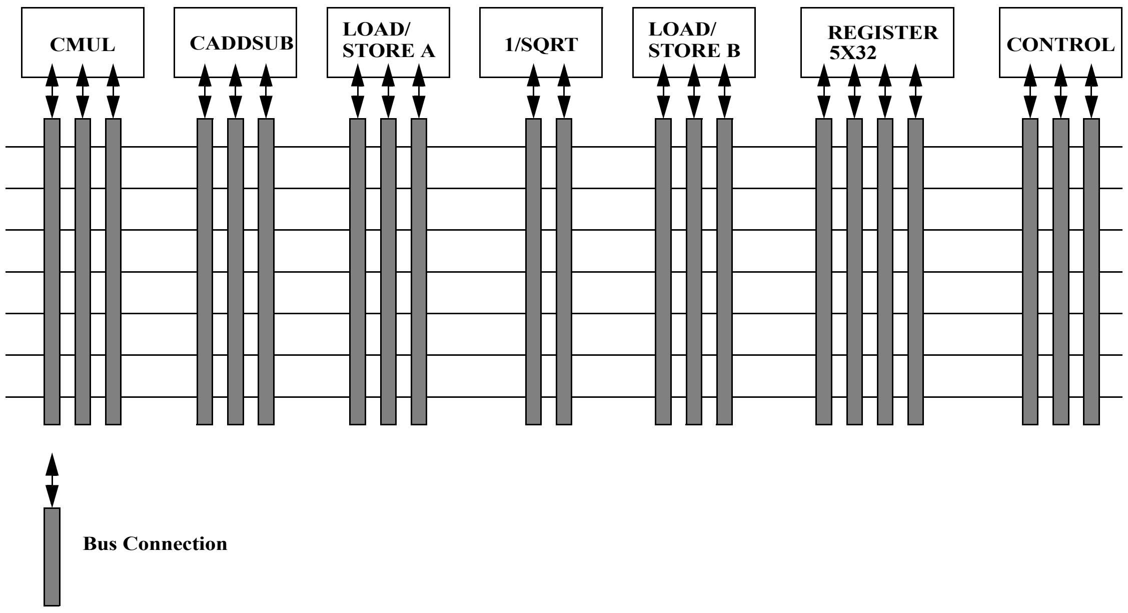


Figure 9.1: Modified Gram Schmidt TTA Processing Architecture

The most demanding module of the processor in terms of computation is the inverse square root operation. To simply the operation, an approximation is used, which exploits the fact that a fixed wordlength of 16 bits is used throughout the architecture, with 4 integer, 11 fractional and 1 sign bit. Firstly, instead of modelling the highly non linear $1/(\sqrt{x})$, the term $1/(\sqrt{1+u})$ is substituted, where u is a shifted version of x, represented by [9.2].

$$x = 0.000, \ldots, 01u \qquad [9.2]$$

where the trailing zeros are denoted by α. Therefore:

$$x \times 2^{\alpha} = 1.u \qquad [9.3]$$

Then with the substitution in place:

$$\frac{1}{\sqrt{x}} = \frac{1}{\sqrt{2^{-\alpha}(1+u)}} = 2^{\frac{\alpha}{2}} \frac{1}{\sqrt{1+u}} \qquad [9.4]$$

As the non-linearity is now softened in the square root calculation, a first order polynomial can be used to approximate the calculation. Seeing as the wordlength is fixed in this case, constant terms for computing the first order polynomial which yield low cost shift arithmetic are obtained via exhaustive search, yielding the expression given in [9.5].

$$\frac{1}{\sqrt{1+u}} \cong 0.965820 - \frac{1}{4}u - \frac{1}{32}u \qquad [9.5]$$

The resulting hardware architecture is able to compute the QRD of 2048 [4x4] input matrices within the time frame given by the coherence time of 1.8ms, using a master clock running at 160MHz. The architecture is generic enough that with minor modification, it can be used to compute other high level functions such as FFT. A fully folded QRD-RLS array could be used to perform the same process, likely with lower hardware cost, due to the use of CORDIC arithmetic, however this would be less flexible for use by other functions.

10 Conclusion

The purpose of this work was to determine the key techniques, challenges and research trends in the implementation of adaptive filters. To achieve this, two objectives were set out at the beginning of the project:

- Complete a practical implementation of the LMS algorithm, investigating architectures of varying parallelism.
- Carry out a survey of the QRD-RLS algorithm to determine the state of the art in terms of implementation architectures.

In this concluding section we will investigate some of the common themes between LMS and QRD-RLS implementation, along with future research directions in the topics touched upon in the thesis, before giving some final concluding remarks.

Common Topics Between LMS and QRD-RLS Implementation

Having studied both LMS and QRD-RLS implementations, it is possible to identify common techniques, considerations and challenges between the two algorithms. In this subsection, the key overlapping areas shall be discussed.

In both implementations the application of pipelining proved difficult. Intuitively, we can understand that this is due to the recursive operations present in the algorithm. For the LMS algorithm, there does not exist a solution which does not result in alteration of the numerical behaviour of the algorithm, whereas for the QRD-RLS algorithm, the only solution which does not alter the numerical behaviour of the algorithm, results in a linear increase in the complexity of the resulting hardware.

Both algorithms can also be unfolded in a variety of ways. The LMS algorithm perhaps offers more flexibility in this regard, as the parallel-serial architecture allows for many combinations of coefficients per processor core. The analog in terms of QRD-RLS unfolding would be the linear array style of architecture, however in this case, the array must be sized as the row with the maximum number of columns. For subsequent rows, an incremental number of processing elements are left idle. This is in contrast to the parallel-serial LMS architecture where every processing element is fully occupied at every time step.

Future Research Direction

As with any project, time was a major constraint, therefore there were parts of the LMS compiler, that if given extra time could have been improved.

In order to form the serial and parallel-serial architectures, control logic was created in order to regulate the flow of input data to the MAC units used to implement the algorithmic operations. In both of the implementations an Addressable Shift Register construct was used to implement the queuing system for the input data. In order to improve timing in the generated LMS compiler circuit, BRAM should be used to implement the input queueing system as an alternative. Due to the coding style used in the LMS compiler, the ASR must be built out of gates, registers and multiplexer logic. Such logic incurs a high critical path on the device due to both combinatorial logic used to select elements from the shift register, and the high fanout of multiplexing logic. Using embedded BRAM allows for low latency memory access to be achieved. This may have allowed the synthesis results to be improved. However, seeing as there is one cycle latency requirement for each write/read operation that is performed, the surrounding control logic may require some redesign to incorporate such a modification.

In addition to the modifications to the memory architecture used in the LMS compiler, further work could be carried out in the pipelining of the serial and parallel-serial architectures. Although in both the fully serial and parallel-serial cases, the input and output of the multiplication operations is pipelined, there are two coding styles used in the VHDL which likely inhibit the registers from being pushed into the DSP48 slice itself for maximum throughput processing. Firstly, the pipeline stages are not inserted at the same level of hierarchy as the multiplication operation itself. This potentially inhibits the synthesis tool from pushing the added registers into the DSP48 slices. Therefore, the pipeline stages should be firstly moved into the same component as the multiplication operation. Once this has been achieved, then carrying out both the multiplication operation and pipelining operation in the same process statement is preferable to ensure that the synthesized circuit will utilize the high speed pipeline registers located inside the DSP48 slice.

With the two modifications discussed for the LMS algorithm in place, and therefore the critical path in the architecture truly minimized, it is possible that the serial and parallel-serial architectures could achieve clock frequencies in the region of 350MHz. From the presented implementation results, we can see that the fully parallel implementation of the complex valued LMS algorithm runs under 25MHz for a 30 coefficient filter. Using the parallel-serial LMS architecture with 10 PCs computing 3 coefficients each and running at the hypothetical 350MHz clock rate, the resulting sampling frequency is given by [10.1].

$$f_s = \frac{f_{clock}}{ceil(\log(PC)) + 9} = \frac{350}{13} = 26MHz \qquad [10.1]$$

Therefore, if the LMS compiler is modified to achieve higher clock frequency, then the resulting filter can be achieved using the parallel-serial option of the LMS compiler, at reduced hardware cost and higher processing rate.

Regarding the research carried out on QRD-RLS implementation, it seems that the pipelining of the systolic array architecture has the potential to be studied further. The current limiting factor of the intra-cell latency is quite limiting for FPGA implementation. A fully parallel CORDIC architecture of 16 iterations for example, is not likely to run above 100MHz, the critical path of 16 adders and other related combinatorial logic is too great to achieve such rates. Although the Annihilation Reordering Look Ahead Transformation can allow for arbitrary pipelining to be applied across the internal cells, there is a linear increase in the hardware complexity, which can result in hardware of great complexity being required.

Concluding Remarks

In the beginning of this thesis, it was implied that the superior convergence properties of the QRD-RLS algorithm would result in widespread usage as adaptive equalizers in future communications systems. It is therefore interesting to note the trend towards reducing the processing requirements of the equalizer in wireless MIMO 4G and 5G communication systems via precoding stage. This decision in the system design results in the LMS algorithm still being in use in cutting edge wireless communications systems, even five decades after it was first presented in the literature. The low complexity implementation, combined with relative robustness to noise results in LMS still being a very popular algorithm.

Although the LMS algorithm is relatively straightforward to implement in the fully parallel form, the serial and parallel-serial architectures are quite complex in terms of the memory and control structures used to shuttle data to/from the shared components. This highlights the difficulty of FPGA design, where every construct (memory, counter etc.) must be explicitly instantiated in the HDL implementation. For engineers working on large systems, it is important to operate at the macro level, rather than spending much of the time working on small, micro-level details. This highlights the requirement for the IP industry, which provide system designers with high level interfaces to control the corresponding low level detail.

The research into QRD-RLS implementation highlighted the issues with the explicit weight extraction architectures. The backsubstitution architecture was shown to be ill suited for implementation on the FPGA due to the numerical ill conditioning of the division operations and the requirement to halt adaptation of the systolic array while the backsubstitution routine is performed. While the downdating architecture uses only multiply and add operations, and does not require halting of the systolic array adaptation, the inverse forgetting factor used in the processing elements amplify errors over time, and hence numerical stability cannot be guaranteed without periodic reinitialization to clear the accumulated error. Although the issues with explicit weight extraction architectures exist, the implicit weight extraction architecture in the form of McWhirter's systolic array with error signal generation is both well suited for FPGA implementation, and numerically stable. Many applications such as adaptive beamforming in phased array systems only require the error signal in order to steer the array, the weight vector is not needed.

As both the data throughput requirements and general complexity of wireless systems increase, we can expect that a QRD-RLS compiler will be required in the coming years. It will be interesting to see if there will be such a time when the requirements of the system mean that RLS algorithms become the norm for implementation rather than the classic LMS algorithm which is still predominately the most popular choice.

