April 2010

Implementation of a Real-Time Beamforming System on Field Programmable Gate Array

David Truong
Worcester Polytechnic Institute

Soe San Win
Worcester Polytechnic Institute

Project Number: XIH - 0910

Implementation of a Real-Time Beamforming System on Field Programmable Gate Array

A Major Qualifying Project Report

Submitted to the Faculty of
Worcester Polytechnic Institute

in partial fulfillment of the requirements for
Bachelor of Science Degree

By:

David Truong

Soe San Win

Date: April 26, 2010

Approved:

Professor Xinming Huang, Major Advisor

`

Abstract:

Beamforming is an important technique in array signal processing and wireless communication systems. In this project, we investigate the Minimum Variance Distortionless Response (MVDR) beamforming technique and its implementation. The QR-RLS algorithm is chosen because of its advantages of numerical stability and systolic array architecture. The team successfully implemented the real-time beamforming of a linear array with 3 receiving antennas on a Xilinx Virtex-5 FPGA platform. Both the simulation and hardware implementation results are presented in this report.

`

Acknowledgements:

Firstly, we would like to acknowledge Professor Xinming Huang of Electrical and Computer Engineering Department for his patience and guidance throughout the Major Qualifying Project. Secondly, we would like to thank Mr. Chen Shen, graduate student at Electrical and Computer Engineering Department for his various help throughout the project, and Mr. Yanjie Peng for helping us with the Simulink Model of the beamformer.

Last but not least, our thanks go to our friends in Electrical and Computer Engineering Department whose encouragements and cheers were invaluable throughout the course of the project and the senior year.

`

Table of Contents

`

`

`

Table of Figures

`

Executive Summary:

The project went about in three phases: theoretical background research, MATLAB simulation and implementation on the Xilinx System Generator Environment, and Xilinx EDK implementation of the architecture of the interface between a computer and the FPGA prototyping board of our choice, namely Virtex 5 LX110T [1].

On the first phase, we learned the theoretical background of general derivation of adaptive beamforming algorithms, the variant adaptive beamforming algorithm of Minimum Variance Distortionless Response, and the efficient implementation of the MVDR algorithm on the hardware platform. As for the general adaptive beamforming algorithms, we learned the Recursive Least Squares adaptive signal processing algorithm, the Kalman Filtering Theory, and the way the special case of Kalman Filtering theory merges with the Recursive Least Squares algorithm to become the general QR decomposition adaptive beamforming algorithm. Finally, we studied the emergence of the MVDR beamforming algorithm from the QR decomposition adaptive beamforming algorithm.

On the second phase, we simulated the algorithm on MATLAB, and move on to designing the MATLAB on the Xilinx System Generator Environment. The MATLAB simulation part went well after we put together the pieces of efficient implementation techniques, i.e., the systolic array approach, from the theoretical MVDR algorithm. We wrote a program on MATLAB that will simulate the MVDR beamforming for three antennas case. As for the implementation of the MATLAB model on the Xilinx System Generator environment, we were required to research on efficient hardware implementations of computationally expensive mathematical operations involved in the MVDR algorithm. We used the CORDIC algorithm for

`

calculating the square-roots, and then Newton-Raphson method to calculate the divisions. Finally, we utilized the principle of code reuse as we designed the individual cells required for the systolic array approach, and the integrating cell units together as the systolic array in the end.

On the third phase, using the Xilinx Embedded Development Kit (EDK) we designed the architecture for our PC to FPGA interface. We designated the necessary interfaces for our design and built our software component around those interfaces. Using the development environment we developed an application that would receive large sets of floating point data via an Ethernet interface from a host PC. The received data would then be converted to fixed-point data format and stored within the Block RAM (BRAM), a type configurable memory module capable of storing varying amounts of data. Once the data was stored it was verified via two methods. One method required a standard output which would display what data was stored and where it was stored. The second method required using the Xilinx Microprocessor Debugger (XMD) which allows you to halt a running application and review what has been stored in various memory controllers of the Virtex 5 Development Board. Having used both methods we were able to verify the functionality of the software component of our design.

In conclusion, we were able to complete the hardware and software components necessary for the completion of our entire project. We were able to verify that the hardware component, the Systolic Array model generated via System Generator, worked flawlessly through a variety of tests that included individual block testing and test-bench waveform generations. We were also able to verify that the software component, the PC to FPGA interaction, worked correctly by reading the data stored in memory after having sent data transmissions from the PC to the FPGA via an Ethernet connection. Although we weren't able to integrate the two components into a single design, it was proven that the functionality of each

`

individual component worked as expected if not better. Therefore, if time wasn't a factor and we were more knowledgeable in both the theoretical concepts and the software used we would have been able to generate an integrated design. However, this shouldn't be a cause for distress. Through our hard work we were able to prove theoretically that the concept of MVDR Beamforming is absolutely applicable in an FPGA environment. With the current work that we have completed it sets a framework for future work in this area of study.

`

Chapter 1:
Introduction

Beamforming is a popular signal processing technique that has been widely used in applications such as radar, wireless communications, and biomedical ultrasounds. Its main functionality is to help adjust the directionality of the transducer array when transmitting and receiving signals without actually having to physically change the direction of the array sensors. In order to perform this procedure, beamforming uses several smaller non-directional sensor arrays or antennae to simulate a larger directional antenna. With the simulated directional sensor array, the receiving antenna is able take into account of outside interference created by other sources. Being able to take into consideration of outside interference allows the receiving antenna to adapt accordingly and thus allows it to adjust the sensor array to one particular direction. This in turn causes a drastic reduction in overall outside interference while also improving the overall signal reception, in short in improves the signal-to-noise ratio (SNR). This signal processing technique also allows the transmitting antenna to better focus its signal at a receiver, which unlike an omni-directional transmitter which transmits its signal in all directions, results in a more focused signal distribution that does not interfere with other signal sources while also improving the signal reception at the receiving antenna. Therefore, due to its popularity in a number of diverse and fascinating applications and its overall improvement as oppose to omni-directional antennae, we have chosen to take it upon ourselves to enhance our understanding and hopefully contribute knowledge to this area of study.

`

Since beamforming is computationally intensive, certain technologies must be utilized in order to obtain real-time results. These technologies are usually separated into two categories: software based and hardware based techniques. Beamformers which employ software processing, generally implemented on a digital signal processor, in order to obtain real-time results are flexible enough that they can be adapted to transmit and/or receive in various directions instantaneously. In contrast, beamformers which employ dedicated hardware processing, such as Application-Specific Integrated Circuits (ASIC), are physically programmed to statically transmit and/or receive in a single direction at any given time. Therefore, the overall goal for this project is to implement a real-time Minimum Variance Distortionless Response (MVDR) adaptive beamforming onto a Xilinx Virtex 5 Development Platform to test its applicability in field-programmable gate arrays (FPGAs). MVDR is an adaptive beamforming algorithm which minimizes the average output power while setting the steering or "target" direction to unity. As the MVDR algorithm continues to adapt the overall noise, which includes interference and white noise, is minimized, resulting in a maximized SNR output. The reason why we wish to implement an adaptive beamforming algorithm onto an FPGA is due to its flexibility, reduced cost, and improved performance and data rate, as oppose to software or hardware processing, because it employs both hardware and software processing techniques. Of course, due its popularity, implementation of a beamforming algorithm onto an FPGA platform has already been conducted by Chris Dick, a Chief DSP Architect from Xilinx [2]. However, because of some discrepancies and the lack of conclusive evidence to support their findings and results, we took it upon ourselves to conduct this study.

The approach that we took for this project referenced Chris Dick's experimental procedure [2]. First, we expanded our basic understanding of various mathematical techniques,

`

adaptive signal processing techniques, and most importantly adaptive beamforming algorithms. Second, after obtaining a firm grasp of adaptive beamforming and its key concepts, we simulated it in MATLAB, "a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation" [3], to verify our understanding and to utilize it as a means to confirm the validity of our end result. Having completed our MATLAB simulation, we moved on to building the bulk of our project in System Generator, a Xilinx tool which allows a user to design various digital signal processing (DSP) applications. Since simulation alone is not enough to validate our implementation, we needed a way to test our beamforming algorithm in real-time on actual hardware. In order to accomplish this task, we needed to interface the FPGA platform with a PC. This was accomplished by using Xilinx's Embedded Design Kit (EDK), a tool which enables the user to design an embedded processor system in a Xilinx FPGA development board. After having completed the hardware and software components, we integrated the two components onto the Virtex 5 development board to test our implementation. The results of our findings, our testing, the steps we took in order to reach our end result and more will be explained in later chapters with more extensive detail.

`

Chapter 2:
Background

This chapter discusses the theoretical background of beamforming algorithm. The organization of this chapter is as follows: First, it will present mathematical theory and techniques required for understanding of the algorithms presented. Second, it will briefly describe two classical adaptive signal processing algorithms, namely Recursive Least Square (RLS) Algorithm and Kalman Filtering Theorem, which are the theoretical roots of QR-RLS adaptive signal processing algorithm, from which classic beamforming algorithms emerged. Third, it will describe the general beamforming problem scenario. Finally, it will describe our focus beamforming algorithm, Minimum Variance Distortionless Response (MVDR) beamforming algorithm, which evolved directly from QR-RLS algorithm.

2.1 Mathematical Theory and Techniques

In this section, we will describe mathematical techniques which are heavily used in both the predecessor adaptive signal processing algorithms, and MVDR beamforming algorithm.

2.1.1 QR Decomposition

QR decomposition is a mathematical technique to decompose a matrix into its orthogonal and right triangular components. Its applications largely lie in solving Least Square Problems. In

`

"Matrix Computations," by Golub and Van Loan [4], the QR decomposition is described as follows:

Let m and n be any positive integer greater than 0. Given any m x n matrix **A** and m > n**,** the QR decomposition of **A** can be described as:

$$\boldsymbol{A} = \boldsymbol{Q}.\boldsymbol{R} \tag{2.1.1.1}$$

Equation: QR Decomposition

Q is a unitary **m** x **n** matrix, i.e., $\boldsymbol{Q}.\boldsymbol{Q}^{H} = I$, where I is identity matrix. **R** is m x n upper right triangular matrix. Equation 2.2.1.1 above can be written in partitioned form as follows:

$$\boldsymbol{A} = \boldsymbol{Q}.\boldsymbol{R} = \boldsymbol{Q}.\begin{bmatrix}\boldsymbol{R_1}\\ \boldsymbol{0}\end{bmatrix} = [\boldsymbol{Q_1} \quad \boldsymbol{Q_2}].\begin{bmatrix}\boldsymbol{R_1}\\ \boldsymbol{0}\end{bmatrix} = \boldsymbol{Q_1}.\boldsymbol{R_1} \tag{2.1.1.2}$$

Equation: QR Decomposition Partitioned Matrix Form

where $\boldsymbol{R_1}$ is n x n triangular matrix, where $\boldsymbol{Q_1}$ is m x n and $\boldsymbol{Q_2}$ is m x (m - n).

2.1.2 Givens Rotation

Givens Rotation is a mathematical technique to orthogonally transform a matrix by rotating it in respective plains. It is a powerful technique for zeroing out the selective elements of a matrix. Usually, Givens Rotation is a method of transformation for calculating QR decomposition. The matrix computation text [4] described Given Rotation as follows:

The Givens Rotation matrix can be defined as:

$$\boldsymbol{G}_{(i,j,\theta)} = \begin{bmatrix} 1 & \dots & 1 & \dots & 1 & \dots & 1 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 1 & \dots & \cos\theta & \dots & \sin\theta & \dots & 1 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 1 & \dots & -\sin\theta & \dots & \cos\theta & \dots & 1 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 1 & \dots & 1 & \dots & 1 & \dots & 1 \end{bmatrix} \tag{2.1.2.1}$$

Equation: Givens Rotation Matrix

`

Basically, $\boldsymbol{G}_{(i,j,\theta)}$ is an identity matrix with the entries at i^{th} and j^{th} rows and columns replaced by the values of Sine and Cosine of the angle we wish to rotate the i^{th} and j^{th} dimensions of a matrix **M** by. Therefore, the matrix multiplication $\boldsymbol{G}_{(i,j,\theta)}.\boldsymbol{M}$ represents a counter-clockwise rotation of the matrix **M** in (i, j) dimensions by θ radians. We can generalize this multiplication with following equation:

$$M_k = \begin{matrix} \cos\theta.M_i - \sin\theta.M_j, if\ k = i \\ \cos\theta.M_j + \sin\theta.M_i, if\ k = j \\ M_k, if\ k\ !=i,j \end{matrix} \quad (2.1.2.2)$$

Equation: Givens Rotation Element Update Calculations

From above equation, we can see the insight that for clockwise rotation of matrix **M**, we just need to negate the Sine terms in the Givens Rotation matrix. Furthermore, we can see the insight that if the angle θ is equal to the angle between the two elements in (i, j) plane, the term M_k where k = i will be zeroed out. This property is used to apply Givens Rotation to zero out the desired elements in any full rank matrix **M**. Consequentially, to zero out the elements of choice in matrix **M**, we just need to move around the Cosine and Sine terms in the Givens Rotation matrix while keeping these terms in the form of a square. For an example 2 x 2 full rank matrix below, we can calculate desired Sine and Cosine terms using standard trigonometric properties as follows:

`

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} . \begin{bmatrix} a & c \\ b & d \end{bmatrix} = \begin{bmatrix} a_{updated} & b_{updated} \\ 0 & c_{updated} \end{bmatrix}$$

$$r = \sqrt{a^2 + b^2}$$

$$\cos\theta = \frac{a}{r} \qquad (2.1.2.3)$$

$$\sin\theta = \frac{b}{r}$$

Equation: Givens Rotation Example

With this property, Givens Rotation can be used to calculate the QR decomposition of matrix **M** by multiplying the matrix with a series of Givens Rotation matrices each of which will annihilate each element in lower or upper triangular portion of **M**, i.e., rotating counter-clockwise or clockwise, resulting in desired upper or lower triangular component of **M**. In this sense, the multiplication of the series of Givens Rotation matrices is the orthogonal matrix in QR decomposition. This can be summarizes as follows:

$$\boldsymbol{M} = \boldsymbol{Q}.\boldsymbol{R}, where$$

$$\boldsymbol{Q} = \boldsymbol{G}_1.\boldsymbol{G}_2.\boldsymbol{G}_3 \ldots\ldots \boldsymbol{G}_i \qquad (2.1.2.4)$$

$$i = number\ of\ elements\ in\ lower\ or\ upper\ triangular\ matrix$$

Equation: QR Decomposition with Givens Rotation

However, all of above Givens Rotation is only correct for real-valued matrices. For complex-valued matrices, we need some modifications in calculating Cosine and Sine values. In the paper "On Computing Givens Rotations Reliably and Efficiently" [5], the authors described the algorithm to calculate the Givens Rotation parameters of complex-valued matrices as follows:

`

Algorithm 2.1.2.1: Complex Givens Rotation Parameter Calculation

$Complex - valued\ matrix = \begin{bmatrix} f \\ g \end{bmatrix}$

```
if g == 0 (also f == g == 0)
     Cosine θ = 1;
     Sine θ = 0;
     r = f;
else if f == 0 (g != 0)
     Cosine θ = 0;
     Sine θ = sign (g');
     r = f;
else (f != 0 and g != 0)
     Cosine θ = |f| / sqrt(|f|^2 + |g|^2);
     Sine θ = sign(f).g' / sqrt(|f|^2 + |g|^2);
     r = sign (f). sqrt(|f|^2 + |g|^2);
end if
```

With this complex parameter calculation variant, Givens Rotation can now be applied to matrix problems involving complex-values, which will become apparently useful later in this report.

2.1.3 Matrix Lemmas

Matrix Lemmas are mathematical properties of matrices used in deriving the theorems that are the root of the beamforming problem. These lemmas are derived by Dr. Simon Haykin in his reputable text book "Adaptive Filter Theory" [6].

The first lemma is Matrix Inversion Lemma. The Lemma states that for positive-definite I-by-I matrices **A**, **B**, positive-definite I-by-J matrix **C** and positive-definite J-by-I matrix **D** such that:

$$\boldsymbol{A} = \boldsymbol{B}^{-1} + \boldsymbol{C}\boldsymbol{D}^{-1}\boldsymbol{C}^{H} \tag{2.1.3.1}$$

Equation: Matrix Inversion Lemma Equation 1

`

Then we can write the inverse of matrix **A** as:

$$\boldsymbol{A}^{-1} = \boldsymbol{B} - \boldsymbol{BC}(\boldsymbol{D} + \boldsymbol{C}^{H}\boldsymbol{BC})^{-1}\boldsymbol{C}^{H}\boldsymbol{B} \quad (2.1.3.2)$$

Equation: Matrix Inversion Lemma Equation 2

This lemma can be proved by multiplying Equation 2.1.3.1 with Equation 2.1.3.2, which yields an identity matrix I.

The second lemma is matrix factorization lemma stated in [6], which states that for any J-by-I matrices **A** and **B**, there exists a relationship:

$$\boldsymbol{A}\boldsymbol{A}^{H} = \boldsymbol{B}\boldsymbol{B}^{H}$$

$$if\ and\ only\ if, \boldsymbol{B} = \boldsymbol{A}\boldsymbol{\Theta} \quad (2.1.3.2)$$

$$where, \boldsymbol{\Theta}\boldsymbol{\Theta}^{\boldsymbol{H}} = \mathbf{I}$$

Equation: Matrix Factorization Lemma

Equation 2.1.3.2 proves the existence of orthogonal matrix **Θ** between any two J-by-I matrices where J is less than or equal to I.

