PARAMETERIZED HIGH PERFORMANCE CORDIC PROCESSOR ARCHITECTURES FOR FPGAS

ARCHITECTURES DE PROCESSEURS CORDIC À HAUTE PERFORMANCE AVEC PARAMÉTRAGE POUR FPGAS

A Thesis Submitted

to the Faculty of the Royal Military College of Canada

by

Guillaume Gilbert, B.Eng.

Captain

In Partial Fulfillment of the Requirements for the Degree of

Master of Applied Science in Computer Engineering

August 2005

Your file Votre référence
ISBN: 978-0-494-36473-4
Our file Notre référence
ISBN: 978-0-494-36473-4

ACKNOWLEDGMENTS

I sincerely and gratefully acknowledge the highly valued assistance, guidance and encouragement given to me by Dr. Dhamin Al-Khalili and Dr. Côme Rozon throughout the entire duration of this thesis. Their patience and understanding have helped a great deal through the more difficult times encountered during the course of this work.

I would also like to thank my dear wife Marie-Chantale, who had to endure many hardships to ensure that this work was completed.

ABSTRACT

Gilbert, Guillaume, M.A.Sc. (Computer Engineering). Royal Military College Of Canada. August 2005. Parameterized High Performance CORDIC Processor Architectures for FPGAs.
Supervisors: Dr. Dhamin Al-Khalili, Dr. Côme Rozon.

The COordinate Rotation Digital Computer (CORDIC) performs general vector rotation in the 2D plane in the circular, linear and hyperbolic coordinate systems. Its strength lies in its ability to perform various elementary functions with a relatively simple hardware implementation. The CORDIC algorithm is highly versatile, and it has been used extensively in a broad range of applications, including Digital Signal Processing, 3D Computer Graphics, Wireless Receivers, Matrix Algebra and Robotics, to name only a few.

Modern applications have ever increasing processing requirements and demand low power consumption. In order to provide the required computing power for such applications, there has been a shift from software to specialized hardware. The need for smaller devices across a wide range of applications has also been a major driving force in the proliferation of hardware solutions. This has led to the integration of entire systems on a single chip. In addition, economic considerations have made the Field Programmable Gate Array (FPGA) an important target platform, since its development costs are much lower than those required for a custom Application Specific Integrated Circuit (ASIC).

This thesis presents the development of several different types of CORDIC processing elements specifically optimized for FPGA implementation, and examines the area-performance trade-offs for each. In order to quickly generate these different processing elements, an automatic code generation tool has been developped. These CORDIC implementations can be used in a wide variety of applications, and are particularly well suited for System on Chip (SoC) designs.

RÉSUMÉ

Gilbert, Guillaume, M.Sc.A. (Génie Informatique). Collège Militaire Royal du Canada. Août 2005. Architecture de processeurs CORDIC à haute performance avec paramétrage pour FPGAs.
Superviseurs: Dr. Dhamin Al-Khalili, Dr. Côme Rozon.

Les champs d'applications modernes requièrent une grande rapidité du traitement des données, tout en exigeant une basse consommation de puissance. Afin de fournir la puissance de calcul requise dans de telles circonstances, il y a eu ces dernières années une transition des systèmes logiciels vers les systèmes matériels. La nécessité d'avoir des appareils de petite taille pour bon nombre d'applications a également contribué à la prolifération des systèmes matériels. Ceci a mené à l'intégration de systèmes entiers sur une seule puce. De plus, l'aspect économique a fait du réseau prédiffusé programmable par l'utilisateur (FPGA en anglais) une plateforme de choix pour les dévelopeurs, vu son coût moindre en comparaison avec les circuits intégrés à application spécifique.

L'algorithme CORDIC (COordinate Rotation Digital Computer) effectue la rotation d'un vecteur dans un plan à deux dimensions pour des coordonnées polaires, linéaires et hyperboliques. L'avantage de cet algorithme est qu'il permet le calcul de plusieurs fonctions élémentaires avec une implémentation matérielle relativement simple. L'algorithme CORDIC est très polyvalent, et il a été utilisé dans bon nombre d'applications, notamment le traitement numérique du signal, l'infographie 3D, les récepteurs sans fil, l'algèbre matricielle et la robotique.

Le présent ouvrage traite du développement de plusieurs différents types de processeurs CORDIC optimisés pour implémentation sur réseau prédiffusé programmable par l'utilisateur, avec un accent sur les compromis entre la surface et la performace de chaque type d'architecture. Afin de pouvoir créer rapidement ces différents processeurs, un outil capable de générer automatiquement le code source a été mis au point. Ces implémentations de l'algorithme CORDIC peuvent être utilisées dans plusieurs applications différentes, et sont particulièrement bien adaptées pour les systèmes sur une puce.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ASIC - Application Specific Integrated Circuit