11 References

[1] S.Haykin, *"Adaptive Filter Theory"*, Fourth Edition, Prentice Hall, 2002.

[2] A.H.Sayed, *"Adaptive Filters"*, First Edition, Wiley, 2011

[3] S.Qureshi, *"Adaptive Equalisation"*, Proceedings of the IEEE, vol.73, pp. 1349-1387, 1985.

[4] B.Widrow, J.McCool, M.Ball, *"The Complex LMS Algorithm"*, Proceedings of the IEEE, vol. 63, Issue 4, pp. 719-720, 1975.

[5] Yi, Y., Woods, R.Woods, Ting, L.K.,Cowan, C.F.N., *"High Speed FPGA-Based Implementations of Delayed-LMS Filters",* The Journal of VLSI Signal Processing, Springer, vol. 39, pp. 113-131, 2005

[6] A.P, Liavas, P.A. Regalia, *"On the Numerical Stability and Accuracy of the Conventional Recursive Least Squares Algorithm",* IEEE Transactions on Signal Processing, vol. 47, no. 1, pp. 88-96, 1999

[7] G.E. Bottomley, *"A Novel Approach for Stabilizing Recursive Least Squares Filters"*, IEEE Transactions on Signal Processing, vol. 39, Issue 8, pp. 1770-1779,1991

[8] R.L. Smith, *"Algorithm 116: Complex Division"*, Communications of the ACM", vol. 5, Issue 8, pp. 435, 1962

[9] R.Woods, J.Mcallistar, Y.Yi, G.Lightbody, *"FPGA-based Implementation of Signal Processing Systems"*, First Edition, Wiley, 2008.

[10] Y.K. Wong, *"An Application of Orthogonolization Process to the Theory of Least Squares",* The Annals of Mathematical Statistics, vol. 6, no. 2, pp. 53-75, 1935

[11] A. Bjorck, *"Numerics of Gram Schmidt Orthogonolization",* Linear Algebra and its Applications, vol. 197-198, pp. 297-316, 1994

[12] A. Bjorck, C.C. Paige, *"Loss and Recapture of Orthogonality in the Modified Gram-Schmidt Algorithm",* Siam Journal on Matrix Analysis and Applications, vol. 13, no.

1, pp. 176-190, 1992

[13] A. S. Householder, *“Unitary Triangularization of a Nonsymmetric Matrix, Journal of the ACM*, vol. 5, no. 4, pp. 339-342, 1958

[14] LAPACK QR Documentation, available from - http://www.netlib.org/lapack/lug/node40.html

[15] W.Givens, *“Computation of Plane Unitary Rotations Transforming a General Matrix to Triangular Form”*, Journal of the Society for Industrial and Applied Mathematics, vol. 6, no. 1, pp. 26-50, 1958

[16] W. M. Gentleman, *“Error Analysis of QR Decomposition by Givens Transformations”*, Linear Algebra and its Applications, vol. 10, no. 3, pp. 189-197, 1975

[17] W.M. Gentleman and H.T. Kung, *“Matrix Triangularization by Systolic Arrays”*. SPIE Proceedings on Real-Time Signal Processing IV, vol. 298, pp. 19-26, 1981

[18] Volder, Jack E., *"The CORDIC Trigonometric Computing Technique,"* IRE Transactions on Electronic Computers, vol. 8, no. 3, pp. 330-334, 1959

[19] J. S. Walther, *“A Unified Algorithm for Elementary Functions”*, in Proc. Sprzng Joint Computer Conf., Atlantic City, NJ, pp. 379-385, 1971

[20] Andraka, R. *“A Survey of CORDIC Algorithms for FPGA Based Computers”*, http://www.andraka.com/files/crdcsrvy.pdf

[21] B. Haller, J.Gotze, J.R. Cavallero, *“Efficient Implementation of Rotation Operations for High Performance QRD-RLS Filtering”*, IEEE International Conference on Application-Specific Systems, Architectures and Processors, pp. 162 -174, 1997

[22] C. Dick., F. Harris, M. Pajic, D. Vuletic, *"Real-Time QRD-Based Beamforming on an FPGA Platform,"* Fortieth Asilomar Conference on Signals, Systems and Computers, pp. 1200-1204, 2006

[23] Xilinx DSP48 User Guide, avaialable from - http://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

[24] Altera White Paper, *“Implementation of CORDIC-Based QRD-RLS Algorithm on Altera Stratix FPGA with Embedded Nios Soft Processor Technology”*, https://www.altera.com/en_US/pdfs/literature/wp/wp_qrd.pdf

[25] J. G. McWhirter, *“Recursive Least Squares Minimization using a Systolic Array”*, Electronics Letters, IEEE, vol. 19, no. 18, pp. 729-730, 1983

[26] J. A. Apolinaro, M.D Miranda, *“QRD RLS Adaptive Filtering”*, First Edition, Springer, Chapter 3, pp. 60-64

[27] Ward, C.; Hargrave, P.; McWhirter, J.G., *"A Novel Algorithm and Architecture for*

Adaptive Digital Beamforming,", IEEE Transactions on Antennas and Propagation, vol. 34, no. 3, pp. 338-346, 1986

[28] Bin Yang; B Ahme, Johann F., *"Rotation-based RLS Algorithms: Unified Derivations, Numerical Properties, and Parallel Implementations,"*,IEEE Transactions on Signal Processing, vol. 40, no. 5, pp. 1151-1167, 1992

[29] M Harteneck, R.W Stewart, J.G. McWhirter, I.K Proudler, *"Algorithmic Engineering Applied to the QR-RLS Algorithm"*, Proceedings of 4th International Conference on Mathematics in Signal Processing, 1996

[30] TT.J Shepard, J. Hudson, *"Parallel Weight Extraction from a Systolic Adaptive Beamformer"*, Proc IMA Conference on Mathematics in Signal Processing, 1988

[31] Parhi, K.; Messerschmitt, D.G., *"Look-Ahead Computation: Improving Iteration Bound in Linear Recursions,"*, IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 12, pp. 1855-1858, 1987

[32] Parhi, K.K.; Messerschmitt, D.G., *"Pipeline Interleaving and Parallelism in Recursive Digital Filters. I. Pipelining using Scattered Look-Ahead and Decomposition,"* , IEEE Transactions on Acoustics, Speech and Signal Processing, vol. 37, no. 7, pp. 1099-1117, 1989

[33] K.K. Parhi, *"VLSI Digital Signal Processing Systems: Design and Implementation, Chapter 10: Pipelined and Parallel Recursive and Adaptive Filters"*, *Example taken from slides, Available at: http://www.ece.umn.edu/users/parhi/SLIDES/chap10.pdf, 1999*

[34] Jun Ma; Parhi, K.K.; Deprettere, E.F., *"Annihilation-Reordering Look-Ahead Pipelined CORDIC-based RLS Adaptive Filters and their Application to Adaptive Beamforming,"* IEEE Transactions on Signal Processing, vol. 48, no. 8, pp. 2414-2431, 2000

[35] Lan-Da Van; Chih-Hong Chang, *"Pipelined RLS Adaptive Architecture using Relaxed Givens Rotations (RGR),"* . ISCAS. IEEE International Symposium on Circuits and Systems, vol. 1, pp. 37-40, 2002

[36] Q. Gao, L. Crockett, R.W. Stewart, *"Coarse Angle Rotation Mode CORDIC Basic Single Processing Element QR-RLS Processor"*, 17th European Signal Processing Conference, 2009

[37] Gao, L.; Parhi, K.K., *"Hierarchical Pipelining and Folding of QRD-RLS Adaptive Filters and its Application to Digital Beamforming,"* IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, vol. 47, no. 12

[38] Salmela, P.; Burian, A.; Sorokin, H.; Takala, J., *"Complex-valued QR Decomposition Implementation for MIMO Receivers,"*, IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 1433-1436, 2008

[39] J. Lee, J.K Han, C. Zhang, *"MIMO Technologies in 3GPP LTE and LTE Advanced"*, Eurasip Journal on Wireless Communication and Networking, 2009

[40] "3GPP TR 25.876 *Multiple Input Multiple Output in UTRA"*, 3rd Generation Partnership Project, Tech Rep, 2005

[41] R. Bachl, P.Gunreben, S. Das, S.Tasesh, *"The Long Term Evolution Towards a New 3GPP Air Interface Standard"*, Bell Labs Technical Journal, vol. 11, no. 4, pp. 25-51, 2007

[42] H. Corporaal, *"Design of Transport Triggered Architectures"*, in 4th Great Lakes Symposium Design Automation of High Performance VLSI Systems, Notre Dame, IN, USA, pp. 130-135, 1994.

[43] Yi-Gang Tai; Chia-Tien Dan Lo; Psarris, K., *"Applying Out-of-Core QR Decomposition Algorithms on FPGA-Based Systems,"* International Conference on Field Programmable Logic and Applications, pp. 86-91, 2007

[44]K.K Parhi, J. Ma, *"QRD RLS Adaptive Filtering"*, First Edition, Springer, *Chapter 10 - "On Pipelined Implementations of QRD-RLS Adaptive Filters"*

12 Acknowledgements

First of all, I would like to thank my supervisor, Prof Robert Stewart for motivating me to keep writing up this thesis, and for helping me to understand key aspects of adaptive filtering theory. Thank you for all of the DSP and FPGA insights you have shared over the last few years and for helping to point me in the right direction in my studies.

I am also deeply indebted to Dr Louise Crockett for reading through the drafts of the thesis and offering much needed feedback. I couldn't have finished this piece of work without the time you spent reviewing it. Thank you so much.

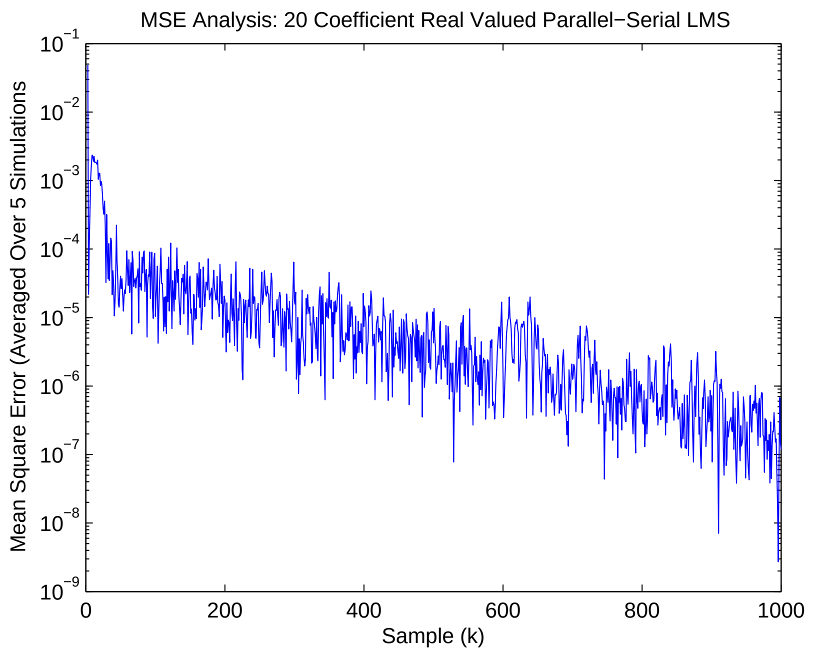
I would also like to thank my MathWorks managers, Bharath Venkataraman and Garrey Rice for encouraging me to write up this thesis. Many thanks to both of you.