2.2 Adaptive Signal Processing Algorithms

Filters employing adaptive signal processing algorithms self-adjust themselves to change the transfer function according to the optimization constraint of the algorithm. Thus, as opposed to normal filters, whose filter coefficients and transfer function behavior are fixed, adaptive filters changes their behaviors in accordance with changing environment to meet the functional requirements. There are several adaptive signal processing algorithms, as described in [6]. However, we will be describing two most important algorithms that became foundation of adaptive beamforming algorithms.

`

2.2.1 Recursive Least Square (RLS) Algorithm

Recursive Least Square algorithm is an adaptive filtering algorithm employing method of least squares, which can estimate the filter coefficient at the current time step given only the least square estimated filter coefficient of previous time step. With its recursive nature, the algorithm is hence named Recursive Least Square (RLS) algorithm [6] described the algorithm in following steps:

The tap-input vector at time step i is defined as:

$$\boldsymbol{u}(i) = [u(i) \quad u(i-1) \quad \dots \quad \dots \quad u(i-M+1)]^T \tag{2.2.1.1}$$

Equation: Tap-input Vector

The tap-weight (coefficient) vector at time step n is defined as:

$$\boldsymbol{w}(\boldsymbol{n}) = [w_0(n) \quad w_1(n) \quad \dots \quad \dots \quad w_{M-1}(n)]^T \tag{2.2.1.2}$$

Equation: Tap-weight Vector

The cost function for the RLS algorithm is defined as:

$$\text{б}\,(n) = \sum_{i=1}^{n} \lambda^{n-i} |e(i)|^2$$
$$where, e(i) = d(i) - \boldsymbol{w^H}(n)\boldsymbol{u}(i) \tag{2.2.1.3}$$

Equation: RLS Cost Function

The optimum value for $\boldsymbol{w^H}$ which the cost function б is minimized can be defined in normal equation form as follows:

$$\boldsymbol{\Phi}(n)\boldsymbol{w}(n) = \boldsymbol{z}(n)$$
$$where, \boldsymbol{\Phi}(n) = \sum_{i=1}^{n} \lambda^{n-i} \boldsymbol{u}(i)\boldsymbol{u}(i)^H \tag{2.2.1.4}$$

`

$$and, \boldsymbol{z}(n) = \sum_{i=1}^{n} \lambda^{n-i} \boldsymbol{u}(i) \boldsymbol{d}^{*}(i)$$

Equation: RLS Optimization Constraints

We can see that the last two equations can be re-written in recursive form as follows:

$$\begin{aligned} \boldsymbol{\Phi}(n) &= \lambda \boldsymbol{\Phi}(n-1) + \boldsymbol{u}(n)\boldsymbol{u}(n)^{H} \\ \boldsymbol{z}(n) &= \lambda \boldsymbol{z}(n-1) + \boldsymbol{u}(n)\boldsymbol{d}^{*}(n) \end{aligned} \tag{2.2.1.5}$$

Equation: RLS Optimization Constraints in Recursive Form

For the next step, we will use the Matrix Inversion Lemma presented in Section 2.1.3 to calculate the inverse of matrix $\boldsymbol{\Phi}(n)$ by setting $\boldsymbol{A} = \boldsymbol{\Phi}^{-1}(n)$, $\boldsymbol{B}^{-1} = \lambda \boldsymbol{\Phi}(n-1)$, $\boldsymbol{C} = \boldsymbol{u}(n)$, $\boldsymbol{D} = 1$ in the first equation of Equation 2.2.1.5. Then, we get the expression:

$$\begin{aligned} \boldsymbol{P}(n) &= \lambda^{-1}\boldsymbol{P}(n-1) - \lambda^{-1}\boldsymbol{k}(n)\boldsymbol{u}(n)^{\boldsymbol{H}}\boldsymbol{P}(n-1) \\ where, \boldsymbol{P}(n) &= \boldsymbol{\Phi}^{-1}(n) \\ and, \boldsymbol{k}(n) &= \frac{\lambda^{-1}\boldsymbol{P}(n-1)\boldsymbol{u}(n)}{1 + \lambda^{-1}\boldsymbol{u}(n)^{\boldsymbol{H}}\boldsymbol{P}(n-1)\boldsymbol{u}(n)} = \boldsymbol{P}(n)\boldsymbol{u}(\boldsymbol{n}) = \boldsymbol{\Phi}^{-1}(n)\boldsymbol{u}(\boldsymbol{n}) \end{aligned} \tag{2.2.1.6}$$

Equation: RLS Parameters

With the parameters, we can now calculate the expression for tap-weight (coefficient) vector from our constraint equation mentioned in Equation 2.2.1.4. By using the derived expressions for RLS parameters, we can derive the expression for the tap-weight vector as follows:

`

$$\boldsymbol{w}(n) = \boldsymbol{\Phi}^{-1}(n)\boldsymbol{z}(n)$$

$$\boldsymbol{w}(n) = \boldsymbol{P}(n)\boldsymbol{z}(n)$$

$$\boldsymbol{w}(n) = \lambda\boldsymbol{P}(n)\boldsymbol{z}(n-1) + \boldsymbol{P}(n)\boldsymbol{u}(n)\boldsymbol{d}^{*}(n)$$

$$\boldsymbol{w}(n) = \boldsymbol{P}(n-1)\boldsymbol{z}(n-1) - \boldsymbol{k}(n)\boldsymbol{u}(n)^{\boldsymbol{H}}\boldsymbol{P}(n-1)\boldsymbol{z}(n-1) + \boldsymbol{P}(n)\boldsymbol{u}(n)\boldsymbol{d}^{*}(n) \quad (2.2.1.7)$$

$$\boldsymbol{w}(n) = \boldsymbol{w}(n-1) - \boldsymbol{k}(n)\boldsymbol{u}(n)^{\boldsymbol{H}}\boldsymbol{w}(n-1) + \boldsymbol{P}(n)\boldsymbol{u}(n)\boldsymbol{d}^{*}(n)$$

$$\boldsymbol{w}(n) = \boldsymbol{w}(n-1) + \boldsymbol{k}(n)\boldsymbol{\xi}^{*}(n)$$

$$where, \boldsymbol{\xi}^{*}(n) = d(n) - \boldsymbol{w}^{H}(n-1)\boldsymbol{u}(n)$$

Equation: RLS Tap-weight Vector and Priori Estimation Error

These derived parameters for RLS algorithm will found to be important in deriving the QR-RLS algorithm later.

2.2.2 Kalman Filtering Algorithm

Kalman filtering algorithm is another important adaptive signal processing algorithm. It also computes the results in the recursive manner, and its derivations were largely dependent on the state-space signal processing concepts. With its unique nature, Kalman filtering theory provided the framework spanning over the families of recursive least square adaptive signal processing algorithms.

Here in this project, we are particularly interested in one variant of Kalman filter called Unforced Dynamic Model because it is directly related to the emergence of QR-RLS algorithm. We will omit the derivation of traditional Kalman filters in this report for the sake of brevity, and will focus mainly on the Forced Dynamic Model variant. For detailed derivation of traditional Kalman filter, it can be referred to [6].

`

Traditional Kalman filter has following parameters:

$$\begin{aligned} \boldsymbol{K}(n) &= \boldsymbol{K}(n, n-1) - \boldsymbol{F}(n, n+1)\boldsymbol{G}(n)\boldsymbol{C}(n)\boldsymbol{K}(n, n-1) \\ \boldsymbol{G}(n) &= \boldsymbol{F}(n+1, n)\boldsymbol{K}(n, n-1)\boldsymbol{C}^{H}(n)[\boldsymbol{C}(n)\boldsymbol{K}(n, n-1)\boldsymbol{C}^{H}(n) + \boldsymbol{Q_2}(n)]^{-1} \\ \boldsymbol{\alpha}(n) &= \boldsymbol{y}(n) - \boldsymbol{C}(n)\boldsymbol{x}(n|y_{n-1}) \end{aligned} \tag{2.2.2.1}$$

$$where, \boldsymbol{C}(n)\ is\ observation\ matrix$$

$$and, \boldsymbol{F}(n, n+1)\ is\ state - transition\ matrix$$

Equation: Kalman filter Parameters

For the Unforced Dynamic Model variant, the process noise becomes zero, and the measuring noise becomes zero-mean white noise process with unit variance. Thus, some of the state-space parameters converge to constants and the parameters for the Kalman filter variant is derived as follows:

$$\begin{aligned} \boldsymbol{F}(n+1, n) &= \boldsymbol{\lambda^{\frac{1}{2}}\mathbf{I}}, \qquad \boldsymbol{C}(n) = \boldsymbol{u}^{H}(n), \qquad \boldsymbol{Q_2}(n) = 1 \\ \boldsymbol{K}(n) &= \lambda^{-1}\boldsymbol{K}(n-1) - \lambda^{-1/2}\boldsymbol{g}(n)\boldsymbol{u}(n)^{\boldsymbol{H}}\boldsymbol{K}(n-1) \\ \boldsymbol{g}(n) &= \frac{\lambda^{-1/2}\boldsymbol{K}(n-1)\boldsymbol{u}(n)}{1 + \boldsymbol{u}(n)^{\boldsymbol{H}}\boldsymbol{K}(n-1)\boldsymbol{u}(n)} \\ \boldsymbol{\alpha}(n) &= y(n) - \boldsymbol{u}^{H}(n)\boldsymbol{x}(n|y_{n-1}) \end{aligned} \tag{2.2.2.2}$$

Equation: Unforced Dynamic Kalman filter Parameters

By manipulating the expressions obtained in Equation 2.2.2.2, we can finally obtain the parameters for Kalman Square-root Information filter parameters as follows:

`

$$\boldsymbol{g}(n) = \lambda^{-1/2}\boldsymbol{K^{-1}}(n)\boldsymbol{u}(\boldsymbol{n})$$

$$\boldsymbol{K}^{-1}(n) = \lambda\boldsymbol{K}^{-1}(n-1) + \boldsymbol{u}(n)\boldsymbol{u}(n)^{H} \qquad (2.2.2.3)$$

$$\boldsymbol{K}^{-1}(n)\,\boldsymbol{x}(n+1|y_n) = \lambda^{-1/2}\boldsymbol{K}^{-1}(n-1)\boldsymbol{x}(n|y_{n-1}) + \lambda^{-1/2}\,\boldsymbol{u}(n)y(n)$$

Equation: Kalman Information Filter Parameters

Then, by re-writing the term $\boldsymbol{K}^{-1}(n) = \boldsymbol{K}^{-H/2}(n)\boldsymbol{K}^{-1/2}(n)$, and utilizing the matrix factorization lemma presented in Section 2.1.3 as well as, we can formulate the Kalman Square-root Information filter into pre-array and post-array forms as follows:

$$\begin{bmatrix} \lambda^{1/2}\mathbf{K}^{-\mathrm{H}/2}(\mathrm{n}-1) & \lambda^{1/2}\boldsymbol{u}(n) \\ \boldsymbol{x}^{H}(n|y_{n-1})\mathbf{K}^{-\mathrm{H}/2}(\mathrm{n}-1) & y^{*}(n) \\ \mathbf{0}^{T} & 1 \end{bmatrix} \Theta\,(\mathrm{n}) = \begin{bmatrix} \mathbf{K}^{-\mathrm{H}/2}(\mathrm{n}) & \mathbf{0} \\ \boldsymbol{x}^{H}(n+1|y_n)\mathbf{K}^{-\mathrm{H}/2}(\mathrm{n}) & \alpha^{*}(n)r^{-1/2}(n) \\ \lambda^{1/2}\boldsymbol{u}^{H}(n)\,\mathbf{K}^{1/2}(\mathrm{n}) & r^{1/2}(n) \end{bmatrix} \qquad (2.2.2.4)$$

Equation: Kalman Square-root Information Filter Calculation

We can use the Equation 2.2.2.4 to recursively calculate the Kalman Filter coefficients.

2.2.3 QR-RLS Algorithm

From Equation 2.2.1.6 and Equation 2.2.2.3, we can see the correspondence between the RLS algorithm, and Kalman Information Filter. By re-writing the term $\boldsymbol{\Phi}(n) = \boldsymbol{\Phi^{1/2}}(n)\boldsymbol{\Phi^{H/2}}(n)$, and inventing the new term $\mathbf{p}\,(\mathrm{n}) = \boldsymbol{\Phi}^{\frac{H}{2}}(n)\mathbf{w}(\mathrm{n}) = \boldsymbol{\Phi}^{-\frac{1}{2}}(n)\mathbf{z}(\mathrm{n})$, we can represent the RLS algorithm in the pre-array and post-array from as we described the Kalman Information filtering algorithm earlier.

`

$$\begin{bmatrix} \lambda^{\frac{1}{2}}\boldsymbol{\Phi}^{\frac{1}{2}}(\mathrm{n}-1) & \boldsymbol{u}(n) \\ \lambda^{\frac{1}{2}}\mathbf{p}^{\mathrm{H}}(\mathrm{n}-1) & d(n) \\ \mathbf{0}^{T} & 1 \end{bmatrix} \Theta\,(\mathrm{n}) = \begin{bmatrix} \boldsymbol{\Phi}^{\frac{1}{2}}(\mathrm{n}) & \mathbf{0} \\ \mathbf{p}^{\mathrm{H}}(\mathrm{n}) & \xi(n)\gamma^{-\frac{1}{2}}(n) \\ \boldsymbol{u}^{H}\,(n)\,\boldsymbol{\Phi}^{-\frac{\mathrm{H}}{2}}(\mathrm{n}) & \gamma^{\frac{1}{2}}(n) \end{bmatrix}$$

$$\boldsymbol{w^{H}}(n) = \mathbf{p}^{\mathrm{H}}(\mathrm{n})\boldsymbol{\Phi}^{-\frac{1}{2}}(\mathrm{n})$$

Equation 2.2.3.1: QR-RLS Algorithm

The QR-RLS algorithm is used to solve the adaptive beamforming problem efficiently due to its recursive nature, and its efficient computational structure and numerical stability [6].

2.3 Adaptive Beamforming

The adaptive beamforming problem is where the adaptive signal processing applications meet the spatial signal processing structures such as antenna arrays. By utilizing the adaptive signal processing algorithms such as QR-RLS algorithm while sampling the received signals with spatial antenna arrays, the technique ensures to keep the gain in the desired direction high while suppressing the received signals from interference sources. With the adaptive nature of the algorithm, the system's transfer function will change according to the desired signal direction to get rid of the interference and ensure the desired signal.

There are many variants of adaptive beamforming in general. However, for the purpose of the project, we will be looking at the receive beamforming utilizing the QR-RLS algorithm.

`

2.4 Minimum Variance Distortionless Response

The Minimum Variance Distortionless Response (MVDR) Problem can be formulated as:

$$\min_{w(n)} \sum_{i=1}^{n} \lambda^{n-i} |e(i)|^2 \tag{2.4.1}$$

where $e(i) = \boldsymbol{w}^H(n)\boldsymbol{u}(i)$ *such that* $\boldsymbol{w}^H(n)\boldsymbol{s}(\theta_0) = 1$

Equation: Minimum Variance Distortionless Response Problem Statement

With the introduction of the constraint $\boldsymbol{w}^H(n)\boldsymbol{s}(\theta_0) = 1$, where $\boldsymbol{s}(\theta_0)$ is the steering vector for the antenna array, the optimization constraint becomes:

$$\boldsymbol{w}(n) = \frac{\boldsymbol{\Phi}^{-1}(\mathrm{n})\boldsymbol{s}(\theta_0)}{\boldsymbol{s}^H(\theta_0)\boldsymbol{\Phi}^{-1}(\mathrm{n})\boldsymbol{s}(\theta_0)} \tag{2.4.2}$$

Equation: MVDR Optimization Constraint

By defining auxiliary vector as: $\boldsymbol{a}(n) = \boldsymbol{\Phi}^{-1/2}(\mathrm{n})\boldsymbol{s}(\theta_0)$, we can re-write the weight vector, estimation error, and new estimation error term as follows:

$$\boldsymbol{w}(n) = \frac{\boldsymbol{\Phi}^{-\mathrm{H}/2}(\mathrm{n})\boldsymbol{a}(n)}{||\boldsymbol{a}(n)||^2}$$

$$\boldsymbol{e}(n) = \frac{\boldsymbol{a}^{\boldsymbol{H}}(n)\boldsymbol{\Phi}^{-\frac{1}{2}}(\mathrm{n})\mathbf{u}(\mathrm{n})}{||\boldsymbol{a}(n)||^2} \tag{2.4.3}$$

$$\boldsymbol{e}'(n) = \boldsymbol{a}^{\boldsymbol{H}}(n)\boldsymbol{\Phi}^{-\frac{1}{2}}(\mathrm{n})\mathbf{u}(\mathrm{n})$$

Equation: MVDR Parameters

With the parameters of the MVDR problem defined, we can finally map MVDR problem to QR-RLS and represent it in the pre-array and post-array as follows:

`

$$\begin{bmatrix} \lambda^{\frac{1}{2}}\boldsymbol{\Phi}^{\frac{1}{2}}(\mathrm{n}-1) & \boldsymbol{u}(n) \\ \lambda^{\frac{1}{2}}\mathbf{a}^{\mathrm{H}}(\mathrm{n}-1) & 0 \\ \mathbf{0}^{T} & 1 \end{bmatrix} \Theta\,(\mathrm{n}) = \begin{bmatrix} \boldsymbol{\Phi}^{\frac{1}{2}}(\mathrm{n}) & \mathbf{0} \\ \mathbf{a}^{\mathrm{H}}(\mathrm{n}) & -e'(n)\gamma^{-\frac{1}{2}}(n) \\ \boldsymbol{u}^{H}\,(n)\,\boldsymbol{\Phi}^{-\frac{\mathrm{H}}{2}}(\mathrm{n}) & \gamma^{\frac{1}{2}}(n) \end{bmatrix} \tag{2.4.4}$$

Equation: MVDR Calculation

After calculating the parameters via Equation 2.4.4, we can calculate the estimation error beam output by calculating as follows:

$$e(n) = \frac{-(-e'(n)\gamma^{-\frac{1}{2}}(n)\gamma^{\frac{1}{2}}(n))}{\|\mathbf{a}(n)\|^2} \tag{2.4.5}$$

Equation: MVDR Estimation Error Calculation

2.5 Chapter Summary

We have successfully presented comprehensive and thorough material on the theoretical background of MVDR beamforming algorithm. The next chapter will present the implementation considerations for the MVDR algorithm on hardware, and the MATLAB simulation for the algorithm.

