Master's Thesis

Reconfigurable Computing for Space-Time Adaptive Processing

Nikhil D. Gupta

Department of Computer Science

Texas Tech University

August 1998

ACKNOWLEDGEMENTS

Today when I look in retrospect to the last two years I find that there is a lot that I have learnt. It is just as Plato has said “Those having torches will pass them on to others.” It is for this learning that I feel deeply indebted to all my teachers.

I would like to express my sincere gratitude and respect to Dr. John K. Antonio, my committee chairperson, for his invaluable guidance and encouragement. Without his knowledge and experience I am sure that this research would not have been as successful.

I would also like to thank Dr. Noe Lopez-Benitez and Dr. William M. Marcy for participating on my committee and for their invaluable guidance and advice.

The successful completion of this thesis is due largely to the support of my colleagues at High Performance Computing Lab. I thank Jack West, Tim Osmulski, and Jeff Muehring for constant support, encouragement, and help. I am especially thankful to Jack for the countless hours he spent adding binary numbers with me. I am thankful to Jeff for being a faithful friend and a source of encouragement and support and all the help with Matlab. I am also thankful to Marcelia Sawyers the HPCL secretary for all her help.

This research was supported by DARPA under contract number F30602-97-2-0297.

Finally, I would like to thank my parents, Dilip and Manda Gupta, and my brother Harshal for their constant encouragement and support throughout my life and making me what I am today.

CONTENTS

ABSTRACT

The output of space-time adaptive processing (STAP) is a weighted sum of multiple radar returns, where the weights for each return in the sum are calculated adaptively and in real-time. The most computationally intensive portion of most STAP approaches is the calculation of the adaptive weight values. Calculation of the weights involves solving a set of linear equations based on an estimate of the covariance matrix associated with the radar return data. The traditional approach for computing the adaptive weights is based on a direct method called QR-decomposition. This method has a fixed computational complexity, which depends on the size of the equation matrix and provides the exact solution. An alternative approach based on an iterative method called Conjugate Gradient is proposed, which allows for trading off accuracy for reduced computational complexity. The two approaches are analyzed and compared.

Existing computational strategies for STAP typically rely on the use of multiple digital signal processors (DSPs) or general-purpose processors (GPPs). An alternative strategy is proposed, which makes use of Field Programmable Gate Arrays (FPGAs) as vector co-processors that perform inner product calculations. Two different "inner-product co-processor" designs are introduced for use with a host DSP or GPP. The first has a multiply-and accumulate structure and the second uses a reduction-style tree structure having two multipliers and an adder. The proposed strategies are implemented and compared to the traditional method.

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Motivation for Space-Time Adaptive Processing

Modern airborne radar platforms are required to provide long-range detection of smaller and smaller targets in the presence of severe interference from both natural and artificial sources. This detection of targets is often performed over land, where ground clutter can be very high [1], and in the presence of electronic countermeasures such as jamming [1, 2]. These radar platforms must have the capability to nullify both clutter and jamming to below the ambient noise level.

The suppression of jamming and clutter has posed a problem to radar engineers since the beginning of radar. Over the years, many techniques have been developed to try and eliminate jamming and clutter; however, the problem is difficult because it is dependent on a number of different inter-related variables. A potential target may be obscured not only by the mainlobe clutter (i.e., the clutter that originates from the same angle as the target) but also by the sidelobe clutter (i.e., the clutter that comes from different angles but has the same Doppler frequency) [2]. Displaced-phase-center-antenna (DPCA) processing was developed to address the problem of clutter in airborne radar platforms [3]. The effects of jamming on radar systems can often be successfully cancelled by adaptive array processing techniques [4].

The above two techniques – DPCA and adaptive array processing – individually provide a partial solution to the problem of clutter and jamming, respectively. These two techniques have been effectively combined in a technique known as Space-Time

Adaptive Processing (STAP), which can be viewed as a generalization of DPCA processing. STAP simultaneously and adaptively combines the signals received on multiple elements of an antenna array – the spatial domain – and from multiple pulse repetition periods – the temporal domain.

STAP offers the potential to improve airborne radar performance in several areas. STAP algorithms can provide improved target detection in the presence of interference through the adaptive nulling of both ground clutter and signal jamming [5]. It can improve low velocity target detection through better mainlobe clutter suppression. It can also be used to detect small targets, which would otherwise be obscured by the presence of sidelobe clutter. STAP also provides a capability to cancel non-stationary interference. Thus, STAP combines both spatial and temporal adaptive processing techniques to cancel out the clutter and interference contained in the radar signals received by an airborne antenna array.

Another significant feature of STAP is that it improves the performance of the antenna array while requiring little or no modification to the basic radar design. However, the computational complexity associated with STAP is generally very high; an extremely large amount of data needs to be processed in real-time. This in turn requires a large computational throughput.

1.2 Candidate Computational Technologies for STAP

The computational requirements of STAP algorithms are well suited for execution on digital signal processors (DSPs), which are special purpose microprocessors optimized to perform arithmetic operations such as multiplication, addition, and subtraction with high efficiency. In addition to their increased performance for certain computations, DSPs are generally silicon conservative, less expensive, and more power efficient than comparable general-purpose microprocessors. Thus, DSPs are often a better choice than general-purpose processors for embedded applications that have strict size, weight, and power (SWAP) restrictions.

Even though DSPs are well-suited for embedded systems, their architectures are still somewhat generic, which means they may have more silicon complexity than is absolutely necessary for any given application. For example, an application specific integrated circuit (ASIC) designed for a given application is generally better matched (i.e., has less complexity and/or better performance) for that application than does a DSP. However, some disadvantages of ASICs include their high cost of design, which makes them ineffective in terms of cost when they need to be produced in small volumes, and their inflexibility (i.e., they cannot generally be re-used for other applications).

An alternative to both DSPs and ASICs is the use of reconfigurable computing devices, which can provide performance near ASIC levels while having programming flexibility similar to DSPs. Reconfigurable computing is usually based on field programmable gate array (FPGA) technology. Because FPGA chips are commercially available, reconfigurable systems based on FPGAs can be developed at a fraction of the cost associated with using ASICs. The recent popularity of reconfigurable systems is consistent with the growing trend toward utilizing commercial-off-the-shelf (COTS) hardware in place of custom-designed ASICs for military applications [6]. The feature of being able to reconfigure FPGAs also allows for the possible use of one system for many different applications.

An FPGA device typically consists of an array of programmable logic blocks interconnected by a programmable routing fabric. The task of "programming" an FPGA is actually similar to that of designing an ASIC in the sense that the programming of an FPGA is expressed through a hardware description language (HDL) such as VHDL [7]. The designer's HDL code is compiled into a binary file called the "bit-stream," which is targeted for a particular FPGA part. The bit-stream defines the internal programming of both the logic blocks and routing resources within the FPGA in order to implement the HDL design.

FPGAs and ASICs are particularly well-suited for embedded applications in which a stream of input data must be processed. In such applications, the required computations are often deterministic, primarily involving numerical operations. Thus, when compared to DSPs, the use of FPGAs and ASICs can provide improvements in speed and throughput by exploiting parallelism and eliminating the overhead associated with load/store operations, branch operations, and instruction decoding.

1.3 Focus of the Thesis

The most computationally intensive part of STAP algorithms is typically the calculation of adaptive weights, which are used in combining the multiple returns (across both time and space). Traditionally the adaptive weights for STAP are computed using the QR-decomposition [8] approach, which is a direct matrix solver that gives exact solutions. QR-decomposition involves a fixed number of floating point operations determined by the size of the equation matrix. For most applications in radar processing precise answers may not be required; approximate answers are sufficient. In such cases it would be very effective in terms of computations and time to compute approximate answers.

This research involves two distinct objectives for improving the performance of STAP processing. The first objective is to investigate alternate approaches to compute the adaptive weights, in which the accuracy of the answers can be traded for the associated computations. Two approaches are investigated and compared: the traditional QR-decomposition technique and a newly proposed approach based on the conjugate gradient (CG) method [8]. The second goal is to use reconfigurable computing platforms to perform a part of the core computations needed in both the QR and CG approaches; improving the throughput of the system as well as the overall characteristics (e.g., size, weight, and power) of the system. The core computation implemented with reconfigurable computing is the calculation of the inner products.

The reconfigurable computing platforms usually use FPGAs as the reconfigurable logic components. The use of reconfigurable hardware maybe divided into three categories:

1. Logic – where FPGAs are used to implement glue logic.

2. Embedded Computing – where FPGA-based reconfigurable co-processors are used along with DSP or general-purpose processors (GPPs) to perform computationally intensive part(s) of an algorithm.

3. General Computing – where completely reconfigurable computing platforms based on FPGAs are used in a system designed for general purpose computing.

In this thesis, the focus is on the use of the second category. The target FPGA platform is the WildOne computing engine made by Annapolis Micro Systems, Annapolis MD [9].

A major challenge of implementing core components of STAP algorithms on FPGA-based computing engines is determining the architectural requirements to perform the these computations. The details related to the size and the optimal number of adder and multiplier circuits are investigated. Because these architectural design details are flexible, the most appropriate configuration depends on the data characteristics (e.g., dynamic range and accuracy) and the data representation used (e.g., floating point vs. block floating point). A basic goal is to investigate the accuracy that can be achieved with the use of block floating point operations instead of floating point operations for a given dynamic range of numbers.

The remainder of this thesis is organized in the following manner. Chapter II gives an overview of radar signal processing and the computational complexity analysis of two known STAP algorithms, namely fully-adaptive STAP and partially-adaptive STAP. Studies conducted to evaluate two different approaches (QR vs. CG) to compute the adaptive weights associated with partially adaptive STAP algorithm are discussed in Chapter III. Chapter IV briefly introduces the basic components of the WildOne reconfigurable computing board that is being used in this work. In Chapter V, two alternate architectures for GPP/FPGA implementations are illustrated along with the discussion of a general design methodology for the design of hardware/software co-system and the use of Unified Modeling Language (UML) [10] to model the system effectively. The common attribute of both the architectures is that the FPGA component serves as an inner product co-processor to the GPP. The architectures differ in how the inner product calculations are performed on the FPGA. Chapter VI presents some numerical studies conducted on the two architectures. Finally, Chapter VII concludes the work with a summary of the research completed, the results and the future work.

CHAPTER II

SPACE-TIME ADAPTIVE PROCESSING

2.1 Introduction

The concept of radar dates back to the 1880's when Hertz first demonstrated that radio waves reflected from metallic and dielectric objects. However, radar technology did not come into its own until its widespread development and application during World War II. Since then, the use of radar has increased phenomenally. Today, radar technology is being used in a range of military, commercial, and private applications.

Modern airborne radar systems are required to detect smaller and smaller targets in the presence of clutter and interference. The cancellation of ground clutter and jamming interference from radar returns has been the topic of research over the years. STAP algorithms were developed to extract desired target signals from returns comprised of Doppler shifts (associated with radar platform motion), ground clutter returns, and jamming interference.

The following sections describe the principles of modern radar systems and the major components of typical STAP algorithms, giving a brief overview of the different stages of computation that are generally required. For a more thorough analysis of STAP algorithms, the reader is referred to [5]. The following section on radar fundamentals borrows extensively from [11] that gives a more complete overview of radar.

2.2 Radar Fundamentals

The fundamental purpose of radar is to detect the presence of an object of interest and provide information concerning that object's range, velocity, angular coordinates, size, and other parameters [12]. The basic principle of radar is very simple. An elementary form of radar consists of a transmitting antenna and a receiving antenna. Radar operates by radiating electromagnetic (EM) energy, oscillating at a predetermined frequency, f, and duration, τ, into free space through the transmitting antenna. In general, the radar antenna forms a beam of EM energy that concentrates the EM wave into a given direction [13]. By effectively rotating and pointing the antenna, the transmitted radar signal can be directed to a desired angular coordinate.

A portion of the radar's transmitted energy is intercepted by an object located in the path of the transmitted beam and is scattered in all directions depending on the target's physical characteristics. In general, some of the transmitted energy will be reflected back in the direction of the radar. This retro-reflected energy is referred to as backscatter [13]. A portion of the backscattered wave, or echo return, is received by the radar antenna. The echo returns, which are gathered by a set of sensors, are sampled, and the resulting data is processed to identify targets and parameter estimation.

The distance to the target is determined by measuring the time taken for the radar signal to travel to the target and back. Furthermore, the angular position of the target may be determined by the arrival direction of the backscattered wave. If relative motion exists between the target and radar, the shift in the carrier frequency of the reflected wave, also known as the Doppler effect, is a measure of the target's relative velocity and may be used to distinguish moving targets from stationary objects [2].

The basic role of the radar antenna is to act as a transducer between the free-space propagation and guided-wave propagation of the EM wave [14]. The specific function of the antenna during transmission is to concentrate the radiated energy into a shape beam directive that illuminates targets in a desired direction. During reception, the antenna collects the energy from the reflected echo returns. Many varieties of radar antennas have been used in radar systems. The type of radar antenna selected for a certain application depends not only on the electrical and mechanical requirements dictated by the radar design specifications but also on its application. In airborne-radar applications, radar antennas must generate beams with shape directive patterns that can be scanned.

The properties offered by antenna arrays are quite appealing to airborne radar systems. Antenna arrays consist of multiple stationary elements, which are fed coherently, and use phase or time-delay control at each element to scan a beam to given angles in space [15]. The primary reason for using radar arrays is to produce a directive beam that can be repositioned electronically. An electronically steerable antenna array, whose beam steering is inertialess, is drastically more cost effective when the mission requires surveying large solid angles while tracking a large number of targets [15]. Additionally, arrays are sometimes used in place of fixed aperture antennas because the multiplicity of elements allows a more precise control of the radiating pattern.

The purpose of moving-target indication (MTI) radar is to reject signal returns from stationary or unwanted slow-moving targets, such as buildings, hills, trees, sea, rain, and snow, and retain detection information on moving targets such as aircraft and missiles [16]. The term Doppler radar refers to any radar capable of measuring the shift between the transmitted frequency and the frequency of reflections received from possible targets [17]. Relative motion between a signal source and a receiver creates a Doppler shift of the source frequency. When a radar system intercepts a moving object that has a radial velocity component relative to the radar, the reflected signal's frequency is shifted.

To illustrate the Doppler effect, consider a radar that emits a pulse of EM energy that is intercepted by both a building (fixed target) and an airplane (moving target) approaching the radar. As previously stated, each of the objects will scatter the intercepted radar signal, which will include a portion of backscatter energy. After the reflected radar signal returns to the radar in a certain time period, a second pulse of EM energy is transmitted. The reflection of the second pulse of energy from the building is returned to the antenna in the same time period as the first pulse. However, the reflection of the second pulse from the moving aircraft returns to the antenna in less time than the first pulse because the aircraft is moving towards the radar. This time change between pulses is determined by comparing the phase of the received signal with the phase of the reference oscillator of the radar [17]. If the phase of received consecutive pulses change, the object of interest is in motion.

2.3 Space-Time Adaptive Processing

The main goal of surveillance radar is to search a specified volume of space for potential targets. A typical radar system used for surveillance consists of multiple array elements mounted on an airborne platform. These radar systems have to detect targets in the presence of clutter and jamming. If the interference is localized in frequency and comes from a limited number of sources, targets can be detected by using adaptive spatial weighting of the data from each element of an antenna array [18]. By applying the computed weights to the data, the effects of interference can be reduced thus increasing the reception of the reflected signal. Because the platform is in motion the Doppler spread of the clutter returns is significantly wider, and the clutter characteristics are highly variable due to the constantly changing ground terrain. Because of the added dimensionality of received data, the weights must now be adapted from the data in both the time and space dimensions. This signal processing method is referred to as STAP, which is an adaptive processing technique that simultaneously combines the signals received from multiple elements of an antenna array (the spatial domain) and from multiple pulses (the temporal domain) of a coherent processing interval (CPI) [5].

The subsections to follow overview two STAP algorithms: the more complex “fully adaptive” STAP algorithm and the less complex “partially adaptive element space post-Doppler” STAP algorithm. The complexity associated with these algorithms is also overviewed. For a more theoretical foundation of STAP, the reader is referred to [5, 18].

2.3.1 Fully Adaptive STAP

Consider an N element antenna array mounted on an airborne platform that transmits a coherent burst of M pulses at a pulse repetition rate of $f_r = 1/T_r$ (where T_r is the pulse repetition interval - PRI). The time interval over which the waveform returns are collected is referred to as the coherent-processing interval (CPI). Thus, the length (in time) of one CPI is equal to MT_r. For each PRI, L samples are collected in the temporal dimension to cover the range interval of the returns. With N channels, M pulses, and L range bins, the received data for one CPI comprises LMN data samples. This set of data can be visually represented as a three-dimensional data cube of size $N \times M \times L$ as shown in Fig. 2.1. This data set is referred to as the CPI data-cube.

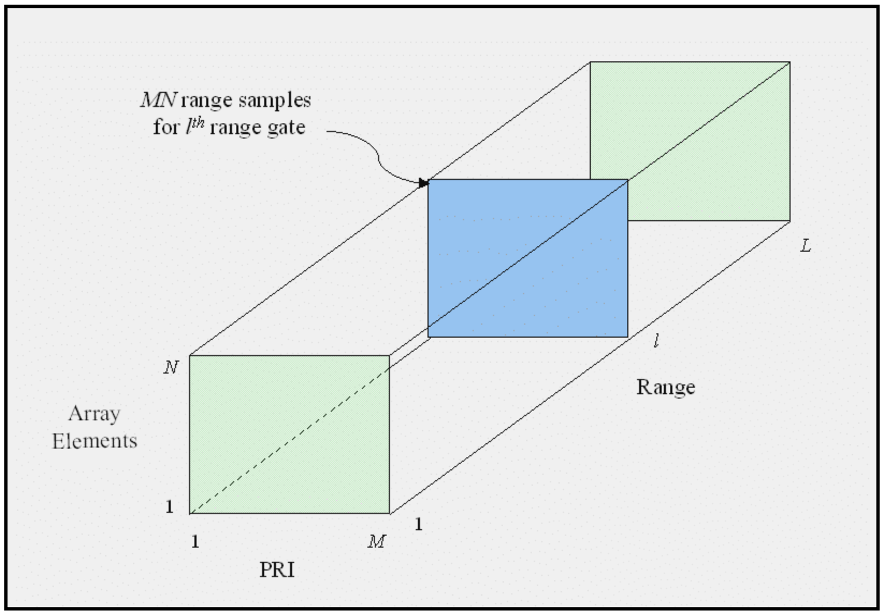

Figure 2.1 The STAP CPI data-cube.

Let x_{nml} represent the n^{th} array element and the m^{th} pulse at the l^{th} range sample time. Next, define $x_{m,l}$ to represent an $N \times 1$ column vector, or a spatial snapshot, composed of the return signals from each array element for the m^{th} pulse and the l^{th} range bin. By combining all of the spatial snapshots at a given range of interest, say l, an $N \times M$ matrix X_l can be formed, where $X_l = [x_{1,l}, x_{2,l}, x_{3,l}, \cdots, x_{M,l}]$. The shaded plane in Fig. 2.2, referred to as a range gate, represents the matrix X_l. To detect the presence of a target in given range gate, a linear adaptive filter is used that combines the MN data samples to produce a scalar output. This scalar is then compared with a threshold value to indicate the presence or absence of a target.

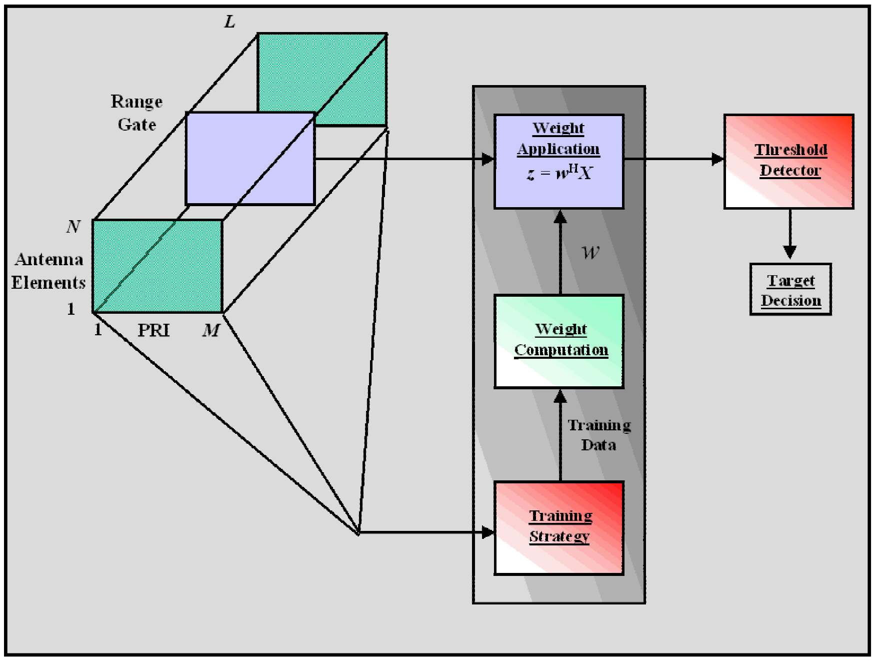

Figure 2.2 Generic computational phases for fully adaptive STAP.

Fully adaptive STAP generally consists of three major phases of computation. First a set of rules, called the training strategy, is applied to the CPI data. The objective of the training strategy is to estimate the interference that is present at the range gate of interest. Because the interference is not known a-priori, it is estimated, adaptively, from the data comprising the CPI data-cube. Typically, the data from several range gates near the range gate of interest are used in the training strategy.

The training data determined in the first phase is used as input to calculate the adaptive weight vector in the second phase. The weight computation phase is the most computation-intensive portion of the space-time processor. The weight computation involves the solution of a system of linear equations [5]. The most common weight computation strategy is called sample matrix inversion (SMI). In an SMI approach, the weight vector is computed based on the covariance matrix of training data. After the weight vector is computed, the final third phase of weight application commences.

In the weight application phase, a scalar output is produced by computing the weighted sum of the elements of the range gate of interest. This scalar output is compared to a threshold value to determine if a target is present at a specified angle and Doppler [5]. Because a potential target's angle and velocity are unknown, the space-time processor computes multiple weight vectors to cover many different target angles, ranges, and velocities at which target detection is to be queried [5].

In fully adaptive STAP, a separate adaptive weight is applied to every array element and pulse of a given range gate. The size of the weight vector for fully adaptive STAP is therefore *MN*. In order to compute the weight vector, a system of linear equations with dimension *MN* must be solved; thus, computing a single weight vector requires a $O((MN)^3)$ operations [5]. For many conventional radar systems, the product of *MN* may vary from several hundred to several thousand with *M* and *N* both ranging from ten to several hundred. Furthermore, a weight vector must be calculated for each training set used. The sheer computational complexity necessary to compute the weight vectors for fully adaptive STAP, in real-time, is typically beyond the capabilities of current embedded computing systems. This fact alone usually renders fully adaptive STAP impractical and provides adequate motivation for the formulation of alternative heuristic algorithms.

2.3.2 Partially Adaptive STAP

The goal of partially adaptive STAP algorithms is to break the fully adaptive problem into a number of reduced-dimension, more manageable adaptive problems while achieving near optimal performance. These reduced dimension adaptive problems are then solved to get the desired weight vectors. Instead of computing an MN-dimensional weight vector for each range gate, weight vectors of size KN are associated with each of the M returns. Typical values of K range from 1 to 3, whereas M may range from 32 to 64 [18]. Thus, solving M sets of linear equations of size KN has an overall complexity of $O(M(KN)^3)$ [5]. If $K << M$, this complexity is superior to that of fully adaptive STAP, which is $O((MN)^3)$. Thus, the computational complexity is reduced substantially by calculating M KN-dimensional weight vectors instead of one MN-dimensional weight vector. The partially adaptive algorithms are classified according to the type of preprocessing done first. For instance, in element-space pre-Doppler STAP, Doppler filtering follows adaptive-processing (see [5] for details on different classifications of partially adaptive STAP algorithms).