I will also take the opportunity to thank my colleagues in the MathWorks Glasgow office for many fruitful discussions and support. Often times a seemingly insurmountable issue is found to be trivial after talking it through.

Lastly but by no mean least, I thank my parents for all of the support and encouragement they have provided over the years.

13 Appendix A: Additional Numerical Performance Analysis Results

13.1 Real Valued Arithmetic Parallel-Serial LMS Filter Results



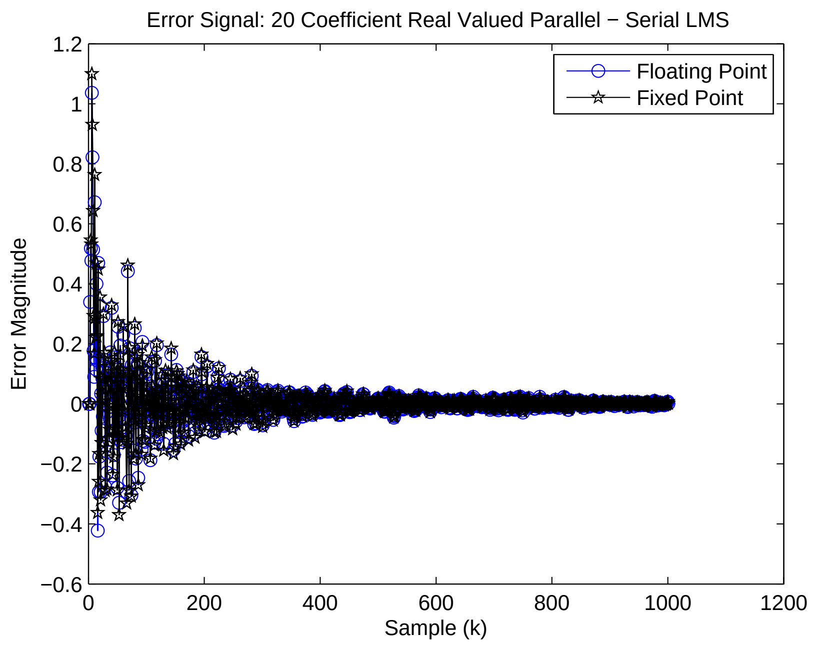
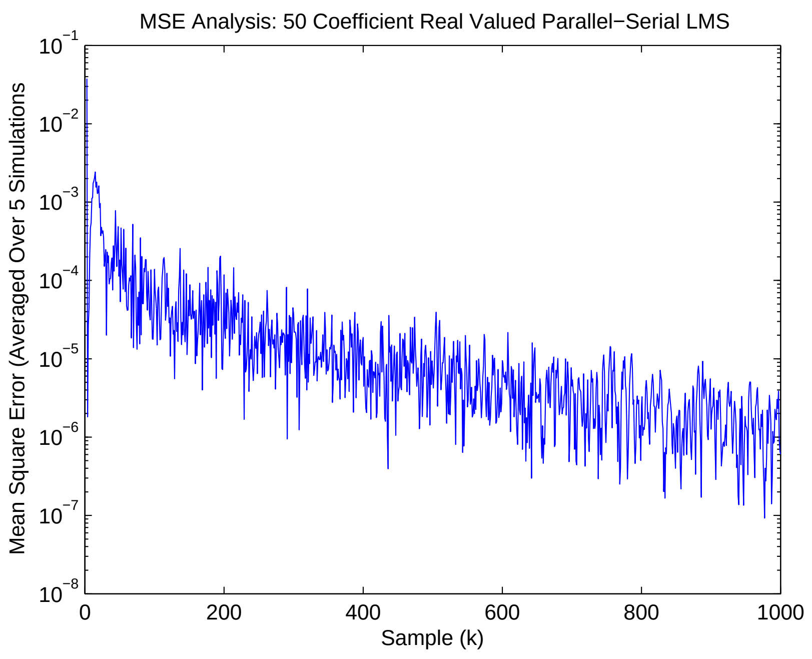


Figure 13.1: Test Case 3: 20 Coefficient Real Valued Parallel - Serial LMS



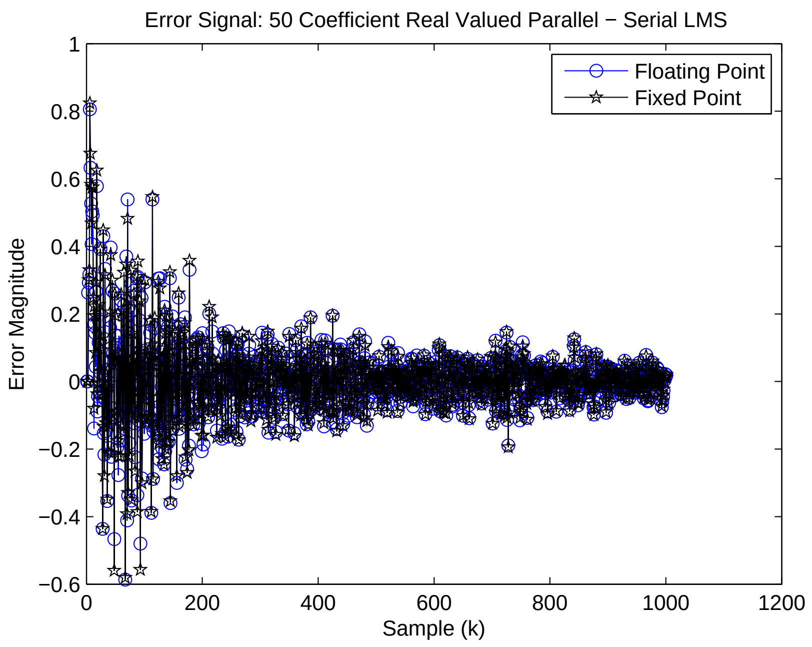
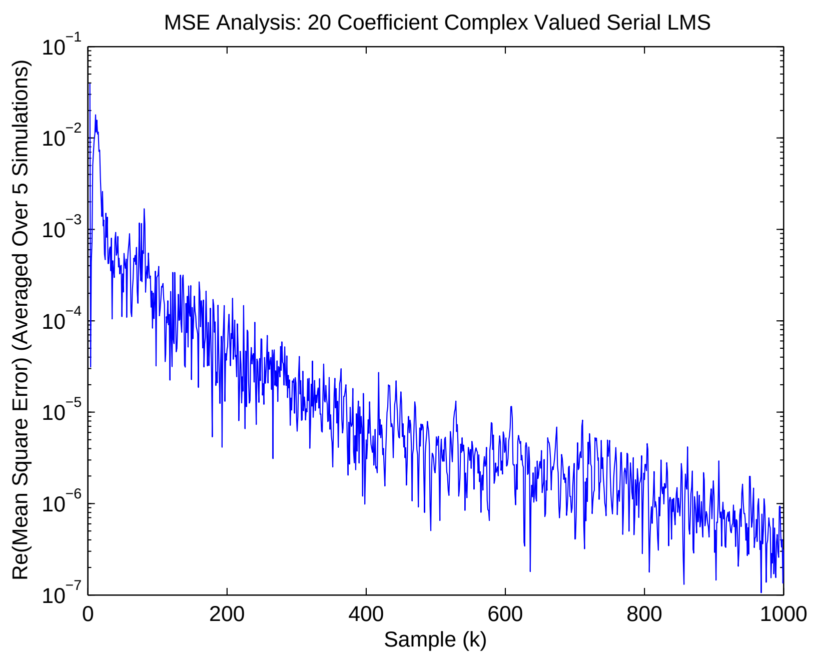


Figure 13.2: Test Case 4: 50 Coefficient Real Valued Parallel - Serial LMS

13.2 Complex Valued Arithmetic Serial LMS Filter Results



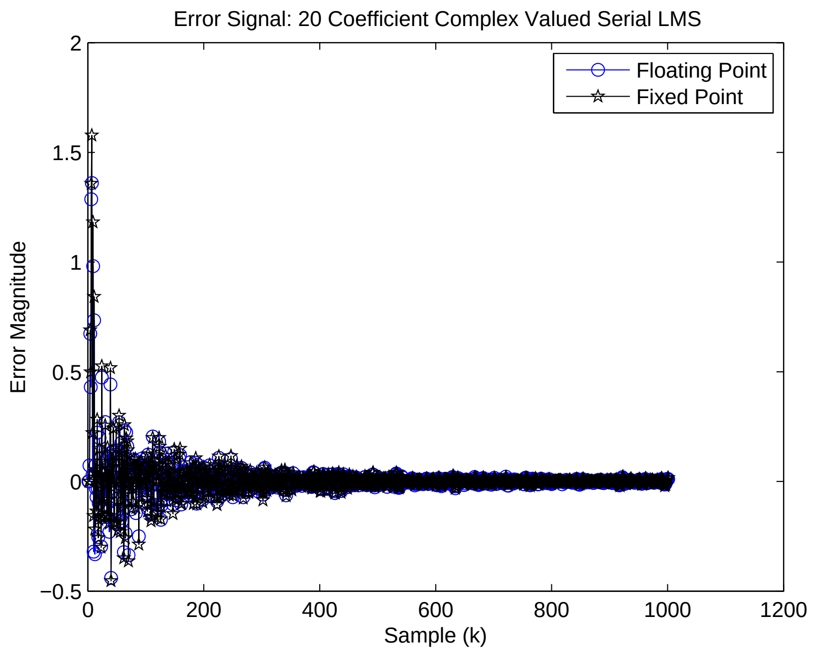
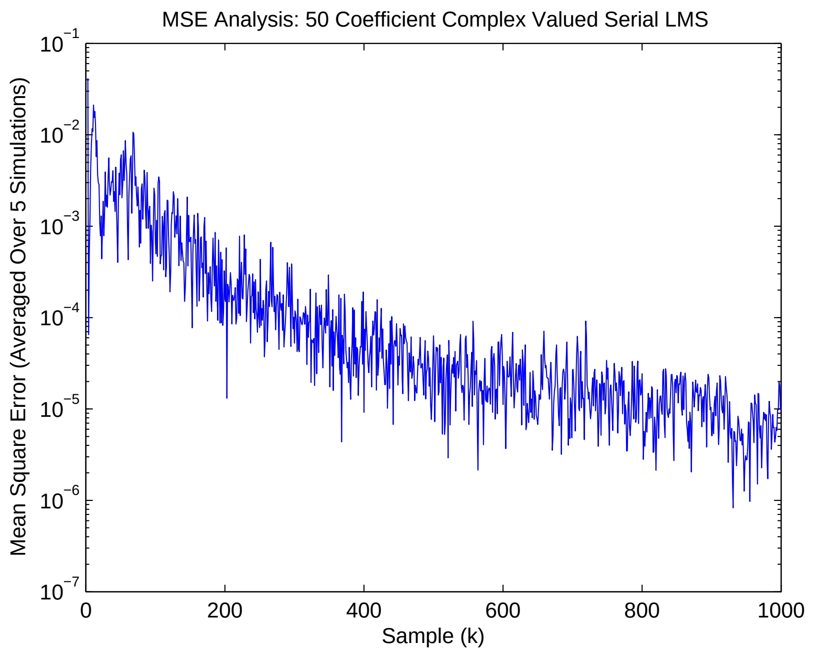


