DCD Algorithm: Architectures, FPGA Implementations and Applications

This thesis is submitted in partial fulfilment of the requirements for Doctor of Philosophy (Ph.D.)

Jie Liu
Communications Research Group
Department of Electronics
University of York

Nov 2008

Abstract

In areas of signal processing and communications such as antenna array beamforming, adaptive filtering, multi-user and multiple-input multiple-output (MIMO) detection, channel estimation and equalization, echo and interference cancellation and others, solving linear systems of equations often provides an optimal performance. However, this is also a very complicated operation that designers try to avoid by proposing different sub-optimal solutions. The dichotomous coordinate descent (DCD) algorithm allows linear systems of equations to be solved with high computational efficiency. It is a multiplication-free and division-free technique and, therefore, it is well suited for hardware implementation.

In this thesis, we present architectures and field-programmable gate array (FPGA) implementations of two variants of the DCD algorithm, known as the cyclic and leading DCD algorithms, for real-valued and complex-valued systems. For each of these techniques, we present architectures and implementations with different degree of parallelism. The proposed architectures allow a trade-off between FPGA resources and the computation time. The fixed-point implementations provide an accuracy performance which is very close to the performance of floating-point counterparts.

We also show applications of the designs to complex division, antenna array beamforming and adaptive filtering. The DCD-based complex divider is based on the idea that the complex division can be viewed as a problem of finding the solution of a 2×2 real-valued system of linear equations, which is solved using the DCD algorithm. Therefore, the new divider uses no multiplication and division. Comparing with the classical complex divider, the DCD-based complex divider requires significantly smaller chip area.

A DCD-based minimum variance distortionless response (MVDR) beamformer employs the DCD algorithm for multiplication-free finding the antenna array weights. An FPGA implementation of the proposed DCD-MVDR beamformer requires a chip area much smaller and throughput much higher than that achieved with other implementations. The performance of the fixed-point implementation is very close to that of floating-point implementation of the MVDR beamformer using direct matrix inversion.

When incorporating the DCD algorithm in recursive least squares (RLS) adaptive filter, a new efficient technique, named as the RLS-DCD algorithm, is derived. The RLS-DCD algorithm expresses the RLS adaptive filtering problem in terms of auxiliary normal equations with respect to increments of the filter weights. The normal equations are approximately solved by using the DCD iterations. The RLS-DCD algorithm is well-suited to hardware implementation and its complexity is as low as $\mathcal{O}(N^2)$ operations per sample in a general case and $\mathcal{O}(N)$ operations per sample for transversal RLS adaptive filters. The performance of the RLS-DCD algorithm, including both fixed-point and floating-point implementations, can be made arbitrarily close to that of the floating-point classical RLS algorithm. Furthermore, a new dynamically regularized RLS-DCD algorithm is also proposed to reduce the complexity of the regularized RLS problem from $\mathcal{O}(N^3)$ to $\mathcal{O}(N^2)$ in a general case and to $\mathcal{O}(N)$ for transversal adaptive filters. This dynamically regularized RLS-DCD algorithm is simple for finite precision implementation and requires small chip resources.

Contents

Acknowledgements

I would firstly like to thank my supervisor, Dr. Yuriy V. Zakharov, for his advice, support and encouragement during the course of my Ph.D. study.

I would also like to thank all my dear colleagues in the Communications Research Group.

This thesis is dedicated to my parents and my wife.

Declaration

Some of the research presented in this thesis has resulted in some publications. These publications are listed at the end of Chapter 1.

All work presented in this thesis as original is so, to the best knowledge of the author. References and acknowledgements to other researchers have been given as appropriate.

Glossary

ADC	Analog-to-Digital Converter
AP	Affine Projection
BAMI	Blockwise Analytic Matrix Inversion
BER	Bit Error Ratio
BPSK	Binary Phase Shift Keying
CD	Coordinate Descent
CDMA	Code Division Multiple Access
CG	Conjugate Gradient
CG-CLF	Conjugate Gradient Control Liapunov Function
CORDIC	COordinate Rotation DIgital Computer
DC	Down-Converted
DCD	Dichotomous Coordinate Descent
DCM	Digital Clock Manager
DoA	Direction of Arrival
EDS	Euclidean Direction Search
ERLS	Exponentially Weighted RLS
FAP	Fast Affine Projection
FIR	Finite-duration Impulse Response
FPGA	Field-Programmable Gate Array
HDL	Hardware Description Language
HSLA	high-speed logarithmic arithmetic
IIR	Infinite-duration Impulse Response
LMS	Least Mean-Square
LNS	Logarithmic Number Systems
LS	Least Squares
LSB	Least Significant Bit
MIMO	Multiple-Input Multiple-Output
MMP	Minus-Minus-Plus
MMSE	Minimum Mean Squared Error
MSB	Most Significant Bit
MSE	Mean Squared Error
MVDR	Minimum Variance Distortionless Response
MULT	Multiplier
MUX	Multiplexer
NLMS	Normalized LMS
OFDM	Orthogonal Frequency-Division Multiplexing
QRD	QR Decomposition

QR-MVDR	MVDR beamforming based on QRD
RB	Redundant-to-Binary
RLS	Recursive Least Squares
SGR	Squared Givens Rotations
SINR	Signal-to-Interference-plus-Noise Ratio
SNR	Signal-to-Noise Ratio
SOR	Successive Over-Relaxation
ULA	Uniform Linear Array
WiMAX	Worldwide Interoperability for Microwave Access

Chapter 1

Introduction

Contents

1.1 Overview

In areas of signal processing and communications such as channel estimation and equalization, multi-user and multiple-input multiple-output (MIMO) detection, antenna array beamforming, adaptive filtering, echo and interference cancellation and others, solving the normal system of equations often provides an optimal performance. However, this is also a complicated operation that designers try to avoid by proposing different sub-optimal algorithms.

Solving the normal equations is usually considered by using matrix inversion. Computing the matrix inversion directly has a complexity of $\mathcal{O}(N^3)$ [1], which is too complicated for real-time implementation. From a numerical point of view, the best approach is to avoid the matrix inversion [2–4]. Consequently for real-time solutions, techniques that solve systems of equations may be preferable. Among them are direct and iterative methods. The direct methods, such as Cholesky decomposition, Gaussian elimination,

QR decomposition (QRD) and others, have complexity of $\mathcal{O}(N^3)$ [1]. Iterative methods, such as the steepest descent method and conjugate gradient (CG) method provide fast convergence when the condition number of the matrix is not very large but require $\mathcal{O}(N^2)$ operations per iteration. The coordinate descent techniques, such as Gauss-Seidel, Jacobi and Successive Over-Relaxation (SOR) methods [1] demonstrate a slower convergence but require only $\mathcal{O}(N)$ operations per iteration. The computational load of these iterative techniques depends on the number of iterations executed (and hence accuracy obtained). These iterative methods require multiplication and division operations, which are complex for implementation, especially in hardware, i.e., they require a significant chip area and high power consumption. Moreover, divisions can lead to numerical instability.

The Dichotomous Coordinate Descent (DCD) algorithm [5] is based on coordinate descent techniques with power of two variable step-size. It is simple for implementation, as it does not need multiplication or division operations. For each iteration, it only requires $\mathcal{O}(N)$ additions or $\mathcal{O}(1)$ additions. Thus, the DCD algorithm is quite suitable for hardware realization.

In this thesis, we investigate the hardware architectures and designs of the DCD algorithm and its variants, and apply them to several practical applications in the communication field. Specifically, we present and compare two variants of the DCD algorithm: cyclic and leading DCD algorithms. A DCD algorithm for complex-valued systems of equations is also presented. We then present Field-Programmable Gate Array (FPGA) designs of these DCD algorithms with different degree of parallelism, including designs with serial and parallel update of the residual vector, as well as trade-off designs with group-updates. These designs show the relationship between the chip area usage and the processing speed. We can choose appropriate designs according to the requirements of practical applications. We also show examples of applications such as the complex division, antenna array beamforming and adaptive filtering. The designs and application results show that the DCD algorithm allows solving complicated signal processing problems requiring matrix inversion and the solution is simple for implementation in hardware. Furthermore, when incorporating the DCD algorithm in classical signal processing techniques, such as Recursive Least Squares (RLS) adaptive filter, we obtain new efficient techniques.

1.2 Contribution

The contributions of this thesis is summarized as following:

- Architectures and FPGA designs of two variants of the DCD algorithm, cyclic and leading DCD algorithms, are presented. For each of these techniques, serial designs, group-2 and group-4 designs, as well as a design with parallel update of the residual vector for the cyclic DCD algorithm are presented. These designs have different degrees of parallelism, thus enabling a trade-off between FPGA resources and the computation time. We also discuss applications of these designs.

- A low complexity complex-valued divider is developed. It is based on the idea that the complex division problem can be viewed as a 2×2 real-valued system of equations, which is solved using the DCD iterations. This complex divider does not use any multiplication or division operations. The area usage is only 527 slices. When operating from a 100 MHz clock, the throughput is at least 1.6 MHz. The maximum quotient error is less than one least significant bit (LSB).

- An efficient FPGA implementation of the Minimum Variance Distortionless Response (MVDR) beamformer is presented. The FPGA design is based on DCD iterations, thus making the whole design very efficient in terms of both the number of FPGA slices and throughput. Antenna beampatterns obtained from weights calculated in a fixed-point FPGA platform show a good match with those of a floating-point implementation by using direct matrix inversion for linear arrays of size 9 to 64 elements.

- An FPGA design of a DCD-based RLS adaptive filtering algorithm has been proposed with two data structures: arbitrary and time-shifted. The RLS-DCD algorithm is obtained by incorporating DCD iterations into the RLS algorithm to solve the system of equations. The algorithm is simple for finite precision implementation and requires small chip resources. A 9-element antenna beamformer based on the arbitrary data structure design can achieve a weight update rate that is significantly higher than that of an FPGA design based on QRD with approximately the same area usage. The design of transversal RLS-DCD algorithm, which exploits the time-shifted data structure can provide the weight update rate as high as 207 kHz and 76 kHz for 16-tap and 64-tap adaptive filters, respectively, while using as little as 1153 and 1306 slices, respectively. Numerical results show that the performance of the fixed-point FPGA implementation of the RLS-DCD algorithm is close to that of the floating-point implementation of the classical RLS algorithm.

- A low complexity dynamically regularized RLS algorithm has been proposed based on the RLS-DCD algorithm. Its FPGA design for complex-valued systems is also presented. The area usage and throughput are approximately the same as in the unregularized RLS-DCD algorithm. This dynamically regularized RLS-DCD algorithm is applied to a communication system with the MVDR beamformer. Numeri-

cal results show that the proposed algorithm provides Bit-Error-Ratio (BER) results close to that of the floating-point regularized classical RLS algorithm.

1.3 Thesis Outline

The structure of the thesis is as follows:

- In Chapter 2, literature review is presented, that describes techniques for solving systems of equations, matrix inversion, related hardware reference designs, adaptive filtering, complex division, MVDR beamforming, as well as FPGA design procedures.
- In Chapter 3, the DCD algorithm and its variants are introduced. Several architectures and FPGA designs of the DCD algorithm are presented. Numerical properties of the DCD algorithm are also analyzed.
- In Chapter 4, a multiplication-free complex divider is implemented in FPGA based on the DCD iterations.
- In Chapter 5, we present an efficient FPGA implementation of the DCD-based MVDR beamformer.
- In Chapter 6, the RLS-DCD algorithm is introduced and implemented into FPGA with arbitrary and time-shifted data structures. The dynamically regularized RLS-DCD algorithm is proposed and implemented into FPGA for solving complex-valued systems with arbitrary data structure. Numerical results, area usage and throughput rate of both algorithms and their FPGA designs are also given.
- In Chapter 7, conclusions and future work are presented.

1.4 Notations

In this thesis, we use capital and small bold fonts to denote matrices and vectors, e.g., $\mathbf{R}$ and $\mathbf{r}$, respectively. Elements of the matrix and vector are denoted as $R_{p,n}$ and r_n. A n-th column of $\mathbf{R}$ is denoted as $\mathbf{R}_{:,n}$. The variable i is used as a time index, i.e., $\mathbf{R}(i)$ is the matrix $\mathbf{R}$ at time instant i, or as a digit bit index of a variable, i.e., x_i is the i-th bit of

the variable X. The variable k is used as an iteration index, i.e., $\mathbf{x}^{(k)}$ is the vector $\mathbf{x}$ at k-th iteration. The symbol j is an imaginary unit $j = \sqrt{-1}$. We denote $\Re(\cdot)$ and $\Im(\cdot)$ the real and imaginary components of a complex number, respectively; $(\cdot)^T$ denotes matrix transpose and $(\cdot)^H$ denotes Hermitian transpose. The symbols $\mathrm{E}[\cdot]$ denotes the statistical expectation operator.

1.5 Publication List

Some of the research presented in this thesis has been published, submitted, or will be submitted to some publications at the time of submission of this thesis.

Journal Papers

1. J. Liu, B. Weaver and Y. Zakharov, "FPGA implementation of multiplication-free complex division", *Electronics Letters*, vol. 44, no. 2, pp. 95-96, 2008.
2. J. Liu and Y. Zakharov, "A low complexity dynamically regularized RLS algorithm", *Electronics Letters*, vol. 44, no. 14, pp. 885-886, 2008.
3. Y. Zakharov, G. White and J. Liu, "Low complexity RLS algorithms using dichotomous coordinate descent iterations", *IEEE Transactions on Signal Processing*, vol. 56, no. 7, pp. 3150-3161, July 2008.
4. J. Liu, Y. Zakharov and B. Weaver, "Architecture and FPGA implementation of dichotomous coordinate descent algorithm", under revision in *IEEE Transactions on Circuits and Systems Part I: Regular Papers*.

Patent

1. Y. Zakharov, G. White, T. Tozer and J. Liu, "Method for solving a sequence of equations and a new family of fast and stable RLS algorithms using coordinate descent iterations", Filed, Nov. 2007, UK patent NO: 0720292.2.

Conference Papers

1. J. Liu and Y. Zakharov, "FPGA implementation of the dynamically regularized RLS adaptive filter using dichotomous coordinate descent iterations", *Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, CA, US*, 26-29 Oct. 2008.

2. J. Liu, Z. Quan and Y. Zakharov, "Parallel FPGA implementation of DCD algorithm", *15th International Conference on Digital Signal Processing, Cardiff, Wales, UK*, pp. 331-334, 1-4 July 2007.

3. J. Liu, B. Weaver, Y. Zakharov and G. White, "An FPGA-based MVDR Beamformer Using Dichotomous Coordinate Descent Iterations", *IEEE International Conference on Communications, Glasgow, Scotland, UK,* pp. 2551-2556, 24-28 June 2007.

4. J. Liu, B. Weaver and G. White, "FPGA implementation of the DCD algorithm," *London Communication Symposium, London, UK*, pp. 125-128, Sep. 2006.

5. J. Liu and Y. Zakharov, "FPGA implementation of the RLS adaptive filter using dichotomous coordinate descent iterations," submitted to *IEEE International Conference on Communications*, 2009.

6. Z. Quan, J. Liu, and Y. Zakharov, "FPGA implementation of DCD based CDMA multiuser detector", *15th International Conference on Digital Signal Processing, Cardiff, Wales, UK*, pp. 319-322, 1-4 July 2007.

7. Y. Zakharov, G. White, and J. Liu, "Fast RLS algorithm using dichotomous coordinate descent iterations", *Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, CA, US*, 4-7 Nov. 2007.

8. Z. Quan, J. Liu, and Y. Zakharov, "FPGA design of box-constrained MIMO detector", submitted to *IEEE International Conference on Communications*, 2009.

Chapter 2

Literature Review

Contents

2.1 Introduction

This chapter presents the background of problems to be solved in this thesis. We first discuss efficient algorithms for solving the normal equations and calculating matrix inversions, including direct methods and iterative methods. Some hardware reference designs are discussed and compared. We also introduce some communication application examples in brief, such as adaptive filtering, complex division, adaptive beamforming. FPGA design procedures are also presented, including the FPGA implementation environment used in this thesis.

The rest of this chapter is organized as follows. In Section 2.2, efficient algorithms for solving systems of equations and performing matrix inversion are analyzed. The related

hardware reference designs are presented in Section 2.4. The literature about adaptive filtering, complex division and adaptive beamforming are presented in Sections 2.3, 2.5 and 2.6, respectively. FPGA design procedures are introduced in Section 2.7. Finally, conclusions are given in Section 2.8.

2.2 Solving Normal Systems of Equations

A wide variety of signal processing and communications applications require the linear least-squares (LS) problem [6] to be solved in real time. Among these are adaptive antenna arrays [6,7], multiuser detection [8], echo cancelation [9], equalization [10], system identification [6], amplifier linearization [11] and many others. The LS problem is known to be equivalent to the solution of a system of linear equations, also referred to as normal equations [3, 7]

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{2.1}$$

where $\mathbf{A}$ is an $N \times N$ symmetric positive definite matrix and both $\mathbf{x}$ and $\mathbf{b}$ are $N \times 1$ vectors. The matrix $\mathbf{A}$ and vector $\mathbf{b}$ are known, whereas the vector $\mathbf{x}$ should be estimated. An exact solution is generally defined as

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}, \tag{2.2}$$

where $\mathbf{A}^{-1}$ is the inverse of the matrix $\mathbf{A}$. It is well known that the computational load of the matrix inversion relates to the size of the matrix N, and is generally regarded as an operation of a complexity $\mathcal{O}(N^3)$ [1]. A standard mathematical calculation software package, such as Matlab, uses one of a variety of techniques from the LAPACK library [12] to solve this problem. From a numerical point of view, the best approach to matrix inversion is avoid to do it explicitly but instead, where possible, to solve an applicable system of equations [2–4]. Consequently for real-time solutions, techniques that solve systems of equations are the most suitable approach to the LS problem. There exist many efficient methods, which can be categorized under two main headings: direct methods and iterative methods. Direct methods compute an exact solution of the system of equations through a finite number of pre-specified operations [2]. In contrast, iterative methods produce a sequence of successively better approximations of the optimal solution [2].

These methods can be also used to calculate the inverse of matrix $\mathbf{A}$. Let $\mathbf{A}\mathbf{X} = \mathbf{I}$, where $\mathbf{I}$ is an $N \times N$ identity matrix and $\mathbf{X} = \mathbf{A}^{-1}$ is an $N \times N$ matrix needed to calculate, we obtain N systems of equations

$$\mathbf{A}\mathbf{X}_{:,n} = \mathbf{I}_{:,n} \qquad n = 1, ..., N. \tag{2.3}$$

By solving these N systems of equations, we could obtain $\mathbf{A}^{-1}$.

In the following two subsections, some direct and iterative algorithms for solving linear systems will be presented.

2.2.1 Direct Methods

The direct methods, such as Gaussian elimination, LU factorization, Cholesky decomposition, QRD and others, obtain an exact solution of the system of equations (2.1) after a finite sequence of pre-specified operations [2]. The key idea of most direct methods is to reduce the general system of equations to an upper triangular form or a lower triangular form, which have the same solution as the original equations. They can be solved easily by back or forward substitutions, respectively, and often provide a high accuracy solution [1].

The basic idea of Gaussian elimination is to modify the original equations (2.1) to obtain an equivalent triangular system by taking appropriate linear combinations of the original equations (2.1) [1]. Specifically, it systematically applies row operations to transform the system of equations (2.1) to an upper triangular system $\mathbf{Ux} = \mathbf{y}$, where $\mathbf{U}$ is an $N \times N$ upper triangular matrix and $\mathbf{y}$ is an $N \times 1$ vector. Then the upper triangular system $\mathbf{Ux} = \mathbf{y}$ can be solved easily through back substitution operations. The complexity of the Gaussian elimination method is as high as $2N^3/3$ operations [1], including multiplications, divisions and additions. Besides high complexity, the main disadvantage of the Gaussian elimination method is that the right-hand vector $\mathbf{b}$ of (2.1) is involved in the elimination process and has to be known in advance for the elimination step to proceed [1]. Therefore, when solving such linear systems of equations with same left-hand matrix, the Gaussian elimination has to perform $2N^3/3$ operations for each one.

The LU factorization (or LU decomposition) can be viewed as a "high-level" algebraic description of the Gaussian elimination [1]. It decomposes the matrix $\mathbf{A}$ of the system of equations (2.1) into a product, $\mathbf{A} = \mathbf{LU}$, where $\mathbf{L}$ is an unit lower triangular matrix with all main diagonal elements equal to one and $\mathbf{U}$ is an upper triangular matrix [1]. Therefore, the solution vector $\mathbf{x}$ can be obtained by solving a lower triangular system $\mathbf{Ly} = \mathbf{b}$ by forward substitutions and an upper triangular system $\mathbf{Ux} = \mathbf{y}$ by back substitutions sequentially [1]. Comparing with the Gaussian elimination technique, the benefit of the LU decomposition method is that the matrix modification (or decomposition) step can be executed independently of the right side vector $\mathbf{b}$ [1]. Thus, when we have solved the system of equations (2.1), we can solve additional systems with the same left side matrix

$\mathbf{A}$ without the matrix decomposition operations [2]. Therefore, the complexity of solving such systems with the same left side matrix is significantly reduced. This property of LU decomposition has a great meaning in practice, which makes the LU decomposition method to be usually the direct scheme of choice in many applications [13]. However, the LU decomposition technique is very complicated, requiring as high as $2N^3/3$ operations [1], including multiplications, divisions and additions.

As the coefficient matrix $\mathbf{A}$ in the normal equations (2.1) is symmetric and positive definite, the efficient Cholesky decomposition method can be used. The Cholesky decomposition is closely related to the Gaussian elimination method [2]. It decomposes the positive definite coefficient matrix $\mathbf{A}$ in exactly one way into a product $\mathbf{A} = \mathbf{U}^T\mathbf{U}$, where $\mathbf{U}$ is an upper triangular matrix with all main diagonal elements positive [2]. Consequently, the system of equations (2.1) can be rewritten as $\mathbf{U}^T\mathbf{U}\mathbf{x} = \mathbf{b}$. Let $\mathbf{y} = \mathbf{U}\mathbf{x}$, we obtain a lower triangular system $\mathbf{U}^T\mathbf{y} = \mathbf{b}$. Therefore, the solution vector $\mathbf{x}$ can be easily obtained by solving a low triangular system $\mathbf{U}^T\mathbf{y} = \mathbf{b}$ and an upper triangular system $\mathbf{U}\mathbf{x} = \mathbf{y}$ sequentially, through forward and back substitutions, respectively. Comparing with the Gaussian elimination method, the Cholesky decomposition method has the advantage that it requires half of the number of operations and half of the memory space [1]. Furthermore, Cholesky decomposition technique is numerically stable [1] as the positive definite matrix $\mathbf{A}$ is nonsingular. However, the complexity of the Cholesky decomposition is as high as $N^3/3$ operations, including multiplication and division operations [1]. It is still too complicated for real-time hardware implementations, especially if the system size N is large [1].

QRD is well known for its numerical stability [1] and widely used in many applications. It solves the system of equation (2.1) in the same way as the LU decomposition [2]; it transforms the coefficient matrix $\mathbf{A}$ as $\mathbf{A} = \mathbf{Q}\mathbf{R}$, where $\mathbf{Q}$ is an orthogonal matrix and $\mathbf{R}$ is an upper triangular matrix [1]. The orthogonal matrices have the property of $\mathbf{Q}\mathbf{Q}^T = \mathbf{Q}^T\mathbf{Q} = \mathbf{I}$ and $\mathbf{Q}^{-1} = \mathbf{Q}^T$ [1], where $\mathbf{I}$ is an $N \times N$ identity matrix. Therefore, the system (2.1) can be transformed into an upper triangular system $\mathbf{R}\mathbf{x} = \mathbf{Q}^T\mathbf{b}$ and the solution vector $\mathbf{x}$ can be obtained easily through back substitution operations.

QRD is equivalent to computing an orthogonal basis for a set of vectors [1]. There are several choices for actually computing the QRD, such as by means of the Householder reflections and Givens rotations [1]. Reflections and rotations are quite computationally attractive as they are easily constructed and can be used to introduce zeros in a vector by properly choosing the rotation angle or the refection plane [1]. The Householder reflections are extremely useful for introducing zeros to annihilate all elements (except the first one) of a vector [1]. In contrast, Givens rotations could introduce zeros to a vector

more selectively, including the first element [1]. Therefore, Givens rotations is usually the transformation of choice [1].

By using Givens rotations, QRD is inherently well-suited to hardware implementation, exploiting parallel processing and pipeline capabilities of a systolic array structure [14] [15], which consists of an array of individual processing cells arranged as a triangular structure. Each individual processing cell in such array has its own local memory and is connected only to its nearest cells [6]. The special architecture of the array makes regular streams of data to be pipelined through the array in a highly rhythmic fashion. This simple and highly parallel systolic array enables simple data flow and high throughput with pipelining [16]. Therefore, it is well suited for implementing complex signal processing algorithms, particularly for real-time and high data bandwidth implementations [6]. However, the complexity of this triangular array architecture is highly related to the system size; the number of processing cells $(N^2+N)/2$ grows dramatically with the increasing of matrix size N, which makes a direct hardware design of the systolic array very expensive for most practical applications, e.g., adaptive beamforming, requiring multiple, rather than one chip solutions [17]. Therefore, the triangular architecture is only feasible for matrices with small size [16].

Alternatively, some less complicated architecture arrays can be used for solving large size matrices. In [18], a linear architecture systolic array for QRD is obtained through direct projection of the triangular architecture systolic array; each processing cell of the linear array corresponds to the processing cells of each row in the triangular array. This linear array reduces the number of processing cells to N. However, the processing cell of the linear array in [18] is much more complicated than that of the triangular array, as it is obtained by merging the functions of the diagonal and off-diagonal cells of the triangular array. Moreover, not all the processing cells are utilized 100%. Another kind of linear architecture systolic array [19] [17] is obtained by using the folding and mapping approach on the triangular systolic array. This kind of linear array retains the local interconnections of the triangular systolic array and requires only $M+1$ processing cells ($N=2M+1$ is the matrix size). These processing cells have the similar complexity with that of the triangular array. Moreover, these processing cells are 100% utilized. In [15], another solution was proposed to avoid the triangular architecture array by combining similar processing cells of the triangular array, adding memory blocks and using a control logic to schedule the data movements between blocks. The combined architecture is less complicated at the expense of heavier latency, compared to the linear architectures. Therefore, the linear architecture arrays [17] [18] and the combined architecture [15] provide a balance trade-off between the performance and complexity [17]; they require more control logic, heavier latency, but the required number of processing cells is significantly reduced [16],

compared to the triangular structure systolic array.

The classical Givens rotations contain square-root, division and multiplication operations [15], which are expensive for hardware implementation. There has been a lot of previous work on efficient systolic array implementation of QRD using Givens rotations on hardware and they can be divided into three main types. The first type of QRD using Givens rotations is based on the the COordinate Rotation DIgital Computer (CORDIC) algorithms. The CORDIC algorithm is an iterative technique for computing trigonometric functions such as sin and cosin [15]. The CORDIC algorithm is simple as it requires bit-shift and addition operations only, and does not require any multiplication, division and square-root operations [15]. Therefore, it is quite suitable for fixed-point hardware implementation. However, due to the limited dynamic range of the fixed-point representation in CORDIC algorithms, the wordlength requirement is much greater than in floating-point implementations for the same accuracy [15]. Moreover, there are larger errors due to many sub-rotations of the CORDIC algorithm [15].

The second type of QRD using Givens rotations is based on a square-root-free version of the Givens rotations or "Squared Givens Rotations" (SGR) [20], which eliminates the need for square-root operations and half number of the multiplications [15]. The SGR method has several benefits compared to the classical Givens rotation and the CORDIC technique. First of all, it is simple as it does not require square-root operations. On the other hand, its numerical accuracy becomes worse, comparing with the classical Givens rotations with square-root operations. Secondly, it is much faster than the CORDIC technique [15] [16]. The SGR-based hardware design requires about twice smaller area at the expense of about twice larger number of block multipliers and results in only about 66% latency compared to the CORDIC-based hardware design [15] [16].

The third type of QRD using Givens rotations is based on Logarithmic Number Systems (LNS) arithmetic. In the LNS arithmetic, the square-root operations of conventional number system become simple bit-shift operations, and multiplication and division operations of conventional number system become addition and subtraction operations, respectively [21]. However, the simple addition or substraction operation in the conventional number system becomes much more costly in the LNS arithmetic [21]. Therefore, even though the Givens rotation operations using the LNS arithmetic are multiplication, division and square-root free, addition operations make it complicated for hardware implementation. In addition, the LNS-based QRD requires a number of converting operations to fit into the conventional number system based design.

Traditionally, systolic array-based QRD is used for large size systems [22] [23]. For

small size matrices, there are some alternative algorithms which are faster, more hardware efficient than the QRD, while still providing sufficient numerical stability [22] [23]. One straightforward way for matrix inversion is the analytic approach [22]. For example, the inversion of a 2×2 matrix using the analytic approach is computed as follows [22]:

$$\mathbf{B}^{-1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}. \tag{2.4}$$

For small size matrices, the complexity of the analytic approach is significantly smaller than that of the QRD. Therefore, the analytic approach is quite efficient for inversions of small size matrices, such as a 2×2 matrix in equation (2.4). However, the complexity of the analytic approach grows very quickly as the size N of the matrix increases, which makes it only suitable for small size matrices. Moreover, the direct analytic matrix inversion is sensitive to finite-length errors [22]. Even for 4×4 matrices, the direct analytic approach is unstable due to the large number of subtractions involved in the computation which might introduce cancellation [22].

In [22], a method called blockwise analytic matrix inversion (BAMI) is proposed to compute the inversion of complex-valued matrices. It partitions the matrix into four smaller matrices, and then computes the inverse based on computations of these smaller parts. For example, to compute a 4×4 matrix $\mathbf{B}$, it is first divided into four 2×2 submatrices [22]

$$\mathbf{B} = \begin{bmatrix} \mathbf{B}_1 & \mathbf{B}_2 \\ \mathbf{B}_3 & \mathbf{B}_4 \end{bmatrix}. \tag{2.5}$$

Consequently, the inversion of the matrix $\mathbf{B}$ can be computed by the inversion of these 2×2 matrices using the analytic method (2.4), i.e., [22]

$$\mathbf{B}^{-1} = \begin{bmatrix} \mathbf{B}_1^{-1} + \mathbf{B}_1^{-1}\mathbf{B}_2(\mathbf{B}_4 - \mathbf{B}_3\mathbf{B}_1^{-1}\mathbf{B}_2)^{-1}\mathbf{B}_3\mathbf{B}_1^{-1} & -\mathbf{B}_1^{-1}\mathbf{B}_2(\mathbf{B}_4 - \mathbf{B}_3\mathbf{B}_1^{-1}\mathbf{B}_2)^{-1} \\ -(\mathbf{B}_4 - \mathbf{B}_3\mathbf{B}_1^{-1}\mathbf{B}_2)^{-1}\mathbf{B}_3\mathbf{B}_1^{-1} & (\mathbf{B}_4 - \mathbf{B}_3\mathbf{B}_1^{-1}\mathbf{B}_2)^{-1} \end{bmatrix}. \tag{2.6}$$

The BAMI approach is more stable than the direct analytic method, due to the fewer number of subtractions and it requires fewer number of bits to keep the precision [22]. Therefore, the BAMI method provides a good alternative to the classical QRD for solving small size matrices such as 4×4 matrices. However, for 2×2 and 3×3 matrices, the direct analytic method is preferred [22].

The Sherman-Morrison equation is a special case of the matrix inversion lemma, allowing the easy computation of the inverse of a series of matrices where two successive matrices differ only by a small perturbation [1]. The perturbation has to have the form of a rank-1 update, e.g., $\mathbf{u}\mathbf{v}^H$, where $\mathbf{u}$ and $\mathbf{v}$ are vectors with appropriate sizes [24]. Given

$\mathbf{A}^{-1}$, the Sherman-Morrison formula is expressed as [1]

$$(\mathbf{A}^{-1} + \mathbf{u}\mathbf{v}^H)^{-1} = \mathbf{A}^{-1} - \frac{(\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^H)\mathbf{A}^{-1}}{1 + \mathbf{v}^H\mathbf{A}^{-1}\mathbf{u}}. \tag{2.7}$$

Consequently, for a series of matrices $\mathbf{A}(i) = \mathbf{A}(i-1) + \mathbf{u}(i)\mathbf{u}^H(i)$, e.g. autocorrelation matrices, where i is the time index and $\mathbf{u}(i)$ is the input vector, $\mathbf{A}^{-1}(i)$ can be computed easily by using the Sherman-Morrison formula [24], i.e.,

$$\begin{aligned} \mathbf{A}^{-1}(i) &= [\mathbf{A}(i-1) + \mathbf{u}(i)\mathbf{u}^H(i)]^{-1} \\ &= \mathbf{A}^{-1}(i-1) - \frac{\mathbf{A}^{-1}(i-1)\mathbf{u}(i)\mathbf{u}^H(i)\mathbf{A}^{-1}(i-1)}{1 + \mathbf{u}^H(i)\mathbf{A}^{-1}(i-1)\mathbf{u}(i)}. \end{aligned} \tag{2.8}$$

This approach of matrix inversion is widely used, e.g. in the MIMO systems [25]. However, the equation (2.8) requires a large number of multiplications and divisions, which makes this method difficult for hardware implementation. In [24], the divisions in (2.8) are translated into multiplications by introducing appropriate scaling, i.e.,

$$\begin{aligned} \tilde{\mathbf{A}}^{-1}(i) &= [\alpha(i-1) + \mathbf{u}^H(i)\tilde{\mathbf{A}}^{-1}(i-1)\mathbf{u}(i)] \left[\tilde{\mathbf{A}}^{-1}(i-1) + \frac{\mathbf{u}(i)\mathbf{u}^H(i)}{\alpha(i-1)}\right]^{-1} \\ &= \tilde{\mathbf{A}}^{-1}(i-1)[\alpha(i-1) + \mathbf{u}^H(i)\tilde{\mathbf{A}}^{-1}(i-1)\mathbf{u}(i)] \\ &\quad -[\tilde{\mathbf{A}}^{-1}(i-1)\mathbf{u}(i)\mathbf{u}^H(i)\tilde{\mathbf{A}}^{-1}(i-1)] \end{aligned} \tag{2.9}$$

where $\tilde{\mathbf{A}}^{-1}(i) = \alpha(i)\mathbf{A}^{-1}(i)$ and the scaling factor

$$\alpha(i) = \alpha(i-1)[\alpha(i-1) + \mathbf{u}^H(i)\tilde{\mathbf{A}}^{-1}(i-1)\mathbf{u}^H(i)] \tag{2.10}$$

with $\alpha(0) = 1$. However, the modified Sherman-Morrison equation (2.9) is still very complicated, with a complexity of $\mathcal{O}(N^2)$ multiplications, which are expensive for hardware design.

The direct methods, such as the Gaussian, Cholesky and QRD and many others, have complexity of $\mathcal{O}(N^3)$ operations, requiring division and multiplication operations. The modified Sherman-Morrison method requires about $\mathcal{O}(N^2)$ multiplications. Therefore, direct methods are difficult for real-time signal processing and hardware implementations. The direct methods compute an exact solution after a finite number of pre-specified operations [1]. They only give out results after executing all prespecified operations. Therefore, if we stop early, the direct methods give out nothing [2]. Moreover, the direct methods may be prohibitively expensive to solve the very large or sparse systems of linear equations [1].

2.2.2 Iterative Methods

In contrast to the direct methods are the iterative methods, which are quite efficient for both very large systems and very sparse systems [2]. The iterative methods produce a sequence of successively better approximations $\mathbf{x}^{(k)}$ (k is the iteration index), which hopefully converge to the optimal solution [2], and essentially involve the coefficient matrix $\mathbf{A}$ only in the context of the matrix-vector multiplication operations [1].

Solving the normal system of equations (2.1) can be formulated as a process of minimization of the quadratic function [2]

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{x}^T\mathbf{b}. \tag{2.11}$$

The minimum value of $f(\mathbf{x})$ is $-\frac{1}{2}\mathbf{b}^T\mathbf{A}^{-1}\mathbf{b}$, obtained by setting $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ which is exactly the solution of the normal equations (2.1) [1]. Consequently, most iterative methods solve the normal equations (2.1) by minimizing the function $f(\mathbf{x})$ iteratively [1]. Each of them begins from an initial guess $\mathbf{x}^{(0)}$ and generates a sequence of iterates $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$, ... At each step (or iteration), $\mathbf{x}^{(k+1)}$ is chosen as $f(\mathbf{x}^{(k+1)}) \leq f(\mathbf{x}^{(k)})$, and $f(\mathbf{x}^{(k+1)}) < f(\mathbf{x}^{(k)})$ is preferable [2]. Therefore, we get closer to the minimum value of $f(\mathbf{x})$ step by step. If we obtain $\mathbf{A}\mathbf{x}^{(k)} = \mathbf{b}$ or nearly so after some iterations, we can stop and accept $\mathbf{x}^{(k)}$ as the solution of the system (2.1).

Computing the step from $\mathbf{x}^{(k)}$ to $\mathbf{x}^{(k+1)}$ has two ingredients: 1) choosing a direction vector $\mathbf{p}^{(k)}$ that indicates the direction in which we will travel to get from $\mathbf{x}^{(k)}$ to $\mathbf{x}^{(k+1)}$; and 2) choosing a point on the line $\mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{p}^{(k)}$ as $\mathbf{x}^{(k+1)}$, where $\alpha^{(k)}$ is the step size chosen to minimize $f(\mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{p}^{(k)})$ [2]. The process of choosing $\alpha^{(k)}$ is called the *line search* [2]. We want to choose an appropriate $\alpha^{(k)}$ to make $f(\mathbf{x}^{(k+1)}) \leq f(\mathbf{x}^{(k)})$. One way to ensure this is to choose $\alpha^{(k)}$ to let

$$f(\mathbf{x}^{(k+1)}) = \min f(\mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{p}^{(k)}); \tag{2.12}$$

this is called an *exact line search*, otherwise it is an *inexact line search* [2].

There are two main types of iterative methods: the nonstationary methods and the stationary methods. Nonstationary methods are a relatively recent development, including the steepest descent method, the CG method and many others; they are usually complicated, but they can be highly effective [26]. Stationary iterative methods are older, simple, but usually not as effective as the nonstationary methods [26]. There are three common stationary iterative methods for linear systems: Jacobi, Gauss-Seidel and SOR methods. These methods will be analyzed below.

One of the well-known iterative techniques is the steepest descent method [1]. It performs the exact line search in the direction of negative gradient

$$\mathbf{p}^{(k)} = -\nabla f(\mathbf{x}^{(k)}) = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)} = \mathbf{r}^{(k)}, \tag{2.13}$$

and we call $\mathbf{r}^{(k)}$ as the residual vector of the solution $\mathbf{x}^{(k)}$ and the step size is chosen as [1]

$$\alpha^{(k)} = \frac{(\mathbf{r}^{(k)})^T\mathbf{r}^{(k)}}{(\mathbf{r}^{(k)})^T\mathbf{A}\mathbf{r}^{(k)}}. \tag{2.14}$$

The method of steepest descent is easy to program, but it often converges slowly [2]. The main reason for its slow convergence is that the steepest descent method may spend time on minimizing $f(\mathbf{x}^{(k)})$ along parallel or nearly parallel search directions [2]. The complexity of the steepest descent method is high, requiring $\mathcal{O}(N^2)$ multiplications, divisions and additions per iteration.

The CG algorithm is a simple variation of the steepest descent method that has a fast convergence speed. It is based on the idea that the convergence speed to the optimal solution could be accelerated by minimizing $f(\mathbf{x}^{(k)})$ over the hyperplane that contains all previous search directions, i.e.,

$$\mathbf{x}^{(k)} = \alpha^{(0)}\mathbf{p}^{(0)} + \alpha^{(1)}\mathbf{p}^{(1)} + ... + \alpha^{(k-1)}\mathbf{p}^{(k-1)}, \tag{2.15}$$

instead of minimizing $f(\mathbf{x}^{(k)})$ over just the line that points down gradient [2], which actually happens in the steepest descent method. Due to its fast convergence, the CG method has already been used for adaptive filtering for a long time (e.g., see [27–30] and references therein). However, the complexity of the CG iteration is $\mathcal{O}(N^2)$, including divisions, multiplications and additions, which is often too high for real time signal processing.

The Jacobi method perhaps is the simplest iterative method [1]. It updates next iterate $\mathbf{x}^{(k+1)}$ beginning with an initial guess $\mathbf{x}^{(0)}$ by solving each element of $\mathbf{x}$ in terms of [1]

$$x_n^{(k+1)} = \left(b_n - \sum_{p\neq n} A_{n,p}x_p^{(k)}\right)/A_{n,n}, \tag{2.16}$$

where $A_{n,p}$, b_n and x_n are (n,p)-th element of the coefficient matrix $\mathbf{A}$, and n-th elements of vector $\mathbf{b}$ and $\mathbf{x}$, respectively. The Jacobi method has the advantage that all elements of the correction $\mathbf{x}^{(k+1)}$ can be performed simultaneously as all elements of new iterate $\mathbf{x}^{(k+1)}$ are independent to each other; therefore the Jacobi method is inherently parallel [2]. On the other hand, the Jacobi method does not use the most recent available information to compute $x_n^{(k+1)}$ [1] as shown in (2.16). Therefore, the Jacobi method needs to store two copies of $\mathbf{x}$, since $\mathbf{x}^{(k)}$ can only be overwritten until the next iterate $\mathbf{x}^{(k+1)}$ is obtained [2].

If the system size N is large, each copy of $\mathbf{x}$ will occupy large memory space. Moreover, the Jacobi method requires nonzero diagonal elements of the matrix $\mathbf{A}$ to avoid division by zero in (2.16), which can usually be achieved by permuting rows and columns if the condition is not already true. Furthermore, the Jacobi method does not always converge to the optimal solution [26]. But the convergence of the Jacobi method is guaranteed under the conditions that are often satisfied (e.g., if matrix $\mathbf{A}$ is strictly diagonally dominant), even though the convergence speed may be very slow [26].

The Gauss-Seidel method is obtained by revising the Jacobi iteration to make it using the most current estimation of the solution $\mathbf{x}$; the Gauss-Seidel iterations are performed using each new component as soon as it has been computed rather than waiting until the next iteration [2], which actually happens in the Jacobi method. This feature gives the Gauss-Seidel method in terms of [2]

$$x_n^{(k+1)} = \left(b_n - \sum_{p<n} A_{n,p} x_p^{(k+1)} - \sum_{p>n} A_{n,p} x_p^{(k)} \right) / A_{n,n}. \tag{2.17}$$

Therefore, the Gauss-Seidel method can store each new element $x_n^{(k+1)}$ immediately in the place of old $x_n^{(k)}$, saving memory space and making programming easier [26]. On the other hand, the Gauss-Seidel iterations can only be performed sequentially as each component of $\mathbf{x}^{(k+1)}$ only depends on previous ones; the Gauss-Seidel is inherently sequential [2]. Moreover, the Gauss-Seidel method also requires some conditions to guarantee its convergence; conditions that are somewhat weaker than those for Jacobi method (e.g., if the matrix is symmetric and positive definite) [26]. However, even though the Gauss-Seidel method may converge very slowly, it converges faster and needs only slightly more than half as many iterations as the Jacobi method to obtain the same accuracy [2]. Due to the explicit division in (2.17), the Gauss-Seidel method also requires non-zero diagonal elements of the matrix $\mathbf{A}$. Gauss-Seidel iterations are widely used in, e.g. adaptive filtering [31, 32].

The relaxation is the process of correcting an equation by modifying one unknown [2]. Therefore, the Jacobi method performs simultaneous parallel relaxation and the Gauss-Seidel method performs successive relaxation [2]. The over-relaxation is the technique making a somewhat bigger correction, rather than making only a correction for which the equation is satisfied exactly [2]. The over-relaxation scheme could accelerate the convergence substantially [2]. The successive over-relaxation method, or SOR method, is obtained by applying extrapolation to the Gauss-Seidel technique; it takes a weighted average between the previous iterate and the current Gauss-Seidel iteration successively, i.e.,

$$\mathbf{x}^{(k+1)} = w\mathbf{x}_{GS}^{(k+1)} + (1-w)\mathbf{x}^{(k)}, \tag{2.18}$$

where $\mathbf{x}_{GS}^{(k+1)}$ is the next iterate given by the Gauss-Seidel method and $w > 1$ is the extrapolation factor [26]. The value of w decides the accelerate rate of the convergence speed. If w is chosen as $w = 1$, the SOR method collapses to the Gauss-Seidel method. The parameter w can also be chosen as $w < 1$, which amounts to under-relaxation, but this choice normally leads to slow convergence [2]. With optimal value for w, the SOR method can be an order of magnitude faster than that of the Gauss-Seidel method [26]. However, choosing optimal w is difficult in general except for special classes of matrices [26]. As SOR is based on the "successive" relaxation (Gauss-Seidel iterations), it also only requires to keep one copy of $\mathbf{x}$ [2].

Comparing with the direct methods, the iterative methods have several advantages: 1) they may require less memory than direct methods; 2) they may be faster than direct methods; and 3) they may handle special structures (such as sparse) in a simpler way [2]. Furthermore, the iterative techniques have the ability to exploit a good initial guess, which could reduce the number of iterations required to get the solution [2]. Even though in theory infinite number of iterations might be required to converge to optimal solution, iterative techniques have the ability to stop the solving process arbitrary according to the required accuracy level. While for the direct methods, they do not have the ability of exploiting an initial guess and they simply execute a predetermined sequence of operations and obtain the solution after all these operations [2].