`

Chapter 3:
MATLAB Simulation of Beamforming

This chapter discusses the MATLAB simulation of the Minimum Variance Distortionless Response (MVDR) beamforming algorithm presented in Chapter 2. The chapter is organized as follows: Firstly, it will present the efficient implementation technique of the MVDR beamforming algorithm on hardware. Secondly, it will present the method used to simulate cycle-by-cycle hardware simulation on MATrix LABoratory (MATLAB) [7] environment. Thirdly, it will present the structural approach of the simulation program. Finally, it will present the simulation results.

3.1 Efficient Implementation

This section will present the efficient implementation technique for matrix computation-based signal processing algorithms, namely Systolic Array Processor approach. This approach has been widely used in Very Large Scale Integrated Circuit (VLSI) implementation of signal processing algorithms.

3.1.1 Systolic Array Processor

As we can see from Chapter 2, the calculation of QR-RLS algorithm and its predecessor MVDR algorithm involves matrix multiplication which is a series of Givens Rotations. It is known that the serial implementations of matrix multiplication are usually inefficient and slow. Especially, for the real-time signal processing algorithms such as QR-RLS and MVDR

`

beamforming algorithms employing matrix computations. The situation calls for a more efficient method to reduce the computation time, and improve the throughput in matrix computations.

In 1978, systolic array processors became proposed for VLSI signal processing systems by Kung and Leiserson in [8]. Systolic Array processor approach is a parallelized approach to many of matrix computations. In a systolic array system, there are individual processing cells arranged as a particular structure. Each individual cell of the system has its own processing functionality, and own local memory. Moreover, only adjacent cells are connected to each other, and there is no direct connection among the cells that are not adjacent.

In this way, when data is fed into a systolic system, the processing cells at the front-end of the system will process the data, store the required data in their own local memory, and then forward resulting data to their adjacent cells in the system. The cells that received the forwarded data from the front-end cells will, in turn, process the data, store the required data, and forward their results to their adjacent cells. This processing and forwarding pattern is continued until the data flow reaches to the end of the system where the desired results are presented. In this way, the data processing flows through the whole system in a rhythmic manner, much like the blood pumping fashion of human heart. The system is thus named systolic array processor.

3.1.2 Systolic Array Approach for MVDR

In the MVDR algorithm presented in Equation 2.4.4, we can see that the resulting post-array is a lower triangular matrix, and the matrix multiplication on the left-hand side is basically applying a series of givens rotations to the pre-array to annihilate the input vector **u** (n). From our derivations from Chapter 2, we can see that the number of elements in the input vector **u** (n) directly corresponds to the number of antennas in the MVDR beamforming scenario. To annihilate the elements of the input vector and set it to zero, we will need to apply a series of

`

Givens Rotations operations. The number of operations required is the same as the number of elements in the input because each Givens Rotation will annihilate exactly one entry in the input vector. Therefore, in general, MVDR beamforming algorithm with K input antennas will require K Givens Rotation operations in calculating the post-array.

Fortunately, these Givens Rotation operations are highly parallelizable in computation because there is no data dependency between the Givens Rotation operation at one entry in the input vector and the Givens Rotation operation at the same entry for subsequent iterations. In short, the calculation of the Givens Rotation for a particular entry in the input vector does not need to wait for the results at other entries. Therefore, we can parallelize the Givens Rotation operations.

If we examine the Givens Rotation presented in Section 2.1.2, we can see that calculation of Givens Rotation involves two major steps. The first step is to calculate the rotation parameter Cosine and Sine values. The second step is to actually rotating the values by performing the vector dot product. However, notice that for each sequence of Givens Rotation to cancel out a corresponding entry in the pre-array, both the rotation parameter calculation phase and the rotation phase involves with only certain elements in the pre-array, and the remaining elements in the pre-array are unaffected. To make this point clearer, the step by step operation of one step of MVDR algorithm calculation for three inputs is presented below.

`

$$\begin{bmatrix} \Phi_{1,1} & 0 & 0 & u_1 \\ \Phi_{2,1} & \Phi_{2,2} & 0 & u_2 \\ \Phi_{3,1} & \Phi_{3,2} & \Phi_{3,3} & u_3 \\ a_1 & a_2 & a_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} cos & 0 & 0 & -sin' \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ sin & 0 & 0 & cos \end{bmatrix}$$

$$\Phi'_{1,1} = (\Phi_{1,1}.cos) + (u_1.sin)$$

$$u'_1 = (\Phi_{1,1}.(-sin^*)) + (u_1.cos) = 0$$

$$\Phi'_{2,1} = (\Phi_{2,1}.cos) + (u_2.sin)$$

$$u'_2 = (\Phi_{2,1}.(-sin^*)) + (u_2.cos)$$

$$\Phi'_{2,2} = \Phi_{2,2}$$

(3.1.2.1)

$$\Phi'_{3,1} = (\Phi_{3,1}.cos) + (u_3.sin)$$

$$u'_3 = (\Phi_{3,1}.(-sin^*)) + (u_3.cos)$$

$$\Phi'_{3,2} = \Phi_{3,2}$$

$$\Phi'_{3,3} = \Phi_{3,3}$$

$$a'_1 = a_1.cos$$

$$a'_2 = a_2$$

$$a'_3 = a_3$$

$$\beta' = a_1.(-sin^*)$$

Equation: First Iteration of the Step

`

$$\begin{bmatrix} \Phi'_{1,1} & 0 & 0 & 0 \\ \Phi'_{2,1} & \Phi'_{2,2} & 0 & u'_2 \\ \Phi'_{3,1} & \Phi'_{3,2} & \Phi'_{3,3} & u'_3 \\ a'_1 & a'_2 & a'_3 & \beta' \\ \beta' & \beta' & \beta' & \beta' \end{bmatrix} . \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos & 0 & -sin' \\ 0 & 0 & 1 & 0 \\ 0 & sin & 0 & cos \end{bmatrix}$$

$$\Phi''_{1,1} = \Phi'_{1,1}$$

$$u''_1 = 0$$

$$\Phi''_{2,1} = \Phi'_{2,1}$$

$$\Phi''_{2,2} = \left(\Phi'_{2,2}.cos\right) + (u'_2.sin)$$

$$u''_2 = \left(\Phi'_{2,2}.(-sin')\right) + (u'_2.cos) = 0 \qquad (3.1.2.2)$$

$$\Phi''_{3,1} = \Phi'_{3,1}$$

$$\Phi''_{3,2} = \left(\Phi'_{3,2}.cos\right) + (u'_3.sin)$$

$$u''_3 = \left(\Phi'_{3,2}.(-sin^*)\right) + (u'_3.cos)$$

$$\Phi''_{3,3} = \Phi'_{3,3}$$

$$a''_1 = a'_1$$

$$a''_2 = (a_2.cos) + (\beta'.sin)$$

$$a''_3 = a'_3$$

$$\beta'' = ((a'_2.(-sin^*)) + (\beta'.cos))$$

Equation: Second Iteration of the Step

`

$$\begin{bmatrix} \Phi''_{1,1} & 0 & 0 & 0 \\ \Phi''_{2,1} & \Phi''_{2,2} & 0 & 0 \\ \Phi''_{3,1} & \Phi''_{3,2} & \Phi''_{3,3} & u''_3 \\ a''_1 & a''_2 & a''_3 & \beta'' \\ \beta'' & \beta'' & \beta'' & \beta'' \end{bmatrix} . \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & cos & -sin' \\ 0 & 0 & sin & cos \end{bmatrix}$$

$$\Phi''' _{1,1} = \Phi'' _{1,1}$$

$$u'''_1 = 0$$

$$\Phi''' _{2,1} = \Phi'' _{2,1}$$

$$u'''_2 = 0$$

$$\Phi''' _{2,2} = \Phi'' _{2,2}$$

$$\Phi'''_{3,1} = \Phi''_{3,1}$$

$$\Phi'''_{3,2} = \Phi''_{3,2}$$

$$\Phi'''_{3,3} = \left(\Phi'' _{3,3} . cos\right) + (u''_3 . sin)$$

$$u'''_3 = \left(\Phi'' _{3,3} . (-sin^*)\right) + (u''_3 . cos)$$

$$a'''_1 = a''_1$$

$$a'''_2 = a''_2$$

$$a''' _3 = (a'' _3 . cos) + (\beta'' . sin)$$

$$\beta''' = ((a'' _3 . (-sin^*)) + (\beta'' . cos))$$

(3.1.2.3)

Equation: Third Iteration of the Step

$$\begin{bmatrix} \Phi'''_{1,1} & 0 & 0 & 0 \\ \Phi'''_{2,1} & \Phi'''_{2,2} & 0 & 0 \\ \Phi'''_{3,1} & \Phi'''_{3,2} & \Phi'''_{3,3} & 0 \\ a'''_1 & a'''_2 & a'''_3 & ?''' \\ \beta''' & \beta''' & \beta''' & \beta''' \end{bmatrix}$$

(3.1.2.3)

Equation: Resulting Post-Array

`

From above calculations, we can see that the u-terms are required for calculating different update values for **Φ** and **a**. Thus, u-terms become the data being forwarded to the adjacent cells in a systolic system. On the other hand, the update calculations of **Φ** and **a** only needs **Φ** and **a** from previous Givens Rotation operation. Therefore, they become the locally stored values in the processing cells of a systolic system.

By putting them altogether, we got the systolic system implementation of MVDR algorithm as described in Equation 2.4.4 and Equation 2.4.5. This implementation follows the systolic implementation 2 as described in [6]. The system block diagram is as follows:

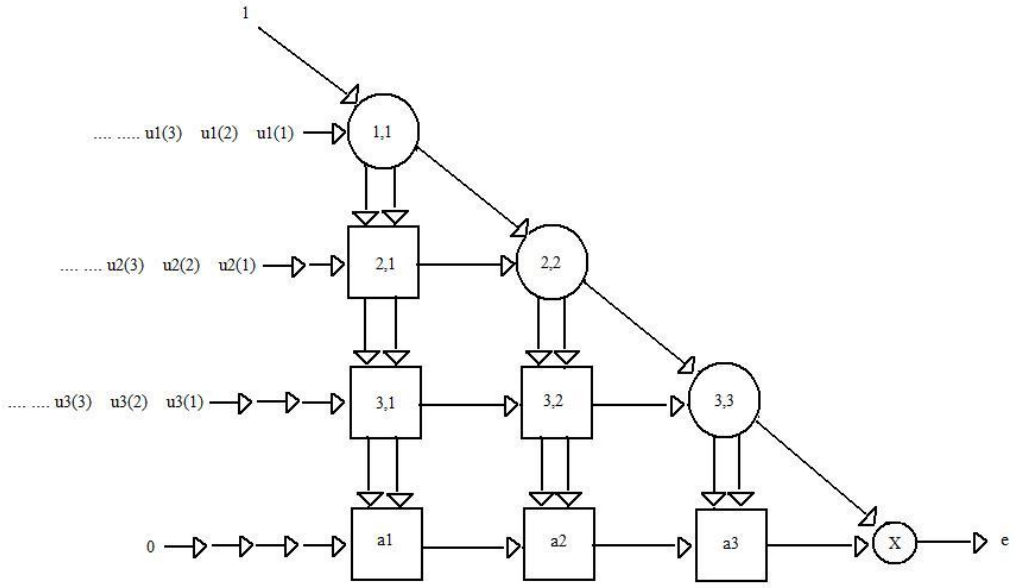

Figure 3.2.1 1: Systolic Implementation of MVDR Algorithm

The circle cells as found in the figure above will calculate the rotation parameters for Givens Rotations as described in Algorithm 2.1.2.1 with an additional computation of e (n) as:

$$e(n) = \frac{-(-e'(n)\gamma^{-\frac{1}{2}}(n)\gamma^{\frac{1}{2}}(n))}{\|\mathbf{a}(n)\|^2} \qquad (3.1.2.2)$$

Equation: Beam-output Error Calculation

`

in the right-most circle cell labeled as “x.” The square cells are responsible for calculating the rotation of the matrix elements. In standard terminology, the circle cells are called Boundary Cells. In other words, the Cosine and Sine terms of the Algorithm 2.1.2.1 are calculated in the Boundary Cells. On the other hand, the square cells are called Internal Cells. The Internal Cells calculate the rotation of the entries, i.e., the matrix multiplication part of Algorithm 2.1.2.1.

3.2 Hardware Behavioral Simulation on MATLAB

We used MATLAB to simulate the MVDR beamforming algorithm implementation in Systolic Array Implementation 2.

The challenges we found in writing a simulation script program in this manner on MATLAB was that all of the execution of the code on MATLAB are sequential while Systolic Array is parallelized algorithm of the MVDR algorithm. In general, we were required to find a way to make the sequential execution of MATLAB into parallel, independent execution of Boundary and Internal Cells on the Systolic System. In addition, we were required to meet the requirement of each of the cells in the system having its own local memory for storage of the values, i.e., the values of the elements of $\boldsymbol{\Phi}$ matrix and **a** matrix. Last but not least, we were also required to meet the cycle-by-cycle update of the values as if it were running on hardware.

To meet these requirements, we created the vector arrays of storage on MATLAB for the storage of each type of cells respectively. And then, we created the external separate functions for functionality of Boundary and Internal Cells respectively. To mimic the independent executions of the cells, we called each Boundary and Internal Cell functions sequentially, i.e., starting with the Boundary cell (1, 1) in Figure 3.1.2.1 and going down row by row ending in the “x” cell, by passing the local storage of each cell to the function calls. Finally, to make the

`

system act on cycle-by-cycle update behavior, we created the global loop iterating the function calls above for the desired number of iterations.