CLB - Configurable Logic Block

CORDIC - COordinate Rotation Digital Computer

DCM - Digital Clock Manager

DSP - Digital Signal Processing

FG - Function Generator

FPGA - Field Programmable Gate Array

I/O - Input Output

LFSR - Linear Feedback Shift Register

LSB - Least Significant Bit

LUT - Look Up Table

MSB - Most Significant Bit

NRE - Non-Recurrent Engineering

PE - Processing Element

RAM - Random Access Memory

ROM - Read Only Memory

RNS - Redundant Number System

SoC - System on Chip

VHDL - VHSIC Hardware Description Language

VHSIC - Very High Speed Integrated Circuit

CHAPTER 1: INTRODUCTION

1.1 General Overview

The CORDIC (*COrdinate Rotation Digital Computer*) is an extremely versatile algorithm for numerical computations. It is essentially a simple hardware implementation of the Given's Transform for vector rotation in the 2D plane, and was first presented in 1959 by J.E. Volder [2]. It was later extended by Walther [3] who proposed a unified CORDIC algorithm capable of performing vector rotations in the circular, hyperbolic and linear coordinate systems. By using a CORDIC rotator, it is possible to calculate various trigonometric and hyperbolic functions, natural logarithms, natural exponential, square root, multiplication, polar to rectangular and rectangular to polar conversion. As such, the CORDIC algorithm can be used in any application which requires such computations, such as Digital Signal Processing, 3D Computer Graphics, Wireless Receivers, Matrix Algebra and Robotics, to name only a few. The floating point version of the CORDIC algorithm is used when large dynamic ranges are required, particularly in 3D Computer Graphics, Numerical Computing, Medical Research and Robotics. Some of the more notable historical applications that have made use of the CORDIC algorithm include airborne navigation (for which the algorithm was originally intended), the Hewlett Packard HP-35 scientific calculator and the Intel 8087 math co-processor.

Modern applications, such as DSP and 3D computer graphics, have ever increasing processing requirements due to the need for higher resolution and real-time applications. In order to provide the required computing power for such applications, there has been a shift from software to specialized hardware. The need for smaller devices across a wide range of applications has also been a major driving force in the proliferation of hardware solutions. An important trend resulting from this and made possible by advances in VLSI technology has been the System on Chip (SoC) approach to hardware design. SoC are complex integrated circuits that include almost all system components and interfacing hardware on a single chip or chip set.

1.2 Motivation

Traditionally, SoC designs were implemented on custom Application Specific Integrated Circuits (ASICs). ASICs require an elaborate development process, have substantial non-recurrent engineering (NRE) costs and may present a high degree of risk. They are simply not a viable option for prototyping and small to medium volume production. An alternative to the high cost ASIC are the Field Programmable Gate Arrays (FPGAs). Advances in FPGA technology now provides the hardware designer with large devices that can run at speeds of several hundred megahertz. In addition to the ever increasing number of logic elements and wiring resources, manufacturers now provide special-purpose resources such as Block RAM, multipliers, analog to digital and digital to analog converters and even entire "hard" microprocessors in a single device. This has made the FPGA a solid platform for SoC applications, along with its much simpler design flow, low NRE costs and very low degree of risk.

Designing optimized logic for implementation on an FPGA is quite different than designing digital circuits at the transistor level or using a standard cell library. The logic elements available on the FPGA impose certain limitations. In order to squeeze out maximum performance or use the least amount of space possible, the designer must find clever ways of using the available logic elements to his advantage.

Since the CORDIC algorithm is used in a wide range of different SoC applications, it is one of the most important algorithms to optimize for implementation on an FPGA. This optimization can be for area, speed, power dissipation or a combination of these. Additionally, there are many applications that require a number of different CORDIC processing elements. Examples of such applications include the Discrete Cosine Transform [4], the Fast Fourier Transform [5] and matrix transformations [6].

1.3 Objective

The aim of this thesis is to develop several different types of CORDIC processing elements specifically optimized for FPGA implementation, and to study the area-performance tradeoffs of each. These different processing elements can then be used in a variety of applications. Particular attention will be devoted to SoC applications, where relatively small area utilization is required.

In order to achieve this aim, a number of different objectives must be met. The first is to conduct an extensive survey of the state of the art in terms of the CORDIC algorithm. There has been an astounding number of different modifications that have been proposed for the algorithm in an attempt to increase its efficiency. However, not all these can be successfully applied to FPGAs, and the architectural details of the FPGA must be carefully considered when determining what CORDIC variants are suitable for implementation. The chosen algorithm will be adapted to different types of architectural styles, taking into consideration precision requirements. Functional simulation will need to be carried out on these different architectures in order to ensure correctness of the implementation. In order to provide usable components for inclusion into larger designs, an automatic code generation tool will be developed. This tool will be capable of generating processing elements according to user specified parameters. Xilinx, the leading FPGA manufacturing company, already provides such a tool called Core Generator, which includes, among other types of modules, a CORDIC module. The generated CORDIC processing elements from the custom code generation tool will be benchmarked against those produced by the Xilinx tool.

1.4 Thesis Outline

This thesis is broken down into eight chapters, including this introductory chapter. Chapter 2 presents the original CORDIC algorithm proposed by Volder and describes some of the more notable modifications presented by various researchers.