However, the iterative methods are complex. The nonstationary iterative methods, such as the steepest descent method and the CG method we presented above, are highly effective with a complexity of $\mathcal{O}(N^2)$ operations per iteration. The stationary iterative methods, such as the Jacobi, Gauss-Seidel and SOR algorithms, are less complicated, with a complexity of $\mathcal{O}(N)$ operations per iteration, at the expense of less efficiency. The iterative methods contain division and multiplication operations, making them expensive for real-time signal processing and hardware designs.

The DCD algorithm [5] is a nonstationary iterative method, but based on stationary coordinate descent techniques. It is simple for implementation, as it does not need multiplication or division operations. For each iteration, it only requires $\mathcal{O}(N)$ additions or $\mathcal{O}(1)$ additions. Therefore, the DCD algorithm is quite suitable for hardware realization. The disadvantage of the DCD algorithm is that it requires infinite number of iterations (or updates) in theory to converge to an optimal solution. Therefore, the number of updates and processing time is individual for each system to be solved. The relationship between the number of updates and the linear systems to be solved is very complicated and it is not possible to predict an accurate number of updates. This is an common drawback for the iterative techniques, such as the steepest descent, CG, Jacobi and Gauss-Seidel meth-

ods. However, the DCD algorithm has the lowest complexity per iteration and performs a similar convergence speed to other iterative methods, such as the CG, Gauss-Seidel and CD algorithms. In this thesis, the DCD algorithm will be analyzed and implemented into FPGA chips.

2.3 Adaptive Filtering

An adaptive filter is a filter that self-adjusts the filter coefficients according to a recursive algorithm, which makes the filter to perform satisfactorily in an environment where the statistics of the input signals are not available or time varying [6]. Adaptive filters are widely used in the areas of system identification, channel equalization, channel identification, interference suppression, acoustic echo cancellation and others.

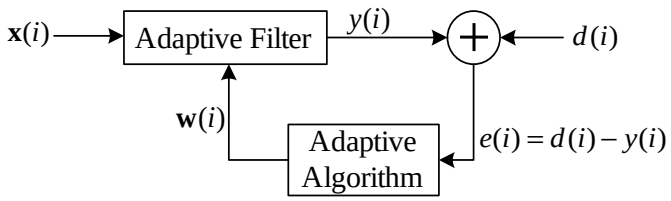


Figure 2.1: Basic structure of an adaptive filter [33]

The basic structure of an adaptive filter is shown in Fig. 2.1, where $\mathbf{x}(i)$, $y(i)$, $d(i)$, $e(i) = d(i) - y(i)$ and $\mathbf{w}(i)$ are the input data vector, output signal, desired response signal, error signal and the filter coefficients (or filter weights) at time instant i, respectively. It is seen that the operation of an adaptive filter involves two basic processes which work interactively with each other: 1) a filtering process to generate an output in response to a sequence of input data $\mathbf{x}(i)$ and the filter coefficients $\mathbf{w}(i)$; and 2) an adaptive algorithm to determine how to modify the filter coefficients $\mathbf{w}(i)$ to minimize a cost function on next iteration by observing the error signal $e(i)$ between the filter output $y(i)$ and the desired response signal $d(i)$ [34].

The adaptive filter can be of either finite-duration impulse response (FIR) or infinite-duration impulse response (IIR) structures [33]. The FIR structure is simple and robust as it does not involve any feedback mechanism. Furthermore, many practical problems can be accurately modeled by an FIR filter, e.g., echo cancellation using adaptive transversal filter and antenna beamforming using adaptive linear combiner. The IIR structure contains feedback mechanisms which make it complicated and sometimes unstable. Therefore, the

FIR adaptive filter is many times preferred over the IIR one and the application of the IIR structure in the area of adaptive filter is rather limited [35].

As already mentioned above, the adaptive algorithm adjusts the filter weights $\mathbf{w}(i)$ by minimizing a cost function which is related to the error signal $e(i)$. The choice of one adaptive algorithm over another often involves a trade-off between certain conflicting performance measures. Some of the most important performance measures to choose an adaptive algorithm are [6]:

- Rate of convergence, i.e., the number of iterations required by the adaptive algorithm to converge close enough to a steady-state solution.
- Misadjustment, which quantifies how close the adaptive filter coefficients are to the ones of the optimal filter.
- Tracking ability, i.e., the performance of the filter when operating in a nonstationary environment.
- Robustness to quantization when implemented in finite-precision.
- Computational complexity, including the number of operations, the memory requirements and the investment required to program the algorithm on a computer.

However, these performance measures are often conflicting. Consequently, specifications on the adaptive filter in terms of these measures cannot in general be guaranteed simultaneously. For example, fast convergence rate usually implies high computational complexity requirement. On the other hand, if low misadjustment is desired, a low complexity adaptive algorithm would most likely suffer from slow convergence. Basically, there are two distinct approaches for deriving recursive algorithms for the operation of linear adaptive filters: stochastic gradient approach and LS estimation [6].

The structural basis for the linear adaptive filter using the stochastic gradient approach is a tapped-delay linear, or transversal filter [6]. The cost function of the stochastic gradient approach is defined as the mean-square error (MSE)

$$J_{\mathbf{w}} = \mathrm{E}[e^2(i)], \tag{2.19}$$

where $\mathrm{E}[\cdot]$ denotes the statistical expectation operator, resulting in the widely known Least Mean-Square (LMS) algorithm [6]. The LMS algorithm is very popular due to its low complexity and robustness. The LMS algorithm updates the coefficient vector by taking

a step in the direction of the negative gradient of the cost function, i.e.,

$$\begin{aligned} \mathbf{w}(i+1) &= \mathbf{w}(i) - \frac{\mu}{2}\frac{\partial J_{\mathbf{w}}}{\partial \mathbf{w}(i)} \\ &= \mathbf{w}(i) + \mu e(i)\mathbf{x}(i), \end{aligned} \tag{2.20}$$

where μ is the step size controlling the stability, convergence speed, and misadjustment [6]. The step size μ should be small compared with $1/\lambda_{\text{max}}$, where λ_{max} is the largest eigenvalue of the autocorrelation matrix $\mathbf{R} = \mathrm{E}[\mathbf{x}(i)\mathbf{x}^T(i)]$ of the input data $\mathbf{x}(i)$ [6]. The complexity of the LMS algorithm is as low as $\mathcal{O}(N)$ operations per sample [6]. However, the main drawbacks of the LMS algorithm are a relatively slow convergence rate and a sensitivity to variations in the condition number of the correlation matrix $\mathbf{R}$ [6].

In the standard form of the LMS algorithm in equation (2.20), the adjustment applied to the weight vector is directly proportional to the input vector $\mathbf{x}(i)$ [6]. Therefore, the standard LMS algorithm suffers from a gradient noise amplification problem when the input data $\mathbf{x}(i)$ is large [6]. To eliminate this problem, a normalized LMS (NLMS) algorithm is obtained by substituting the step size μ in equation (2.20) with a time-varying step size

$$\mathbf{w}(i+1) = \mathbf{w}(i) + \mu(i)e(i)\mathbf{x}(i), \tag{2.21}$$

where $\mu(i) = \widetilde{\mu}/(||\mathbf{x}(i)||^2 + \delta)$, $||\mathbf{x}(i)||$ is the Euclidean norm of the input data vector $\mathbf{x}(i)$, $\widetilde{\mu}$ is a fixed step size, and δ is a small positive constant to avoid division by a small value of the squared norm $||\mathbf{x}(i)||^2$. Therefore, the NLMS algorithm mitigates the gradient noise amplification problem. Furthermore, the NLMS algorithm converges faster than the standard LMS algorithm [6]. However, it still converges slowly for colored noise input signals [6].

The method of LS minimizes a cost function that is defined as the sum of weighted error squares [6],

$$J_{\mathbf{w}} = \sum_{k=1}^{i} \lambda^{i-k} e^2(k), \tag{2.22}$$

where $0 < \lambda \leq 1$ is an exponential scaling factor (or forgetting factor). The method of LS may be formulated with two structures, block estimation and recursive estimation [6]. The block estimator updates the filter coefficients on a block-by-block basis; all blocks contain equal length (duration) of input data stream. In contrast, the recursive estimator estimates the filter coefficients on a sample-by-sample basis [6]. Due to its less memory storage, the recursive estimation structure is widely used in practice and results in the well known RLS algorithm.

Differentiating the cost function $J_{\mathbf{w}}$ (2.22) with respect to $\mathbf{w}(i)$ and solving for minimum, we obtain [6]

$$\mathbf{R}(i)\mathbf{w}(i) = \boldsymbol{\beta}(i), \tag{2.23}$$

where

$$\begin{aligned} \mathbf{R}(i) &= \sum_{k=1}^{i} \lambda^{i-k}\mathbf{x}(k)\mathbf{x}^T(k) \\ &= \lambda\mathbf{R}(i-1) + \mathbf{x}(i)\mathbf{x}^T(i) \end{aligned} \tag{2.24}$$

is the correlation matrix of the input data vector $\mathbf{x}(i)$, and

$$\begin{aligned} \boldsymbol{\beta}(i) &= \sum_{k=1}^{i} d(k)\mathbf{x}(k) \\ &= \lambda\boldsymbol{\beta}(i-1) + d(i)\mathbf{x}(i) \end{aligned} \tag{2.25}$$

is the cross correlation vector between the input data vector $\mathbf{x}(i)$ and the desired response $d(i)$. Thus, the filter weights can be obtained as

$$\mathbf{w}(i) = \mathbf{R}^{-1}(i)\boldsymbol{\beta}(i). \tag{2.26}$$

In practice, the classical RLS algorithm employs the matrix inversion lemma to calculate the inversion of matrix $\mathbf{R}(i)$ recursively, avoiding direct computing the matrix inversion $\mathbf{R}^{-1}(i)$ at each time instant. The convergence rate of the RLS algorithm is much faster than that of the LMS algorithm [6], which makes it widely used in many signal processing areas. However, such improvement is achieved at the expense of an increase in computational complexity, which is $\mathcal{O}(N^2)$ operation per sample [6].

Another major limitation of the classical RLS algorithm is the potential divergence behavior in finite-precision environment [33]. The stability problems are usually caused by lost symmetry and positive definiteness of the matrix inversion $\mathbf{R}^{-1}(i)$ [33]. Most robust implementations of the RLS adaptive algorithm are based on QRD of the matrix $\mathbf{R}(i)$. The QRD-RLS algorithm works directly with the incoming data $\mathbf{x}(i)$, rather than working with the (time-average) correlation matrix $\mathbf{R}(i)$ as in the standard RLS algorithm [6]. Accordingly, the QRD-RLS algorithm is robust to numerical errors compared to the classical RLS algorithm [6] [35]. However, the QRD-RLS algorithm is complicated, requiring $\mathcal{O}(N^2)$ operations per sample. Moreover, as we present above, the hardware design of the QRD is very complicated, requiring a large number of logic resources with a heavy processing latency.

2.4 Hardware Reference Implementations

The processes of solving linear systems of equations and matrix inversion have been for a long time considered to be too hard a task to be implemented within real-time systems. Consequently, hardware reference designs have only started appearing relatively recently. Most of the related hardware designs are based on the QRD using Givens rotations [4, 15–18, 36–41].

Karkooti *et al.* [15] implemented a 4×4 floating-point complex-valued matrix inversion core on the Xilinx Virtex4 XC4VLX200 FPGA chip, based on the QRD algorithm via SGR. The design uses 21-bit data format; 14 bits for mantissa, 6 bits for exponent of floating-point number and 1 sign bit. The design was implemented using the Xilinx System Generator tool [42], calling on the Xilinx Core Generator [43] to implement a floating-point divider block. To make the design able to fit in one single chip, a combined architecture of the systolic array was implemented. Internally, the design consists of one diagonal cell, one off-diagonal internal cell and a back substitution block. It also requires block RAMs and a control unit to schedule the movement of data between these blocks. Compared to the triangular architecture systolic array, the combined architecture makes the design less complicated at the expense of heavier latency. The area usage of this design is about 9117 logic slices and 22 DSP48 (also known as "XtremeDSP") blocks [44] with a latency of 933 clock cycles (777 cycles for QRD and 156 cycles for back substitution). The Virtex-4's DSP48 block is a configurable multiply-accumulate block based on an 18-bit×18-bit hardware multiplier [44]. The design can be extended to other size matrices with a slight modification of the control unit and the RAM size.

Edman *et al.* [18] implemented a 4×4 fixed-point complex-valued matrix inversion core on the Xilinx Virtex-II FPGA chip using QRD via SGR. The design is based on a linear architecture systolic array which is obtained through direct projection of the triangular architecture systolic array. This linear array requires only $2N$ processing cells for QRD and back substitutions. However, the processing cells are very complicated and not all of them are of 100% utilization. The most complicated complex-valued divider was realized using 9 multipliers, 3 adders and a look-up table, with 5 pipelining stages. The implementation with 19-bit fixed-point data format consumes 86% of the Virtex-II chip area with a latency of 175 cycles. Unfortunately, the paper does not point out the FPGA chip model and the area usage in terms of the logic slices, RAMs and multipliers in detail.

Liu *et al.* [17] implemented a floating-point QRD array processor based on SGR for adaptive beamforming on a TSMC (Taiwan Semiconductor Manufacturing Company)

0.13 micron chip. The design is based on a N-cell linear architecture systolic array obtained by using the folding and mapping approach on the triangular systolic array. The processing cell has similar complexity to the cell in the triangular systolic array and has a utilization of 100%. By using parameterized arithmetic processors, the design provides a very elegant and direct approach to create a generic core for implementing the QR array processor. For the case of a 41-element antenna system in which data is represented by a 14-bit mantissa and 5-bit exponent, the linear array QR processor comprises 21 processing cells and utilizes 1060-K gates, corresponding to a maximum clock rate of 150 MHz. The authors did not implement their linear array into FPGA chip. However, as the basic operations of SGR is complex, we could get a conclusion that this linear array is complicated for FPGA implementation and not possible to implement a large size system such as 41-element in a single FPGA chip.

Myllyla *et al.* [16] implemented a fixed-point complex-valued minimum mean square error (MMSE) detector for MIMO OFDM (orthogonal frequency-division multiplexing) system for both 2×2 and 4×4 cases on the Xilinx Viretex-II XC2V6000 FPGA chip. The matrix operation of this design is based on the systolic array, using CORDIC-based QRD and SGR-based QRD. A fast and parallel triangular structure of the systolic array is considered for 2×2 antenna systems and a less complicated linear architecture with easy scalability and time sharing processing cells is considered for 4×4 systems. The CORDIC-based design is implemented using VHDL, whilst the SGR-based design is implemented using the System Generator. For 2×2 and 4×4 systems, the CORDIC-based QRD design in which data is represented in a 16-bit fixed-point data format, requires 11910 and 16805 slices, 6 and 101 block RAMs, 20 and 44 18-bit×18-bit embedded multipliers, with 685 and 3000 cycles latency, respectively. The SGR-based QRD only implemented for the 2×2 system, requires 6305 slices, 8 block RAMs and 59 18-bit×18-bit embedded multipliers with a latency of 415 cycles; it uses 19-bit fixed-point data format. Obviously, the CORDIC-based design requires more slices and less multipliers compared to the SGR-based design. This is because the SGR is based on normal arithmetic operations, while the CORDIC is based on multiplier- and divider-free rotation operations [16].

There are also many commercial QR intellectual property (IP) cores that use the CORDIC algorithm. Altera has published a CORDIC-based QRD-RLS design [36] [37] using their CORDIC IP core [45] that supports applications such as smart antenna-beamforming [46], WiMAX [47], channel estimation and equalization of 3G wireless communications [48]. Altera's CORDIC IP block has a deeply pipelined parallel architecture enabling speed over 250MHz on their Stratix FPGAs. In [36] [37], they explore a number of different degrees of parallelism of CORDIC blocks for performing matrix decomposition for 64 input vectors for 9-element antenna with 16-bit data on an Altera

Stratix FPGA. The required logic resources of the CORDIC cores can be as low as 2600 logic elements (equivalent to 1300 Xilinx slices [49] [50]). When the CORDIC cores run at 150 MHz, the design obtains an update rate of 5 kHz and the processing latency is about 29700 cycles. The logic resources of other modules are not given. An embedded NIOS processor is used to perform the back substitution with a latency about 12000 cycles for 9×9 matrix. However, the QRD and back substitution can not be executed in a pipelined scheme. Therefore, the total latency of the design is about 41700 cycles.

Xilinx has a similar CORDIC IP core [51]. Dick *et al.* [38] implemented a complex-valued folded QRD with subsequent back-substitution on a Virtex-4 FPGA device using the Xilinx System Generator tool [42]. The folded design contains one diagonal cell (CORDIC-based), one off-diagonal cell (DSP48-based) of the systolic array, and a back-substitution cell, with block RAMs and a control unit to schedule the movement of data between these blocks. All of them together cost about 3530 slices, 13 DSP48 blocks and 6 block RAMs. For solving a 9×9 system of equations, the proposed design results in about 10971 cycles. The area usage of the CORDIC-based QRD is much smaller compared to Dick's 2005 [15] 4×4 matrix inversion core using SGR-based QRD; this is because it uses fixed-point rather than floating-point arithmetic, and the SGR operation is based on normal arithmetic operations, while the CORDIC operation is based on multiplier- and divider-free rotation operations [16].

Xilinx also has QR decomposition, QR inverse and QRD-RLS spatial filtering IP cores available in the AccelWare DSP IP Toolkits [52] originally from AccelChip, Inc. AccelWare is a library of floating-point Matlab model generators that can be synthesized by AccelDSP into efficient fixed-point hardware. These AccelWare DSP cores are used in WiMAX (Worldwide Interoperability for Microwave Access) baseband MIMO systems [53] and for beamforming [4] applications. Uribe *et al.* [4] described a 4-element beamformer based on the QRD-RLS algorithm with CORDIC-based Givens rotations. The resources required on the target device (a Xilinx Virtex-4 XC4VSX55 FPGA) are 3076 logic slices and one DSP48 block. The number of slices and DSP48 blocks is further reduced compared to the design in [38]; this is because the design in [4] is mainly based on CORDIC operation, whilst only the diagonal cell is based on CORDIC operation in [38]. The sample throughput rate of the design is quoted at 1.7 MHz. The authors do not state the clock speed of the device they are using, but their chosen device (an XC4VSX55) is available in 400MHz, 450MHz and 500MHz variants, which would give 235, 265 and 294 cycles, respectively.

Matousek *et al.* implemented a diagonal cell of the systolic array for QRD using their high-speed logarithmic arithmetic (HSLA) library [39]. In this design, the LNS

format is applied, i.e., the number is divided into an integer part, which always has 8 bits, and a fractional part, the size of which depends on the data precision. For 32-bit LNS format, their QRD diagonal cell consumes about 3000 slices of Xilinx Virtex-E XCV2000E FPGA devices with 13 cycles latency. For comparison, a floating-point QRD diagonal cell in which data is represented by a 23-bit mantissa and 8-bit exponent is also implemented, which requires 3500 slices with a latency of 84 cycles. It is obvious that the LNS arithmetic based design is much faster than the conventional arithmetic based design.

Schier *et al.* uses the same HSLA library as in [39] to implement floating-point operations for Givens rotations [40] and QRD-RLS algorithm [41], based on a Xilinx Virtex-E XCV2000E FPGA. Two LNS data formats are implemented: one is a 19-bit LNS format and the other is a 32-bit LNS format. Only one diagonal cell and one off-diagonal cell of the systolic array, not the full array, are implemented. Since addition and subtraction become the most computationally complex modules in an LNS system, they are evaluated using a first-order Taylor-series approximation with look-up tables. Even by using an error correction mechanism and a range-shift algorithm [54] to minimize the size of the look-up table, the logarithmic addition/subtraction block still requires a large number of slices and memory space. For 19-bit and 32-bit LNS format, one addition/subtraction block requires about 8% and 13% slices, and 3% and 70% block RAMs of a single Xilinx Virtex-E XCV2000E FPGA, respectively. Thus, for 19-bit LNS format, these two cells (one diagonal cell and one off-diagonal cell) require about 4492 slices and 30 block RAMs. The diagonal cell has 11 cycles latency and the off-diagonal element has 10 cycles latency. These two cells could run at 75 MHz and get a throughput about 6.8 MHz since both types of cells are fully pipelined. In contrast, for 32-bit LNS data format, only one logarithmic addition/subtraction block can fit on the Xilinx Virtex-E XCV2000E FPGA limited by the number of block RAMs.

Eilert *et al.* [23] implemented a 4×4 complex-valued matrix inversion core in floating-point format for the Xilinx Virtex-4 FPGA based on their BAMI algorithm [22]. The design was implemented using the Xilinx Core Generator [43] to generate all basic units, such as the floating-point real adders, subtractor, real multipliers and real dividers. As the multiplications and divisions are executed using logic gates, not the embedded multipliers or the DSP48 blocks, the BAMI design requires a large chip area. It requires overall 7312 slices and 9474 slices for 16-bit and 20-bit floating-point data formats, respectively. The 16-bit design and 20-bit design run at 120 MHz and 110 MHz, respectively, both with 270 cycles latency.

LaRoche *et al.* [24] synthesised a 4×4 complex-valued matrix inversion unit based

Table 2.1: Comparison of FPGA-based matrix inversion and linear system of equation solvers. (MULT = multiplier)

Matrix Size	**Technique**	**Logic Slices**	**Extras**	**Latency Cycles**
2×2	QRD-SGR [16]	6305	59 MULTs	415
4×4	QRD-SGR [15]	9117	22 DSP48s	933
2×2	QRD-CORDIC [16]	11910	20 MULTs	685
4×4	QRD-CORDIC [16]	16805	44 MULTs	3000
4×4	QRD-CORDIC [4]	3076	1 DSP48s	265
9×9	QRD-CORDIC [36]	1300	1 NIOS Processor	41700
9×9	QRD-CORDIC [38]	3530	13 DSP48s	10971
4×4	BAMI [23]	9474	-	270
4×4	modified Sherman-Morrison [24]	4446	101 MULTs	64
1 diagonal cell	QRD-LNS [39]	3000	-	13
1 diagonal cell	QRD [39]	3500	-	84
1 diagonal cell 1 off-diagonal cell	QRD-LNS [41] [40]	4492	-	11 10

on the modified Sherman-Morrison equation (2.9) using Xilinx Synthesis Technology (XST) [55] on a Xilinx Virtex II XC2V600 FPGA chip. Even though this modified Sherman-Morrison equation (2.9) does not contain division operations, its FPGA implementation is still very complicated due to a large number of multiplication operations. The design has four main blocks, a matrix-matrix multiplication block, a matrix-vector multiplication block, a vector-vector multiplication block and a scalar-matrix multiplication block, which consume about 3108, 765, 187 and 780 logic slices with 64, 16, 4 and 16 18-bit×18-bit embedded multipliers, respectively. The total area usage written in [24] is 4446 slices and 101 18-bit×18-bit embedded multipliers, which is smaller than the sum of usage of the four main blocks. The author did not give explanation about this difference. Moreover, the number of required RAMs is also not given out. The latency of the design is about 64 cycles.

Table 2.1 compares some of the FPGA implementations that have been mentioned above. In summary, it can be seen that current approaches to the problem of solving normal equations and matrix inversion demand relatively high computational resources, making notable use of hardware multipliers. Only small-size problems can be efficiently solved in real-time hardware design.

2.5 Complex Division

Complex numbers are very common in the field of communications where they are often used to represent signals or signal-related quantities. The division of two complex numbers has been used diversely in signal processing areas, such as acoustic pulse reflectometry [56], astronomy [57], optics [58], image processing [59] and non-linear RF measurement [60]. It also appears in many signal processing linear algebra problems such as the complex singular value decomposition [61].

The most straightforward method for complex division is to pre-multiply both the divisor $d = d_r + jd_j$ and the dividend $r = r_r + jr_j$ by the complex conjugate of the divisor

$$q = \frac{r}{d} = \frac{r_r + jr_j}{d_r + jd_j} = \frac{r_r d_r + r_j d_j}{{d_r}^2 + {d_j}^2} + j\frac{r_j d_r - r_r d_j}{{d_r}^2 + {d_j}^2} \tag{2.27}$$

where $q = q_r + jq_j$ is the quotient and $j = \sqrt{-1}$. This is the most fundamental method of performing complex division, and refers to as the fundamental equation of complex division [62]. However, this basic equation requires six real multiplications, two real divisions and three real additions. One problem associated with the fundamental equation is that of the dynamic range, i.e., the multiplications in equation (2.27) may generate very large values or very small values. This may cause overflow or underflow, respectively, and errors will be introduced [63] [62].

The Smith algorithm [64] avoids the problems of overflow and underflow by converting multiplications in (2.27) as

$$q = \frac{r}{d} = \frac{r_r + jr_j}{d_r + jd_j} = \begin{cases} \dfrac{r_r + r_j(d_j/d_r)}{d_r + d_j(d_j/d_r)} + j\dfrac{r_j - r_r(d_j/d_r)}{d_r + d_j(d_j/d_r)}, & \text{if } |d_r| \geq |d_j|, \\ \dfrac{r_r(d_r/d_j) + r_j}{d_r(d_r/d_j) + d_j} + j\dfrac{r_j(d_r/d_j) - r_r}{d_r(d_r/d_j) + d_j}, & \text{if } |d_r| < |d_j|. \end{cases} \tag{2.28}$$

Thus, the product ranges of the multiplications in (2.28) are much better defined compared to that of (2.27). Overflow error will only occur if the operands themselves are very close to overflowing. It is seen that the Smith method requires three divisions, three multiplications and three additions. Comparing with the fundamental equation (2.27), Smith method obtains a higher numerical accuracy at the expense of computational load through the increased number of divisions.

In [65, 66], a complex division algorithm using a digit-recursion scheme is proposed. The key idea of the approach is to apply a digit-recurrence division algorithm, where real and imaginary quotient bits are computed in a linear sequence, with prescaled

operands [65, 66]. That is, the proposed algorithm uses a standard residual recurrence with complex variables

$$w^{(k+1)} = w^{(k)} - q_{k+1}d \tag{2.29}$$

where $w^{(k)}$ is the complex partial remainder at the k-th iteration and $w^{(0)} = r$ is the complex dividend, d is the complex divisor, and q_k is the k-th complex quotient digit. The attractiveness of the digit-recursion division scheme is that the recursion is based on a very simple quotient digit selection function; the quotient digits q_k are actually a rounded version of the most significant bits of the partial remainder $w^{(k)}$, which makes the equation (2.29) very simple. The major disadvantage of the digit-recurrence division scheme is that it only works if the dividend (or rather, one component of the dividend) is equal to or very close to 1. Therefore, a rather cumbersome complex prescaling (i.e. magnitude scaling and rotation) operation is required prior to the digit-recurrence division scheme commencing. The complex prescaling operation is performed based on a set of hardware multipliers [67] [62]. The scaling factor is calculated through interpolation from values stored in look-up tables, and these look-up tables can be quite large [67] [62]. In practice, the storage requirement of look-up tables can be reduced by using bipartite tables [67] [62], which significantly reduce the number of values needed to be stored at the expense of a handful of arithmetic operations performed during each look-up table [62].

Alternatively, the complex division problem can be viewed as a problem of finding the solution of a system of linear equations [68]

$$\begin{bmatrix} d_r & -d_j \\ d_j & d_r \end{bmatrix} \begin{bmatrix} q_r \\ q_j \end{bmatrix} = \begin{bmatrix} r_r \\ r_j \end{bmatrix}. \tag{2.30}$$

However, using the algorithms we presented above, both the direct and iterative methods, to solve this simple 2×2 real-valued linear equations is complicated, requiring division and multiplication operations.

Division of complex numbers based on the techniques we presented in this section is very complicated and expensive for real-time hardware implementation. Therefore, complex division has traditionally been a computationally-intensive process and largely implemented in software [65].

2.6 MVDR Adaptive Beamforming

The adaptive beamformer performs spatial filtering, wherein an array of individual sensors is exploited to receive a source signal from the interested direction and to attenuate

signals from other directions and uncorrelated noise [6]. Fig. 2.2 shows a generic adaptive beamforming system that uses a linear array of N individual sensors, where $\mathbf{x}(i)$ are the array outputs (or also referred as snapshot) at time instant i, $\mathbf{w}(i)$ is the array weights, $\boldsymbol{\beta}(i)$ is the steering vector of the interest direction and $y(i) = \mathbf{w}^H(i)\mathbf{x}(i)$ is the beamformer output. The beamforming process is carried out by weighting the array outputs $x(i)$, thereby adjusting their amplitudes and phases such that when added together they form an electronically-steerable beamformer output $y(i)$ [69]. An adaptive algorithm is employed to optimize the array weights automatically based on the array outputs $\mathbf{x}(i)$ and the steering vector $\boldsymbol{\beta}(i)$ of the interested direction. Therefore, by choosing an appropriate adaptive algorithm, the adaptive beamformer has two main benefits: 1) steering capability, whereby the interested source signal is always listened; 2) cancellation of interferences and uncorrelated noise, therefore the obtained signal-to-interference-plus-noise ratio (SINR) is maximized [6].

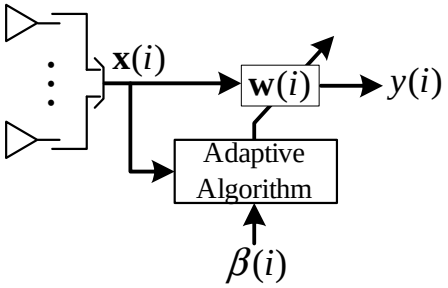


Figure 2.2: A generic adaptive beamforming system

Adaptive beamformer based on the MVDR criterion (or also referred as Capon beamforming) [70] is considered as the optimal beamformer. Refer to Fig. 2.2, the MVDR beamformer selects the array weights $\mathbf{w}(i)$ by minimizing the variance (i.e., average power) of the beamforming output $y(i)$, subject to maintaining unity response in the look direction (the direction of the interested source) [6] [71],

$$\begin{aligned} \underset{\mathbf{w}(i)}{\text{minimize}} \quad & \mathbf{w}^H(i)\mathbf{R}(i)\mathbf{w}(i), \\ \text{subject to} \quad & \mathbf{w}^H(i)\boldsymbol{\beta}(i) = 1, \end{aligned} \tag{2.31}$$

where $\mathbf{R}(i) = \mathrm{E}[\mathbf{x}(i)\mathbf{x}^H(i)]$ is the correlation matrix of the snapshot $\mathbf{x}(i)$. Applying the method of Lagrange to (2.31), we could get the following solution for the weight vector

$$\mathbf{w}(i) = \frac{\mathbf{R}^{-1}(i)\boldsymbol{\beta}(i)}{\boldsymbol{\beta}^H(i)\mathbf{R}^{-1}(i)\boldsymbol{\beta}(i)}. \tag{2.32}$$

Therefore, the MVDR beamformer (2.32) obtains the maximum SINR by keeping the unity response on the desired source signal constant while minimizing the total output noise, including the interference signals and uncorrelated noise [71]. However, the

MVDR beamforming (2.32) requires matrix inversion and is considered too computationally complicated for practical implementation.

Alternatively, the equation (2.32) can be solved by computing

$$\mathbf{h}(i) = \mathbf{R}^{-1}(i)\boldsymbol{\beta}(i), \tag{2.33}$$

which can be represented as the normal equations

$$\mathbf{R}(i)\mathbf{h}(i) = \boldsymbol{\beta}(i). \tag{2.34}$$

Consequently, the optimal weights $\mathbf{w}(i)$ can then simply be computed as

$$\mathbf{w}(i) = \frac{\mathbf{h}(i)}{\boldsymbol{\beta}^H(i)\mathbf{h}(i)}. \tag{2.35}$$

There exist some efficient techniques for solving the normal equations (2.34), such as the QRD-RLS algorithm. However, the QRD using Givens rotations is very complicated for hardware implementation, even though it could exploit parallel processing and pipelining capabilities using systolic array architectures. Some other techniques, such as SGR, CORDIC and LNS, are applied to the QRD to reduce the complexity. As we have shown above, the QRD algorithms based on SGR, CORDIC and LNS are still expensive for hardware designs, requiring a large number of logic resources and hardware multipliers, which makes large size beamformers impractical on all but the largest available FPGA chips.

The performance of the MVDR adaptive beamformer is not robust and is sensitive to the steering vector errors caused by imprecise sensor calibrations [72]. Diagonal loading on the correlation matrix $\mathbf{R}(i)$ is a popular method to improve the performance of the MVDR beamformer [72], i.e.,

$$\mathbf{R}(i) = \mathrm{E}[\mathbf{x}(i)\mathbf{x}^H(i)] + \delta(i)\mathbf{I}, \tag{2.36}$$

where $\delta(i)$ is an extra diagonal loading at time instant i and $\mathbf{I}$ is an $N \times N$ identity matrix. For the classical RLS algorithm, this extra diagonal loading does not allow to use the matrix inversion lemma and increases the complexity to $\mathcal{O}(N^3)$ operations per sample as it requires matrix inversion at each time instant [6]. The leaky RLS adaptive algorithm [73] allows solving the RLS problem with a diagonal loading with complexity of $\mathcal{O}(N^2)$. It is based on using a recursive update of the eigenvalue decomposition of the correlation matrix. However, the eigenvalue decomposition is very complicated for real-time implementation. The QRD-RLS algorithm processes the input vector $\mathbf{x}(i)$ directly avoiding estimation of the correlation matrix to reduce the complexity [6]. We did not find any technique that implements the QRD-RLS algorithm with extra diagonal loading.

We could use the QRD method on the regularized correlation matrix to solve the normal equations discretely at each time instant. However, this discrete QRD scheme can not be expressed recursively like the QRD-RLS to reduce the complexity, which makes it very complicated. Therefore, the regularized MVDR beamforming problem is very complicated and expensive for real-time hardware implementation of large size beamformers.

2.7 FPGA Implementation Procedures

Basically, an FPGA is a large-scale integrated circuit containing programmable logic blocks, programmable interconnects and programmable input-output blocks [74]. The architecture of a Xilinx Virtex II pro chip is shown in Fig. 2.3. The programmable logic blocks can be programmed to duplicate the functionality of basic logic gates such as AND, OR, XOR, NOT or more complex combinatorial functions such as flip-flops, memory elements, decoders or simple mathematical functions [74]. The programmable input-output blocks at the periphery of the devices provide programmable input and output capabilities [74]. By programming the hierarchy of programmable interconnects, the programmable logic blocks and programmable input-output blocks can be interconnected to perform whatever logical functions and input-output connections are required [74]. During the past decade, FPGAs have experienced extensive architecture innovations [75]. Many advanced technologies have been applied to FPGA devices that enable the development of higher density and much more powerful devices [75]. Now most FPGA devices also have block RAMs, hardware multipliers and embedded microprocessors besides traditional logic blocks and interconnects. Therefore FPGA devices become extremely well suited to the high-performance real-time signal processing [75].

Defining the behavior of an FPGA chip can be done using a Hardware Description Language (HDL) such as VHDL and Verilog to describe the functions directly. The hand-written code can be guaranteed as optimal by the designer in the sense that one can be sure what is got as an output [16]. However, the optimality of the design is highly related to the experience of the designer which makes the HDL design method difficult for inexperienced designers. Alternatively, defining the behavior of an FPGA can be done using a schematic based design tool, such as the System Generator [42] we mentioned above. The System Generator provides blocks of pre-defined functions, which can be arranged through a graphical user interface, as shown in Fig. 2.4. Therefore, the System Generator is easy for designers, especially for persons unexperienced with HDL design method. After defining the behavior using either the HDL method or the schematic method, a technology-mapped netlist is generated using an electronic design automation tool [74].

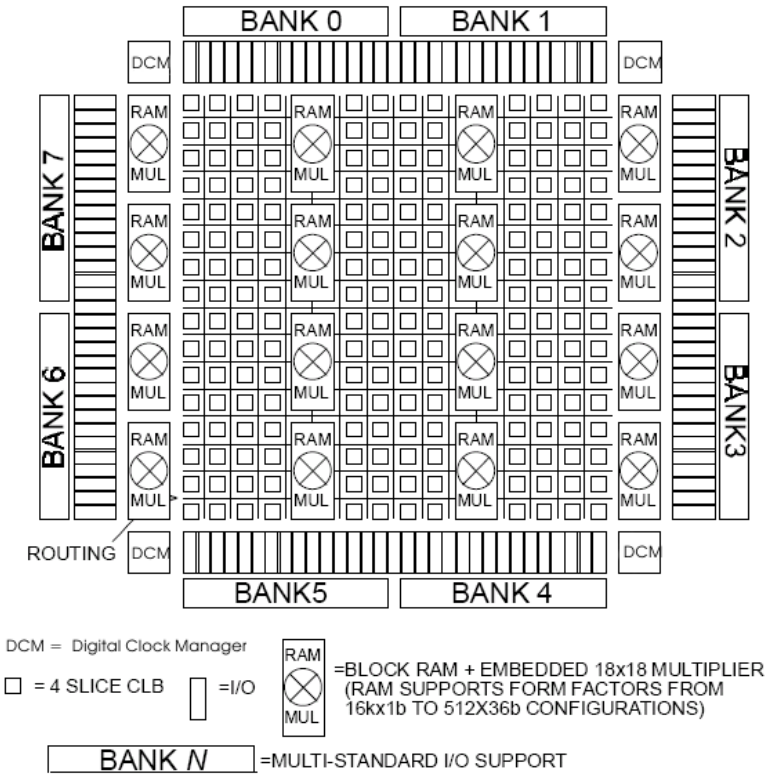


Figure 2.3: Architecture of an FPGA chip (Copied from [76])

The netlist can then be fitted to the actual FPGA architecture using a process called place-and-route, usually performed by the FPGA company's proprietary place-and-route software [74]. The user will validate the map, place and route results via timing analysis, simulation, and other verification methodologies [74]. Once the design and validation process is complete, the binary file can be generated (also using the FPGA company's proprietary software) and downloaded to (re)configure the FPGA device [74].

To simplify the design of complex systems in FPGAs, there exist libraries of predefined complex functions and circuits that have been tested and optimized to speed up the design process. These predefined circuits are commonly called IP cores, such as the CORDIC cores we mentioned above, and are available from FPGA vendors and third-party IP suppliers [74]. The FPGA device vendors also provide related software to support their chips, such as the Xilinx Integrated Software Environment (ISE). With assistance of these software tools and IP cores, FPGA design is simpler now.

In this thesis we choose the Xilinx Virtex-II Pro Development System [77] to implement our designs. This is an evaluation board that features a Xilinx Virtex-II Pro XC2VP30 FPGA [76] (FF896 package, speed grade 7), which in turn features 13696 logic slices, 136 18k-bit block-RAM components, 136 18-bit-by-18-bit embedded mul-

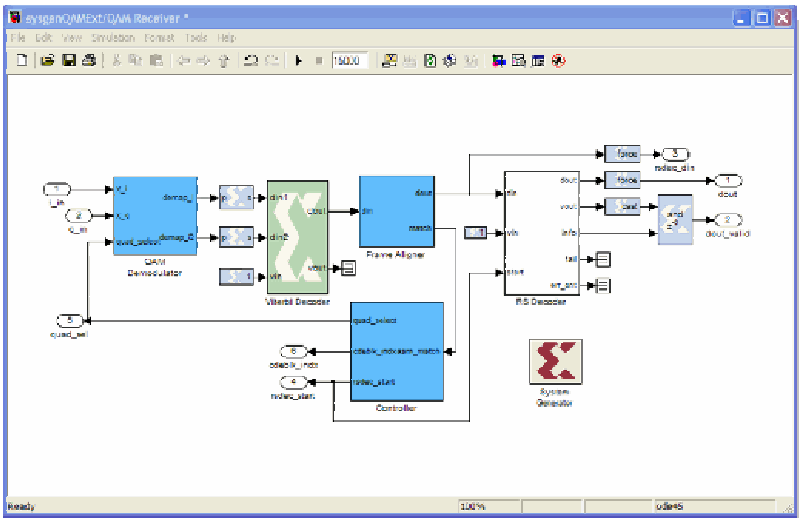


Figure 2.4: Schematic Design using System Generator (Copied from [42])

tipliers etc. The whole device is organized as an array of logic elements and programmable routing resources [75]. Besides the traditional programmable logic blocks, programmable interconnects and programmable input-output blocks, this FPGA device also contains some dedicated resources such as on-chip memory, DCM (Digital Clock Manager) and embedded hardware multipliers. These dedicated resources are quite useful for the high-performance real-time processing.

We use VHDL to describe our designs, and they are synthesized and downloaded to the target platform using the Xilinx ISE 8.1i software package. All our designs presented in this thesis operate synchronously from a single 100 MHz clock signal which is generated by the crystal oscillator circuit on the evaluation board. This 100 MHz clock signal connects to each logic slice through the DCM Logic module and the Global Clock Distribution Network [76]. The DCM here acts as a Delay Lock Loop (DLL) to de-skew and enhance the fan-out on the clock signal [76]. The Global Clock Distribution Network is a dedicated copper-layer which could guarantee that the delay difference of the clock signal arriving at each flip flop could be ignored. Distributing the global clock signal like this can ensure uniform delay between the system clock source and each logic slice in our synchronous designs.

2.8 Conclusions

In this chapter we presented reviews on the problems to be solved in the next chapters. We first introduced well-known algorithms for solving the normal equations and calculating matrix inversion, including both direct and iterative methods. Some reference hardware designs were also presented and compared with each other. However, these efficient algorithms are complicated, requiring division and multiplication operations. Therefore, they demand relatively high hardware resources, making them expensive for real-time hardware design.

The RLS algorithm attracts a lot of interest in adaptive filtering applications due to its fast convergence speed. However, the fast convergence is obtained at the expense of high complexity. The classical RLS algorithm employs the matrix inversion lemma to reduce the complexity to $\mathcal{O}(N^2)$ operations per sample. The widely used robust QRD-RLS algorithm, using QRD to work directly with the incoming data, is also very complicated with $\mathcal{O}(N^2)$ operation per sample. Therefore, hardware design of the RLS algorithm is complicated, requiring large number of logic resources with heavy latency.

The division of two complex numbers is another common problem in the areas of signal processing and communications. Some techniques for solving this problem were analyzed, including the fundamental method, the Smith algorithm, the digit-recursion algorithm and the method of solving a system of linear equations. However, even though the complex division is easy to understand in theory, it is computationally complicated and largely implemented in software.

The MVDR beamforming obtains a high SINR performance and is considered as the optimal beamformer. However, it has high computational complexity, requiring matrix inversion at each time instant. Alternatively, the MVDR beamforming can be achieved by solving normal equations. However, solving the normal equations is also a computationally complex process and expensive for real-time hardware design. To achieve the array beamforming with robust performance, the diagonal loading on the correlation matrix of input snapshots is the popular choice. However, this extra diagonal loading increases the computational complexity and it is difficult for real-time hardware implementation.

FPGA design procedures were also introduced, including overview of the architecture of an FPGA chip and the design flow to develop an FPGA implementation. The FPGA design environment used in this thesis was also introduced.

In the rest of this thesis, we will be solving the problems considered above. A low complexity normal equations solver based on the multiplication-free DCD iterations will be presented in Chapter 3. In Chapter 4, a multiplication-free complex divider is developed based on the idea that the multiplication-free DCD algorithm is used to solve the system of equations in the complex division problem. In Chapter 5, an efficient FPGA implementation of the DCD-based MVDR beamformer is developed with a small area usage and high throughput. In Chapter 6, a low complexity RLS algorithm based on DCD iterations is introduced and implemented in FPGA. It solves the RLS problem with a small number of multiplications, without divisions. The FPGA design of the RLS-DCD algorithm is highly efficient, requiring small chip area and obtains high throughput. Furthermore, the regularized version of the RLS-DCD algorithm is still of low complexity and easy for hardware design.

Chapter 3

Architectures and FPGA Implementations of DCD Algorithm

Contents

3.1 Introduction

The LS approach is widely used in the areas of signal processing and communication. The solution of the LS problem is often based on solving the normal equations. However, solving the normal equations is very complicated. It can be done by using matrix inversion. The complexity of direct matrix inversion is generally regarded as $\mathcal{O}(N^3)$ arithmetic operations [1] where N is the system size, which is not practical for many real-time solutions.

From a numerical point of view, the best approach is to avoid matrix inversion [2–4]. Consequently for real-time solution, techniques that solve system of equations without performing matrix inversion may be preferable. These can be classified into direct and iterative methods. The direct methods, such as Cholesky decomposition, Gaussian elimination, QRD and others, also have complexity of $\mathcal{O}(N^3)$ [1]. Iterative methods, such as the steepest descent method and CG method, provide fast convergence but require $\mathcal{O}(N^2)$ operations per iteration [1]. The coordinate descent (CD) techniques, such as Gauss-Seidel, Jacobi and SOR [1] methods, demonstrate a slower convergence but require only $\mathcal{O}(N)$ operations per iteration. The computational load of these techniques depends on the number of iterations executed (and hence accuracy obtained). These iterative methods require multiplication and division operations, which are complex for implementation, especially in hardware, i.e., they require significant chip area and high power consumption. Moreover, divisions can lead to numerical instability.