In the next chapter, a particular partially adaptive STAP algorithm is discussed and two approaches to solving the given problem of calculating the adaptive weights are proposed. Results of some numerical studies comparing the performance of the two approaches in terms of floating point operations needed, and the relative error are presented.

CHAPTER III

COMPARISION OF TWO ALGORITHMS FOR ADAPTIVE WEIGHT COMPUTATION

The focus of this chapter is the adaptive weight calculation of a partially adaptive STAP technique called the K^{th}-order Doppler-factored STAP algorithm [18]. Two candidate approaches to compute the adaptive weights for this algorithm are described. The first approach, which is characterized as a direct method, is based on performing a QR-decomposition [19, 20] on the data used to calculate the covariance matrix. In contrast to this approach, the second approach uses the CG technique, which is an iterative method, to compute the adaptive weights.

This chapter is organized as follows. In the next section, the K^{th}-order Doppler-factored STAP algorithm is described. Section 3.2 describes the QR-based technique for computation of the adaptive weights and Section 3.3 describes the conjugate-gradient approach. Comparisons between the two approaches, based on numerical studies using actual radar return data, are discussed in Section 3.4.

3.1 K^{th}-Order Doppler-Factored STAP

The K^{th}-order Doppler-factored STAP algorithm can be one of the most practical and effective STAP techniques for clutter and interference suppression. The architecture of K^{th}-order Doppler-factored STAP is composed of Doppler processing of data across all the pulse repetition intervals (PRIs) followed by adaptive filtering [18], i.e., the calculation and application of the adaptive weights. The adaptive filtering utilizes both the spatial and the temporal degrees of freedom. The spatial degree of freedom is provided by the N antenna array channels (Figure 2.1), while the temporal degree of freedom is provided by using K adjacent Doppler bins centered around the Doppler bin for which the weights are being calculated (see Figure 3.1). Here, Doppler bin refers to the dimension along the pulse (PRI) dimension of the data-cube after Doppler processing, and K indicates the order of the partially adaptive STAP algorithm.

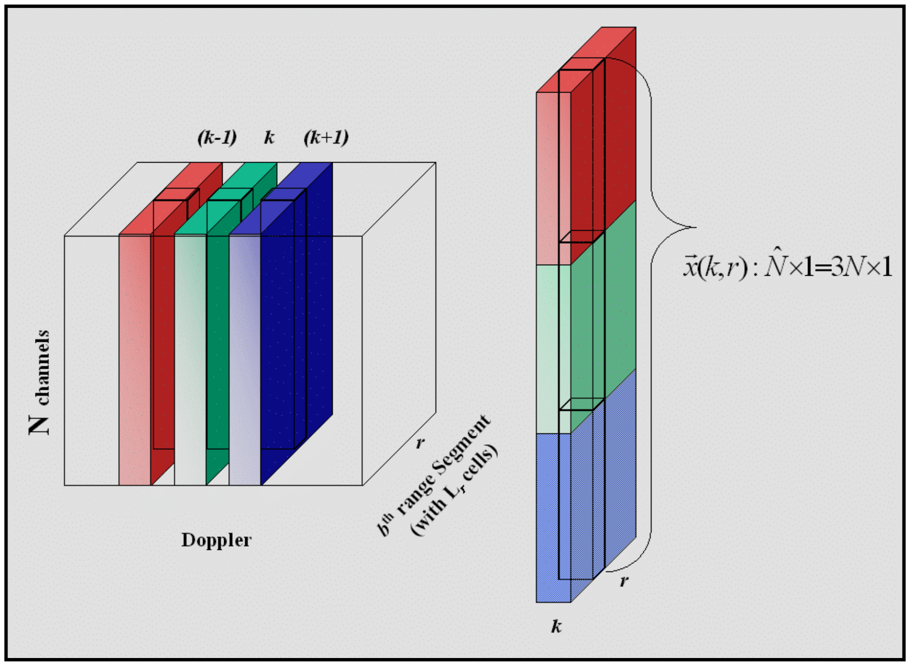
(k-1)
k
(k+1)
N channels
Doppler
r
b^th range Segment (with L_r cells)
$\vec{x}(k,r) : \hat{N}\times 1 = 3N\times 1$
r
k

Figure 3.1 Space-time snapshot of K^{th}-order Doppler-factored STAP for $K = 3$.

The adaptive weights for a particular range cell r and Doppler bin k are computed from the space-time snapshot vector consisting of data across the N channels and K adjacent Doppler bins $k_{\min}$ through $k_{\max}$. For K^{th}-order Doppler-factored STAP

$$k_{\min} = \text{mod}_M\left(k - \lfloor (K-1)/2 \rfloor\right), \text{ and} \tag{3.1}$$

$$k_{\max} = \mathrm{mod}_M \left(k - \lceil (K-1)/2 \rceil\right), \tag{3.2}$$

where M is the number of pulses. For example, if $M = 32$, $K = 3$, and $k = 13$, then

$k_{\min} = 12$ and $k_{\max} = 14$.

The space-time snapshot vector is then defined as

$$\vec{x}(k,r) = \left[x_d(1,k_{\min},r) \cdots x_d(N,k_{\min},r) \cdots x_d(1,k_{\max},r) \cdots x_d(N,k_{\max},r)\right]^T, \tag{3.3}$$

where $x_d(n,k,r)$ represents the data sample corresponding to rth range cell, kth Doppler bin and nth channel, and the superscript "T" denotes matrix transpose[1]. This space-time snapshot vector is shown in Figure 3.1 for K = 3. In practice, the range dimension is divided into non-overlapping segments, called range segments. Let L_r denote the number of range bins in each range segment, and let B denote the number of range segments; thus, $B = L/L_r$. The covariance matrix for the kth Doppler bin and bth range segment, denoted as $\psi(k,b)$, is estimated by averaging over the outer product of the snapshot vectors. That is

$$\psi(k,b) = \frac{1}{L_r} \sum_{r=(b-1)L_r+1}^{bL_r} \vec{x}(k,r)\,\vec{x}^H(k,r) \tag{3.4}$$

where $k = 1, 2, \ldots M$ and $b = 1, 2, \ldots B$. Note that $\psi(k,b)$ is a square matrix of dimension $KN \times KN$.

An alternate expression for $\psi(k,b)$ can be derived based on the $\hat{N} \times L_r$ space-time data matrix, denoted $X(k,b)$, which is defined to be

$$X(k,b) = \left[\vec{x}(k,\,(b-1)L_r+1) \cdots\cdots \vec{x}(k,bL_r)\right]. \tag{3.5}$$

[1] Related notation used in this thesis is the superscript "H" for Hermetian transpose of the matrix and "*" denotes complex conjugate of the matrix.

Based on this definition, the estimate of the covariance matrix of Eq. (3.4) can be expressed as

$$\psi(k,b) = \frac{1}{L_r} X(k,b)\, X^H(k,b)\,. \tag{3.6}$$

Let $\vec{w}(k,b)$ denote the vector of adaptive weights associated with the kth Doppler bin and the bth range segment. The value for $\vec{w}(k,b)$ is determined by solving the following equation

$$\psi(k,b)\, \vec{w}(k,b) = \vec{s}\,, \tag{3.7}$$

where $\vec{s}$ is a known steering vector. The values of the elements of the steering vector are dependent on the angle (relative to the radar platform) at which the target is to be detected and the speed of the target [5]. Thus, the above equation must be solved for each steering vector (i.e., each target position and speed) of interest.

Substituting Eq. (3.6) into (3.7) gives the following:

$$X(k,b)\, X^H(k,b)\, \vec{w}(k,b) = L_r \vec{s}\,. \tag{3.8}$$

The following two sections discuss weight computation strategies for the above system of linear equations. The two different approaches discussed here are the QR-decomposition method and the Conjugate Gradient method.

3.2 QR-decomposition Method

This method operates by first performing a QR-decomposition on the space-time data matrix $X^T(k,b)$. The QR-decomposition produces a $L_r \times L_r$ orthogonal matrix Q, and an $\hat{N} \times L_r$ upper triangular matrix R such that $X^T(k,b) = QR$. The matrix R can be written as $[R_1^T \;\; 0]$ where R_1 is a $\hat{N} \times \hat{N}$ (i.e., $KN \times KN$) full rank upper triangular matrix. The matrix product (Eq. 3.8) then decomposes to

$$X(k,b)\,X^H(k,b) = R^T\,Q^T\,Q^*\,R^* = R_1^T\,R_1^*, \tag{3.9}$$

where the last equality above is due to the orthogonality of Q; i.e., $Q^T Q^* = I$.

Following QR-decomposition, Eq. (3.8) is written as

$$R_1^T R_1^*\, w(k,b) = L_r\,\vec{s}\,, \tag{3.10}$$

where it should be noted that both R_1^T and R_1^* are triangular matrices. Letting $\vec{p} = R_1^*\,\vec{w}(k,b)$, Eq. (3.10) becomes

$$R_1^T\,\vec{p} = L_r\,\vec{s}\,,$$

which enables the determination of $\vec{p}$ using simple forward elimination. Once $\vec{p}$ is known, then $\vec{w}(k,b)$ is determined with backward substitution based on the definition of $\vec{p}$:

$$R_1^*\,\vec{w}(k,b) = \vec{p}\,. \tag{3.11}$$

3.3 Conjugate Gradient Method

The Conjugate Gradient method provides a general means for solving a system of linear equations of the form

$$A\,x = b,$$

where A is symmetric and positive definite [19, 20]. This approach can be applied to the problem of computing the adaptive weights because the covariance matrix is symmetric and positive definite.

The Conjugate Gradient method is based on the idea of minimizing the function

$$f(x) = \frac{1}{2} x^T A x - bx .$$

The function is minimized when its gradient

$$\nabla f = Ax - b ,$$

is zero, which corresponds to the solution of the original system of linear equations. The minimization is carried out by generating a succession of search directions p_i and improved minimizers x_i, where the subscript i denotes the iteration count. At each iteration a quantity α_i is found that minimizes $f(x_i + \alpha_i p_i)$. The value of x_{i+1} is then updated to $x_i + \alpha_i p_i$. The values of p_i and α_i are formed in such a way that x_{i+1} is also the minimizer of f over the whole vector space of directions already taken, i.e., $\{p_1, p_2, \ldots, p_i\}$.

For this study, the estimate of the covariance matrix as given in Eq. (3.6) takes the place of A and the known vector $\vec{s}$ is used in place of b. Thus, the conjugate gradient method is to be applied to the system of equations given by Eq. (3.7).

The conjugate gradient method consists of three distinct stages: initialization, iteration, and checking for convergence.

3.3.1 The Initialization Phase

The initialization phase consists of selecting an initial solution, and setting the initial direction [20]. The initial solution is typically selected to be the zero vector, however, a better initial guess may be used if known. Assuming a vector of zeros, the initial guess for the weight vector is denoted by

$$\vec{w}_0 = 0. \tag{3.13}$$

The initial direction, $\vec{d}_0$, is then defined to be

$$\vec{d}_0 = \vec{s} - \psi\vec{w}_0, \tag{3.14}$$

which is a conjugate of the initial gradient $\vec{g}_0$, i.e.,

$$\vec{g}_0 = -\vec{d}_0. \tag{3.15}$$

3.3.2 The Iteration Phase

During the iteration phase, successive direction vectors are generated that are the conjugates of the successive gradient vectors obtained as the method progresses. Thus, the directions are not known beforehand but are generated sequentially at each iteration [20]. At each step the current negative gradient vector is evaluated and a linear combination of the previous direction vectors is added to it to obtain a new direction vector along which to move. At each stage a quantity α_i is calculated which minimizes f $(w_i + \alpha_i\, d_i)$. The following operations are performed during each iteration :

$$\alpha_i = \frac{\vec{g}_i^T \vec{d}_i}{\vec{d}_i^T \psi \vec{d}_i} \tag{3.16}$$

$$\vec{w}_{i+1} = \vec{w}_i + \alpha_i \vec{d}_i \tag{3.17}$$

$$\vec{g}_{i+1} = \psi\, \vec{w}_{i+1} - \vec{s} \tag{3.18}$$

$$\vec{d}_{i+1} = -\vec{g}_{i+1} + \left[\frac{\vec{g}_{i+1}^T \psi \vec{d}_i}{\vec{d}_i^T \psi \vec{d}_i} \right] \vec{d}_i \quad . \tag{3.19}$$

The iteration phase continues making progress toward the solution at each step.

3.3.3 Checking for Convergence

After each iteration, a check for convergence is done to determine if the solution has reached a desired accuracy. Convergence is typically checked by evaluating the difference between the current and previous values of the solution vector. If the difference is small enough (i.e., is smaller than a specified tolerance), then the iteration phase is stopped.

3.4 Numerical Studies on MCARM STAP data

Numerical studies were conducted using Matlab implementations of the QR-decomposition and the Conjugate Gradient methods on actual STAP data collected by the Multi-Channel Airborne Radar Measurement (MCARM) system of Rome Lab [21]. This data consists of one CPI data-cube having 24 channels, 32 pulses, and 2500 range cells (i.e., the size of the CPI data-cube is $24 \times 32 \times 2500$).

Two cases were considered. In the first case the range cells were segmented into blocks of 125 (i.e., $L_r = 125$) while in the second case the range cells were divided into blocks of 250 (i.e., $L_r = 250$). The two different approaches – QR-decomposition and Conjugate Gradient methods – were then used to calculate the adaptive weights. Figure 3.2 shows the number of flops (floating-point operations) needed for each approach for $L_r = 125$, and Figure 3.3 shows the number of required flops for $L_r = 250$. Note that the number of flops needed decreases for the iterative method as the value of the convergence tolerance is increased, which implies decreased accuracy.

For L_r = 125 the flop count for the iterative approach is comparable to the flop count of the QR-decomposition approach when the convergence tolerance is relatively high. However, for L_r = 250 the flop count for the iterative approach is less than the flop count for the QR-decomposition approach, even for very high desired accuracy (i.e., small convergence tolerance). This is because the complexity of the QR approach depends on the value of L_r. However, the complexity of the Conjugate Gradient method is nearly independent of L_r, depending primarily on the dimension of $\psi(k,b)$, which is the same for the two cases considered.

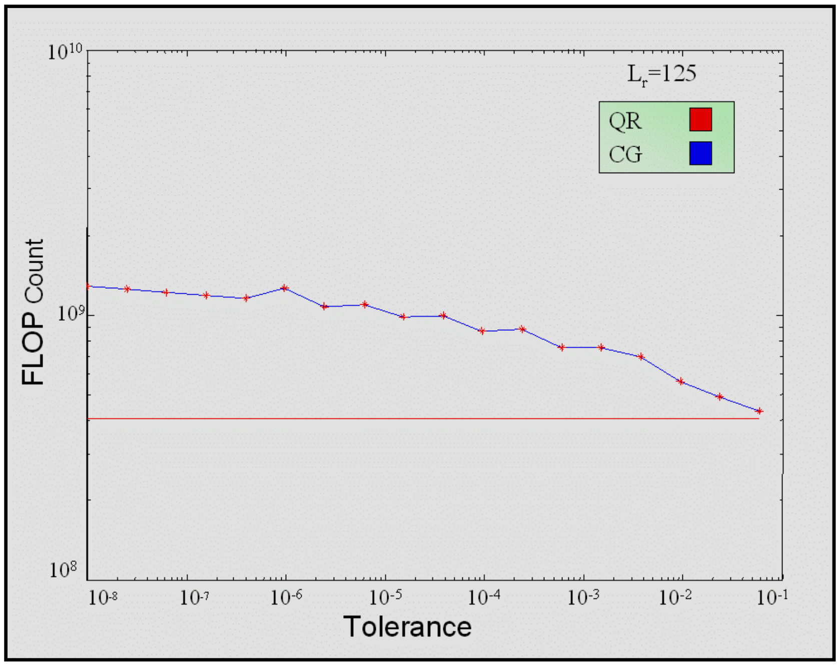

Figure 3.2. Flop count versus tolerance for L_r = 125.

The relative error between the weights obtained by the QR-decomposition and the weights obtained by the iterative approach (as a function of the tolerance for the iterative approach) is illustrated in the Figures 3.4 and 3.5 and is defined by

$$err = \frac{\left\| \vec{w}_{qr} - \vec{w}_{cg} \right\|}{\left\| \vec{w}_{qr} \right\|}, \tag{3.20}$$

where $\vec{w}_{qr}$ is the adaptive weight vector calculated using the QR-decomposition method and $\vec{w}_{cg}$ is the adaptive weight vector calculated using the Conjugate Gradient method.

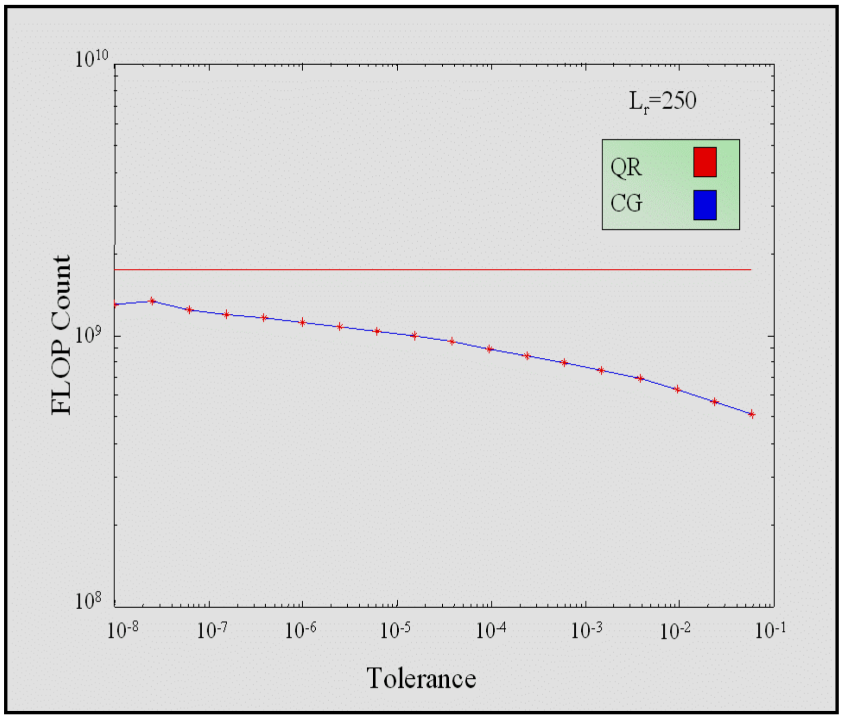

Figure 3.3. Flop count versus tolerance for $L_r = 250$.

As shown in Figure 3.4, the relative error for a convergence tolerance of 10^{-8} is approximately 10^{-8}, and for a convergence tolerance of 10^{-1} the relative error is approximately 10^{-1} (or 10%). The graph for L_r = 250 shown in Figure 3.5 illustrates similar characteristics. Thus, the Conjugate Gradient method provides for a trade-off between flops and accuracy. This trade-off may be important for STAP because reasonable (i.e., not perfect) accuracy may be sufficient in some circumstances, especially if the hardware required for determining the exact solution is prohibitive. Another motivation for using an iterative approach is the ease of implementation with FPGA technology.

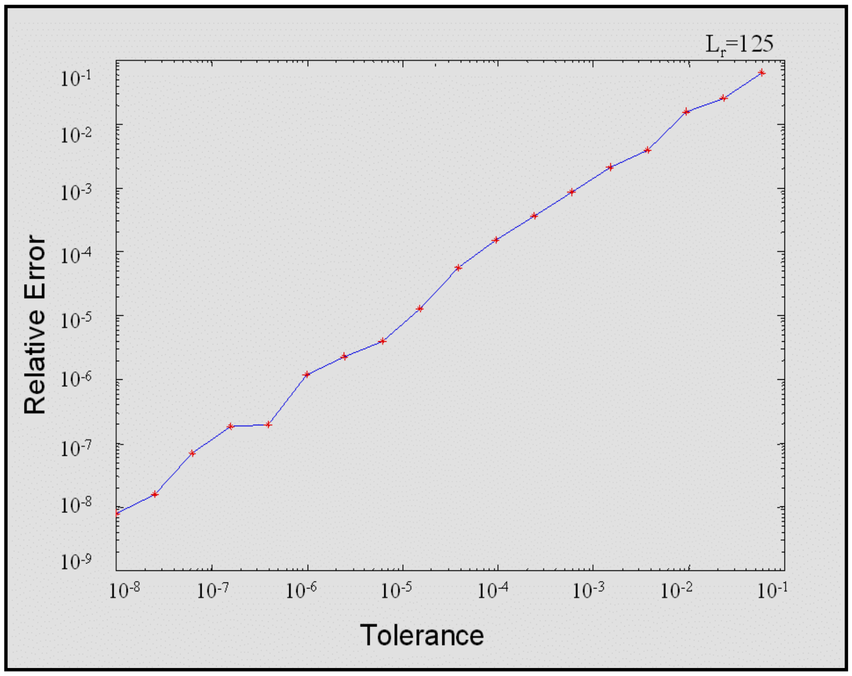

Figure 3.4. Relative error versus tolerance for $L_r = 125$.

3.5 Motivation for Research

A second major focus of this research is to demonstrate that at least some of the computationally intensive requirements of real-time STAP can be effectively implemented on a reconfigurable (FPGA-based) computing platform. Of particular interest is the computation required for determining the adaptive weights. It is proposed that a reconfigurable computing platform may be used as a “co-processor” to improve the performance of the host processors (e.g., a DSPs or GPPs). Performance maybe improved by off-loading some execution cycles to the reconfigurable coprocessor. The regular and repetitive nature of the iterative Conjugate Gradient approach described above makes it a prime candidate for implementation on an FPGA-based co-processor.

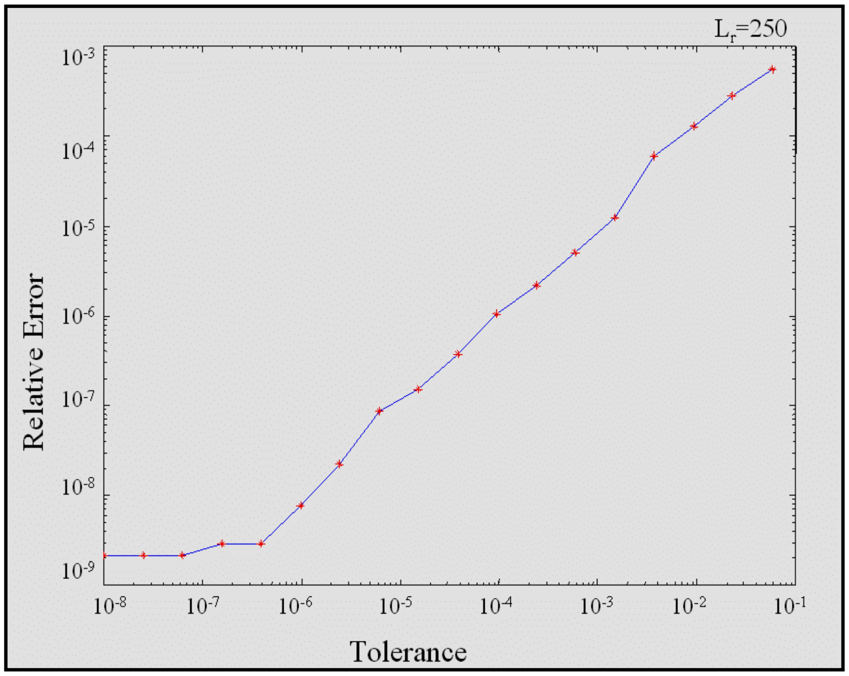

Figure 3.5. Relative error versus tolerance for $L_r = 250$.