Chapter 3 examines the low level details of the programmable logic platform chosen for this thesis, the Xilinx Virtex-II. This chapter will also examine the four broad architectural styles that have been deemed the most useful for implementation of the CORDIC algorithm on an FPGA. Chapter 4 includes a comparison of the two most promising alternative CORDIC algorithms and selection of the best algorithm for implementation on the target platform. This chapter also includes error analysis and details on the internal data path format. Chapter 5 contains the hardware implementation details of the chosen CORDIC algorithm for all four architectural styles. Chapter 6 presents a novel technique used for automatic VHDL code generation and describes the tool used to generate the different CORDIC processing elements. Chapter 7 provides characterization data and benchmarks for the different CORDIC PE architectures. Finally, chapter 8 contains the conclusions and recommendations for future work resulting from this thesis.

CHAPTER 2: BACKGROUND THEORY

The term *COordinate Rotation Digital Computer* (CORDIC) was first coined by J. E. Volder in 1959 in his landmark paper *The CORDIC Trigonometric Computing Technique* [2]. He initially developed the CORDIC special purpose digital computer for real-time computation in support of airborne navigation. The CORDIC computer described by Volder essentially rotates a vector $\vec{v}$ in the 2D plane in incremental steps, and it has two different modes of operation: *rotation* and *vectoring*.

In *rotation* mode, a vector with the specified x and y end coordinates is rotated about the origin by an angle α as specified by the Given's Transform equation:

$$x' = x\cos\alpha - y\sin\alpha \tag{2.1}$$

$$y' = y\cos\alpha + x\sin\alpha \tag{2.2}$$

In *vectoring* mode, the inputs are the vector end coordinates and the results are the vector magnitude and angle argument:

$$\|\vec{v}\| = \sqrt{x^2 + y^2} \tag{2.3}$$

$$\alpha = \arctan\frac{y}{x} \tag{2.4}$$

The basic computing technique in both of these modes is a step-by-step sequence of incremental rotations that will either be equivalent to the desired overall rotation (*rotation* mode) or result in a final angle residual of zero (*vectoring* mode).

2.1 CORDIC Algorithm

2.1.1 Given's Transform

The CORDIC algorithm is elegantly demonstrated by making use of the Given's transform which performs 2D vector rotations in the Cartesian plane:

$$x' = x\cos\alpha - y\sin\alpha \tag{2.5}$$

$$y' = y\cos\alpha + x\sin\alpha \tag{2.6}$$

Equations 2.5 and 2.6 gives the relationship for converting the initial (x, y) coordinates into the new (x', y') coordinates after rotating the vector by an angle of α in the cartesian plane. The first key concept in the CORDIC algorithm is that a rotation by an angle α in the cartesian plane can also be accomplished by successive rotations of smaller angles α_i such that:

$$\sum_{i=0}^{N-1} \alpha_i = \alpha \tag{2.7}$$

The equations defining these smaller iterations are thus:

$$x_{i+1} = x_i\cos\alpha_i - y_i\sin\alpha_i \tag{2.8}$$

$$y_{i+1} = y_i\cos\alpha_i + x_i\sin\alpha_i \tag{2.9}$$

2.1.2 Pseudo-Rotations and Scaling Constant

In an attempt to further simplify the basic iteration equations, it is possible to take out the $\cos\alpha_i$ factor out of each iteration equation. Consider that cosine is an even function, that is $\cos(\alpha) = \cos(-\alpha)$. Thus, for two consecutive rotations, we have:

$$x_{i+1} = \cos\alpha_i(x_i - y_i\tan\alpha_i) \tag{2.10}$$

$$y_{i+1} = \cos\alpha_i(y_i + x_i\tan\alpha_i) \tag{2.11}$$

$$x_{i+2} = \cos\alpha_{i+1}(x_{i+1} - y_{i+1}\tan\alpha_{i+1}) \tag{2.12}$$

$$y_{i+2} = \cos\alpha_{i+1}(y_{i+1} + x_{i+1}\tan\alpha_{i+1}) \tag{2.13}$$

Considering only the x coordinate, we have:

$$x_{i+2} = \cos\alpha_{i+1}[\cos\alpha_i(x_i - y_i\tan\alpha_i) - \cos\alpha_i(y_i + x_i\tan\alpha_i)\tan\alpha_{i+1}] \tag{2.14}$$

$$x_{i+2} = \cos\alpha_i\cos\alpha_{i+1}[(x_i - y_i\tan\alpha_i) - (y_i + x_i\tan\alpha_i)\tan\alpha_{i+1}] \tag{2.15}$$

Hence, the $\cos\alpha_i$ factors can be taken out from each iteration, and can be taken care of separately. What is left is a pair of new iteration equations which are called *pseudo-rotations*:

$$x_{i+1} = x_i - y_i\tan\alpha_i \tag{2.16}$$

$$y_{i+1} = y_i + x_i\tan\alpha_i \tag{2.17}$$

Considering the relationship between *pseudo-rotations* and pure rotations as per the Given's Transform, we have:

$$x_i - y_i\tan\alpha_i = \cos^{-1}\alpha_i(x_i\cos\alpha_i - y_i\sin\alpha_i) \tag{2.18}$$

$$y_i + x_i\tan\alpha_i = \cos^{-1}\alpha_i(y_i\cos\alpha_i + x_i\sin\alpha_i) \tag{2.19}$$