The flow control of the simulation program can be found as follows:

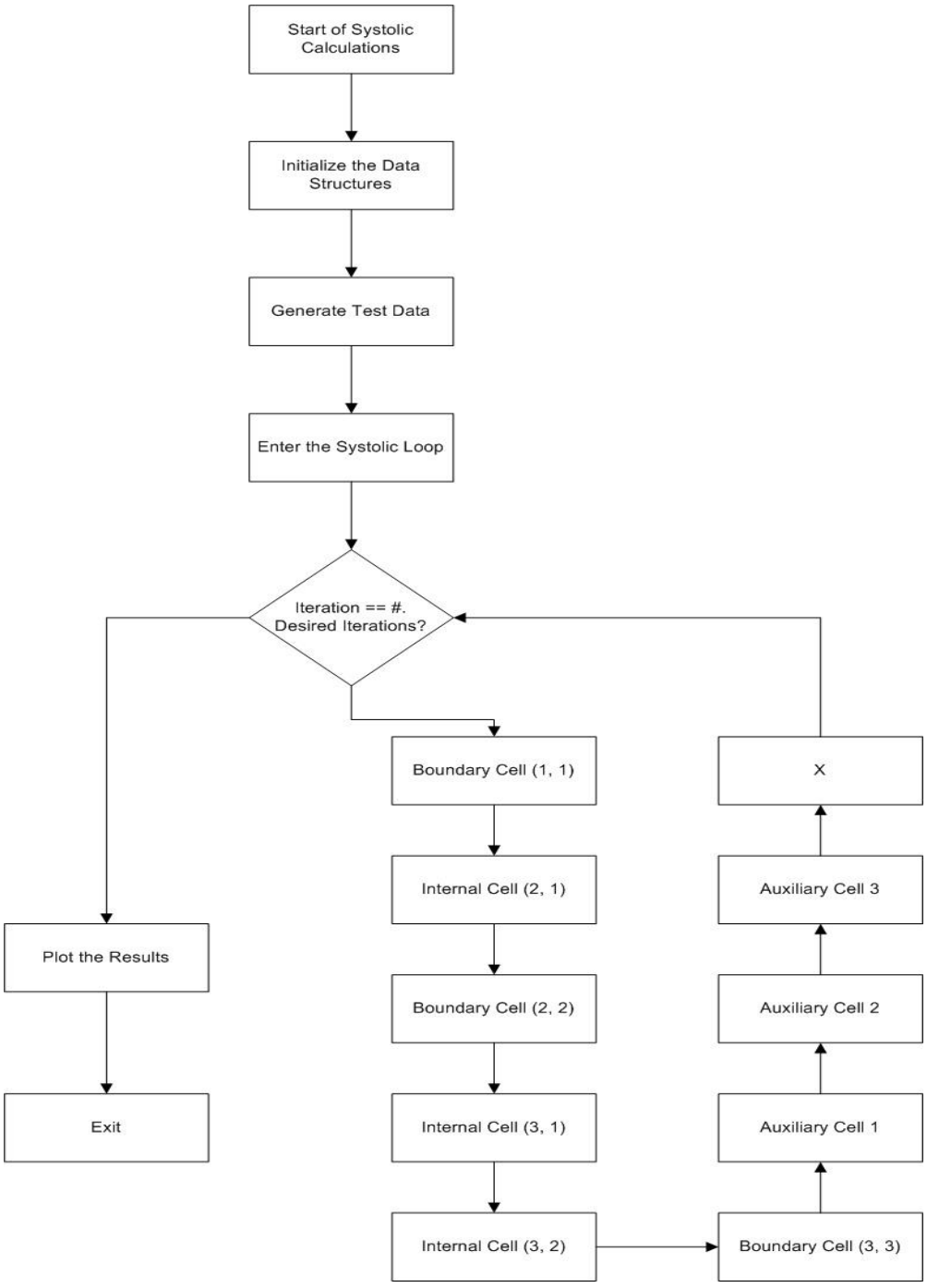

Figure 3.2.1 2: Flow Control for the Simulation Program

`

3.3 Scenarios for Simulation

The scenario for the simulation is as follows:

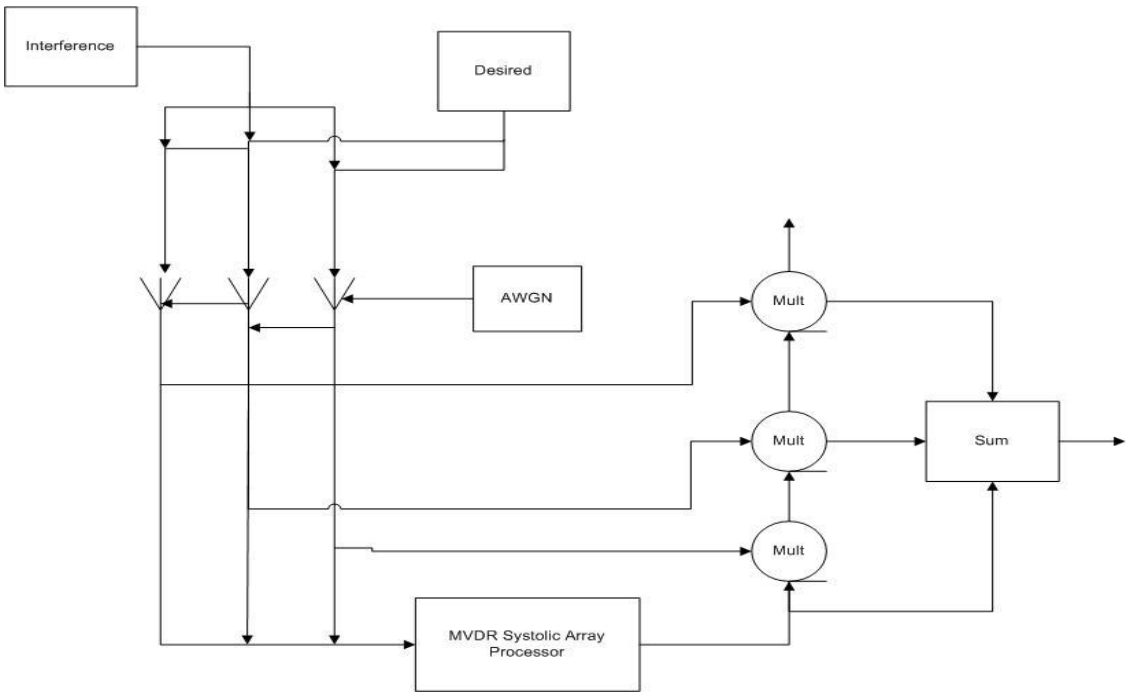

Figure 3.3 1: Scenario for the Simulation

We have three antennas, and the systolic array processor. We will have one desired signal source, and one interference signal source. Due to the channel, and the thermal heat in the equipment, there will be Additive White Gaussian Noise (AWGN) added at the receiving antenna. Therefore, the input values coming into the MVDR systolic array processor are mixtures of the desired signal, interference signal and the AWGN.

We keep the Signal-to-Noise ratio (SNR) and Interference-to-Noise ratio (INR) as variables in the simulation. In this way, we can calculate different desired signal and interference signal strengths by following equations:

`

$$Desired\ Signal\ Strength = \sqrt{10^{\left(\frac{SNR}{10}\right)}}$$

$$Interference\ Signal\ Strength = \sqrt{10^{\left(\frac{INR}{10}\right)}} \quad (3.3.1)$$

Equation: Desired and Interference Signal Strength

In addition, we can vary the desired signal direction and interference direction to play around with slightly different scenarios. The steering vector as mentioned in Chapter 2 is basically the vector of desired angle with phase delays.

We synthesize the input data for the MVDR systolic array processor by multiplying the desired signal amplitude with the steering vector, doing the same for the interference signal amplitude, generating the normal Gaussian noise, and adding all of them together.

3.4 Simulation Results

We set the SNR to be 10, and INR to be 40. We set the desired angle to be -70 degrees, and the interference angle to be 30 degrees. Even though we can play around with a lot of combination for different settings here, we have decided that the readers can make educated guesses about the behavior of the algorithm when playing around with the parameters, and thus demonstrating everything here will not be relevant to the important points we are trying to present. For example, a reader can guess that decreasing the SNR while increasing the INR will worsen the performance as well as making the desired angle and interference angle so close to each other will make the algorithm fail as the algorithm will also filter out the desired signal.

`

Therefore, we will only keep one standard setting while we experiment on the important points of the algorithm when simulated on hardware behavior.

Afterwards, we ran simulations on MATLAB with these settings in two different ways: floating point calculations versus hardware-style scaled calculation.

3.4.1 Floating Point Calculations

Since MATLAB uses double precision floating point numbers, we do not need to do anything for the floating point calculations. We just ran the program with the aforementioned settings. We ran the program, and analyzed the behavior with different number of iterations. We will discuss the performance with 10, 30, 50, 100, and 200 iterations.

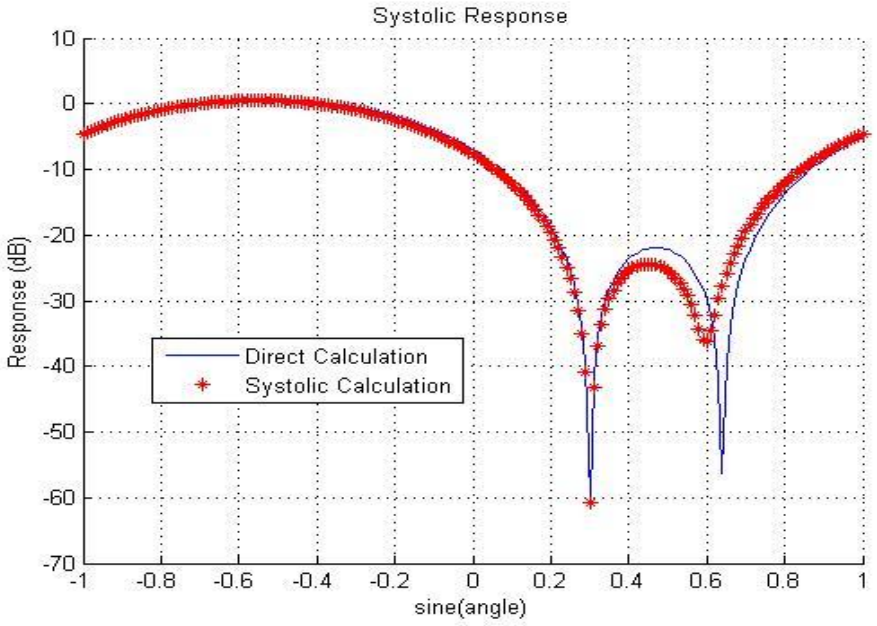

Figure 3.4.1 1: Floating Point 10 Iterations

Here, we can see t hat the algorithm is not performing well yet, i.e., it has not fully converged to the final, converged value which is calculated via the one-step direct calculation of the MVDR algorithm, and presented with the Blue line in above figure. Nevertheless, we can see

`

that the algorithm met its goal of having the unit response at the desired angle and having a dip response at the interference angle.

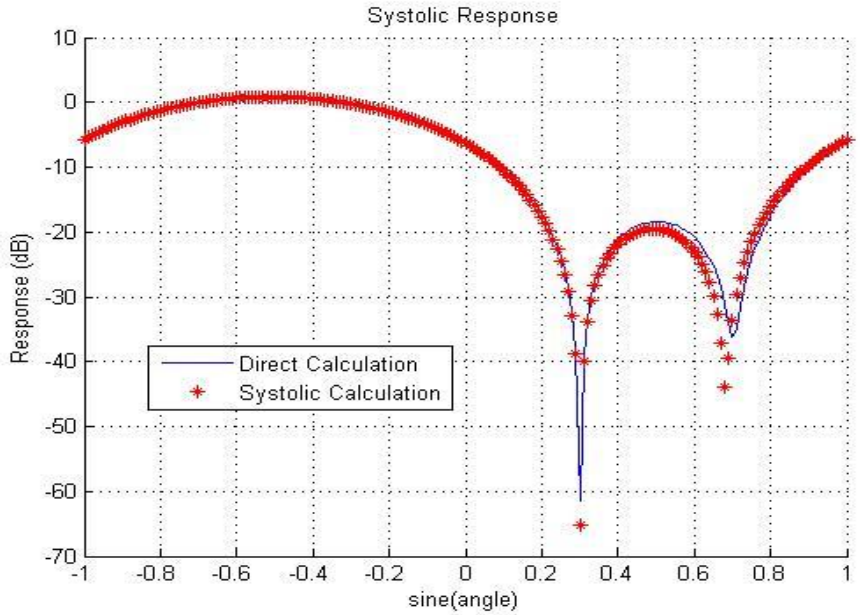

Figure 3.4.1 2: Floating Point 20 Iterations

Here, we can still see that the algorithm still has not converged. But it does show some improvement over 10 more iterations especially in the dip between 0.6 and 0.8.

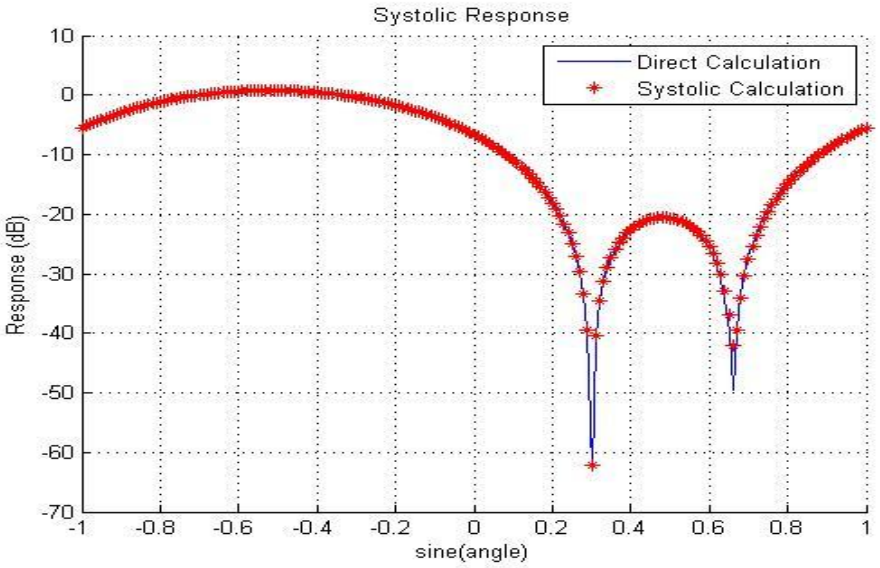

Figure 3.4.1 3: Floating Point 50 Iterations

`

As we would expect, the algorithm showed a lot of improvement over 30 more iterations. There is a perfect match on the dip between 0.2 and 0.4. Also, we can see that the dip between 0.6 and 0.7 is improved a lot better.

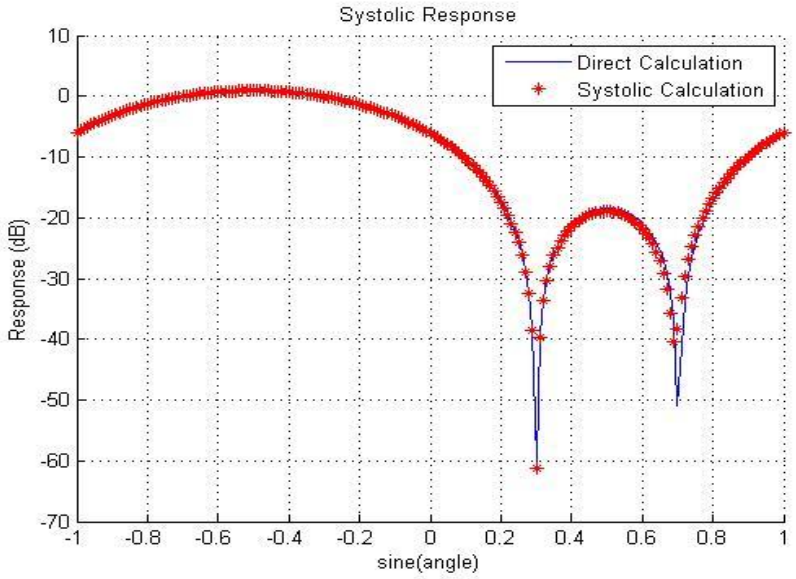

Figure 3.4.1 4: Floating Point 100 Iterations

Here, we can see that there is not much of improvement for 50 more iterations. We will iterate more to see if there will be any more improvement.

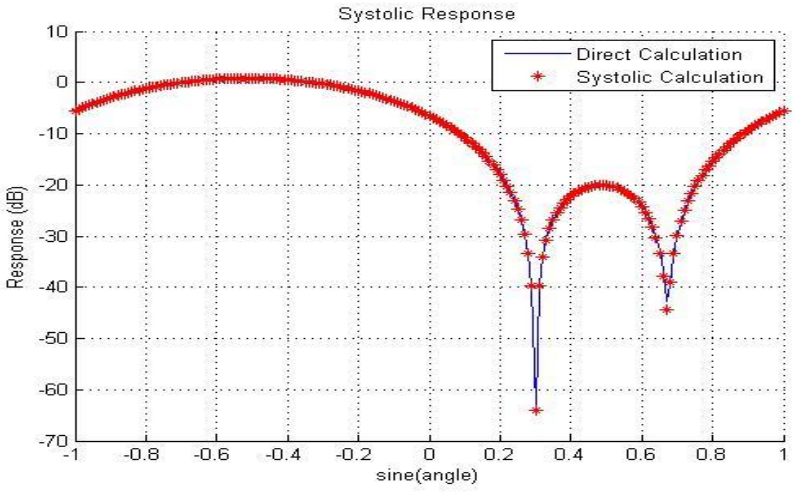

Figure 3.4.1 5: Floating Point 200 Iterations

`

Finally, after 200 iterations, the algorithm finally converged to its one-step direct calculation. We can see the step-wise improvement of the algorithm, as it is to be expected from the adaptive signal processing algorithm.

3.4.2 Hardware Scaled Calculations

On hardware, representing numbers in multiple precision floating-point numbers is expensive. Usually, this practice is avoided unless alternative is impossible. The numbers are represented in fixed-point format on typical DSP hardware. Since we would like to reduce the resource usage of our final design, we will simulate and analyze the behavior of the algorithm on limited resources.

We can simulate this easily by scaling the input values by a factor of 2. This is because all of the values and calculations will be done in fixed point binary numbers. Since MATLAB uses double precision floating point numbers, we will need to divide the numbers by powers of 2 to get the desired scaled values. In hardware, dividing by 2 is basically shifting the fixed point number to the right, which would not cost anything in terms of hardware resources.

With the scaling of the data, there will be loss of precision, which can be modeled as a form of noise. However, since the MVDR algorithm is an adaptive algorithm, we expect it to be able to converge to the final value regardless of loss of precision at the expense of requiring more iteration to arrive at that point.

Here, we will analyze the performance of the algorithm with different scaling factors, and determine the number of iterations required to be certain that algorithm converges.

We will discuss our results from using the scaling factors of 2^{-2}, 2^{-5}, 2^{-8}, and 2^{-10}. We would expect that the first two scaling values would not require a lot of iterations whereas the last two will require a lot of iterations to get the algorithm to converge.