3.6 Conclusion

In this chapter, the results of studies that compare the computational complexities of two competing approaches for solving the adaptive weights associated with STAP were presented. The trade-off between accuracy and the required computations associated with the Conjugate Gradient method was discussed. The results indicate that significant performance gains (in terms of required flops) can be realized at the cost of sacrificing some numerical accuracy. It is also seen that both the above methods for solving for adaptive weights involve computing a number of vector inner-products. A significant improvement in performance can be achieved if the inner-product computations can be off-loaded to the FPGA board while the GPP/DSP is free to do other tasks. While DSPs like SHARCS can do a single-cycle multiply they cannot perform an add operation very efficiently. The FPGA on the other hand can be programmed to perform a single-cycle multiply-and-add/accumulate, which makes it a very attractive for computing vector inner products.

CHAPTER IV

OVERVIEW OF THE WILDONE™ RECONFIGURABLE COMPUTING ENGINE

The use of reconfigurable computing platforms based on FPGAs is, in some instances, becoming more popular approach to implementing application-specific computing systems than designing special-purpose ASICs. A decade ago, when an embedded system required special-purpose computing hardware, it was usually designed into a special chip. However, manufacturing an ASIC is prohibitively expensive if the number of chips needed is very small, which is often the case with special-purpose embedded applications. A way to avoid the high cost of ASICs for special applications is to use relatively general-purpose chips like DSPs, along with application-specific software. Although DSPs are often well tuned for embedded applications, these devices are still relatively general in the sense that they are not designed specifically for any one application. Thus, the performance may not be as good as if ASICs were used in the implementation.

Reconfigurable computing systems that use FPGAs as the primary logic are providing a relatively new alternative to both DSP and ASIC based designs. Instead of application software controlling the processor(s), as is the case in DSP-based systems, *software* is actually used to configure the FPGA, thereby defining its functionality. The *software* used in this process is typically an encoding expressed using a hardware descriptor language such as VHDL. The use of FPGAs allows the matching of the hardware design with the processing needs of the application with the design directly implemented in hardware. Thus reconfigurable computing devices provide the flexibility of *soft* design methodology along with the performance advantage of hardware implementation.

4.1 WildOne™ Reconfigurable Computing Engine

In recent years, the manufacturers of reconfigurable devices have responded to the market demand of using reconfigurable hardware for some core components of high performance computing systems. This has led to improvements in reconfigurable hardware design to the point that FPGAs are now a viable implementation alternative. These developments have resulted in a number of reconfigurable computing systems, that can be used as a plug-in board, being manufactured by many companies with their own distinctive features. The WildOne reconfigurable computing board that is being used in this research is made by Annapolis Micro Systems.

The WildOne system offers a variety of configuration options. A high level diagram of the WildOne reconfigurable computing system is illustrated in Figure 4.1. The board has two processing elements, processing element 0 (PE0) and processing element 1 (PE1), which are Xilinx 4000 series FPGAs. (See the Appendix for more details of Xilinx 4000 series FPGAs.) These two processing elements (PEs) are connected to the on board bus through a dual port memory controller or through a FIFO. The FIFOs are each 36 bits wide and 512 words deep. The dual port memory control allows the access of the memory to the host as well as the PEs. There are also a number of fixed and reconfigurable internal data paths on the board to allow communication between the PEs, or between the PEs and external I/O cards that may include other FPGA boards, DSPs, ASICs and microprocessors. Section 4.2 describes the various data paths in detail.

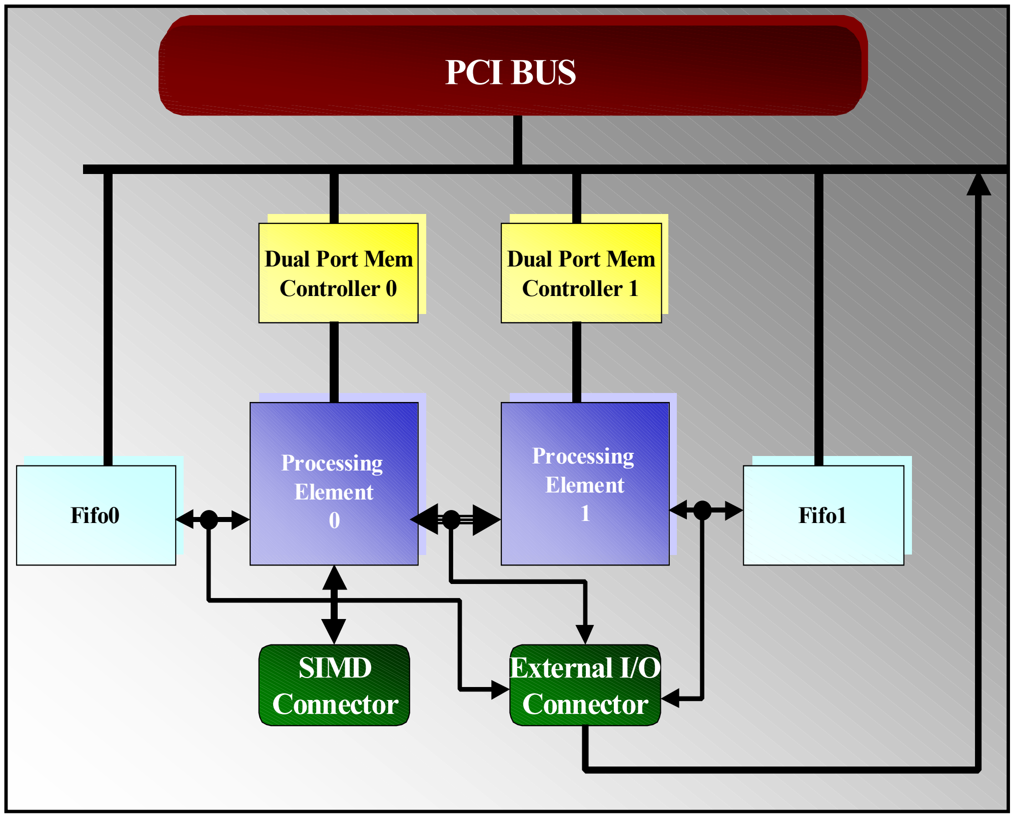

Figure 4.1 Block diagram of the WildOne™ reconfigurable computing board.

4.2 WildOne Data Paths

The host system may communicate with the board through the PCI bus. The communication between the host and the processing elements can be done using the FIFOs or the dual-port memory controller or through interrupt signals. In addition, the board can communicate with other boards via the SIMD connector. The SIMD connector is well-suited for high speed and real-time applications with the data coming directly into the board from an external transducer or from other boards on the same host system or boards on another host. There are also a number of fixed and reconfigurable internal data paths that allow the processing elements to communicate with each other. One of the most useful data paths between PE0 and PE1 is the *Direct Data* bus. This bus is very useful when the design is partitioned across the two processing elements.

Another useful bus similar in function to the *Direct Data* bus is *the PE1_Right systolic* bus. This bus, in addition to connecting PE0 and PE1, is also present on the external I/O connectors. This enables external inputs to be sent directly to both processing elements and allows the outputs to be a combination of signals from either processing element. Another set of internal signals is the handshaking and the auxiliary handshaking signals. These signals are called *CPE_PE1_Bus* and *CPE_PE1_AuxBus* in PE0 and *PE_CPE_Bus* and *PE_CPE_AuxBus* in PE1. The PE0 is also known as the controlling processing element (CPE) while the PE1 is also called the processing element (PE). The signals called *CPE_PE1_Bus* and *CPE_PE1_AuxBus* are signal names for the busses from the CPE or PE0 side while *PE_CPE_Bus* and *PE_CPE_AuxBus* are the signal names visible from PE or PE1 side. These signals are bi-directional and may be used in whatever mode the user desires. They may be used to for handshaking, starting a process, signaling process completion etc. For a more complete description of the WildOne board the reader is referred to [9].

The board is supported by an application programming interface (API) library that offers a set of C++ functions that execute low level run-time library functions not visible to the application programmer. Each board is accessible via a set of API routines, a corresponding set of data structures and constant definitions.

The next chapter discusses a general design methodology for designing an application for hardware/software co-system like the WildOne board. Two alternate architectures for computing the inner products are presented and the use of UML to model hardware/software co-systems is illustrated. The chapter also describes the architectural details of the two implementations.

CHAPTER V

INNER-PRODUCT CO-PROCESSOR DESIGN

By definition, methodology is an art that represents an orderly approach in performing a task or finding a solution to a problem. A good methodology helps in creating a design that is easy to understand and implement. In this chapter an application design methodology to design for a hardware/software co-system such as the WildOne board and its related host software is discussed. The general design methodology is illustrated first and then the application of this methodology to the problem at hand is discussed illustrating each step in the design process. The methodology uses UML to model the system and its components.

5.1 Application Design Methodology

When designing an application for the WildOne boards, it is helpful to divide the application design cycle into various stages as shown in Figure 5.1. Because the board is re-programmable, it is best not to attempt to design the entire application before testing, but rather to design in stages. This incremental design allows the designer to test the functionality of their design before progressing any further, thus decreasing the amount of time spent in testing the entire application.

5.1.1 Application Concept

The first step in any design process is to understand what the application is to accomplish. The designer should outline exactly what needs to be accomplished within the design. The designer should be familiar with the format of the input and output data sets, and the processes that need to be performed in order to process the data. The application should be thoroughly researched and all possible strategies reviewed. In the application at hand, inner product computations need to be performed on the FPGA board. Moreover, the computations need to be pipelined in order to perform single-cycle multiply-and-add operations to meet the throughput requirements needed for applications like STAP. Considering the limited resources available on an FPGA a block-floating-point [22] architecture is to be implemented with the mantissa width specified to be 16 bits with the most significant bit indicating the sign of the number and the block exponent being 8 bits wide. Block-floating-point arithmetic format is a fixed point arithmetic with one exponent for all data in a common block (i.e., vector). Block-floating-point numbers provide a compromise between the accuracy of fixed point numbers and the dynamic range of the floating-point numbers, without the full complexity or speed degradation associated with full floating-point operations.

Two different strategies are selected for performing this processing on the FPGA board. The first strategy uses the multiply-and-accumulate operations to reduce two N-vectors to a fixed number of partial sums (equal to the number of stages in the accumulate pipe). The host may then add up these few partial sums or may send these partial sums back to another FPGA for addition. In the second strategy, multiply-and-add operations reduce two *N*-vectors to *N/2* partial sums, which then need to be added on the host. The two strategies are shown in Figures 5.2 and 5.3.

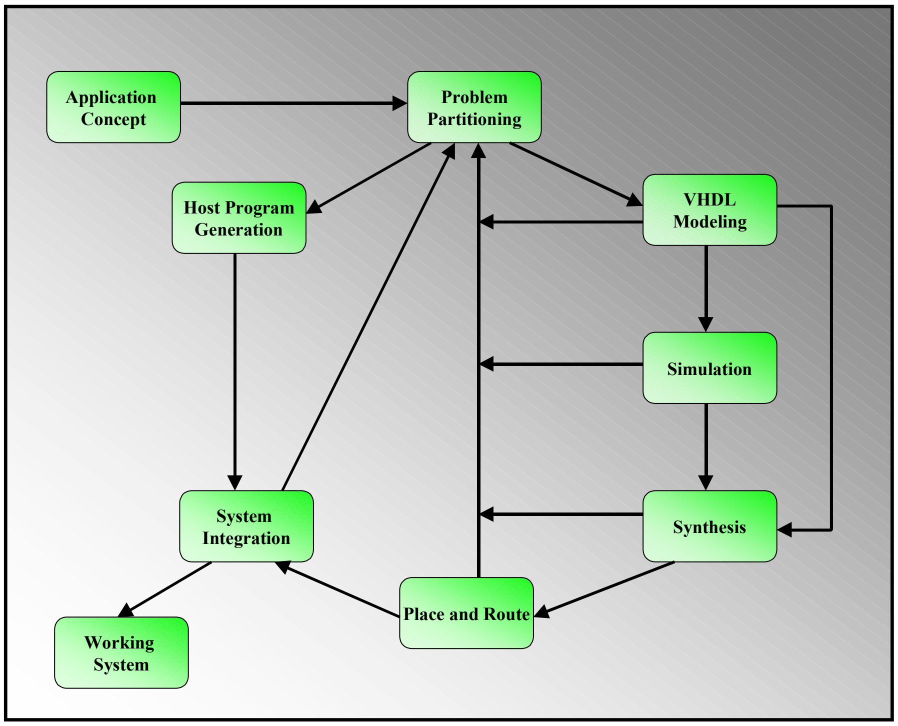

Figure 5.1 Illustration of application design cycle.

5.1.2 Problem Partitioning

After the strategy for performing the inner product has been chosen and verified, it is then partitioned into blocks that are independent of each other. For example, the two designs may be divided into separate blocks like the *Input Block*, *Data Processing Block*, and *Output Block*. Each of the individual blocks can then be subdivided into sub-blocks, e.g., the *Input Block* may have controllers for getting the data in and out of FIFO and the memory. The *Data Processing Block* may be divided into smaller subtasks like

Multiplying Unit and *Accumulator Unit*, which may subsequently be broken down into lower level entities (needed to create higher level designs). This kind of design is known as top-down design. The designer must also determine how the host system would handle the data that needs to be sent to the board as well as the processed data from the board. During this phase the use of UML is also a great help in partitioning and visualizing the different system components and modeling the system as a whole. Figures 5.4 through 5.8 show the UML class diagram representation of the *Data Processing Block*, which is called the *Inner Product Co-Processor*.

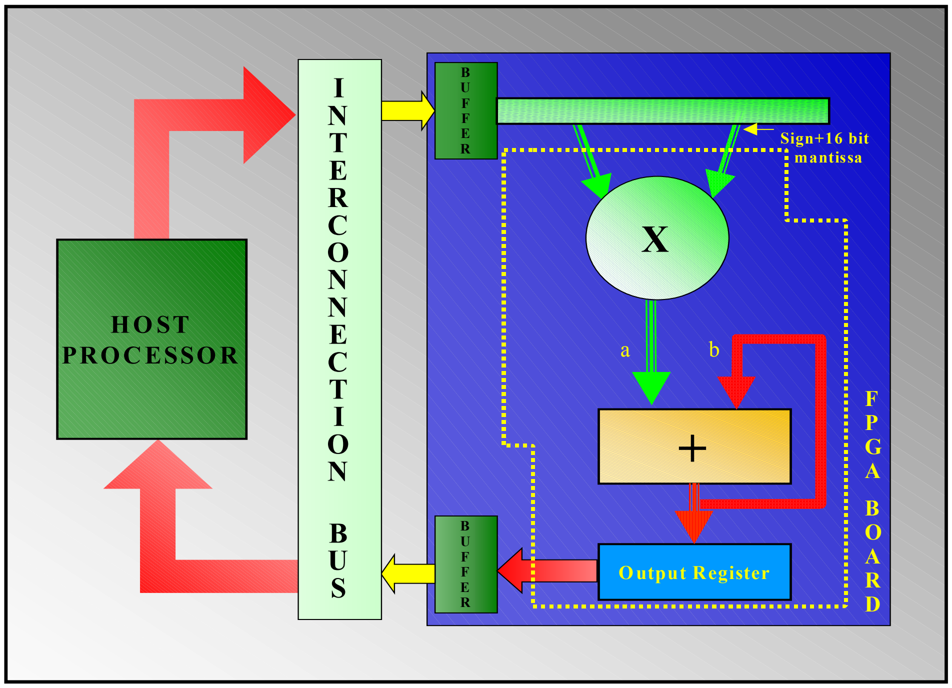

Figure 5.2 Multiply-and-accumulate strategy.

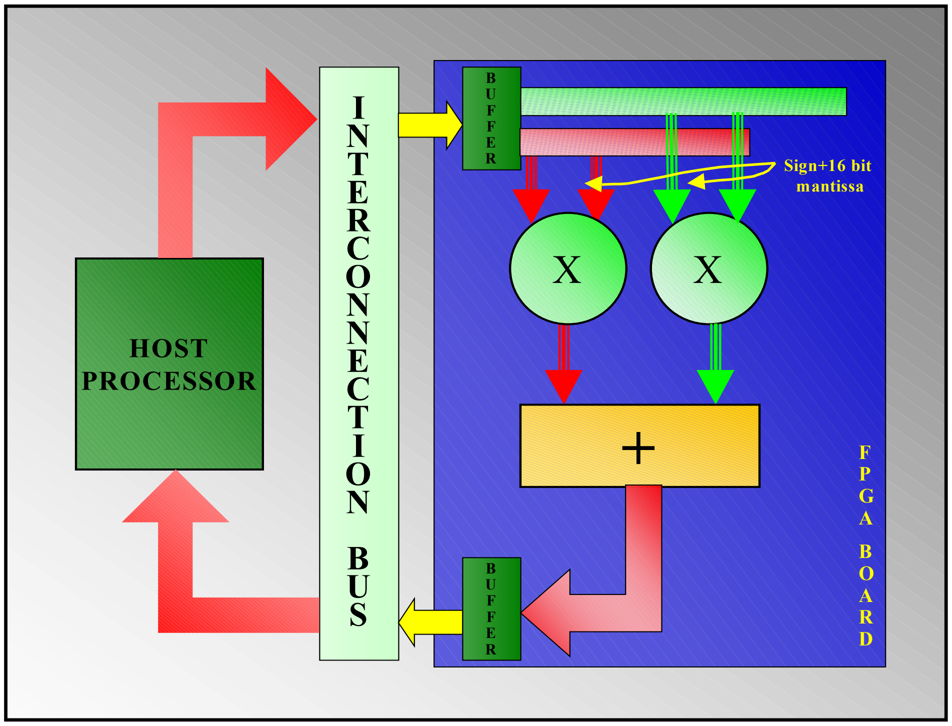

Figure 5.3 Multiply-and-add strategy.

5.1.2.1 UML Class Diagrams

The class diagram is one of the core components of a UML model [10]. A class diagram illustrates the important abstractions in the system including relationships. The primary elements included on a class diagram are class icons and relationship icons. Figure 5.4 shows the class diagram of the inner-product co-processor. The rectangular boxes represent the classes, while the lines connecting the classes signify relationships. A solid line with a hollow diamond at one end indicates an aggregation relation (i.e., one object is composed of another object). A solid line represents an association between the objects. The numbers shown at each end of the association denotes the number of potential objects participating in that relationship.

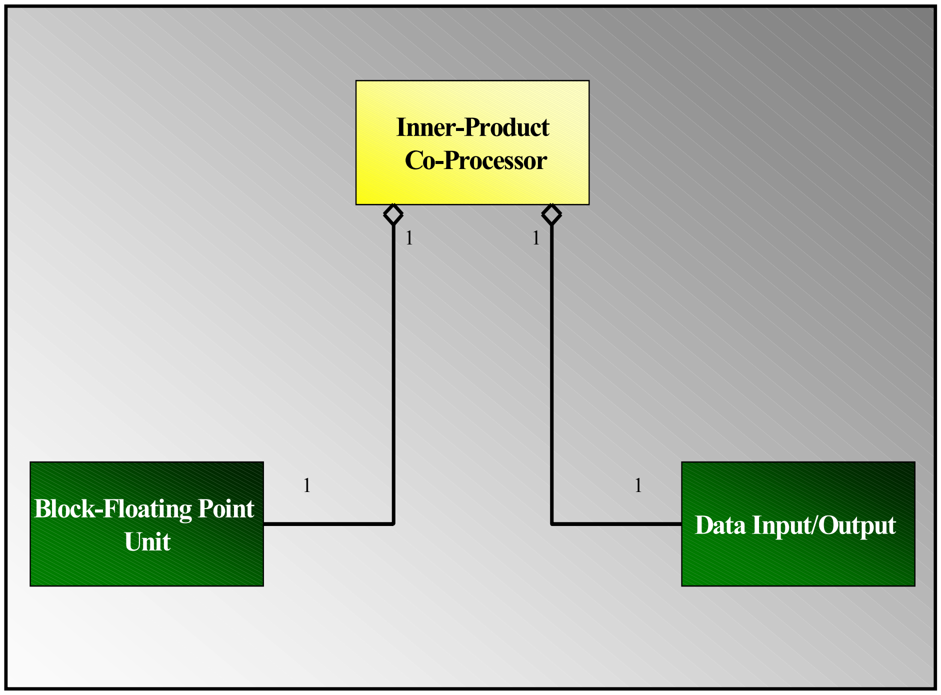

Figure 5.4 Inner-product co-processor UML class diagram.

The main class Inner-Product Co-Processor in Figure 5.4 is composed of two classes, the *Block-Floating-Point Unit* and the *Data Input/Output*. The *Block-Floating-Point Unit* gets its data from *the Data Input/Output* and also sends data back to the *Data Input/Output* when it has finished processing. Figure 5.5 illustrates the UML class diagram of the Block-Floating-Point Unit. This diagram implies that the floating-point unit is composed of one *Multiplying Unit* one and one *Accumulator*. The multiplying unit in turn is composed of registers, multiply stages and 4-bit adders while the accumulator is composed of registers, a 3-bit adder, 4-bit adders, a normalizing unit and a complementing unit. The class diagrams for the multiplying unit and the accumulator are shown in Figure 5.6 and 5.7 respectively. The normalizing unit, shown in Figure 5.8, is composed of a subtractor, a magnitude comparator and a number of registers, which are the lowest level of components in the design. Figure 5.9 shows the class diagram for the Data Input component. This component consists of a memory controller and FIFO controller.

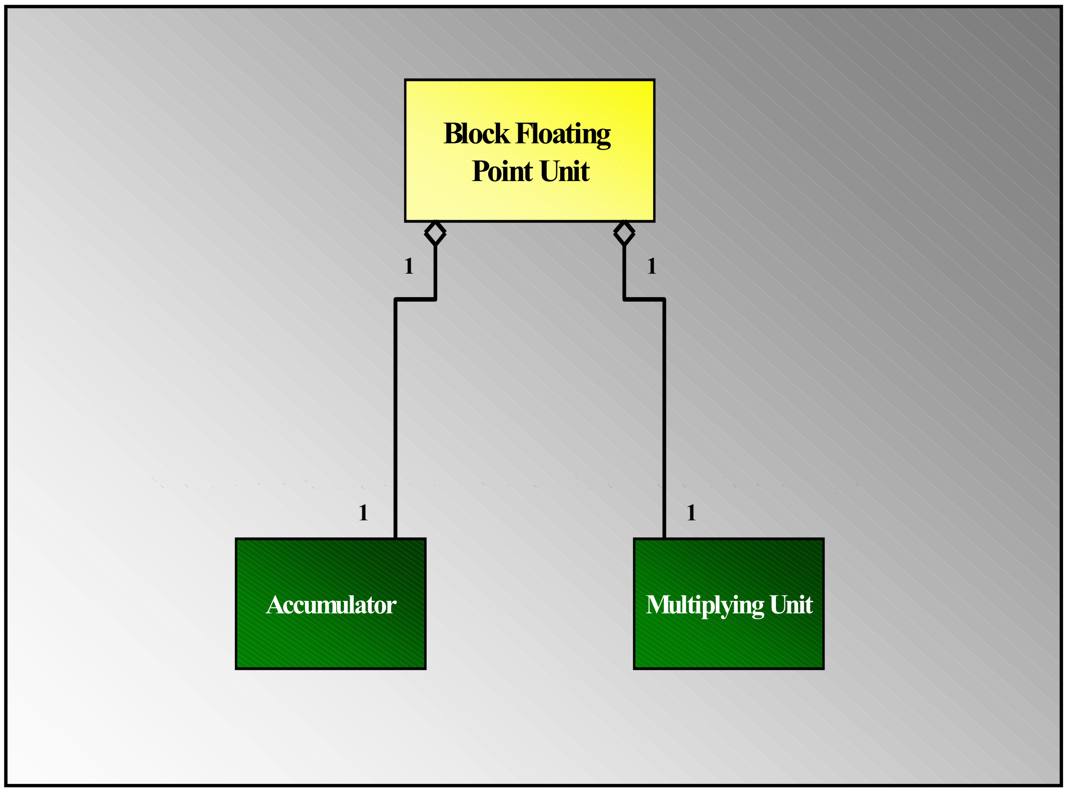

Figure 5.5 Block Floating-Point Unit UML class diagram.