The QR decomposition is most often used for implementing matrix operations in hardware [4, 11, 36, 78–81]. However, this technique requires multiplications and divisions, and some implementations [15] also require square-root operations; all of these operations are difficult for implementation in hardware. Although CORDIC is an efficient technique for implementing Givens rotations for QRD [4, 36, 38, 80], CORDIC blocks require substantial chip resources and large number of clock cycles. In total, implementation of QRD by using systolic arrays consumes large silicon area [23] and requires a large number of clock cycles. The use of the BAMI allows significant reduction in the number of clock cycles [23] for solving small size matrices; however, the occupied chip area is high. Most of the implementations use extra computing resources such as Xilinx DSP48 block [44], Altera NIOS processor [36], or custom logic. However, even with such extra computational engines, only small problems ($N \leq 9$) can be practically implemented when using these traditional methods.

The DCD algorithm [5] is an iterative algorithm for solving normal equations. It is based on the CD iterations with power of two variable step size. It does not need multiplications and division. For each iteration, it only requires $\mathcal{O}(N)$ or $\mathcal{O}(1)$ additions. Thus, it is well suited to hardware implementation.

In this chapter, we present several architectures and FPGA implementations of two variants of the DCD algorithm, known as the cyclic and leading DCD algorithm, and show how these implementations are comparable to each other and which of the variants are more suitable for different applications. Specifically, we propose partly-parallel implementations of the cyclic and leading DCD algorithms, thus allowing a trade-off in chip area and computation time; we call these implementations group-2 and group-4 imple-

mentations. In particular, the group-2 implementations can be efficiently used for solving complex-valued systems of equations. We present and compare serial, partly-parallel and parallel implementations of the cyclic and leading DCD algorithms. This includes a comparison of the complexity and the convergence speed of the implementations. We present numerical results and discuss preferable applications of these implementations. Finally conclusions are given.

The rest of this chapter is organized as follows. Coordinate descent optimization is introduced in Section 3.2, where we also introduce cyclic and leading DCD algorithms for real-valued and complex-valued systems. Sections 3.3 to 3.6 present architectures and FPGA implementations of the DCD algorithms. Numerical results are given in Section 3.7. Finally, Section 3.8 gives conclusions.

3.2 Coordinate Descent Optimization and DCD Algorithm

Many techniques can be used to solve the normal system of equations

$$\mathbf{R}\mathbf{h} = \boldsymbol{\beta} \tag{3.1}$$

where $\mathbf{R}$ is an $N \times N$ symmetric positive definite matrix and both $\mathbf{h}$ and $\boldsymbol{\beta}$ are $N \times 1$ vectors. The matrix $\mathbf{R}$ and vector $\boldsymbol{\beta}$ are known, whereas the vector $\mathbf{h}$ should be estimated. As discussed in Chapter 2, solving the normal equations (3.1) is equivalent to minimizing the quadratic function

$$f(\mathbf{h}) = \frac{1}{2}\mathbf{h}^T\mathbf{R}\mathbf{h} - \mathbf{h}^T\boldsymbol{\beta}. \tag{3.2}$$

Line search methods [1] are iterative methods for minimizing the function $f(\mathbf{h})$. In a line search method, at each iteration k, the solution $\mathbf{h}^{(k)}$ is updated as $\mathbf{h}^{(k)} = \mathbf{h}^{(k-1)} + \alpha^{(k)}\mathbf{d}^{(k)}$ in a direction $\mathbf{d}^{(k)}$ that is chosen to be non-orthogonal to the residual vector $\mathbf{r}^{(k-1)} = \boldsymbol{\beta} - \mathbf{R}\mathbf{h}^{(k-1)}$, i.e., $(\mathbf{d}^{(k)})^T\mathbf{r}^{(k-1)} \neq 0$. The step size $\alpha^{(k)}$ minimizing the function $f\left(\mathbf{h}^{(k)} + \alpha^{(k)}\mathbf{d}^{(k)}\right)$ is $\alpha^{(k)} = (\mathbf{d}^{(k)})^T\mathbf{r}^{(k-1)}/(\mathbf{d}^{(k)})^T\mathbf{R}\mathbf{d}^{(k)}$; this step size corresponds to the exact line search method [82, 83]. A general description of the exact line search method is given in Table 3.1, where N_u denotes the number of iterations (or the number of updates of the solution vector).

An efficient variant of the line search method is the CG algorithm [1] shown in Table 3.2. At the first iteration, $k = 1$, the direction vector is the residual vector: $\mathbf{d} = \mathbf{r}$.

Table 3.1: Exact line search method

Step	Equation
	Initialization: $\mathbf{h} = \mathbf{0}$, $\mathbf{r} = \boldsymbol{\beta}$
	for $k = 1, \ldots, N_u$
1	Choose $\mathbf{d}$ such that $\mathbf{d}^T\mathbf{r} \neq \mathbf{0}$
2	$\mathbf{v} = \mathbf{R}\mathbf{d}$
3	$\alpha = \mathbf{d}^T\mathbf{r}/\mathbf{d}^T\mathbf{v}$
4	$\mathbf{h} = \mathbf{h} + \alpha\mathbf{d}$
5	$\mathbf{r} = \mathbf{r} - \alpha\mathbf{v}$

Table 3.2: Conjugate gradient algorithm

Step	Equation	$\times$	$+$	$\div$
	Initialization: $\mathbf{h} = \mathbf{0}$, $\mathbf{r} = \boldsymbol{\beta}$, $\rho^{(0)} = \mathbf{r}^T\mathbf{r}$, $\mathbf{d} = \mathbf{r}$ for $k = 1, \ldots, N_u$			
1	if $k > 1$, $\mathbf{d} = \mathbf{r} + (\rho^{(k-1)}/\rho^{(k-2)})\mathbf{d}$	N	N	1
2	$\mathbf{v} = \mathbf{R}\mathbf{d}$	N^2	$N^2 - N$	
3	$\alpha = \rho^{(k-1)}/\mathbf{d}^T\mathbf{v}$	N	$N - 1$	1
4	$\mathbf{h} = \mathbf{h} + \alpha\mathbf{d}$	N	N	
5	$\mathbf{r} = \mathbf{r} - \alpha\mathbf{v}$	N	N	
6	$\rho^{(k)} = \mathbf{r}^T\mathbf{r}$	N	$N - 1$	
Total:	$(N^2 + 5N)N_u$ mults, $(N^2 + 4N - 2)N_u$ adds and N_u divs			

At other iterations, $k > 1$, the direction $\mathbf{d}$ is updated to guarantee $\mathbf{R}$-conjugacy of the direction vectors. Due to its fast convergence, the CG method has already been used for solving normal equations, e.g., in adaptive filtering [27]. Although, the CG algorithm shows fast convergence, its complexity is too high for many applications; in general, it is $\mathcal{O}(N^2)$ operations per update. The algorithm also requires divisions at steps 1 and 3.

The CD algorithm chooses the directions as Euclidean coordinates, i.e., $\mathbf{d} = \mathbf{e}_n$, where all elements of the vector $\mathbf{e}_n$ are zeros, except the n-th element which is one. The iterations are significantly simplified. In this case, for the exact line search, $\mathbf{v} = \mathbf{R}\mathbf{d} = \mathbf{R}_{:,n}$ is the n-th column of the matrix $\mathbf{R}$. Thus, the most complicated step of the line search method (step 2 in Table 3.1), requiring the matrix-vector multiplication of complexity $\mathcal{O}(N^2)$, is eliminated. The other steps are also simplified: $\mathbf{d}^T\mathbf{r} = r_n$, $\mathbf{d}^T\mathbf{v} = R_{n,n}$, $\alpha = r_n/R_{n,n}$, and $h_n = h_n + \alpha$. If the directions are chosen in a cyclic order, $n = 1, \ldots, N$, we arrive at Gauss-Seidel iterations [1]. The Gauss-Seidel method is used in many signal processing applications, including adaptive filtering [31], multiuser detection [84], and others. The CD algorithm with cyclic passes through N elements of $\mathbf{h}$ is presented in Table 3.3. One update in this cyclic CD algorithm requires only N multiplications, $N + 1$ additions, and one division.

However, the cyclic order of the directions is not efficient when solving a system of

Table 3.3: Cyclic CD algorithm

Step	Equation	$\times$	$+$	$\div$
	Initialization: $\mathbf{h}=\mathbf{0}$, $\mathbf{r}=\boldsymbol{\beta}$, $k=0$			
	for $n=1,\ldots,N$			
1	$\alpha = r_n/R_{n,n}$			1
2	$h_n = h_n + \alpha$		1	
3	$\mathbf{r} = \mathbf{r} - \alpha\mathbf{R}_{:,n}$	N	N	
4	$k = k+1$			
5	if $k = N_u$, algorithm stops			
Total:	NN_u mults, $(N+1)N_u$ adds and N_u divs			

Table 3.4: Leading CD algorithm

Step	Equation	$\times$	$+$	$\div$
	Initialization: $\mathbf{h}=\mathbf{0}$, $\mathbf{r}=\boldsymbol{\beta}$			
	for $k=1,\ldots,N_u$			
1	$n = \arg\max_{p=1,\ldots,N}\{\lvert r_p\rvert\}$		$N-1$	
2	$\alpha = r_n/R_{n,n}$			1
3	$h_n = h_n + \alpha$		1	
4	$\mathbf{r} = \mathbf{r} - \alpha\mathbf{R}_{:,n}$	N	N	
Total:	NN_u mults, $2NN_u$ adds and N_u divs			

equations which only needs a small number of updates, e.g., in adaptive filtering [31, 85]. A more efficient method for selecting the (leading) index n is therefore important to speed up convergence. The CD algorithm chooses the leading index n according to

$$n = \arg\max_{p=1,\ldots,N}\{|r_p|\}. \tag{3.3}$$

This CD algorithm is presented in Table 3.4. Each update in the algorithm requires N multiplications, $2N$ additions, and one division.

3.2.1 Real-Valued Cyclic DCD Algorithm

An exact line search method provides the fastest descent for a particular iteration. Inexact line search methods, though not providing the maximum decrement of $f(\mathbf{h})$ for a particular iteration, can improve the convergence speed in a sequence of iterations [82, 86]. The DCD algorithm is an inexact line search method. In the RLS adaptive filtering, it can provide faster convergence than the CD algorithm (to be shown in Chapter 6).

The DCD algorithm, presented in Table 3.5, updates the solution in directions of Euclidean coordinates in the cyclic order $n = 1, 2, \ldots, N$ which is similar to the cyclic CD algorithm. Thus, we refer to this DCD algorithm as the *cyclic* DCD algorithm. The

Table 3.5: Real-valued cyclic DCD algorithm

Step	Equation	+
	Initialization: $\mathbf{h} = \mathbf{0}$, $\mathbf{r} = \boldsymbol{\beta}$, $\alpha = H$, $k = 0$	
	for $m = 1, \ldots, M_b$	
1	$\alpha = \alpha/2$	
2	Flag = 0	
	for $n = 1, \ldots, N$	
3	if $\lvert r_n \rvert > (\alpha/2) R_{n,n}$	1
4	$h_n = h_n + \text{sign}(r_n)\alpha$	1
5	$\mathbf{r} = \mathbf{r} - \text{sign}(r_n)\alpha \mathbf{R}_{:,n}$	N
6	$k = k + 1$, Flag = 1	
7	if $k = N_u$, algorithm stops	
8	if Flag = 1, repeat from step 2	
Total:	$\leq N(2N_u + M_b - 1) + N_u$ adds	

selection of the step-size α is as follows. For some iterations (*unsuccessful iterations*), the step-size is zero; for the other iterations (*successful iterations*), it is chosen as a power of two and, in general, it reduces with the number of iterations. The parameter H defines the initial value of the step-size. The parameter M_b indicates how many times the step-size may be reduced; M_b can be considered as the number of bits used for a fixed-point representation of elements of the solution vector. For every change of the step size, the algorithm repeats iterations until all elements of the residual vector $\mathbf{r}$ become so small that the condition at step 3 is not met for all n, i.e., the last N iterations are unsuccessful. In this case, the step-size is reduced at step 1. The computational load posed by this algorithm is mainly due to the successful iterations, and to limit the complexity with an uncertain error in the solution, a limit on the number of successful iterations (updates) N_u is predefined. The selection of the step-size results in existence of unsuccessful iterations which require the only comparison at step 3 and thus reduces the complexity compared to the CD algorithm. However, it is even more important that this also results in no explicit multiplication and no explicit division as all the multiplications and divisions can be replaced by simple bit shifts.

The complexity of this cyclic DCD algorithm can be considered as a random number with an upper bound corresponding to a worst-case scenario as follows. For the m-th bit, $m = 1, \ldots, M_b - 1$, within one pass ($n = 1, \ldots, N$) there is one "successful" iteration and then, in another pass, N "unsuccessful" iterations; this will require $(3N+1)(M_b-1)$ additions. For the last (least significant) bit, $m = M_b$, there are $(N_u - M_b + 1)$ passes each with one "successful" iteration; this will require $(2N + 1)(N_u - M_b + 1)$ additions. Thus, the worst-case complexity is $N(2N_u + M_b - 1) + N_u$ additions. Notice that the average complexity will be lower.

Table 3.6: Real-valued leading DCD algorithm

Step	Equation	+
	Initialization: $\mathbf{h}=\mathbf{0}$, $\mathbf{r}=\boldsymbol{\beta}$, $\alpha=H$, $m=1$	
	for $k=1,\ldots,N_u$	
1	$n=\arg\max_{p=1,\ldots,N}\{\lvert r_p\rvert\}$, go to step 4	$N-1$
2	$m=m+1$, $\alpha=\alpha/2$	
3	if $m>M_b$, algorithm stops	
4	if $\lvert r_n\rvert\leq(\alpha/2)R_{n,n}$, then go to step 2	1
5	$h_n=h_n+\text{sign}(r_n)\alpha$	1
6	$\mathbf{r}=\mathbf{r}-\text{sign}(r_n)\alpha\mathbf{R}_{:,n}$	N
Total:	$\leq(2N+1)N_u+M_b$ adds	

3.2.2 Real-Valued Leading DCD Algorithm

For the cyclic DCD algorithm in Table 3.5, if $N_u >> M_b$, the complexity is approximately upper bounded by $2NN_u$ additions. However, if N_u is small $N_u < M_b/2$, the term NM_b will dominate in the complexity. The leading DCD algorithm shown in Table 3.6 can eliminate this term. Similar to the leading CD algorithm, this variant of the DCD algorithm identifies a leading (n-th) element in $\mathbf{h}$ to be updated corresponding to an element of the residual vector $\mathbf{r}$ which has the largest absolute value. This differs from the cyclic DCD algorithm, where the elements to be updated are chosen in a cyclic order. With N_u updates, the complexity of the leading DCD algorithm is upper limited by $(2N+1)N_u+M_b$ additions. This corresponds to a worst-case scenario when the algorithm makes use of all N_u updates and the condition at step 3 is never satisfied.

3.2.3 Complex-Valued Cyclic DCD Algorithm

The DCD algorithms in Table 3.5 and Table 3.6 can only solve real-valued equations. To use these techniques for solving complex-valued systems, we need to form an equivalent real-valued system according to the following rule. Let $\mathbf{R}=(\mathbf{A}_1+j\mathbf{A}_2)$, $\mathbf{h}=(\mathbf{b}_1+j\mathbf{b}_2)$ and $\boldsymbol{\beta}=(\mathbf{c}_1+j\mathbf{c}_2)$, where $\mathbf{A}_1$ and $\mathbf{A}_2$ are $N/2\times N/2$ real-valued matrices and $\mathbf{b}_1$, $\mathbf{b}_2$, $\mathbf{c}_1$ and $\mathbf{c}_2$ are $N/2\times 1$ real-valued vectors. The $N/2\times N/2$ complex-valued system (3.1) is equivalent to the $N\times N$ real-valued system

$$\mathbf{A}\mathbf{b}=\mathbf{c} \tag{3.4}$$

where

$$\mathbf{A}=\begin{bmatrix}\mathbf{A}_1 & -\mathbf{A}_2\\ \mathbf{A}_2 & \mathbf{A}_1\end{bmatrix},\quad \mathbf{b}=\begin{bmatrix}\mathbf{b}_1\\ \mathbf{b}_2\end{bmatrix},\quad \mathbf{c}=\begin{bmatrix}\mathbf{c}_1\\ \mathbf{c}_2\end{bmatrix}. \tag{3.5}$$

Table 3.7: Complex-valued cyclic DCD algorithm ($N/2 \times N/2$ system)

Step	**Equation**	$+$
	Initialization: $\mathbf{h} = \mathbf{0}$, $\mathbf{r} = \boldsymbol{\beta}$, $\alpha = H$, $k = 0$, $s = 1$	
	for $m = 1, \ldots, M_b$	
1	$\alpha = \alpha/2$	
2	Flag = 0	
	for $n = 1, \ldots, N/2$	
3	if $s = 1$ then $r_{tmp} = \Re(r_n)$, else $r_{tmp} = \Im(r_n)$ if $\lvert r_{tmp} \rvert > (\alpha/2) R_{n,n}$	1
4	$h_n = h_n + \text{sign}(r_{tmp}) s\alpha$	1
5	$\mathbf{r} = \mathbf{r} - \text{sign}(r_{tmp}) s\alpha \mathbf{R}_{:,n}$	N
6	$k = k + 1$, Flag = 1	
7	if $k = N_u$, algorithm stops	
8	if $s = 1$, then $s = j$, goto step 3; else $s = 1$	
9	if Flag = 1, then goto step 2	
Total:	$\leq N(2N_u + M_b - 1) + N_u$ adds	

However, even though this representation does not contain any arithmetic operations, from a hardware implementation point of view it introduces extra processing overhead, and the memory space required for the matrix $\mathbf{A}$ is doubled compared to the complex-valued matrix $\mathbf{R}$.

Now we consider solving a complex-valued system of equations by integrating the real-valued DCD algorithms in Table 3.5 and Table 3.6 with the pre-processing described in equation (3.5). For a real-valued system, the DCD algorithm tests possible updates of the solution vector in two coordinate directions: negative real and positive real. For a complex-valued system, the DCD algorithm tests possible updates of the solution vector in four coordinate directions: negative real, negative imaginary, positive real and positive imaginary.

The complex-valued cyclic DCD algorithm is shown in Table 3.7. Notice that the matrix $\mathbf{R}$ and vectors $\boldsymbol{\beta}$, $\mathbf{r}$ and $\mathbf{h}$ here are all complex-valued. The real and imaginary components of each element of the residual vector $\mathbf{r}$ are processed sequentially. A variable s indicates which component is being processed - real ($s = 1$) or imaginary ($s = j = \sqrt{-1}$). The remaining operations are the same as in the real-valued cyclic DCD algorithm in Table 3.5. The complexity of the complex-valued cyclic DCD algorithm is upper limited by $N(2N_u + M_b - 1) + N_u$ real-valued additions for an $N/2 \times N/2$ complex-valued system.

Table 3.8: Complex-valued leading DCD algorithm ($N/2 \times N/2$ system)

Step	Equation	+
	Initialization: $\mathbf{h} = \mathbf{0}$, $\mathbf{r} = \boldsymbol{\beta}$, $\alpha = H$, $m = 0$	
	for $k = 1, \ldots, N_u$	
1	$[n, s] = \arg\max_{p=1,\ldots,N/2}\{\lvert\Re(r_p)\rvert, \lvert\Im(r_p)\rvert\}$ go to step 4	$N-1$
2	$m = m + 1$, $\alpha = \alpha/2$	
3	if $m > M_b$, algorithm stops	
4	if $s = 1$, then $r_{tmp} = \Re(r_n)$, else $r_{tmp} = \Im(r_n)$ if $\lvert r_{tmp}\rvert \leq (\alpha/2)R_{n,n}$, then go to step 2	1
5	$h_n = h_n + \text{sign}(r_{tmp})s\alpha$	1
6	$\mathbf{r} = \mathbf{r} - \text{sign}(r_{tmp})s\alpha\mathbf{R}_{:,n}$	N
Total:	$\leq (2N + 1)N_u + M_b$ adds	

3.2.4 Complex-Valued Leading DCD Algorithm

The complex-valued leading DCD algorithm is presented in Table 3.8. Similar to the real-valued leading DCD algorithm in Table 3.6, it finds a leading element in the solution vector $\mathbf{h}$ to be updated. This corresponds to a component in the residual vector $\mathbf{r}$, which has the maximum absolute value. Two variables n and s are used to locate this maximum. The other operations are the same as in the real-valued leading DCD algorithm in Table 3.6. The complexity of this algorithm is upper limited by $(2N + 1)N_u + M_b$ real-valued additions for an $N/2 \times N/2$ complex-valued system.

3.3 Real-Valued Serial Architecture DCD Algorithm

We have developed and implemented FPGA cores for the real-valued DCD algorithms in Table 3.5 and Table 3.6. The implementation is based on using a Xilinx Virtex-II Pro Development System [77] with an XC2VP30 (FF896 package, speed grade 7) FPGA [76]. VHDL is used to describe the implementation, which is synthesized and downloaded to the target platform using the Xilinx ISE 8.1i software package. The whole implementation operates from a single 100 MHz clock and we make use of the FPGA Digital Clock Manager and Global Clock Distribution Network [76] to ensure a uniform delay between the system clock source and each logic slice. Our implementation uses the 16-bit Q15 number format [87] to represent elements of the matrix $\mathbf{R}$; they are limited to the range $[-1, 1)$. To avoid overflow errors, elements of the vectors $\boldsymbol{\beta}$, $\mathbf{r}$ and $\mathbf{h}$ are all stored using a 32-bit fixed-point Q15 format; they are limited to the range $[-2^{16}, 2^{16})$. The matrix $\mathbf{R}$ and vectors $\boldsymbol{\beta}$ and $\mathbf{h}$ are stored in $\mathbf{R}$ RAM, $\boldsymbol{\beta}$ RAM and $\mathbf{h}$ RAM, respectively. During

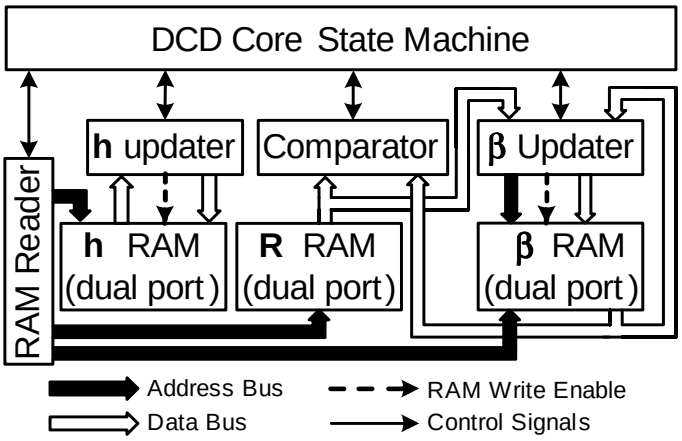


Figure 3.1: Block-diagram of the processor for the cyclic DCD algorithm

the computation, we treat the vector stored in $\boldsymbol{\beta}$ RAM as the residual vector $\mathbf{r}$ since $\mathbf{r}$ is initialized as $\boldsymbol{\beta}$. These three RAM components are dual-port (ports A and B) with each port having their own address, data-in and data-out connections, thus allowing the RAM to be accessed through both ports simultaneously. The data width of these RAMs can be configured arbitrarily from 1 to 256 bits [76]. For various DCD implementations, the configuration of the RAMs differs; this is described below in detail.

3.3.1 Implementation of Real-Valued Cyclic DCD Algorithm

The architecture of the cyclic DCD algorithm is shown in Fig. 3.1. Besides the three dual-port block RAM components, the DCD processor contains five modules: 1) DCD Core State Machine; 2) RAM Reader; 3) Comparator; 4) $\boldsymbol{\beta}$ Updater; and 5) $\mathbf{h}$ Updater. The data widths of the $\mathbf{R}$, $\boldsymbol{\beta}$ and $\mathbf{h}$ RAM are configured to be 16, 32 and 32 bits, respectively. Thus, each of them can provide one element of the matrix $\mathbf{R}$ or vectors $\boldsymbol{\beta}$ and $\mathbf{h}$ at each read or write operation. The RAM Reader is used for addressing the $\boldsymbol{\beta}$ RAM and $\mathbf{R}$ RAM for reading data, and the $\mathbf{h}$ RAM for both reading and writing data. The Comparator decides whether the iteration is successful. The $\mathbf{h}$ Updater and $\boldsymbol{\beta}$ Updater update elements of $\mathbf{h}$ and $\boldsymbol{\beta}$ (the residual vector $\mathbf{r}$), respectively. These modules work in a fully pipelined manner under the control of the DCD Core State Machine.

The cyclic DCD algorithm in Table 3.5 has been optimized for FPGA implementation as described in Table 3.9 and Fig.3.2. We simplify the implementation by exploiting the fact that the step size α only changes when a new m-th bit is considered. We remove α from steps 3 and 5 of Table 3.5 and arrange that the $\boldsymbol{\beta}$ RAM stores a version of the vector $\mathbf{r}$ that is incrementally scaled to compensate for α. The scaling of $\mathbf{r}$, controlled by

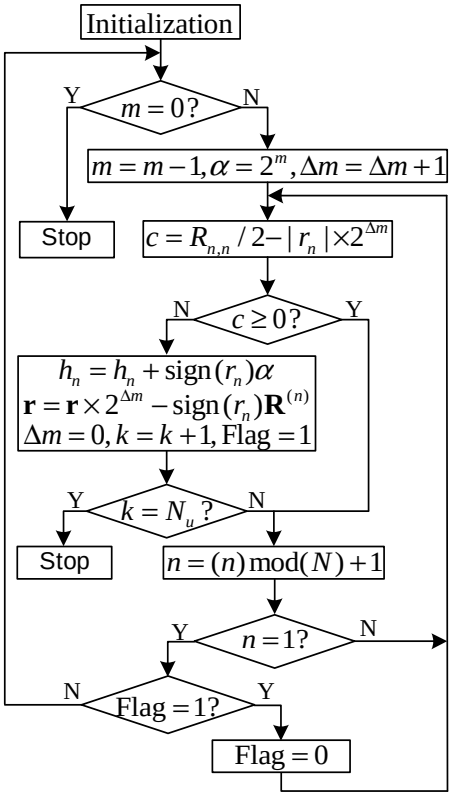


Figure 3.2: Flow-chart of the cyclic DCD algorithm

a parameter Δm, is performed in conjunction with the first successful update of each new m-th bit. The division-by-two of $R_{n,n}$ in state 2 is hardwired. The following paragraphs explain the operations of the optimized algorithm.

During the initialization (state 0), the DCD Core State Machine initializes control signals as shown in Table 3.9.

In state 1, the DCD Core State Machine updates the bit counter m, step size α, and the prescaling counter Δm. The step size α is chosen to be equal to $\alpha = 2^m$. If the current value of m is equal to 0 (i.e., $\alpha = 1$), then the least significant bits of the solution have been updated and the DCD processor halts and indicates that it has finished solving the system of equations. If m is non-zero, the DCD processor proceeds to state 2 and then state 3 to perform the comparison.

Two cycles (due to the RAM read latency) before each comparison, the RAM Reader asserts the address of r_n in the $\boldsymbol{\beta}$ RAM and $R_{n,n}$ in the $\mathbf{R}$ RAM. Then, when the comparison (state 2) commences, the Comparator reads r_n from the $\boldsymbol{\beta}$ RAM and $R_{n,n}$ from the $\mathbf{R}$ RAM. If the vector $\mathbf{r}$ has not been prescaled for a new m-th bit, the Comparator scales r_n. It then performs the comparison and passes on the sign bit of the result c to the DCD

Table 3.9: Real-valued cyclic DCD algorithm for serial FPGA implementation

State	Operation	Cycles
0	Initialization: $\mathbf{h} = \mathbf{0}$, $\mathbf{r} = \boldsymbol{\beta}$, Flag=0, $m = M_b$, $k = 0$, $\Delta m = 0$, $n = 1$	
1	if $m = 0$, algorithm stops else, $m = m - 1$, $\alpha = 2^m$, $\Delta m = \Delta m + 1$	1
2	$c = R_{n,n}/2 - \lvert r_n \rvert \times 2^{\Delta m}$	1
3	if $c \geq 0$, goto state 5	1
4	$h_n = h_n + \text{sign}(r_n)\alpha$ $\mathbf{r} = \mathbf{r} \times 2^{\Delta m} - \text{sign}(r_n)\mathbf{R}_{:,n}$ $\Delta m = 0$, $k = k + 1$, Flag = 1 if $k = N_u$, algorithm stops	N
5	$n = (n)\text{mod}(N) + 1$ if $n = 1$ and Flag = 1, then Flag = 0, goto state 2 elseif $n = 1$ and Flag = 0, then goto state 1 else, goto state 2	1
Total:	$\leq 4NN_u + 3N(M_b - 1) + M_b$ cycles	

Core State Machine. This comparison executes in a single cycle.

In state 3, the DCD Core State Machine examines the comparison result to decide whether this iteration is successful, in which case the algorithm proceeds to state 4 to update the solution element h_n and the residual vector $\mathbf{r}$; otherwise it proceeds to state 5 without any update.

In state 4, the $\mathbf{h}$ Updater and $\boldsymbol{\beta}$ Updater perform updates on h_n and $\mathbf{r}$, respectively. The RAM Reader sequentially asserts addresses of elements of the column $\mathbf{R}_{:,n}$ and the vector $\mathbf{r}$ in the $\mathbf{R}$ RAM and the $\boldsymbol{\beta}$ RAM, respectively. The $\boldsymbol{\beta}$ Updater reads elements of $\mathbf{r}$ from port B of the $\boldsymbol{\beta}$ RAM, updates them, and writes the result back to the $\boldsymbol{\beta}$ RAM through port A. The $\mathbf{h}$ Updater reads the element h_n from the $\mathbf{h}$ RAM, updates it, and writes the updated h_n back to the $\mathbf{h}$ RAM. Then, the DCD Core State Machine sets the prescaling counter Δm to 0 and the variable 'Flag' to 1, indicating at least one successful iteration. The successful iteration counter k is also updated; if k is equal to the predefined limit N_u, the DCD processor stops; otherwise, it proceeds to state 5.

In state 5, the counter n is updated to indicate the next element of $\mathbf{h}$ to be analyzed. The DCD Core State Machine then tests n and 'Flag' to decide how the algorithm should loop.

The number of clock cycles required for each state is shown in Table 3.9. Due to pipelining, state 4 requires only N cycles for updating all elements of the vector $\mathbf{r}$ and

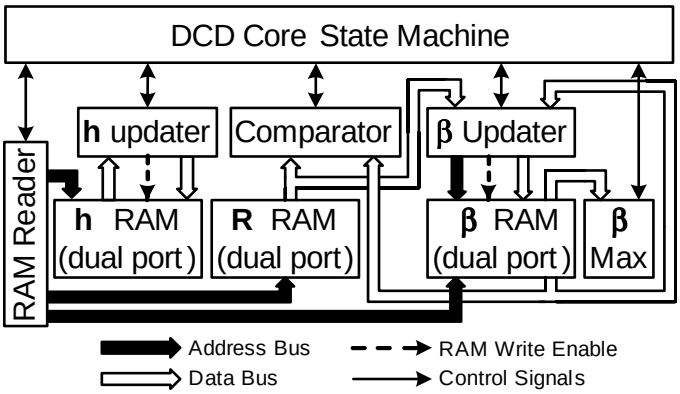


Figure 3.3: Block-diagram of the processor for the leading DCD algorithm

the element h_n. All other states require only a single cycle each. The total number of cycles required for solving a system of equations varies depending on the system size, the solution sparseness, the condition number of the system matrix, as well as the algorithm parameters N_u and M_b. The number of cycles can be considered as a random number with an upper bound corresponding to the worst-case complexity of the cyclic DCD algorithm in Table 3.5. For given N, N_u and M_b, the maximum number of cycles is $4NN_u+3N(M_b-1)+M_b$, or for high N_u we have approximately $4NN_u$. This number of clock cycles corresponds to an unlikely situation when, in every pass through the system, only one successful iteration happens. In a typical situation, there are many successful iterations in every pass, and the average number of clock cycles will be close to NN_u.

3.3.2 Implementation of Real-Valued Leading DCD Algorithm

The hardware architecture of the leading DCD algorithm is shown in Fig. 3.3. Besides the modules required for the cyclic DCD architecture shown in Fig. 3.1, there is a β Max module that is used to find the maximum absolute value in the residual vector $\mathbf{r}$. The function of other modules are similar to those in the serial cyclic DCD algorithm. The leading DCD algorithm has been optimized for FPGA implementation and this is shown in Table 3.10 and Fig.3.4.

Here, state 1 is used to find the index n of the leading element for the first iteration; it executes only once. In this state, the β Updater asserts sequential addresses of elements of vector $\mathbf{r}$ to the β RAM, the β Max module reads the elements of $\mathbf{r}$ and finds the element possessing the maximum absolute value. The index n of this element is fed to the DCD Core State Machine.

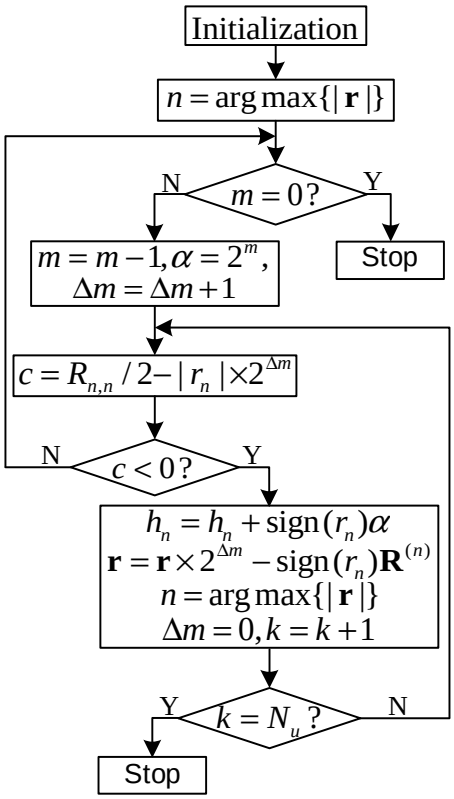


Figure 3.4: Flow-chart of the leading DCD algorithm

Table 3.10: Real-valued leading DCD algorithm for serial FPGA implementation

State	Operation	Cycles
0	Initialization: $\mathbf{h} = \mathbf{0}$, $\mathbf{r} = \boldsymbol{\beta}$, $m = M_b$, $k = 0$, $\Delta m = 0$	
1	$n = \arg\max_{p=1,\ldots,N}\{\lvert r_p\rvert\}$	$N+3$
2	if $m = 0$, algorithm stops else, $m = m-1$, $\alpha = 2^m$, $\Delta m = \Delta m + 1$	1
3	$c = R_{n,n}/2 - \lvert r_n\rvert \times 2^{\Delta m}$	2
4	if $c < 0$, goto state 5 else, goto state 2	1
5	$h_n = h_n + \text{sign}(r_n)\alpha$ $\mathbf{r} = \mathbf{r} \times 2^{\Delta m} - \text{sign}(r_n)\mathbf{R}_{:,n}$ $n = \arg\max_{p=1,\ldots,N}\{\lvert r_p\rvert\}$ $\Delta m = 0$, $k = k+1$ if $k = N_u$, algorithm stops; else, goto state 3	$N+3$
Total:	$\leq (N+6)N_u + N + 4M_b$ cycles	

The operations in states 2 to 5 are similar to the operations in the cyclic DCD algorithm (states 1 to 5) in Table 3.9. The only differences are due to the inclusion of the $\boldsymbol{\beta}$ Max module which works in parallel with the $\boldsymbol{\beta}$ Updater. The $\boldsymbol{\beta}$ RAM is configured in *Transparent Mode*, meaning that input data is simultaneously written into the memory and placed on the output data port [76]. When the $\boldsymbol{\beta}$ Updater writes elements of $\mathbf{r}$ into the $\boldsymbol{\beta}$ RAM, these elements are also available to the $\boldsymbol{\beta}$ Max module through the output data port with one cycle latency. The $\boldsymbol{\beta}$ Max module reads these elements, seeks the element with the maximum absolute value, and provides its index n to the DCD Core State Machine.

The number of clock cycles required by each state of this design are listed in Table 3.10. Execution of state 1 costs $N + 3$ cycles, and it is only executed once during the whole algorithm. Indeed, in some applications, these $N + 3$ cycles can be removed, as the $\boldsymbol{\beta}$ Max module could be designed to work simultaneously with the hardware used to generate the vector $\boldsymbol{\beta}$. The comparison in state 3 costs two cycles, one for the RAM read latency and one for the arithmetic comparison. As this design is pipelined, only $N + 3$ cycles are needed in state 5 for updating h_n and $\mathbf{r}$ and finding a new leading element. States 2 and 4 each requires one cycle. For a given N_u, the maximum number of cycles is $(N + 6)N_u + N + 4M_b$, which corresponds to the worst case in which the condition $m = 0$ in state 2 is never satisfied.

3.3.3 FPGA Resources for Real-Valued Serial Implementations

The FPGA resources required for the serial implementations of the real-valued cyclic and leading DCD algorithms for the cases $N = 16$ and $N = 64$ with $M_b = 15$ are presented in Table 3.11. The whole implementation requires at most 3.6% of FPGA slices available on the Xilinx Virtex-II Pro chip. The overhead posed by the increase of the system size is small – the increase in slice count can be accounted for by the increase in the address bus-widths and the address counter-widths. The area usage of the leading DCD algorithm is larger than that of the cyclic DCD algorithm; this is mainly due to the introduction of the $\boldsymbol{\beta}$ Max module.

The area usage of the implementations is very small (342–491 slices). It is comparable to the area of an 18-bit$\times$18-bit multiplier, which costs about 208 slices in a Virtex-4 FPGA chip [88]. Therefore, any other technique for solving linear equations that requires multiplications will be more costly than our implementations.

In these serial implementations, the residual vector is updated sequentially, requiring

Table 3.11: FPGA resources for real-valued serial implementations

Algorithms	Cyclic DCD		Leading DCD	
Resources	$N = 16$	$N = 64$	$N = 16$	$N = 64$
Slices	342 (2.5%)	364 (2.7%)	453 (3.3%)	491 (3.6%)
D-FFs	193 (0.7%)	214 (0.8%)	208 (0.8%)	243 (0.9%)
LUT4s	597 (2.2%)	645 (2.3%)	841 (3.1%)	902 (3.3%)
Block RAMs	3 (2.2%)	6 (4.4%)	3 (2.2%)	6 (4.4%)
Max Cycles:	$64N_u + 687$	$256N_u + 2703$	$22N_u + 76$	$70N_u + 124$

N cycles for each successful iteration. The serial architecture guarantees that the implementation is hardware efficient, but the update rate (throughput) is hindered. As the slice requirements of the DCD core is very low, it would be possible to implement many such cores on a single FPGA device and operate them in a pipelined fashion, giving a direct increase in the overall update rate. However, in the word we are interested in implementations that increase the update rate of the DCD processor itself.

3.4 Complex-Valued Serial Architecture DCD Algorithm

The architectures of the complex-valued cyclic and leading DCD algorithms are the same as the architectures of real-valued implementations shown in Fig. 3.1 and Fig. 3.3, respectively. However, now we assume that the complex-valued system to be solved is of the size $N/2 \times N/2$. Since an $N/2 \times N/2$ complex-valued system is equivalent to an $N \times N$ real-valued system, this allows correct comparison of hardware resources and performance of the two types of design. Each element of a complex-valued system of equations is represented using two 16-bit Q15 numbers (in turn representing the real component and the imaginary component). Real and imaginary components of $\mathbf{r}$ and $\mathbf{h}$ are stored using the 32-bit fixed-point Q15 format. To obtain a high update rate, the real and imaginary components of $\mathbf{r}$ are processed in parallel. Thus the configuration of the RAM components and operation of other modules are naturally different compared to the real-valued implementation. The $\mathbf{R}$ RAM now has a 32-bit data width and the $\boldsymbol{\beta}$ RAM has a 64-bit data width. This enables them to support both components of a complex-valued element simultaneously. The $\mathbf{h}$ RAM still has a 32-bit data width, as at each iteration only one component of a complex-valued element is required.

3.4.1 Implementation of Complex-Valued Cyclic DCD Algorithm

The complex-valued cyclic DCD algorithm optimized for FPGA implementation is shown in Table 3.12. At each iteration, the Comparator selects, according to the signal $s \in \{1, j\}$, either the real ($s = 1$) or imaginary ($s = j$) component of the element r_n to compare with $R_{n,n}$. The $\boldsymbol{\beta}$ Updater has two adders for processing the real and imaginary components, respectively, so that they are updated simultaneously. The $\mathbf{h}$ Updater only updates one component of h_n, as selected by the signal s. The other operations are similar to the operations of the real-valued cyclic DCD algorithm in Table 3.10.

The number of cycles required in each state is shown in Table 3.12. For each iteration, three cycles are used for comparison and other operations, and $N/2$ cycles for updating vectors $\mathbf{h}$ and $\mathbf{r}$. For a given N_u, the maximum number of cycles is $7NN_u/2 + 3N(M_b - 1) + M_b$. However, in a typical situation when there are many successful iterations within every pass of N iterations, the number of clocks will be close to $NN_u/2$.

Considering that the complex-valued system is equivalent to a double-size real-valued system, the overall update rate of this complex-valued implementation is faster than that of the implementation of the real-valued cyclic DCD algorithm, as both components of the residual vector $\mathbf{r}$ are updated simultaneously.

3.4.2 Implementation of Complex-Valued Leading DCD Algorithm

A hardware-optimized version of the complex-valued leading DCD algorithm is shown in Table 3.13. In this implementation, the $\boldsymbol{\beta}$ Max module has two units each seeking the maximum absolute value among the real and imaginary components of $\mathbf{r}$, respectively. The maximum of the two values is selected and the index (n, s) which indicates the maximum-value position is fed to the DCD Core State Machine. The functions of other modules are similar to those of the implementation of the complex-valued cyclic DCD algorithm.

For each iteration, three cycles are used for comparison, and $N + 4$ cycles are used for updating $\mathbf{r}$ and h_n, and finding the next leading element. Compared to the implementation of the real-valued leading DCD algorithm, the extra cycle in states 1 and 5 is caused by the $\boldsymbol{\beta}$ Max module which now has to compare the maximum absolute values of real and imaginary components of $\mathbf{r}$. Thus, for given N, N_u, and M_b, the maximum number of cycles is $(N/2 + 7)N_u + N/2 + 4M_b + 1$.

Table 3.12: Complex-valued cyclic DCD algorithm for serial FPGA implementation ($N/2 \times N/2$ system)

State	Operation	Cycles
0	Initialization: $\mathbf{h} = \mathbf{0}$, $\mathbf{r} = \boldsymbol{\beta}$, $m = M_b$, $k = 0$, $\Delta m = 0$, $s = 1$, $n = 1$	
1	if $m = 0$, algorithm stops else, $m = m - 1$, $\alpha = 2^m$, $\Delta m = \Delta m + 1$	1
2	if $s = 1$, then $r_{tmp} = \Re(r_n)$; else, $r_{tmp} = \Im(r_n)$ $c = R_{n,n}/2 - \lvert r_{tmp}\rvert \times 2^{\Delta m}$	1
3	if $c < 0$, then goto state 4 else, goto state 5	1
4	$h_n = h_n + \text{sign}(r_{tmp}) s\alpha$ $\mathbf{r} = \mathbf{r} \times 2^{\Delta m} - \text{sign}(r_{tmp}) s\mathbf{R}_{:,n}$ $\Delta m = 0$, $k = k + 1$, Flag = 1 if $k = N_u$, algorithm stops	$N/2$
5	if $s = 1$, then $s = j$, goto state 2 else, $s = 1$, $n = (n)\text{mod}(N/2) + 1$ if $n = 1$ and Flag = 1, then Flag = 0, goto state 2 elseif $n = 1$ and Flag = 0, then goto state 1 else, goto state 2	1
Total:	$\leq 7NN_u/2 + 3N(M_b - 1) + M_b$ cycles	

Table 3.13: Complex-valued leading DCD algorithm for serial FPGA implementation ($N/2 \times N/2$ system)

State	Operation	Cycles
0	Initialization: $\mathbf{h} = \mathbf{0}$, $\mathbf{r} = \boldsymbol{\beta}$, $m = M_b$, $k = 0$, $\Delta m = 0$	
1	$(n, s) = \arg\max_{p=1,\ldots,N/2}\{\lvert\Re(r_p)\rvert, \lvert\Im(r_p)\rvert\}$	$N/2 + 4$
2	if $m = 0$, algorithm stops else, $m = m - 1$, $\alpha = 2^m$, $\Delta m = \Delta m + 1$	1
3	if $s = 1$, $r_{tmp} = \Re(r_n)$; else, $r_{tmp} = \Im(r_n)$ $c = R_{n,n}/2 - \lvert r_{tmp}\rvert \times 2^{\Delta m}$	2
4	if $c < 0$, goto state 5 else, goto state 2	1
5	$h_n = h_n + \text{sign}(r_{tmp}) s\alpha$ $\mathbf{r} = \mathbf{r} \times 2^{\Delta m} - \text{sign}(r_{tmp}) s\mathbf{R}_{:,n}$ $(n, s) = \arg\max_{p=1,\ldots,N/2}\{\lvert\Re(r_p)\rvert, \lvert\Im(r_p)\rvert\}$ $\Delta m = 0$, $k = k + 1$ if $k = N_u$, algorithm stops; else, goto state 3	$N/2 + 4$
Total:	$\leq (N/2 + 7)N_u + N/2 + 4M_b + 1$ cycles	

Table 3.14: FPGA resources for complex-valued implementations

Algorithms	Complex Cyclic DCD		Complex Leading DCD	
Resources	$N = 16$	$N = 64$	$N = 16$	$N = 64$
Slices	610 (4.5%)	640 (4.7%)	837 (6.1%)	873 (6.4%)
D-FFs	263 (1.0%)	292 (1.1%)	346 (1.3%)	374 (1.4%)
LUT4s	1004 (3.7%)	1045 (3.8%)	1411 (5.2%)	1488 (5.4%)
Block RAMs	5 (3.7%)	6 (4.4%)	5 (3.7%)	6 (4.4%)
Max Cycles:	$56N_u + 687$	$224N_u + 2703$	$15N_u + 69$	$39N_u + 93$

3.4.3 FPGA Resources for Complex-Valued Implementations

The FPGA area requirements for the complex-valued DCD algorithms are shown in Table 3.14 for the cases of $N = 16$ and $N = 64$ with $M_b = 15$. The simultaneous processing of the real and imaginary components is equivalent to introducing 2-element parallelism with respect to the real-valued implementations. As a result, the areas of these implementations are approximately twice as large than that of the corresponding real-valued serial DCD implementations in Table 3.11, while the number of cycles used to update the residual vector is approximately twice as small. The area usage is still small and the implementations occupy less than 10% of the whole chip. We can draw upon this results and increase the degree of parallelism for obtaining higher update rates.