Figure 13.3: Test Case 5: 20 Coefficient Complex Valued Serial LMS



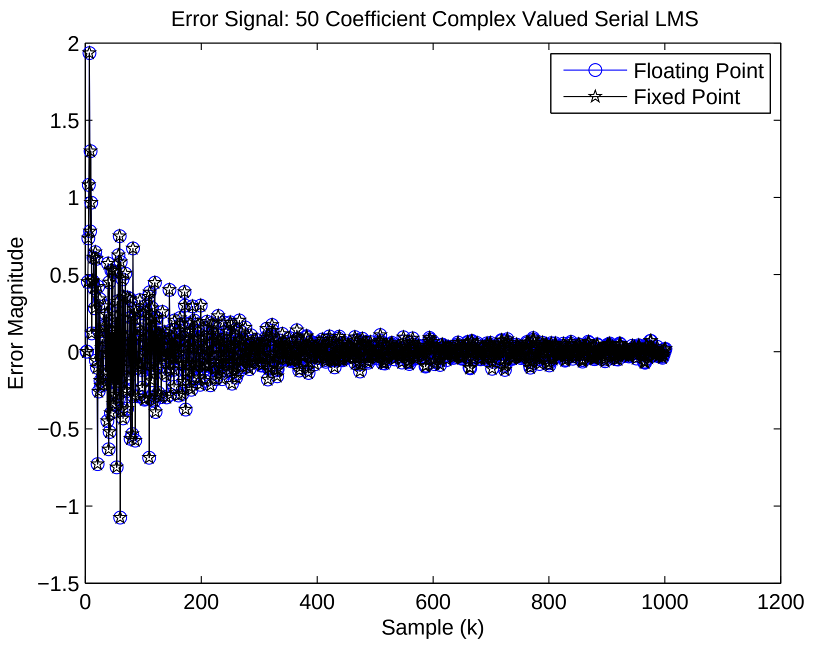
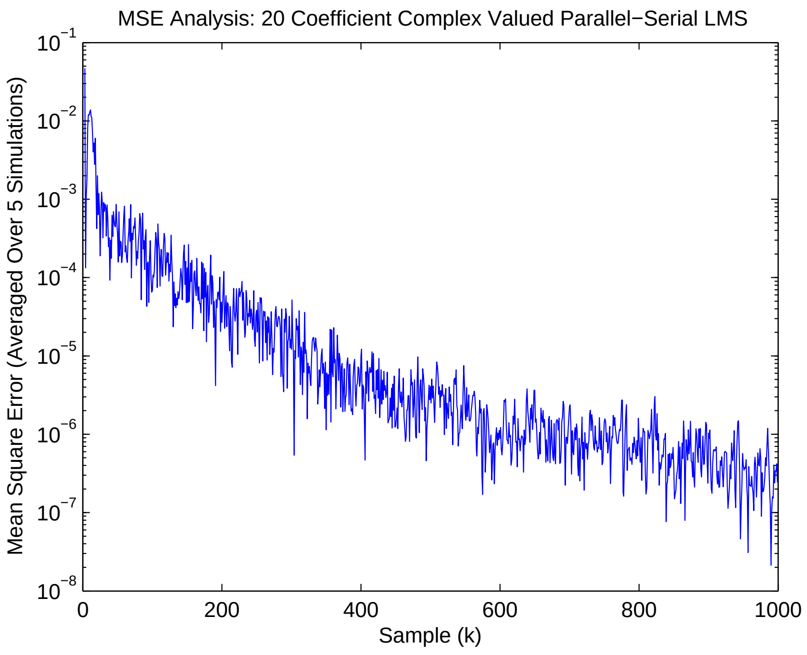


Figure 13.4: Test Case 6: 50 Coefficient Complex Valued Serial LMS

13.3 Complex Valued Arithmetic Parallel-Serial LMS Filter



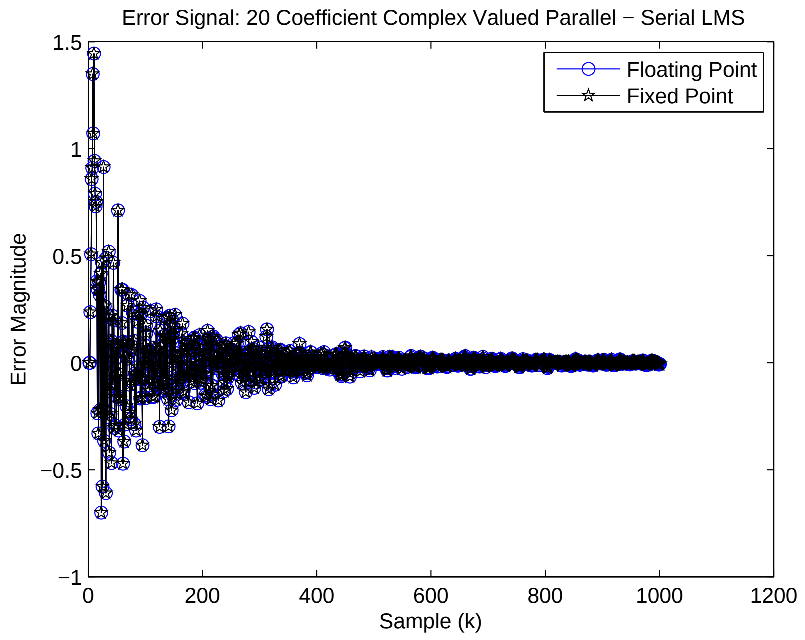
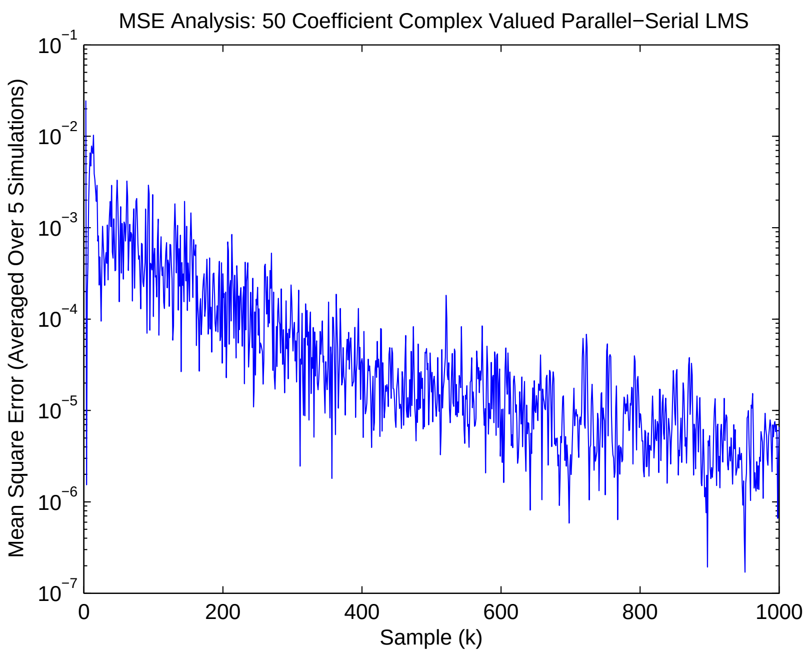


Figure 13.5: Test Case 7: 20 Coefficient Complex Valued Parallel - Serial LMS



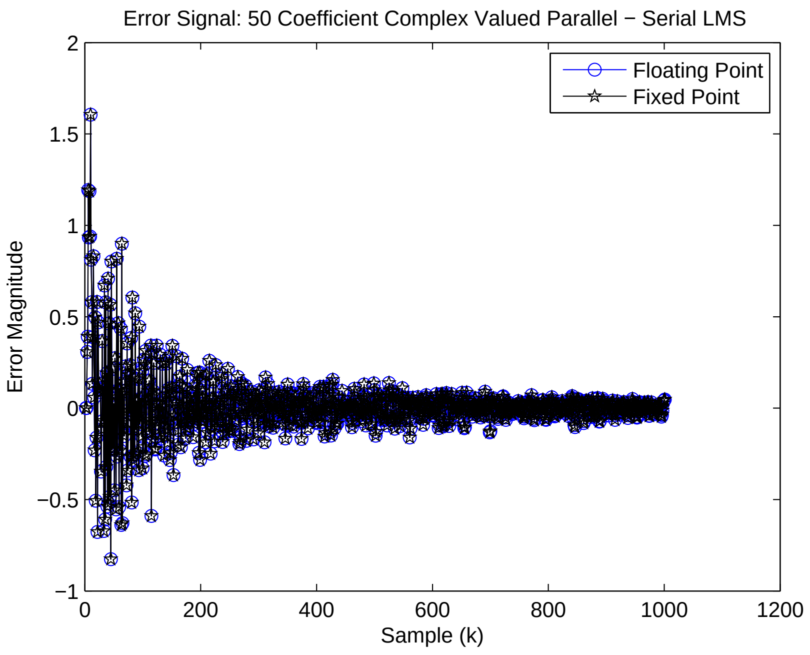
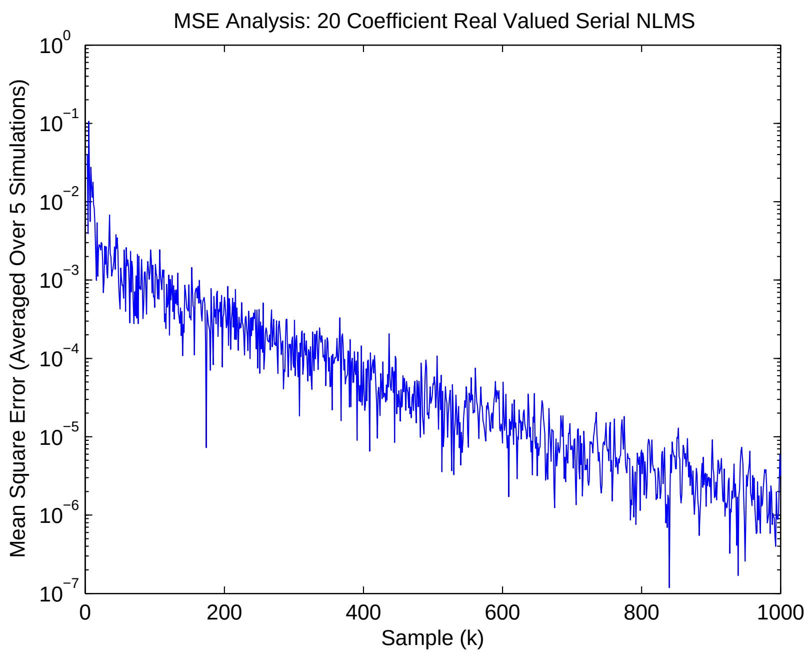