After the different components have been defined using the UML class diagram, the next step is to define the functionality of each component of the system and how it interacts with the host system or with other components. This can be done by using the UML statecharts and activity models [10].

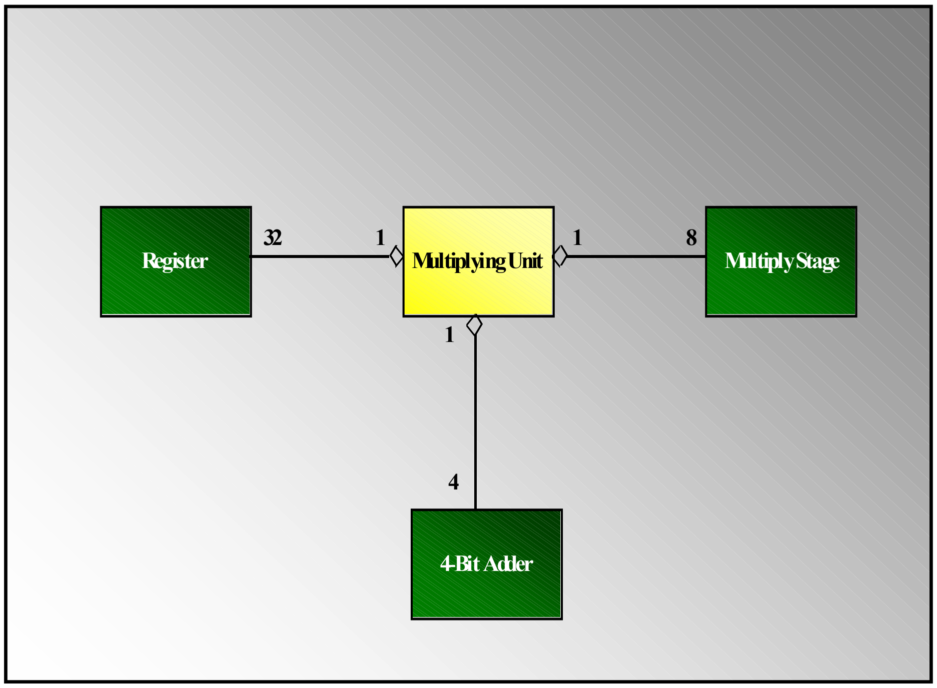

Figure 5.6 Multiplying unit UML class diagram.

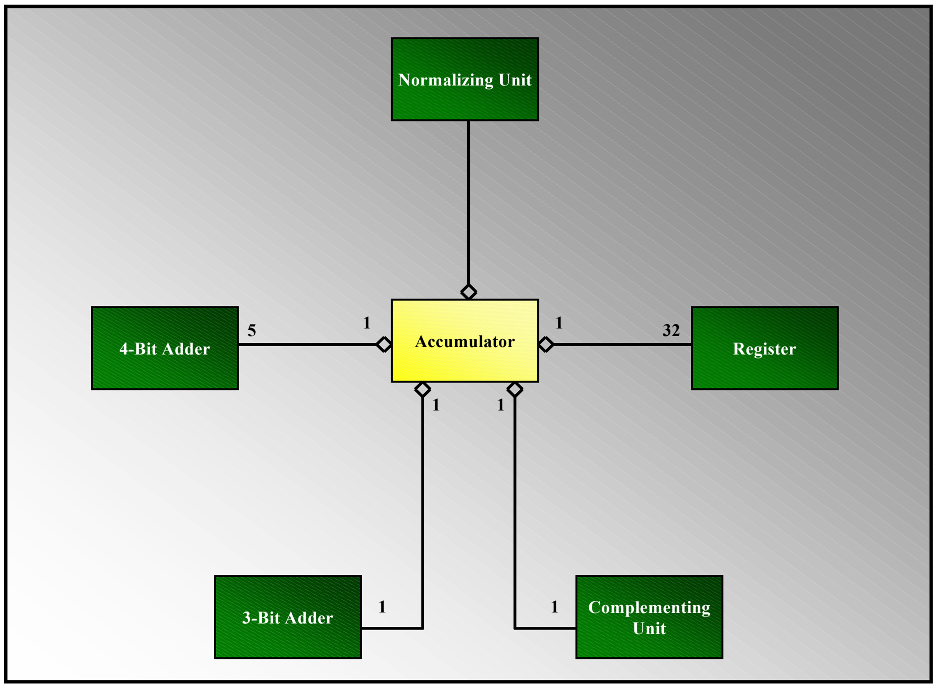

Figure 5.7 Accumulator UML class diagram.

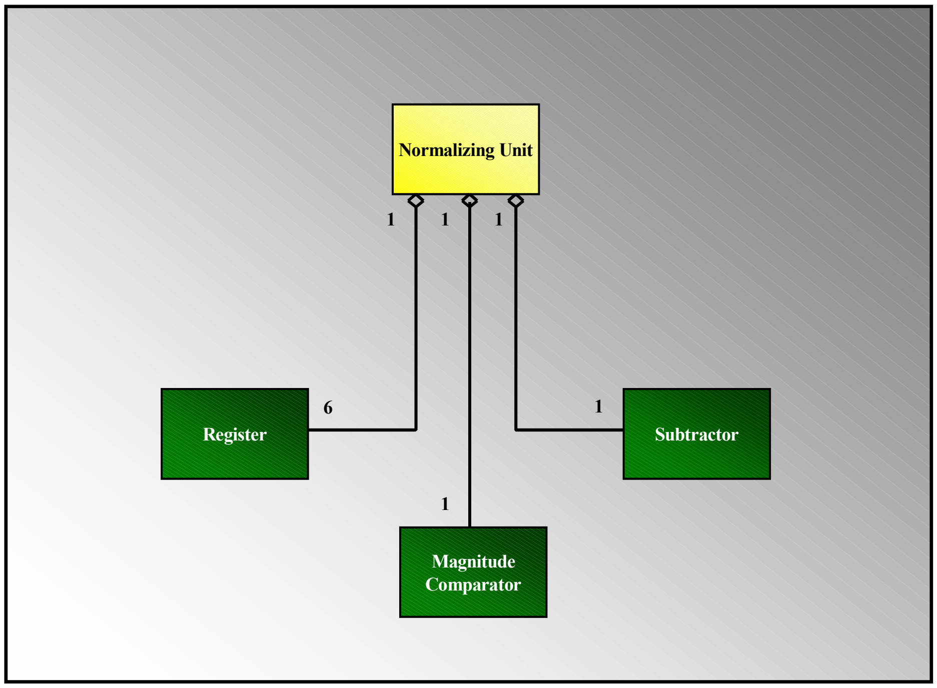

Figure 5.8 Normalizing unit UML class diagram.

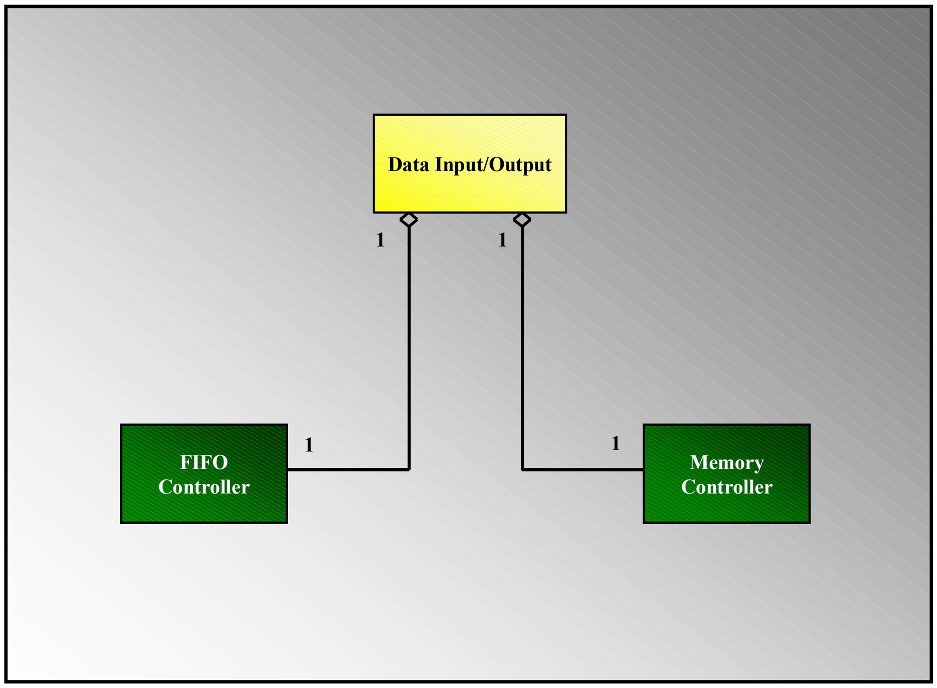

Figure 5.9 Data input/output system UML class diagram.

5.1.2.2 Inner-Product Co-Processor UML Statechart Diagram

The UML statechart models are based on the finite state machines using an extended Harel state chart notation [10]. A statechart diagram represents a state machine and illustrates the sequence of states that an object goes through during its life cycle. The states are represented by a rectangle with rounded corners and arrows connecting the states represent transitions. The initial state is shown as a small filled dot representing any transition to the enclosing state [10]. A final state is shown as a small filled dot enclosed by a circle. In a state chart diagram the occurrence of an event may trigger a state transition.

A UML activity model is a variation of a state machine in which the states are activities representing the performance of operations and transitions are triggered by completion of an operation. Activity diagrams focus on the flows driven by internal processing. While activity charts are used to model synchronous events a statechart diagram should model any asynchronous events.

Figure 5.10 shows the statechart diagram for the Inner-Product Co-Processor. The statechart indicates the events and transitions that occur to get the data from the host to the FPGA board and the subsequent processing. The statechart represents two distinct state machines, one on the host system and the other on the FPGA board. The two state machines are running concurrently and the state transitions in one of the state machines may be triggered as a result of an event occurring in the other state machine. Dashed lines between the two state machines shown in Figure 5.10 indicate these events.

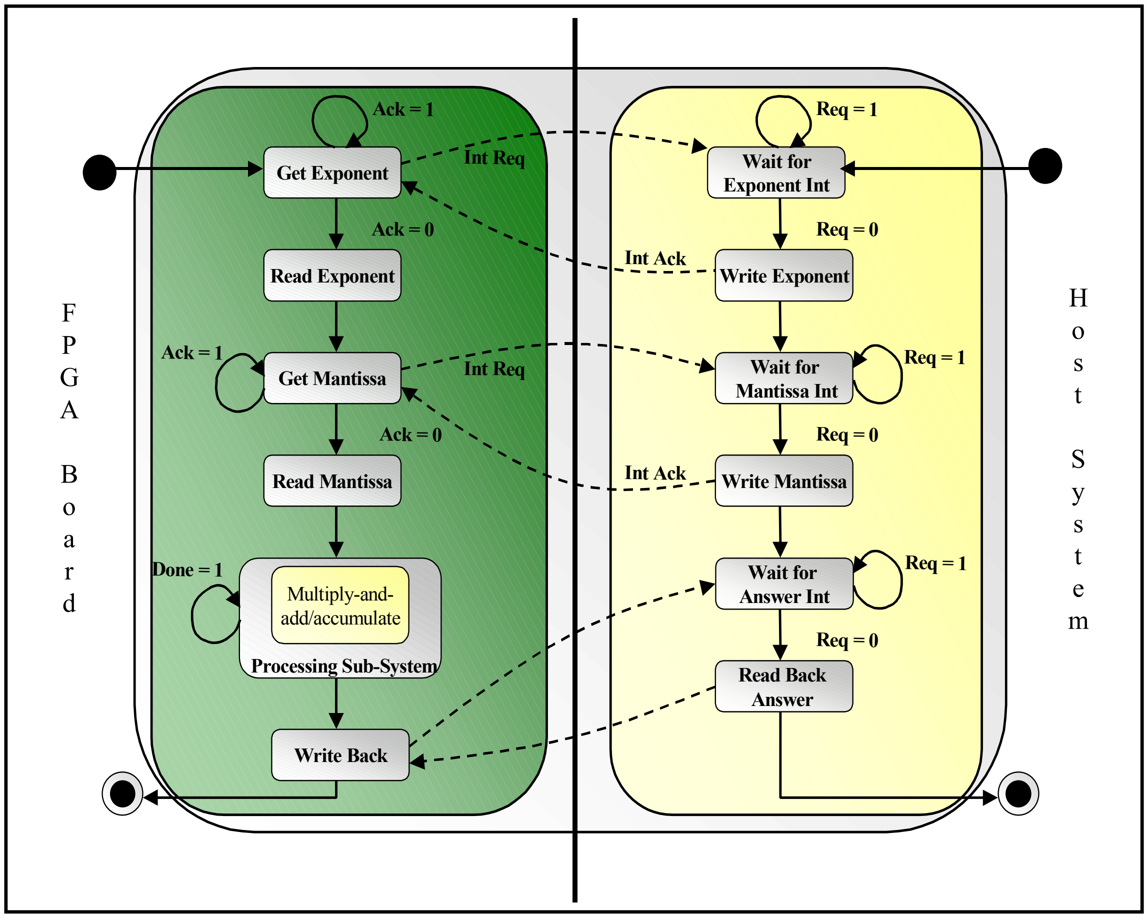

Figure 5.10 Inner-product co-processor statechart diagram.

A complex interaction takes place between the FPGA board state machine and the host system state machine to get the data to the FPGA board where it is processed and then written back to the host system for further processing. The FPGA state machine interrupts the host system and requests for the exponent part for performing block-floating point computations. The host state machine at the same time is waiting for the interrupt request. When it gets the request, it writes the exponent to the FPGA FIFO and sends back an interrupt acknowledge signal back to the FPGA board. The FPGA state machine then reads in this data from its on-board FIFO and then sends in an interrupt request for the mantissa portion of the data. The host then writes the mantissa parts of the vectors (whose inner-product is to be found) to the FPGA FIFO and then returns an interrupt acknowledge signal back to the FPGA board. The FPGA system then reads in the mantissa data and multiplies the corresponding data in the vectors and either writes these back to the FIFO memory or keeps on accumulating the partial sums depending on the algorithm being used. This processing continues until all the data has been read and a done flag is set. The FPGA state machine then sends an interrupt request to the host system to indicate that it has finished processing the data and the host system can read back the processed data. The host then reads back the data from the FPGA memory or the FPGA FIFO.

The co-processor is a synchronous system with all the operations being performed synchronously and is best represented by using the activity diagrams instead of the statechart diagrams. An activity diagram is a special case of a state diagram in which the states are action states where all (or at least most) of the transitions are triggered by completion of the action in the source state [10]. The processing is done differently depending on whether the multiply-and-add algorithm is being used or the multiply-and-accumulate algorithm is being used. The activity diagrams for both algorithms are shown in Figures 5.11 and 5.12.

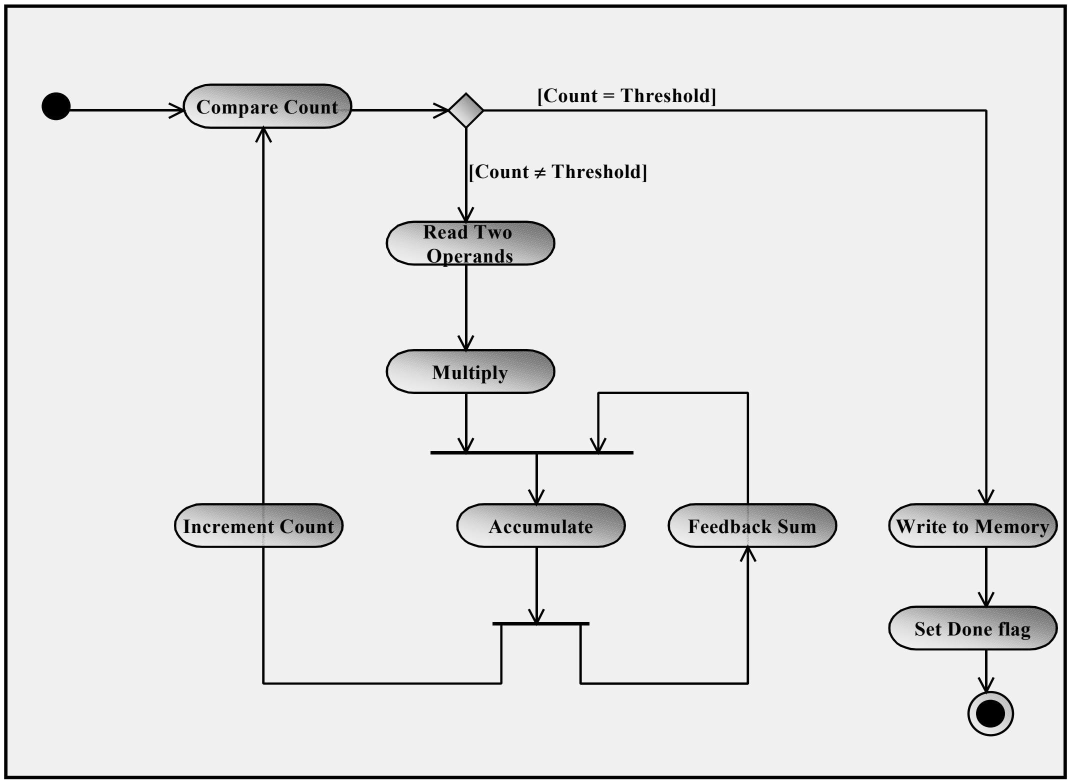

Figure 5.11 Multiply-and-accumulate circuit activity diagram.

5.1.3 VHDL Modeling

After the design has been partitioned into blocks and their functionality determined the implementation of the design begins. Because the same VHDL is used for both simulation and synthesis, it is important that the code be written for synthesis, i.e., the designs should be synchronous and the designer should avoid using constructs that are not synthesizable. As it is easy to test a design in stages it is important to design an application in stages. Each stage needs to be tested thoroughly and independently of the other stages of the design. The design implementation usually begins by developing the lower level entities first. The design is then built up on these components in a bottom-up fashion.

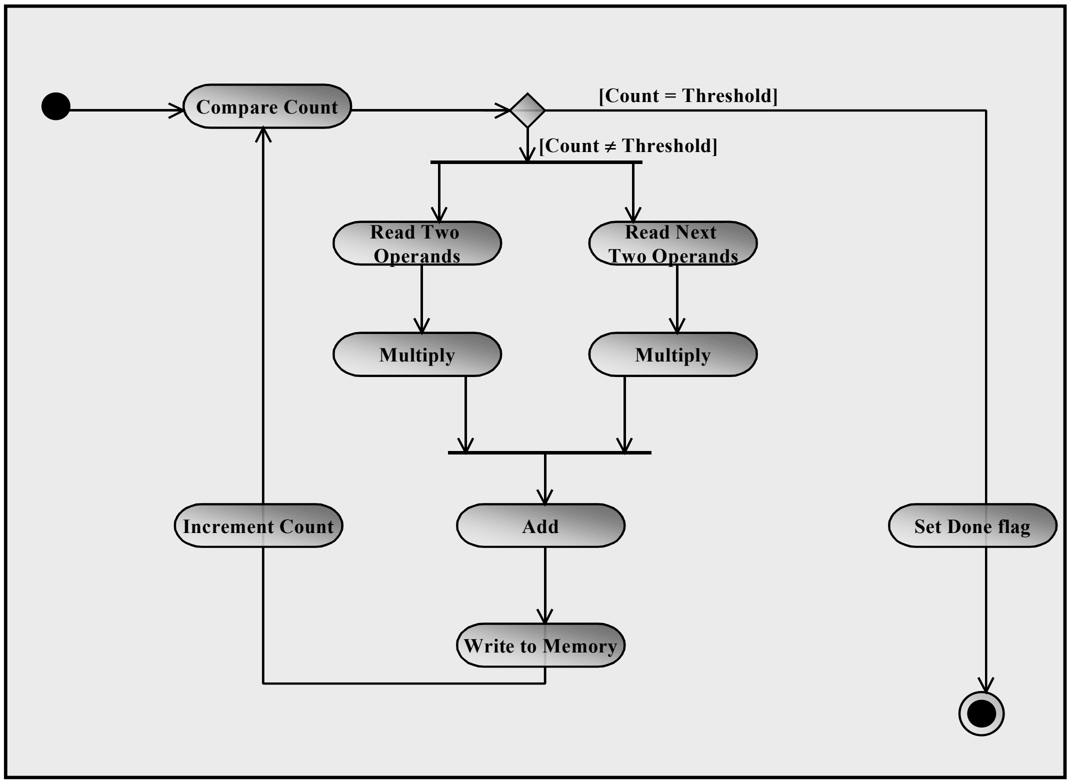

Figure 5.12 Multiply-and-add circuit activity diagram.

5.1.4 Simulation

After each component of the design is implemented in VHDL, it is very important that the code be simulated using a VHDL simulator to verify the component's functionality. This saves a lot of time when trying to test the design on the board. Although simulation is reasonably accurate it generally cannot give an exact representation of how the system will actually respond, since it does not contain any information about timing internal to the design. If any major problem in functionality is seen in the simulation a redesign becomes necessary.

5.1.5 Synthesis

The synthesis tools define the gate level logic for the target architecture from the VHDL implementation. It is useful to synthesize often, even if the design is not yet ready for the place and route step, because the synthesis tool provides preliminary estimates of the resource utilization and speed, which can be used to determine if the current design approach is viable or not. If the design is not going to meet the timing requirements or is too large to fit in the available real estate, then redesign becomes necessary. Many iterations of the above process are needed for a fair sized project.

5.1.6 Place and Route

After the design has been verified, the next stage is the place and route where the gate level logic generated during the synthesis stage is used to configure the FPGA. The output of this process is a binary file, which is sent to the WildOne board to program the processing elements.

5.1.7 Host Program Generation

Once the processing element images have been generated, the next step is to write a C++ program that allows the host system to communicate with the WildOne board. This program makes use of the WildFire API functions to communicate with the board. While writing the host program the synchronization points between the board and the host program must be defined. The interaction between the host program and board is as shown in the sequence diagram shown in Figure 5.13.