3.5 Real-Valued Group-4 Architecture DCD Algorithm

We have extended the concept of the 2-element (group-2) parallelism of the complex-valued DCD algorithms to process four consecutive elements of the residual vector simultaneously. These group-4 implementations bear understandable resemblance to the complex-valued implementations.

3.5.1 Group-4 Implementation of Cyclic DCD Algorithm

The architecture of the group-4 real-valued cyclic DCD algorithm is similar to the architecture of the real-valued cyclic DCD algorithm in Fig. 3.1. The matrix $\mathbf{R}$ can be treated as a resized matrix of size of $N/4$ by N, with each location containing the bitwise concatenation of four elements of $\mathbf{R}$. Likewise, the residual vector $\mathbf{r}$ can be viewed as a matrix of size 4 by $N/4$. The data widths of the $\mathbf{R}$ RAM and $\boldsymbol{\beta}$ RAM are configured to be 64

and 128 bits, respectively. Consequently four elements of the matrix $\mathbf{R}$ and the residual vector $\mathbf{r}$ can be accessed in a single cycle. In this implementation, the Comparator selects a diagonal element $R_{n,n}$ of $\mathbf{R}$ from four output elements of the $\mathbf{R}$ RAM and an element r_n of $\mathbf{r}$ from four output elements of the $\boldsymbol{\beta}$ RAM, compares them and outputs the comparison result to the DCD Core State Machine. The comparison costs one cycle. The $\boldsymbol{\beta}$ Updater has four processing units to update four rows of the 'matrix' $\mathbf{r}$ in parallel. Thus, $N/4$ cycles are required for updating $\mathbf{r}$. The other operations are similar to the operations of the serial implementation of the real-valued cyclic DCD algorithm shown in Table 3.9. For given N, N_u, and M_b, the maximum number of cycles is $(13N/4)N_u+3N(M_b-1)+M_b$. However, in a typical situation when there are many successful iterations in a pass of N iterations, the number of clocks will be close to $NN_u/4$.

3.5.2 Group-4 Implementation of Leading DCD Algorithm

The hardware architecture of the group-4 leading DCD algorithm is the same as the architecture of the real-valued leading DCD algorithm in Fig. 3.3. The RAM configuration for this implementation is the same as in the group-4 cyclic DCD algorithm. The comparison here costs 2 cycles, one for the RAM read latency and one for the arithmetic operation. The $\boldsymbol{\beta}$ MAX module contains four processing units and executes in two stages. In the first stage, it finds four maximum absolute values of the four rows of the 'matrix' $\mathbf{r}$; this costs $N/4$ cycles. In the second stage, it selects the maximum from these four values; this costs 2 cycles. The $\boldsymbol{\beta}$ MAX module then outputs the index of the largest element to the DCD Core State Machine. Taking into account the latency of the hardware, the total number of cycles used for the updating and finding the next leading element is $N/4+5$. The other operations are the same as in the serial implementation of the real-valued leading DCD algorithm shown in Table 3.10. For given N, N_u, and M_b, the maximum number of cycles is $(N/4+8)N_u+N/4+4M_b+2$. Comparing to the serial implementation of the leading DCD algorithm, the processing speed of this implementation is enhanced by a factor of approximately four.

3.5.3 FPGA Resources for Group-4 Implementations

The area usage of both group-4 implementations is presented in Table 3.15 for the cases of $N = 16$ and $N = 64$ with $M_b = 15$. The maximum possible number of cycles is also shown. The area usage of each implementation is about 3 times larger and the number of cycles for updating the residual vector $\mathbf{r}$ is one quarter compared to the same size serial

Table 3.15: FPGA resources for group-4 implementations

Algorithms	Cyclic DCD Group-4		Leading DCD Group-4	
Resources	$N = 16$	$N = 64$	$N = 16$	$N = 64$
Slices	986 (7.2%)	1019 (7.4%)	1468 (10.7%)	1522 (11.1%)
D-FFs	332 (1.1%)	356 (9.3%)	500 (1.8%)	538 (2.0%)
LUT4s	1618 (5.9%)	1670 (6.1%)	2607 (9.5%)	2703 (9.9%)
Block RAMs	7 (5.2%)	9 (6.6%)	7 (5.2%)	9 (6.6%)
Max Cycles:	$52N_u + 687$	$208N_u + 2703$	$12N_u + 66$	$24N_u + 77$

implementations.

Although the area usage of the group-4 implementations increases significantly when compared to the serial implementations, they still occupy at most only 11% of the FPGA chip. We may even desire to increase the number of group elements further in order to obtain a higher processing speed. For the group-4 implementation, the $\boldsymbol{\beta}$ Max unit spends two cycles comparing the four absolute values at stage 2. As the number of elements in the group increases, the number of clock cycles required at this stage will increase. At some point, this will become significant. Depending on the required processing speed of the application and the practical area limitation, one can decide upon a suitable number of elements in the group. For the cyclic algorithm, there is no such limitation. We can choose to increase the parallelism, even to update all elements of $\mathbf{r}$ simultaneously in a single cycle. This is explored in the following section.

3.6 Real-Valued Parallel Architecture Cyclic DCD Algorithm

We now consider a modification of the cyclic DCD algorithm where all N elements of the residual vector $\mathbf{r}$ are updated in a single cycle. The matrix $\mathbf{R}$ and vector $\boldsymbol{\beta}$ are stored in the Input RAM. Since execution is fully parallel, all elements of the residual vector $\mathbf{r}$ should be accessed simultaneously; therefore, the elements are stored in registers. As only one element of the vector $\mathbf{h}$ is involved in computation at each iteration, $\mathbf{h}$ is stored in a single RAM ($\mathbf{h}$ RAM). Also, since at each iteration, all elements in column $\mathbf{R}_{:,n}$ have to be read concurrently. We consider to store it in registers (Register-based DCD implementation) or in block RAMs (RAM-based DCD implementation).

Table 3.16: Cyclic DCD algorithm for parallel FPGA implementation

State	Operation	Cycles
0	Initialization: $\mathbf{h} = \mathbf{0}$, $\mathbf{r} = \boldsymbol{\beta}$, Flag=0, $m = M_b$, $k = 0$, $n = 1$	
1	if $m = 0$, algorithm stops else, $m = m - 1$, $\alpha = 2^m$, $\mathbf{r} = 2\mathbf{r}$	1
2	$\mathbf{r}_{tmp} = \mathbf{r} - \text{sign}(r_n)\mathbf{R}_{:,n}$	1
3	$c = R_{n,n}/2 - \lvert r_n \rvert$	1
4	if $c < 0$ $\mathbf{r} = \mathbf{r}_{tmp}$ $h_n = h_n + \text{sign}(r_n)\alpha$ $k = k + 1$, Flag = 1 if $k = N_u$, algorithm stops $n = (n)\text{mod}(N) + 1$ if $n = 1$ and Flag = 1, then Flag = 0, goto state 2 elseif $n = 1$ and Flag = 0, then goto state 1 else, goto state 2	1
Total:	$\leq 3NN_u + 3N(M_b - 1) + M_b$ cycles	

3.6.1 Register-based DCD Implementation

This register-based DCD implementation consists of three sub-modules: Control Logic, Input RAM Reader and $\boldsymbol{\beta}$ Updater, as shown in Fig. 3.5. The $\boldsymbol{\beta}$ Updater is designed to update all elements of $\mathbf{r}$ in parallel; it comprises N adder/subtractor units, as shown in Fig. 3.6.

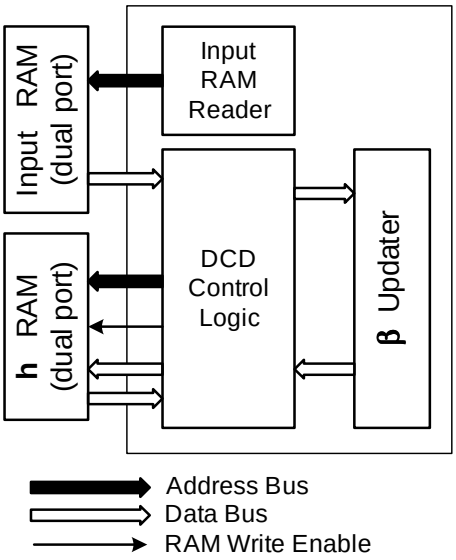


Figure 3.5: Architecture of Register-based DCD Core

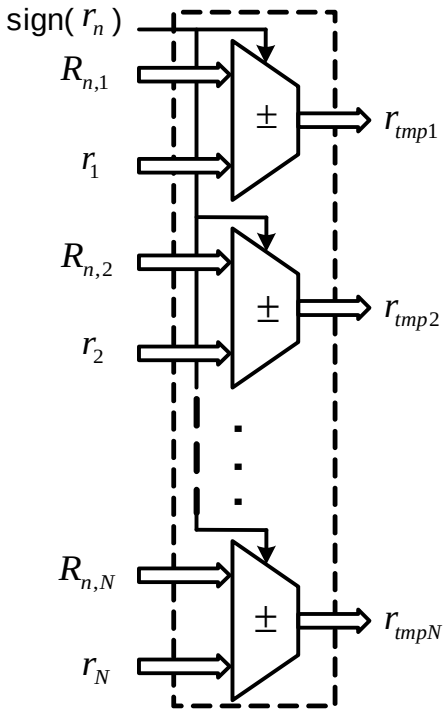


Figure 3.6: Architecture of $\boldsymbol{\beta}$ Updater

The cyclic DCD algorithm optimized for parallel FPGA implementation is shown in Table 3.16. In general, the operations are similar to the serial implementation of the cyclic DCD algorithm as shown in Table 3.9. During initialization (state 0), the matrix $\mathbf{R}$ and vector $\boldsymbol{\beta}$ are copied into the registers from the Input RAM. In state 1, since the vector $\mathbf{r}$ is stored in registers and all elements can be accessed simultaneously, the left-shift of the elements is executed in the same cycle as the update of the step-size α. In state 2, the DCD Core State Machine passes all elements of the column $\mathbf{R}_{:,n}$ and vector $\mathbf{r}$ to the $\boldsymbol{\beta}$ Updater. The total time for accessing the vector elements and computation on the FPGA chip is approximately 15 ns, which is one-and-a-half clock cycle. Hence, the results are read after 2 cycles. In state 3, the DCD Core State Machine compares r_n and $R_{n,n}$. State 4 is similar to states 3 to 5 of Table 3.9. The only difference is that the vector $\mathbf{r}$ is updated by replacing the contents of the $\boldsymbol{\beta}$ registers with the output $\mathbf{r}_{tmp}$ of the $\boldsymbol{\beta}$ Updater. This requires one cycle. Thus, for each iteration, three clock cycles are required. The maximum number of clock cycles used for the complete algorithm is $3NN_u+3N(M_b-1)+M_b$. However, in a typical situation, this figure will be significantly smaller. Since all elements of the matrix $\mathbf{R}$ are saved in registers, the area usage of this register-based implementation is high.

3.6.2 RAM-based DCD Implementation

The RAM-based implementation uses an array of relatively small RAMs to store the matrix $\mathbf{R}$; this allows significant reduction in the area usage. The architecture of the RAM-based DCD core is shown in Fig. 3.7. Besides Control Logic, $\boldsymbol{\beta}$ Updater and Input RAM Reader, it contains an $\mathbf{R}$ RAM Array Writer and $\mathbf{R}$ RAM Array. The RAM Array contains N block RAMs, which correspond to N rows of the matrix $\mathbf{R}$, as shown in Fig. 3.8. During the initialization (state 0), $\mathbf{R}$ RAM Array Writer copies the rows of $\mathbf{R}$ from Input RAM and writes each in a corresponding RAM in $\mathbf{R}$ RAM Array. The N output ports of $\mathbf{R}$ RAM Array are connected to corresponding adder's input ports in $\boldsymbol{\beta}$ Updater. To access elements of $\mathbf{R}_{:,n}$, the same address is presented to each RAM. The operation of this RAM-based implementation has the same five states and its time efficiency is exactly the same as that of the register-based implementation. However, the RAM-based implementation significantly reduces the area usage compared to that of the register-based implementation.

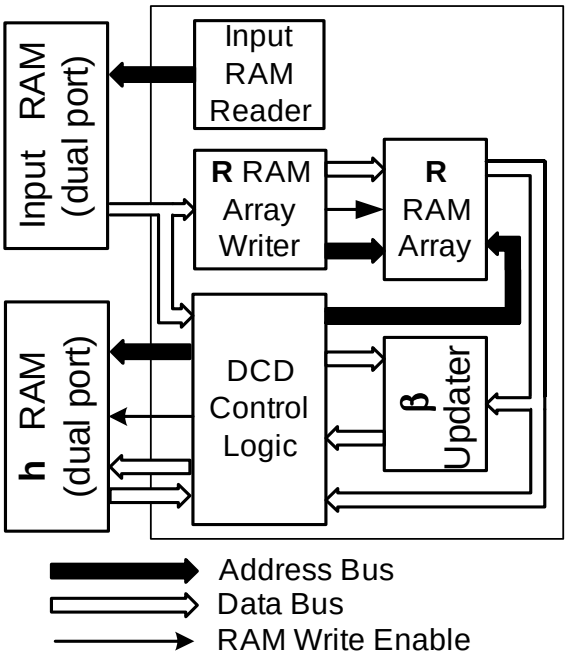


Figure 3.7: Architecture of RAM-based DCD Core

Table 3.17: FPGA resources for parallel implementation of cyclic DCD algorithm

Architectures:	Register-based		RAM-based	
Resources	$N = 16$	$N = 18$	$N = 16$	$N = 64$
Slices	7176(52.4%)	11775(86.0%)	1465(10.7%)	5307(38.6%)
D-FFs	5123(18.7%)	6201(22.6%)	802(2.9%)	2549(9.3%)
LUT4s	5646(20.6%)	18242(66.6%)	2754(10.1%)	10062(36.4%)
Block RAMs	2(1.5%)	2(1.5%)	18(13.2%)	70(51.4%)
Max Cycles	$48N_u + 687$	$54N_u + 770$	$48N_u + 687$	$192N_u + 2703$

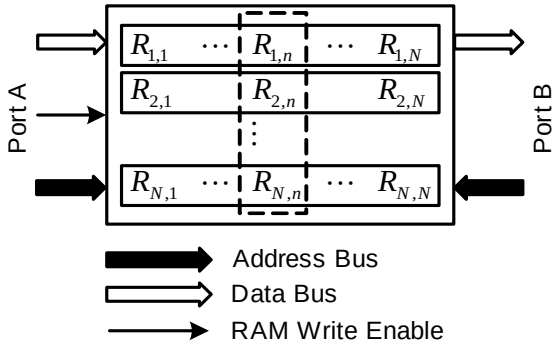


Figure 3.8: Architecture of RAM Array

3.6.3 FPGA Resources for Parallel Implementations

The area usage of both register-based and RAM-based implementations is presented in Table 3.17 for the cases of $N = 16$, $N = 18$ and $N = 64$ with $M_b = 15$. The maximum number of cycles is also shown. The parallel implementation does not need extra cycles to update the residual vector $\mathbf{r}$ as the residual vector $\mathbf{r}$ is updated simultaneously with the comparison operation. The area usage of the register-based implementation is quite high as all elements of the matrix $\mathbf{R}$ and residual vector $\mathbf{r}$ are stored in registers constructed from the regular FPGA fabric. It requires about 53% and 87% slices for cases of $N = 16$ and $N = 18$, respectively, and is not possible for larger size cases. The RAM-based implementation uses smaller number of slices than the register-based implementation by storing the matrix $\mathbf{R}$ in RAMs. From the area usage point of view, the RAM-based implementation is much more attractive as it is possible to solve large size systems of equations using a single chip. However, the area usage of the RAM-based implementation is still quite higher than the serial, group-2 and group-4 implementations. The required number of RAMs is highly related to the system size.

3.7 Numerical Results

In this section, we present numerical results that show that the different DCD designs may be useful for different applications. Specifically, the convergence speed (in terms of the number of updates and number of clock cycles) of the designs is demonstrated for different systems of equations.

We now compare the misalignment obtained by the proposed fixed-point designs of the DCD algorithms against their floating-point counterparts and other iterative techniques. Only real-valued systems are considered. In each trial, the matrix $\mathbf{R}$ is generated as $\mathbf{R} = \mathbf{S}\mathbf{S}^T$ where $\mathbf{S}$ is an $N \times M$ matrix whose elements are independent zero mean random numbers with normal distribution. We can view columns of $\mathbf{S}$ as spreading waveforms and, accordingly, the matrix $\mathbf{R}$ as a spreading waveform correlation matrix in a CDMA multiuser system with N users and a spreading factor $M \geq N$ [8]. By changing the relationship between M and N, we can change the condition number of $\mathbf{R}$, i.e., $\lambda_{max}/\lambda_{min}$, where λ_{max} and λ_{min} are the maximum and minimum eigenvalues of $\mathbf{R}$, respectively. The condition number is generally higher when N is closer to M. In such cases, we deal with highly loaded multiuser scenarios, where the multiuser detection based on solving the system $\mathbf{R}\mathbf{h} = \boldsymbol{\beta}$ with $\boldsymbol{\beta}$ being the vector of matched filter outputs, is especially complicated. We also generate a random $N \times 1$ vector $\mathbf{h}_0$ with elements uniformly distributed on $[-1, +1]$ representing transmitted data in a multiuser system. The vector $\boldsymbol{\beta}$ is generated as $\boldsymbol{\beta} = \mathbf{R}\mathbf{h}_0$. By solving (3.1), the DCD algorithm obtains a solution $\mathbf{h}$. The misalignment between vectors $\mathbf{h}_0$ and $\mathbf{h}$ is calculated as

$$\xi = \frac{[\mathbf{h} - \mathbf{h_0}]^T[\mathbf{h} - \mathbf{h_0}]}{\mathbf{h}_0^T\mathbf{h}_0}. \tag{3.6}$$

The values ξ obtained in 100 trials are averaged and plotted against the number of updates N_u or number of clock cycles.

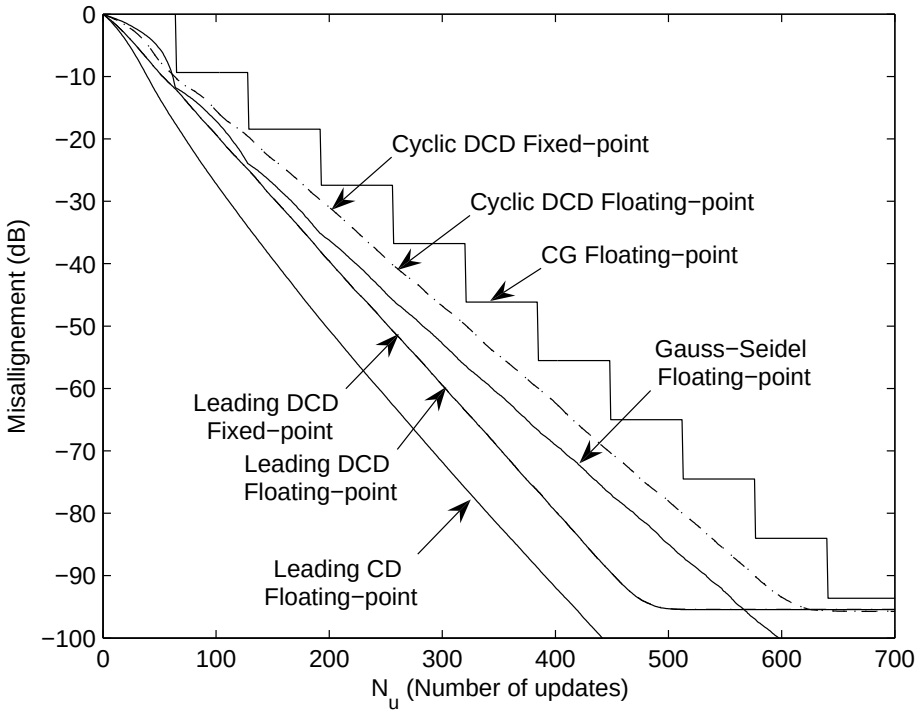


Figure 3.9: Misalignment for low condition numbers: $N = 64$, $M_b = 15$, $M = 512$, cond$\{\mathbf{R}\}$=[3,5].

We consider in Fig. 3.9 the system matrix $\mathbf{R}$ with small condition numbers. Note that here we only choose systems with condition numbers in the interval $[3, 5]$. It is seen that the performance of the fixed-point FPGA designs of the DCD algorithms are not distinguishable from that of the floating-point counterparts. The leading DCD algorithm shows a slightly faster convergence than the cyclic DCD algorithm. For this scenario, all considered iterative methods (DCD, CD, CG and Gauss-Seidel methods) demonstrate similar convergence speed. Note that one CG update of Table 3.1 is counted here as N updates. This allows comparison of different techniques with an approximately fixed complexity in terms of the number of arithmetic operations.

Note that if the accuracy required is very high, the total complexity of the DCD algorithm can be as high as $\mathcal{O}(N^2)$ or $\mathcal{O}(N^3)$. For example, in Fig. 3.9, if the required misalignment is chosen as -95 dB, the leading DCD algorithm and cyclic DCD algorithm require about 500 and 600 updates, and obtain a total complexity about $16N^2$ and $18N^2$ additions, respectively. However, one of the advantage of the iterative DCD algorithm is that we could stop the computation when the required accuracy is achieved, which means the total complexity of the DCD algorithm could be very low when the required accuracy is not high.

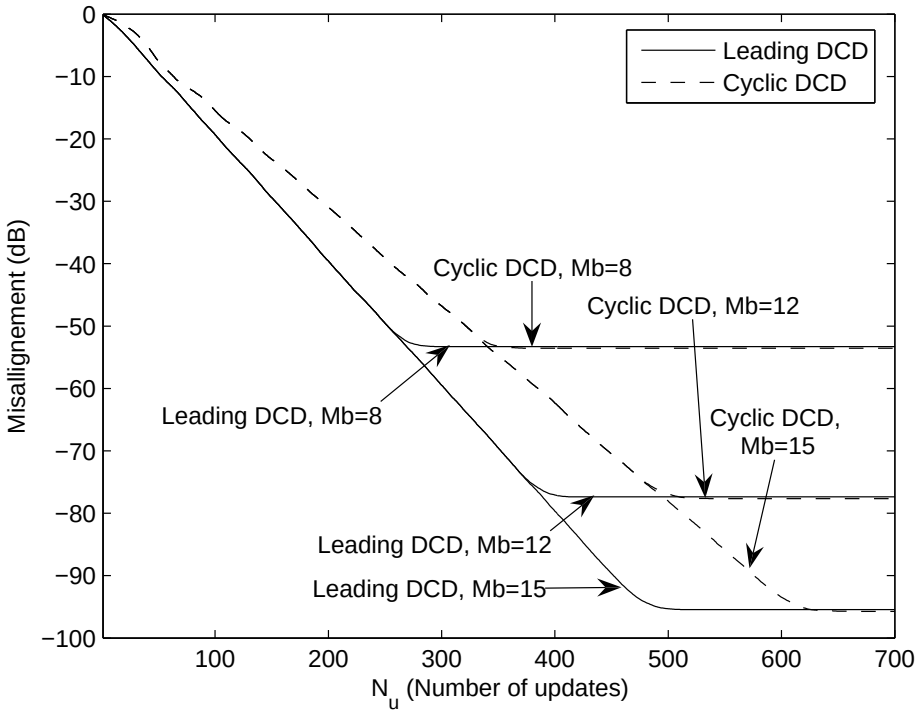


Figure 3.10: Misalignment for low condition numbers vs. M_b for fixed-point implementation of the DCD algorithms: $N = 64$, $M = 512$, cond$\{\mathbf{R}\}$ = [3,5].

Fig. 3.10 shows the dependence of the misalignment for different values of M_b. It can be seen that with increase in M_b, the steady state misalignment is reduced. However, even for $M_b = 8$, as small misalignment as -53 dB is achieved.

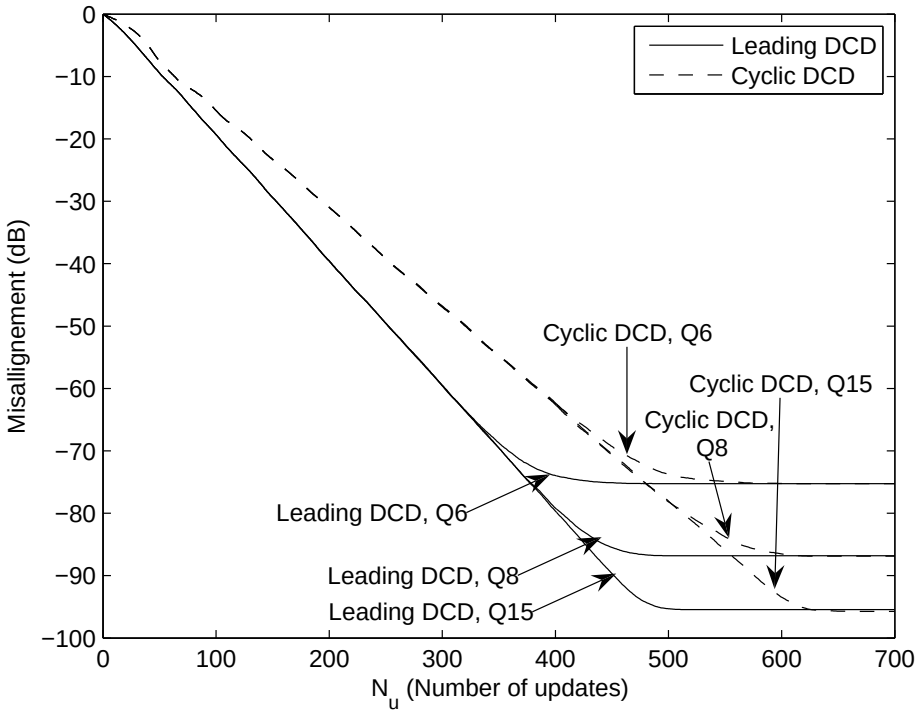


Figure 3.11: Misalignment for low condition numbers vs. the word bit-length (Q6, Q8, and Q15) for fixed-point implementation of the DCD algorithms: $N = 64$, $M = 512$, cond$\{\mathbf{R}\}$=[3,5].

Fig. 3.11 shows the dependence of the misalignment for different number of bits used for representation of the input data (matrix $\mathbf{R}$ and vector $\boldsymbol{\beta}$). Again, this only affects the steady-state misalignment. However, it is seen that even for the Q6 format (7 bits, including 1 sign bit and 6 fraction bits, used for representation of the input data), a very low steady-state misalignment is achieved.

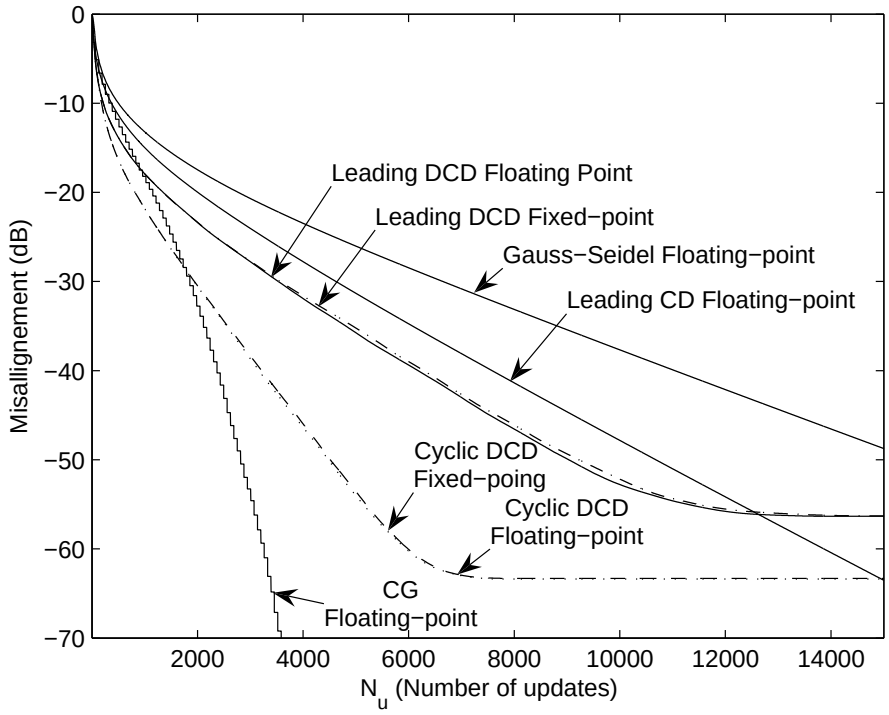


Figure 3.12: Misalignment for high condition numbers: $N = 64$, $M_b = 15$, $M = 75$, cond$\{\mathbf{R}\}$ = [400,500].

Results for high condition numbers are shown in Fig. 3.12. The difference between the fixed-point and floating-point implementations of the DCD algorithm is again very small. The convergence for both the DCD algorithms is now slower than in Fig. 3.9. A slightly faster convergence is provided by the cyclic DCD algorithm. For this scenario, the fastest convergence is demonstrated by the CG algorithm. The cyclic DCD algorithm requires approximately twice greater number of updates than the CG algorithm to achieve a misalignment of -63 dB. However, for lower accuracy the difference in the performance of the two techniques is smaller. It is interesting to note that both the DCD algorithms provide faster convergence than the two other coordinate descent techniques.

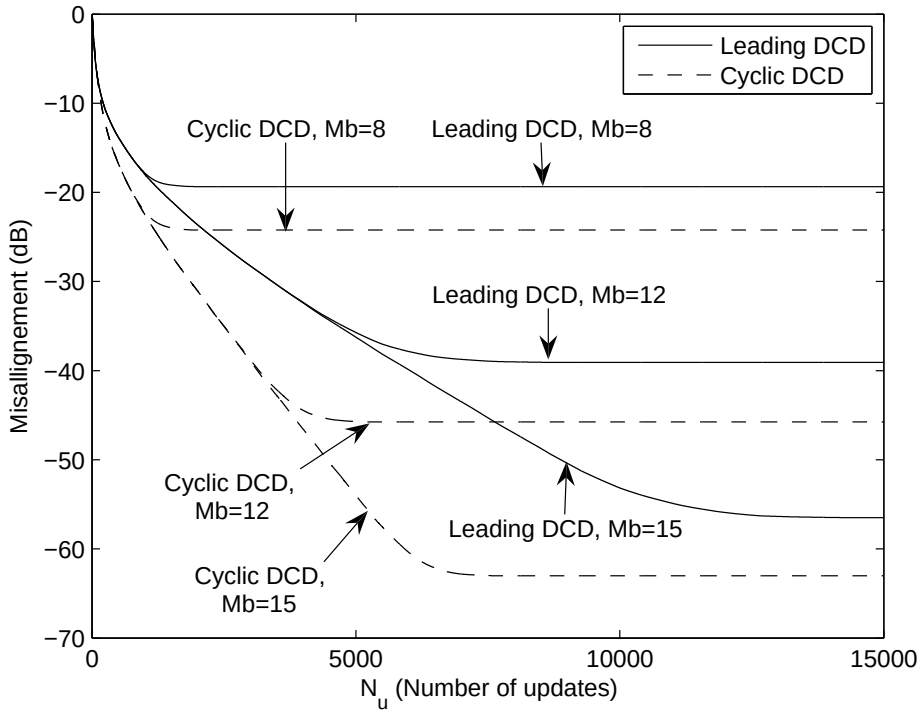


Figure 3.13: Misalignment for high condition numbers vs. M_b for fixed-point implementation of the DCD algorithms: $N = 64$, $M = 512$, cond$\{\mathbf{R}\}$ = [400,500].

Fig. 3.13 shows the dependence of the misalignment for different values of M_b. With increase in M_b, the steady state misalignment is reduced. Comparing to the case of small condition numbers (Fig.3.9), now the parameter M_b should be higher to achieve the same steady-state misalignment. For a fixed M_b, the cyclic DCD algorithm provides a lower steady-state misalignment than the leading DCD algorithm.

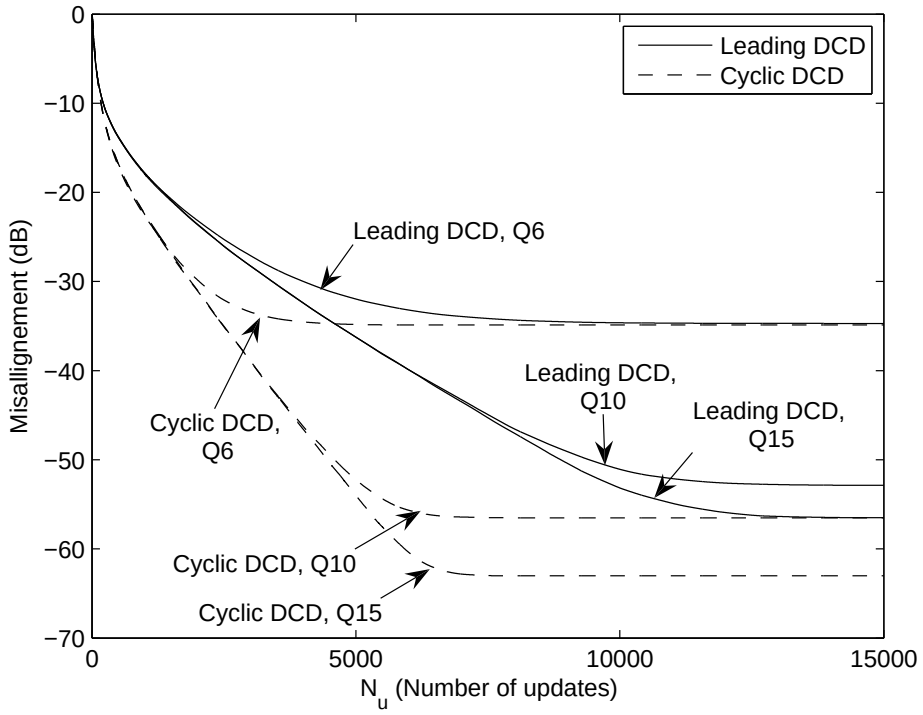


Figure 3.14: Misalignment for high condition numbers vs. the word bit-length (Q6, Q10, and Q15) for fixed-point implementation of the DCD algorithms: $N = 64$, $M = 512$, cond$\{\mathbf{R}\}$ = [400,500].

Fig. 3.14 shows the dependence of the misalignment for different number of bits used for representation of the input data. Again, this only affects the steady-state misalignment. However, it is seen that even for the Q6 format, a low steady-state misalignment is still achieved.

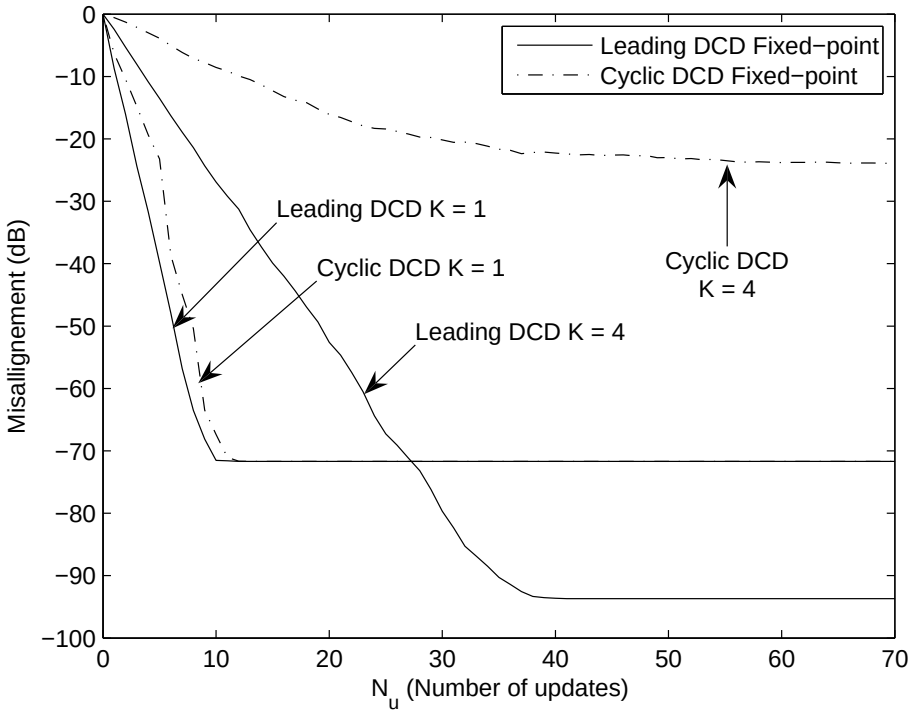


Figure 3.15: Misalignment for high condition numbers and sparse solutions: $N = 64$, $M_b = 15$, $M = 75$, cond($\mathbf{R}$)=[400, 500].

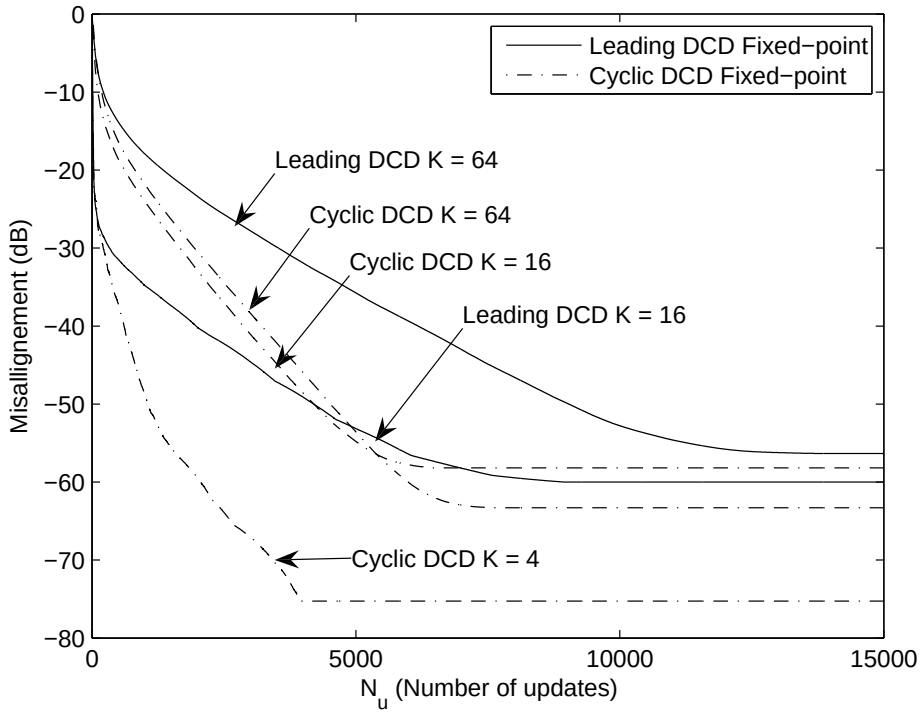


Figure 3.16: Misalignment for high condition numbers and sparse solutions: $N = 64$, $M_b = 15$, $M = 75$, cond($\mathbf{R}$)=[400, 500].

Now we consider scenarios with sparse vectors $\mathbf{h}_0$; in every trial, only K randomly chosen elements of the vector are non-zero. Scenarios with sparse true solutions appear in many applications. For example, in multipath communication channels, the number of multipath components can be very low with respect to the delay uncertainty interval. Since optimal channel estimation, such as maximum likelihood or minimum mean square error estimation, is usually based on solving the normal equations [89], in sparse multipath channels we arrive at systems with sparse true solutions $\mathbf{h}_0$. Another example is the multiuser communication, when the number of active users K involved in communication is smaller than the expected number of users N [90].

If a true solution is sparse, we can expect a reduction in the number of updates N_u required to achieve a predefined misalignment. This is due to existence of unsuccessful iterations. Fig. 3.15 and Fig. 3.16 support this conclusion for the case of high condition numbers. The sparser the solution, i.e., the smaller the value of K, the faster the convergence of the DCD algorithm. Note that this is a property of the DCD algorithm that is not general for other iterative techniques.

It is seen that for $K = 16$, the number of updates required to achieve a misalignment of –50 dB for the leading DCD algorithm is reduced by a factor of about two compared to non-sparse systems. For the cyclic DCD algorithm, the reduction is insignificant. For $K = 4$, for the leading DCD algorithm N_u is reduced by about 50 times, whereas for the cyclic DCD algorithm, this reduction is about 4 times. Thus, for highly sparse scenarios, the use of the leading DCD algorithm is preferable as it allows significant reduction in the number of updates. However, for non-sparse systems the cyclic DCD algorithm may be preferable as it provides faster convergence and smaller steady-state misalignment.

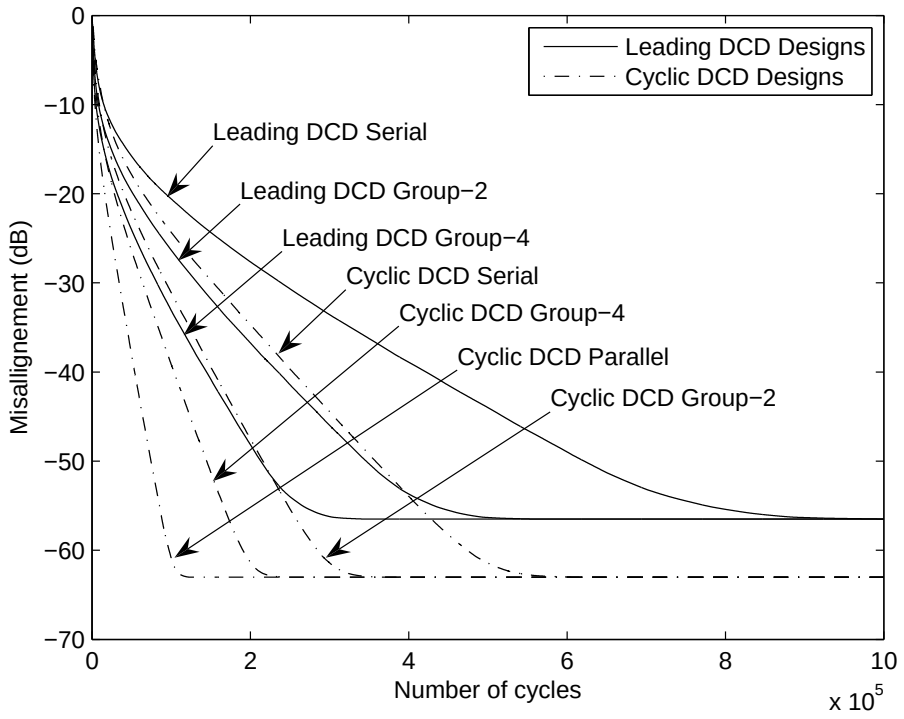


Figure 3.17: Misalignment against number of clock cycles: N=64, M_b=15, $M = 75$, cond($\mathbf{R}$) = [400, 500].

Fig. 3.17 shows the misalignment against the number of clock cycles for high condition numbers and non-sparse solutions. It is seen that, within a design group the cyclic DCD algorithm provides faster convergence in terms of the number of clocks than the leading DCD algorithm. For the cyclic DCD algorithm, by comparing results in Fig. 3.17 and Fig. 3.12, we can see that the average number of clocks for the four designs (serial, group-2, group-4, and parallel) can be estimated as $1.3NN_u$, $0.8NN_u$, $0.5NN_u$, and $0.25NN_u$, respectively. These figures are significantly lower than the maximum number of clocks $4NN_u$, $3.5NN_u$, $3.25NN_u$, and $3NN_u$, corresponding to the worst-case scenarios as discussed above. They are closer to the scenarios where there are many (significantly

more than one) successful iterations in every pass of N iterations: NN_u, $0.5NN_u$, and $0.25NN_u$ for the first three designs, respectively.