Figure 13.6: Test Case 8: 50 Coefficient Complex Valued Parallel - Serial LMS

13.4 Real Valued Arithmetic Serial NLMS Filter Results



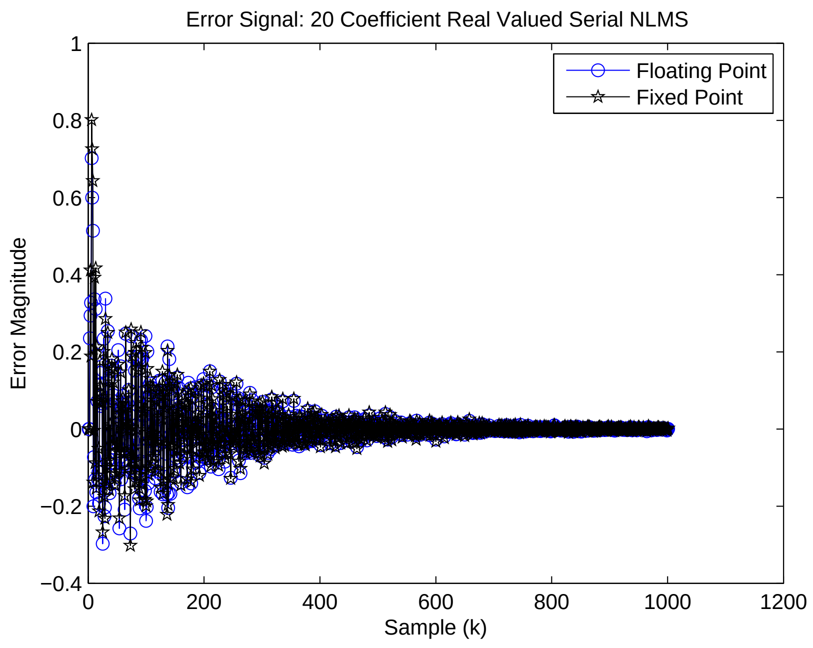
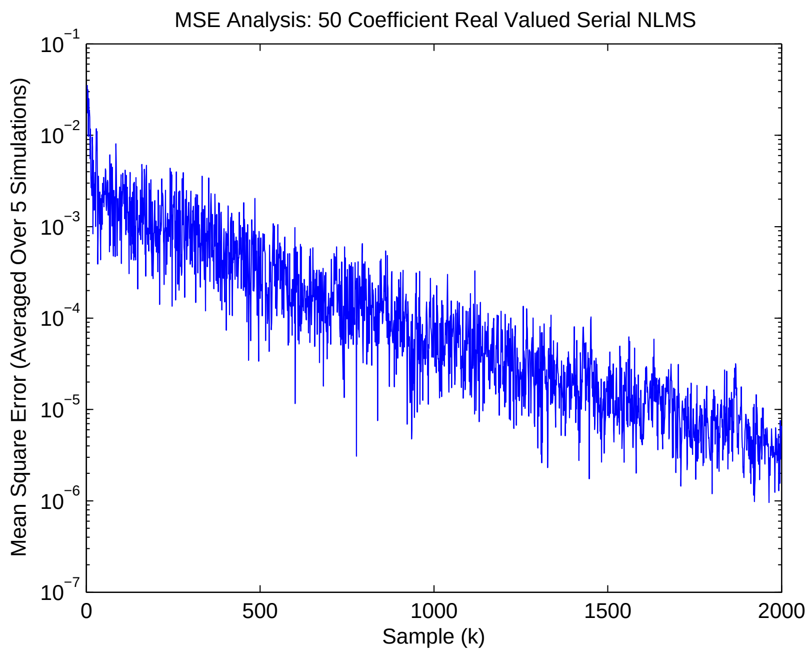


Figure 13.7: Test Case 9: 20 Coefficient Real Valued Serial NLMS



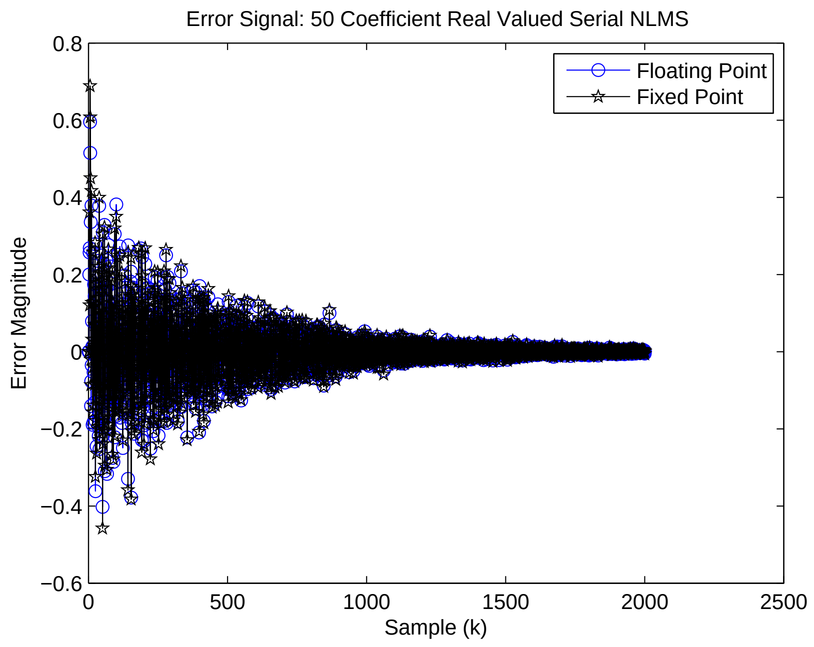
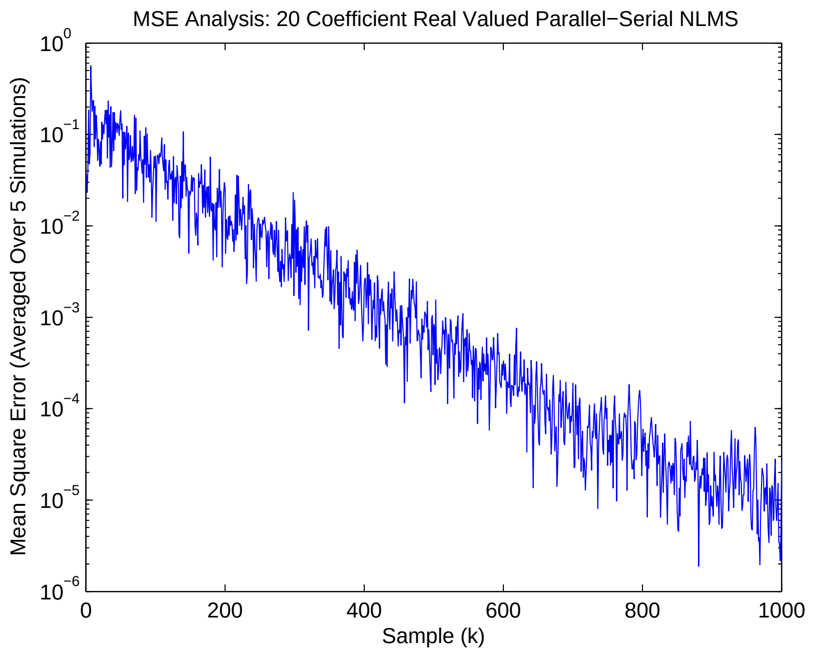


Figure 13.8: Test Case 10: 50 Coefficient Real Valued Serial NLMS

13.5 Real Valued Arithmetic Parallel-Serial NLMS Filter Results



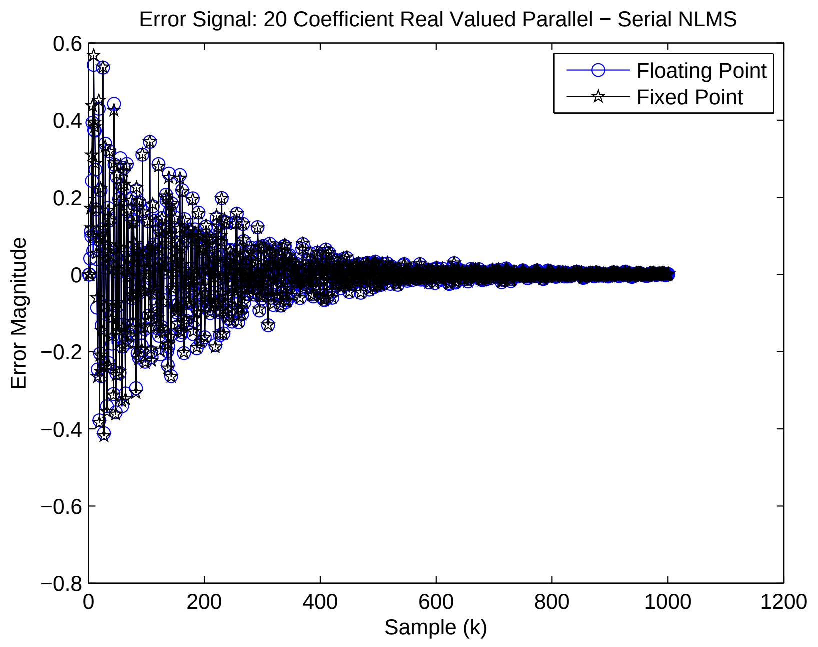
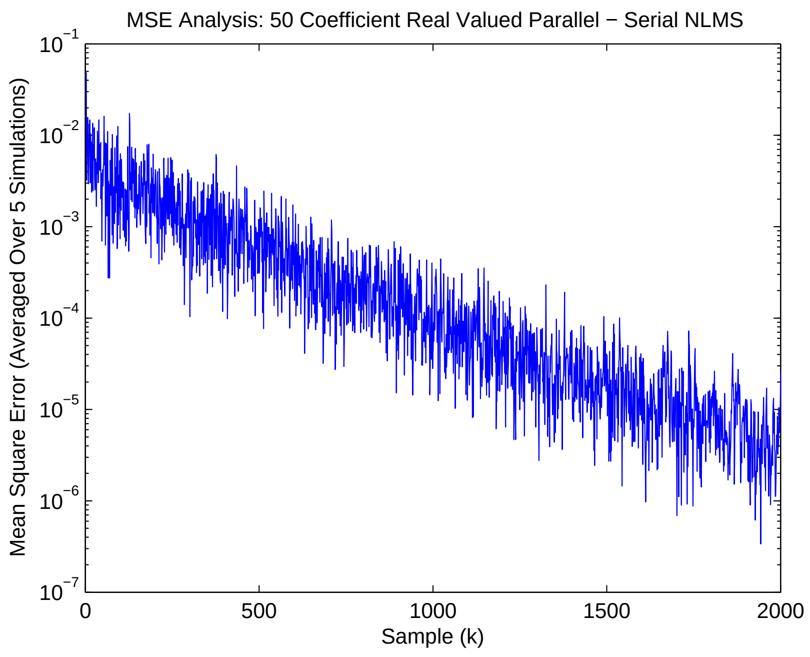