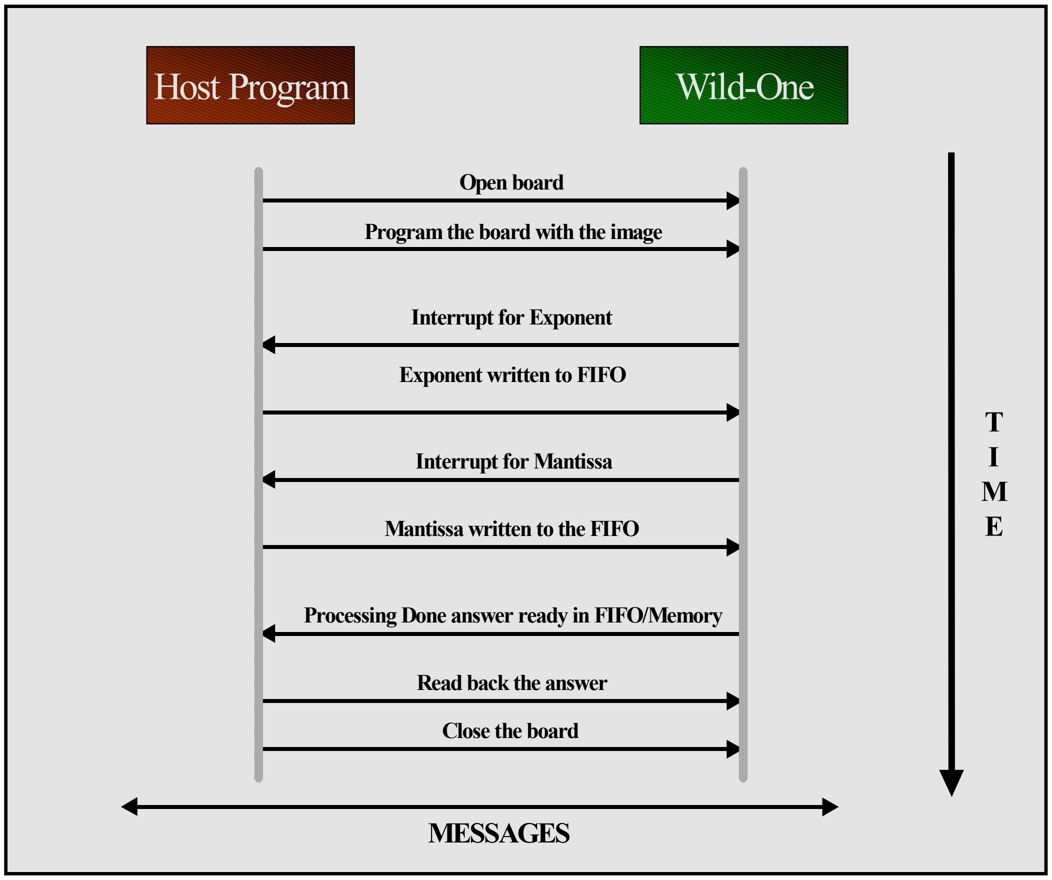

Figure 5.13 Sequence diagram for interaction between WildOne board and the host.

5.1.8 System Integration

The application is now ready to execute on the WildOne system. In this stage the simulation results are verified with the actual system results. Once the host code has been verified sections are added piece by piece to the design incrementally and the results verified with the simulation results. The design process proceeds through this design cycle until a complete working system is achieved.

5.2 Architectural Details of the Implementations

The previous sections in this chapter discussed the different entities and their relationships. This section discusses the architectural details of the two implementations. The multipliers in both the units are 15-bit multipliers, i.e., the multiplier unit takes in two 15-bit numbers to produce a 30-bit result. Because the block-floating-point format is being used, all the numbers to the multipliers have the same exponent. Thus, the exponent of the result is two times the input exponent. The implementation of the pipelined multipliers is based on the implementation discussed in [23] and has 13 pipeline stages.

The accumulation unit in the multiply-and-accumulate circuit consists of a normalizing unit and a 23-bit pipelined adder. The mantissa in IEEE floating point numbers is 23-bit wide. To make the output compatible with the IEEE representation the adder and the accumulator units are 23 bits wide with another 8 bits for the exponent and one bit for the sign. The normalizing unit is necessary because the answer is being accumulated and it is possible that the exponent of the accumulated sum changes. If the exponent of the accumulated sum becomes greater than the incoming operand, then the incoming operand needs to be shifted a certain number of bits (equal to the difference in the exponents of the two numbers) to the right. If the exponent of the incoming number is greater than the exponent of the accumulated sum then the accumulated sum is shifted to the right. The 23-bit pipelined adder is implemented using five stages of 4-bit fast adders [24] and one stage of 3-bit fast adder. It should also be noted here that the output of the multiplying unit is 30-bit wide, however, the width of the adders is only 24-bits.

Therefore the result of the multiplier is truncated to 23-bits before entering the adder circuit.

The addition unit in the multiply-an-add circuit has exactly the same architecture as the adder unit in the accumulation circuit. However, the normalizing unit is not needed in this implementation because the exponents of all the operands coming in are the same. The output of both implementations is in the IEEE floating-point format.

5.3 Features of the Two Implementations

The two implementations perform part of the computations for computing the vector inner products in hardware. The multiply-and-accumulate circuit takes in two *N*-vectors as input and gives out 17 (the number of pipeline stages in the accumulation circuit) partial sums, which then need to be added on the host to get the final inner-product. The accumulator pipe needs to be flushed [24] when all the operands have been added. This circuit takes in two operands and performs two operations per clock cycle (multiply and an accumulate) after which the sum is fed back to the accumulator to be added to the new product coming into the accumulator. The multiply-and-add circuit takes in two *N*-vectors and gives out *N/2* partial sums, which then need to be added on the host. The multiply-and-add circuit takes in four operands and performs three block-floating-point operations (two multiplies in and an addition) per clock cycle. Consider the operation of each circuit at 40 MHz, which implies a throughput of 80 block-floating-point-operations per second for the multiply-and-accumulate circuit and 120 block-floating-point operations for the multiply-and-add circuit. The multiply-and-accumulate circuit however, needs less work to be done on the host side than the multiply-and-add circuit. Thus, although this circuit has a slower peak throughput, it can perform more of the computations than the multiply-and-add design.

Both the circuits contain components that are common to both circuits. The multipliers used in both the circuits are based on an implementation proposed in [23]. The adders used are carry-look-ahead adders discussed in [24]. An important aspect of the multiply-and-add circuit, however, was how to get four 16-bit operands into the circuit because only 32 input pins were available. This problem was solved by clocking the state machine, which inputs the data to the processing circuit, at twice the frequency of the clock used to clock the actual processing circuit. This is illustrated in Figure 5.14. So at each falling edge of the data processing clock, all the four operands are available.

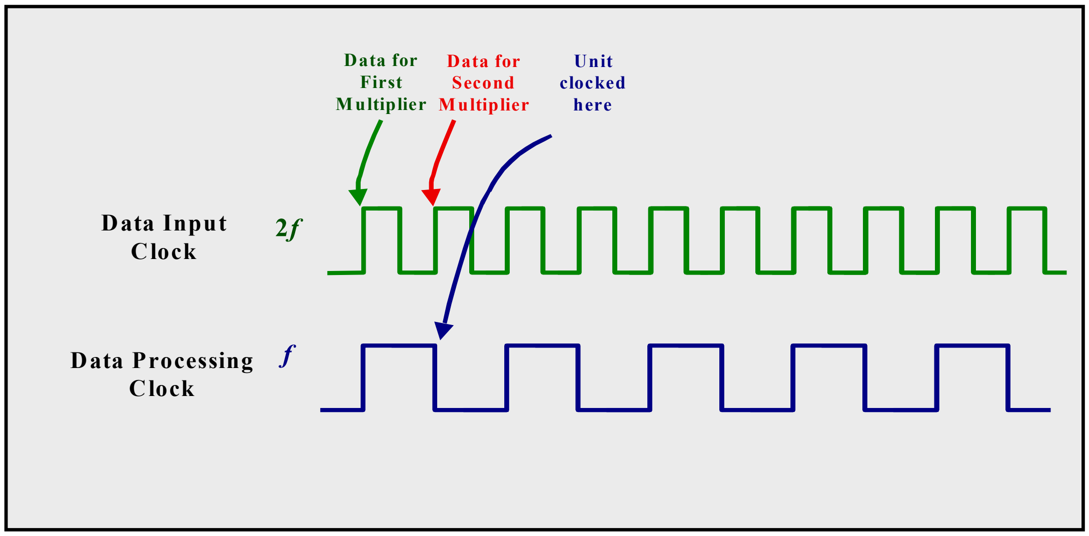

Figure 5.14 Multiply-and-add circuit clock waveforms.

This chapter overviewed the design methodology for the design with FPGA board. The various steps of this methodology were discussed and the application of the same to the designs at hand helped visualize its functionality. In the next chapter the results of numerical studies carried out to analyze the two circuit implementations are presented.

CHAPTER VI

ANALYSIS OF THE IMPLEMENTATIONS

An important aspect of this research is the determination of the accuracy of the two FPGA inner product implementations described in the previous chapter. Some accuracy is sacrificed as a result of a block-floating-point implementation as compared with a full floating-point implementation of the same computation. However, the block-floating-point implementation was necessary because of the limited resources available on the FPGA. In this chapter, the accuracy of the two FPGA implementations of the inner-product computation, which use a block-floating-point format, are compared with the inner-product computation performed on the host machine, where full floating-point arithmetic is utilized.

6.1 Accuracy Experiments

The implemented circuits were tested for accuracy by sending a set of test vectors to each of the two FPGA implementations. The output of the two circuits were then compared with the results for the same data set obtained from the host machine, which performed the computation using full floating-point arithmetic. The accuracy is measured by dividing the answer returned by the FPGA board by the corresponding answer computed by the host machine. Because the elements of the test vector were positive values, the output of the inner product circuits were always positive. Furthermore, because the widths of the block-floating-point mantissas were 15 bits compared to 23 bits for the full floating-point numbers, the block-floating-point numbers are always less than or equal to the full floating-point numbers. This implies that the ratio of final answers – those computed using block-floating-point representation and those computed using full floating-point arithmetic are always less than or equal to one.

The following cases present the accuracy and the dynamic range of the FPGA implementations. The data sets consist of two vectors, each having 512 elements. All the numbers are generated using a pseudo-random number generator with a uniform distribution over a specified range of values.

6.1.1 Analysis of Uniform Data Value Distribution for Zero Order of Magnitude

Figure 6.1 shows the histogram of the distribution of the data values for the range of 0 to 1. The figure shows a relatively uniform distribution over the entire range. Because the block-floating-point format is being used, it is interesting and insightful to look at the exponent space of the data values as well. This is because the exponent values dictate how many bits are shifted out of any number and thus the accuracy of the final answer. The number of bits shifted out of a given vector element is the difference between the maximum exponent value for all vector elements and the given element. Thus, if a given element has an exponent of 126 and the maximum exponent is 128, then two least significant bits would be shifted out of the mantissa of the given element.

Figure 6.2 shows the histogram of the exponents for the numbers associated with the distribution of Figure 6.1. As shown in Figure 6.2, the maximum exponent is 128 and the minimum is 118. (All the exponents are in the excess 127 format [25].) This implies that the maximum number of bits that are shifted out of any element is 10 (128 – 118). However, most of the exponents are closer to the maximum exponent value. This results in a very good accuracy for most elements. It should also be noted here that the FPGA implementations use a 15-bit mantissa; therefore, because the numbers are originally in the IEEE floating-point format, which has a 23-bit mantissa, the lower 8 bits of the mantissas of all the numbers are always truncated. This effect by itself introduces some inaccuracy in the answer.

Figures 6.3 and 6.4 show the accuracy histograms achieved by the two FPGA implementations relative to the answers provided by the host machine, which utilizes the IEEE floating-point arithmetic.

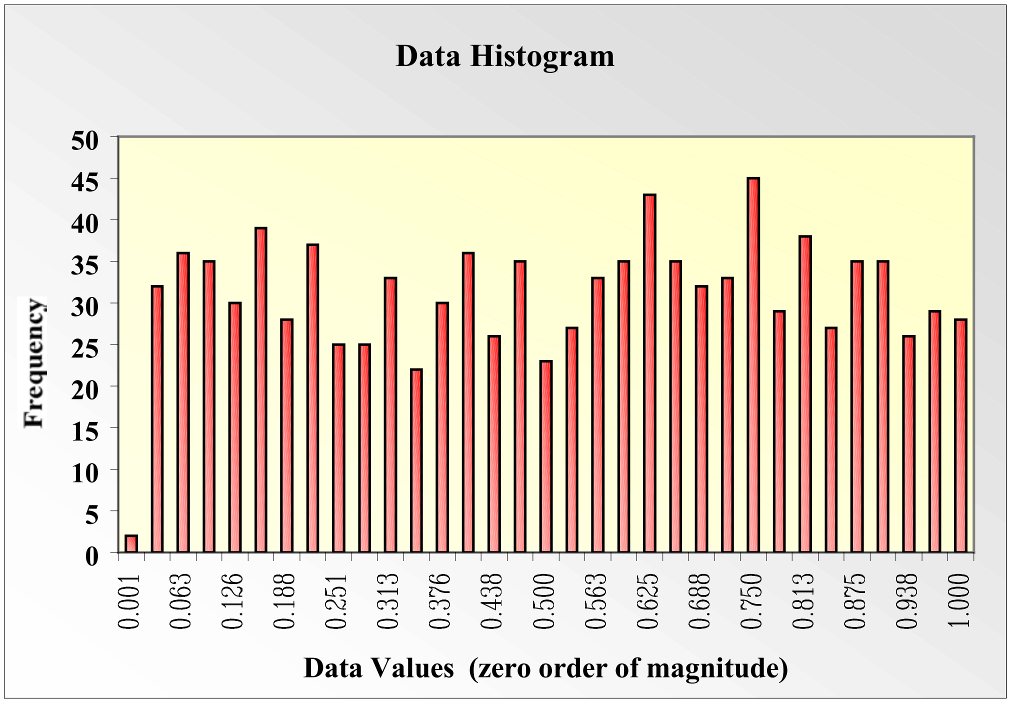

Figure 6.1 Histogram of the input vector data values.

Figure 6.3 shows the accuracy for the multiply-and-add circuit that produces 256 partial sums. As can be seen from this figure, the least accurate answer returned by the circuit is about 99.84% accurate. The multiply-and-accumulate circuit outputs 17 partial sums. Figure 6.4 shows the accuracy of the multiply-and-accumulate circuit as compared to the answer computed by the host processor. As can be seen from the figure all the answers are above 99.98% accurate.

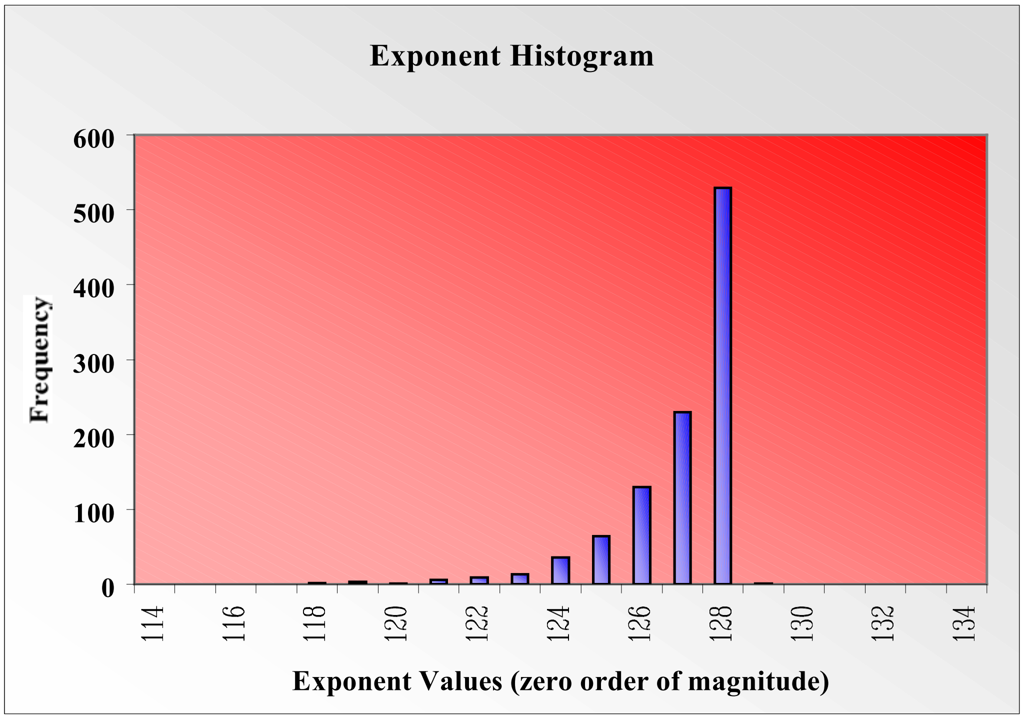

Figure 6.2 Histogram of exponents of the input vector data values.

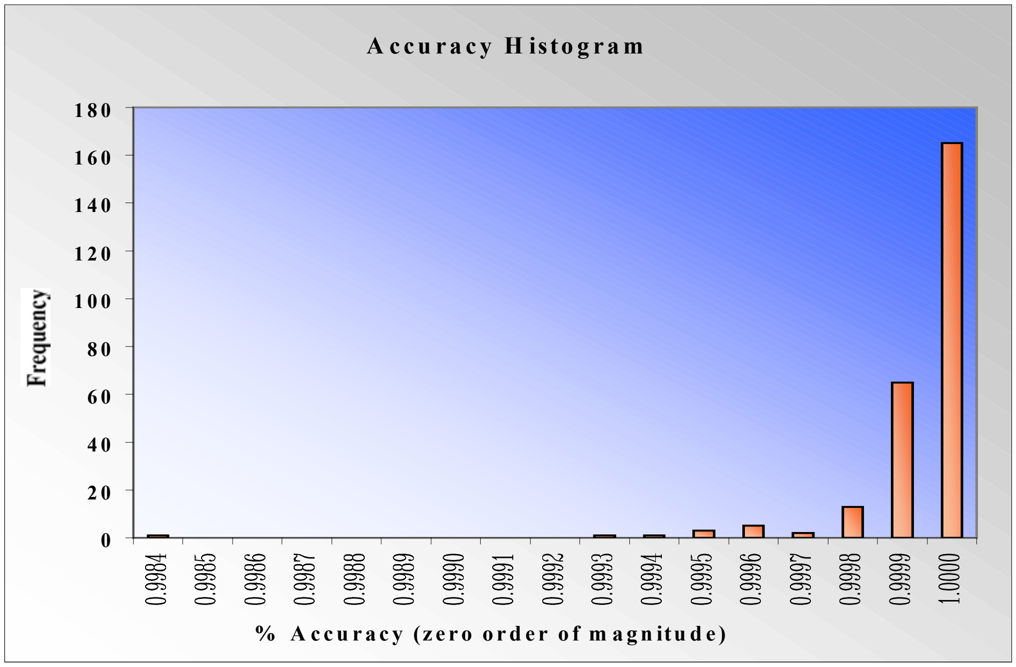

Figure 6.3 Accuracy Histogram for the multiply-and-add circuit.

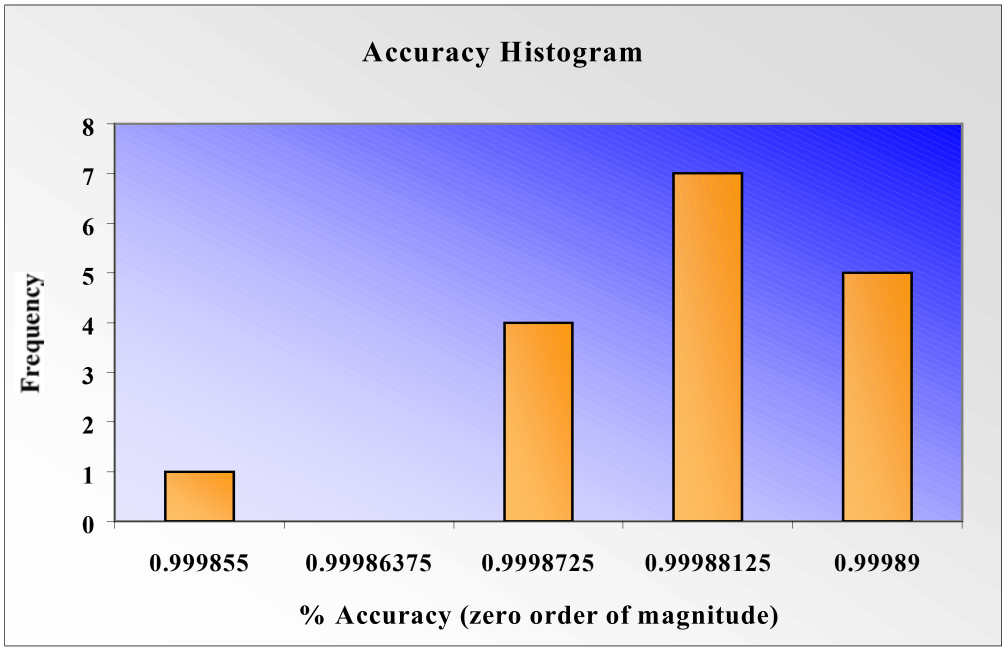

Figure 6.4 Accuracy histogram for the multiply-and-accumulate circuit.

6.1.2 Analysis of Uniform Data Value Distribution for Two Orders of Magnitude

Figures 6.5 through 6.8 are analogous to Figures 6.1 through 6.4 with the exception that the uniform data range is over two orders of magnitude (approximately 0 to 100). Figure 6.5 shows the uniform distribution of the data over this range. Figure 6.6 shows the corresponding distribution of the exponents. As can be seen from this distribution, the maximum exponent is 135, and the minimum exponent is 119. Thus the maximum number of bits shifted out is (135 – 119 = 16). So, the mantissa associated with the exponent value of 119 is completely shifted out, i.e., it is normalized to zero. Figure 6.7 shows the accuracy of the multiply-and-add circuit. As can be seen, the circuit is still very accurate with the least accurate result being 99.39% accurate. Figure 6.8 shows the accuracy of the multiply-and-accumulate circuit. This circuit also is very accurate with a lowest accuracy of 99.89%.

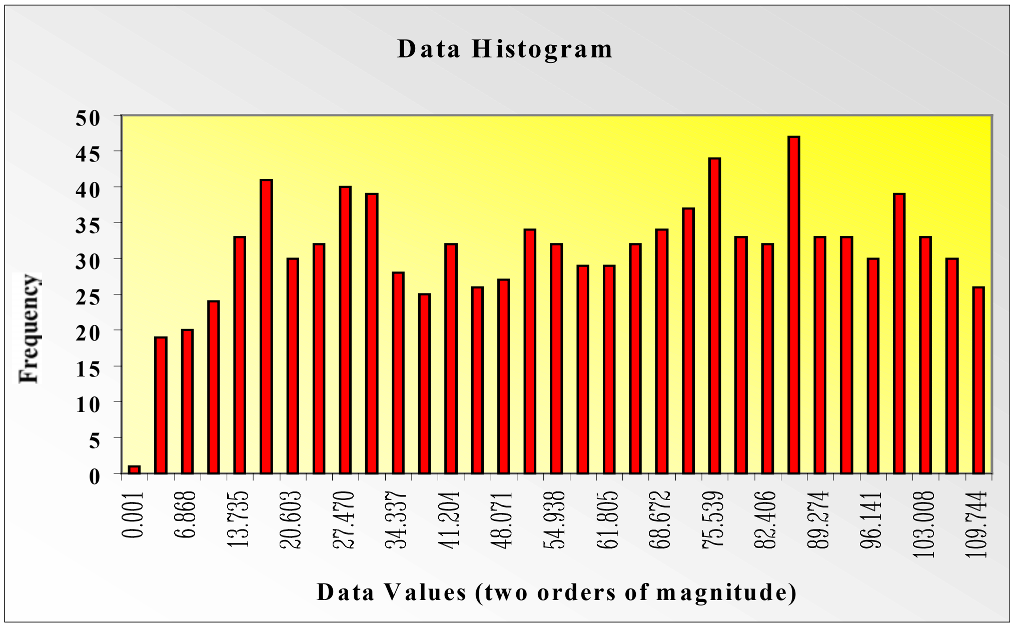

Figure 6.5 Histogram of the input vector data values.