Equations 2.18 and 2.19 clearly indicate that pseudo-rotations will induce growth in the vector being rotated. The magnitude of this growth is determined by the scaling factor K. For N pseudo-rotations, the final coordinates are given by:

$$x' = [x_{N-1}\cos\alpha_{N-1} - y_{N-1}\sin\alpha_{N-1}] \tag{2.20}$$

$$y' = [y_{N-1}\cos\alpha_{N-1} + x_{N-1}\sin\alpha_{N-1}] \tag{2.21}$$

$$x' = K^{-1}[x_{N-1} - y_{N-1}\tan\alpha_{N-1}] \tag{2.22}$$

$$y' = K^{-1}[y_{N-1} + x_{N-1}\tan\alpha_{N-1}] \tag{2.23}$$

Where the *scaling factor* K is defined as:

$$K = \prod_{i=0}^{N-1}\cos^{-1}\alpha_i \tag{2.24}$$

2.1.3 Arc Tangent Radix

Equations 2.16 and 2.17 provide for efficient computation, since they are composed of only one addition and one multiplication. By choosing the values of $\tan \alpha_i$ so that they are fractional powers of two, multiplication by $\tan \alpha_i$ is performed using simple right shift operations. Also note that for the special case of $\alpha = \pi/2$ rad:

$$x' = -y \tag{2.25}$$

$$y' = +x \tag{2.26}$$

Following is a partial table of tangent values that are fractional powers of two:

Tangent	Corresponding Angle
$\tan \alpha_s = \infty$	$\alpha_s = \pi/2\,\text{rad}$ (special case)
$\tan \alpha_0 = 1$	$\alpha_0 = 0.785398\,\text{rad}$
$\tan \alpha_1 = 2^{-1}$	$\alpha_1 = 0.463648\,\text{rad}$
$\tan \alpha_2 = 2^{-2}$	$\alpha_2 = 0.244979\,\text{rad}$
$\tan \alpha_3 = 2^{-3}$	$\alpha_3 = 0.124355\,\text{rad}$
$\tan \alpha_4 = 2^{-4}$	$\alpha_4 = 0.062419\,\text{rad}$
$\tan \alpha_5 = 2^{-5}$	$\alpha_5 = 0.031240\,\text{rad}$
$\tan \alpha_6 = 2^{-6}$	$\alpha_6 = 0.015624\,\text{rad}$
$\tan \alpha_7 = 2^{-7}$	$\alpha_7 = 0.007812\,\text{rad}$

These angles can be combined together in a sequence to form a special radix representation for a given angle α as follows:

$$\alpha = \sum_{i=0}^{N-1} \sigma_i \alpha_i \tag{2.27}$$

where α_i represents the Arc Tangent Radix (ATR) constants and σ_i are the ATR digits. For reasons of computational efficiency, the ATR digits are restricted to $\{-1, +1\}$.

For example, an angle of $\frac{5\pi}{180} \approx 0.087266$ rad would be obtained in the following manner:

$$+\pi/2 - \arctan(2^0) - \arctan(2^{-1}) - \arctan(2^{-2}) + \arctan(2^{-3})$$
$$- \arctan(2^{-4}) - \arctan(2^{-5}) - \arctan(2^{-6}) - \arctan(2^{-7}) \approx 0.084032\,\text{rad}$$

It's ATR digit vector is thus $\sigma = (+1, -1, -1, -1, +1, -1, -1, -1, -1)$, and forms an alternative representation of the angle. These digits indicate whether the next rotation will be positive or negative. Also note that the precision of the ATR representation obtained for an angle is dependent on the smallest angle present in the ATR constants set.

From the preceding discussion, it can be seen that the ATR constants are also used in order to calculate the scaling factor K. By keeping $\sigma_i \in \{-1, 1\}$, a constant scaling factor is ensured since $\cos(-\alpha_i) = \cos\alpha_i$. The expression for $\cos\alpha_i$ is derived as follows:

$$\tan\alpha_i = \frac{y}{x} = \frac{1}{2^i}$$
$$\vec{v} = \sqrt{y^2 + x^2} = \sqrt{1 + 2^{2i}}$$
$$\cos\alpha_i = \frac{x}{\vec{v}} = \frac{2^i}{\sqrt{1 + 2^{2i}}} = \frac{1}{\sqrt{1 + 2^{-2i}}}$$

And the scaling factor becomes:

$$K = \prod_{i=0}^{N-1} \sqrt{1 + 2^{-2i}} \tag{2.28}$$

For an infinite number of iterations, the inverse scaling factor would evaluate to the following series:

$$K^{-1} = \prod_{i=0}^{\infty} \frac{1}{\sqrt{1 + 2^{-2i}}} = 0.607253 \tag{2.29}$$

2.1.4 CORDIC Pseudo-Rotation Equations

When implementing the CORDIC algorithm, an angle accumulator is used in order to make a decision on the direction of the next rotation.

In *rotation* mode, the target angle value α is first loaded in the accumulator. At each step, a decision will be made in order to diminish the magnitude of the residual angle in the register. If the residual angle is positive, the next ATR constant needs to be subtracted from the accumulator; otherwise, the next ATR constant has to be added to the accumulator. At the end of all the iterations, only a very small residual angle should be left in the accumulator.