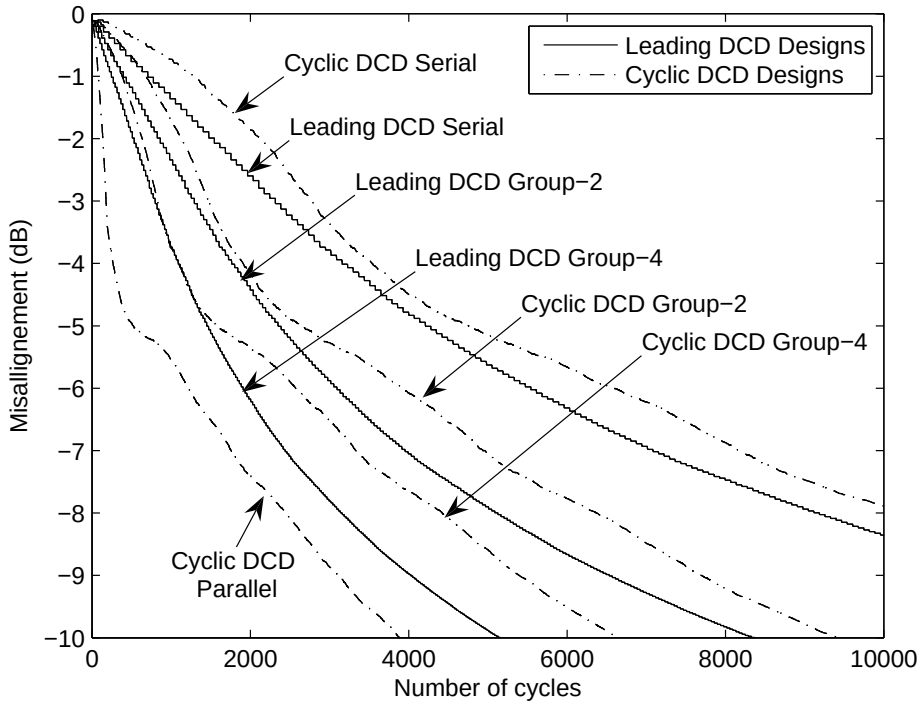


Figure 3.18: Misalignment against number of clock cycles: N=64, M_b=10, $M = 75$, cond($\mathbf{R}$) = [400,500].

If the required accuracy is not high, as is the case in an iteration of an adaptive filter [91, 92], the leading DCD algorithm may provide a better performance with a smaller number of cycles. This is seen from results in Fig. 3.18. However, the cyclic DCD algorithm requires a chip area that is $25\% - 33\%$ smaller (see Table 3.18).

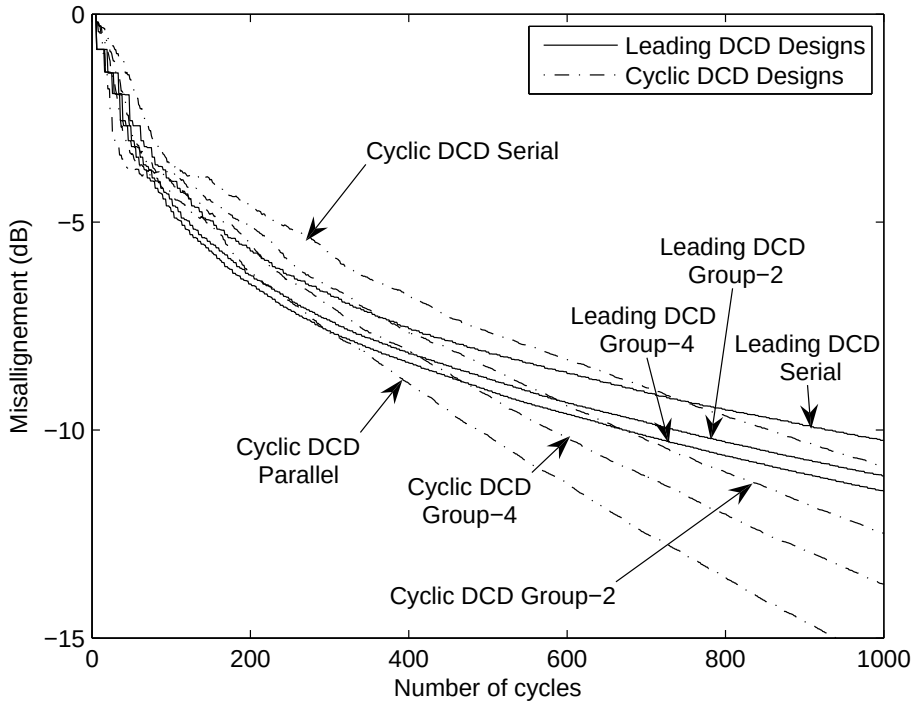


Figure 3.19: Misalignment against the number of clock cycles: N=8, M_b=8, $M = 8$.

Finally, in Fig. 3.19, results are presented for a small size problem, $N = 8$, and $M = 8$. This scenario corresponds to the MMSE detector for complex-valued symbols in a MIMO system with 4 transmit and 4 receive antennas in Rayleigh fading channels. The system model is represented as

$$\mathbf{y} = \mathbf{H}\mathbf{x} + \mathbf{n}, \tag{3.7}$$

where $\mathbf{y}$ and $\mathbf{x}$ are the 4×1 receive and transmit data vectors, respectively and, $\mathbf{H}$ and $\mathbf{n}$ are the 4×4 channel matrix and 4×1 noise vector, respectively. The data vector $\mathbf{x}$ can be found as the solution vector $\mathbf{h}$ of the linear system $\mathbf{R}\mathbf{h} = \boldsymbol{\beta}$, where $\mathbf{R} = \mathbf{H}^H\mathbf{H}+\sigma^2\mathbf{I}$ for an MMSE detector, σ^2 is the noise variance, $\mathbf{I}$ is an $N \times N$ identity matrix and $\boldsymbol{\beta} = \mathbf{H}^H\mathbf{y}$. In this scenario, a misalignment of –10 dB may be considered to be good enough for reliable detection. It is seen that the parallel design of the cyclic DCD algorithm requires about 500 clocks to reach the misalignment -10 dB, which is comparable to the designs in [93] (388 clocks). However, our design requires only 847 slices and no multiplier against 8513 slices and 64 multipliers in [93].

The convergence speed in terms of the number of cycles improves with increase in parallelism. For example, in the case of $N = 64$ (see Fig. 3.17) for the cyclic DCD algorithm, the group-4 design speeds up the convergence by about 2.5 compared with the

serial design; however, this also requires increase in the chip area by a factor of about 2.8. The parallel design has the fastest convergence which for this scenario is about 5 times faster than that of the serial design, but the chip area increases by a factor of about 15. However, for smaller-size problems, e.g., $N = 16$, as seen from Table 3.18, FPGA resources for the parallel design are comparable to those of the group-4 design. Moreover, for $N = 8$, the parallel design requires even fewer resources than the group-4 design.

Finally, Table 3.18 shows the power consumption for the designs (at the clock frequency 100 MHz). For a fixed N and the same design type, the power consumption is slightly smaller for the cyclic DCD algorithm than for the leading DCD algorithm. It increases with the increase in parallelism, with the parallel design requiring the highest power.

Table 3.18: Number of FPGA slices and power consumption of DCD designs

	Number of slices			Power consumption (mW)	
Design	$\mathbf{N = 8}$	$\mathbf{N = 16}$	$\mathbf{N = 64}$	$\mathbf{N = 16}$	$\mathbf{N = 64}$
Serial Cyclic	-	342	364	11	21
Serial Leading	-	453	491	16	20
Group-2 Cyclic	-	610	640	10	19
Group-2 Leading	-	837	873	17	22
Group-4 Cyclic	978	986	1019	18	25
Group-4 Leading	-	1468	1522	27	31
Parallel Cyclic	847	1465	5307	44	120

3.8 Conclusions

In this chapter, we have proposed and compared several FPGA designs of the DCD algorithm that solves normal equations. Two variants of the DCD algorithm were considered: cyclic and leading DCD algorithms. We have demonstrated that each of the two variants may be useful in different applications. For example, if the true solution is sparse, as in multipath channel estimation or multiuser detection with an unknown number of active users, the leading DCD algorithm is preferable. The sparser the true solution, the faster the convergence of the DCD algorithm. If accuracy is not an issue, as in an iteration of an adaptive filter, the leading DCD algorithm provides a faster convergence compared to the cyclic DCD algorithm. However, if the system matrix has a high condition number and the system is not sparse, the cyclic DCD algorithm may provide faster convergence.

The DCD algorithm is multiplication-free and division-free and, therefore, is well

suited to hardware implementation. The proposed fixed-point FPGA designs provide an accuracy performance which is very close to the performance of floating-point counterparts. The number of bits used for representation of the solution vector and the input data only affect the steady-state misalignment. The proposed designs require significantly lower FPGA resources than techniques based on QR decomposition. The serial designs require the smallest FPGA resources; they are well suited for applications where many parallel solvers are required, e.g., for detection in MIMO-OFDM [94, 95] systems. The parallelism introduced in the proposed group-2 and group-4 designs allows faster convergence to the true solution at the expense of an increase in the FPGA resources. The design with parallel update of the residual vector provides the fastest convergence speed; however, if the system size is high, it results in significant increase in FPGA resources. For the system matrix with a high condition number, within a design group, the cyclic DCD algorithm provides faster convergence in terms of the number of clocks than the leading DCD algorithm. Although, for a large system size, the increase in the parallelism reduces the number of clocks, the corresponding increase in FPGA resources may be significant thus making the serial design more attractive for implementation. For small-size systems ($N \leq 16$), the parallel design can be more attractive than the partly parallel designs from the viewpoint of both the number of clock cycles and FPGA resources.

In theory, the DCD algorithm requires infinite number of iterations (or updates) to converge to an optimal solution; this is a drawback for all iterative methods. Therefore, the number of updates and processing time is individual for each system to be solved. However we could estimate the required number of iterations based on the information of the system size, the conditional number of the system matrix and required accuracy. Comparing with other iterative methods, such as the CG, Gauss-Seidel and CD algorithms, the DCD algorithm performs a similar convergence speed with the lowest complexity per iteration.

In the next chapters, we will consider applications of architectures and implementations of the DCD algorithm developed in this chapter for complex division in Chapter 4, MVDR beamforming in Chapter 5 and adaptive filtering in Chapter 6.

The cyclic DCD algorithm is suitable for solving systems of equations requiring large number of updates. It converges faster and obtains a lower steady-state misalignment than the leading DCD algorithm. However, its convergence at initial updates is slower than that of the leading DCD algorithm. Therefore, we may consider to accelerate the convergence of the cyclic DCD algorithm by combining it with the leading DCD algorithm. Different combinations can be considered as future work in Chapter 7.

Chapter 4

Multiplication-Free Complex Divider

Contents

4.1 Introduction

The division of two complex numbers are used widely in the areas of signal processing. The fundamental method of complex division (2.27) eliminates the imaginary component of the divisor by multiplying both the dividend and the divisor by the complex conjugate of the divisor. This method requires six real multiplications, three real additions and two real divisions. Therefore, the computational load of the fundamental method is high. Moreover, the conventional method may cause overflow or underflow errors. The Smith algorithm [64] (2.28) avoids the problems of overflow and underflow of the conventional method. However, the Smith algorithm's higher numerical accuracy comes at the cost of computational complexity. Some other algorithms, such as the digit-recursion scheme [65, 66] as discussed in Section 2.5, are also complicated and not suitable for real-time hardware processing.

Alternatively, the complex division problem $q = r/d$, where $q = q_r + jq_j$ is the

quotient, $r = r_r + jr_j$ is the dividend and $d = d_r + jd_j$ is the divisor, can be viewed as a problem of finding the solution of a system of linear equations [68]

$$\begin{bmatrix} d_r & -d_j \\ d_j & d_r \end{bmatrix} \begin{bmatrix} q_r \\ q_j \end{bmatrix} = \begin{bmatrix} r_r \\ r_j \end{bmatrix}. \tag{4.1}$$

Thus, we can use the DCD algorithm to solve this 2×2 system of equations to realize the complex division without multiplication and division operations. This leads to a widely-applicable complex division core with remarkably low FPGA footprint.

The rest of this chapter is organized as follows. In Section 4.2, the DCD algorithm for the complex division is introduced. FPGA implementation of this DCD-based complex divider is discussed in Section 4.3. In Section 4.4, results of the implementation are analyzed and compared to the direct implementation of the fundamental method of complex division described in equation (2.27). Finally, conclusions are given in Section 4.5.

4.2 Algorithm Description

As described in [68], the system of equations (4.1) can be solved by using the DCD iterations as follows:

1. The algorithm compares the absolute values of r_r and $d_r\alpha/2$, where α is a system step size parameter. If $|r_r|$ is greater than $|d_r\alpha/2|$ (such comparison is labelled "successful") then q_r and both components of r are updated. If $|r_r| \leq |d_r\alpha/2|$ then the comparison is labelled "unsuccessful" and no update takes place. This is illustrated as

$$\begin{aligned} &\text{if } |r_r| > |d_r\alpha/2| \text{ then} \\ &\quad q_r = q_r + \text{sign}(r_r)\alpha, \\ &\quad r = r - \text{sign}(r_r)\alpha d. \end{aligned} \tag{4.2}$$

2. Then the algorithm compares $|r_j|$ and $|d_j\alpha|$. If $|r_j|$ is greater, q_j and both components of r are updated according to

$$\begin{aligned} &\text{if } |r_j| > |d_j\alpha| \text{ then} \\ &\quad q_j = q_j + \text{sign}(r_j)\alpha, \\ &\quad r = r - j\ \text{sign}(r_j)\alpha d. \end{aligned} \tag{4.3}$$

3. After executing (4.2) and (4.3), the algorithm applies (4.2) once more. The algorithm then reduces the step size α by half and moves on to the next level of precision. The process repeats until the LSB of the quotient is obtained.

In hardware, the initial step size α is chosen as a power of 2 so that the multiplication and division operations in (4.2) and (4.3) can be implemented by bit-shifting.

For division operation, value range of the operands must be limited to avoid quotient overflow and also low-value quotients that do not make good use of the available resolution [68]. For example, in a real division, suitable operands might involve the absolute value of dividend being in the range $[1/4, 1)$ and the absolute value of divisor being in the range of $[1/2, 1)$, obtaining an absolute value of the quotient in the range $[1/4, 1)$ [68]. For this complex division algorithm, a similar limitation can be performed on the modulus value range of the dividend and divisor. Specifically, the modulus of the dividend is less than $1/2$; this is achieved by limiting the absolute value of each component of the dividend to no larger than $1/4$. The absolute value of one component of the divisor must be larger than $1/2$. Moreover, the value d_r should be positive and larger than the absolute value of d_j. Therefore, the left side matrix in system of equations (4.1) is positive definite and the convergence of DCD iterations is guaranteed. These constraints are illustrated as

$$|r_r| \leq 1/4 \text{ and } |r_j| \leq 1/4, \tag{4.4}$$

$$1/2 \leq d_r < 1 \text{ and } d_r \geq |d_j|\,. \tag{4.5}$$

In hardware, these constraints can be met easily through bit-shift operations, component transpositions, and negations. As analyzed in [68], both components of the final quotient are faithfully rounded, and the maximum error value is less than one LSB.

4.3 FPGA Implementation of the Divider

VHDL is used to describe the DCD based multiplication-free complex divider, which is synthesised and downloaded to the target platform using the Xilinx ISE 8.1i software package. All initial components of r and d are represented in the 16-bit Q15 format [87] which implies a range of $[-1, 1)$. To avoid overflow during computation, the variables are stored internally in 32-bit Q15 format inside the FPGA core.

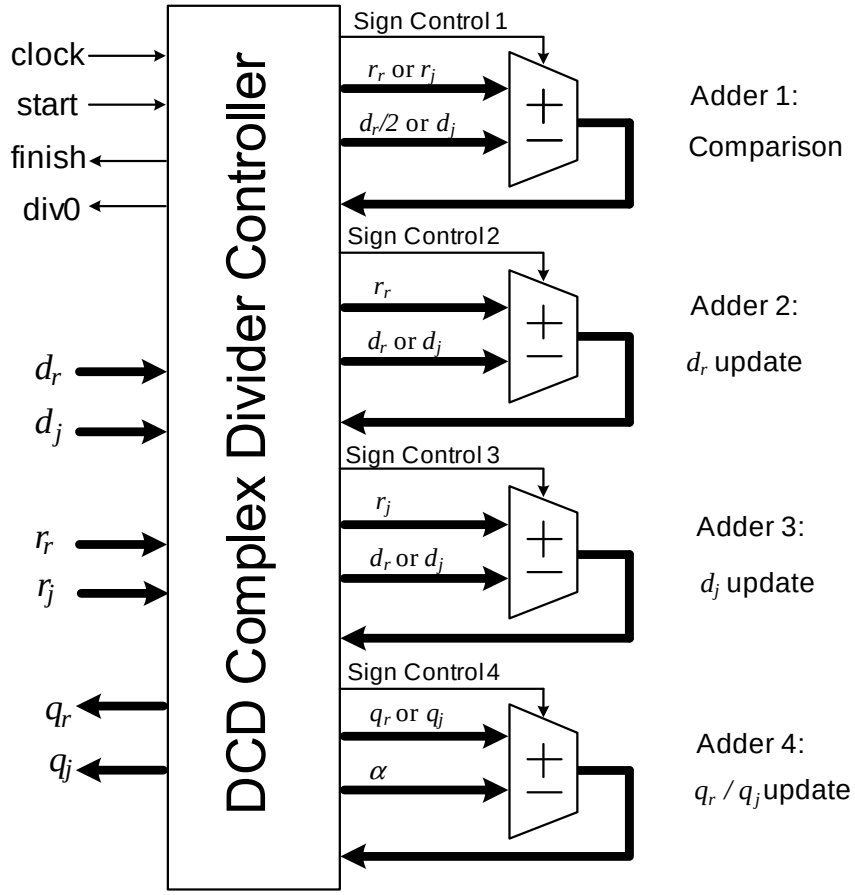


Figure 4.1: Architecture of the complex divider

The implementation consists of a controller and four adder/subtractor modules, as shown in Fig. 4.1. The whole implementation is operated synchronously from an external clock. The divider is triggered by a "start" signal and asserts a "finish" signal to acknowledge the completion of the division. When facing a zero divisor, the divider asserts a "div0" error signal. All components of r, d and q are transferred on separate ports.

The initial step size α is set to 0.5 and is halved at the end of each level of precision by means of a right bit-shift. Instead of computing $d_r\alpha$ and $d_j\alpha$ in equations (4.2) and (4.3) by right bit-shifting d_r and d_j, we choose to left-shift r_r and r_j at the end of each step size (or each level of precision) and also during the preprocessing. As well as avoiding any underflow conditions in our chosen number format, this simplifies the overall computation.

The operations of the DCD complex division, which have been optimized for FPGA implementation, are shown in Table 4.1. The algorithm can be divided into four states.

Table 4.1: The DCD complex division for FPGA implementation.

State	Operation	Cycles
1	$q = 0$, $\alpha = 0.5$, $m = M_b$ prescale r, d and α if $d = 0$, algorithm stops	17 (max) or 3 (min)
2	if $\|r_r\| > d_r/2$ $r_r = r_r - \text{sign}(r_r)d_r$ $r_j = r_j - \text{sign}(r_r)d_j$ $q_r = q_r + \text{sign}(r_r)\alpha$	1
3	if $\|r_j\| > \|d_j\|$ $r_r = r_r + \text{sign}(r_j)d_j$ $r_j = r_j - \text{sign}(r_j)d_r$ $q_j = q_j + \text{sign}(r_j)\alpha$	1
4	if $\|r_r\| > d_r/2$ $r_r = r_r - \text{sign}(r_r)d_r$ $r_j = r_j - \text{sign}(r_r)d_j$ $q_r = q_r + \text{sign}(r_r)\alpha$ if $m = 1$ algorithm stops else $r = 2r$, $\alpha = \alpha/2$, $m = m - 1$ goto state 2	1

In state 1, the core initializes the quotient q, step size α, and precision level signal m to 0, 0.5 and M_b respectively, where M_b is the number of fractional bits used to represent the quotient. The values of r and d are then prescaled according to equations (4.4) and (4.5). If r is shifted right, α is shifted the same number of bits left. Likewise, if d is transposed and/or shifted left, r is also transposed in the same manner and/or α is shifted the same number of bits left. Accordingly, the quotient q will not be affected in any way and the process or postscaling is avoided. The number of cycles required for this state is variable and will depend upon both the initial value of d and its notation format. For the 16-bit Q15 format, the minimum and maximum number of cycles required are 3 and 17 respectively.

The remaining three states (state 2, 3 and 4), which correspond to the three iterations per level of precision as explained in the algorithm description section, represent the main functionality of the DCD complex division. Each state executes in a single clock cycle. The comparison and three additions are executed using four adder/subtractor units (herein referred to as adders) as shown in Fig. 4.1. During each state, the controller provides operands and sign control signals to four adders, which in turn provide valid results after a short propagation delay. On the rising clock edge of the next state, the controller tests the sign bit (MSB) of the output of adder 1, which indicates the result of the comparison.

If comparison is "successful", the controller reads the outputs of adders 2, 3 and 4 to overwrite r_r, r_j, and q_r or q_j, respectively. If comparison is "unsuccessful" the controller ignores outputs of the adders. The controller then provides new data to the adders for the next iteration.

After three cycles of these compare-update iterations, the controller shifts r one bit left and α one bit right for the next precision level, and then goes back to state 2. The computation stops when the LSB of the quotient has been processed.

4.4 FPGA Resources and Throughput of the Divider

Our implementation has been implemented on a Xilinx Virtex-II Pro Development System [77] with an XC2VP30 (FF896 package, speed grade 7) FPGA chip [76]. The overall time taken by the DCD complex division depends on the demanded accuracy (number of bits M_b beyond the decimal point) and the operand notation (Qn) format. The maximum number of cycles for each division is $3M_b+n+2$. For the current implementation (M_b=15, n=15 and 100 MHz clock), the maximum number of cycles is 62 and so the throughput is at least 1.6 MHz. The maximum error of this implementation is less than 2^{-15}. As no multiplication and division operations are required, only 527 logic slices are required for this implementation.

The number of cycles cost on the DCD iterations (states 2 to 4) is fixed to be 45 as $M_b = 15$. While the number of cycles required by the state 1 varies from 3 to 17, according to the initial value of the divisor. Considering that the initial absolute value of the divisor has a uniform distribution in the range [0, 1], therefore the average number of cycles of this complex divider is around 55 cycles.

For comparison, we have also implemented a fundamental complex divider based on equation (2.27) with equivalent operand restrictions and same frequency clock. The required multipliers and divisors are implemented as Xilinx IP cores working on the 100 MHz system clock. The implementation also uses a 16-bit Q15 format to represent the initial data and 32-bit Q15 to represent the quotient. To obtain the same accuracy, the internal divisor ${d_r}^2 + {d_j}^2$ is shifted 15 bits right prior to the fixed-point division. And to obtain the smallest area usage as possible, the divider can not be pipelined and the number of latency cycles for each division is fixed as 50, which is by 12 clock cycles smaller than that of the DCD complex divider. However the number of logic slices required to implement the implementation is 2318, which is at least approximately 4.4 times higher

than that required by our DCD-based complex division implementation.

It should be noted that hardware and timing requirements will vary according to implementation platform and precise application requirements, but comparing these two implementations we can say that our DCD-based implementation trades off a certain proportion of execution time in return for a very small FPGA footprint. Given that complex divisions occur less frequently in signal processing algorithms than common operations such as additions and multiplications, it makes sense adopt this small-footprint approach.

4.5 Conclusions

We have presented an FPGA implementation of a complex divider based on the idea that the complex division problem can be viewed as a problem of finding the solution of a 2×2 real-valued system of linear equations, which is solved using the DCD algorithm. To obtain high accuracy results, constraints on the operands are introduced. However, these constraints can be achieved easily in hardware through bit-shift operations, component transpositions and negations. Therefore, the implementation is simple and does not use any multiplication or division operations. The area usage of this implementation is only 527 logic slices for M_b=15 and Q15 number representation. When operating from a 100 MHz clock, the throughput is at least 1.6 MHz. The maximum quotient error is less than one LSB.

Although the throughput of the fundamental complex divider is about 24% higher than that of the worst case of the DCD divider, our implementation requires 4.4 times smaller number of slices to provide the same accuracy. Furthermore, the average number of cycles of the DCD divider will be lower.

Chapter 5

Application: FPGA-based MVDR Beamforming using DCD Iterations

Contents

5.1 Introduction

Adaptive beamforming based on the MVDR criterion [70] (2.32) achieves high levels of interference cancellation, although MVDR requires matrix inversion and is considered too computationally complex for practical implementation. Efficient algorithms for MVDR beamforming based on QRD with Givens rotations (QR-MVDR) [6, 96] are inherently well-suited to programmable logic platforms, exploiting their parallel processing and pipelining capabilities using systolic array structures. Application of the CORDIC algorithm to QR-MVDR could facilitate practical FPGA implementation by enabling multiplication-free vector rotations [97]. However, implementation of CORDIC-based QR-MVDR is still hindered greatly by the large FPGA slice count required for CORDIC

algorithmic units [36, 51], making large beamforming arrays impracticable on all but the largest available FPGA devices.

In this chapter, we propose an efficient iterative method for MVDR beamforming that employs the DCD algorithm for multiplication-free solution of the normal equations. An FPGA implementation of the proposed method is shown to have complexity, in terms of slice count, much lower than can be achieved with other implementations. Performance of the fixed-point implementation is shown to be very close to the performance of a floating-point implementation of the MVDR beamformer by using direct matrix inversion.

The rest of this chapter is organized as follows. In Section 5.2, the beamforming configuration is introduced. In Section 5.3, FPGA implementation is discussed in detail. The numerical results and the FPGA implementation results are presented in Section 5.5. Finally, conclusions are given in Section 5.7.

5.2 Beamforming Configuration

The configuration that we examine is a linear antenna array, comprising N individual receiving elements. The signal from each antenna element is down-converted (DC) to baseband and digitized using an analog-to-digital converter (ADC). The instantaneous complex-valued $N \times 1$ vector $\mathbf{x}(i)$ provided by the array outputs at time instant i is referred to as a "snapshot". A stream of K snapshots is used for calculating the sample correlation matrix

$$\mathbf{R}(i) = \frac{1}{K} \sum_{k=i}^{i-K+1} \mathbf{x}(k)\mathbf{x}^H(k). \tag{5.1}$$

The complex-valued $N \times 1$ steering vector $\boldsymbol{\beta}(i)$ is assumed to be provided by an external DoA (direction of arrival) estimator or similar component. This topology is shown in Fig. 5.1.

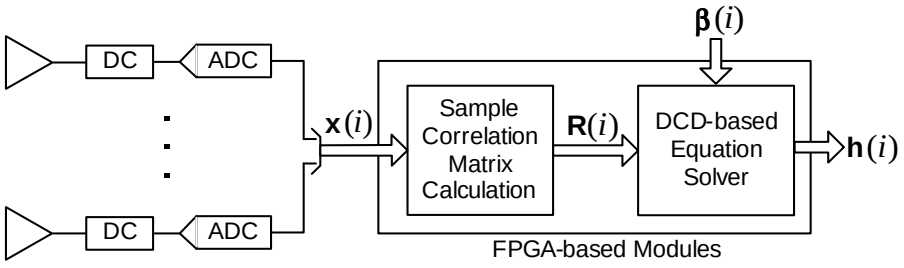


Figure 5.1: Beamforming Topology

According to the MVDR criterion (2.32) [70], the optimal weights $\mathbf{w}(i)$ for the antenna receiving elements at time instant i are given by

$$\mathbf{w}(i) = \frac{\mathbf{R}^{-1}(i)\boldsymbol{\beta}(i)}{\boldsymbol{\beta}^H(i)\mathbf{R}^{-1}(i)\boldsymbol{\beta}(i)}. \tag{5.2}$$

This can be solved by computing

$$\mathbf{h}(i) = \mathbf{R}^{-1}(i)\boldsymbol{\beta}(i), \tag{5.3}$$

which can be represented as the normal equations

$$\mathbf{R}(i)\mathbf{h}(i) = \boldsymbol{\beta}(i). \tag{5.4}$$

The optimal weights $\mathbf{w}(i)$ can then simply be computed as

$$\mathbf{w}(i) = \frac{\mathbf{h}(i)}{\boldsymbol{\beta}^H(i)\mathbf{h}(i)}. \tag{5.5}$$

5.3 FPGA Implementation of MVDR-DCD Beamformer

We have developed and implemented an FPGA core for beamforming as described above based on a Xilinx Virtex-II Pro Development System [77] with an XC2VP30 (FF896 package, speed grade 7) FPGA chip [76]. VHDL is used to describe our core, and it is synthesised and downloaded to the target platform using the Xilinx ISE 8.1i software package. The hardware architecture of the DCD-based MVDR beamformer is shown in Fig. 5.2. The whole implementation operates from a single 100MHz clock and we make use of the FPGA Digital Clock Manager and Global Clock Distribution Network [76] to ensure a uniform delay between the system clock source and each logic slice. For clarity, the clock distribution modules are not shown in Fig. 5.2. The Master State Machine coordinates the operation of the whole system. Three dual-port block RAMs are used to store the matrix $\mathbf{R}(i)$, vector $\boldsymbol{\beta}(i)$ and vector $\mathbf{h}(i)$. An $\mathbf{x}$ RAM used for storing $(K+1)$ snapshots $[\mathbf{x}(i-K), \mathbf{x}(i-K+1), ..., \mathbf{x}(i)]$ is located inside the Transceiver module; therefore it is not shown in Fig. 5.2. In this implementation, we chose $K = 256$. Multiplexers (MUXs) are used for multi-accessing these RAMs. The Transceiver handles the data communication between the FPGA board and a Host Computer. At each time instant, the Transceiver receives the snapshot $\mathbf{x}(i)$ and steering vector $\boldsymbol{\beta}(i)$ from the Host Computer, saves them in the $\mathbf{x}$ RAM and $\boldsymbol{\beta}$ RAM, respectively. After computation, the Transceiver reads the solution vector $\mathbf{h}(i)$ from the $\mathbf{h}$ RAM and transmits it to the Host Computer. The Transceiver also initialises the $\mathbf{h}$ RAM for the next time instant.

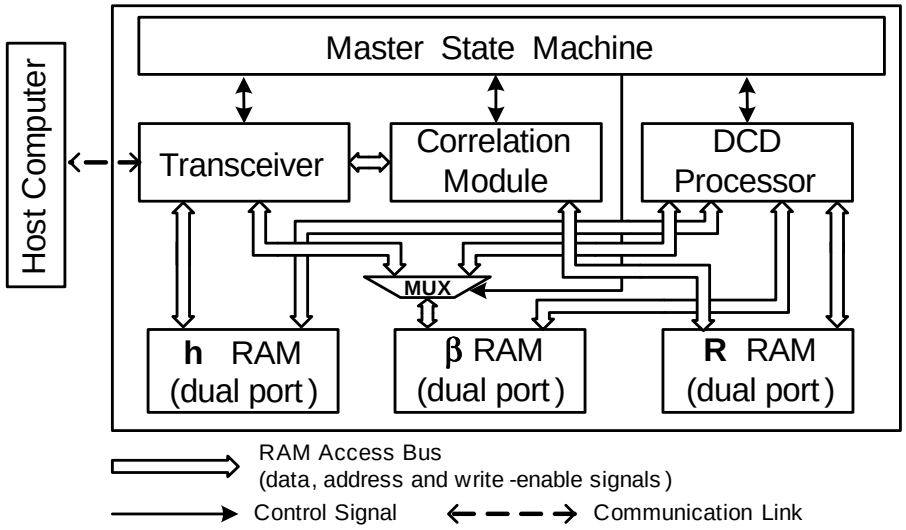


Figure 5.2: FPGA Architecture of the DCD Beamformer.

For representation of the complex-valued input data vector $\mathbf{x}(i)$, the implementation uses 16-bit fixed-point words in the Q15 format [87] for each component. Components of the matrix $\mathbf{R}(i)$ and vectors $\boldsymbol{\beta}(i)$ and $\mathbf{h}(i)$ are 32-bit fixed-point words in the Q15 format. The multiplication $\mathbf{x}(k)\mathbf{x}^H(k)$ results in two 32-bit fixed-point words in the Q30 format for real component and imaginary component, respectively. These two 32-bit Q30 words are extended to 40-bit Q30 format to avoid overflow errors when accumulating $\mathbf{R}_{\text{SUM}}(i) = \sum_{k=i}^{i-K+1} \mathbf{x}(k)\mathbf{x}^H(k)$ as we choose $K = 256 = 2^8$ in current implementation. Each component of $\mathbf{R}_{\text{SUM}}(i)$ is kept in 40-bit Q30 format. Therefore, the matrix $\mathbf{R}(i)$ is obtained by barrel shifting elements of $\mathbf{R}_{\text{SUM}}(i)$ 8 bits right. Considering that $\mathbf{R}_{\text{SUM}}(i)$ is 40-bit Q30 format and $\mathbf{R}(i)$ is 32-bit Q15 format, therefore the barrel shifting operation is performed by choosing the left 17 bits of elements of $\mathbf{R}_{\text{SUM}}(i)$ and then extending the obtained 17-bit Q15 format $\mathbf{R}(i)$ to 32-bit Q15 format.

The architecture of the Correlation Module is shown in Fig. 5.3. The Correlation Module estimates the correlation matrix $\mathbf{R}(i)$ using $K = 256$ snapshots and writes it to the $\mathbf{R}$ RAM at each time instant i. The estimation of matrix $\mathbf{R}(i)$ can be expressed recursively as

$$\mathbf{R}(i) = \frac{1}{K}\mathbf{R}_{\text{SUM}}(i), \tag{5.6}$$

where

$$\mathbf{R}_{\text{SUM}}(i) = \mathbf{R}_{\text{SUM}}(i-1) - \mathbf{x}(i-K)\mathbf{x}^H(i-K) + \mathbf{x}(i)\mathbf{x}^H(i), \tag{5.7}$$

All $(K+1)$ snapshots $[\mathbf{x}(i-K), \mathbf{x}(i-K+1), ..., \mathbf{x}(i)]$ are stored in the $\mathbf{x}$ RAM configured as a circular buffer. At each time instant i, the Transceiver receives the snapshot $\mathbf{x}(i)$ from

the Host Computer and writes it in the space of the snapshot $\mathbf{x}(i-K-1)$, instead of moving all snapshots. The $\mathbf{x}$ RAM Reader A and the $\mathbf{x}$ RAM Reader B assert addresses of $\mathbf{x}(i-K)$ and $\mathbf{x}(i)$ to $\mathbf{x}$ RAM port A and port B, respectively. The MUY1 Writer and MUY2 Writer read elements of $\mathbf{x}(i)$ and $\mathbf{x}(i-K)$ from the both ports of $\mathbf{x}$ RAM, write them to the complex multiplier MUY1 and the complex multiplier MUY2 to compute the upper triangular elements of $\mathbf{x}(i)\mathbf{x}^H(i)$ and $\mathbf{x}(i-K)\mathbf{x}^H(i-K)$, respectively. Both the complex multipliers MUY1 and MUY2 are composed of three 18-bit$\times$18-bit embedded multipliers. The $\mathbf{R}_{\text{SUM}}$ Reader writes addresses to $\mathbf{R}_{\text{SUM}}$ RAM through port B to read the upper triangular part of $\mathbf{R}_{\text{SUM}}(i-1)$. The $\mathbf{R}_{\text{SUM}}$ RAM Writer reads the multiplication results from MUY1 and MUY2 and upper triangular elements of $\mathbf{R}_{\text{SUM}}(i-1)$ from $\mathbf{R}_{\text{SUM}}$ RAM port B, computes the upper triangular elements of $\mathbf{R}_{\text{SUM}}(i)$ following the equation (5.7), and then writes them to $\mathbf{R}_{\text{SUM}}$ RAM through port A. Simultaneously, the $\mathbf{R}_{\text{SUM}}$ RAM Writer reads left 17 bits of $\mathbf{R}_{\text{SUM}}(i)$ (to perform the division by $1/K$ in (5.6)), extends them to 32-bit, and writes the obtained 32-bit Q15 format $\mathbf{R}(i)$ to the $\mathbf{R}$ RAM. All these operations are pipelined under the control of the Correlation Module State Machine. The real and imaginary components are processed simultaneously. Therefore, only one cycle is required for updating each element of the correlation matrix $\mathbf{R}(i)$. Considering that only the upper triangular part of $\mathbf{R}(i)$ is involved, $(N^2+N)/2+6$ cycles are required for updating the correlation matrix $\mathbf{R}(i)$ with 6 cycles of latency.

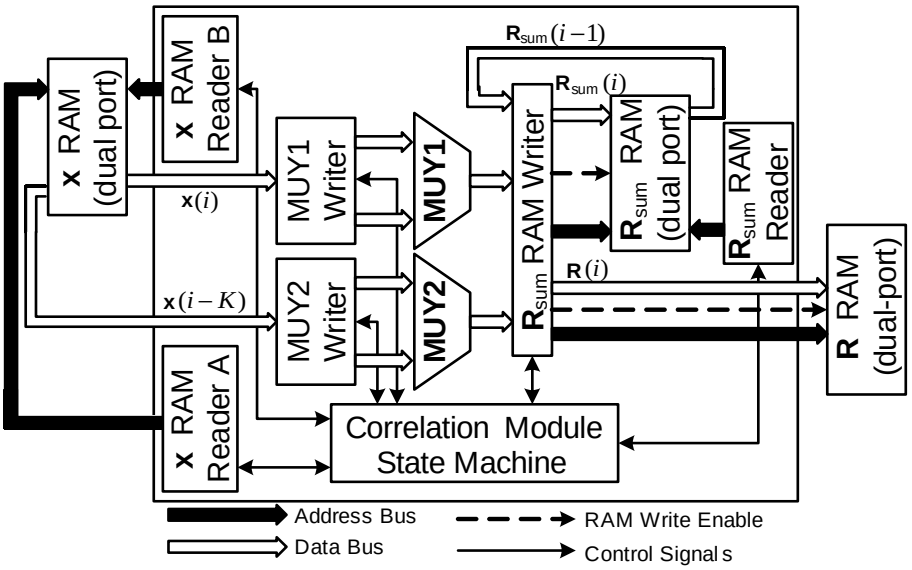


Figure 5.3: FPGA Architecture of the Correlation Module

The DCD Processor uses the DCD algorithm for solving the normal equations. Here, we use the serial implementation of the complex-valued cyclic DCD algorithm described in Section 3.4.1.

Table 5.1: FPGA Requirements for 9-element and 64-element Beamformer

Resources:	$N = 9$	$N = 16$	$N = 32$	$N = 64$
Slices	1491(10.67%)	1491(10.67%)	1685(12.06%)	1750(12.53%)
D-FFs	1183(4.32%)	1185(4.33%)	1234(4.51%)	1282(4.68%)
LUT4s	2276(8.31%)	2271(8.29%)	2640(9.64%)	2656(9.70%)
Block RAMs	18(13.24%)	18(13.24%)	30(22.06%)	70(51.47%)
Block Multipliers	6(4.41%)	6(4.41%)	6(4.41%)	6(4.41%)

5.4 FPGA Resources for the MVDR-DCD Beamformer

The area usage of the Correlation Module and DCD Processor used for implementing 9-element, 16-element, 32-element and 64-element antenna array beamformer are summarized in Table 5.1. The area usage of the 9-element and 16-element are approximately the same; this is because the 9-element implementation is obtained by simply revising the address counters of the 16-element implementation. It shares the same configurations of the block RAMs and same width address-buses and address counters with the 16-element implementation. Due to the fact that all the matrices and vectors are stored in block RAMs, these implementations occupy small number of slices and the increments due to the system size is very small that can be viewed as the width increments of the address-bus and address counters.

However, as there are $(K + 1)$ snapshots stored in the $\mathbf{x}$ RAM and there is an internal $\mathbf{R}_{\text{SUM}}$ RAM, the number of block RAMs is very high and it increases significantly when the number of elements increases. We may consider to use a forgetting factor λ $(0 < \lambda \leq 1)$ to estimate the correlation matrix as

$$\mathbf{R}(i) = \lambda \mathbf{R}(i-1) + \mathbf{x}(i)\mathbf{x}^H(i). \tag{5.8}$$

Thus, only the most recent snapshot $\mathbf{x}(i)$ is required to store in memory and the internal $\mathbf{R}_{\text{SUM}}$ RAM can be eliminated. If we choose a forgetting factor $\lambda = 1 - 2^{-P}$, where P is an integer, then multiplications by λ can be replaced by addition and bit-shift operations. This approach will be used in Chapter 6 in application to adaptive filtering.

5.5 Numerical Results

Our fixed-point FPGA implementation has been tested against a floating-point MVDR beamformer using direct matrix inversion and a floating-point MVDR-DCD beamformer. Our test scenario involved simulating a desired user at -50.53° and two interfering users

Table 5.2: Update Rates for FPGA-based MVDR-DCD Beamformer

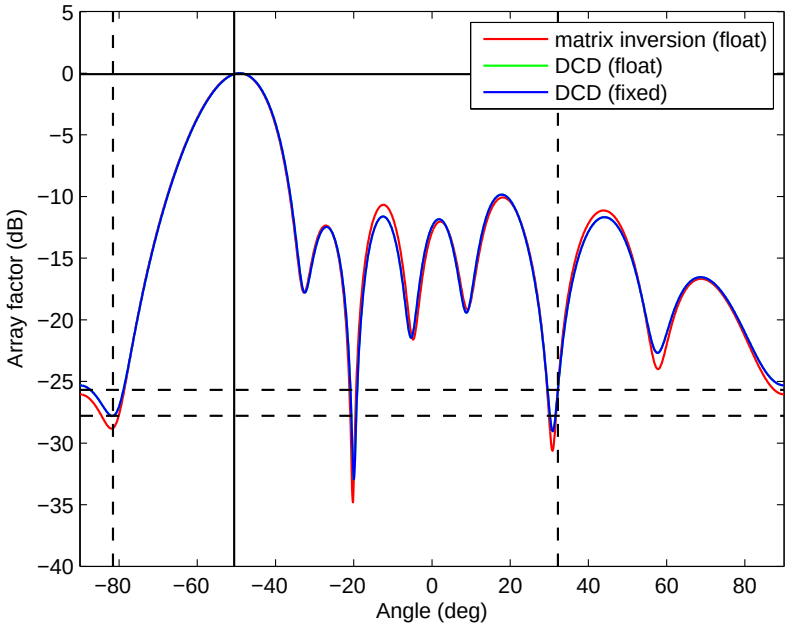
Antenna Elements	Equation Size	Maximum Number of Updates N_u	Worst-case Cycles	Required Cycles	Update Rate (Hz)
9	9×9	400	25,971	5,180	19,305
		500	32,271	8,047	12,427
16	16×16	500	57,359	10,140	9,862
		600	68,559	13,888	7,200
32	32×32	900	204,303	3,3153	3,016
		1,200	271,503	4,9631	2,015
64	64×64	1,500	677,391	103,552	966
		2,500	1,125,391	190,710	524

at angles of -81.59° and $+32.19^\circ$ degrees from the normal axis of the simulated antenna array. The interfering users are at a power level of 0 dB relative to the desired user. We simulate the scenario using 9, 16, 32 and 64 receiving elements, which places the demand of solving 9×9, 16×16, 32×32 and 64×64 complex-valued systems of equations, respectively. The resultant beampatterns are shown in Figs. 5.4 to 5.7. For each antenna array, we provide two beampatterns with different N_u. The solid and dashed vertical lines represent positions of the desired user and interfering users respectively, whilst the horizontal lines highlight the antenna gain for each user.

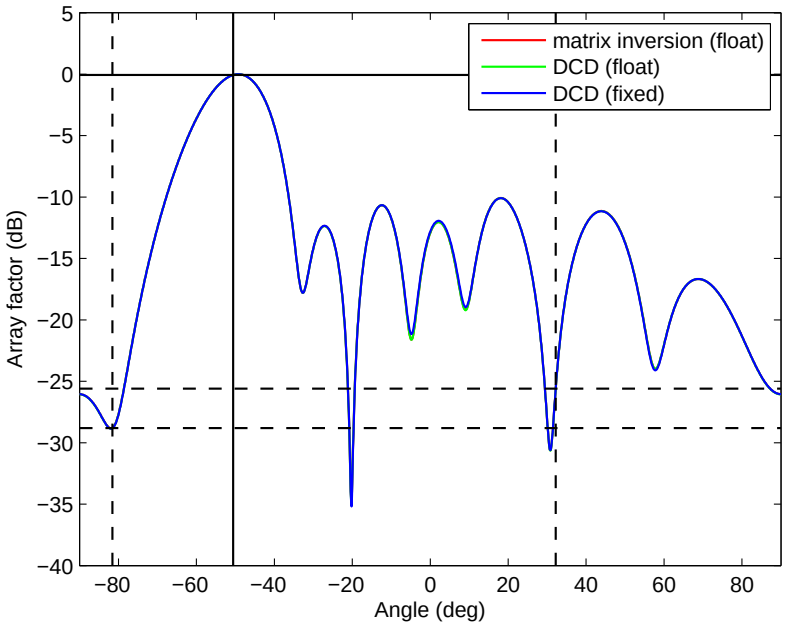
It is seen that for all cases by choosing suitable value of N_u, the DCD-based solver, both floating-point and fixed-point implementations, could obtain approximately the same beampatterns with that of the floating-point direct matrix inversion. However, the throughput of MVDR-DCD implementation is hindered greatly (analyised in Table 5.2) as the DCD processor requires a large number of updates N_u to obtain approximately the same performance as that of the direct matrix inversion. To improve the throughput of MVDR-DCD beamformer, we may consider to reduce the number of updates N_u at the cost of worse beampattern performance.