`

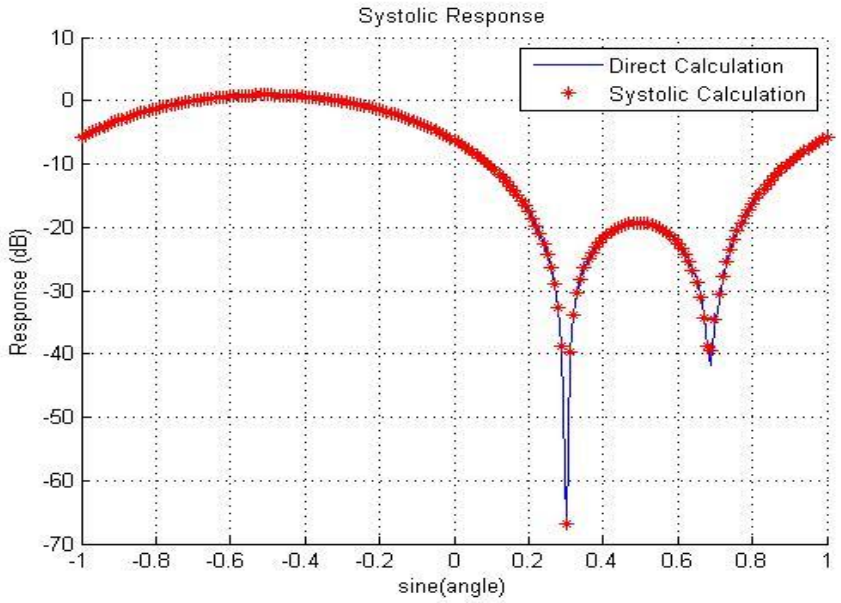

Figure 3.4.2 1: Fixed Point Scaling Factor 2^{-2}

For the scaling factor of 2^{-2}, it did not take more than the normal floating point calculation converge require for convergence. In fact, it requires exactly the same number as the former case, where it requires 200 iterations to converge.

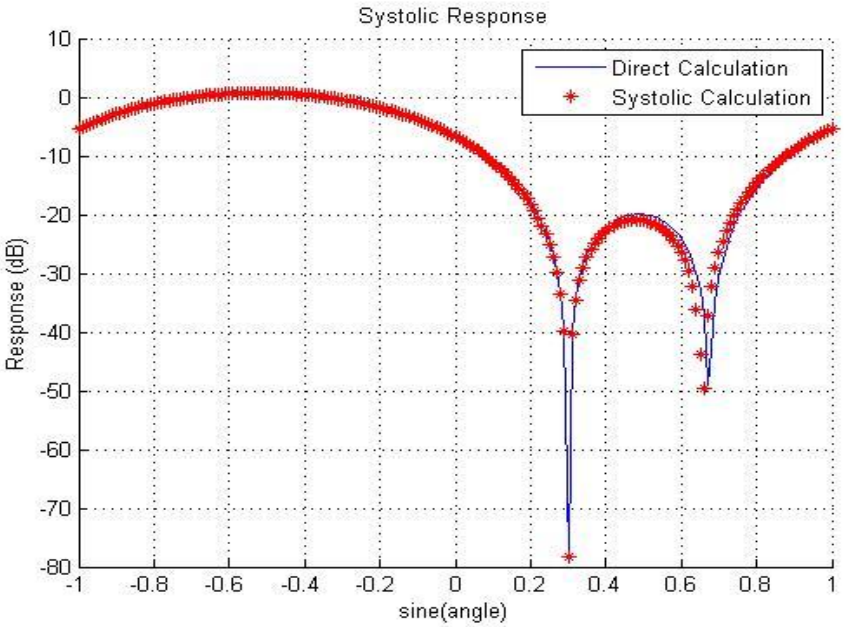

Figure 3.4.2 2: Fixed Point Scaling Factor 2^{-5}

`

For the scaling factor of 2^{-5}, we start to see the change. We can see the jump of 700 iterations from 200 to 900 iterations required to converge just by scaling three times more. We can expect a lot of exponentially growing behavior of the converging time.

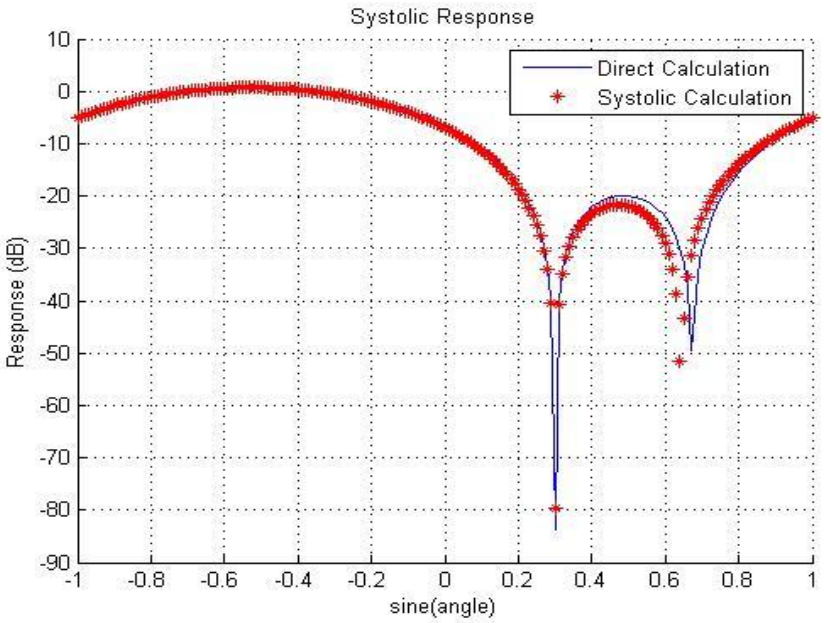

Figure 3.4.2 3: Fixed Point Scaling Factor 2^{-8}

For the scaling factor of 2^{-8}, we can see that the iteration requirement exponentially grew as we expected. With this scaling factor, the algorithm required 30000 iterations to converge. With this behavior, we can see that the algorithm will require exponentially longer time with respect to the number of bits we shifted, i.e., the amount we scaled.

`

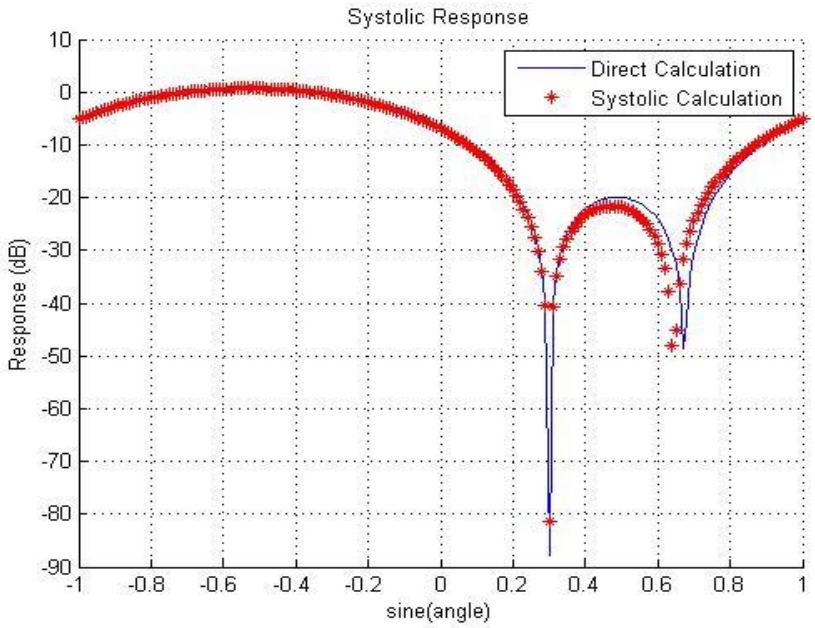

Figure 3.4.2 4: Fixed Point Scaling Factor 2-13

Finally, with the scaling factor 2^{-13}, we saw the iteration requirement of 500000.

With these experimental results, we can conclude that the scaling of the inputs to reduce the hardware resource usage has its trade-off, especially when we are designing for the real-time processing.

We will use these results into account when we design the hardware, and it is a really useful piece of information for the design process.

3.5 Chapter Summary

In this chapter, we presented the efficient implementation technique for MVDR beamforming algorithm and its simulation in the MATLAB, with different scenario and performance results.

`

Chapter 4:

System Generator Model of Beamforming

This chapter discusses the implementation process of beamforming model presented in previous chapters. Firstly, the chapter goes over the development environment we have chosen to implement the beamforming project, namely System Generator Environment [9] from Xilinx [10] design suite. Secondly, the chapter goes over the Xilinx System Generator Environment's available libraries for Digital Signal Processing (DSP). Thirdly, it discusses about the translation of the beamforming model we implemented in MATLAB as discussed in the previous chapter into System Generator Model. Subsequent sections of the chapter describe the test methodologies we employed during the implementation process, the results of the simulation, and the system resource usage.

4.1 Xilinx System Generator

This chapter describes a brief background on Xilinx System Generator development environment for DSP systems, which we chose to implement our MATLAB model. Xilinx System Generator development environment employs visual programming paradigm for development of DSP systems on Field Programmable Gate Arrays (FPGAs) by interfacing between Xilinx libraries for programmable logic devices development and MATLAB's visual programming interface Simulink.

`

4.1.1 Xilinx Corporation

Xilinx Corporation is founded in 1984, and it has headquarters in San Jose, California, United States of America. It is the world's largest supplier of Complex Programmable Logic Devices (CPLDs), and is the inventor of Field Programmable Gate Arrays (FPGAs).

Xilinx's market spans over various programmable logic products including integrated circuits (ICs), software design tools, functionally predefined intellectual property (IP) cores, design services, customer training, field engineering and technical support. Xilinx sells both FPGAs and CPLDs for electronic equipment manufacturers in end markets such as communications, industrial, consumer, automotive and data processing. Two largest families of Xilinx FPGAs are Spartan Family and Virtex Family. Spartan family targets the low-power applications, and Virtex family usually targets the System-on-Chip (SoC) solutions.

Xilinx ISE Design Suite is the Electronic Design Automation (EDA) product family featuring design and synthesis supporting Verilog or VHDL, place-and-route (PAR), verification and debug using ChipScope Pro tools, and manager for the bit files that are used to configure the chip (iMPACT).

4.1.2 System Generator for DSP

System Generator for DSP is part of Xilinx ISE design suite. The software provides visual programming interface for developing and prototyping high performance DSP systems on Xilinx FPGA families, enabling developers with little FPGA development experience to be productive on DSP hardware development.

System Generator provides system level modeling and automatic code generation by aggregating with MATLAB and Simulink modeling environments. In this way, developers can build system models on MATLAB and Simulink environment, generate Verilog and VHDL code

`

from the models, and directly compile the generated code into the bit file. The abstraction to avoid directly dealing with low-level Register Transfer Level (RTL) Verilog or VHDL code reduces the system development time significantly. It also provides hardware co-simulations between MATLAB, Simulink and the hardware platform via Ethernet or Joint Tag Action Group (JTAG). Furthermore, hardware and software co-design of embedded systems is made possible by providing soft processor cores such as Xilinx Micro-Blaze which can be programmed ahead of time, and downloaded to the hardware platform.

4.1.3 System Generator Libraries for DSP Design

System generator adds the Xilinx Blockset libraries [11] to Simulink in addition to the normal Simulink libraries. There are three library sets added to the Simulink modeling environment. They are:

1. Xilinx Blockset Library
2. Xilinx Reference Blockset Library
3. Xilinx Xtreme DSP Kit Library

Xilinx Blockset Library provides the logic design building blocks that can be used to build complex systems in different categories as follows:

1. Basic Elements
2. Communication
3. Control Logic
4. DSP
5. Data Types
6. Index

`

7. Math
8. Memory
9. Shared Memory
10. Tools

Xilinx Reference Blockset Library provides the reference models of certain complex functionality design blocks. Usually, this library is meant to be used for rapid prototyping the systems, and the implementation of the functional blocks might not be optimal for particular design projects. This library provides following categories:

1. Communication
2. Control Logic
3. DSP
4. Imaging
5. Math

Xilinx Xtreme DSP Kit Library is designed for interfacing with the Digital-to-Analog Converters (DACs), Analog-to-Digital Converters (ADCs), External RAMs and Light Emitting Diodes (LEDs) on the hardware development platforms.

The user interface for the System Generator DSP hardware modeling works exactly the same as Simulink modeling except for the fact that certain three rules must be followed to be able to generate the hardware description language code from the model directly. These specific rules are as follows:

`

1. System Generator Token must be included for the top-level design. The options for code generation and hardware platform targeting can be selected from the options made available via the System Generator Token. System Generator Token is available in Basic Elements, Index and Tools categories.
2. Xilinx Gateway-In and Gateway-Out blocks must be included in the top-level design. The Gateway-In blocks are responsible for casting the input data-type from MATLAB multi-precision floating-point into Xilinx's fixed-point data-type. The Gateway-In blocks become the top-level inputs to the system when code is generated and synthesized. The Gateway-Out blocks are responsible for casting the output data-type from Xilinx fixed-point to MATLAB's multi-precision floating-point or output data-type of the hardware platform. The Gateway-Out blocks become the top-level data outputs of the system when code is generated and synthesized.
3. Most important of all, only the functional blocks from the three Xilinx Blockset Libraries can generate the hardware description language code, and are synthesizable.

With these rules in mind, any developer with basic understanding of digital logic design can model, and synthesize functional and high performance DSP or control systems.

`

4.2 Translation of MATLAB Simulation Model

This chapter discusses about the process of translating the MATLAB simulation model of Minimum Variance Distortionless Response (MVDR) beamforming into System Generator model. There are some major challenges we faced along this process.

4.2.1 Challenges

To identify the major challenges for the implementation of the MATLAB model on System Generator, we first tried to directly implement the MATLAB model on System Generator environment. Then, we found some major challenges in terms of mathematical operations as well as in terms of design due to the limitations of the hardware platforms we had to work on.

There were two major challenges in terms of mathematical operations during the translation process: square-root and division in calculating the Givens Rotation parameters as described in Algorithm 2.1.2.1. As simple and trivial the mathematical operations they may seem, hardware implementation algorithms for these mathematical operations are usually expensive in terms of hardware resources or execution time or storage space. We researched on clever mathematical tricks to overcome this challenge by computing the square-root and division in un-conventional ways.

The big challenge in terms of design was resource usage. Most of the hardware platforms available for the project have limited resources on the FPGA one way or another, especially in terms of embedded multipliers and Look-Up Tables (LUTs). We tackled the resource limitation challenge by carefully designing the system components and system integration.

`

4.2.2 Solutions

Square-root:

For square-root operations, most of the solutions are based on the Look-Up Tables (LUTs). However, due to the unpredictability of the dynamic range of the data-input, we chose to calculate the square-root values by using the COordinate Rotation DIgital Computer (CORDIC) algorithm. CORDIC algorithm is an approximation algorithm usually used to calculate the trigonometric angles for vectors as well as the translation of component vectors into the magnitude and angle of the composite vector. For more details, please refer to [12]. Xilinx Blockset Library provides us with CORDIC IP block in DSP category [11]. In Translate mode of operation, the CORDIC IP block can receive two inputs as X and Y vector components in two-dimensional plane, and output the magnitude and phase of the composite vector. The magnitude and phase of a vector can be described as:

$$Magnitude = \sqrt{X^2 + Y^2}$$
$$Phase = \tan^{-1}(\frac{Y}{X}) \quad (4.2.2)$$

Equation: Magnitude and Phase of a Composite Vector

Thus, the square-root operations as required by Algorithm 2.1.2.1 can be calculated as the magnitude portion of the vector translate process or CORDIC algorithm. Even though there is execution trade-off of the choice for CORDIC algorithm because it is an iterative algorithm, we found out that other alternatives would pose even more restrictions in calculation of the square-root. Therefore, we decided to go with the CORDIC implementation of the square-root operation. The CORDIC blockset for System Generator can be seen in following figure:

`

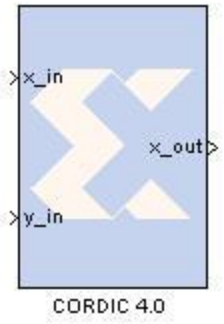

Figure 4.2.2 1: Xilinx CORDIC Blockset

Division:

For the Division calculation, we faced a similar situation since most of the hardware calculations for division uses LUTs or use CORDIC algorithm. However, because of the unpredictable nature of the dynamic range of the data-inputs, we could not use LUT approach, and we could not use CORDIC approach because the CORDIC division method available on the CORDIC blockset was not suitable for our scenario. Therefore, we did some research into the available division algorithms, and we came up with the Newton-Raphson Division method, which is a low-iteration method to find division by first finding the reciprocal of the divisor my approximation, and multiplying the dividend and the reciprocal of the divisor. The method can be generically defined as follows:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \qquad (4.2.2.1)$$

Equation: Newton's Method

`

The method requires a function which converges to zero at $f(x) = {}^{1}/_{Divisor}$ by definition to find the reciprocal of the divisor. We choose such a function as $f(x) = \frac{1}{x} - Divisor$. By plugging into Equation 4.2.2.1, we get the expression:

$$x_i = (2 - Divisor * x_i) \tag{4.2.2.2}$$

Equation: Newton-Raphson Iteration Equation

We can use Equation 4.2.2.2 for successive iteration of the Newton-Raphson method to find the reciprocal of the divisor. To find the first approximate of the iteration, we can use the approximation equation: $x_0 = T_1 - (D * T_2)$. Within the interval [0.5, 1], the constant T_1 is calculated to be 2.9142 and the constant T_2 is calculated to be 2. Therefore, the final expression for the first approximation iteration of the Newton-Raphson division technique can be derived as follows:

$$x_0 = 2.9142 - (D * 2) \tag{4.2.2.3}$$

Equation: Newton-Raphson First Iteration

For further and detailed information about Newton-Raphson Division process, please refer to [13]. With this technique, we can get a highly precise quotient within three iterations of New-Raphson technique.