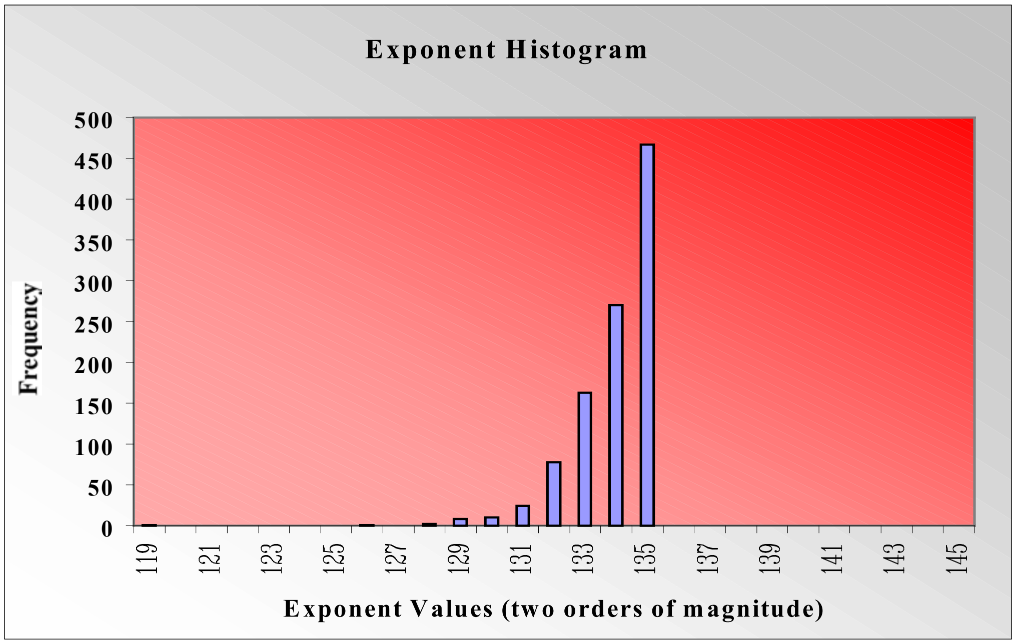

Figure 6.6 Histogram of exponents of the input vector data values.

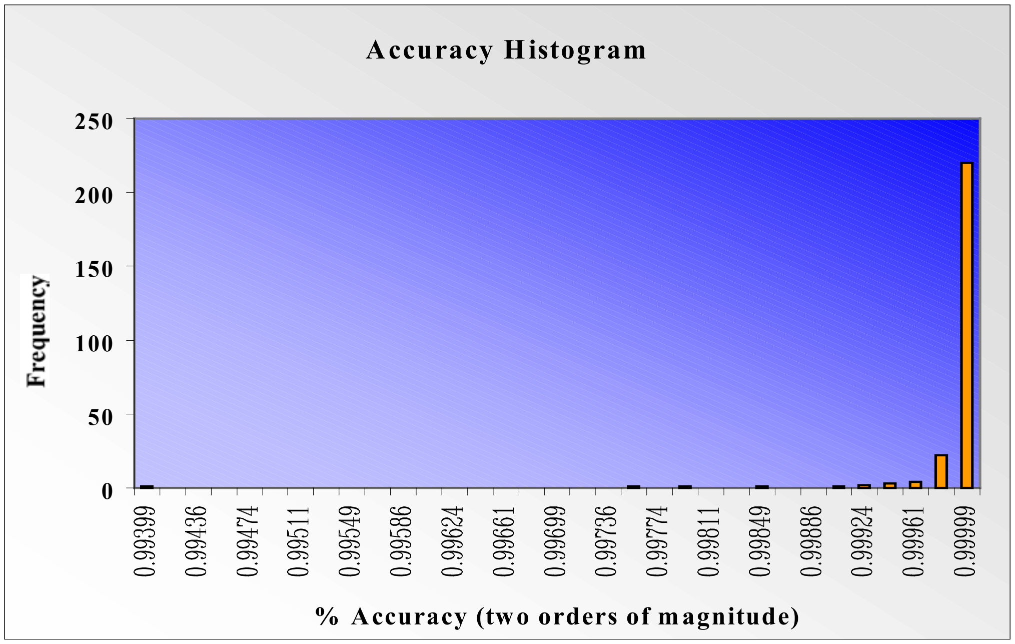

Figure 6.7 Accuracy histogram for multiply-and-add circuit.

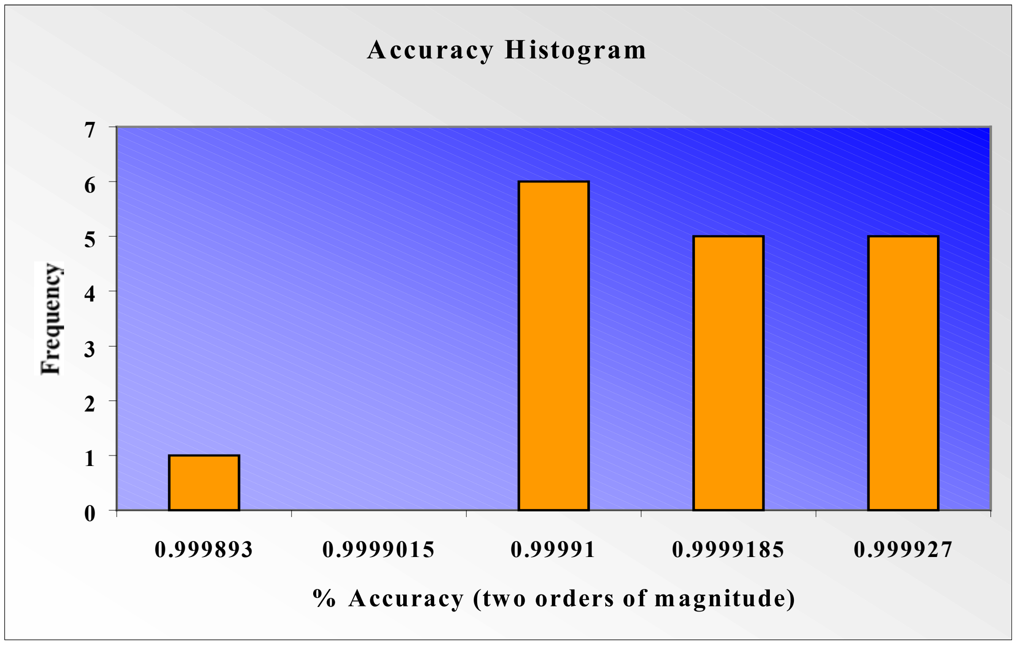

Figure 6.8 Accuracy histogram for the multiply-and-accumulate circuit.

6.1.3 Analysis of Uniform Data Value Distribution for Three Orders of Magnitude

Figures 6.9 through 6.12 show the distribution of the data values, the corresponding distribution of the exponents and the accuracy of the results of the two implementations for three orders of magnitude dynamic range. As can be seen from the accuracy histogram of the two implementations, the accuracy decreases as compared with previous experiments. This is because of the number of bits being shifted out of vector elements is increased.

6.1.4 Analysis of Uniform Data Value Distribution for Four Orders of Magnitude

Figures 6.13 through 6.16 show the distribution of the data values, the corresponding distribution of the exponents, and the accuracy of the results of the two implementations for four orders of magnitude dynamic range. As can be seen from the accuracy histogram for the multiply-and-add circuit (Figure 6.15), the accuracy for a small number of the partial results is less than 50%. However the value of these numbers themselves are relatively very small compared to some of the larger values and their effect on the overall accuracy of the final result is negligible. For example, if two numbers 0.0001 and 1000 are multiplied together, the result is 0.1. Consider two other numbers, say 10 and 1000, then the result of multiplication is 10000. Now, 0.1 is relatively small compared to 10000 and if it is added to 10000 the percentage change in the final answer is negligible. Thus approximating 0.1 to zero is reasonable in this context. This is also indicated by the accuracy histogram of the multiply-and-accumulate circuit (Figure 6.16) which does not show any significant degradation in performance for the same data set. The degradation in accuracy for some individual partial sums (Figure 6.15) that are relatively very small is expected because of the number of bits being shifted out.

6.1.5 Analysis of Uniform Data Value Distribution for Five Orders of Magnitude

Figures 6.17 through 6.20 show the distribution of the data values, the corresponding distribution of the exponents, and the accuracy of the results of the two implementations for the data range of five orders of magnitude. It can be seen that some of the partial sums are again very inaccurate; some of them being zero percent accurate. Again, it should be pointed out that these numbers are relatively small compared to some of the larger numbers and their effect on the overall accuracy of the final result is generally negligible. This is also indicated by the accuracy histogram of the multiply-and-accumulate circuit (Figure 6.20), which again does not show any significant degradation in performance.

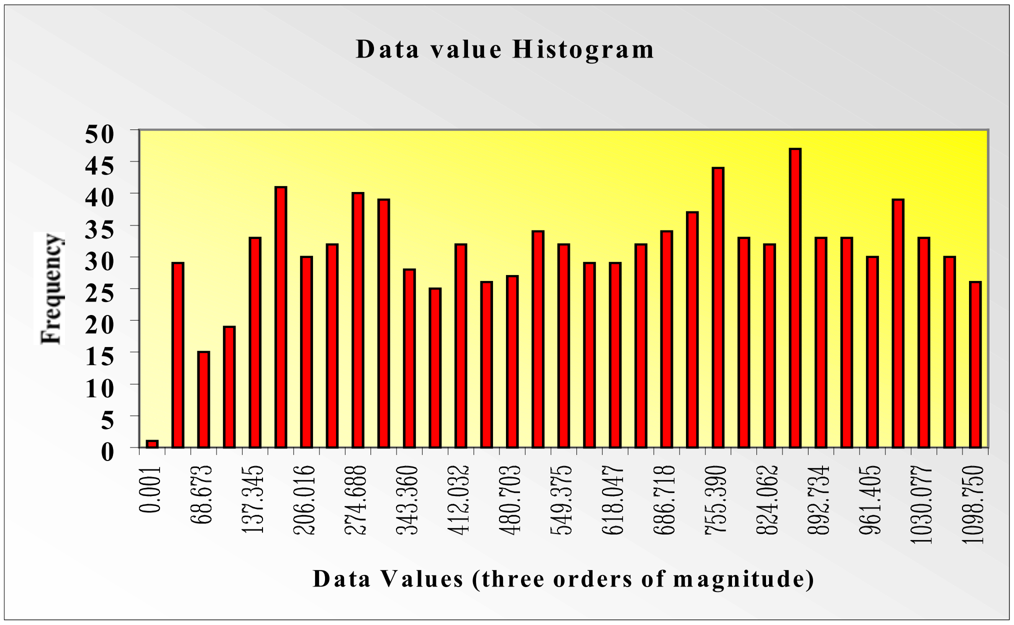

Figure 6.9 Histogram of the input vector data values.

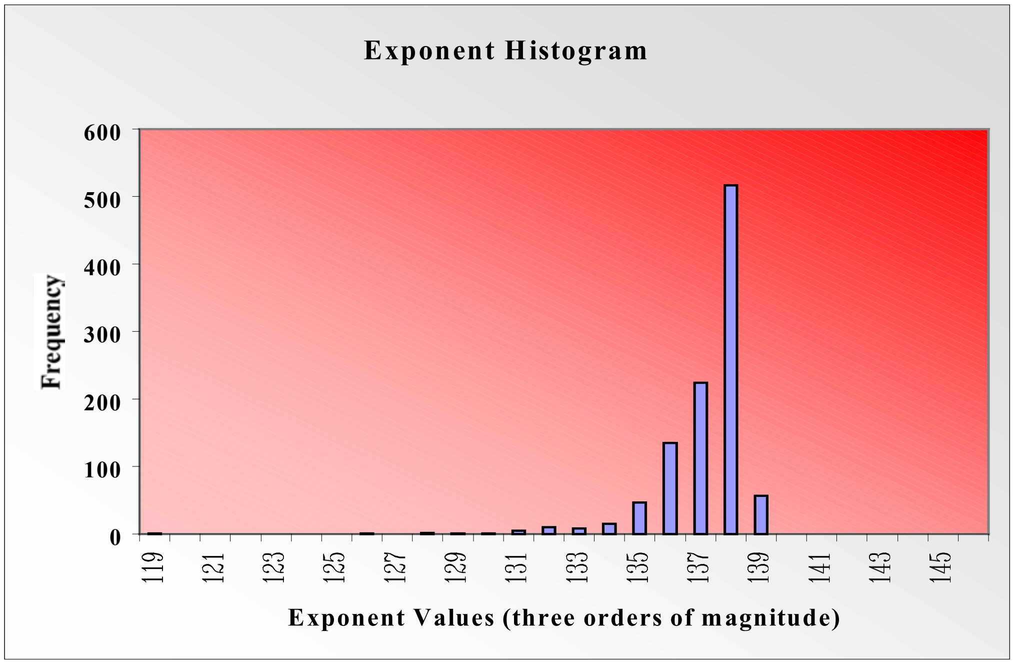

Figure 6.10 Histogram of the exponents of
input vector data values.

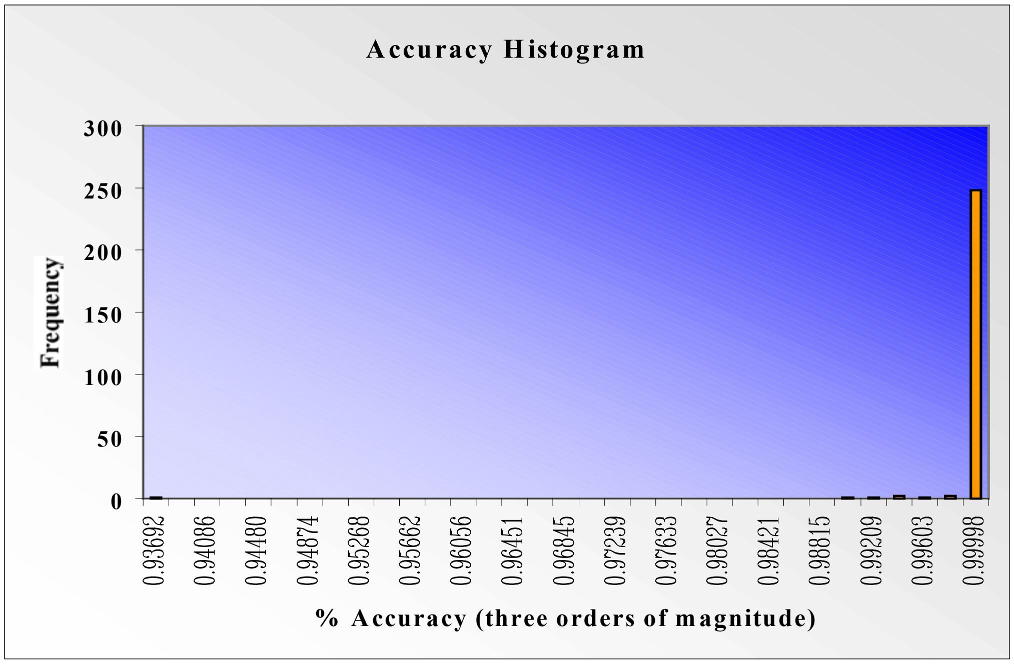

Figure 6.11 Accuracy histogram for the
multiply-and-add circuit.

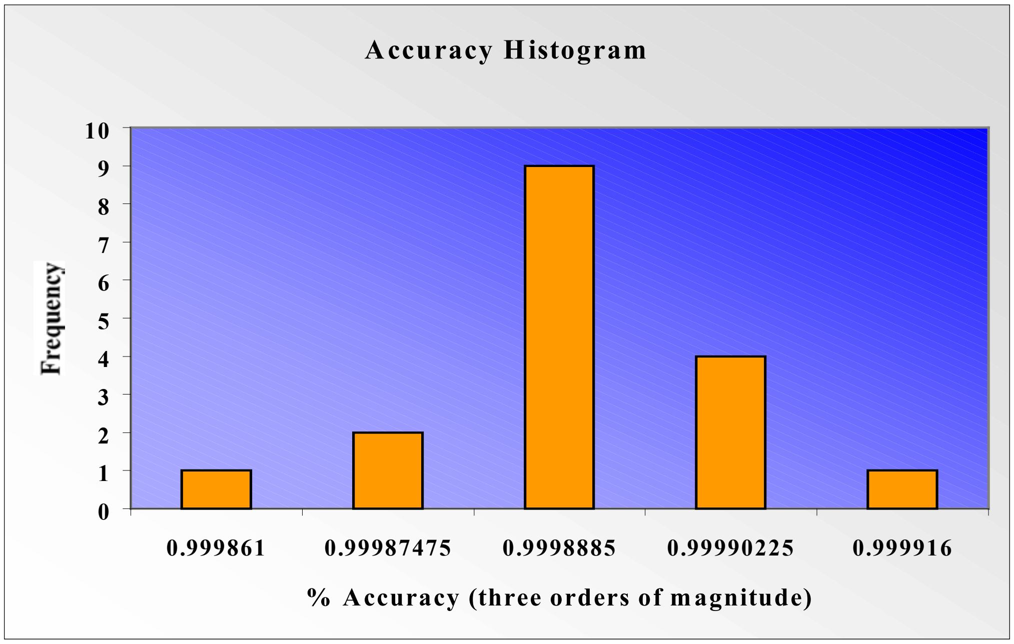

Figure 6.12 Accuracy histogram for the multiply-and-accumulate circuit.

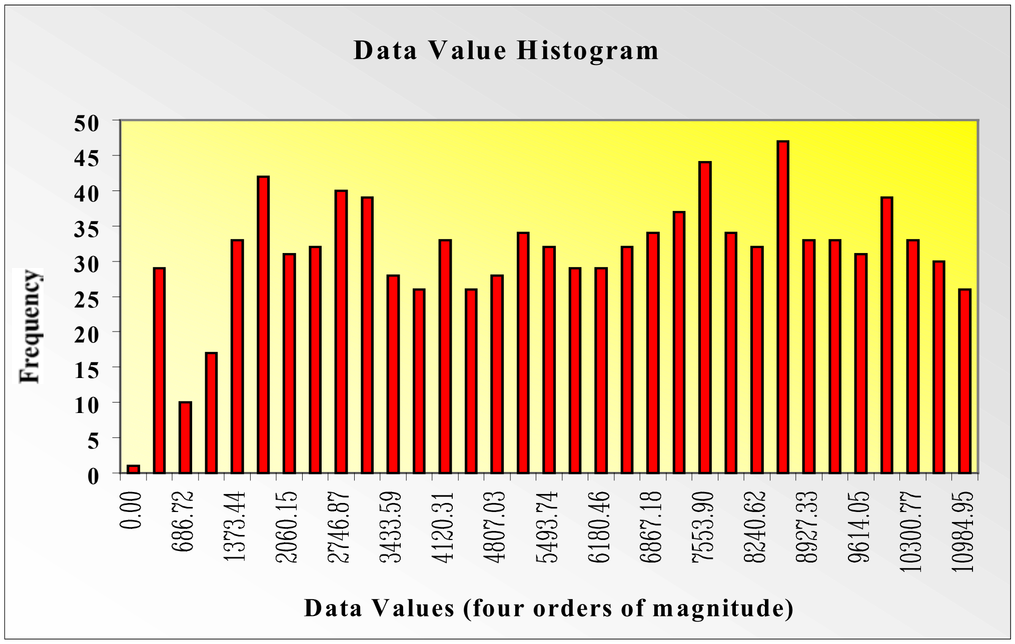

Figure 6.13 Histogram of the input vector data values.

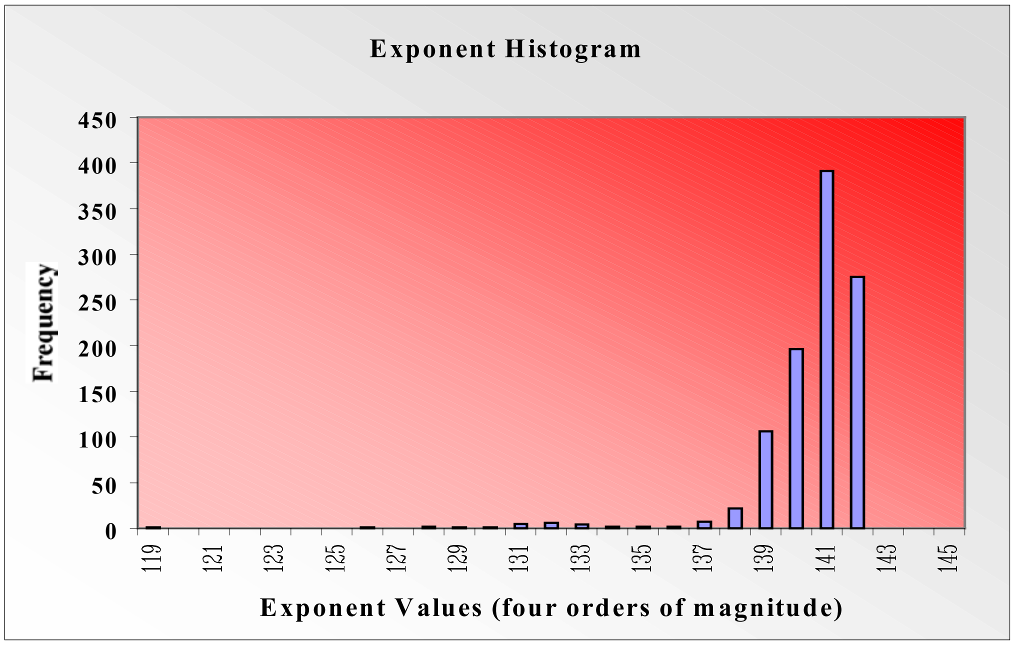

Figure 6.14 Histogram of the exponents of input vector data values.

6.1.6 Analysis of Non-Uniform Data Value Distribution

All of the previous cases had a very uniform distribution of data over the entire range of the bounding data values. Now consider a case where the data value distribution is not very uniform. In particular, consider a "worst case" situation in which most of the data is grouped together and there is one outlying number that is much greater that all the other numbers. This type of data distribution is shown in Figure 6.21 and its corresponding exponent distribution is shown in Figure 6.22. This is a worst case scenario because virtually all of the data values have most of their bits shifted out in accordance with the maximum exponent value.

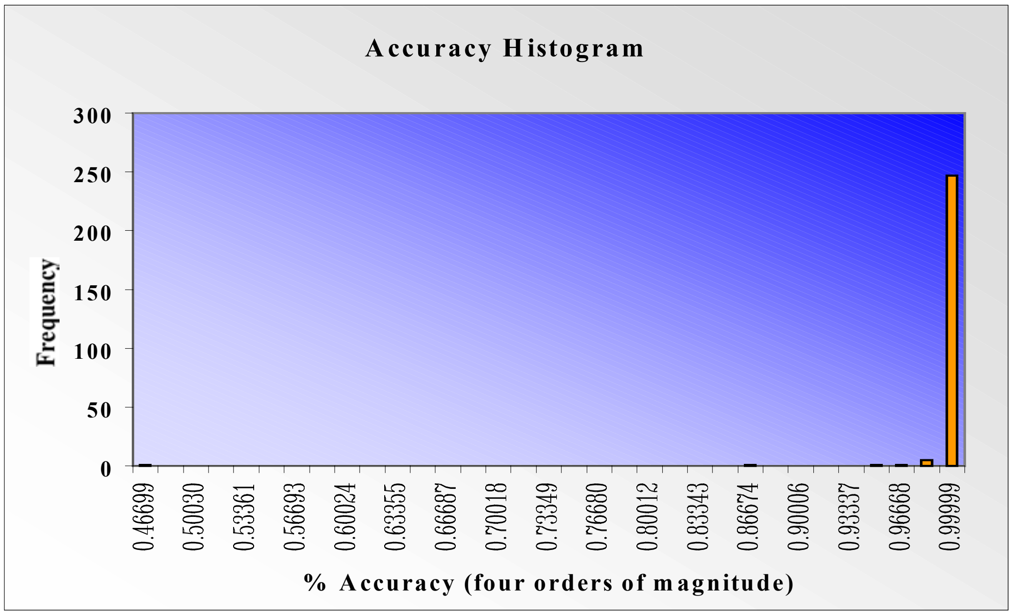

Figure 6.15 Accuracy histogram for the multiply-and-add circuit.