In *vectoring* mode, the decision will be made in order to diminish the magnitude of the current y component. At the end of all the pseudo-rotations, the residual y component has to be as close to zero as possible. In this way, the last calculated x value will contain the magnitude of the vector, and the angle register will contain the total rotation angle α. At each step, if the residual y value is positive, the next ATR constant has to be added to the accumulator; otherwise, the next ATR constant gets subtracted from the angle accumulator.

From these statements, the CORDIC pseudo-rotation equations for *rotation* mode are as follows:

$$x_{i+1} = x_i - \sigma_i y_i 2^{-i} \tag{2.30}$$

$$y_{i+1} = y_i + \sigma_i x_i 2^{-i} \tag{2.31}$$

$$z_{i+1} = z_i - \sigma_i \alpha_i \tag{2.32}$$

$$\sigma_i = \begin{cases} -1 & \text{if } z_i < 0 \\ +1 & \text{otherwise} \end{cases} \tag{2.33}$$

The equations for *vectoring* mode are exactly the same. This is obvious, since the rotation operation is the same. The only difference is the condition for the rotation direction, σ_i, which now becomes:

$$\sigma_i = \begin{cases} +1 & \text{if } y_i < 0 \\ -1 & \text{otherwise} \end{cases} \tag{2.34}$$

2.2 Unified CORDIC Algorithm

By slightly modifying the pseudo-rotation equations, Walther [3] was able to present a unified CORDIC algorithm capable of performing rotations in the linear and hyperbolic coordinate systems in addition to the circular rotations first presented by Volder. The unified CORDIC pseudo-rotation equations are given by:

$$x_{i+1} = x_i - m\sigma_i 2^{-S(m,i)} y_i \tag{2.35}$$

$$y_{i+1} = y_i + \sigma_i 2^{-S(m,i)} x_i \tag{2.36}$$

$$z_{i+1} = z_i - \sigma_i \alpha_{m,i} \tag{2.37}$$

where $m = 1$ for circular, $m = 0$ for linear and $m = -1$ for hyperbolic coordinate systems. The direction of rotation for both rotation and vectoring modes are still $\sigma_i = sign(z_i)$ and $\sigma_i = -sign(y_i)$ respectively. The $S(m, i)$ variables indicate the amount of shift for each iteration. Throughout this text, if the m variable is missing, it is assumed that $m = 1$, or S(i) = S(1,i), corresponding to the circular coordinate system. In the circular and linear coordinate systems, the shift sequences are given by $S(1, i) = S(0, i) = (0, 1, 2, 3, ...)$. In order to preserve convergence, the hyperbolic coordinate system requires that certain elements in the shift sequence be repeated. Convergence in the CORDIC algorithm will be discussed in further sections. Thus, the shift sequence for the hyperbolic coordinate system is $S(-1, i) = (1, 2, 3, 4, 4, 5, ..., 12, 13, 13, 14, ...)$ where the repeated elements are given by $\{3k + 1\}, k \in N$. The scaling factor for the hyperbolic and circular coordinate systems is now given by:

$$K_{m,i} = \prod_{i=0}^{N-1} \sqrt{1 + m\sigma_i^2 2^{-2S(m,i)}} \tag{2.38}$$

Note that scaling is not applicable to the linear coordinate system. Also, the angle basis for the unified CORDIC now becomes:

$$\alpha_{m,i} = \frac{1}{\sqrt{m}} \arctan\left(\sqrt{m} 2^{-S(m,i)}\right) \tag{2.39}$$

2.3 Algorithm Modifications

Following the initial work on the CORDIC algorithm by Volder and Walther, there has been a tremendous amount of work aimed at improving the basic CORDIC algorithm. The bulk of this work has been in devising "short-cut" methods in order to speed up the algorithm. The most notable ideas are presented in this section and are grouped into broad categories.

2.3.1 Hybrid Radix Sets and Rotation Prediction

The main bottleneck of the CORDIC algorithm lies in the fact that the direction of rotation for a particular iteration depends on the results of the previous iteration. If the Arc Tangent Radix representation of the angle is known in advance, then there is no dependency on the previous iterations and there is no need for the z data path (in rotation mode). However, the input angle is usually encoded in conventional two's complement representation. There have been numerous publications which discuss the process of deriving the ATR representation directly from the two's complement representation of the angle. These are usually based on the Hybrid Radix Set discussed in [7].

Hybrid radix sets rely on the following relationship:

$$\lim_{k\to\infty} \frac{\tan 2^{-k}}{2^{-k}} = 1 \tag{2.40}$$

As k increases, the difference between the circular ATR angles and the radix-2 shift coefficients becomes more and more negligeable. However, for small values of k, this difference is quite large. The solution proposed in [7] is to use the circular ATR angle rotations for the first n iterations, and use the linear coefficients for the remainder of the iterations. In fact, it is shown in [7] that $n = \left\lceil \frac{N-\log_2 3}{N} \right\rceil$. This *Mixed-Hybrid Radix Set* is of the form:

$$\{\overbrace{\arctan\infty, \arctan 1, \arctan 2^{-1}, \ldots, \arctan 2^{-n+1}}^{\text{most significant part}}, \quad \underbrace{2^{-n}, \ldots, 2^{-N-1}}_{\text{least significant part}} \quad \}$$