Figure 13.9: Test Case 11: 20 Coefficient Real Valued Parallel - Serial NLMS



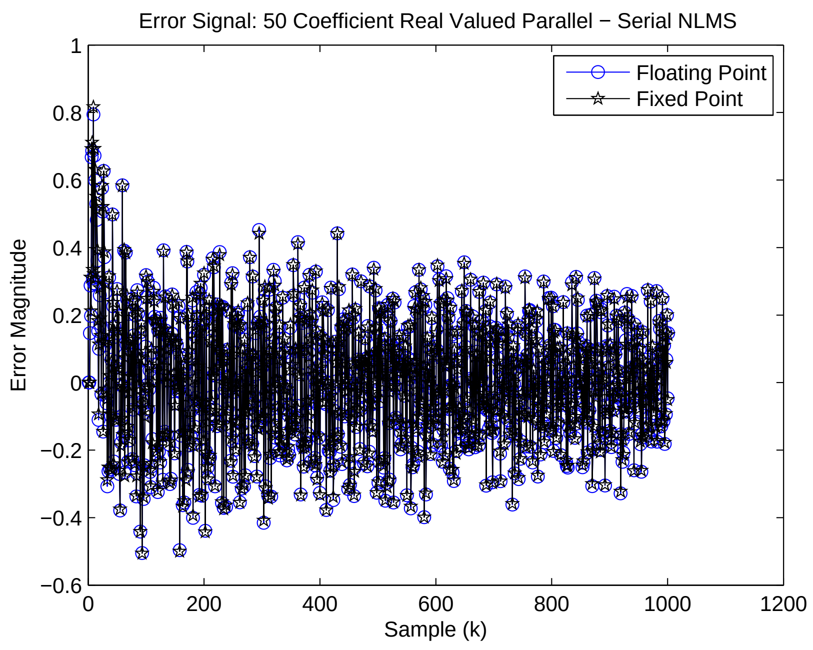
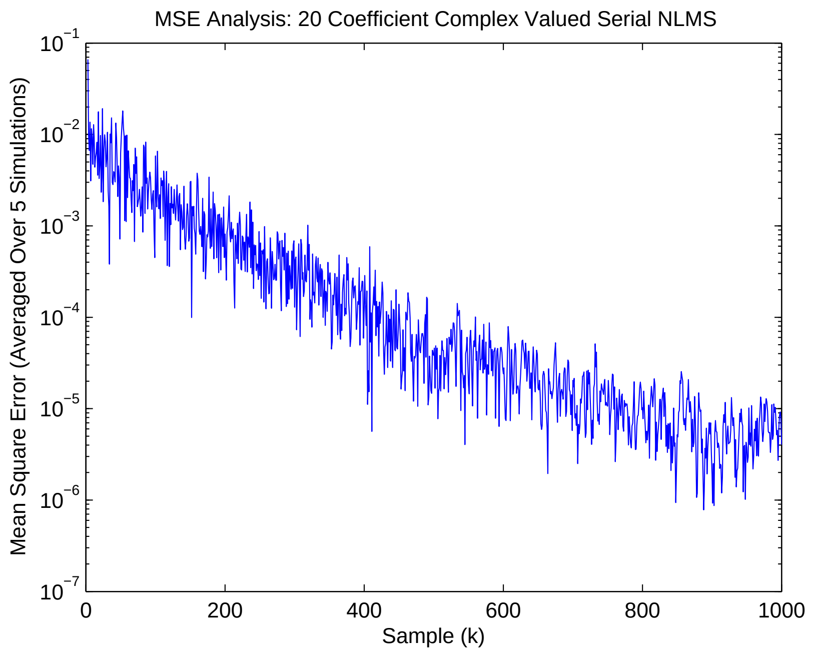


Figure 13.10: Test Case 12: 50 Coefficient Real Valued Parallel-Serial NLMS

13.6 Complex Valued Arithmetic Serial NLMS Filter Results



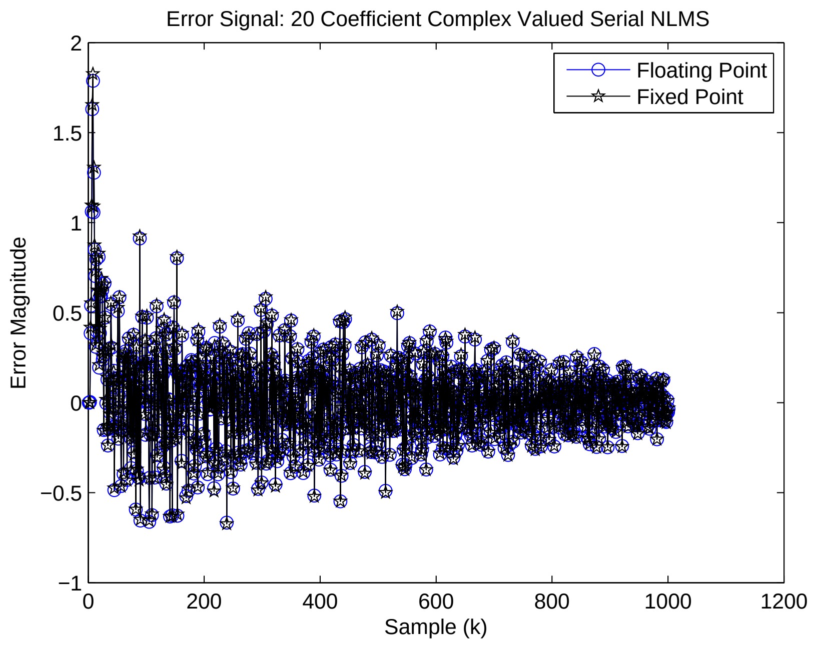
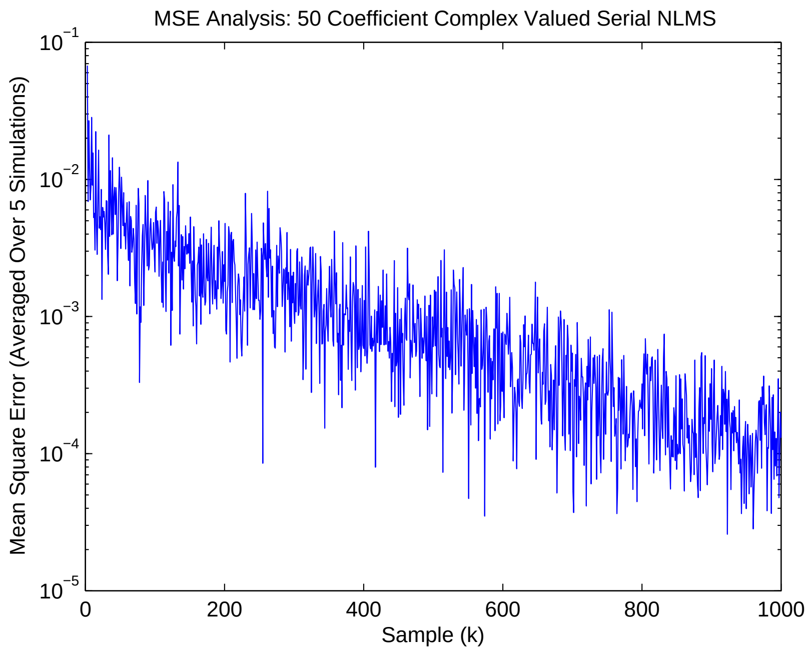


Figure 13.11: Test Case 13: 20 Coefficient Complex Valued Serial NLMS



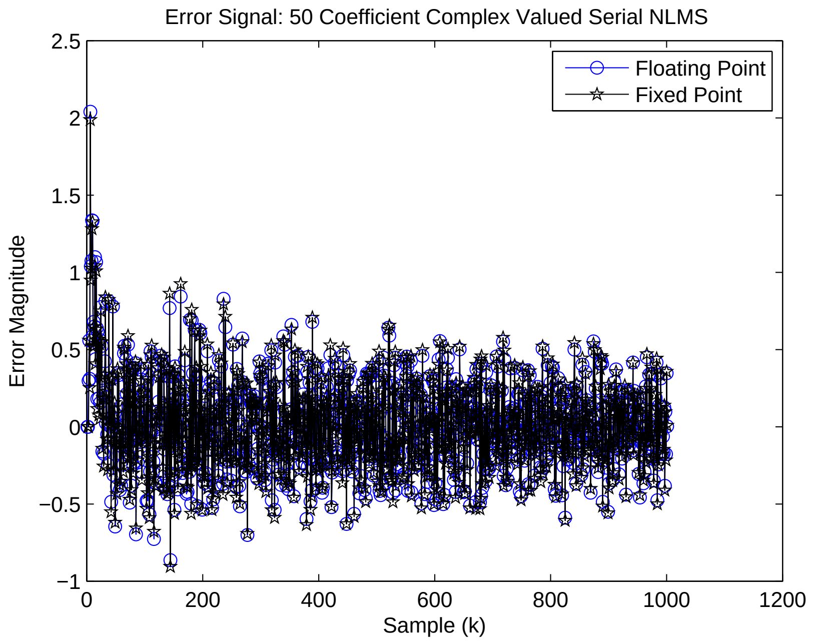
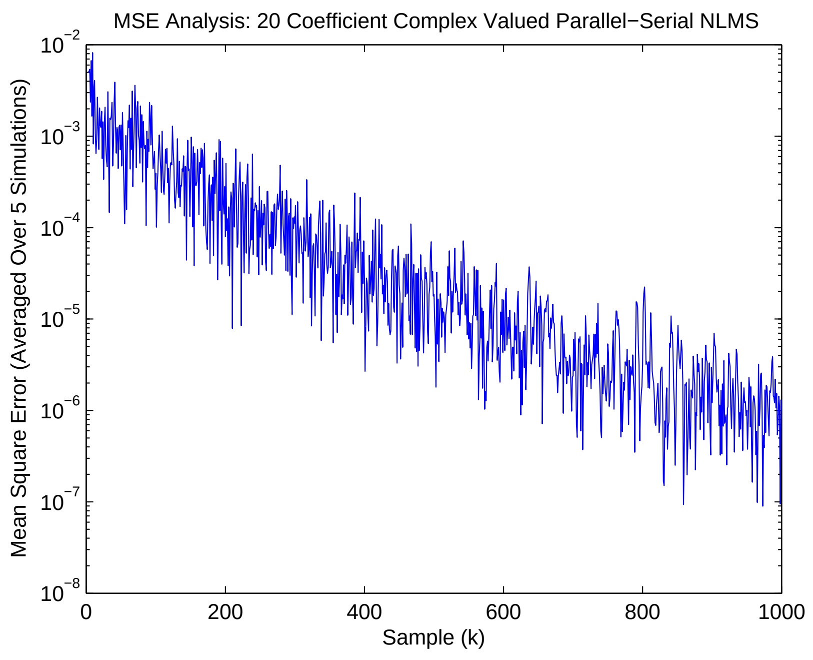