Resource Limitations:

As for the resource limitations, we overcome the challenge by designing the boundary and internal cells to utilize the minimum number of multipliers possible by translating some of the implementation into serial instead of parallel as we did in MATLAB simulation. Also, we made global control logic instead of localized control logic because all of the boundary cells are the same and all of the internal cells are the same. In addition, we made the global unit of commonly used constants for boundary cells and internal cells. In this way, we not only reduced

`

the overuse of logic control blocks and constant blocks but also improved the cohesiveness of the design. Consequently, the use of LUTs are reduced with reduced number of the constants being used.

4.2.3 Boundary Cell Design

To design the boundary cell, we first break it up into several independent units which we can verify the functionality of as we move along. According to Algorithm 2.1.2.1, boundary cell is the most computationally intensive cell since there are calculations of square-root and division. We break up the boundary cell into individual units as follows:

1. Square-root
2. Reciprocal
3. Multiplication

Square-root:

For the square-root operation, in our scenario, the input "f" as described by Algorithm 2.1.2.1 will be always real-valued input, and the input "g" will always be complex value. In the actual system, the input "f" maps to the Phi value, and the input "g" maps to the complex input value to the top level, namely u_real and u_imag. Notice here that we will need to find the magnitude of the complex input first to calculate the whole square-root expression. To find the magnitude of a complex number is the same as finding the magnitude of the vector. Therefore, we can use another CORDIC block to calculate the magnitude of the complex input g. Therefore, the square-root unit is basically two cascaded CORDIC blocks in appropriate scaling and precision settings. The layout of the unit can be described as follows:

`

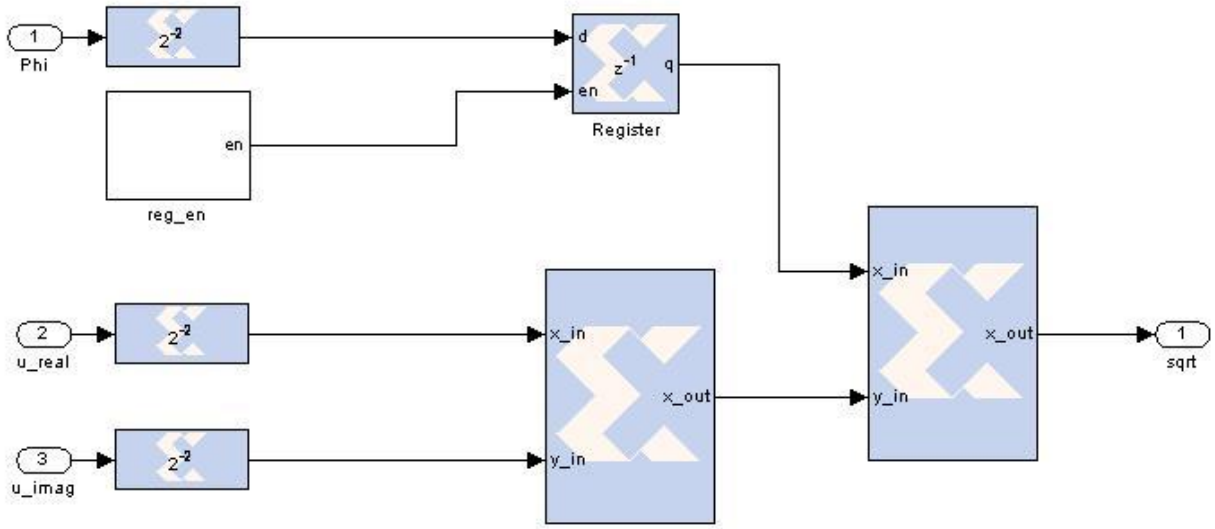

Figure 4.2.3 1: Square-root Calculation Unit

Reciprocal:

We used the Newton-Raphson division technique described in Section 4.2.2 to implement the reciprocal calculation. Basically, we implemented Equation 4.2.2.2 and Equation 4.2.2.3 in a unit, and the iteration logic is implemented by serially looping back the intermediate results to feed back into the block implementing Equation 4.2.2.2 for two times. In this way, we get our desired iteration of three on Newton-Raphson reciprocal finder to get the desired precision. The system layout can be seen as in following figure.

`

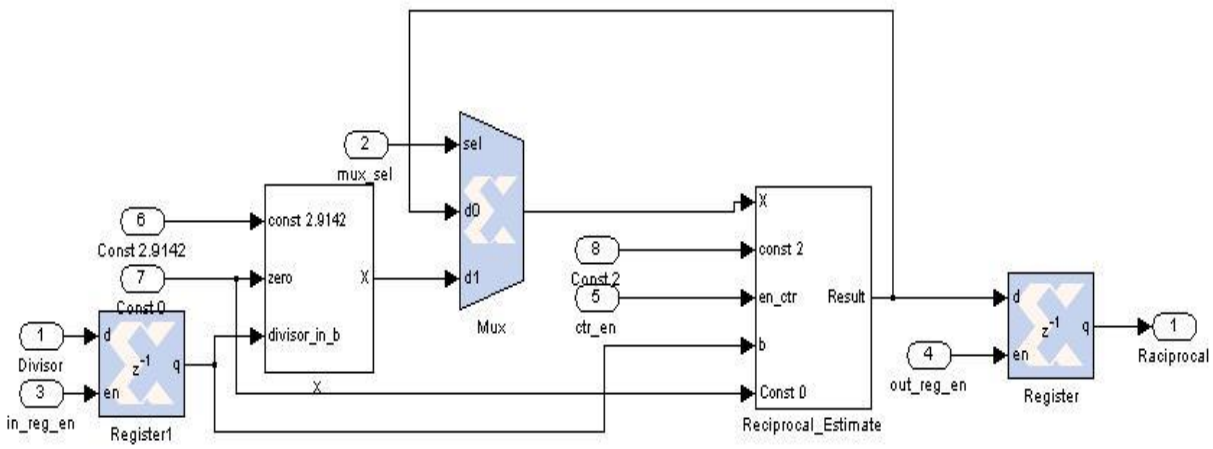

Figure 4.2.3 2: Newton-Raphson Reciprocal Calculating Unit

Multiplication:

Multiplication is done on the top-most level of the boundary cell. Here, we multiply the dividend by the reciprocal we calculated using the reciprocal unit. In following figure, the way the square-root calculating unit, the reciprocal unit, global control unit, and global constants unit along with a multiplier becomes the top-most level of boundary cell.

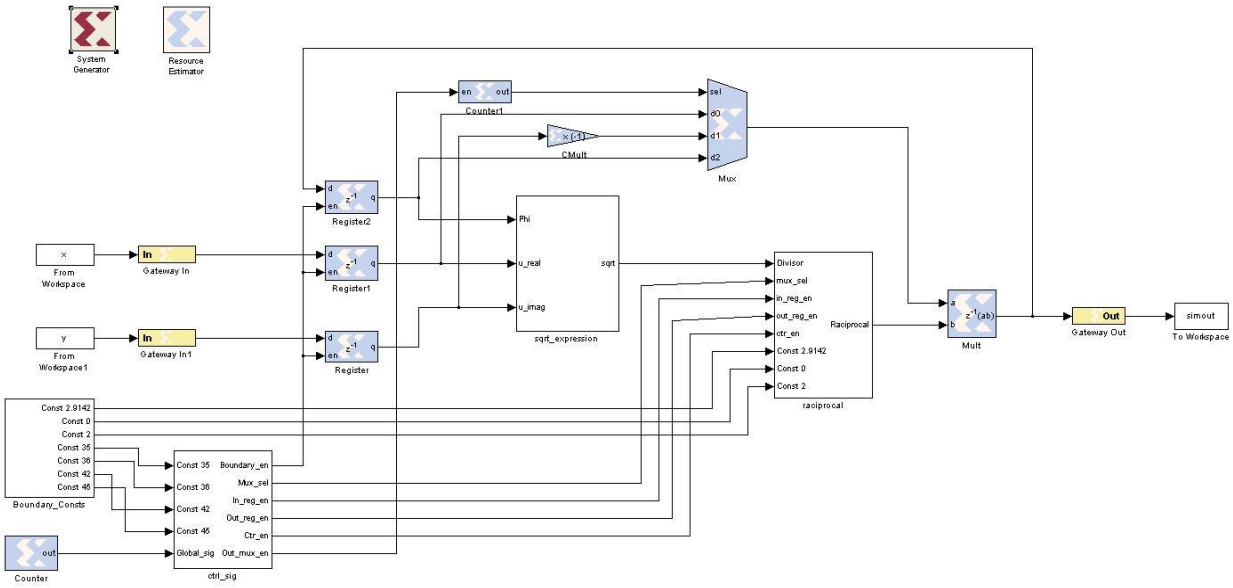

Figure 4.2.3 3: Multiplication and Top-level Unit of Boundary Cell

`

In terms of latency, the boundary cell took 51 clock cycles to get results from one complete calculation. Majority of the latency was contributed by the two cascaded CORDIC units in square-root unit which has a lot of latency in trade off for the desired level of precision.

4.2.4 Internal Cell Design

For internal cell, the main concern that came up was resource limitation since the direct, parallel implementation of an internal cell will require twelve multipliers, totaling up to 72 multipliers for the whole Systolic Array with three antenna case. Most of the accessible hardware platforms do not have that much resource in terms of multipliers, and the main goal of the internal cell design was to reduce the number of multipliers involved, by implementing the cell in serial instead of parallel. However, this way, new challenges are introduced because now, we will need to buffer the inputs and intermediate results. We utilized the block RAMs provided by Xilinx Blockset Library to buffer the input data and intermediate values. We used RAM blocks because there are plenty of distributed RAM blocks on the accessible hardware platforms, and the scenario calls for the use of RAM since the data will be read and written alternately. The rest of the unit was trivial because mathematical calculation in internal cell only involves multiplication and addition or subtraction. We put in the control logic units for controlling the address, read and write control signals for RAM blocks as well as the control unit for switching between addition and subtraction. After the condition checks, for the control units are functional, the whole system works. One notable thing is that internal cell does not have much independent individual units to test, and it is basically one large unit. In addition to the control signals, we also employed the global constants concept we used in designing the boundary cell to reduce the LUT usage. The layout of the whole system can be found in following figure.

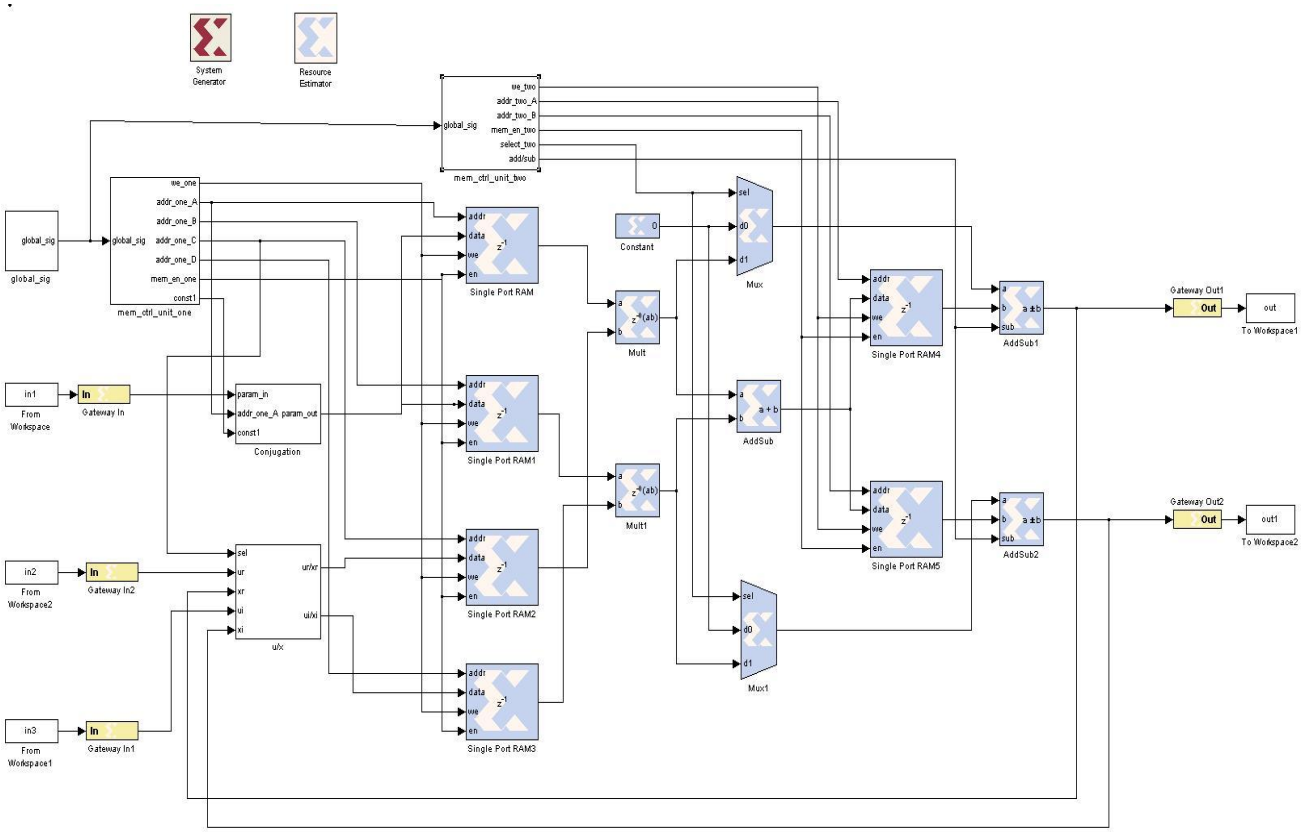

Figure 4.2.4 1: Top-level Unit of Internal Cell

Even though the internal cell's latency is not as much as 51 clock cycles, we sample and hold the outputs to release them only at the 51st clock cycle to make the whole system run uniformly, and keep the synchronization mechanisms simple.

4.2.5 System Level Integration

As for the system level integration, we connected the input and output of the boundary and internal cells as described by the Systolic Array model in MATLAB. We used iterative debug and development process described in subsequent section to make sure that all of the system component units, i.e., boundary and internal cells, are working well with each other.

`

4.3 System Test Methodologies

This section describes the test methodologies we employed for the development of the system. We followed the test-driven development process to make sure that individual pieces work as well as the whole system works.

4.3.1 Unit Testing

Unit testing is used extensively to build the system in terms of test-driven development process. We build the systems in terms of units which can be tested individually, and tested each unit before moving on. MATLAB and Simulink interface provided us with a lot of useful features to perform these unit tests easily. We can simply construct the input vectors in the MATLAB workspace, and import them into Simulink workspace to feed into the units we are testing. In addition, Simulink provides us with the feature to export the outputs of the unit in Simulink workspace to MATLAB workspace.

In this way, we can make sure that each computational unit is working by putting it through test cases, and checking the answers against the pre-computed values for the calculation.

4.3.2 Integration Testing

Integration testing was required to make sure that each functional unit we developed and verified via unit testing are working and interfacing well with each other. The most important purpose of the integration testing was to find out about the timing errors between different units. The testing process starts by first two functionally verified units and making sure that there is no timing error and the two units are communicating and interfacing with each other in expected manner.

`

Similarly, we utilized the data import and export between MATLAB and Simulink to perform the initial integration testing. We put the gradually integrated system under comprehensive test cases to make sure that we got the step-by-step working system. After the whole system was put together via the iterative integration testing, we finally tested, and verified the whole system by generating the test-bench waveforms provided in the System Generator Token.

4.3.3 Final System Results

Based upon the observations about the required number of iterations we learned as described in Section 3.4, we expected that with the quantization of 2^{-13}, it will take an extended period of time because. Therefore, we let our system run for approximately $(50 * 500000 = 25000000)$ iterations. Then, we extracted the final Phi matrix and Auxiliary matrix values to calculate the final weights, and plotted them on spectrum plot to verify that we have the weights matching our expected system behavior.