When such an angle quantization method is used, iterations 0 to iteration $(n-1)$ are performed as in the original CORDIC. At the end of these first iterations, there will be an angle residual left in the angle accumulator. The value of the residual angle is the radix-2 representation (approximation) of the remaining angle rotation in the linear coordinate system. In the case of a positive residual angle, a 1 indicates a positive rotation and a 0 indicates a non-rotation. A similar approach can be used for negative residual angles. Introducing non-rotations will lead to a non-constant scaling factor, since for a rotation by $\alpha_i = 0$, we have $\cos\alpha_i = \cos(\arctan 0) = 1$. The only way to keep the scaling factor constant is to perform either a positive or negative rotation at every iteration, since $\cos(\arctan(-2^{-i})) = \cos(\arctan 2^{-i})$. The authors in [7] also note that the computation of α_H, the most significant part of α might impact α_L, the least significant part of α. Hence, there must be a separation of the rotation prediction for α_H and α_L.

The problem then becomes the prediction of the first n terms which are not directly available from the two's complement representation of the input angle. One way of accomplishing this is by compressing the α_H rotations into a single rotation that is implemented by the use of a ROM as discussed in [7]. Others have developed combinational means of generating the α_H ATR digits [8] [9]. Usage of the Booth encoding algorithm has also been used in order to recode angles [10] [11].

2.3.2 Modified ATR Sets

The term *Angle Set* designates the elementary angles which form a radix representation of an angle. The Angle Set is an important factor when examining the CORDIC algorithm. One obvious impact in the choice of an Angle Set is in the accuracy of the results. The Angle Set also has a direct influence on the scaling factor and on rotation direction determination. An excellent generalization concerning Angle Set Selection can be found in [12]. In this work, the authors discuss the concept of Angle Quantization, which is defined as the process of decomposing an arbitrary angle θ in terms of a set of sub-angles θ_i. In Volder's original work [2], the term Arc Tangent Radix (ATR) was used in order to stress the fact that these sub-angles were expressed as $\alpha_i = \arctan(2^{-i})$. However, the terminology found in [12] will be used in order to demonstrate this concept.

The *Elementary Angle Set* (EAS) is defined as:

$$S_1 = \left\{\arctan(\sigma 2^{-k}) : \sigma \in \{-1, 0, 1\}, k \in \{0, 1, \ldots, N-1\}\right\} \tag{2.41}$$

From this definition, it can be seen that Volder's ATR representation is a subset of the EAS where $\sigma \in \{-1, 1\}$. The ATR representation ensures that for a fixed number of iterations, the scaling factor will remain constant. However, the definition of the EAS allows for the possibility that a rotation might not be executed during a particular iteration (i.e. when $\sigma = 0$). In this case, the scaling factor does not remain constant.

The *Extended Elementary Angle Set* (EEAS) is defined as:

$$S_2 = \left\{\arctan(\sigma_0 2^{-k_0} + \sigma_1 2^{-k_1}) : \sigma_0, \sigma_1 \in \{-1, 0, 1\}, k_0, k_1 \in \{0, 1, \ldots, N-1\}\right\} \tag{2.42}$$

The EEAS allows for more flexibility in the choice of angles that make up the set. This flexibility can actually be used in order to "steer" the scaling factor to a Signed Power of Two (SPT), which would eliminate the need for a costly post-scaling multiplier.

2.3.3 Combined Successive Iterations

In order to solve different types of bottlenecks with the CORDIC iterations, researchers have combined successive iterations. The way that successive iterations are combined depends on the problem that is being addressed. Some of these solutions are concerned with angle convergence, others are concerned with scaling factor and others still try to increase the parallelism of the CORDIC process.

There have been several attempts made at combining two successive CORDIC iterations into a single iteration. *Double Rotation CORDIC* is derived as follows:

$$x_{i+1} = x_i - y_i \sigma_i 2^{-i} \tag{2.43}$$

$$y_{i+1} = y_i + x_i \sigma_i 2^{-i} \tag{2.44}$$

$$x_{i+2} = x_{i+1} - y_{i+1} \sigma_{i+1} 2^{-i} \tag{2.45}$$

$$x_{i+2} = (x_i - y_i \sigma_i 2^{-i}) - (y_i + x_i \sigma_i 2^{-i}) \sigma_{i+1} 2^{-i+1} \tag{2.46}$$

$$x_{i+2} = x_i(1 - \sigma_i \sigma_{i+1} 2^{-i} 2^{-i+1}) + y_i(\sigma_i 2^{-i} + \sigma_{i+1} 2^{-i+1}) \tag{2.47}$$

$$y_{i+2} = y_i(1 - \sigma_i \sigma_{i+1} 2^{-i} 2^{-i+1}) - x_i(\sigma_i 2^{-i} + \sigma_{i+1} 2^{-i+1}) \tag{2.48}$$

In [13], the combination of two successive iterations provides for a new angle recoding scheme. The recoding scheme uses a ROM based approach in order to encode the required rotation directions for each angle. In this manner, the algorithm has a non-constant scaling factor, and relies on a fast variable scale factor decomposition and compensation algorithm.