Figure 13.12: Test Case 14: 50 Coefficient Complex Valued Serial NLMS



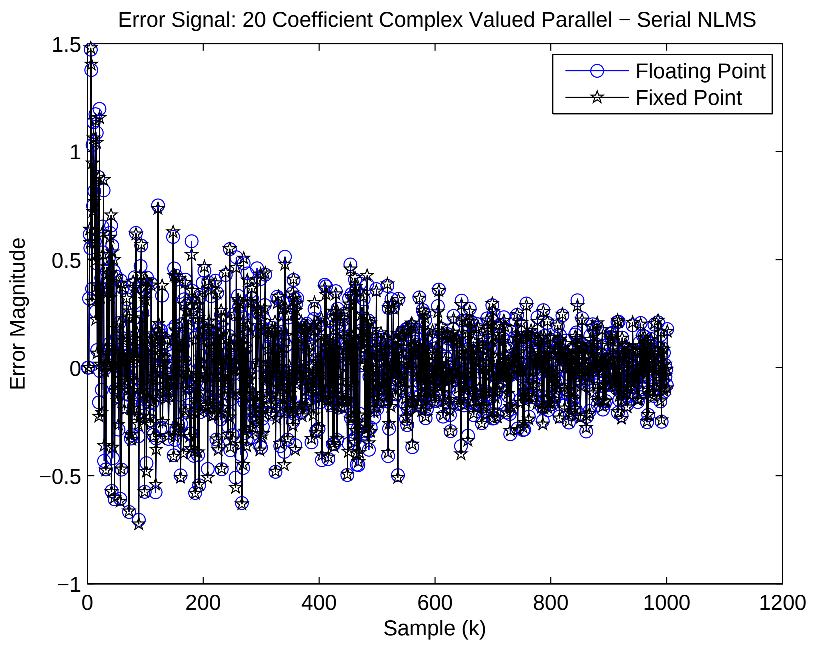
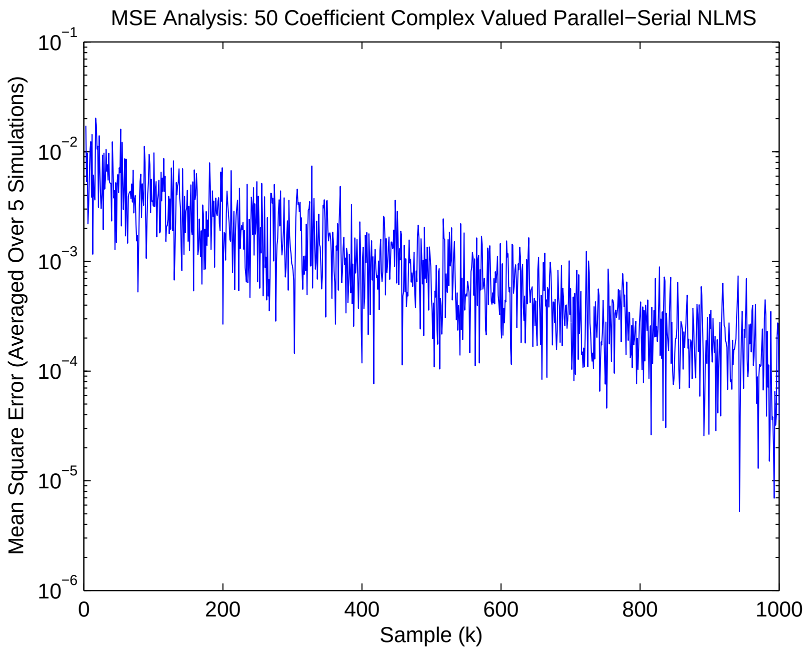


Figure 13.13: Test Case 15: 20 Coefficient Complex Valued Parallel - Serial NLMS

13.7 Complex Valued Arithmetic Parallel-Serial NLMS Filter Results



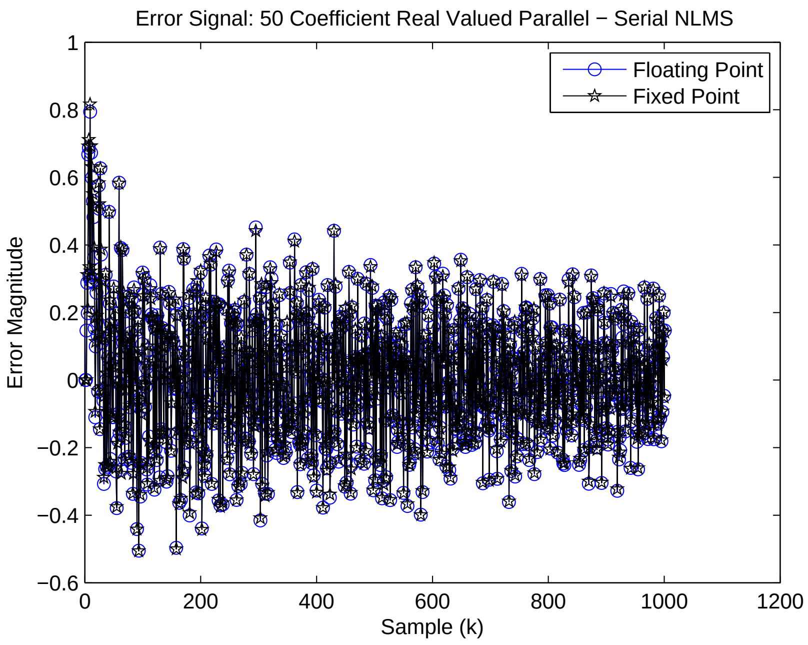
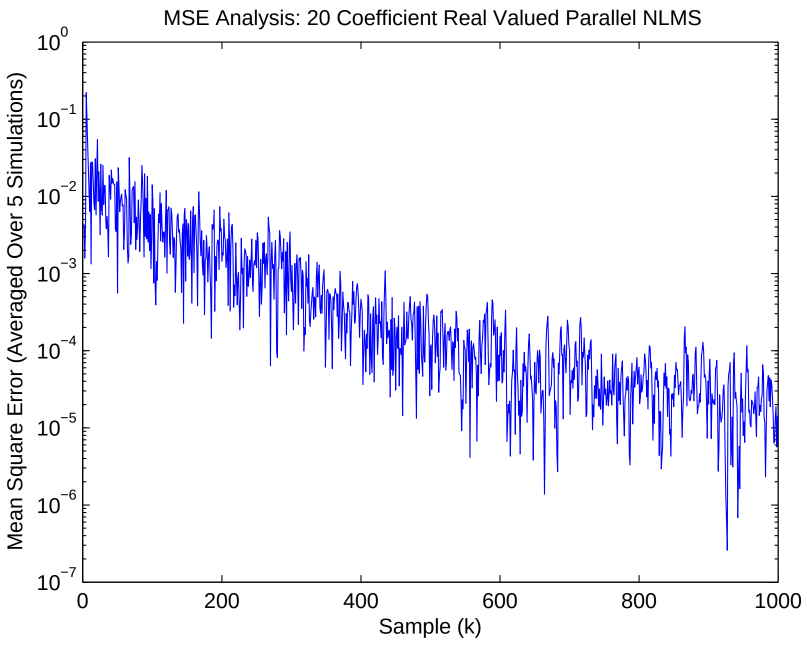


Figure 13.14: Test Case 16: 50 Coefficient Complex Valued Parallel - Serial NLMS

13.8 Real Valued Arithmetic Parallel NLMS Filter Results



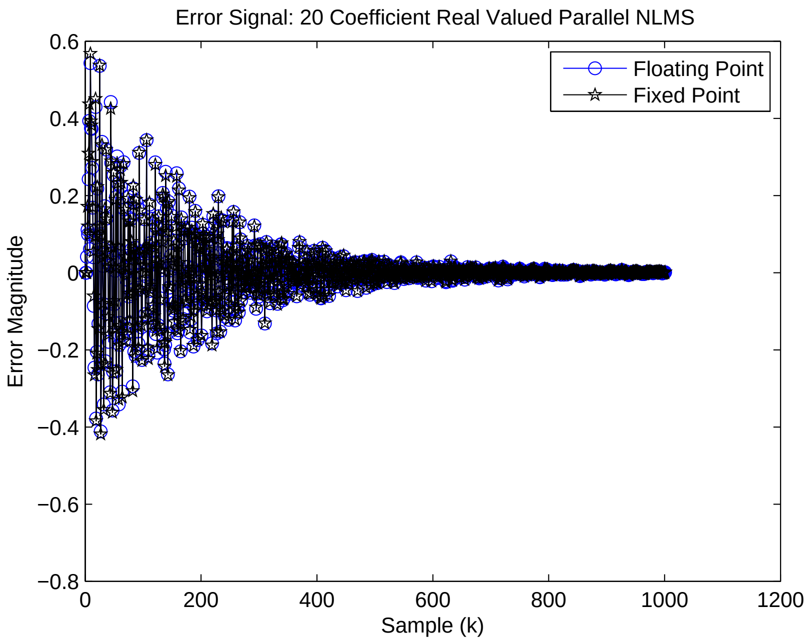
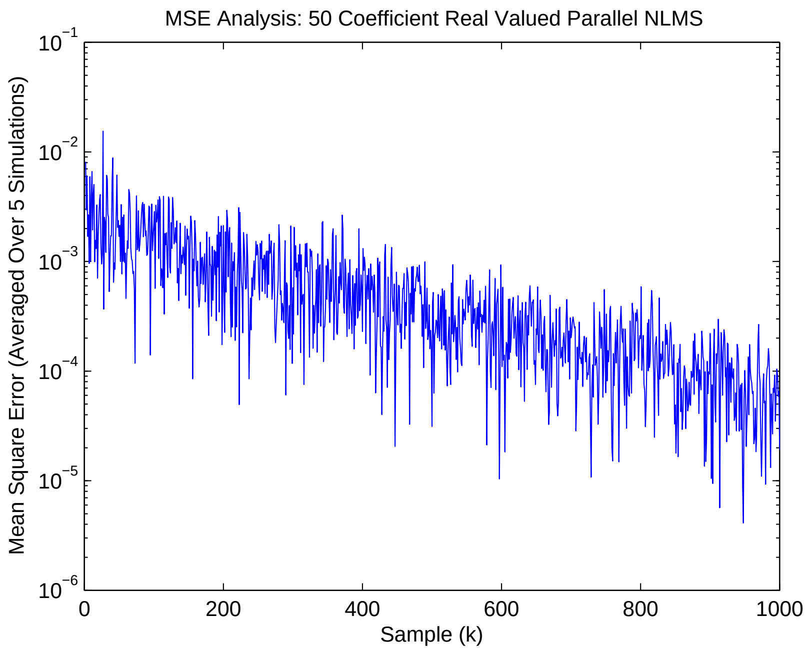