The number of cycles required by the DCD core to solve the system of equations depends on the system size, system sparseness and condition number of the correlation matrix. The figures in Table 5.2 provide examples of the number of cycles required by the DCD processor in the scenarios described above and give a rough idea of the performance of the beamformer. The worst-case number of cycles required by the DCD processor is also presented, as analyzed in Section 3.4. It is seen that the required number of cycles is much smaller than that for the worst-case scenarios.

Our beamformer implementation compares favourably with an 9-element Altera FPGA-based reference implementation described in [36]. The reference implementation, running from a 150 MHz clock, can sustain a latency of 41700 cycles, whereas

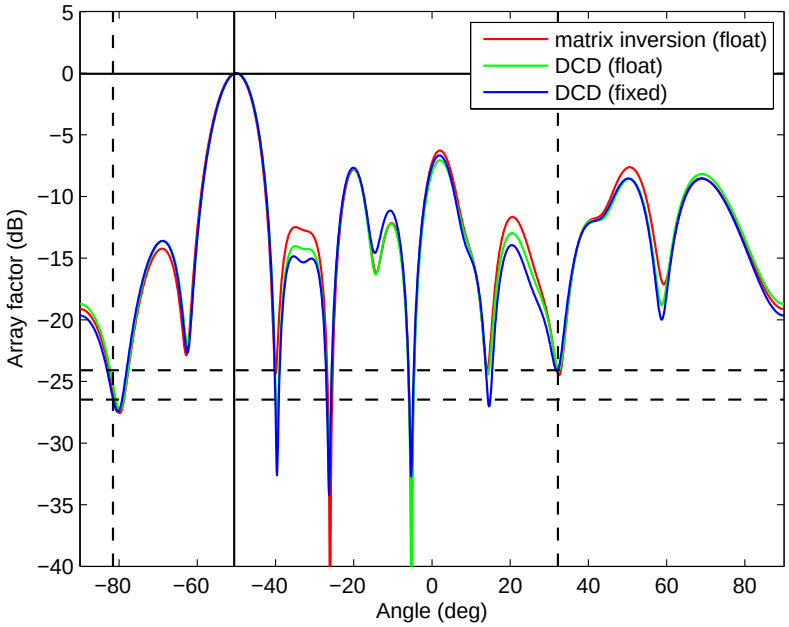


(a)

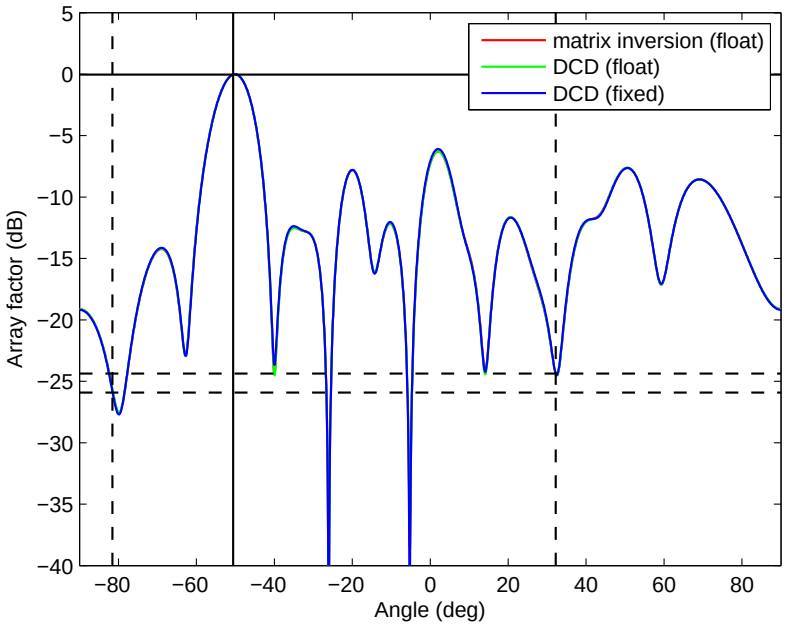


(b)

Figure 5.4: Beampattern for $N = 9$ elements, $M_b = 15$: (a) $N_u = 400$; (b) $N_u = 500$.

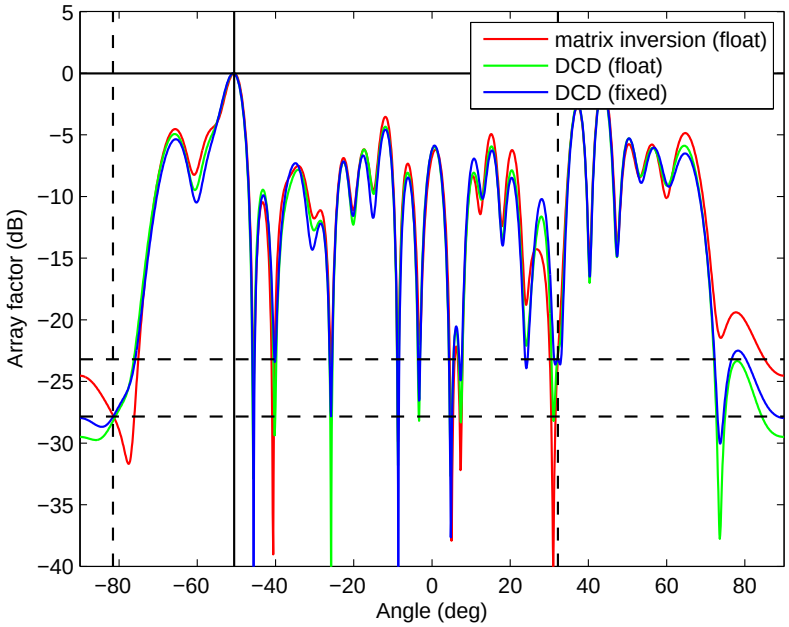


(a)

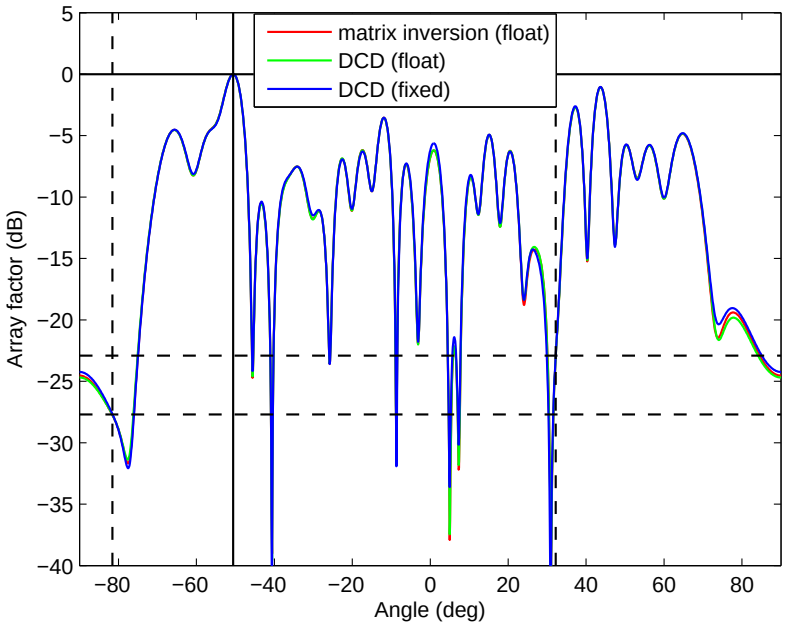


(b)

Figure 5.5: Beampattern for $N = 16$ elements, $M_b = 15$: (a) $N_u = 500$; (b) $N_u = 600$.

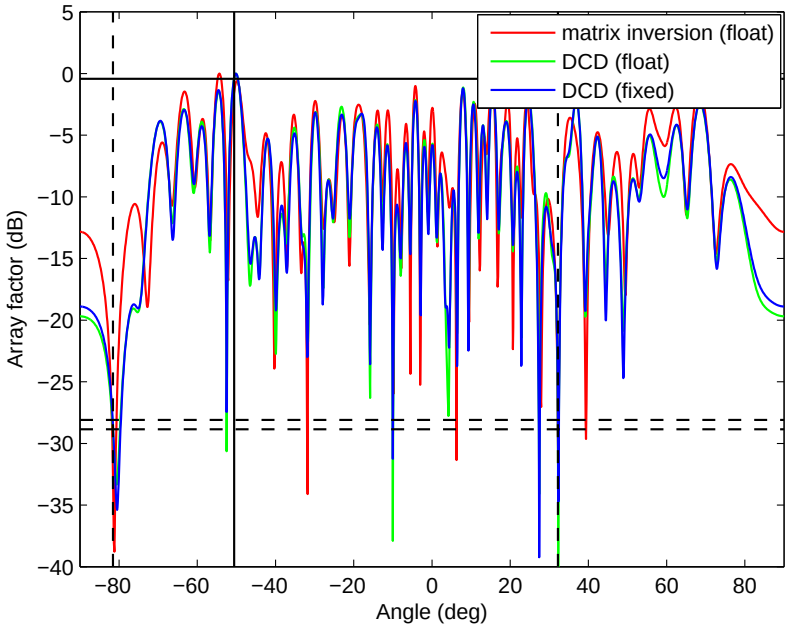


(a)

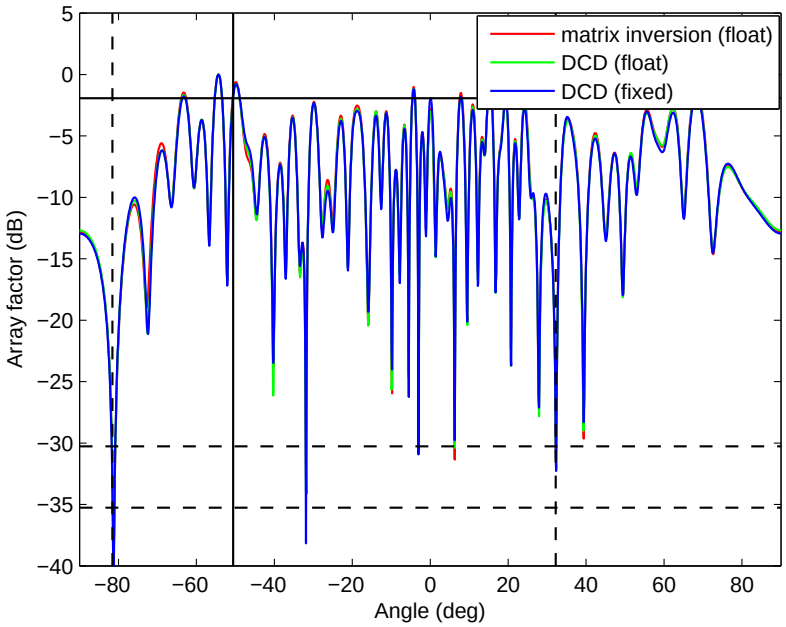


(b)

Figure 5.6: Beampattern for $N = 32$ elements, $M_b = 15$: (a) $N_u = 900$; (b) $N_u = 1200$.



(a)



(b)

Figure 5.7: Beampattern for $N = 64$ elements, $M_b = 15$: (a) $N_u = 1500$; (b) $N_u = 2500$.

our implementation (9-element, $N_u = 500$) sustains a latency of 8047 cycles running on a lower 100MHz clock. This equates to a five-fold throughput increase. The Altera implementation also uses in excess of 2600 logic elements (approximately 1300 Xilinx slices [49,50]), and also requires an additional embedded soft processor. Comparing with a Xilinx CORDIC-based QRD implementation [38], which requires 3530 slices and 13 DSP48 blocks for solving a 9×9 system of equations with a latency of 10971 cycles, our implementation achieves about 1.4 times higher throughput and with approximately 2.3 times smaller chip area.

5.6 MVDR DoA Estimation

The MVDR DoA estimation method minimizes the mean output power of the antenna array subject to unity constraint in the look direction [71]. It is widely used due to its ability of suppressing strong interference. The power spectrum for an angle θ at time instant i can be expressed as [71]

$$P_\theta(i) = \frac{1}{\boldsymbol{\beta}_\theta^H \mathbf{R}^{-1}(i) \boldsymbol{\beta}_\theta}, \tag{5.9}$$

where $\boldsymbol{\beta}_\theta$ is the steering vector of angle θ. At each time instant, by calculating (5.9) for all directions, the power spectrums of all directions can be plotted and assessed for a series of DoA angles in all look directions.

Obviously, the power spectrum $P_\theta(i)$ can be computed as

$$P_\theta(i) = \frac{1}{\boldsymbol{\beta}_\theta^H \mathbf{h}_\theta(i)}, \tag{5.10}$$

where $\mathbf{h}_\theta(i)$ is the solution of the normal equations

$$\mathbf{R}\mathbf{h}_\theta(i) = \boldsymbol{\beta}_\theta. \tag{5.11}$$

Therefore, we could obtain a hardware implementation of MVDR DoA estimator based on DCD iterations by using the implementation of the MVDR-DCD beamformer. However, these computations will be more efficient when using the recursive DCD solver which will be discussed in Chapter 6.

5.7 Conclusions

We have presented an efficient FPGA implementation of the MVDR antenna array beamformer. The FPGA implementation is based on DCD iterations. Comparing with other implementations with the same main parameters, our design achieves higher throughput with smaller chip area. Antenna beampatterns obtained from weights calculated in a fixed-point FPGA platform have been compared with those of a floating-point MVDR-DCD implementation and a floating-point MVDR implementation using direct matrix inversion. For 9-element antenna array, our design obtains about 6 times higher throughput with approximately the same chip area comparing with an Altera CORDIC-based QRD-RLS implementation. Compared to 9-element Xilinx's CORDIC-based QRD implementation, our design achieves about 1.4 times higher throughput and with approximately 2.3 times smaller chip area. The comparison has shown a good match for linear arrays of size 9 to 64 elements. Moreover, by using the same implementation, we could build an MVDR DoA estimator based on DCD iterations.

In this implementation, the correlation matrix $\mathbf{R}(i)$ is estimated using a sample averaging scheme with K snapshots. By updating $\mathbf{R}(i)$ recursively and storing $(K+1)$ snapshots in a RAM configured as a circular buffer, a small number of cycles are required for $\mathbf{R}(i)$ estimation. However, the memory required by the correlation module is very large because all $(K+1)$ snapshots and matrix $\mathbf{R}_{\text{SUM}}$ are kept in RAMs. Moreover, at each time instant, the solution vector $\mathbf{h}(i)$ of this DCD-based MVDR beamformer does not explore the information obtained at previous time instants. Therefore, a large number of DCD iterations may be required to achieve high accuracy.

In the next chapter, a low complexity RLS adaptive filter using DCD iterations is presented. The correlation matrix estimate is based on exponential weighting by using a forgetting factor; therefore the memory requirement for the correlation matrix estimation is significantly reduced. Moreover, at each time instant, the solution vector $\mathbf{h}(i)$ is initialized by the solution vector $\mathbf{h}(i-1)$ obtained at previous time instant $(i-1)$. Therefore, the DCD algorithm will require a smaller number of iterations to obtain high accuracy and the overall throughput will be increased.

Chapter 6

Application: Low Complexity RLS Adaptive Filters using DCD Iterations and their FPGA Implementations

Contents

6.1 Introduction

In adaptive filtering, the RLS algorithm is known to possess fast convergence, but also to have a high complexity of $\mathcal{O}(N^2)$ operations per sample (N being the filter length) [6, 7]. When N is large, the RLS algorithm may become expensive from a hardware implementation point of view [6]. Therefore, there is a strong motivation to reduce the complexity

of the RLS algorithms to have $\mathcal{O}(N)$ arithmetic operations per sample, i.e., similar to the complexity of the LMS algorithm, $2N$ multiplications per sample.

The fast *fixed-order* RLS adaptive filters, exploiting the shifted structure of data vectors (i.e., transversal adaptive filters), have a complexity of $\mathcal{O}(N)$ per sample [7]. The fastest among them in terms of multiplications is the fast Kalman filter that requires $6N$ multiplications per sample [7]. The fixed-order algorithms suffer from numerical instability in finite precision implementation. This problem is partly overcome by using stabilization techniques. However, these make the algorithms more complicated, and, even with such techniques, they can still exhibit instability [7]. Another group of fast adaptive algorithms is the *lattice* algorithms [7]. However, lattice algorithms do not provide the filter weights required in many applications, and their complexity is still high; the techniques considered in [7] require at least $20N$ multiplications and divisions per sample. Recently, the KaGE RLS algorithm was introduced [98]; it uses the shifted structure of data vectors, generates the filter weights, and its complexity is $\mathcal{O}(N \log_2 N)$, more specifically, $13N \log_2 N$ multiplications per sample. However, the KaGE algorithm also requires $\mathcal{O}(N \log_2 N)$ divisions per sample.

Many adaptive algorithms require division and square-root operations, which are complex for implementation, especially in hardware, i.e., they require a significant chip area and high power consumption. Although simpler than divisions, multiplications are still significantly more difficult for implementation than additions. Therefore, it is important to design algorithms that have no division, no square-root operations, and as few multiplications as possible.

Many fast adaptive algorithms are based on matrix inversion which results in instability in finite precision implementation. An alternative approach based on solving the normal equations [99] often results in stable adaptive algorithms. Such an approach is used in the *direction set* (or line search) based adaptive algorithms. These techniques have either a good RLS-like performance but a high complexity of $\mathcal{O}(N^2)$ per sample, e.g., the CG [27, 28, 30, 100] or Euclidean direction search (EDS) [85] adaptive algorithms, or a low complexity of $\mathcal{O}(N)$ per sample but a low performance, e.g., the fast EDS algorithm [85, 101, 102] or the stochastic line search algorithm [103].

The classical RLS adaptive algorithm usually uses an initial regularization to stabilize the solution to the RLS problem [6]. Because the initial regularization decays exponentially in time, we may have to add additional diagonal loading to maintain robustness [104]. However, diagonal loading increases the complexity to $\mathcal{O}(N^3)$ operations per sample as it requires matrix inversion at each time instant [6]. This makes the RLS

algorithm impractical [104]. The leaky RLS adaptive algorithm [73] allows solving the RLS problem with a diagonal loading with complexity of $\mathcal{O}(N^2)$. It is based on using a recursive update of the eigenvalue decomposition of the correlation matrix. However, the eigenvalue decomposition is complicated for real-time implementation.

In this chapter, we express the RLS adaptive filtering problem in terms of auxiliary normal equations with respect to increments of the filter weights. The normal equations are then approximately solved by using the low complexity DCD algorithm [5]. Therefore, the obtained RLS-DCD algorithm is well suited to hardware implementation and the complexity is as low as $3N$ multiplications per sample for transversal filtering problem. Moreover, it results in a stable finite precision implementation. The performance of the RLS-DCD algorithm can be made arbitrarily close to that of the classical RLS algorithm. However, the RLS-DCD algorithm does not allow the regularization to be used except the initial regularization that is used in the classical RLS algorithm. A low complexity dynamically regularized RLS adaptive filtering algorithm based on the RLS-DCD algorithm is also proposed. The dynamically regularized RLS-DCD algorithm reduces the complexity of the regularized RLS problem to $\mathcal{O}(N^2)$ in a general case, and to $\mathcal{O}(N)$ for transversal adaptive filters. The derivation of this algorithm mostly follows the steps of derivation of the RLS-DCD algorithm. A fixed-point FPGA implementation of this dynamically regularized RLS-DCD algorithm is also presented.

The main body of this chapter is organized as follows. In Section 6.2, the RLS-DCD algorithm is introduced and applied to the exponentially weighted RLS filtering case with unstructured data vector and transversal RLS filtering case with time-shifted structured data vector in Section 6.2.1 and Section 6.2.2, respectively. In Section 6.3, the dynamically regularized RLS-DCD algorithm for solving complex-valued systems is described. In Section 6.4 and Section 6.5, fixed-point FPGA implementations of RLS-DCD algorithm and dynamically regularized RLS-DCD algorithm are described, respectively. Section 6.6 and Section 6.7 provide numerical results for the RLS-DCD and dynamically regularized RLS-DCD algorithms, respectively. Finally, Section 6.8 gives conclusions.

6.2 RLS-DCD Adaptive Filtering Algorithms

In the RLS problem, at every time instant i ($i = 0, 1, 2, \ldots$), an adaptive algorithm should find a solution to the normal equations

$$\mathbf{R}(i)\mathbf{h}(i) = \boldsymbol{\beta}(i), \tag{6.1}$$

where $\mathbf{R}(i)$ is assumed to be a symmetric positive-definite (correlation) matrix of size $N \times N$, $\boldsymbol{\beta}(i)$ and $\mathbf{h}(i)$ are N- length vectors. The matrix $\mathbf{R}(i)$ and vector $\boldsymbol{\beta}(i)$ are known, whereas the vector $\mathbf{h}(i)$ should be estimated. Direct methods for solving the system are too complex for most applications of adaptive filtering, especially if N is high; e.g., the Cholesky decomposition finds the solution with a complexity $\mathcal{O}(N^3)$ [1]. In the classical RLS algorithm, the solution is represented in the form [7]: $\mathbf{h}(i) = \mathbf{P}(i)\boldsymbol{\beta}(i)$, where $\mathbf{P}(i) = \mathbf{R}^{-1}(i)$; $\mathbf{P}(i)$ can be computed recursively with a complexity of $\mathcal{O}(N^2)$ [7]. The RLS-DCD algorithm adopts another approach, which is based on transforming the original sequence of normal equations (6.1) into a sequence of auxiliary normal equations that are then solved by using iterative techniques.

Let, at time instant $(i-1)$, a system of equations $\mathbf{R}(i-1)\mathbf{h}(i-1) = \boldsymbol{\beta}(i-1)$ be approximately solved, and the approximate solution is $\hat{\mathbf{h}}(i-1)$. Let

$$\mathbf{r}(i-1) = \boldsymbol{\beta}(i-1) - \mathbf{R}(i-1)\hat{\mathbf{h}}(i-1) \tag{6.2}$$

be a residual vector for this solution. At time instant i, the system (6.1) is to be solved. We denote $\Delta\mathbf{R}(i) = \mathbf{R}(i) - \mathbf{R}(i-1)$, $\Delta\boldsymbol{\beta}(i) = \boldsymbol{\beta}(i) - \boldsymbol{\beta}(i-1)$, and

$$\Delta\mathbf{h}(i) = \mathbf{h}(i) - \hat{\mathbf{h}}(i-1). \tag{6.3}$$

To find a solution $\hat{\mathbf{h}}(i)$ of the system (6.1) by exploiting the previously obtained solution $\hat{\mathbf{h}}(i-1)$ and residual vector $\mathbf{r}(i-1)$, the system (6.1) can be rewritten as

$$\mathbf{R}(i)[\hat{\mathbf{h}}(i-1) + \Delta\mathbf{h}(i)] = \boldsymbol{\beta}(i) \tag{6.4}$$

and represented as a system of equations with respect to the unknown vector $\Delta\mathbf{h}(i)$:

$$\begin{aligned} \mathbf{R}(i)\Delta\mathbf{h}(i) &= \boldsymbol{\beta}(i) - \mathbf{R}(i)\hat{\mathbf{h}}(i-1) \\ &= \boldsymbol{\beta}(i) - \mathbf{R}(i-1)\hat{\mathbf{h}}(i-1) - \Delta\mathbf{R}(i)\hat{\mathbf{h}}(i-1) \\ &= \mathbf{r}(i-1) + \Delta\boldsymbol{\beta}(i) - \Delta\mathbf{R}(i)\hat{\mathbf{h}}(i-1). \end{aligned} \tag{6.5}$$

Instead of solving the original problem (6.1), one can find a solution $\Delta\hat{\mathbf{h}}(i)$ to the auxiliary system of equations

$$\mathbf{R}(i)\Delta\mathbf{h}(i) = \boldsymbol{\beta}_0(i), \tag{6.6}$$

where

$$\boldsymbol{\beta}_0(i) = \mathbf{r}(i-1) + \Delta\boldsymbol{\beta}(i) - \Delta\mathbf{R}(i)\hat{\mathbf{h}}(i-1), \tag{6.7}$$

and obtain an approximate solution of the original system (6.1) as

$$\hat{\mathbf{h}}(i) = \hat{\mathbf{h}}(i-1) + \Delta\hat{\mathbf{h}}(i). \tag{6.8}$$

Table 6.1: Recursively solving a sequence of systems of equations

Step	Equation
	Initialization: $\mathbf{r}(-1) = \mathbf{0}$,$\boldsymbol{\beta}(-1) = \mathbf{0}$,$\hat{\mathbf{h}}(-1) = \mathbf{0}$
	for $i = 0, 1, \ldots$
1	Find $\Delta\boldsymbol{R}(i)$ and $\Delta\boldsymbol{\beta}(i)$
2	$\boldsymbol{\beta}_0(i) = \mathbf{r}(i-1) + \Delta\boldsymbol{\beta}(i) - \Delta\mathbf{R}(i)\hat{\mathbf{h}}(i-1)$
3	Solve $\mathbf{R}(i)\Delta\mathbf{h} = \boldsymbol{\beta}_0(i) \Rightarrow \Delta\hat{\mathbf{h}}(i),\ \mathbf{r}(i)$
4	$\hat{\mathbf{h}}(i) = \hat{\mathbf{h}}(i-1) + \Delta\hat{\mathbf{h}}(i)$

It is seen from (6.7) that this approach requires the residual vector $\mathbf{r}(i)$ for the solution $\hat{\mathbf{h}}(i)$ to the original system (6.1) to be known at each time instant i. After some algebra, we obtain that the residual vector for the solution $\Delta\hat{\mathbf{h}}(i)$ to the auxiliary system (6.6) is also equal to $\mathbf{r}(i)$, i.e.,

$$\begin{aligned} \mathbf{r}(i) &= \boldsymbol{\beta}(i) - \mathbf{R}(i)\hat{\mathbf{h}}(i) && (6.9) \\ &= \boldsymbol{\beta}_0(i) - \mathbf{R}(i)\Delta\hat{\mathbf{h}}(i). && (6.10) \end{aligned}$$

Thus, we can now formulate a recursive approach for solving a sequence of systems of equations as presented in Table 6.1.

This approach allows us, at each time instant i, instead of solving the original problem (6.1) with respect to the filter weights $\mathbf{h}(i)$, to deal with an auxiliary problem (6.6) with respect to the increment of the filter weights $\Delta\mathbf{h}(i)$. The system (6.6) takes into account the accuracy of the previous solution through the residual vector $\mathbf{r}(i-1)$, as well as the variation of the problem to be solved through the increments $\Delta\mathbf{R}(i)$ and $\Delta\boldsymbol{\beta}(i)$. If a true solution to the system (6.6) is found then $\hat{\mathbf{h}}(i)$ is the true solution to the problem (6.1) as well.

When using direct methods for solving the normal equations, both approaches would require approximately the same computational load. However, when using iterative techniques, the new approach is preferable, since it corresponds to solving the original problem with (implicit) initialization by the solution of the problem for the previous time instant. Therefore, with the same accuracy of calculating the vector $\mathbf{h}(i)$, the proposed approach will typically require a smaller number of iterations. If, in addition to finding a solution vector $\Delta\hat{\mathbf{h}}(i)$, the iterative equation solver produces the residual vector $\mathbf{r}(i)$ at a low computational cost, and a simple way of computing the product $\mathbf{R}(i)\Delta\hat{\mathbf{h}}(i)$ also exists, the complexity of adaptive filtering based on the new approach will be lower than that with the original approach. There are many iterative techniques, such as the CG algorithm, the CD algorithm and the DCD algorithm as discussed in Chapter 3, that can be used to solve the system (6.6). We call the obtained algorithms as RLS-CG, RLS-CD and RLS-DCD algorithms, respectively. The numerical performance and complexity of these

Table 6.2: Leading DCD algorithm

Step	Equation	+
	Initialisation: $\Delta\hat{\mathbf{h}}(i)=0$, $\mathbf{r}(i)=\boldsymbol{\beta}_0(i)$, $\alpha=H/2$, $m=1$	
	for $k=1,\ldots,N_u$	
1	$n=\arg\max_{p=1,\ldots,N}\{\lvert r_p(i)\rvert\}$, go to step 4	$N-1$
2	$m=m+1$, $\alpha=\alpha/2$	0
3	if $m>M_b$, the algorithm stops	0
4	if $\lvert r_n(i)\rvert\le(\alpha/2)R_{n,n}(i)$, then go to step 2	1
5	$\Delta\hat{h}_n(i)=\Delta\hat{h}_n(i)+\text{sign}[r_n(i)]\alpha$	1
6	$\mathbf{r}(i)=\mathbf{r}(i)-\text{sign}[r_n(i)]\alpha\mathbf{R}_{:,n}(i)$	N
Total:	$\le(2N+1)N_u+M_b$ adds	

algorithms will be compared in Section 6.6.

The DCD algorithm provides both a solution $\Delta\hat{\mathbf{h}}(i)$ and the residual vector $\mathbf{r}(i)$. As we analyzed in Chapter 3, the leading DCD is preferable for solving the adaptive filtering problem due its fast convergence at initial iterations. The real-valued leading DCD algorithm is represented here in Table. 6.2 for the time-varying system (6.6). Below, the RLS-DCD algorithm is applied to the exponentially weighted RLS filter and transversal RLS filter problems. It can also apply to the sliding window RLS problem as shown in [92].

6.2.1 Exponentially Weighted RLS-DCD Algorithm

The exponentially weighted RLS (ERLS) problem deals, at every time instant i, with a N-length data vector $\mathbf{x}(i)$ and a scalar desired signal $d(i)$. An adaptive algorithm should find a vector $\mathbf{h}(i)$ that minimizes the error [7]

$$E_{\mathbf{e}}(i)=\lambda^{i+1}\mathbf{h}^T(i)\mathbf{\Pi}\mathbf{h}(i)+\sum_{j=0}^{i}\lambda^{i-j}[d(j)-\mathbf{h}^T(i)\mathbf{x}(j)]^2 \tag{6.11}$$

where $\mathbf{\Pi}$ is a regularization matrix and $0<\lambda\le 1$ is a forgetting factor. The regularization matrix is usually chosen as a diagonal matrix $\mathbf{\Pi}=\eta\mathbf{I}$, where the regularization parameter $\eta>0$ is a small positive number and $\mathbf{I}$ is the $N\times N$ identity matrix [6, 7]. The vector $\mathbf{h}(i)$ can be found by solving the normal equations (6.1) with the system matrix and the right-hand vector given by [6]

$$\mathbf{R}(i) = \lambda\mathbf{R}(i-1)+\mathbf{x}(i)\mathbf{x}^T(i), \tag{6.12}$$

$$\boldsymbol{\beta}(i) = \lambda\boldsymbol{\beta}(i-1)+d(i)\mathbf{x}(i). \tag{6.13}$$

To apply the method in Table 6.1 to this problem, the vector $\boldsymbol{\beta}_0(i)$ should be expressed in terms of $\mathbf{x}(i)$ and $d(i)$. From (6.12) and (6.13), we obtain

$$\Delta\mathbf{R}(i) = (\lambda-1)\mathbf{R}(i-1) + \mathbf{x}(i)\mathbf{x}^T(i), \tag{6.14}$$

$$\Delta\boldsymbol{\beta}(i) = (\lambda-1)\boldsymbol{\beta}(i-1) + d(i)\mathbf{x}(i). \tag{6.15}$$

By using (6.2) and (6.14), we obtain

$$\Delta\mathbf{R}(i)\hat{\mathbf{h}}(i-1) = (\lambda-1)[\boldsymbol{\beta}(i-1) - \mathbf{r}(i-1)] + \mathbf{x}(i)y(i) \tag{6.16}$$

where $y(i)$ is the adaptive filter output at time instant i,

$$y(i) = \mathbf{x}^T(i)\hat{\mathbf{h}}(i-1). \tag{6.17}$$

Using (6.16), we obtain step 2 for the method in Table 6.1:

$$\boldsymbol{\beta}_0(i) = \lambda\mathbf{r}(i-1) + e(i)\mathbf{x}(i), \tag{6.18}$$

where $e(i)$ is the *a priori* estimation error,

$$e(i) = d(i) - y(i). \tag{6.19}$$

Finally, the exponentially weighted RLS algorithm is summarized in Table 6.3, which also shows the complexity of different steps of the algorithm in terms of multiplications and additions. Note that due to the symmetry of $\mathbf{R}(i)$ only its upper triangle part is calculated to reduce the complexity. Moreover, the forgetting factor λ is chosen as $\lambda = 1-2^{-P}$, where P is a positive integer number. Therefore, the multiplications by λ can be replaced by bit-shifts and additions, thus reducing the number of multiplications that are significantly more complicated for implementation than additions and bit-shifts.

The auxiliary systems are solved using the real-valued leading DCD algorithm, with a complexity of $P_m = 0$ multiplications and $P_a = (2N+1)N_u + M_b$ additions and M_b is the number of bits used for a fixed-point representation of elements of the solution vector, where N_u denotes the number of updates, as discussed in Section 3.2.2. Thus, the complexity of the exponentially weighted RLS-DCD (ERLS-DCD) algorithm is $(N^2+5N)/2$ multiplications and $N^2+4N+(2N+1)N_u+M_b$ additions. We can also use other iterative techniques, such as CG and CD algorithms, to solve the exponentially weighted RLS problem. The computational complexity and numerical results of the obtained exponentially weighted RLS-CG (ERLS-CG) and exponentially weighted RLS-CD (ERLS-CD) will be analyzed and compared to that of the ERLS-DCD, classical RLS algorithm and NLMS algorithm in Section 6.6.

Table 6.3: Exponentially weighted RLS (ERLS) algorithm

Step	Equation	$\times$	$+$
	Initialization: $\hat{\mathbf{h}}(-1)=0, \mathbf{r}(-1)=0, \mathbf{R}(-1)=\mathbf{\Pi}$		
	for $i=0,1,\ldots$		
1	$\mathbf{R}(i)=\lambda\mathbf{R}(i-1)+\mathbf{x}(i)\mathbf{x}^T(i)$	$N(N+1)/2$	$N(N+1)$
2	$y(i)=\mathbf{x}^T(i)\hat{\mathbf{h}}(i-1)$	N	$N-1$
3	$e(i)=d(i)-y(i)$	0	1
4	$\boldsymbol{\beta}_0(i)=\lambda\mathbf{r}(i-1)+e(i)\mathbf{x}(i)$	N	$2N$
5	$\mathbf{R}(i)\Delta\mathbf{h}(i)=\boldsymbol{\beta}_0(i)\Rightarrow\Delta\hat{\mathbf{h}}(i),\mathbf{r}(i)$	P_m	P_a
6	$\hat{\mathbf{h}}(i)=\hat{\mathbf{h}}(i-1)+\Delta\hat{\mathbf{h}}(i)$	0	N
Total:	mults$=(N^2+5N)/2+P_m$; adds $\leq N^2+4N+P_a$		

6.2.2 Transversal RLS-DCD Algorithm

The exponentially weighted RLS algorithm as presented in Table 6.3 can be used in applications with arbitrary data vectors (regressors) $\mathbf{x}(i)$. One of such applications is the antenna array beamforming [6, 7]. In other applications, e.g., in echo cancellation, equalization and noise reduction [6], the regressors have a time-shifted structure

$$\mathbf{x}(i)=[x(i-N+1)\ \ldots\ x(i-1)\ x(i)]^T, \tag{6.20}$$

where $x(i)$ is a discrete-time signal. In this case, updating the correlation matrix $\mathbf{R}(i)$ is significantly simplified. Specifically, the upper-left $(N-1)\times(N-1)$ block of $\mathbf{R}(i)$ can be obtained by copying the lower-right $(N-1)\times(N-1)$ block of $\mathbf{R}(i-1)$. The only part of the $\mathbf{R}(i)$ that should be directly updated is the last column:

$$\mathbf{R}_{:,N}(i)=\lambda\mathbf{R}_{:,N}(i-1)+x(i)\mathbf{x}(i). \tag{6.21}$$

As a result, the number of multiplications and additions at step 1 is reduced to N and $2N$, respectively. The transversal filter RLS algorithm is shown in Table 6.4. For the transversal RLS-DCD algorithm, the total number of multiplications and additions are reduced to $3N$ and $6N+(2N+1)N_u+M_b$, respectively.

Table 6.4: Transversal RLS algorithm

Step	Equation	$\times$	$+$
	Initialization: $\hat{\mathbf{h}}(-1)=0, \mathbf{r}(-1)=0, \mathbf{R}(-1)=\mathbf{\Pi}$		
	for $i=0,1,\ldots$		
1	$\mathbf{R}_{:,N}(i)=\lambda\mathbf{R}_{:,N}(i-1)+x(i)\mathbf{x}^T(i)$	N	$2N$
2	$y(i)=\mathbf{x}^T(i)\hat{\mathbf{h}}(i-1)$	N	$N-1$
3	$e(i)=d(i)-y(i)$	0	1
4	$\boldsymbol{\beta}_0(i)=\lambda\mathbf{r}(i-1)+e(i)\mathbf{x}(i)$	N	$2N$
5	$\mathbf{R}(i)\Delta\mathbf{h}(i)=\boldsymbol{\beta}_0(i)\Rightarrow\Delta\hat{\mathbf{h}}(i), \mathbf{r}(i)$	P_m	P_a
6	$\hat{\mathbf{h}}(i)=\hat{\mathbf{h}}(i-1)+\Delta\hat{\mathbf{h}}(i)$	0	N
Total:	mult $=3N+P_m$; adds $\leq 6N+P_a$		

6.3 Dynamically Regularized RLS-DCD Adaptive Filtering Algorithm

In the previous section, the RLS-DCD algorithm is applied to real-valued systems. For the complex-valued systems, the matrix $\mathbf{R}(i)$ is calculated as

$$\mathbf{R}(i)=\sum_{k=0}^{i}\lambda^{i-k}\mathbf{x}(k)\mathbf{x}^H(k)+\delta(i)\mathbf{I}, \tag{6.22}$$

where $\mathbf{x}(i)$ is an N-length complex-valued input data vector, λ is a forgetting factor $0<\lambda\leq 1$, $\delta(i)$ is a time-varying diagonal loading and $\mathbf{I}$ is the $N\times N$ identity matrix [6]. In the classical RLS algorithm and in the RLS-DCD algorithm, an initial regularization $\mathbf{R}(-1)=\eta\mathbf{I}$ is used, $\eta>0$, and the matrix $\mathbf{R}(i)$ is updated as [6]

$$\mathbf{R}(i)=\lambda\mathbf{R}(i-1)+\mathbf{x}(k)\mathbf{x}^H(k). \tag{6.23}$$

This corresponds to the time varying regularization $\delta(i)=\lambda^{i+1}\eta$ which exponentially decays in time. However, it is often of interest to add an extra diagonal loading to maintain the algorithm robustness. Moreover, in some applications [105] this diagonal loading can vary in time. We consider that the regularization parameter $\delta(i)$ may vary in time without restarting the adaptive filter [105], resulting in a dynamically regularized RLS algorithm.

From equation (6.22), we obtain

$$\mathbf{R}(i)=\lambda\mathbf{R}(i-1)+\mathbf{x}(i)\mathbf{x}^H(i)+[\delta(i)-\lambda\delta(i-1)]\mathbf{I}, \tag{6.24}$$

and

$$\Delta\mathbf{R}(i)=(\lambda-1)\mathbf{R}(i-1)+\mathbf{x}(i)\mathbf{x}^H(i)+[\delta(i)-\lambda\delta(i-1)]\mathbf{I}. \tag{6.25}$$

Table 6.5: Dynamically regularized RLS algorithm

Step	Equation	$\times$	$+$
	Initialisation: $\hat{\mathbf{h}}(-1)=0, \mathbf{r}(-1)=0$ $\mathbf{R}(-1)=\eta\mathbf{I}$		
	for $i=0,1,\ldots$		
1	$\mathbf{R}(i)=\lambda\mathbf{R}(i-1)+\mathbf{x}(i)\mathbf{x}^H(i)$ $+[\delta(i)-\lambda\delta(i-1)]\mathbf{I}$	$2N^2+2N$	$3N^2+4N+2$
2	$y(i)=\mathbf{x}^H(i)\hat{\mathbf{h}}(i-1)$	$4N$	$4N-2$
3	$\boldsymbol{\beta}_0(i)=\lambda\mathbf{r}(i-1)+(1-\lambda)\boldsymbol{\beta}(i)$ $+\Delta\boldsymbol{\beta}(i)-\mathbf{x}(i)y(i)$ $-[\delta(i)-\lambda\delta(i-1)]\hat{\mathbf{h}}(i-1)$	$8N$	$12N+2$
4	$\mathbf{R}(i)\Delta\mathbf{h}(i)=\boldsymbol{\beta}_0(i)\Longrightarrow\Delta\hat{\mathbf{h}}(i),\mathbf{r}(i)$	P_m	P_a
5	$\hat{\mathbf{h}}(i)=\hat{\mathbf{h}}(i-1)+\Delta\hat{\mathbf{h}}(i)$	$-$	$2N$
Total:	mults=$2N^2+14N+P_m$; adds=$3N^2+22N+P_a+2$		

Thus, the auxiliary vector $\boldsymbol{\beta}_0(i)$ of step 2 in Table 6.1 can be represented as

$$\boldsymbol{\beta}_0(i)=\lambda\mathbf{r}(i-1)+(1-\lambda)\boldsymbol{\beta}(i)+\Delta\boldsymbol{\beta}(i)-\mathbf{x}(i)y(i)-[\delta(i)-\lambda\delta(i-1)]\hat{\mathbf{h}}(i-1), \quad (6.26)$$

where $y(i)=\mathbf{x}^H(i)\hat{\mathbf{h}}(i-1)$ is the filter output at time instant i.

Finally, the dynamically regularized RLS algorithm is summarized in Table 6.5. The complexity of each step is measured in terms of real-valued multiplications and additions. Similarly to the unregularized RLS-DCD algorithm in Table 6.3 and Table 6.4, multiplications by λ and $(1-\lambda)$ are replaced by addition and bit-shift operations as $\lambda=1-2^{-P}$ with a positive integer P and we only compute the upper triangular part of the symmetric correlation matrix $\mathbf{R}(i)$. The complex-valued auxiliary equations are solved using the complex-valued leading DCD algorithm presented in Chapter 3. The complex-valued leading DCD algorithm is presented in Table 6.6 for time-varying complex-valued system (6.6) with a worst-case complexity of $P_a=(4N+1)N_u+M_b$ real-valued additions and $P_m=0$ multiplication. Thus, the complexity of this proposed algorithm for an N-size complex-valued system is $2N^2+14N$ real-valued multiplications and $3N^2+22N+(4N+1)N_u+M_b+2$ real-valued additions.

The dynamically regularized RLS-DCD algorithm in Table 6.5 can be used in applications with arbitrary data structures. In a similar way, we can obtain a dynamically regularized transversal RLS-DCD algorithm. The complexity of this algorithm is $\mathcal{O}(N)$ arithmetic operations per sample.

Table 6.6: Complex-valued Leading DCD Algorithm ($N \times N$ system)

Step	Equation	$+$
	Initialization: $\Delta\hat{\mathbf{h}}(i) = 0$, $\mathbf{r}(i) = \boldsymbol{\beta}_0(i)$, $\alpha = H$, $m = 0$	
	for $k = 1, \ldots, N_u$	
1	$[n, s] = \arg\max_{p=1,\ldots,N}\{\lvert\Re(r_p(i))\rvert, \lvert\Im(r_p(i))\rvert\}$ go to step 4	$2N-1$
2	$m = m + 1$, $\alpha = \alpha/2$	
3	if $m > M_b$, algorithm stops	
4	if $s = 1$, then $r_{tmp} = \Re(r_n(i))$, else $r_{tmp} = \Im(r_n(i))$ if $\lvert r_{tmp}\rvert \leq (\alpha/2)R_{n,n}(i)$, then go to step 2	1
5	$\Delta\hat{h}_n(i) = \Delta\hat{h}_n(i) + \text{sign}(r_{tmp})s\alpha$	1
6	$\mathbf{r}(i) = \mathbf{r}(i) - \text{sign}(r_{tmp})s\alpha\mathbf{R}_{:,n}(i)$	$2N$
Total:	$\leq (4N + 1)N_u + M_b$ adds	

6.4 FPGA Implementation of RLS-DCD Adaptive Filtering Algorithms

The hardware architecture of the RLS-DCD algorithm is shown in Fig. 6.1. The whole implementation operates from a single 100MHz clock and we make use of the FPGA Digital Clock Manager and Global Clock Distribution Network [76] to ensure a uniform delay between the system clock source and each logic slice. For clarity, the clock distribution modules are not shown in the Fig. 6.1. The Master State Machine coordinates the operation of the whole system. The Transceiver handles the data communication between the FPGA board and a host computer. Four dual-port block RAMs are used to store the matrix $\mathbf{R}(i)$, the vector $\boldsymbol{\beta}_0(i)$, the vector $\hat{\mathbf{h}}(i)$ and the input data $\mathbf{x}(i)$. The scalar $d(i)$ is stored in a register. The $\mathbf{x}$ RAM used for storing vectors $\mathbf{x}(i)$ is located inside the Transceiver module; therefore it is not shown in Fig. 6.1. Multiplexers (MUXs) are used for multi-accessing these RAMs. The Correlation Module updates the matrix $\mathbf{R}(i)$ and calculates the vector $\boldsymbol{\beta}_0(i)$ according to steps 1 to 4 of Table 6.3. The DCD Processor uses the DCD algorithm for solving the normal equations and generating the residual vector $\mathbf{r}(i)$ at step 5 of Table 6.3; step 6 is also incorporated in the DCD Processor.