Another well known paper that uses this approach is found in [14]. In this paper, double rotations are introduced in order to correct the scale factor issues associated with the *Redundant CORDIC*, and it uses the redundant binary number representation with the digit set of $\{\bar{1}, 0, 1\}$. The *Double Rotation CORDIC* performs three types of rotations: negative, positive and non-rotation. These three types of rotations are performed with the help of two consecutive sub-rotations (hence the term *Double Rotation*). The smaller sub rotation angles are chosen such that their arctangent value are half those of regular CORDIC, that is $\arctan \frac{2^{-i}}{2} = \arctan 2^{-i-1}$. Interestingly enough, all possible rotation angles performed in *Regular CORDIC* will also be used in this algorithm; however, each rotation will be applied twice. The double rotations are derived in a slightly different way:

$$x_{i+1} = x_i - y_i \sigma_i 2^{-i-1} \tag{2.49}$$

$$y_{i+1} = y_i + x_i \sigma_i 2^{-i-1} \tag{2.50}$$

Considering only the x data path, we have:

$$x_{i+2} = x_{i+1} - y_{i+1}\sigma_{i+1}2^{-i-1} \tag{2.51}$$

$$x_{i+2} = \left[x_i - y_i\sigma_i 2^{-i-1}\right] - \left[y_i + x_i\sigma_i 2^{-i-1}\right]\sigma_{i+1}2^{-i-1} \tag{2.52}$$

$$x_{i+2} = x_i - 2^{-1}\left[\sigma_i + \sigma_{i+1}\right] y_i 2^{-i} - \left[\sigma_i \cdot \sigma_{i+1}\right] x_i 2^{-2i-2} \tag{2.53}$$

Setting $q_i = 2^{-1}\left[\sigma_i + \sigma_{i+1}\right]$ and $p_i = \left[\sigma_i \cdot \sigma_{i+1}\right]$ we have

$$x_{i+2} = x_i - q_i y_i 2^{-i} - p_i x_i 2^{-2i-2} \tag{2.54}$$

The positive, negative and non rotations are thus defined as follows:

$$\begin{aligned} \text{negative rotations}: \ \sigma_i = \bar{1}, \sigma_{i+1} = \bar{1} &\Rightarrow q_i = \bar{1}, p_i = 1 \\ \text{nonrotations}: \ \sigma_i = \bar{1}, \sigma_{i+1} = 1 &\Rightarrow q_i = 0, p_i = \bar{1} \\ \text{positive rotations}: \ \sigma_i = 1, \sigma_{i+1} = 1 &\Rightarrow q_i = 1, p_i = 1 \end{aligned}$$

Since there is no need for an $i+1$ iteration, i and $i+2$ effectively become two successive iterations. Note also that the decision for the direction of the rotation (q_i and p_i) are based on the three most significant digits of z_j. Thus, the pseudo-rotation and decision equations can be written as follows:

$$x_{j+1} = x_j - q_j y_j 2^{-j} - p_j x_j 2^{-2j-2} \tag{2.55}$$

$$y_{j+1} = y_j + q_j x_j 2^{-j} - p_j y_j 2^{-2j-2} \tag{2.56}$$

$$z_{j+1} = z_j - q_j 2 \arctan 2^{-i-1} \tag{2.57}$$

$$(q_j, p_j) = \begin{cases} (\bar{1}, 1) & \text{if } \left[z_j^{j-1} z_j^j z_j^{j+1}\right] < 0, \\ (0, \bar{1}) & \text{if } \left[z_j^{j-1} z_j^j z_j^{j+1}\right] = 0, \\ (1, 1) & \text{if } \left[z_j^{j-1} z_j^j z_j^{j+1}\right] > 0. \end{cases} \tag{2.58}$$

As previously mentioned, the main limiting factor of the CORDIC algorithm is that it is iterative in nature, i.e. the value of x_{i+1} and y_{i+1} are a function of both

x_i and y_i. By successive substitutions of x_i and y_i in the basic CORDIC equations, it is possible to derive an expression for the final values x_N and y_N that is only dependent on x_0 and y_0 [15]. This type of algorithm is called the *Flat CORDIC*, and the resulting expressions become quite simple to implement at the hardware level. However, the drawback of Flat CORDIC is that the ATR representation of the angle must be available before processing; this is accomplished with the use of a specialized circuit called the *signed digit generator* in [15], and is based on rotation prediction as described in section 2.3.1.

In the derivation of the *Merged CORDIC* algorithm [16], matrix operations are used in order to combine successive iterations. The idea behind merging the iterations is to rearrange their sequence in such a way that the ith iteration is next to the $(n-i+1)$ iteration. The resulting iteration equations of the merged CORDIC thus become:

$$x_{i+1} = x_i - (\sigma_i 2^{-i} + \sigma_{n-i+1} 2^{-n+i-1}) y_i \tag{2.59}$$

$$y_{i+1} = y_i + (\sigma_i 2^{-i} + \sigma_{n-i+1} 2^{-n+i-1}) x_i \tag{2.60}$$

$$z_{i+1} = z_i - \sigma_i \alpha_i - \sigma_{n-i+1} \alpha_{n-i+1} \tag{2.61}$$

With merged CORDIC there are considerable hardware savings since n-bit shifters are not required; only $\frac{n}{2}$ -bit shifters are needed.