Figure 13.15: Test Case 17: 20 Coefficient Real Valued Parallel NLMS



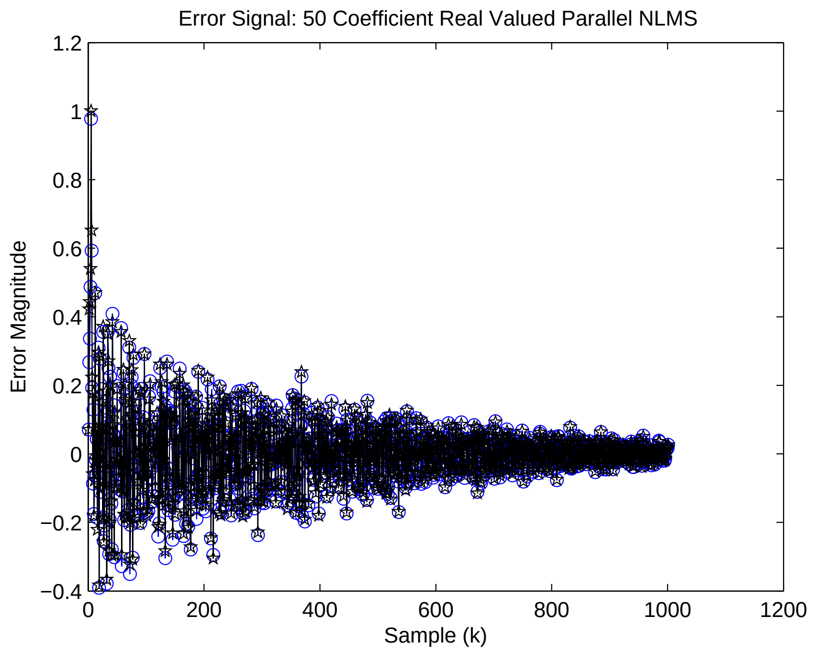
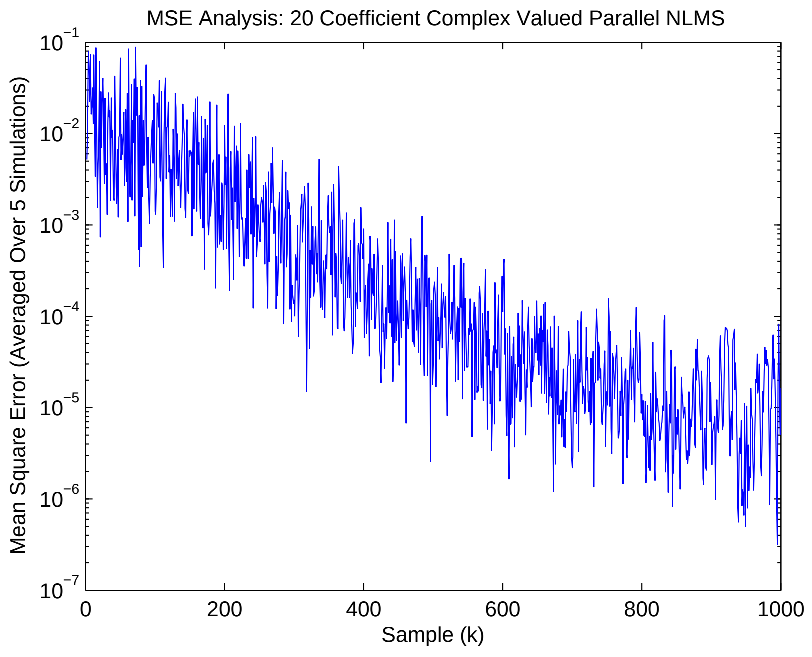


Figure 13.16: Test Case 18: 50 Coefficient Real Valued Parallel NLMS

13.9 Complex Valued Arithmetic Parallel NLMS Filter Results



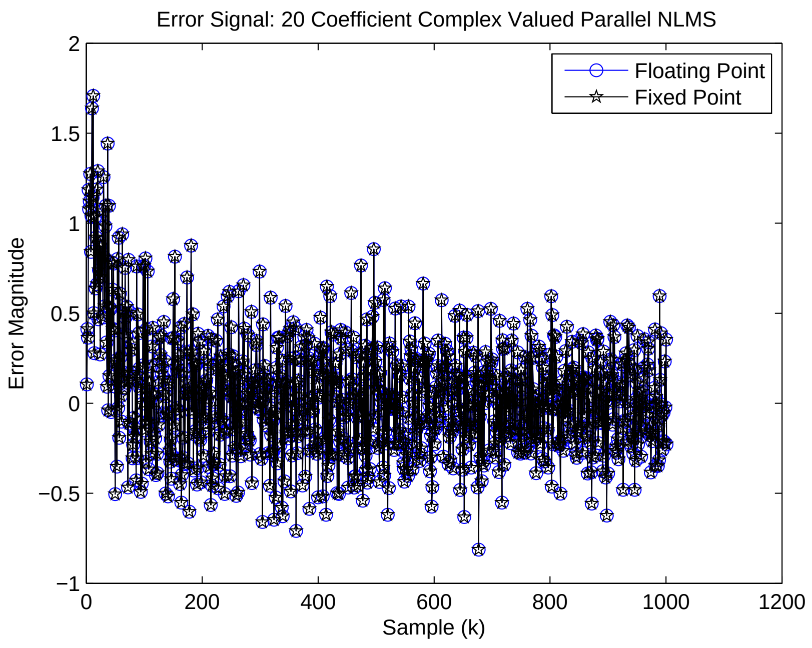
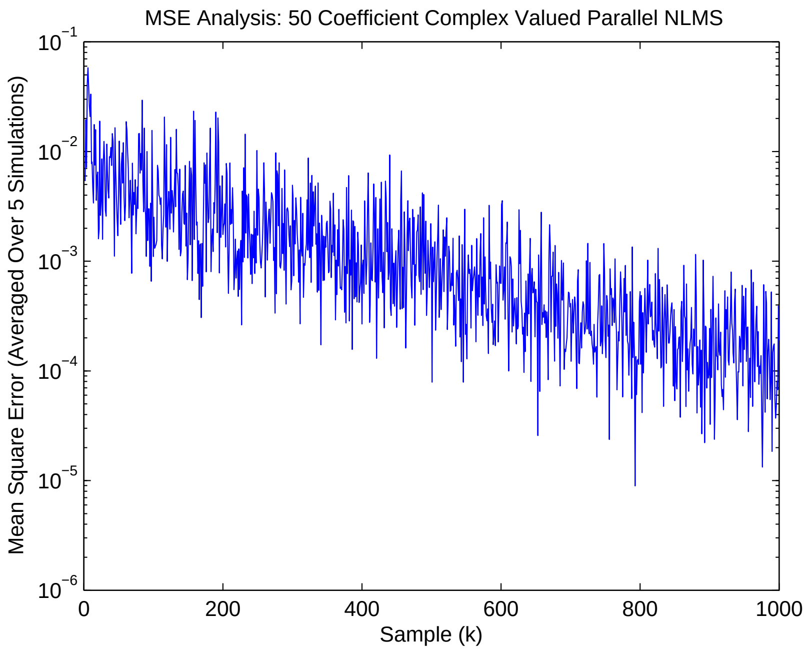


Figure 13.17: Test Case 19: 20 Coefficient Complex Valued Parallel NLMS



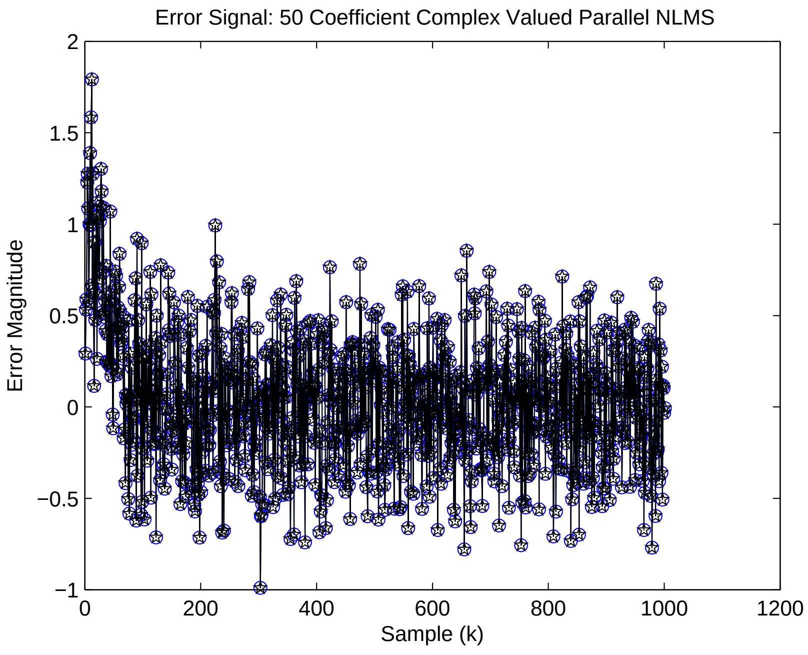


Figure 13.18: Test Case 20: 50 Coefficient Complex Valued Parallel NLMS