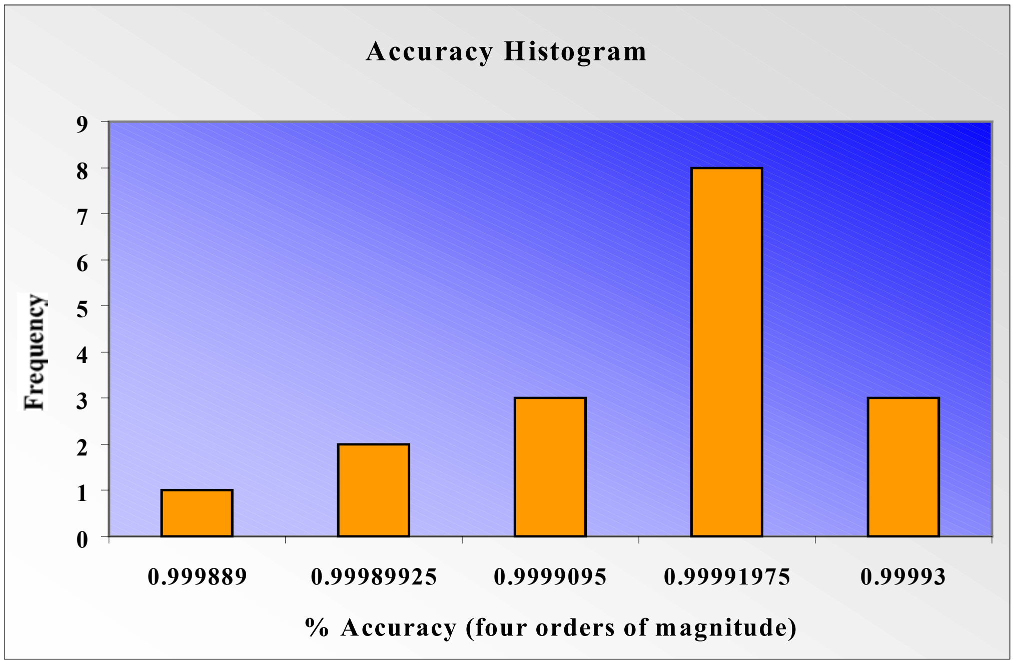

Figure 6.16 Accuracy histogram for the multiply-and-accumulate circuit.

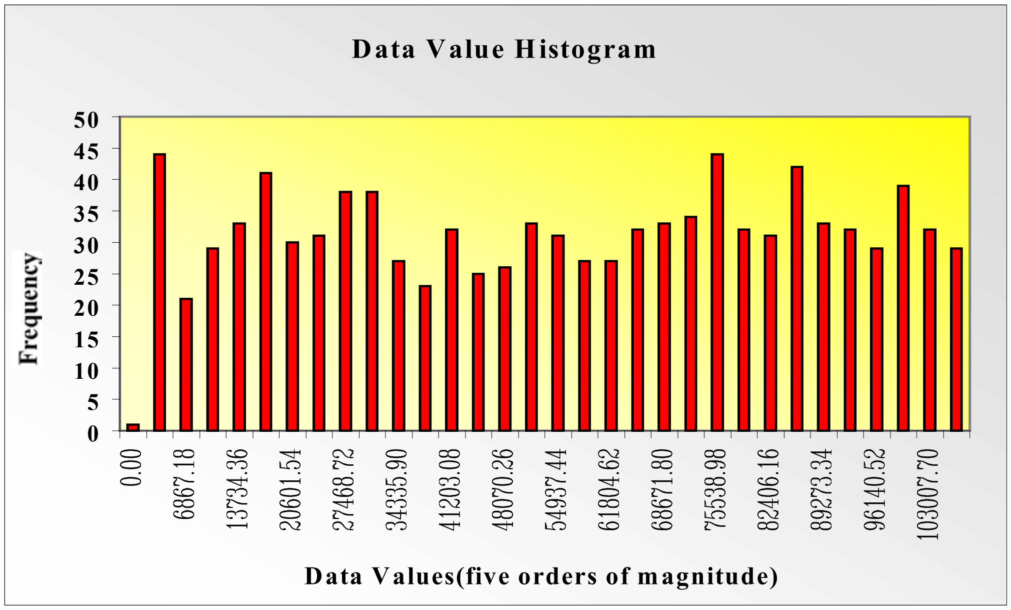

Figure 6.17 Histogram of the input vector data values.

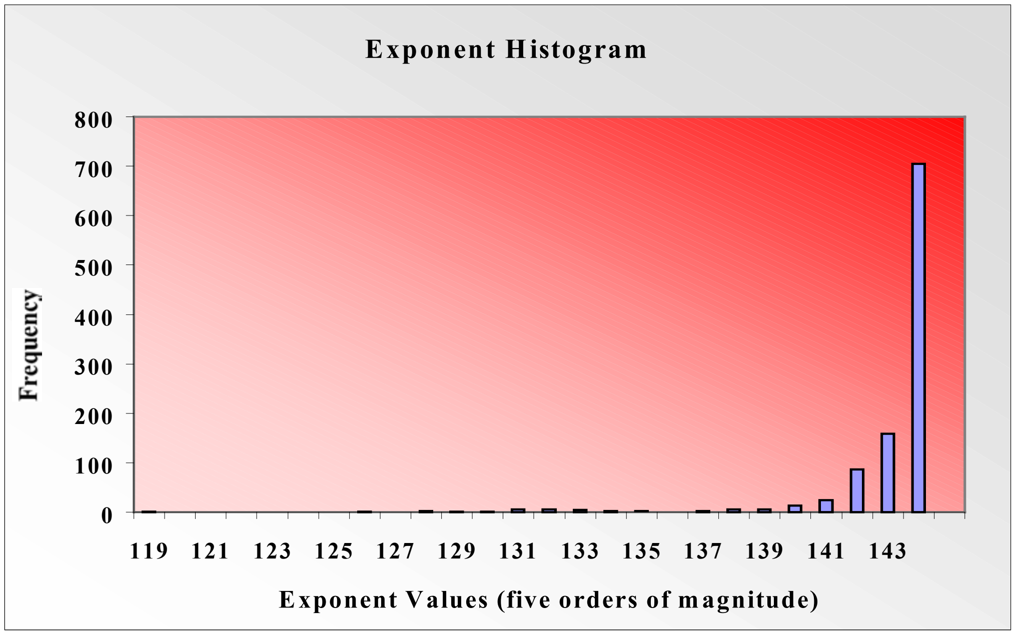

Figure 6.18 Histogram of the exponents of input vector data values.

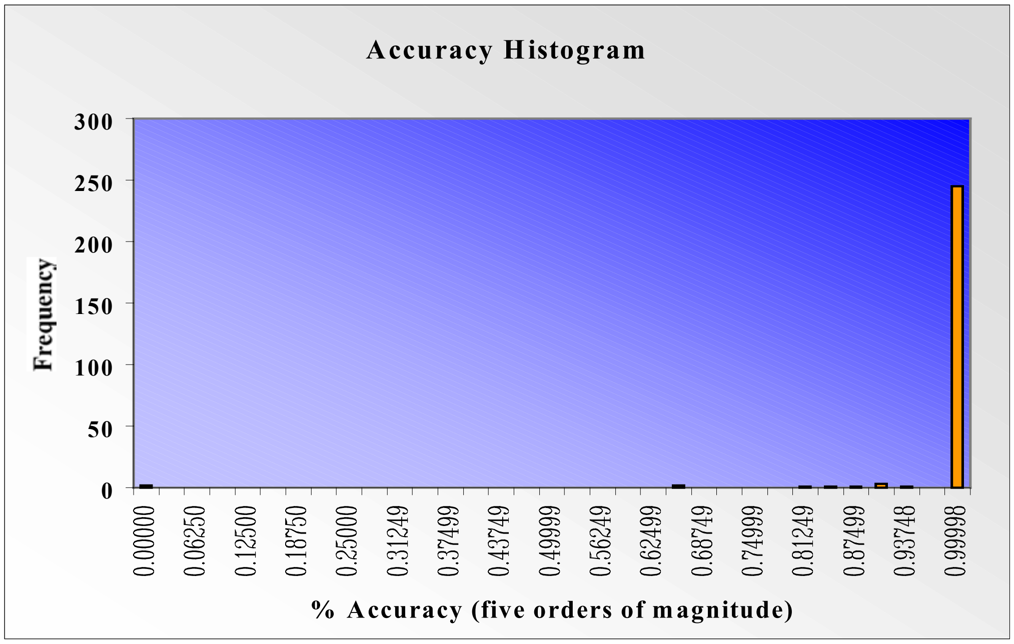

Figure 6.19 Accuracy histogram for the multiply-and-add circuit.

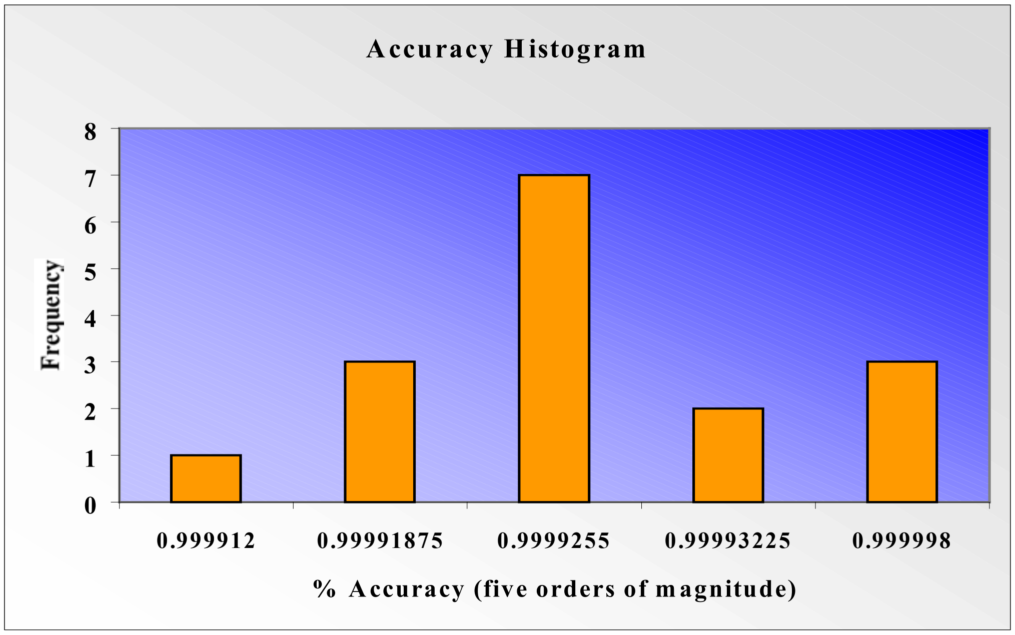

Figure 6.20 Accuracy histogram for the multiply-and-accumulate circuit.

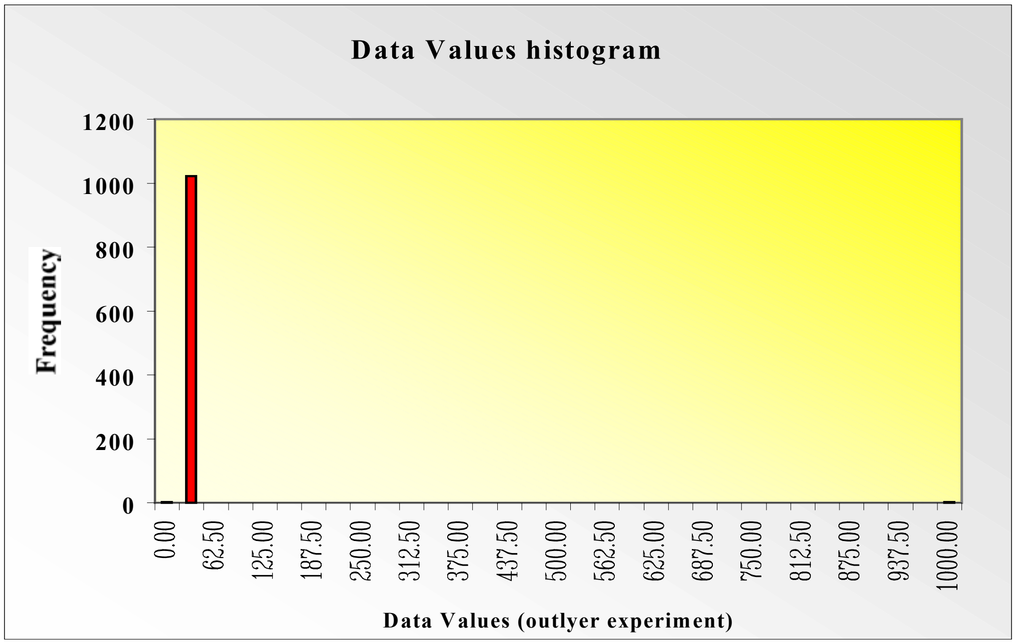

Figure 6.21 Histogram of the input vector data values.

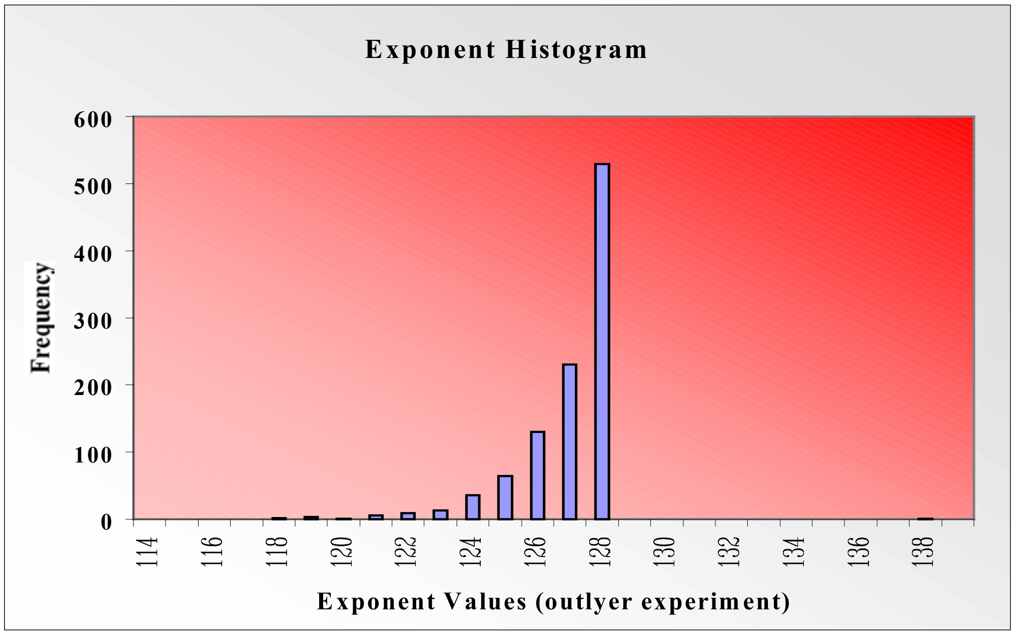

Figure 6.22 Histogram of the exponents of input vector data values.

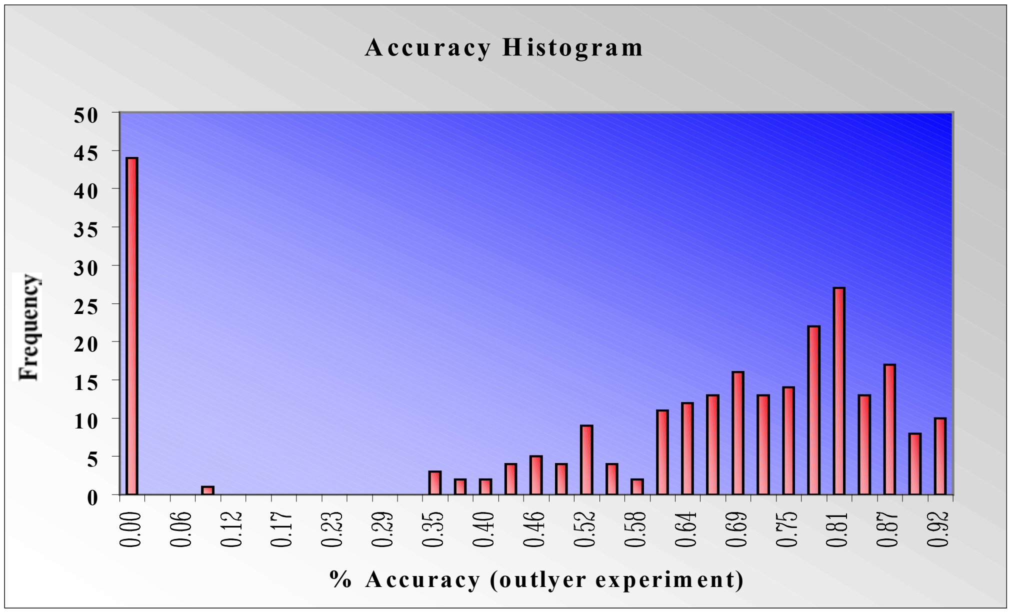

Figure 6.23 Accuracy histogram of the multiply-and-add circuit.

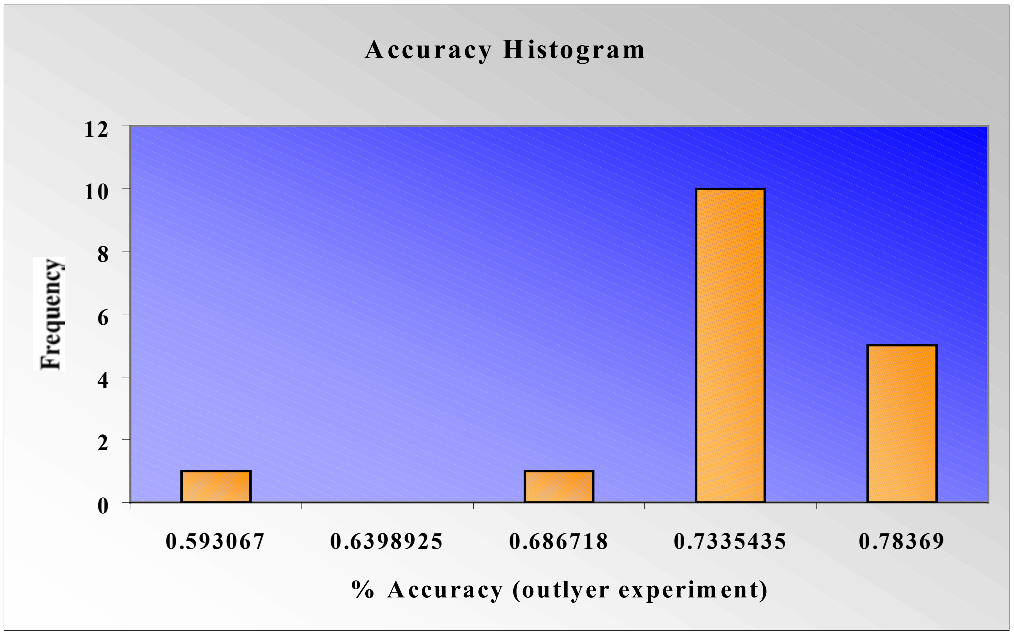

Figure 6.24 Accuracy histogram of the multiply-and-accumulate circuit.

6.2 Conclusions

As demonstrated in the numerical studies described above, it is seen that the circuits produce excellent accuracy results for data distributions that are uniformly distributed. Poor results are obtained for the case where one or a few of the elements are much larger than the rest of the numbers. This is because of the block-floating-point architecture being used to implement these circuits. The block-floating-point architecture normalizes all the exponents to the maximum exponent by shifting out the least significant bits so that all the exponents are equal and then all the operations are integer arithmetic operations, which are much easier to perform than the floating point operations. The shifting out of the bits produces an inaccuracy in the computations. For all the ranges of numbers considered, if the numbers are uniformly distributed, then the exponent distribution has an increasing exponential shape with a majority of the numbers close to the maximum value in the exponent domain. This results in a smaller number of bits from the mantissas of the numbers being shifted out, on the average. Another important point is that the multiply implementation uses a 15-bit mantissa, which implies that the mantissa of the input floating-point number is truncated to 15 bits from 23 bits, thereby introducing some inaccuracies.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

The computational complexity associated with STAP can easily overwhelm the computational capabilities of processors used on current airborne platforms. The typical computational throughput required by a third-order Doppler-factored STAP is 39.81 billion floating-point operations per second (Gflops/s) [18]. The most computationally intensive part of the STAP algorithm is the computation of the adaptive weights, and constitutes approximately 91% of all the computations needed in adaptive processing [18]. For some applications in radar processing, precise answers may not be required but a close approximation will be as effective. This fact may be used to reduce the complexity associated with the traditional approach for computing the adaptive weights. Another aspect is the application of reconfigurable computing, which can be used to achieve a speedup associated with doing the computations in hardware. An important aspect of using reconfigurable computing is that it can be fine-tuned for the application at hand, so the same system can be used to perform other kinds of processing.

The first major part of this research was to find an alternative approach to solving for adaptive weights where the accuracy of the answers can be traded for computations. This concept was investigated in Chapter III where the CG approach (an iterative approach) was compared with the traditional QR-decomposition approach for computing adaptive weights. The results illustrated in Section 3.4 show that the conjugate-gradient approach reduces the computations needed while losing some accuracy.

The second goal of the research was to implement some of the computations needed in adaptive weight calculation on a reconfigurable computing system, thereby getting an improvement in performance. It was observed that the most frequent operation needed in computing adaptive weights was the computation of the inner-product of two vectors. It was decided to program the reconfigurable computing system to compute the inner-product of two vectors using block-floating-point arithmetic. Two different architectures were implemented, each having its own tradeoffs. Numerical studies were carried out on the two implementations and the results were presented in Chapter VI. A basic conclusion of the study is that acceptable accuracy can be obtained provided that the distribution of the data values is relatively uniform.

The current research implements just a part of the computations needed to compute the adaptive weights. This is because of the fact that the number of reconfigurable resources on the current system is very limited. As boards with more reconfigurable resources become available, the possibility of implementing all adaptive weight computation on FPGAs becomes realistic. The current design of the system does not perform any reconfiguration of the board on the fly. The reconfiguration of the board on the fly would allow the reconfigurable board to toggle between different configurations that work on the processed data from the previous configuration. The challenges associated with such dynamic reconfiguration is an excellent area for future research.

REFERENCES

[1] M. I. Skolnik, *Introduction to Radar Systems*, McGraw Hill Inc., New York, NY, 1962.

[2] J. C. Toomay, *Radar Principles for the Non-Specialist*, Van Nostrand Reinhold, New York, NY, 1989.

[3] F. M. Staudaher, "Airborne MTI," Chapter 16, *Radar Handbook*, editor M.I. Skolnik, McGraw-Hill, New York, NY, 1990.

[4] R. J. Mailloux, *Phased Array Antenna Handbook*, Artech House, Boston, MA, 1993.

[5] J. Ward, Space-*Time Adaptive Processing for Airborne Radar*, Technical Report 1015, Massachusetts Institute of Technology, Lincoln Laboratory, Lexington, MA, 1994.

[6] Title3 – Executive Order 12931 of October 13, 1994, Section 1, paragraph (d); available Fed. Reg. Vol. 59, No. 199, Monday, October 17, 1994.

[7] K. Skahill, *VHDL for Programmable Logic*, Addison-Wesley, Menlo Park, CA, 1996.

[8] G.H. Golub and F.V. Charles, *Matrix Computations*, 2nd Edition, John Hopkins University Press, Baltimore, MD, 1989.

[9] *WildOne Hardware Reference Manual* 11927-0000 Revision 0.1, Annapolis Micro Systems Inc., MD, 1997.

[10] G. Booch, I. Jacobson, and J. Rumbaugh, "The Unified Modeling Language for Object Oriented Development," Documentation Set Version 1.1, September 1997.

[11] J. M. West, *Simulation of Communication time for a Space-Time Adaptive Processing Algorithm on a Parallel Embedded System*, Section 2.3.1, Masters Thesis, Texas Tech University, August 1998.