For representation of the input data $\mathbf{x}(i)$, the implementation uses 16-bit fixed-point words in the Q15 format [87]. The samples $d(i)$ are represented by 32-bit fixed-point words in the Q15 format. Elements of the matrix $\mathbf{R}(i)$ and vectors $\mathbf{r}(i)$, $\boldsymbol{\beta}_0(i)$, and $\hat{\mathbf{h}}(i)$ are 32-bit fixed-point words in the Q15 format, which are same as the previous implementations of DCD algorithms. When computing the filter output $y(i)$, each multiplication results in a 47-bit fixed-point word in the Q30 format; after the accumulation of the N products, $y(i)$ is truncated to a 32-bit fixed-point word in the Q15 format. The error signal

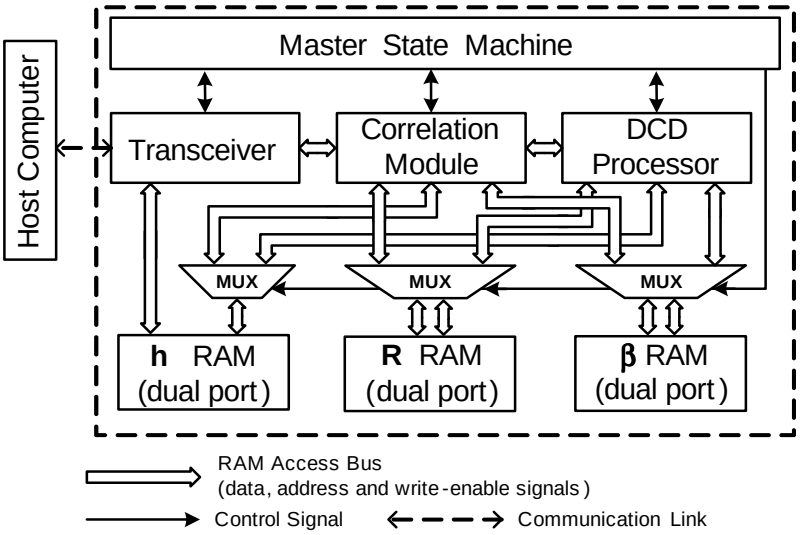


Figure 6.1: Block-diagram of the FPGA implementation of the RLS-DCD adaptive filtering algorithm.

$e(i)$ is represented by 32-bit fixed-point words in the Q15 format.

6.4.1 FPGA Implementation for Arbitrary Data Vectors

The Correlation Module is shown in Fig. 6.2. It initializes the $\mathbf{R}$ RAM according to the initialization step in Table 6.3. For updating the matrix $\mathbf{R}(i)$, the $\mathbf{x}$ RAM Reader B writes addresses to the $\mathbf{x}$ RAM through port B. The MUY1 Writer reads elements of $\mathbf{x}(i)$ and writes them into both operand ports of the multiplier MUY1 to produce upper triangular elements of the vector product $\mathbf{x}(i)\mathbf{x}^T(i)$. The MUY1 is a 16-bit$\times$16-bit multiplier. The $\mathbf{R}$ RAM Reader writes addresses to the $\mathbf{R}$ RAM to read upper triangular elements of the matrix $\mathbf{R}(i-1)$. The $\mathbf{R}$ RAM Writer reads the elements of $\mathbf{R}(i-1)$ and multiplication results from the MUY1, computes upper triangle elements of $\mathbf{R}(i)$ according to step 1 in Table 6.3 and writes them into the $\mathbf{R}$ RAM through port A. The whole process is pipelined under the control of the Correlation Module State Machine and requires one cycle for updating one element of $\mathbf{R}(i)$ with a 4-cycle latency. In total, the matrix update requires $(N^2+N)/2+4$ cycles.

The updating of the vector $\boldsymbol{\beta}_0(i)$ is carried out in two stages. The first stage is the computation of the error signal $e(i)$ according to steps 2 and 3 in Table 6.3. The $\mathbf{x}$ RAM Reader A writes sequentially addresses of elements of the vector $\mathbf{x}(i)$ into the $\mathbf{x}$ RAM port A. The $\mathbf{h}$ RAM Reader writes sequentially addresses of elements of the vector $\hat{\mathbf{h}}(i-1)$ into the $\mathbf{h}$ RAM. The MUY2 Writer reads elements of $\mathbf{x}(i)$ and $\hat{\mathbf{h}}(i-1)$, and writes them

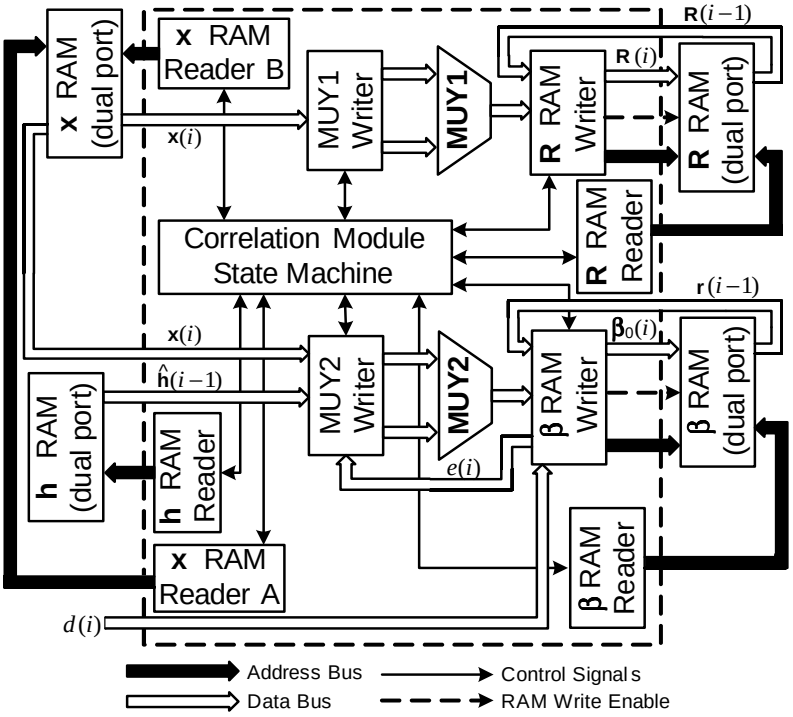


Figure 6.2: Block-diagram of the Correlation Module.

into two operand ports of the multiplier MUY2. The MUY2 is a 16-bit×32-bit multiplier configured from two 18-bit×18-bit embedded multipliers. The $\boldsymbol{\beta}$ RAM Writer reads multiplication results from the MUY2 and accumulates them into the signal $y(i)$. The $\boldsymbol{\beta}$ RAM Writer reads $d(i)$ stored in a register and calculates the error signal $e(i)$ according to step 3 in Table 6.3.

The second stage is executed according to step 4 in Table 6.3. The $\mathbf{x}$ RAM Reader A writes addresses of elements of the vector $\mathbf{x}(i)$ sequentially into the $\mathbf{x}$ RAM. The $\boldsymbol{\beta}$ RAM Reader writes addresses of elements of $\mathbf{r}(i-1)$ into the $\boldsymbol{\beta}$ RAM. The MUY2 Writer reads elements of $\mathbf{x}(i)$ from the $\mathbf{x}$ RAM and $e(i)$ from the $\boldsymbol{\beta}$ RAM Writer, and writes them into two operand ports of the multiplier MUY2. The $\boldsymbol{\beta}$ RAM Writer reads the multiplication results from the MUY2 and elements of the vector $\mathbf{r}(i-1)$ from the $\boldsymbol{\beta}$ RAM port B sequentially, computes elements of the vector $\boldsymbol{\beta}_0(i)$, and writes them into the $\boldsymbol{\beta}$ RAM through port A. The latency of each stage is 3 cycles. Thus, the updating of the vector $\boldsymbol{\beta}_0(i)$ requires $2N+6$ cycles.

The DCD Processor here is mostly the same as that of the real-valued leading DCD algorithm in Chapter 2 except for finding the leading element for the first iteration at each time instant i. The $\boldsymbol{\beta}$ RAM is configured in "Transparent Mode", i.e., the input data are simultaneously written into the memory and placed on the data output port [76]. At each time instant, when writing the vector $\boldsymbol{\beta}_0(i)$ into the $\boldsymbol{\beta}$ RAM in the Correlation

Module Fig. 6.2, the $\boldsymbol{\beta}$ Max module of the DCD processor works synchronously with the Correlation Module. The $\boldsymbol{\beta}$ Max reads elements of the vector $\boldsymbol{\beta}_0(i)$ sequentially, finds the maximum absolute value and outputs its index n to the DCD Core State Machine for the first iteration at a time instant i. Therefore, the maximum number of cycles for the real-valued leading DCD implementation is $(N+6)N_u + 4M_b - 1$ per sample.

In the Correlation Module, updating the matrix $\mathbf{R}(i)$ and vector $\boldsymbol{\beta}_0(i)$ are executed simultaneously. In the case of arbitrary regressors $\mathbf{x}(i)$, updating the correlation matrix requires significantly more cycles than updating $\boldsymbol{\beta}_0(i)$. Therefore, the maximum number of cycles for the RLS-DCD adaptive filter with an arbitrary data structure is $(N^2+N)/2+(N+6)N_u + 4M_b + 3$ per sample.

6.4.2 FPGA Implementation for Time-Shifted Data Vectors (Transversal Adaptive Filter)

For the transversal RLS-DCD adaptive filter, the vector $\mathbf{x}(i)$ has a time-shifted structure (6.20) and the upper-left $(N-1)\times(N-1)$ block of $\mathbf{R}(i)$ can be obtained by shifting the lower-right $(N-1)\times(N-1)$ block of $\mathbf{R}(i-1)$. The direct shifting would require at least $(N-1)\times(N-1)$ cycles, as the block RAMs can only be accessed at one memory space per cycle [76]. A simple memory address modification is performed instead of the direct copying. The data do not change the position in the RAM and only the corresponding address counters are modified. The $\mathbf{x}$ RAM is now configured as a circular buffer. At each time instant i, the Transceiver writes an element $x(i)$ into the $\mathbf{x}$ RAM to overwrite the first element of the vector $\mathbf{x}(i-1)$.

To update the matrix $\mathbf{R}(i)$, the $\mathbf{x}$ RAM Reader B generates the address of the element $x(i)$ and then sequentially generates addresses of elements of the vector $\mathbf{x}(i)$. The MUY1 Writer reads elements $\mathbf{x}(i)$ from the $\mathbf{x}$ RAM, writes the element $x(i)$ into an operand port of the multiplier MUY1, and sequentially writes N elements of $\mathbf{x}(i)$ into another operand port to produce the product $x(i)\mathbf{x}(i)$ according to (6.21). The $\mathbf{R}$ RAM Reader writes addresses of elements in the last column $\mathbf{R}_{:,N}(i-1)$ of $\mathbf{R}(i-1)$ into the $\mathbf{R}$ RAM. The $\mathbf{R}$ RAM Writer reads elements of $\mathbf{R}_{:,N}(i-1)$ from the $\mathbf{R}$ RAM and multiplication results from the multiplier MUY1, computes elements of the column $\mathbf{R}_{:,N}(i)$, and writes them into the $\mathbf{R}$ RAM through port A to overwrite the column $\mathbf{R}_{:,1}(i-1)$. As only the upper triangular part of the matrix $\mathbf{R}(i)$ is involved, the column $\mathbf{R}_{:,n}(i)$ is accessed in the following order: $R_{1,n}(i), \ldots R_{n,n}(i), R_{n,n+1}(i), \ldots R_{n,N}(i)$. The whole process is fully pipelined under the control of the Correlation Module State Machine. As only one

column of $\mathbf{R}(i)$ is involved in the computation, the matrix update requires $N + 4$ cycles. The other modules of the transversal RLS-DCD adaptive filter operate similarly to the modules of the RLS-DCD filter with arbitrary regressors. In the case of time-shifted regressors $\mathbf{x}(i)$, updating $\mathbf{R}(i)$ requires less cycles than updating $\boldsymbol{\beta}_0(i)$. The "worst-case" number of cycles for the transversal RLS-DCD adaptive filter is $2N+(N+6)N_u+4M_b+5$ per sample.

6.4.3 FPGA Resources for RLS-DCD Adaptive Filtering Algorithm

The RLS-DCD algorithm has been implemented for the cases $N = 16,\ 18$ and 64 with $M_b = 15$ and $N_u = 16$, for both arbitrary and time-shifted data structures. The choice of the high number of iterations $N_u = 16$ will guarantee filtering results close to that of the classical RLS algorithm (as analyzed in Section 6.6). In some applications, it can be significantly reduced, as even as small number of iterations as $N_u = 1$ can provide performance close to that of the classical RLS algorithm (to be shown in Section 6.6).

FPGA resources for four implementations are presented in Table 6.7. The area usage of the Transceiver module is not included as it is application specific. The whole implementation requires at most 9.5% of the resources available on the FPGA chip. The overhead in the slice count posed by the increase of the filter size is small and is mostly due the increase of the address bus-widths and the address counter-widths.

The RLS-DCD implementation for arbitrary regressors is applicable, for example, to adaptive antenna beamforming. For an 9-element antenna array with complex-valued weights (corresponding to a 18-tap adaptive filter), we obtain the update rate at least 162 kHz (619 cycles per sample). Comparing with an 9-element Altera CORDIC-based QRD-RLS implementation [36], our implementation achieves about 67 times higher throughput and with approximately the same chip area. Moreover, the Altera implementation requires an additional NIOS processor to perform the back substitutions. Our implementation also achieves about 17 times higher throughput and with approximately 3 times smaller chip area, compared to a Xilinx CORDIC-based QRD implementation [38], which requires 3530 slices and 13 DSP48 blocks for solving a 9×9 system of equations with a latency of 10971 cycles. The throughput of the $N = 18$ RLS-DCD implementation is about 13 times higher than that of an 9-element MVDR beamformer in Chapter 5 using the complex-valued leading DCD algorithm to directly solve the system (6.1). Comparing with the area usage of $N = 16$ serial implementation of the complex-valued leading DCD algorithm in Table 3.14, this RLS-DCD implementation only has an extra 500 slices. For a 32-element antenna (64-tap adaptive filter), we obtain a 31 kHz update rate at least, which

Table 6.7: FPGA resources for RLS-DCD adaptive filter

Algorithms	RLS-DCD		Transversal RLS-DCD	
Resources:	$N = 18$	$N = 64$	$N = 16$	$N = 64$
Slices	1103 (7.6%)	1174 (8.6%)	1153 (8.4%)	1306 (9.5%)
Block RAM	5 (3.7%)	11 (8.1%)	4 (2.9%)	11 (8.1%)
Multiplier	3 (2.2%)	3 (2.2%)	3 (2.2%)	3 (2.2%)
Update Rate	162 kHz	31 kHz	207 kHz	76 kHz

is 5 times higher than that of the implementation based on the direct use of the complex-valued DCD algorithm in Chapter 5. The DCD algorithm implemented here uses a serial implementation with a maximum of $N+6$ cycles for one iteration. The weight update rate of the RLS-DCD adaptive filter can be further increased by using a group implementation or a parallel implementation of the DCD algorithm as we discussed in Chapter 3.

6.5 FPGA Implementation of Dynamically Regularized RLS-DCD Adaptive Filtering Algorithm

The architecture of the implementation is similar to the implementation of unregularized RLS-DCD algorithm which is shown in Fig. 6.1. The system clock is 100 MHz and distributed using the FPGA Digital Clock Manager and Global Clock Distribution Network [76] to ensure a uniform delay between the system clock source and each logic slice. The Master State Machine coordinates the operation of the whole system. The Transceiver handles the data communication between the FPGA board and a Host Computer.

The Host Computer provides the input data $\mathbf{x}(i)$, $\delta(i)$ and $\boldsymbol{\beta}(i)$ to the FPGA board and receives the filter weights $\hat{\mathbf{h}}(i)$ and the filter output $y(i)$ from the board. The scalar $\delta(i)$ is stored in a register. Five dual-port block RAMs are used to store the matrix $\mathbf{R}(i)$ and vectors $\boldsymbol{\beta}_0(i)$, $\hat{\mathbf{h}}(i)$, $\boldsymbol{\beta}(i)$, $\Delta\boldsymbol{\beta}(i)$ and $\mathbf{x}(i)$. The $\mathbf{x}$ RAM and $\boldsymbol{\beta}_{in}$ RAM used for storing vectors $\mathbf{x}(i)$, $\boldsymbol{\beta}(i)$ and $\Delta\boldsymbol{\beta}(i)$ are located inside the Transceiver module; they are not shown in Fig. 6.1. The vector $\Delta\boldsymbol{\beta}(i)$ is computed inside the Transceiver. Multiplexers (MUXs) are used for multi-accessing these RAMs. The Correlation Module updates the matrix $\mathbf{R}(i)$ and calculates the vector $\boldsymbol{\beta}_0(i)$ according to steps 1 to 3 in Table 6.5. The DCD Processor uses the DCD algorithm for solving the auxiliary equations (6.6) and generating the residual vector $\mathbf{r}(i)$ at step 4 in Table 6.5; step 5 is also incorporated in the DCD Processor.

For representation of $\delta(i)$ and each component of vectors $\mathbf{x}(i)$, $\boldsymbol{\beta}(i)$ and $\Delta\boldsymbol{\beta}(i)$, 16-

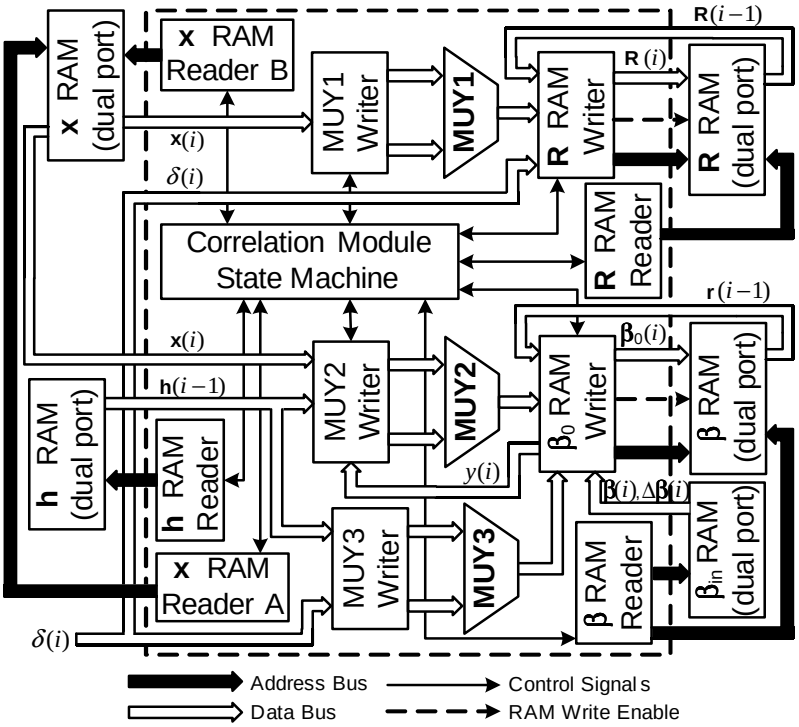


Figure 6.3: Block-diagram of the Correlation Module.

bit fixed-point words in the Q15 format [87] are used. Components of the matrix $\mathbf{R}(i)$ and vectors $\mathbf{r}(i)$, $\boldsymbol{\beta}_0(i)$, and $\hat{\mathbf{h}}(i)$ are 32-bit fixed-point words in the Q15 format. When computing the filter output $y(i)$ according to step 2 in Table 6.5, after accumulation of the N products, both components of $y(i)$ are truncated to 32-bit fixed-point words in the Q15 format.

The Correlation Module is shown in Fig. 6.3. The functions and operations of each module are similar to that of the correlation module of the implementation of RLS-DCD algorithm in Fig 6.2. For updating the correlation matrix $\mathbf{R}(i)$, at each time instant i, the $\mathbf{R}$ RAM Writer reads the elements $\mathbf{x}(i)\mathbf{x}^H(i)$ from the multiplier MUY1, elements of $\mathbf{R}(i-1)$ from $\mathbf{R}$ RAM port B, $\delta(i-1)$ from internal register and $\delta(i)$ from input register, computes elements of $\mathbf{R}(i)$ according to step 1 in Table 6.5 and writes them to the $\mathbf{R}$ RAM through port A. After updating all upper triangular elements of $\mathbf{R}(i)$, the $\mathbf{R}$ RAM Writer replaces $\delta(i-1)$ in its internal register with $\delta(i)$. The remaining operations for updating the matrix $\mathbf{R}(i)$ are exactly the same as the operations in the implementation of RLS-DCD algorithm with arbitrary data vector in Section 6.4.1. The whole process is pipelined under the control of the Correlation Module State Machine and requires one cycle for updating one element of $\mathbf{R}(i)$ with a 6-cycle latency. In total, the matrix update requires $(N^2+N)/2+6$ cycles.

Similar to the previous implementation, the updating of the vector $\boldsymbol{\beta}_0(i)$ is also carried

out in two stages. The first stage is the computation of the filter output $y(i)$ according to step 2 in Table 6.5. The $\mathbf{x}$ RAM Reader A and the $\mathbf{h}$ RAM Reader assert addresses to $\mathbf{x}$ RAM and $\mathbf{h}$ RAM to read the vectors $\mathbf{x}(i)$ and $\mathbf{h}(i-1)$, sequentially and respectively. The MUY2 Writer reads elements of $\mathbf{x}(i)$ and $\hat{\mathbf{h}}(i-1)$, and writes them into two operand ports of the multiplier MUY2, sequentially. The $\boldsymbol{\beta}$ RAM Writer reads multiplication results from the MUY2 and accumulates them into the signal $y(i)$.

The second stage is executed according to step 3 in Table 6.5. The $\mathbf{x}$ RAM Reader A and the $\mathbf{h}$ RAM Reader read the vectors $\mathbf{x}(i)$ and $\mathbf{h}(i-1)$ once more, respectively. The MUY2 Writer reads elements of $\mathbf{x}(i)$ from the $\mathbf{x}$ RAM and $y(i)$ from the $\boldsymbol{\beta}$ RAM Writer, and writes them into two operand ports of the multiplier MUY2. The MUY3 Writer reads elements of $\hat{\mathbf{h}}(i-1)$, $\delta(i)$ and $\delta(i-1)$ from its internal register, and writes $[\delta(i)-\lambda\delta(i-1)]$ and elements of $\hat{\mathbf{h}}(i-1)$ into two operand ports of MUY3. Simultaneously, the $\boldsymbol{\beta}$ RAM Reader writes addresses to the $\boldsymbol{\beta}$ RAM and the $\boldsymbol{\beta}_{in}$ RAM to read $\mathbf{r}(i-1)$, $\boldsymbol{\beta}(i)$ and $\Delta\boldsymbol{\beta}(i)$, respectively and sequentially. The $\boldsymbol{\beta}$ RAM Writer reads the multiplication results from the MUY2 and MUY3, elements of the vector $\mathbf{r}(i-1)$ from the $\boldsymbol{\beta}$ RAM port B, and elements of $\boldsymbol{\beta}(i)$ and $\Delta\boldsymbol{\beta}(i)$ from the $\boldsymbol{\beta}_{in}$ RAM, computes elements of the vector $\boldsymbol{\beta}_0(i)$, and writes them into the $\boldsymbol{\beta}$ RAM through port A. After computing all N elements sequentially and in a pipelined manner, the MUY3 Writer stores $\delta(i)$ to replace the $\delta(i-1)$ in register. The latency of each stage is 5 cycles. Thus, the updating of the vector $\boldsymbol{\beta}_0(i)$ requires $2N+10$ cycles.

We should notice that the implementation here processes the complex-valued systems and that the real components and the imaginary components are processed simultaneously. The MUY1 is a 16-bit$\times$16-bit complex-valued multiplier configured from three 18-bit$\times$18-bit embedded multipliers. The MUY2 is a 16-bit$\times$32-bit complex-valued multiplier configured from six 18-bit$\times$18-bit embedded multipliers. The MUY3 contains two 16-bit$\times$32-bit real-valued multipliers for real parts and imaginary parts respectively. Each 16-bit$\times$32-bit real-valued multiplier is composed by two 18-bit$\times$18-bit embedded multipliers. Thus, 13 18-bit$\times$18-bit embedded multipliers are required in the Correlation Module.

The DCD Processor here is slightly different from the serial implementation of the complex-valued leading DCD algorithm in Section 3.4.2; finding the leading component for the first iteration at each time instant i is optimized for this implementation. The $\boldsymbol{\beta}$ RAM is configured in "Transparent Mode", i.e., the input data are simultaneously written into the memory and stored in the data output port [76]. At each time instant, when writing the vector $\boldsymbol{\beta}_0(i)$ into the $\boldsymbol{\beta}$ RAM in the Correlation Module Fig. 6.2, the $\boldsymbol{\beta}$ Max module of the DCD processor works synchronously with the Correlation Module. The $\boldsymbol{\beta}$ Max reads

Table 6.8: FPGA resources for dynamically regularized RLS-DCD

Size	Slices	Block RAM	Multiplier	Update Rate
$N = 16$	2680(19.57%)	8(5.88%)	13(9.57%)	176 kHz
$N = 64$	2814(20.55%)	21(15.44%)	13(9.57%)	31 kHz

elements of the vector $\boldsymbol{\beta}_0(i)$ sequentially, finds the maximum absolute value and outputs its index (p, s) to the DCD Core State Machine for the first iteration at a time instant i. The remaining operations of the DCD processor are the same as the implementation in Section 3.4.2. The maximum cycles cost by the DCD processor is $(N+7)N_u + 4M_b - 3$.

In the Correlation Module, Updating the matrix $\mathbf{R}(i)$ and vector $\boldsymbol{\beta}_0(i)$ are executed in parallel. Therefore, the "worst-case" number of cycles for the dynamically regularized RLS-DCD adaptive filter is $(N^2 + N)/2 + (N + 7)N_u + 4M_b + 3$.

FPGA resources (excluding the Transceiver module) are presented in Table 6.8 for the cases $N = 16$ and $N = 64$ with $M_b = 15$ and $N_u = 16$. The area usage of the Transceiver module is not included as it is application specific. The whole implementation requires at most 21% of the resources available on the FPGA chip. The overhead in the slice count posed by the increase of the filter size is small and is mostly due the increase of the address bus-widths and the address counter-widths. The design processes the real components and imaginary components simultaneously. The area usage is approximately twice of the real-valued implementation of unregularized RLS-DCD algorithm in Section 6.2.1. The choice of the high number of iterations $N_u = 16$ will guarantee filtering results close to that of the regularized classical RLS algorithm.

The implementation with arbitrary regressors is applicable, for example, to adaptive antenna beamforming. For an 16-element antenna array, we obtain the update rate at least 191 kHz (567 cycles per sample) that is about 74 times higher than an 9-element Altera CORDIC-based QRD-RLS implementation [36] with approximately two times larger chip area, and 19 times higher than a 9×9 Xilinx CORDIC-based QRD implementation [38] with approximately 1.4 times smaller chip area. For a 64-element antenna, we obtain a 31 kHz update rate at least. The DCD algorithm implemented here uses a serial implementation with a maximum of $N + 7$ cycles for one iteration. The weight update rate can be further increased by using a group-element implementation or a parallel implementation of the DCD algorithm as we discussed in Chapter 3.

Table 6.9: Complexity of proposed and known transversal adaptive algorithms

Algorithm	$\times$	+	$\div$
ERLS-CG	$N^2N_u + 5NN_u + 3N$	$N^2N_u + 4NN_u + 6N$	$2N_u - 1$
ERLS-CD	$NN_u + 3N$	$2NN_u + 6N$	N_u
ERLS-DCD	$3N$	$2NN_u + 6N$	-
RLS	$N^2 + 5N + 1$	$N^2 + 3N$	1
NLMS	$2N + 3$	$2N + 3$	1

6.6 Numerical Results for RLS-DCD Adaptive Filtering Algorithm

Table 6.9 shows the complexity of the proposed and known transversal adaptive filters; the complexity of the RLS and NLMS algorithms is from [7]. The complexity of the ERLS algorithms takes into account the choice of the forgetting factor as $\lambda = 1 - 2^{-P}$ with a positive integer P. For additions, we only show figures that are $\mathcal{O}(N^2)$ or $\mathcal{O}(N)$ and ignore figures that are $\mathcal{O}(N_u)$, $\mathcal{O}(M_b)$ or $\mathcal{O}(1)$. It is seen that the transversal ERLS-DCD algorithm requires only $3N$ multiplications per sample and no division.

Below, we present numerical results for RLS-DCD adaptive filtering algorithm obtained by computer simulation. We compare the MSE performance of the proposed adaptive algorithms against the classical exponentially weighted RLS algorithm, NLMS algorithm, and a recently proposed efficient conjugate gradient control Liapunov function (CG-CLF) algorithm with complexity $\mathcal{O}(N^2)$ [30]. Only scenarios with the time-shifted structure of input data, corresponding to the transversal adaptive filter, are considered. The input data are generated according to

$$d(i) = \mathbf{h}^T(i)\mathbf{x}(i) + n(i) \tag{6.27}$$

where $n(i)$ is the additive zero-mean Gaussian random noise with variance σ^2. The vector $\mathbf{x}(i) = [x(i)\ x(i-1)\ \ldots\ x(i-N+1)]^T$ contains either a real speech signal or autoregressive correlated random numbers given by

$$x(i) = \nu x(i-1) + w(i) \tag{6.28}$$

where ν is the autoregressive factor ($0 \leq \nu < 1$) and $w(i)$ are uncorrelated zero-mean random Gaussian numbers of unit variance. The MSE in a simulation trial is calculated as

$$\varepsilon(i) = \frac{[\mathbf{h}(i) - \hat{\mathbf{h}}(i)]^T[\mathbf{h}(i) - \hat{\mathbf{h}}(i)]}{\mathbf{h}^T(i)\mathbf{h}(i)}. \tag{6.29}$$

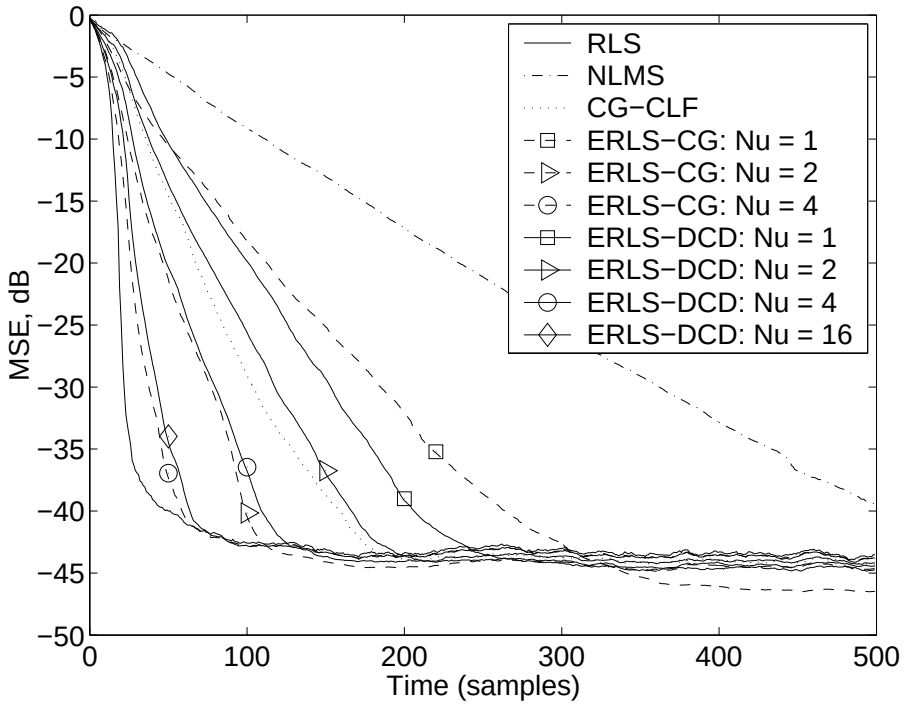
Table 6.10: Complexity of adaptive algorithms ($N = 16$)

Algorithm	$\times$	+	$\div$
ERLS-CG(N_u=1)	384	416	1
ERLS-CD(N_u=1)	64	128	1
ERLS-DCD(N_u=1)	48	128	-
ERLS-CG(N_u=4)	1392	1376	7
ERLS-CD(N_u=4)	112	224	4
ERLS-DCD(N_u=4)	48	224	-
ERLS-CG(N_u=16)	5424	5216	31
ERLS-CD(N_u=16)	304	608	16
ERLS-DCD(N_u=16)	48	608	-
RLS	337	304	1
NLMS	35	35	1

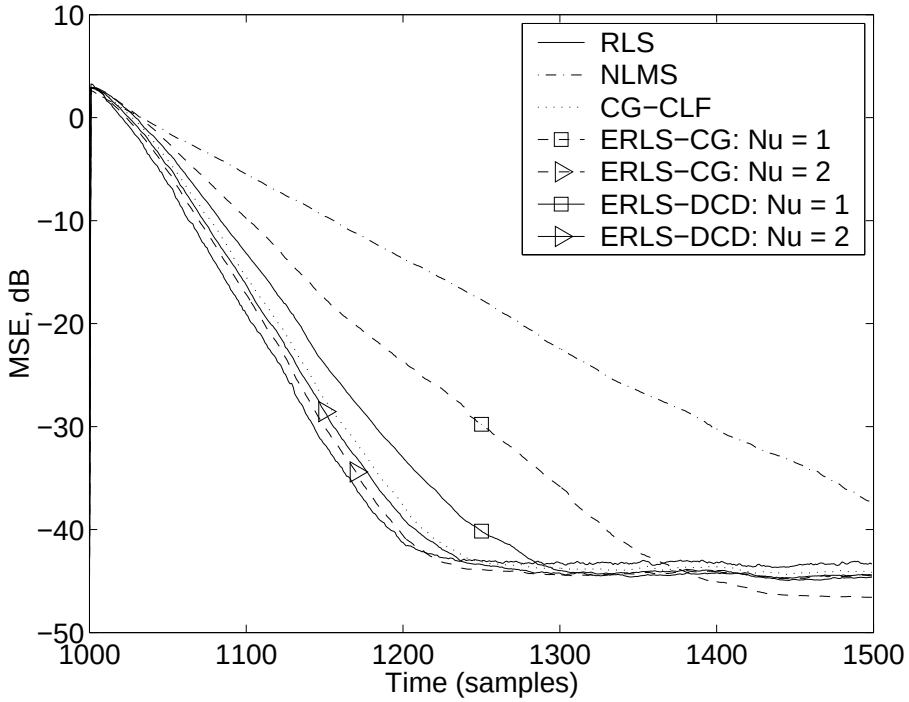
The MSEs obtained in N_{mc} trials are averaged and plotted against the time index i. Results in Figs. 6.4 to 6.7 below are obtained by floating point simulation. Fig. 6.8 compares floating and fixed point simulation results.

Fig. 6.4 shows the MSE performance of the ERLS-CG and ERLS-DCD algorithms against the RLS, NLMS, and CG-CLF algorithms. All elements of the impulse response $\mathbf{h}(i)$ are kept constant over the first 1000 samples; the elements are independent random numbers uniformly distributed on $[-1, +1]$. At time instant $i = 1000$, a new vector $\mathbf{h}$ is generated and kept constant over the remaining samples. It is seen that, in the case of $N_u = 1$, the ERLS-DCD algorithm outperforms the ERLS-CG algorithm, but is inferior to the CG-CLF algorithm. For $N_u = 2$, the ERLS-DCD and CG-CLF algorithms demonstrate similar performance, whereas the ERLS-CG algorithm converges faster. For $N_u \geq 4$, the ERLS-DCD and ERLS-CG algorithms outperform the CG-CLF algorithm. For a fixed N_u, the ERLS-CG algorithm converges faster than the ERLS-DCD algorithm. However, this is achieved at the expense of a significant increase in the complexity (see Table 6.10). Under a fixed complexity, the ERLS-DCD algorithm provides significantly faster convergence than the ERLS-CG algorithm. Fig. 6.4b shows that after a change of the impulse response, only two updates ($N_u = 2$) are enough for both the ERLS-CG and ERLS-DCD algorithms to approach the RLS performance. The results for $N_u > 2$ are not shown as they are not distinguishable from that of the classical RLS algorithm.

Fig. 6.5 compares the performance of the ERLS-CD and ERLS-DCD algorithms. It is seen that, with increase in N_u, the ERLS-CD algorithm approaches the RLS performance. However, the performance of the ERLS-DCD algorithm is superior to that of the ERLS-CD and, as seen from Table 6.10, it requires a significantly fewer number of multiplications.

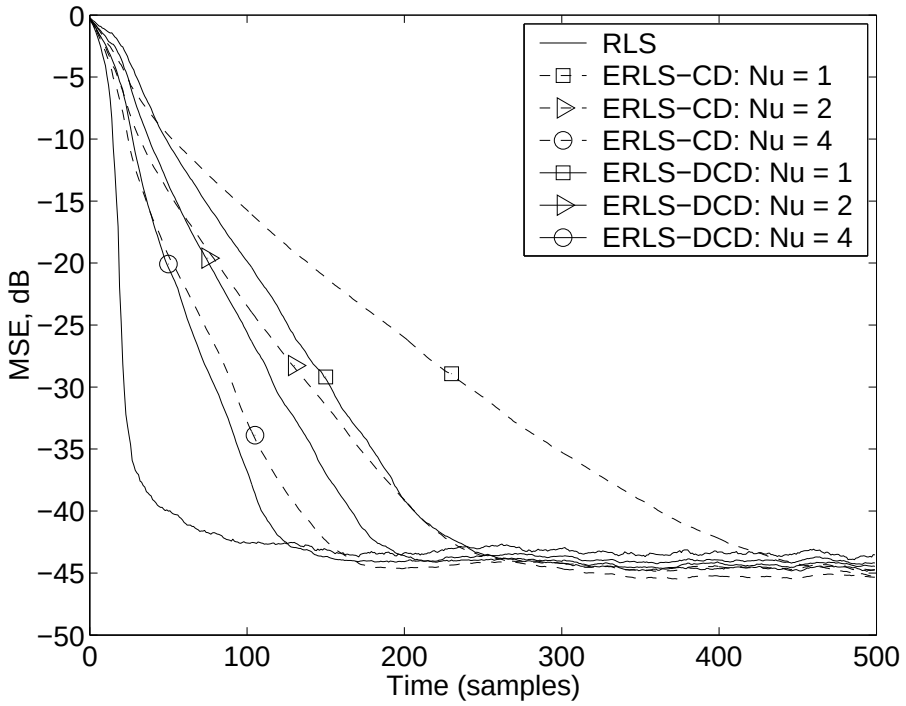


(a)

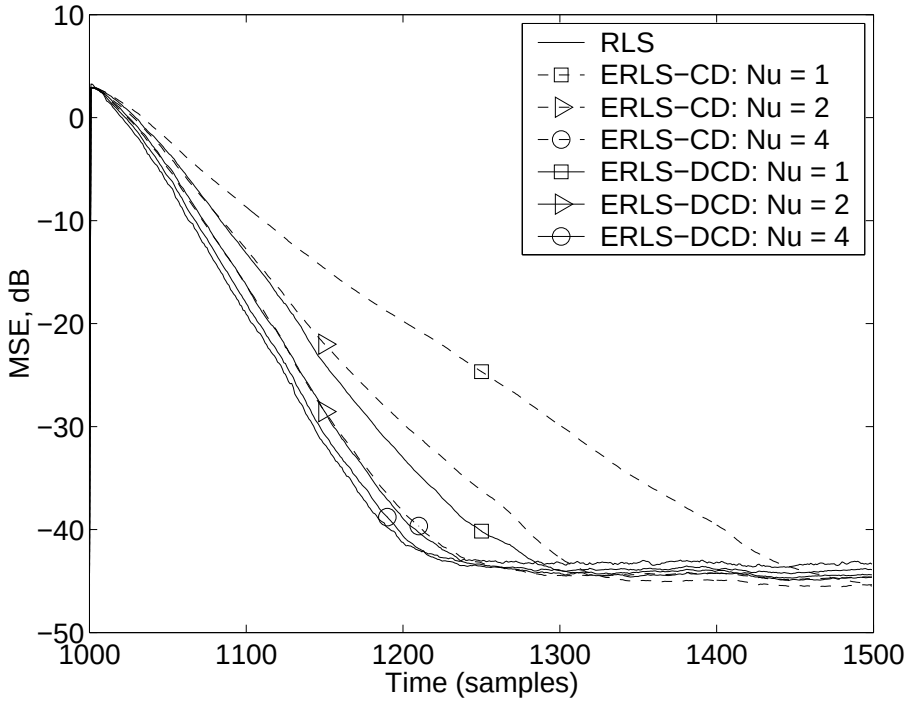


(b)

Figure 6.4: MSE performance of the ERLS-CG and ERLS-DCD algorithms against the RLS, NLMS, and CG-CLF algorithms; $N = 16$, $\lambda = 1 - 1/(2N) \approx 0.969$, $\eta = 10^{-3}$, $H = 1$, $M_b = 16$, $\nu = 0.9$, $\sigma = 0.01$, $N_{mc} = 100$: (a) initial convergence; (b) convergence after a change of the impulse response.



(a)



(b)

Figure 6.5: MSE performance of the ERLS-CD and ERLS-DCD algorithms against the RLS and NLMS algorithms; $N = 16$, $\lambda = 1 - 1/(2N) \approx 0.969$, $\eta = 10^{-3}$, $H = 1$, $M_b = 16$, $\nu = 0.9$, $\sigma = 0.01$, $N_{mc} = 100$: (a) initial convergence; (b) convergence after a change of the impulse response.

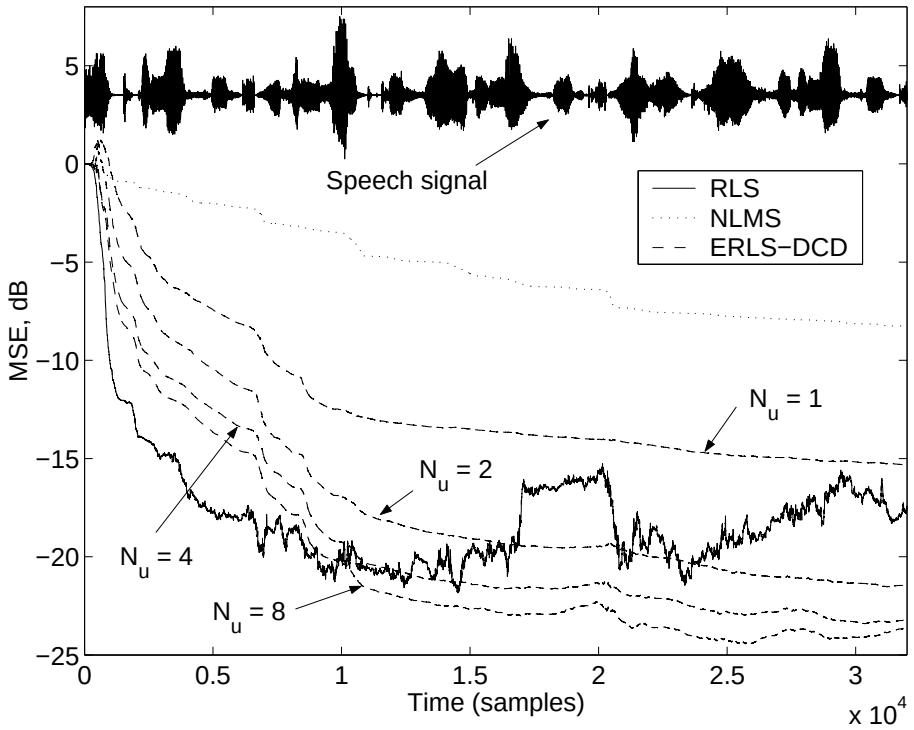


Figure 6.6: Echo cancellation experiment with a real speech signal. MSE performance of the ERLS-DCD vs. RLS and NLMS algorithms: $N = 512$, SNR = 30 dB, $\lambda = 1 - 1/(4N) \approx 0.9995$, $\eta = 0.015$, $H = 1$, $M_b = 16$, $N_{mc} = 1$.

The results in Fig. 6.6 correspond to the application of adaptive filtering to acoustic echo cancellation with a long impulse response, $N = 512$. Elements of the impulse response h_n, $n = 1, \ldots, N$, are independent zero-mean random numbers with variance $\exp(-0.005n)$, which corresponds to a typical acoustic impulse response [106]. The vectors $\mathbf{x}(i)$ contain a real speech signal sampled at a frequency of 8 kHz. It is seen that with $N_u = 1$, the ERLS-DCD algorithm significantly outperforms the NLMS algorithm. With increase in N_u, the MSE performance of the ERLS-DCD algorithm is significantly improved and, in the steady-state, for $N_u \geq 2$, it outperforms the RLS algorithm. Table 6.11 shows the complexity of the three algorithms. It is seen that the complexity of the ERLS-DCD algorithm is significantly lower than that of the RLS algorithm and it requires only 50% more multiplications than the NLMS algorithm.