`

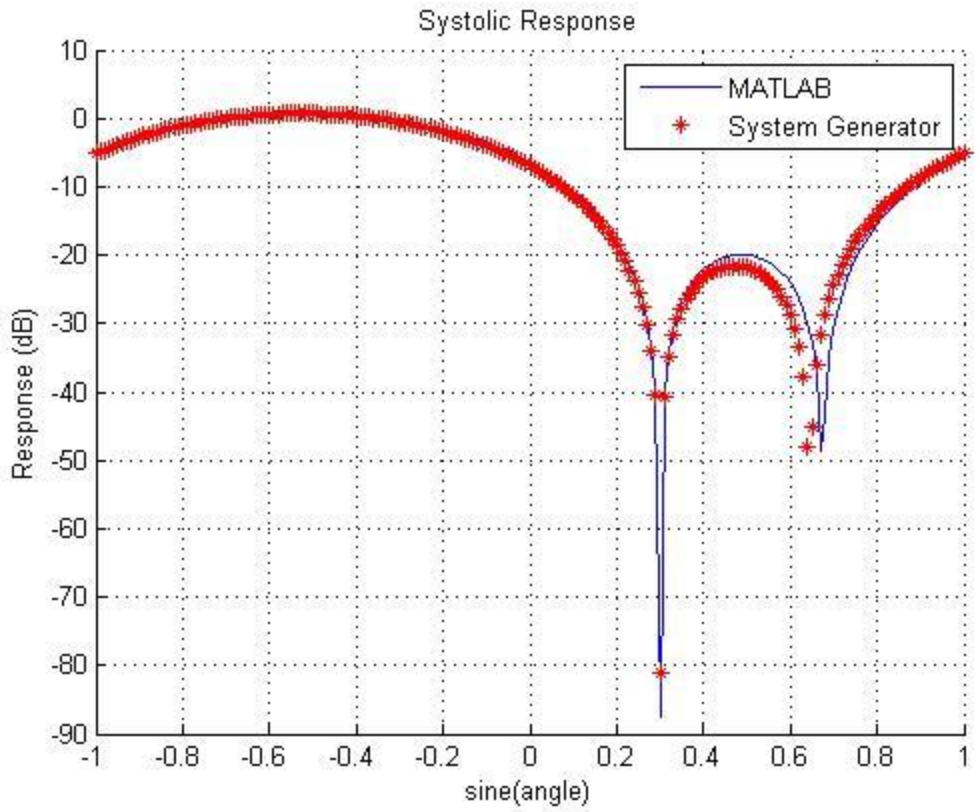

Figure 4.3.3 1: Final System Test Result

Here, for our system run, we got the desired direction at -0.7 radians, and interference direction at 0.3 radians. The blue plot was the spectrum plot for the adaptive weights from the values calculated with no precision loss on MATLAB model. The red plot was the results from the System Generator model. The System Generator result we got was not perfect and aligning with the MATLAB results because of the loss of precision in casting the values into the System Generator Model at Xilinx Gateway In blocks, and further precision loss in the process of calculating the adaptive values of Phi and Auxiliary matrix. However, the result did match with the behavior of the MATLAB simulation we had earlier.

`

4.5 Summary

In summary, we described the steps we took toward building the System Generator Model of MVDR beamforming algorithm: we described and introduced the Xilinx System Generator Environment, described the hardware algorithms for creating the functional blocks for certain mathematical operations, described the development process of the System Generator model, and discussed about the results.

`

Chapter 5:
PC to FPGA Interface

As was previously stated, the overall goal of this project was to implement a real-time MVDR adaptive beamforming algorithm onto the Virtex 5 Development Platform. In order to accomplish such a task we needed to build the hardware and software components simultaneously. Having said this, this section will discuss the software component of the project; specifically, it will examine what elements were considered and used in the implementation of the PC to FPGA interface.

5.1 Xilinx Embedded Development Kit (EDK)

Our project required that we have a working PC to FPGA interface in order to verify that our MVDR adaptive beamforming algorithm was working accurately and in a real-time environment. Therefore, we needed an application that allowed us to design an embedded processor which allows us interact with the various hardware peripherals of the Virtex 5 Development Board. Among the provided Xilinx tools, we found that Xilinx's Embedded Development Kit (EDK) was the most suitable application for this specific portion of the project because it allows us to "develop a complete embedded processor system for implementation in a Xilinx FPGA device" [14].

So what is EDK and how does it help us in implementing an interface between the PC and the FPGA? Due to the complexity of embedded systems, EDK, a collection of applications and

`

Intellectual Properties (IP), was designed by Xilinx to help propel and simplify the integration of hardware and software components of the embedded design within an FPGA. The collections of tools and IP include:

- Xilinx Platform Studio (XPS) – a development environment that helps the user design the hardware component of their embedded processor system based on the Microblaze (MB) and PowerPC (PPC) processors. To be more explicit, the XPS development tool allows the user to create and import hardware peripherals, manipulate the peripheral parameters to suit the user's needs, and even includes the ability to generate and view the system block diagram and/or design report. These tools and more are implemented into XPS to help facilitate the user's development of the embedded system's hardware component [14].
- Software Development Kit (SDK) – a complimentary development environment to the XPS helps facilitate the creation and verification of the software component of the embedded processor system through C/C++ software applications [14].
- Miscellaneous tools and IP:
    - Hardware IP for the Xilinx embedded processors
    - Drivers and libraries for the embedded software development
    - GNU compiler and debugger for C/C++ software development targeting the MB and PPC processors [14]

In order to help facilitate the embedded processor system design, all the tools provided with EDK each contribute actively in all parts of the development process. The following diagram shows the basic embedded design process performed in EDK.

`

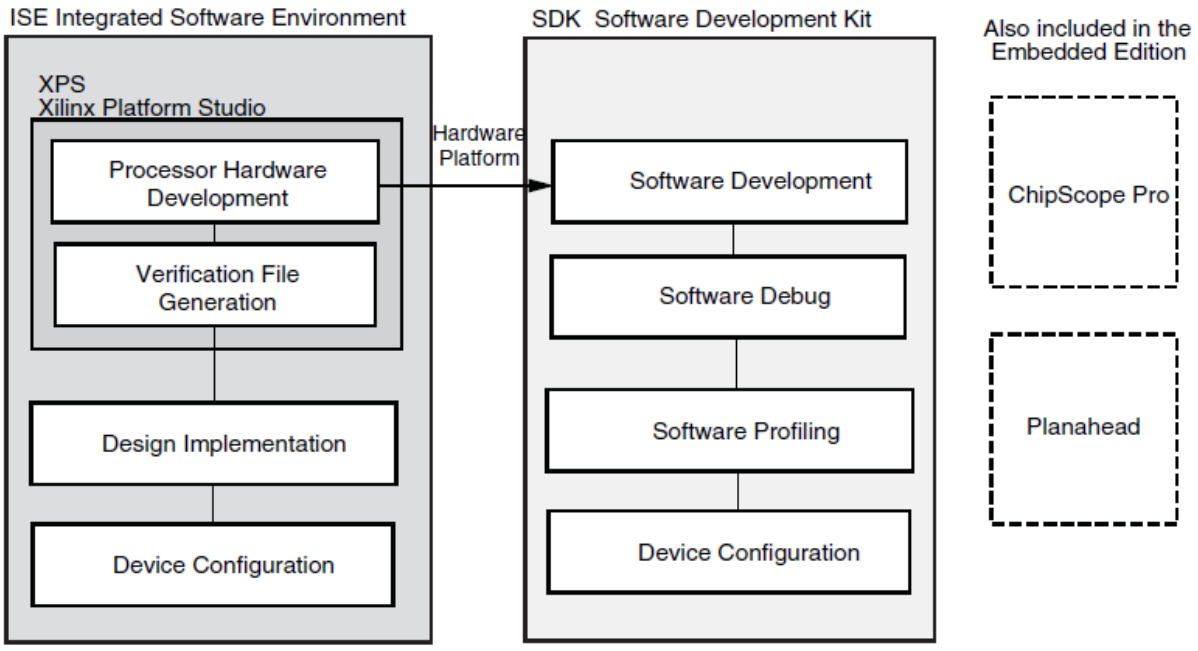

Figure 5. 1: Basic Embedded Design Flow Process [14]

5.2 I/O Interfaces

The Virtex 5 Development Board has several I/O interfaces and among those interfaces three were considered as possible candidates for our PC to FPGA interaction. Those three interfaces consisted of the following ports: RS-232 Serial Port, USB Port, and the Ethernet Port. Thus, we will discuss in moderate detail the different interfaces we took into consideration and explain as to why we chose one interface over the other two interfaces.

5.2.1 RS-232 Serial Port (UART)

The most simplistic and easy to implement among the PC to FPGA interfaces provided with the Virtex 5 Development board is the DB9, also known as the DE9, RS-232 serial connector. So what is a RS-232 serial connector? Simply speaking, an RS-232 (Recommended Standard 232) serial connector is a telecommunications standard for the communication between two or

`

more devices using single-ended binary serial data that was first introduced in 1962. In 1969 Electronics Industries Association (EIA) revised the RS-232 standards to include elements pertaining to electrical signal characteristics, pin characteristics, and various circuit interfaces of the RS-232 device [15]. However, the most interesting elements were the elements that were not defined by the standards, which include the following:

- character encoding (i.e. American Standard Code for Information Interchange (ASCII))
- framing of characters in the data stream (bits/character, start/stop bits, parity)
- error detection protocols and/or data compression algorithms
- transmission bit rates

Because these elements were not defined in the standards, a separate single integrated circuit called a Universal Asynchronous Receiver/Transmitter (UART), which converts parallel data to serial data, was needed for control over character format and transmission bit rates [15].

Having briefly reviewed RS-232, we will now briefly explain how the PC and the Virtex 5 Development Board establish a connection through their respective serial ports and why or why not the RS-232 was chosen as our choice of PC-to-FPGA interface. To establish a physical connection between the two devices a null modem, a physical link used to connect two host devices, was required because both the PC and Virtex 5 utilize serial ports that are wired as host devices. In addition, because the FPGA only utilizes the transmit (TX) and receive (RX) data pins on the serial port, all the other RS-232 signals, such as hardware flow-control, are ignored and should be disabled [16]. To disable flow-control during communications with a PC, all one has to do is properly modify the settings of the PC's terminal programming (the software medium/link) that is being used to communicate with the FPGA. That being said, we opted not to choose the RS-232 interface because of two main reasons. The first reason was because of its

`

relative simplicity and mediocrity in relative respect to an MQP, which we believe should utilize more complex designs and interfaces. The second reason was because the maximum data rate that a serial port can operate at is 115200 Baud (bits per second), which we felt was too slow if we wanted a system which operated in real-time.

5.2.2 USB vs. Ethernet

Since we decided on disregarding the RS-232 interface as a possible option due to its simplicity we were left with only two choices: USB interface and Ethernet interface. Due to the fact that they were both equally complex we wanted a simple and quick answer in ascertaining which one would be the most suitable for our project. In order to do this we looked into two different sources which we thought would give us a reasonable answer: (1) the Xilinx Forums and (2) the Xilinx example codes pertaining to USB/Ethernet to PC interface. From the data we gathered from these two resources we were able to determine that Ethernet to PC interface would be the most suitable solution as compared to the USB to PC interface. From what we found, the USB interface, although designed to be a host and peripheral device, has never been used to interact with the PC, only with keyboards and mice. On the other hand, various people have used the Ethernet interface to interact with the PC to send and receive data, exactly what we needed. Another reason, why we chose to shy away from the USB controller was because the USB and the System ACE controllers share the same data bus. In the event that we plan to use the System ACE peripheral in future work we would need to revise our configurations to use the Ethernet peripheral as oppose to the USB peripheral. Therefore, to decrease the amount of potential future work and because Ethernet interactions has already been determined as a viable solution, we decided to choose the Ethernet interface over the USB interface.

`

5.3 Lightweight IP (lwIP)

In this project, we chose the Ethernet port as our PC to FPGA interface that was used to send data between the two devices. In local area network (LAN), Ethernet is a physical medium that is used to facilitate the ongoing communication (frame/packet transmission) between two or more devices in a network. TCP/IP (Transmission Control Protocol/Internet Protocol) was a protocol that defines how packets are to be transferred in a network. TCP/IP, as the main protocol within the Internet Protocol Suite, which consists of Application, Transmission, Internet, and Link, handles how data is transmitted between two devices through a set of predetermined rules. However, because of amount of time we would have needed in order to learn TCP/IP protocol and the difficulty it would have been in designing and implementing the protocol within the Xilinx and Virtex 5 environment, we had to look into other alternatives. Instead of revisiting the possibilities of RS-232 and USB, we looked into predesigned TCP/IP networking stacks already available in EDK. After researching Ethernet TCP/IP possibilities in the context of Xilinx software, we were able to find a simplified open-source implementation of the TCP/IP protocol stack designed for embedded systems called lightweight IP (lwIP) [17].

Originally developed by Adam Dunkels, lwIP is used by several manufacturers of embedded systems, such as Altera, Xilinx, Analog Devices, and Honeywell. The lwIP TCP/IP implementation was designed to reduce resource usage while still having a full scale TCP, thus making lwIP suitable for embedded systems [18]. Similar to other implementations of TCP/IP, lwIP was based on the layered protocol design where each protocol is implemented as its individual component while having only a few functions acting as access points in the individual protocols. However, there are some differences in the implementation of the TCP protocol in lwIP as oppose to standard TCP implementations which Dunkels found to be necessary in order

`

to improve processing speed and memory usage. Unlike typical TCP standards, when authenticating the checksum of an incoming TCP packet and when de-multiplexing a packet, the TCP module must know the IP address of the source and destination. Due to performance issues, the TCP module is already conscious of the IP header structure, thus it is able to extract necessary information by itself without the need of a function call [19].

Once we had determined that lwIP was the most suitable solution to our problem, we went about finding example applications that were already developed by Xilinx so as to ease the development time of our overall project. We found the template design in Xilinx's "LightWeight IP (lwIP) Application Examples" documentation sheet [20]. In this documentation, Xilinx states specifically that provided with EDK is the lwIP software customized specifically to run on Xilinx Embedded systems containing either MicroBlaze (MB) or PowerPC (PPC), which was exactly what we needed. The documentation also described in moderate detail how we were to utilize the lwIP library in order to add networking capabilities to our design by providing several different applications: echo server, web server, TFTP server, and receive and transmit throughput tests. In addition, each of these applications were available in both RAW and Socket mode, allowing us to determine which one was more suitable for our needs [20].

Having read through and tested the various applications in both modes, it was determined that the Echo Server in Socket mode was the most suitable and easy to understand. So what is the echo server? As the name states, it is simply a program that echoes back what was received through the network interface. In the socket mode, the echo server has a main thread which continually listens on a specified echo server port, which in this case was port 7. Assuming there are multiple connections from different computers, the echo server spawns a separate echo

`

service thread upon connection request, thus allowing it to communication with multiple devices. That being said, one must note that the socket mode provides a straightforward Application Programming Interface (API) that blocks on socket reads and writes until they have been completed. Due to this, the socket API requires a multithreaded kernel (Xilinx's kernel, xilkernel) which results in a slower performance and lower throughput as compared to the RAW socket mode [20].

`

Chapter 6:
Implementation and Integration

6.1 Ethernet Interface

Once we had decided to use the Ethernet interface for our PC to FPGA interactions and the echo server application as our template design we immediately began configuring the software necessary for PC to FPGA communications. To begin we will look at the echo server developed by Xilinx that we based our entire Ethernet interface design on. For a view of the entire software design of our Ethernet interface please refer to Appendix A. For a view of the original echo server as well as the other lwIP example applications please refer to reference [20].

The software design contains two parts, the main code which sets up the various threads necessary for Ethernet communication and the echo server code which echoes back whatever input it had received via the network back through STDOUT, which is connected to the serial output. Upon initialization of the system the program's main code will display the IP address of the board as well as the various applications that the user can interact with, since we are only concerned with the echo server all other applications were removed. After the preliminary setup has been completed the program creates a main thread that continually listens for a connection request on the specified echo server port. Upon a request for connection, the program instantaneously generates an echo service thread while at the same time listening for additional connections at the echo port [20]. This is accomplished with the following code snippet:

`

```
while (1) {
      /* Spawn a separate handler for each request */
      new_socket = lwip_accept(sock, (struct sockaddr *)&remote, &size);
      sys_thread_new(process_echo_request, (void*)new_socket,
DEFAULT_THREAD_PRIO);
}
```

Having initiated an echo service thread a new socket descriptor is created and used as the service threads input. The reason why this socket descriptor is important is because it is where the echo service thread will read received data [20]. The following snippet of code shows only the most important aspects of the receiving and echoing process:

```
while (1) {
      /*    Read a maximum # of RECV_BUF_SIZE bytes from socket
            destination and store them in recv_buf    */
      n = read(socket, recv_buf, RECV_BUF_SIZE)) < 0)

      /* Handle request: Echoes back input */
      nwrote = write(socket, recv_buf, n)) < 0)
}
```

Once the program was completed we tested its validity by connecting the PC to the Virtex 5 board via a cross-over Ethernet cable for data communications and connected the PC to Virtex 5 with a null modem serial cable for standard output. The standard output is a necessary component in our testing phase because it provided us with another method of confirming the validity of the data stored in the BRAM aside from the Xilinx Microprocessor Debugger (XMD). To thoroughly test our software component (PC to FPGA interaction via Ethernet) we followed the following procedures:

1. Connect the PC and Virtex 5 with an Ethernet and null modem cable.
2. Establish a connection between the PC and FPGA using two HyperTerminal connections:
    a. HyperTerminal Connection 1 (Serial) uses the following settings:
        i. Connection Type: COM1
        ii. Bits per second (BPS): 9600

`

iii. Data bits: 8 Parity: None Stop Bits: 1 Flow Control: None

b. HyperTerminal Connection 2 (Ethernet) uses the following settings:

i. Host Address: 192.168.1.10 – IP Address of the FPGA Board

ii. Port Number: 7 – Echo Server Port Number

iii. Connection Type: TCP/IP (Winsock)

3. Run the software application via the Xilinx EDK XMD – enables the PC and FPGA to communicate with one another.
4. Upon establishing a connection, send a text file containing test data from the PC (HyperTerminal) to the FPGA via TCP/IP (lwIP) connection.
    a. Stored within the test file was sixty real decimal numbers
    b. For actual implementation a minimum of twelve hundred real decimal numbers are to be used. The reasoning behind this is because the Systolic Array implementation requires six inputs, two hundred sets of data per input.
5. Upon data reception the data stored within BRAM will be outputted via standard output in both 32-bit and 64-bit hexadecimal value.
6. After all the data has been sent and received the debugger is paused and the data within the BRAM is read in order to verify the results obtained from standard output.