[12] A. W. Rihaczek, *Principles of High-Resolution Radar*, McGraw Hill Inc., New York, NY, 1969.

[13] J. L. Eaves and E. K. Reedy, *Principles of Modern Radar*, Van Nostrand Reinhold, New York, NY, 1987.

[14] M. I. Skolnik, *Radar Handbook*, Second Edition, McGraw Hill Inc., New York, NY, 1990.

[15] G. V. Morris, *Airborne Pulsed Doppler Radar*, Artech House, Norwood, MA, 1988.

[16] P. K. Rowe, *COTS Radar and Sonar Systems Solutions,* Multiprocessor Toolsmiths Inc., Kanata , ON Canada, 1996.

[17] D. Taylor and C. H. Westcott, *Principles of Radar*, Cambridge University Press, London, 1948.

[18] K. C. Cain, J. A. Torres, and R. T. Williams, "*Real-Time Space-Time Adaptive Processing Benchmark*", Mitre Technical Report: MTR 96B0000021, Mitre, Center for Air Force C3 Systems, Bedford, MA, February 1997.

[19] W. H. Press, S. A. Teukolsky, W. T. Vellerling, B. P. Flannery, *Numerical Recipes in C, The Art of Scientific Computing*, Cambridge University Press, London, 1993.

[20] D. G. Luenberger, *Linear and Nonlinear Programming*, Second Edition, Addison-Wesley, Reading, MA, 1984.

[21] Real-Time MCARM Data Sets, http://sunrise.oc.rl.af.mil, Rome Laboratory.

[22] W. W. Smith, *Handbook of Real-Time Fast Fourier Transforms*, IEEE Press, Piscataway, NJ, 1995.

[23] T. T. Do, H. Kropp, M. Schwiegershausen, and P. Pirsch, "Implementation of Pipelined Multipliers on Xlinix FPGAs," *Proceedings of the 7th International Workshop Field-Programmable Logic and Applications*, W. Luk, P. Y. K. Cheung, M. Glesner, eds., Springer Verlag, September 1997.

[24] M. M. Mano, *Digital Logic and Computer Design,* Prentice Hall Inc., Englewood Cliffs, N.J., 1992.

[25] M. M. Mano, *Computer System Architecture*, Third Edition, Prentice Hall Inc., Englewood Cliffs, N.J., 1993.

[26] *Xilinx XC4000E and XC4000X (XC4000EX/XC4000XL) FPGA Series Datasheet*, v1.4, http://www.xilinx.com, 1997.

APPENDIX

THE XILINX 4000 SERIES FPGA

An FPGA device consists of an array of programmable logic cells called configurable logic blocks (CLBs) interconnected by wires called routing channels running vertically and horizontally, and surrounded by a perimeter of programmable Input/Output Blocks (IOBs). The CLBs consist of a number of diverse logic gates and registers. The routing channels are connected by programmable elements, which when programmed, define the interconnection between the various CLBs and IOBs. Thus, these logic resources can be programmed in various ways to define the required logic function. The following sections discuss briefly the architectural features of the Xilinx 4000 series FPGAs. This material is summarized from [25].

A.1 The Xilinx 4000 Series

The Xilinx 4000 series FPGAs are implemented with a regular, flexible, programmable architecture of CLBs, interconnected by a powerful hierarchy of versatile routing resources and surrounded on the periphery by IOBs. The CLBs provide the functional elements for constructing the user logic while the IOBs provide the interface between the package pins and the internal signal lines. The programmable interconnect resources provide the routing paths to connect the input and outputs of these configurable elements to the appropriate networks. The functionality of each circuit block is customized during configuration by programming internal static memory cells. The values stored in these memory cells determine the logic functions and the interconnections implemented in FPGA. The XC4028 EX FPGA, which will be used in this project consists of 32×32 CLB matrix for a total of 1024 CLBs, 2432 logic cells, a maximum of 28000 logic gates, 2560 flip-flops and 256 user I/O. Each of these circuits is described in the following sections.

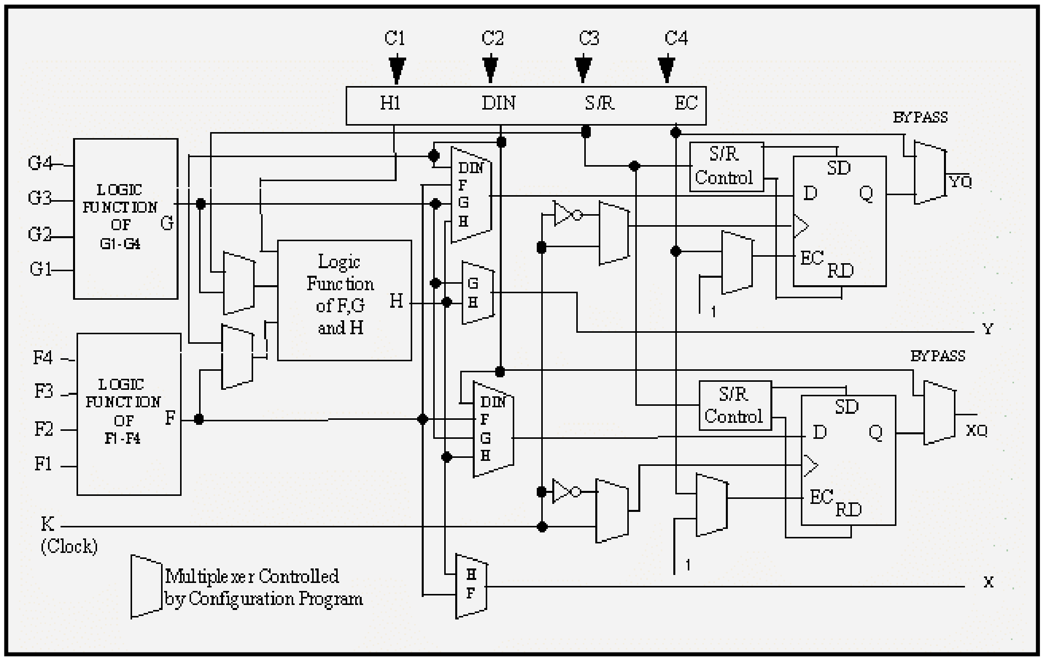

Figure A.1. The Xilinx XC4000 series CLB.

A.2 The XC4000 Series Configurable Logic Block

The Configurable Logic Blocks implement most of the logic in an FPGA. The principle elements of a CLB are shown in Figure A.1. Each CLB contains a pair of flip-flops and two independent 4-input function generators. The two 4-input function generators (F and G) offer unrestricted versatility. Most combinatorial logic functions need four or fewer inputs. However, a third function generator (H) is provided. The H function generator has three inputs. One or both of these inputs can be the outputs of F and G; the other input(s) are from outside the CLB. The CLB can therefore implement certain functions of up to nine variables. Each CLB contains two flip-flops that can be used to store the function generator outputs. However, the flip-flops and function generators can also be used independently. The input labeled DIN can be used as a direct input to either of the two flip-flops. H1 can drive the other flip-flop through the H function generator.

Function generator outputs can also be accessed from outside the CLB, using two outputs independent of the flip-flop outputs. This versatility increases logic density and simplifies routing. Thirteen CLB inputs and four CLB outputs provide access to the function generators and storage elements. These inputs and outputs connect to the programmable interconnect resources outside the block.

The versatility of the CLB function generators significantly improves the system speed. In addition, the design software tools can deal with each function generator independently thus improving cell usage. Each of the function generators F and G in a CLB contain dedicated arithmetic logic for the fast generation of carry and borrow signals. Figure A.2 shows the fast carry logic present within each CLB. This extra output is passed on to the next CLB function generator above or below. The carry chain is independent of normal routing resources. Dedicated fast carry logic greatly increases the efficiency and performance of adders, subtracters, accumulators, comparators and counters. The two four input function generators can be configured as a 2-bit adder with built-in hidden carry that can be expanded to any length. This dedicated carry circuitry is so fast and efficient that conventional speed-up methods like carry generate/propagate are generally meaningless even at the 16-bit level, and of marginal benefit at the 32-bit level.

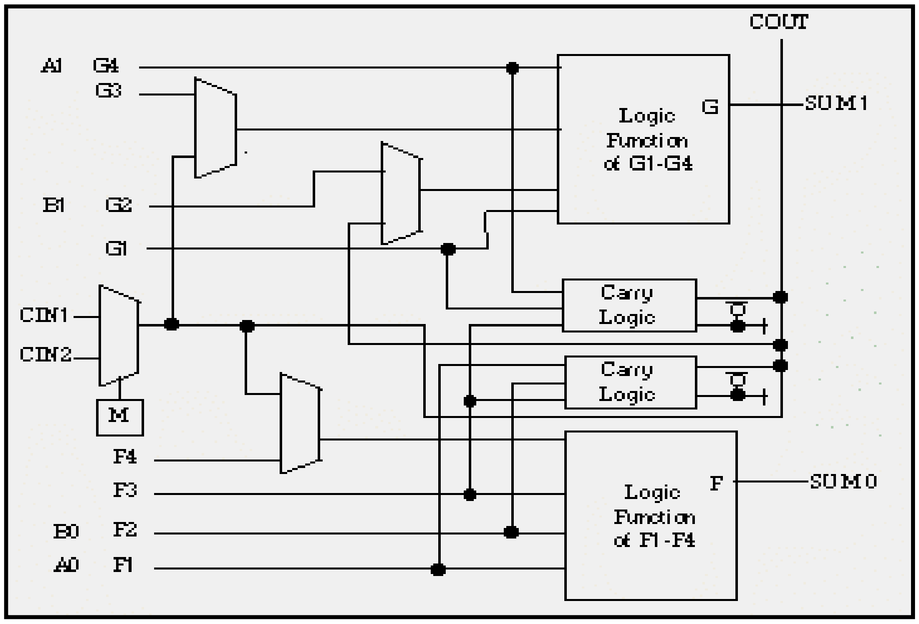

Figure A.2. The dedicated fast carry logic in the XC4000.

A.3 The XC 4000 Series Input/Output Blocks

The user-configurable input/output blocks (IOBs) in the XC4000 series provide the interface between external package pins and the internal logic. Each IOB controls one package pin and can be defined for input, output, or bi-directional signals. Figure A.3 shows a simplified block diagram of the XC4000 IOB. Two paths, labeled I1 and I2, bring input signals into the array. Inputs also connect to an input register that can be programmed as either an edge-triggered flip-flop or a level-sensitive transparent-low latch. The I1 and I2 signals that exit the block can each carry either the direct or registered input signal. The input and output storage elements in each IOB have a common clock enable input, which through configuration can be activated individually for the input or output flip-flop or both. This clock enable operates exactly like the EC pin on the XC4000E CLB. It cannot be inverted within the IOB.

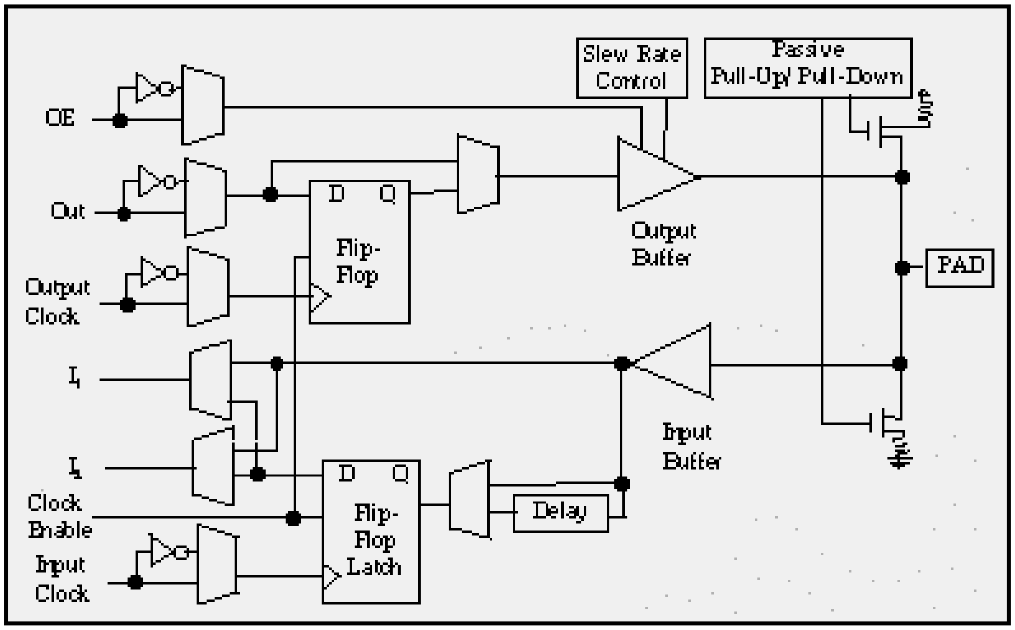

Figure A.3. The XC4000 Input/Output block.

A.4 The XC4000 Series Programmable Interconnect

The programmable interconnect consists of structured, hierarchical matrix of routing resources running vertically and horizontally between the CLBs to achieve efficient automated routing. All the internal connections are composed of metal segments with programmable switching points and switching matrices to implement the desired routing. The number of routing channels is scaled to the size of the array; i.e., it increases with the array size. The CLB inputs and outputs are distributed on all four sides of the block, providing additional routing flexibility (see Figure A.4).

There are four main types of interconnect, three are distinguished by the relative length of their segments: single-length lines, double-length lines and longlines. In addition, eight global buffers drive fast, low-skew nets most often used for clocks or global control signals. The single-length lines are a grid of horizontal and vertical lines that intersect at a switch matrix between each block. Figure A.4 illustrates the single-length interconnect surrounding one CLB in the array. Each switch matrix consists of programmable n-channel pass transistors used to establish connection between the single-length lines. For example, a signal entering on the right side of the Switch Matrix can be routed to a single-length line on the top, left or bottom sides, or any combination thereof. Single-length lines are normally used to conduct signals within a localized area and to provide the branching for nets with fanout greater than one. The function generator and control inputs to the CLB (F1-F4, G1-G4, and C1-C4) can be driven from any adjacent single-length line segment. The CLB clock (K) input can be driven from one-half of the adjacent single-length lines. Each CLB output can drive several of the single-length lines, with connections to both the horizontal and vertical longlines.

The doubled-length lines shown in Figure A.5 consists of a gird of metal segments twice as long as the single-length lines; i.e., a double-length line runs past two CLBs before entering a switch matrix. Double-length lines are grouped in pairs with the switch matrices staggered so that each line goes through a matrix switch at every other CLB location in that row or column. As with single-length lines, all the CLB inputs except K can be driven from any adjacent double-length line, and each CLB output can be drive by nearby double-length lines in both the vertical and horizontal planes. Double-length lines provide the most efficient implementation of intermediate length, point-to-point interconnections.

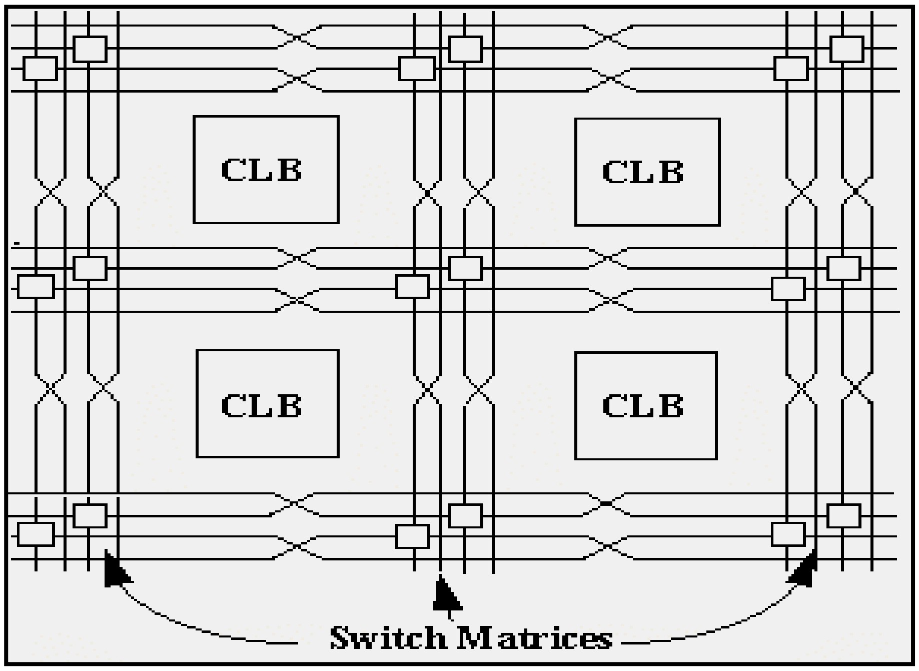

Figure A.4. Single length line in the XC4000.

Longlines form a grid of metal interconnects that run the entire length or width of the array. Special global buffers can drive additional vertical longlines, designed to distribute clocks and other high fanout control signals throughout the array with minimal skew. Longlines are intended for high fanout, time-critical signal nets. Each longline has a programmable splitter switch at its center, which can separate the line into two independent routing channels, each running half the width or height of the array. CLB inputs can be driven from a subset of the adjacent longlines. CLB outputs are routed to the longlines via tri-state buffers or the single-length interconnected lines. The XC 4000 long lines are shown in Figure A.6. The horizontal and vertical single and double length

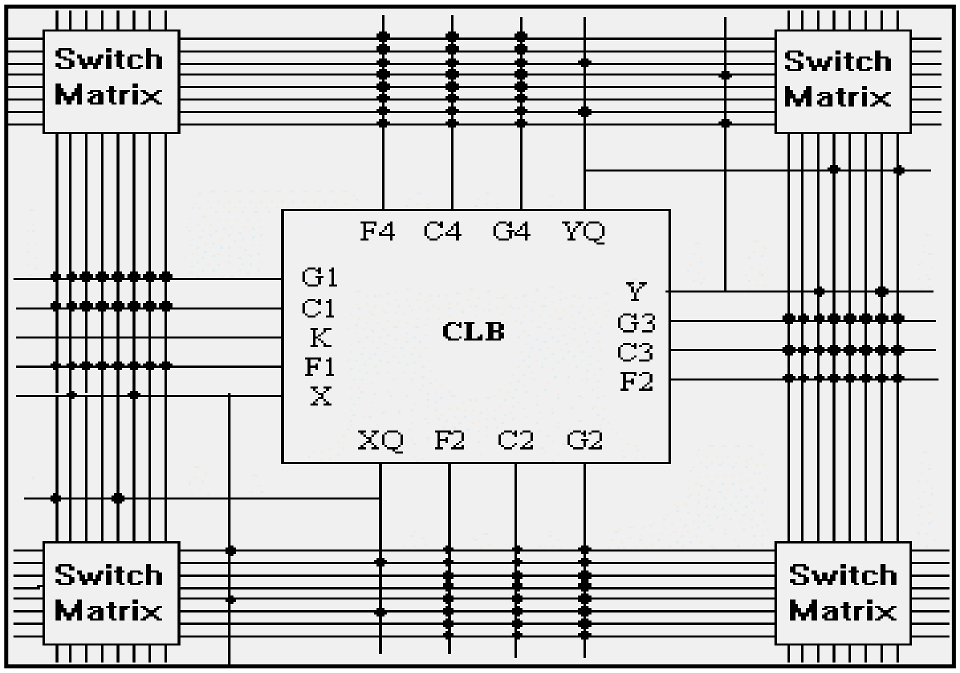

Figure A.5 Illustration of double length lines in the XC4000.

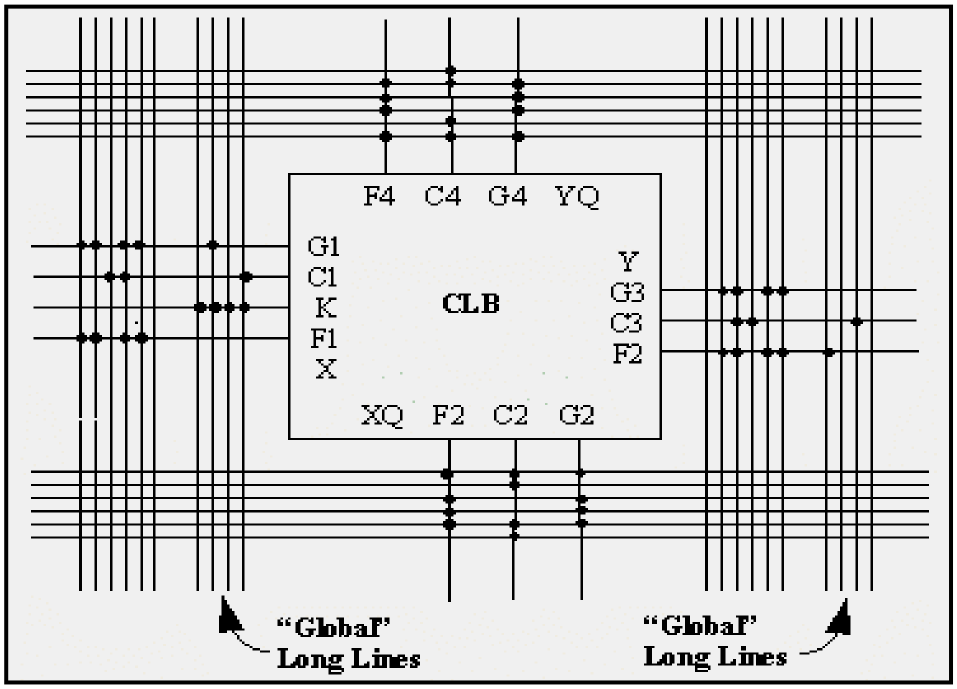

Figure A.6. Illustration of longlines of the XC4000.

lines intersect at a box called the programmable switch matrix (PSM). Each switch matrix consists of programmable pass transistors used to establish connections between the lines. The programmable switch matrix is shown in Figure A.7. For more details about the XC4000 the reader is referred to [26].

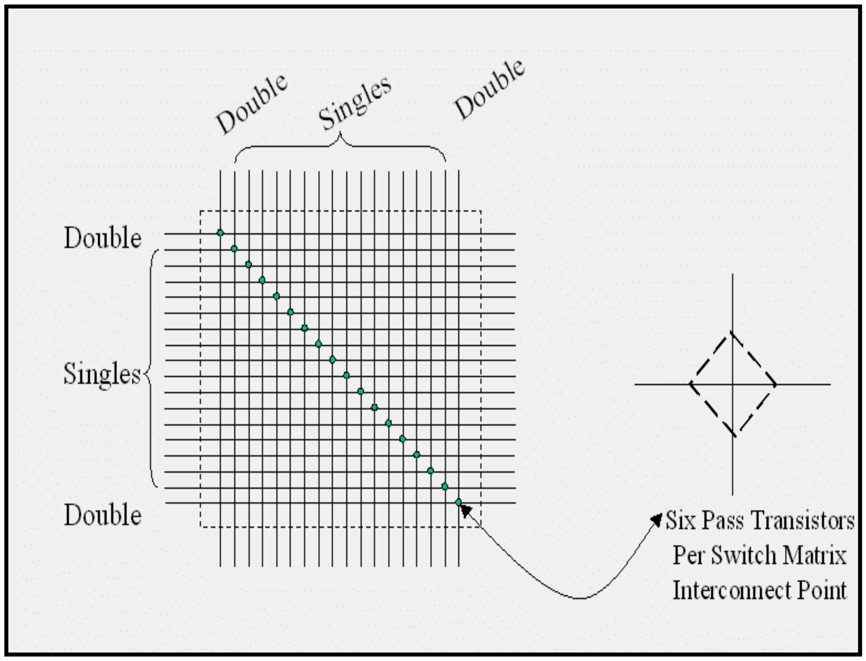

Figure A.7. Illustration of the programmable switching matrix of the XC4000.