Fig. 6.7 shows the tracking performance of the ERLS-DCD algorithm in a time-varying environment. The n-th element $h_n(i)$ of the impulse response $\mathbf{h}(i)$ varies in time according to

$$h_n(i) = h_n(0)\cos(2\pi F i + \phi_n) \tag{6.30}$$

where ϕ_n are independent random numbers uniformly distributed on $[-\pi, \pi)$, $h_n(0)$ are independent zero-mean Gaussian random numbers of unit variance, and F is the variation

Table 6.11: Complexity of adaptive algorithms ($N = 512$)

Algorithm	$\times$	+	$\div$
ERLS-DCD(N_u=1)	1536	4096	-
ERLS-DCD(N_u=2)	1536	5120	-
ERLS-DCD(N_u=4)	1536	7168	-
ERLS-DCD(N_u=8)	1536	11264	-
RLS	264705	263680	1
NLMS	1027	1027	1

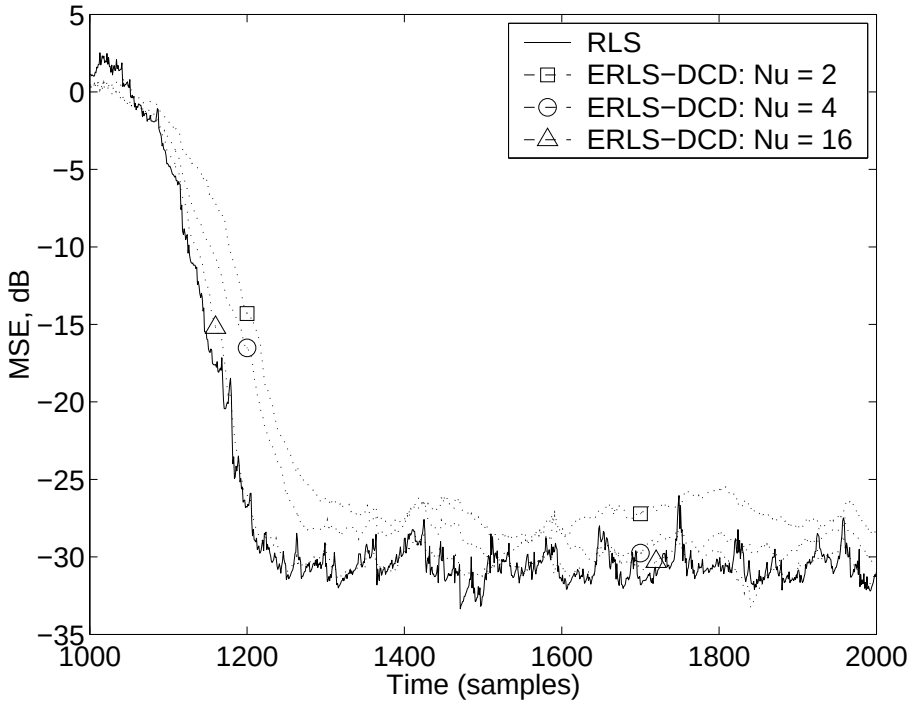


Figure 6.7: The tracking performance of the ERLS-DCD algorithm in a time-varying environment: $F = 10^{-4}$, $\nu = 0.9$, $\sigma = 0.001$, $N = 64$, $\lambda = 0.975$, $\eta = 10^{-3}$, $N_{mc} = 1$.

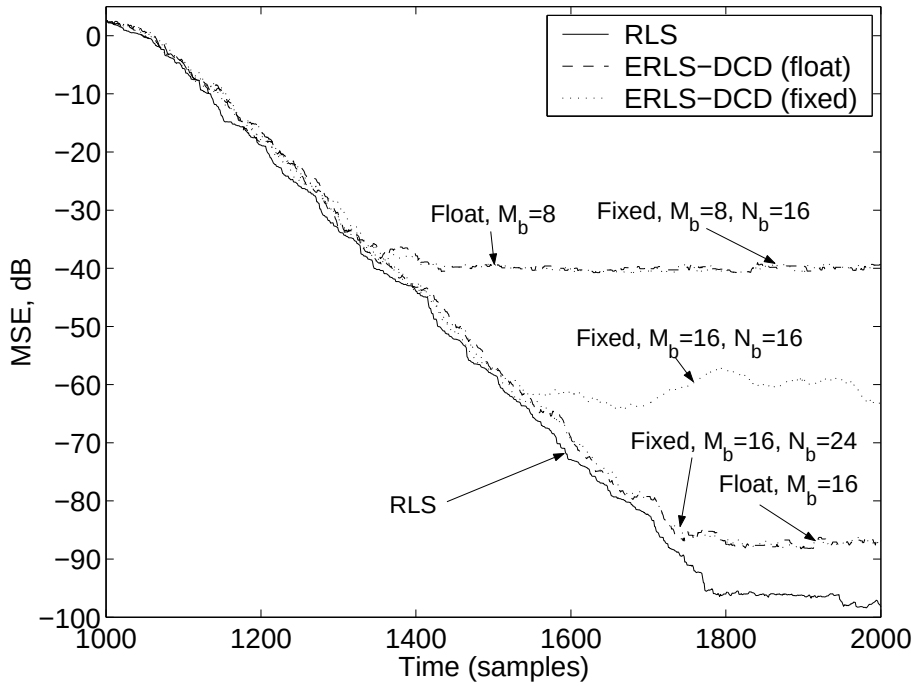


Figure 6.8: The MSE performance of a fixed-point implementation of the ERLS-DCD algorithm against the floating point ERLS-DCD and classical RLS algorithms: $N = 64$, $\sigma = 10^{-5}$, $\lambda = 1 - 1/N \approx 0.984$, $\eta = 2^{-10} \approx 10^{-3}$, $N_u = 2$, $\nu = 0$, $N_{mc} = 1$.

rate. It is seen that as N_u increases, the MSE performance of the ERLS-DCD algorithm is approaching that of the RLS algorithm.

Fig. 6.8 shows the performance of a fixed-point implementation of the ERLS-DCD algorithm against the ERLS-DCD and classical RLS algorithms implemented in floating-point. For representation of all variables in the algorithm, including the input data $d(i)$ and $\mathbf{x}(i)$, elements of the matrix $\mathbf{R}$ and vector $\mathbf{r}$, etc., N_b bits are used ($N_b = 16$ or $N_b = 24$). It can be seen that the accuracy of both the fixed-point ERLS-DCD and floating-point ERLS-DCD algorithms depends on the parameter M_b that defines the number of bits for representation of the solution vector $\hat{\mathbf{h}}$. As M_b increases, the steady-state MSE approaches that of the RLS algorithm. For the fixed-point ERLS-DCD algorithm, for a fixed M_b, the steady-state MSE depends on N_b. In this scenario, for $M_b = 16$, the parameter $N_b = 16$ limits the algorithm performance, while $N_b = 24$ provides enough accuracy to achieve the floating-point performance.

6.7 Numerical Results for Dynamically Regularized RLS-DCD Adaptive Algorithm

We present numerical results for a communication system with the MVDR beamformer [70]. The configuration we examine is a N-element uniform linear array (ULA). The complex-valued $N \times 1$ vector $\mathbf{x}(i)$ provided by the array at time i is referred to as a "snapshot". A stream of snapshots is used for calculating the correlation matrix $\mathbf{R}(i)$ according to (6.22). The complex-valued $N \times 1$ steering vector $\boldsymbol{\beta}(i)$ is assumed to be provided by an external DoA estimator.

Our test scenario involves a desired user at angle 10° and two interfering users at angles -40° and 60° from the normal axis of a 16-element ULA. Binary Phase Shift Keying (BPSK) modulation scheme is employed. The interfering users are at a power level of 0 dB relative to the desired user. The SNR is set at 20 dB. The forgetting factor λ is chosen as $(1 - 2^{-8}) \approx 0.9961$. We choose a fixed $\delta(i) = 0.1$ (100 times of the noise variance). Fig. 6.9 shows the BER performance of the regularized RLS-DCD algorithm against the regularized and unregularized classical RLS algorithm. The BER obtained in 4000 trials is averaged and plotted against the time index i. It is seen that the regularization significantly improves the performance of the classical RLS algorithms. The convergence speed of the regularized RLS-DCD algorithm depends on the algorithm accuracy which is determined by the parameters M_b and N_u. For a fixed $M_b = 15$, as shown in Fig. 6.9, when N_u increases, the steady-state BER of the regularized RLS-DCD algorithm is reduced. With $N_u = 16$, the regularized RLS-DCD algorithm obtains approximately the same BER performance as that of the regularized RLS algorithm. The BER performance results for the fixed-point FPGA implementation are also presented to show that there is no significant difference with the floating-point results.

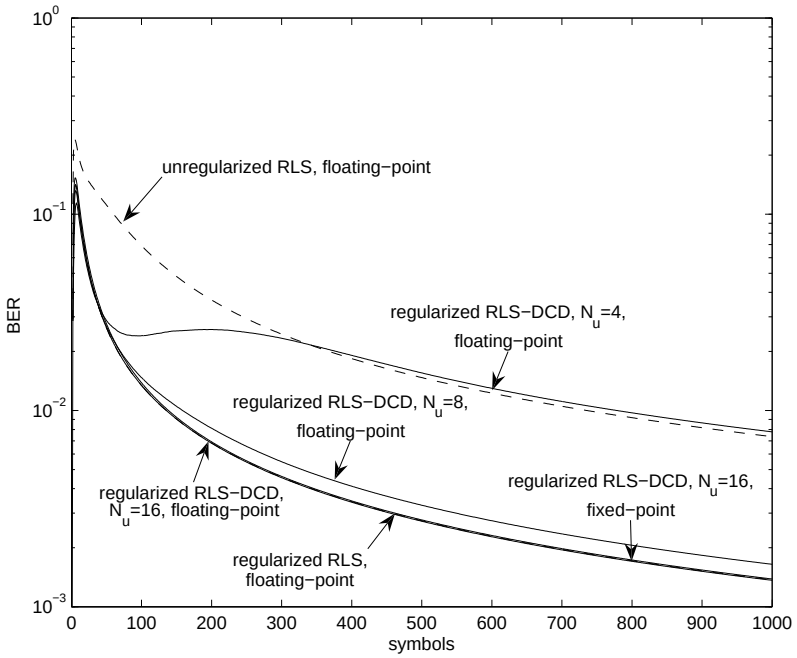


Figure 6.9: BER performance of the dynamically regularized RLS-DCD algorithm against the classical RLS algorithm ($M_b = 15$)

6.8 Conclusions

In this chapter, we have derived low-complexity RLS adaptive filtering algorithms. The RLS problem is represented as a sequence of auxiliary normal equations which are then approximately solved by using iterative line search methods. The leading DCD algorithm is applied to solve the obtained normal equations. Both exponentially weighted RLS filtering case with unstructured data vector and transversal RLS filtering case with time-shift structured data vector were considered. Simulation results show that the performance of the proposed adaptive algorithms can be made arbitrarily close to that of the classical RLS algorithm. The convergence properties of the proposed algorithms were discussed. A fixed-point FPGA implementation of the exponentially weighted DCD-based RLS algorithms has also been described, which shows that the proposed algorithms are simple for finite precision implementation, require small chip resources, and exhibit numerical stability. However, the RLS-DCD algorithm does not allow additional regularization to maintain robustness and its initial regularization decays rapidly.

A low complexity dynamically regularized RLS adaptive filtering algorithm based on the RLS-DCD algorithm is proposed and implemented into FPGA. The derivation of this algorithm mostly follows the steps of derivation of the RLS-DCD algorithm. We have shown that this algorithm is simple for finite precision implementation and requires small chip resources. Numerical results show that the performance of the fixed-point FPGA implementation of the proposed algorithm provides BER performance results close to that of the floating-point regularized classical RLS algorithm in a communication system with the MVDR beamformer, while it only requires $\mathcal{O}(N^2)$ operations per sample.

Chapter 7

Conclusions and Future Work

Contents

7.1 Summary of the Work

In this thesis, we have investigated various architectures and implementations of the DCD algorithm (Chapter 3) and applied these implementations to several practical applications in signal processing and communications areas, namely complex division (Chapter 4), MVDR beamforming (Chapter 5) and RLS adaptive filtering (Chapters 6). Other applications, such as the MVDR DoA estimation and affine projection (AP) adaptive filtering, will be mentioned in Section 7.2.

In Chapter 3, we started with the introduction of the DCD algorithm and its variants: the cyclic DCD algorithm and leading DCD algorithm for both real-valued systems and complex-valued systems. Several hardware architectures and implementations of the DCD algorithm have been developed. The serial implementation of the real-valued DCD algorithm is the smallest hardware implementation and it is notable for smaller than any other methods requiring multiplication operations. However, the update rate is limited as the residual vector is updated sequentially. The proposed serial implementation of the complex-valued DCD algorithm introduces 2-element parallelism to update the real and imaginary components of the residual vector simultaneously. Thus, the update rate of the residual vector is twice higher than in the serial implementation of the real-

valued DCD algorithm. The area usage of these implementations are still very low and we have also implemented the real-valued DCD algorithm which extends the concept of the complex-valued DCD algorithm in order to process four consecutive data elements simultaneously. These 4-element group implementations bear understandable resemblance to the complex-valued serial architecture implementations. The update rate of the residual vector is enhanced by a factor of four compared with the serial implementation of the real-valued DCD algorithm. Two architectures of the parallel implementations, register-based and RAM-based, were investigated; the RAM-based implementation is much more attractive because of smaller area usage. The parallel implementation updates all elements of the residual vector in a single clock cycle simultaneously with the comparison operation. Comparing with the serial implementation and group-4 implementation, the update rate of parallel implementation is significantly increased. These architectures and implementations of the DCD algorithm provide several choices of throughput. Thus, one can choose suitable architectures and implementations of the DCD processor according to requirements of practical applications.

We have analyzed the numerical properties of the cyclic DCD algorithm and the leading DCD algorithm, including both floating-point and fixed-point implementations. The fixed-point implementations can obtain approximately the same accuracy as the floating-point implementations. For solving the system of equations that require small number of updates, sparse systems or systems with a small condition number, the leading DCD algorithm is much more suitable than the cyclic DCD algorithm due to its fast convergence speed. For systems requiring a large number of updates, the cyclic DCD algorithm is more suitable as it converges faster and holds a lower error level than the leading DCD algorithm. However, at the initial iterations, the leading DCD algorithm has a faster convergence speed.

We have also compared the convergence speed of the DCD algorithm with other well-known iterative techniques, such as the CG, Gauss-Seidel and CD algorithms. The DCD algorithm has the same drawback as other iterative method; it requires infinite number of iterations to converge to the optimal solution and it is very complicated to predict the required number of iterations. However, the DCD algorithm performs a similar convergence speed with the lowest complexity per iteration.

In Chapter 4, a low complexity complex divider was developed based on the DCD iterations. The division of two complex numbers is used diversely in signal processing areas. However, it has traditionally been a computationally-intensive process and largely implemented in software. The conventional routine requires six multiplications, three additions and two divisions; the computational load of this conventional method is high.

Alternatively, the complex division problem can be viewed as the solution of a 2×2 real-valued system of linear equations. We used the DCD algorithm to solve this 2×2 system of equations and implemented it on the FPGA chip. The implementation is simple and does not use any multiplication or division operations. Comparing with a conventional complex divider which provides the same accuracy as our DCD-based complex divider, we have found that our DCD-based multiplication-free complex divider has a 4.4 times smaller chip area usage while the throughput is approximately the same.

In Chapter 5, we have presented an efficient FPGA implementation of the MVDR beamformer based on DCD iterations. The MVDR beamforming achieves high levels of interference cancellation, but it is considered too computationally complex for practical implementation as it requires matrix inversion. The FPGA implementation of MVDR-DCD is very efficient in terms of both the number of FPGA slices and speed. Antenna beampatterns obtained from weights calculated in a fixed-point FPGA platform have been compared with those of a floating-point implementation. The comparison has shown a good match for linear arrays of size 9 to 64 elements. Comparing with a 9-element implementation based on the Altera CORDIC-based QRD-RLS algorithm, our 9-element MVDR beamformer obtains about 6 times higher throughput with approximately the same chip area. Comparing with a 9-element Xilinx's CORDIC-based QRD implementation, our implementation achieves about 1.4 times higher throughput and with approximately 2.3 times smaller chip area. Moreover, these two reference implementations require additional embedded processor and multipliers, respectively. The implementation of MVDR-DCD technique can also be used to solve the MVDR DoA estimation problem.

In Chapter 6, a low-complexity RLS adaptive filter using DCD iterations has been introduced and implemented on FPGA. The RLS-DCD algorithm expresses the RLS adaptive filtering problem in terms of auxiliary normal equations with respect to increments of the filter weights. The equations are solved using the DCD iterations that require no multiplication and no division and, therefore, they are well suited to hardware implementation. The RLS-DCD algorithm is implemented for two data structures: arbitrary and time-shifted. The complexity of the RLS-DCD algorithm for input data with the time-shifted structure is as low as $3N$ multiplications per sample. An 9-element antenna beamformer based on the arbitrary data structure implementation can achieve a weight update rate that is about 67 times higher than that of an Altera CORDIC-based QRD-RLS FPGA implementation with approximately the same area usage, and about 17 times higher than that of a Xilinx CORDIC-based QRD implementation with 3 times smaller chip area. The implementation of the transversal RLS-DCD algorithm, which has the time-shifted data structure can provide a weight update rate as high as 207 kHz and 76 kHz for 16-tap and 64-tap adaptive filters, respectively, while using as little as 1153 and

1306 slices, respectively. Numerical results show that the performance of the fixed-point FPGA implementation of the RLS-DCD algorithm is close to that of the floating-point classical RLS algorithm.

The classical RLS algorithm usually uses an initial regularization to stabilize the solution to the RLS problem. Because the initial regularization decays exponentially in time, we may have to add additional diagonal loading to maintain robustness. However, such extra diagonal loading increases the complexity to $\mathcal{O}(N^3)$ as it requires matrix inversion at each time instant, which makes the RLS algorithm impractical. The RLS-DCD algorithm described above does not allow the regularization to be used except the initial regularization that is used in the classical RLS algorithm. A low complexity dynamically regularized RLS adaptive filtering algorithm based on the RLS-DCD algorithm has been proposed and its FPGA implementation has been presented. The dynamically regularized RLS-DCD algorithm reduces the complexity of the regularized RLS problem to $\mathcal{O}(N^2)$ operations per sample. We have shown that this algorithm is simple for finite precision implementation and requires small chip resources. For $N = 16$ and $N = 64$ complex-valued system, our implementation achieves an update rate of 171 kHz and 31 kHz with 2680 and 2814 slices, respectively. Numerical results show that the performance of the fixed-point FPGA implementation of the proposed algorithm provides BER results close to that of the floating-point regularized classical RLS algorithm in a communication system with the MVDR beamformer.

7.2 Future Work

Some suggestions for future work based on this thesis are given below.

1) In Chapter 3, we have shown the numerical properties of the DCD algorithm and its variants. The cyclic DCD algorithm provides a faster convergence speed and a lower steady-state misalignment than the leading DCD algorithm for non-sparse systems with a large condition number. However, the cyclic DCD algorithm has a slower convergence speed than the leading DCD algorithm at initial iterations. Therefore, we may consider to solve the non-sparse systems with both cyclic DCD iterations and leading DCD iterations to perform a fast convergence speed and obtain a low steady-state misalignment. We call the obtained DCD algorithm as combined DCD algorithm.

Two combined DCD algorithms can be considered. One uses the leading and cyclic DCD iterations sequentially; the leading DCD iterations are used at initial iterations and

the cyclic DCD iterations are used at the other iterations.

The other method views the residual vector and the solution vector as containing several element-groups. At each iteration, the combined DCD algorithm updates a leading element within the current solution element-group, corresponding to the maximum absolute value element in the current residual element-group. Then, it moves to the next element-group; the element-groups are selected in a cyclic order.

Through these two combinations, the combined DCD algorithm may obtain faster convergence speed and achieve similar misalignment compared to the cyclic DCD algorithm when solving the non-sparse systems with large number of updates. Furthermore, the hardware implementations of the combined DCD algorithm can be realized based on the architectures and implementations of the cyclic DCD algorithm and the leading DCD algorithm presented in Chapter 3.

2) In Chapter 4, we have discussed the MVDR DoA estimation using DCD algorithm to solve the system of equations directly. The hardware implementation can be similar to that of the MVDR-DCD beamformer. Alternatively, we could use the RLS-DCD algorithm and the dynamically regularized RLS-DCD algorithm to solve the systems of equations for the MVDR DoA estimation. The DCD-based MVDR DoA estimator could be configured in three different modes using the hardware blocks we have presented in Chapter 6. The fastest mode employs M RLS-DCD cores for M angles needed to listen to and these RLS-DCD cores share one correlation matrix estimation module. At each time instant, these cores solve the systems of equations for different angles simultaneously and find out directions of source signals. This configuration obtains a high throughput at the cost of large area usage. The slowest mode uses only one RLS-DCD core. At each time instant, it scans from one side to the other, angle by angle sequentially. The area usage of this configuration is about M times lower than that of the fastest mode. However, the throughput is also about M times lower. The third mode uses a small number of RLS-DCD cores than in the fastest mode. These cores could scan the angle range using the efficient search method, which is similar to the search method used in [107]. Comparing with other two modes, this configuration needs smaller number of cores than in the fastest mode and achieves higher throughput than in the slowest mode.

3) As we have mentioned in Chapter 2, the NLMS algorithm is well known due to its low computational complexity and robustness. However, one of the major limitations of the NLMS algorithm is its low convergence speed for colored input signals [6]. The AP adaptive filter can be treated as a generalization of the NLMS algorithm that reuses past and current information [6] to balance the convergence speed and the computational com-

plexity [33] of the NLMS algorithm. Therefore, the AP adaptive filter may be viewed as an intermediate adaptive filter between the NLMS filter and the RLS filter. Consequently, comparing with the NLMS filter, the AP filter provides a significant improvement in the convergence speed. However, it is still complicated for implementation. The fast AP (FAP) adaptive algorithm allows a significant simplification [108] of the AP algorithm. However, it requires matrix inversion which is complicated and numerically unstable. Many iterative techniques were proposed for matrix inversion in the FAP algorithm, such as the CG iterations [109] and the Gauss-Seidel iterations [31] [110]. On the other hand, the FAP algorithm based on recursive matrix inversion is computationally efficient only if the step size, which controls the convergence speed and the steady-state output (residual) error, is close to one [108]. However, the step size should be reduced to reduce the residual error after the algorithm has converged [108]. Moreover, when the step size is close to one, the FAP algorithm is sensitive to the input noise [108].

In [108], an efficient numerical implementation of the FAP algorithm with an arbitrary step size based on DCD iterations is proposed. The proposed DCD-FAP algorithm is simple, computationally stable and well-suited to real-time hardware implementation. It also demonstrates performance close to that of the ideal FAP algorithm which computes the matrix inversion, and the complexity smaller than that of the Gauss-Seidel-FAP algorithms [108]. Recently, a low complexity implementation of the AP algorithm was proposed in [111] by incorporating the DCD algorithm into the filter update of the AP algorithm to solve the systems of equations. Comparing with the DCD-FAP algorithm which employs the DCD algorithm to solve the system of equations directly, the complexity of this DCD-AP algorithm is reduced significantly. It allows the performance of the AP algorithm to be approached with a complexity that can be even smaller than the NLMS complexity. Furthermore, the proposed DCD-AP algorithm is well-suited to hardware implementation as it is division-free and the filter update is multiplication-free. The performance of the DCD-AP algorithm can be made arbitrarily close to that of the ideal AP algorithm. However, both the DCD-FAP algorithm and the DCD-AP algorithm have not been implemented in hardware. By using the architectures and implementations of the DCD algorithm and the RLS-DCD algorithm presented in this thesis, one could implement the DCD-FAP algorithm and the DCD-AP algorithm in hardware.

Bibliography

[1] G. H. Golub and C. F. Van Loan, *Matrix computations*, The Johns Hopkins University Press, Baltimore, 3rd edition, 1996.

[2] D. S. Watkins, *Fundamentals of matrix computations*, Hoboken, N. J.: Wiley, 2002.

[3] T. K. Moon and W. C. Stirling, *Mathematical methods and algorithms for signal processing*, Prentice Hall, 2000.

[4] R. Uribe and T. Cesear, "Implementing matrix inversions in fixed-point hardware", *Xilinx DSP Magazine*, vol. 1, no. 1, pp. 32–25, Oct. 2005.

[5] Y. V. Zakharov and T. C. Tozer, "Multiplication-free iterative algorithm for LS problem", *Electronics Letters*, vol. 40, no. 9, pp. 567–569, April 2004.

[6] S. Haykin, *Adaptive filter theory*, Prentice Hall, Inc. New Jersey, 4th edition, 2001.

[7] A. H. Sayed, *Fundamentals of adaptive filtering*, Hoboken, N. J.: Wiley, 2003.

[8] S. Verdu, *Multiuser detection*, Cambridge University Press, 1998.

[9] A. M. Kondoz, *Digital speech: coding for low bit rate communication systems*, Wiley, 2nd edition, Nov. 2004.

[10] J. Proakis and D. Manolakis, *Digital signal processing: principles, algorithms, and applications*, Macmillan, New York, 2nd edition, 1992.

[11] S. D. Muruganathan and A. B. Sesay, "A QRD-RLS-based predistortion scheme for high-power amplifier linearization", *IEEE Trans. Circuits and Systems - II: Express Briefs*, vol. 53, no. 10, pp. 1108–1112, 2006.

[12] "LAPACK – linear algebra PACKage", `http://www.netlib.org/lapack`, Feb. 2007, Version 3.1.1.

[13] J. R. White, "Mathematical methods VI. Numerical solution of algebraic equations - the LU decomposition method", *Lecture Notes, University of Massachusetts Lowell, USA*, http://www.tmt.ugal.ro/crios/Support/ANPT/Curs/math/s6/s6lud/s6lud.html.

[14] B. Louka and M. Tchuente, "Givens elimination on systolic array", *Proceedings of the 2nd International Conference on Supercomputing, St. Malo, France*, pp. 638–647, 1988.

[15] M. Karkooti, J. Cavallaro, and C. Dick, "FPGA implementation of matrix inversion using QRD-RLS algorithm", *Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, California*, pp. 1625–1629, Nov. 2005.

[16] M. Myllyla, J.-M. Hintikka, J. R. Cavallaro, M. Juntti, M. Limingoja, and A. Byman, "Complexity analysis of MMSE detector architectures for MIMO OFDM systems", *Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, California*, pp. 75–81, Nov. 2005.

[17] Z. Liu, J. V. McCanny, G. Lightbody, and R. L. Walke, "Generic SoC QR array processor for adaptive beamforming", *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 50, no. 4, pp. 169–175, April 2003.

[18] F. Edman and V. Owall, "A scalable pipelined complex valued matrix inversion architecture", *IEEE International Symposium on Circuits and Systems (ISCAS), Kobe, Japan*, vol. 5, pp. 4489–4492, May 2005.

[19] G. Lightbody, R. Woods, and R. L. Walke, "Design of a parameterizable silicon intellectual property core for QR-based RLS filtering", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 4, pp. 659–678, Aug. 2003.

[20] R. Dohler, "Squared Givens rotation", *IMA Journal of Numerical Analysis*, , no. 11, pp. 1–5, 1991.

[21] X. J. Wang, "Variable precision floating-point divide and square root for efficient FPAG implementation of image and signal processing algorithms", *PhD thesis, Northeastern University, USA*, Dec. 2007, http://www.ece.neu.edu/groups/rcl/theses/xjwang_phd2007.pdf.

[22] J. Eilert, W. Di, and D. Liu, "Efficient complex matrix inversion for MIMO Software Definited Radio", *IEEE International Symposium on Circuits and Systems (ISCAS), New Orleans, LA, US.*, pp. 2610–2613, May 2007.

[23] J. Eilert, D. Wu, and D. Liu, "Implementation of a programmable linear MMSE detector for MIMO-OFDM", *IEEE International Conference on Acoustics, Speech,*

and Signal Processing (ICASSP), Las Vegas, Nevada, US., pp. 5396–5399, March 2008.

[24] I. LaRoche and S. Roy, "An efficient regular matrix inversion circuit architecture for MIMO processing", *IEEE International Symposium on Circuits and Systems (ISCAS), Island of Kos, Greece*, May 2006.

[25] Y. Jia, C. Andrieu, R. J. Piechocki, and M. Sandell, "Guassian approximation based mixture reduction for near optimum detection in MIMO systems", *IEEE Communication Letters*, vol. 9, no. 11, pp. 997–999, Nov. 2005.

[26] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Vorst, *Templates for the solution of linear systems: building blocks for iterative methods*, SIAM, 2nd edition, 1994.

[27] G. K. Boray and M. D. Srinath, "Conjugate gradient techniques for adaptive filtering", *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 39, no. 1, pp. 1–10, 1992.

[28] P. S. Chang and A. N. Willson Jr., "Analysis of conjugate gradient algorithms for adaptive filtering", *IEEE Transactions on Signal Processing*, vol. 48, no. 2, pp. 409–418, 2000.

[29] M. Fukumoto, T. Kanai, H. Kubota, and S. Tsujii, "Improvement in the stability of the BCGM-OR algorithm", *Electronics and Communications in Japan, Part 3*, vol. 83, no. 5, pp. 42–52, 2000.

[30] O. Diene and A. Bhaya, "Adaptive filtering algorithms designed using control Liapunov functions", *IEEE Signal Processing Letters*, vol. 13, no. 4, pp. 224–227, 2006.

[31] F. Albu, J. Kadlec, N. Coleman, and A. Fagan, "The Gauss-Seidel fast affine projection algorithm", *IEEE Workshop on Signal Processing Systems (SiPS), San Diego, CA, US.*, pp. 109–114, Oct. 2002.

[32] F. Albu and M. Bouchard, "The Gauss-Seidel fast affine projection algorithm for multichannel active noise control and sound reproduction systems", *International Journal of Adaptive Control and Signal Processing*, vol. 19, no. 2-3, pp. 107–123, 2005.

[33] S. Werner, "Reduced complexity adaptive filtering algorithms with applications to communications systems", *PhD thesis, Helsinki University of Technology, Finland*, Nov. 2002, `http://lib.tkk.fi/Diss/2002/isbn9512260875/isbn9512260875.pdf`.

[34] S. Stergiopoulos, *Advanced signal processing hand book, theory and implementation for radar, sonar, and medical imaging real-time systems*, CRC Press, 2000.

[35] B. Farhang-Boroujeny, *Adaptive filters theory and applications*, Hoboken, N. J.: Wiley, 1998.

[36] D. Boppana, K. Dhanoa, and J. Kempa, "FPGA based embedded processing architecture for the QRD-RLS algorithm", *Proceeding of the 12th Annual IEEE Symposium on Filed-Programmable Custom Computing Machines (FCCM'04)*, pp. 330–331, Apr. 2004.

[37] M. P. Fitton, S. Perry, and R. Jackson, "Reconfigurable antenna processing with matrix decomposition using FPGA based application specific integrated processors", *Software Defined Radio Technical Conference and Product Exposition (SDR'04), Phoenix, Ariz, US.*, Nov. 2004.

[38] C. Dick, F. Harris, M. Pajic, and D. Vuletic, "Implementing a real-time beamformer on an FPGA platform", *XCell Journal*, , no. 60, pp. 36–40, April 2007.

[39] R. Matousek, M. Tichý, Z. Pohl, J. Kadlec, C. Softley, and N. Coleman, "Logarithmic number system and floating-point arithmetics on FPGA", *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications, Montpellier(La Grande-Motte), France*, pp. 627–636, Sept. 2002.

[40] J. Schier and A. Hermanek, "Using logarithmic arithmetic for FPGA implementation of the Givens rotations", *Proceedings of the Sixth Baiona Workshop on Signal Processing in Communications, Baiona, Spain*, pp. 199–204, Sept. 2003.

[41] J. Schier and A. Hermanek, "Using logarithmic arithmetic to implement the recursive least squares (QR) algorithm in FPGA", *14th International Conference on Field-Programmable Logic and Applications, Leuven, Belgium*, pp. 1149–1151, Auguest 2004.

[42] Xilinx Inc., "System Generator for DSP", `http://www.xilinx.com/ise/optional_prod/system_generator.htm`.

[43] Xilinx Inc., "Core Generator", `http://www.xilinx.com/products/design_tools/logic_design/design_entry/coregenerator.htm`.

[44] Xilinx Inc., "Virtex-4 libraries guide for HDL designs", `http://toolbox.xilinx.com/docsan/xilinx8/books/docs/v4ldl/v4ldl.pdf`.

[45] Altera Inc., "Application Note 263: CORDIC reference design v1.4, June 2005", http://www.altera.com/literature/an/an263.pdf.

[46] Altera Inc., "Smart antennas - beamforming", http://www.altera.com/end-markets/wireless/advanced-dsp/beamforming/wir-beamforming.html.

[47] Altera Inc., "Accelerating WiMAX system design with FPGAs", http://www.altera.com/literature/wp/wp_wimax.pdf.

[48] Altera Inc., "Altera wireless solutions channel card series cellular infrastructure", http://www.altera.com/literature/po/ss_wrless_cellular_infra.pdf.

[49] C. Carmichael, E. Fuller, P. Blain, and M. Caffrey, "SEU mitigation techniques for Virtex FPGAs in space applications", *MAPLD 1999 - Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference, 2nd, Johns Hopkins University , APL, Laurel, MD, USA*, Sept. 1999.

[50] Altera Inc., "Comparing Altera APEX 20KE and Xilinx Virtex-E logic densities", http://www.altera.com/products/devices/apex/features/apxcompdensity.html.

[51] "Xilinx LogiCORE CORDIC v3.0", *Xilinx Product Specification DS249*, Apr. 2005.

[52] Xilinx Inc., "AccelDSP synthesis tool", http://www.xilinx.com/ise/dsp_design_prod/acceldsp.

[53] T. Hill, "Researching FPGA implementations of baseband MIMO algorithms using AccelDSP", http://www.xilinx.com/products/design_resources/dsp_central/resource/wimax_mimo_acceldsp.pdf, April 2007.

[54] J. N. Coleman, E. I. Chester, C. I. Softley, and J. Kadlec, "Arithmetic on the European logarithmic microprocessor", *IEEE Transactions on Computer*, vol. 49, no. 7, pp. 702–715, 2000.

[55] Xilinx Inc., "Synthesis technology", www.xilinx.com/products/design_tools/logic_design/synthesis/xst.htm.

[56] A. Li, D. B. Sharp, and B. J. Forbes, "Improving the high frequency content of the input signal in acoustic pulse reflectometry", *Proceedings of the International Symposium on Musical Acoustics, Perugia, Italy*, pp. 391–394, Sep. 2001.

[57] S. R. Dicker, S. J. Melhuish, R. D. Davies, C. M. Gutierrez, R. Rebolo, D. L. Harrison, R. J. Davis, A. Wilkinson, R. J. Hoyland, and R. A. Watson, "CMB observations with the Jodrell Bank - IAC interferometer at 33 GHz", *Monthly Notices of the Royal Astronomical Society*, vol. 309, no. 3, pp. 750–760, Nov. 1999.

[58] Reindeer Graphics Inc., "Foveapro interactive deconvolution", `http://www.reindeergraphics.com`, 2005.

[59] J. Burke and H. Helmers, "Complex division as a common basis for calculating phase differences in electronic speckle pattern interferometry in one step", *Applied Optics*, vol. 37, no. 13, pp. 2589–2590, 1998.

[60] G. Vandersteen, F. Verbeyst, P. Wambacq, and S. Donnay, "High-frequency non-linear amplifier model for the efficient evaluation of inband distortion under non-linear loadpull conditions", *58th ARFTG Conference Digest: RF Measurements for a Wireless World*, Nov. 2001.

[61] S. Blackford, "Singular value decomposition", `http://www.netlib.org/lapack/lug/node53.html`, Oct. 2005.

[62] B. J. Weaver, "Computationally-efficient signal processing algorithms for communications systems", *PhD thesis, University of York, UK*, Sep. 2008.

[63] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin, B. J. Thompson, T. Tung, and D. J. Yoo, "Design, implementation and testing of extended and mixed precision BLAS", *ACM Transactions on Mathmatical Software*, vol. 28, no. 2, pp. 152–205, June 2002.

[64] R. L. Smith, "Algorithm 116: complex division", *Communication of the ACM*, vol. 5, no. 8, pp. 435, 1962.

[65] M. D. Ercegovac and J. M. Muller, "Design of a complex divider", *Proceedings of IEEE International Conference on Application-Specific Systems, Architectures, and Processors, The Hague, The Netherlands*, pp. 304–314, June 2003.

[66] M. D. Ercegovac and J. M. Muller, "Complex division with prescaling of operands", *IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pp. 304–314, June 2003.

[67] M. J. Schulte and J. E. Stine, "Approximating elementary functions with symmetric bipartite tables", *IEEE Transactions on Computers*, vol. 48, no. 8, pp. 842–847, Aug. 1999.

[68] B. J. Weaver, Y. V. Zakharov, and T. C. Tozer, "Multiplication-free division of complex numbers", *Sixth IMA Conference on Mathematics in Signal Processing, Cirencester, UK*, pp. 211–214, Dec. 2004.

[69] J. Litva and T. K-Y. Lo, *Digital beamforming in wireless communications*, Artech House, 1st edition, 1996.

[70] J. Capon, "High-resolution frequency-wavenumber spectrum analysis", *Proceedings of the IEEE*, vol. 57, no. 8, pp. 1408–1418, Aug. 1967.

[71] L. C. Godara, "Application of antenna arrays to mobile communications, part II: Beam-forming and direction-of-arrival considerations", *Proceedings of the IEEE*, vol. 85, pp. 1195–1245, Aug. 1997.

[72] J. Li and P. Stoica, *Robust adaptive beamforming*, Hoboken, N. J.: Wiley, 2005.

[73] E. Horita, K. Sumiya, H. Urakami, and S. Mitsuishi, "A leaky RLS algorithm: its optimality and implementation", *IEEE Transactions on Signal Processing*, vol. 52, no. 10, Oct. 2004.

[74] Wikipedia, "Field-programmable gate array", `http://en.wikipedia.org/wiki/FPGA`, June 2008.

[75] C. H. Dick and H. M. Pedersen, "Design and implementation of high-performance FPGA signal processing datapaths for software defined radios", *VEMbus Systems*, Aug. 2001.

[76] Xilinx Inc., "Virtex-II Pro and Virtex-II Pro X platform FPGAs: complete data sheet, Xilinx datasheet DS083 (v4.5), Oct. 2005", `http://direct.xilinx.com/bvdocs/publications/ds083.pdf`.

[77] Xilinx Inc., "Xilinx XUP Virtex-II pro development system", `http://www.xilinx.com`.

[78] T. Cesear and R. Uribe, "Exploration of least-squares solutions of linear systems of equations with fixed-point arithmetic hardware", *Software Defined Radio Technical Conference, Orange County, CA, US.*, Nov. 2005.

[79] V. V. Zaharov and M. Teixeira, "SMI-MVDR beamformer implementations for large antenna array and small sample size", *IEEE Trans. Circuits and Systems - I: Regular Papers, to be published*, 2008.

[80] A. Elnashar, S. Elnoubi, and A. El-Mikati, "Performance analysis of blind adaptive MOE multiuser receivers using inverse QRD-RLS algorithm", *IEEE Trans. Circuits and Systems - I: Regular Papers*, vol. 55, no. 1, pp. 398–411, 2008.

[81] H. H. Chen, S. C. Chan, and K. L. Ho, "Adaptive beamforming using frequency invariant uniform concentric circular arrays", *IEEE Trans. Circuits and Systems - I: Regular Papers*, vol. 54, no. 9, pp. 1938–1949, 2007.

[82] Z. J. Shi and J. Shen, "Convergence of nonmonotone line search method", *Journal of Computational and Applied Mathematics*, vol. 193, pp. 397–412, 2006.

[83] S. Boyd and L. Vandenberghe, *Convex optimization*, Cambridge University Press, 2006.

[84] U. Fawer and B. Aazhang, "A multiuser receiver for code division multiple access communications over multipath channels", *IEEE Transactions on Communications*, vol. 43, no. 2/3/4, pp. 1556–1565, 1995.

[85] T. Bose and G. F. Xu, "The Euclidean direction search algorithm in adaptive filtering", *IEICE Transactions on Fundamentals*, vol. E85-A, no. 3, pp. 532–539, March 2002.

[86] D. K. Faddeev and B. N. Faddeev, *Numerical methods of linear algebra*, Lan, Saint Petersburg, 2002, (in Russian).

[87] A. Bateman and I. Paterson-Stephens, *The DSP handbook: algorithms, applications and design techniques*, Prentice Hall, 2002.

[88] Xilinx Inc., "Xilinx LogicCore multiplier V10.1, Xilinx datasheet DS255, April, 2008", `http://www.xilinx.com/support/documentation/ip_documentation/mult_gen_ds255.pdf`.

[89] S. F. Cotter and B. D. Rao, "Sparse channel estimation via matching pursuit with application to equalization", *IEEE Transactions on Communications*, vol. 50, no. 3, pp. 374–377, 2002.

[90] W. C. Wu and K. C. Chen, "Identification of active users in synchronous CDMA multiuser detection", *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 9, pp. 1723–1735, 1998.

[91] Y. Zakharov and F. Albu, "Coordinate descent iterations in fast affine projection algorithm", *IEEE Signal Processing Letter*, vol. 12, no. 5, pp. 353–356, May 2005.

[92] Y. Zakharov, G. White, and J. Liu, "Low complexity RLS algorithms using dichotomous coordinate descent iterations", *IEEE Transactions on Signal Processing*, vol. 56, no. 7, pp. 3150–3161, July 2008.

[93] H. S. Kim, W. Zhu, K. Mohammed, A. Shah, and B. Daneshrad, "An efficient FPGA based MIMO-MMSE detector", *15th European Signal Processing Conference, Poznan, Poland*, pp. 1131–1135, Sept. 2007.

[94] S. Noh, Y. Jung, S. Lee, and J. Kim, "Low-complexity symbol detector for MIMO-OFDM-based wireless LANs", *IEEE Trans. Circuits and Systems - II: Express Briefs*, vol. 53, no. 12, pp. 1403–1407, 2006.

[95] Y. Jung, J. Kim, S. Lee, H. Yoon, and J. Kim, "Design and implementation of MIMO-OFDM baseband for high-speed wireless LANs", *IEEE Trans. Circuits and Systems - II: Express Briefs*, vol. 54, no. 7, pp. 631–635, 2007.

[96] J. G. McWhirter and T. J. Shepherd, "Systolic array processor for MVDR beamforming", *IEE Proceedings: F - Radar and Signal Processing*, vol. 136, no. 2, pp. 75–80, Apr. 1989.

[97] J. Ma, K. K. Parhi, and E. F. Deprettere, "Annihilation-reordering look-ahead pipelined CORDIC-based RLS adaptive filters and their application to adaptive beamforming", *IEEE Transactions on Signal Processing*, vol. 48, pp. 2414–431, Aug. 2000.

[98] I. D. Skidmore and I. K. Proudler, "The KaGE RLS algorithm", *IEEE Transactions on Signal Processing*, vol. 51, no. 12, pp. 3094–3104, Dec. 2003.

[99] J. H. Husoy, *Lecture notes in computer science: numerical analysis and its applications*, vol. 3401/2005, chapter "Adaptive filters viewed as iterative linear equation solvers", pp. 320–327, Springer-Verlag Berlin Heidelberg, 2005.

[100] T. Bose and M. Q. Chen, "Conjugate gradient method in adaptive bilinear filtering", *IEEE Transactions on Signal Processing*, vol. 43, no. 6, pp. 1503–1508, 1995.

[101] G. F. Xu and T. Bose, "Analysis of the Euclidean direction set adaptive algorithm", *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Seattle, Washington, USA*, vol. 3, pp. 1689–1692, May 1998.

[102] M. Q. Chen, *Applied and computational control, signals and circuits: recent developments*, chapter "A direction set based algorithm for adaptive least squares problems in signal processing", pp. 213–236, Kluwer Academic Publishers, 2001.

[103] C. E. Davila, "Line search algorithms for adaptive filtering", *IEEE Transactions on Signal Processing*, vol. 41, no. 7, pp. 2490–2494, 1993.

[104] H. L. V. Trees, *Optimum Array Processing, Part IV of Detection, Estimation, and Modulation Theory*, Hoboken, N. J.: Wiley, 2002.

[105] S. L. Gay, "Dynamically regularized fast RLS with application to echo cancellation", *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 2, pp. 957–960, 1996.

[106] S. L. Gay and J. Benesty, *Acoustic signal processing for telecommunication*, Norwell: Kluwer Academic Publishers, 2001.

[107] Y. V. Zakharov and T. C. Tozer, "Frequency estimator with dichotomous search of periodogram peak", *Electronics Letters*, vol. 35, no. 19, pp. 1608–1609, Sept. 1999.

[108] Y. Zakharov and F. Albu, "Coordinate descent iterations in fast affine projection algorithm", *IEEE Signal Processing Letter*, vol. 12, no. 5, pp. 353–356, May 2005.

[109] H. Ding, "A stable fast affine projection adaptation algorithm suitable for low-cost processors", *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Turkey*, vol. 1, pp. 360–363, 2000.

[110] E. Chau, H. Sheikhzadeh, and R. Brennan, "Complexity reduction and regularization of a fast affine projection algorithm for oversampled subband adaptive filters", *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), QC, Canada*, vol. 5, pp. 109–112, May 2004.

[111] Y. V. Zakharov, "Low complexity implementation of the affine projection algorithm", *IEEE Signal Processing Letters*, vol. 12, no. 5, pp. 353–356, April 2008.

List of Figures

List of Tables