Through various tests using different sets of test data (i.e. positive and negative numbers and large and small numbers) we were able to verify that the data was properly stored within the BRAM upon reception over the network.

`

6.2 Integration

Now that we had both the software and hardware components working we had to integrate the two components into one design. However, due to time constraints and various other issues we faced along the way we were unable to combine the two components into one design so as to verify if the complete system actually functioned properly. Nevertheless, if we were to integrate the portions of the design together the following would be our approach:

1. Create a Base System Builder (BSB) file containing the specifications particular to the Virtex 5 Development Board (i.e. Local Memory size, Processor configurations, and relative I/O interfaces).
    a. The specific I/Os necessary for the completion of our design that are available on the Virtex 5 are the following: DDR2 RAM, UART, Interrupt Controller, Ethernet, Timer, and System ACE.
2. Create a project within the applications directory based off of the software component.
3. Create a peripheral based off of the Systolic Array and integrate it to the entire project.

The overall design would have looked similar to that of Figure 6.1 where the BRAM interfaces with Microblaze through a Local Memory Bus (LMB) and the various hardware peripherals including the Systolic Array interfaces with Microblaze via the Processor Local Bus (PLB). As the brain of the entire operation, Microblaze is constantly processing the data that is being received through the Ethernet interface. Once the data has been received via Ethernet, Microblaze instantly routes the data to the BRAM for storage and future use. As the Systolic Array is ready to accept data for calculation it notifies Microblaze via a predetermined signal. When the signal is received Microblaze routes the data from the BRAM to the Systolic Array for processing and the result is routed back and stored within a different section of the BRAM.

`

Unfortunately, because of time constraints and our lack of knowledge with EDK and its peripheral creation system we were unable to successfully create a peripheral from our hardware component.

`

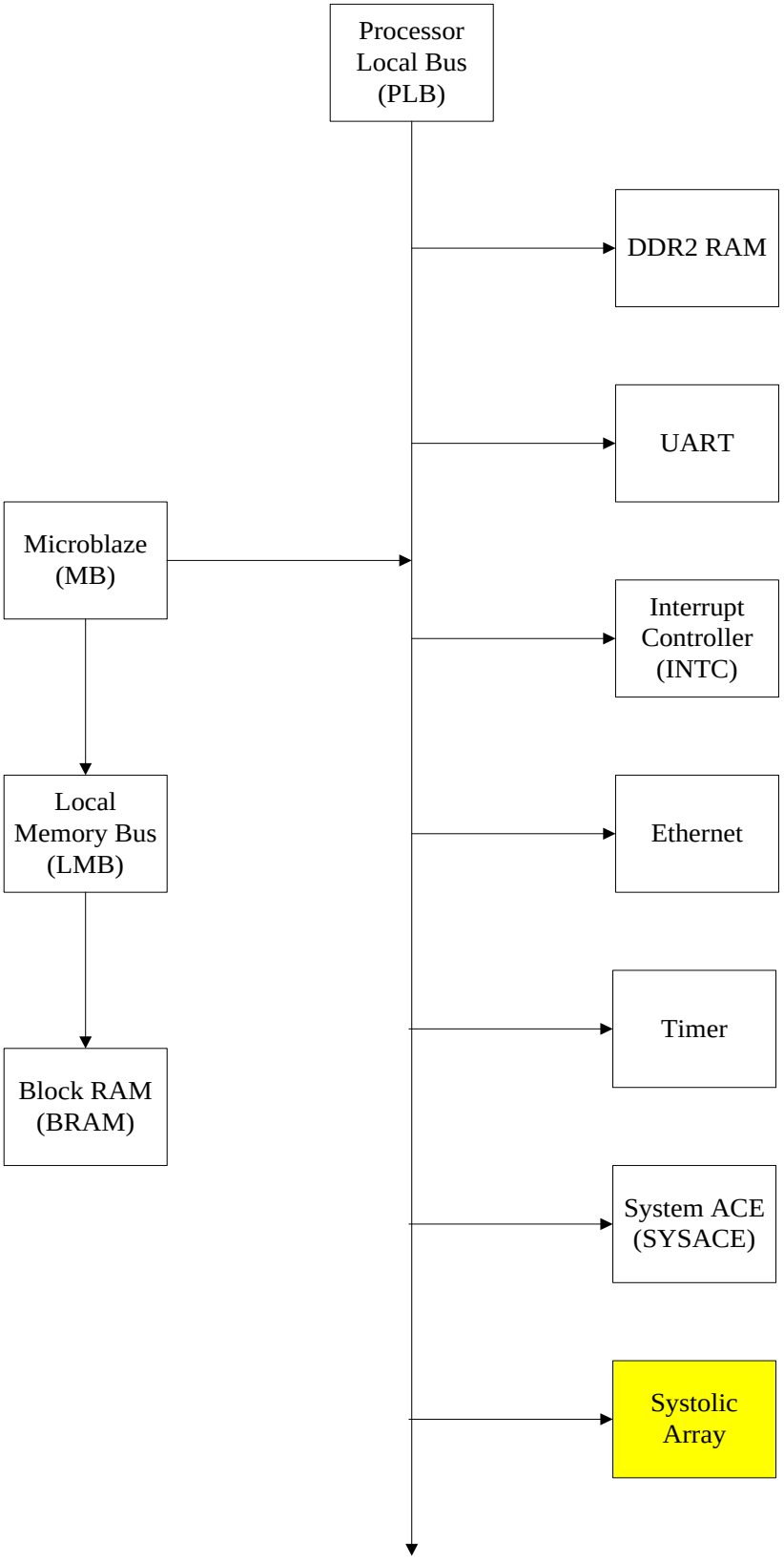

Figure 6.2 1: Overall Design

`

Chapter 7:
Conclusion

The organization of this section will be two-fold. Firstly, we will present the concluding remarks on the project, summarizing the approaches we took, challenges we went through and the results. Secondly, since our team is the pilot team in tacking the project of this nature from the undergraduate level, we shall state the recommendations based on our experiences for the future project team tackling the similar projects. This will specify the recommended course of the project phases and the features and system expansions that will be nifty to be added.

7.1 Concluding Remarks

This section will focus on the remarks on the course of the project. Firstly, we will go over the remarks we have on the overall approach, and state the remarks on the major challenges we encountered along the way. Secondly, we will state the summary of the results of the project.

7.1.1 Approach and Challenges

We approached the project in the order: learning the theoretical background on adaptive signal processing algorithms in general and MVDR beamforming algorithm, simulating the algorithm on MATLAB, building the units for the systolic array on the System Generator Environment, integrating the whole systolic array, and implementing the system on Xilinx Virtex 5 prototyping board using Xilinx Embedded Development Kit (EDK).

`

For learning the adaptive signal processing and beamforming algorithms, the major challenge we faced in is that the task requires a lot of mathematical background, i.e., from matrix computations to state space analysis. As a matter of fact, all of the text book resources available to learn the material from require a lot of mathematical maturity. The major challenge we encountered in this part of the project was due to the mathematically demanding nature of the problem statement of the MVDR beamforming algorithm. To sum up, it will be ideal to have a lot of background in mathematics to fully understand and appreciate the derivation of the algorithms.

As for MATLAB simulation part, we did not have much challenge apart from the challenges we faced while learning the algorithms. Once we could fully appreciate the algorithms, the MATLAB simulations went smoothly. The only other part where one would find challenging was the fixed-point simulation on the MATLAB, and anyone who has done a course or a project in digital signal processing would not find this intimidating.

As for building the units for the systolic array part, the major challenge we faced was finding the suitable hardware algorithms to perform expensive mathematical operations such as square-roots and divisions. These computation operations are usually implemented in hardware using the Look-up Table method. However, due to the hardware resource limitations on the available Field Programmable Gate Array processors, we were required to look into different alternatives. We learnt COordinate Rotation DIgital Computer method for finding square-roots, and Newton-Raphson method for performing divisions. These are the mathematical techniques that are usually used by the hardware designers but are rarely found in undergraduate curriculums. Thus, it posed some challenge in researching and finding the right algorithm for the implementation of these mathematical operations.

`

As for the integration of the systolic arrays out of the units we built, the major challenge we faced was the timing synchronization among the units. We spent considerable amount of our time to make sure that the timing of data flowing through the systolic array.

As for the implementation of the system on the Virtex 5 board, the major challenge we faced was learning to use the Xilinx EDK environment in a limited amount of time. Again, integrating the FPGA-based processing systems on the FPGA platform utilizing the soft-core processors on EDK environment is a standard practice in industry, and there is no class in the curriculum that prepares for the skills required to perform the operation. Therefore, learning the material in a limited amount of time was a challenging task for us.

7.1.2 Results

The results we got for the systolic processor core were decent as we described in Chapter 4. The implementation got the criteria for the MVDR response for the fixed-point implementation.

As for the integration on the Virtex 5 prototyping board using the Xilinx EDK environment, we configured the EDK environment, and wrote the necessary programs for the data communications of data between the prototyping board and the computer as described in Chapter 6. We verified that we can send and receive the simulation data back and forth between the computer and the prototyping board.

`

7.2 Future Recommendations

The nature of the project is a mixture of Communication Theory, i.e., Minimum Variance Distortionless Beamforming, and Computer Engineering, i.e., implementation of the algorithm on Field Programmable Gate Arrays.

Thus, the project requires a lot of background in mathematical communication theory and a considerable amount of experience in logic design, system-level design and moderate amount of skills in high-level language programming. In addition, the knowledge of Xilinx EDK environment and soft-core processors as well as the hardware algorithms for performing computationally extensive computations will really help for success of the project.

Here, we will describe the recommended procedures for organizing the project as well as the additional features that will be eventually added to the project if more time permits.

7.2.1 Organization of the Project

We organized the project in the following order: learning the theoretical background, simulating the algorithms on MATLAB, building the model on System Generator environment, and integrating the system on Xilinx EDK environment and Virtex 5 prototyping board.

We did have challenges for the individual parts of the project in the way we organized, and we overcame the majority of the challenges we faced. However, the major improvement that could have been done was in the integration on the Xilinx EDK and the prototyping board. The project organization would have been a lot more efficient if we organized in the way such that system generator model implementation and Xilinx EDK integration will go concurrently.

`

7.2.2 Additional Features

This section will describe the additional features that will be nifty to be added to our project, and the improvements on the system that will make it more robust, portable and power-efficient.

7.2.2.1 Direction Estimation Unit

One possible feature to be added to the system is directional estimation unit. This can be done by implementing the MUltiple SIgnal Classification (MUSIC) algorithm on the same FPGA core we implemented the system.

There are several literatures out there documenting about the implementation of the MUSIC algorithm on the FPGA platform. The most recent and comprehensive documentation on the implementation is: [21][22].

7.2.2.2 Resource Reduction

Currently, our system still takes up a lot of resources on the Virtex 5 prototyping board we are using. Critically, we are still using a lot of multipliers available on the Virtex 5 FPGA, and as we know, multipliers are limited resource on most FPGA platforms. In addition, we are using a lot of Look-up Tables available on the Virtex 5 FPGA.

One possible further improvement of the system will be reducing the resource usage, especially the limited resource on the FPGA platforms, by employing clever computational tricks, and re-designing the layout of the system. We did considerable amount of work in re-designing some parts of the boundary cells and internal cells into serial operations taking advantage of the latency we cannot avoid in our implementation of the system. In this way, we reduced the multiplier and Look-up Table usage to some extent. However, the system we came

`

up with still lacks the portability since it is still using a lot of multipliers that some of the low-power FPGAs does not possess.

7.2.2.4 Latency Reduction

In addition to the resource reduction, another possible feature to add on to the system will be the reduction of the latency of the system so that the system can run faster sampled signals. There is a design trade-off between the latency and resource usage, so decreasing one will probably increase the other.

However, there should be some ways of reducing the latency if there were not the time limitation. For example, we could find a better algorithm for finding the square-root in place of the CORDIC algorithm since most of the latency came from the CORDIC computation of the square-root.

7.2.2.5 Theoretical Optimizations

In a recent publication on QR-RLS adaptive beamforming algorithm [23], the authors stated that there can still be further optimization for the numerical stability for the QR-RLS, and ultimately the MVDR beamforming algorithm. This is a possible future improvement if the candidates working on the QR-RLS-based beamforming algorithms.

`

References:

[1] Xilinx Corporation, "Xilinx University Program XUPV5-LX110T Development System," [Online]. Available: http://www.xilinx.com/univ/xupv5-lx110t.htm. [Accessed Mar. 15, 2010].

[2] C. Dick, F. Harris, M. Pajic, and D. Vuletic, "Implementing a Real-Time Beamformer on an FPGA Platform: We designed a flexible QRD-based beamforming engine using Xilinx System Generator," *XCell Journal*, pp. 36-40. 2007.

[3] Mathworks, Inc, *MATLAB – Introduction and Key Features*, Mathworks, 2010. [Online]. Available: http://www.mathworks.com/products/matlab/description1.html. [Accessed: February 24, 2010].

[4] G.H. Golub & C.F. Van Loan, *Matrix Computations.* 3rd ed., Baltimore and London: The Johns Hopkins University Press, 1996, pp. 223 – 236.

[5] D. Bindel, J. Demmel & W.Kahan, "On Computing Givens Rotations Reliably and Efficiently," *Acm Trans. Mathematical Software*, pp. 206–238, 2002

[6] S. Haykin, *Adaptive Filter Theory.* 3rd ed., Englewood Cliffs, N.J: Prentice Hall, 1995, p. 585.

[7] The Mathworks, Inc., "The Mathworks, MATLAB and Simulink for Technical Computing," [Online]. Available: http://www.mathworks.com. [Accessed Mar. 15, 2010].

[8] H.T. Kung & C.E. Leiserson, "Systolic Arrays (for VLSI)," *Sparse Matrix Proc., Soc.ind.appl.math.*, pp. 256–282, 1978.

[9] Xilinx Corporation, "System Generator for DSP," [Online]. Available: http://www.xilinx.com/tools/sysgen.htm. [Accessed Mar. 15, 2010].

[10] Xilinx Corporation, "FPGA and CLPD Solutions from Xilinx, Inc.," [Online]. Available: http://www.xilinx.com. [Accessed Mar. 15, 2010].

[11] Xilinx Corporation, "System Generator Reference Guide," [Online]. Available: http://www.xilinx.com/support/sw_manuals/sysgen_ref.pdf. [Accessed Mar. 15, 2010].

[12] J. E. Volder, "The Cordic Trigonometric Computing Technique," *IRE Transactions on Electronic Computers*, pp. 330-334, Sept. 1959

[13] T. Granlund and P. L. Montgomery, "Division by Invariant Integers using Multiplication," *ACM SIGPLAN Notices*, vol. 29, no. 6, pp. 61-72, June 1994.

`

[14] “EDK Concepts, Tools, and Techniques: A Hands-On Guide to Effective Embedded System Design,” Xilinx, Dec. 12, 2009. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/edk_ctt.pdf. [Accessed: Jan. 12, 2010].

[15] “RS-232”, 2010. [Online]. Available: http://en.wikipedia.org/wiki/RS-232. [Accessed: March 5, 2010].

[16] “ML505/ML506/ML507 Evaluation Platform – User Guide,” Xilinx, 2009. [Online]. Available: http://www.xilinx.com/support/documentation/boards_and_kits/ug347.pdf. [Accessed: March 5, 2010].

[17] A. Dunkels, “The lwIP TCP/IP Stack,” *lwIP – A LightWeight TCP/IP Stack*, May 02, 2004. [Online]. Available: http://www.sics.se/~adam/lwip/. [Accessed: March 5, 2010].

[18] “lwIP”, Dec. 9, 2009. [Online]. Available: http://en.wikipedia.org/wiki/LwIP. [Accessed: March 6, 2010].

[19] A. Dunkels, “Design and Implementation of the lwIP TCP/IP Stack,” Swedish Institute of Computer Science, Feb. 20, 2001. [Online]. Available: http://www.sics.se/~adam/lwip/doc/lwip.pdf. [Accessed: March 6, 2010].

[20] S. Velusamy, “LightWeight IP (lwIP) Applications Example,” Xilinx, June 15, 2009. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp1026.pdf. [Accessed: Dec. 20, 2009].

[21] H. Wang & M. Glesner, "Hardware implementation of smart antenna systems," *Advances in Radio Science*, 2006. [Online]. Available: http://www.adv-radio-sci.net/4/185/2006/ars-4-185-2006.pdf. [Accessed Mar. 15, 2010].

[22] M. Kim, K. Ichige & H.Arai, "Real-time Smart Antenna System Incorporating FPGA-based Fast DOA Estimator," *Ieeexplore*, 2004. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=01399953&tag=1. [Accessed Mar. 15, 2010].

[23] Z.L. Yu, S. Wee & S.Rahadja, "QR-RLS Based Minimum Variance Distortionless Response Beamformer," *IEEE Int. Conf. Acoustics, Speech Signal Processing.* , 2006.