2.3.4 Scaling Factor Compensation

As discussed earlier, the pseudo-rotations will induce magnitude growth in the rotated vector. In order to obtain unscaled values, the final x and y values must be multiplied by the inverse scaling factor given by:

$$\frac{1}{K} = \prod_{i=0}^{N-1} \frac{1}{\sqrt{1+2^{-2i}}} \tag{2.62}$$

The most straightforward method of compensation is to multiply the final x and y values by the inverse scaling factor (post-scaling). It is also possible to perform the multiplication prior to the CORDIC iterations (pre-scaling). Multiplier recoding into Canonical Signed Digit can be used in order to simplify the multiplication.

Another method of performing compensation is to introduce *normalization steps* in between CORDIC iterations as presented in [17]. The scaling factor can be expressed as:

$$\frac{1}{K_m(n)} = \prod_{i=0}^{N-1}(1 - m\gamma_{m,i}2^{-S(m,i)}) \tag{2.63}$$

where $\gamma_{m,i} \in \{0,1\}$. Finally, the normalization steps are:

$$x_{i+1,norm} = x_{i+1} - mx_{i+1}\gamma_{m,i}2^{-S(m,i)} \tag{2.64}$$

$$y_{i+1,norm} = y_{i+1} - my_{i+1}\gamma_{m,i}2^{-S(m,i)} \tag{2.65}$$

As mentioned in 2.3.2, the *double shift* iterations of the EEAS angle set can be used in order to steer the inverse scaling to a simple sum of signed powers of two. The generalized equation of the unified CORDIC for double shift is given by:

$$x_{i+1} = x_i - m\sigma_i y_i 2^{-S(m,i)} - m\sigma_i\eta_{m,i}y_i2^{-S'(m,i)} \tag{2.66}$$

$$y_{i+1} = y_i + m\sigma_i x_i 2^{-S(m,i)} + m\sigma_i\eta_{m,i}x_i2^{-S'(m,i)} \tag{2.67}$$

$$\alpha_i = \frac{1}{\sqrt{m}}\arctan\left[\sqrt{m}\left(2^{-S(m,i)} + \eta_{m,i}2^{-S'(m,i)}\right)\right] \tag{2.68}$$

where $\eta_{m,i} \in \{-1,0,1\}$ and S'(m,i) is a second shift factor.

Another variation that is quite similar to the double shift CORDIC is the *compensated CORDIC* presented in [5]. The iteration equations for the compensated CORDIC are given by:

$$x_{i+1} = x_i - m\sigma_i y_i 2^{-S(m,i)} + \eta_{m,i}x_i2^{-S(m,i)} \tag{2.69}$$

$$y_{i+1} = y_i + m\sigma_i x_i 2^{-S(m,i)} + \eta_{m,i}y_i2^{-S(m,i)} \tag{2.70}$$

$$\alpha_i = \frac{1}{\sqrt{m}}\arctan\left[\frac{\sqrt{m}}{2^{S(m,i)} + \eta_{m,i}}\right] \tag{2.71}$$

2.3.5 Floating Point CORDIC

The shift and add nature of the CORDIC algorithm makes use of the fixed point format. Performance degradation would occur if these operations were to be carried out in Floating Point format, which requires more overhead.

In most cases, Floating Point implementations will convert the Floating Point input to a Fixed Point internal representations, and reconvert this internal representation to Floating Point at the output. In order to do so, a working exponent must first be determined, which is the largest of the two exponents. The mantissas of both inputs have to be aligned in order to perform the CORDIC iterations. Examples of usage of block floating point can be found in [18] [19] [20]. Another point to consider in the use of the floating point representation is that in certain applications, a fixed point representation might be all that is needed [20].

One recent paper discusses the use of the Floating Point Format for internal calculations in support of the Singular Value Decomposition Problem [21]. The authors state that the parallelism of the systolic array architecture reduces the penalty from the overhead of floating point normalization in each pipeline stage.

2.3.6 Alternative Number Representation System

In attempting to improve the CORDIC algorithm, Redundant Number Systems have been used in order to speed up the arithmetic operation of addition in each iteration. The inherent difficulty in the use of such a number representation is the determination of the next direction for rotation, since the sign of a binary number can no longer be determined by simply examining the MSB. Usually, the sign of an RNS number can only be determined by starting from the MSB, and examining each digit in sequence until a non-zero digit is encountered. As discussed in [14], it is possible to examine only a limited number of bits which form a *window*, as all bits to the left of this window are guaranteed to be zero in the CORDIC process. Others have proposed the Branching and Double Step Branching CORDIC algorithm [22]

[23]. The Branching CORDIC requires two CORDIC processors which perform the iterations in parallel. When the sign of the next rotation can be determined from the bits within the *window*, the two processors execute the same rotation. When the direction cannot be determined, they take different rotation directions; in this way, one of the two processors is guaranteed to converge. When the next branching decision is triggered by one of the two processors, that processor is the one that has taken the proper direction for convergence.

Traditionally, radix-2 based number representation systems have been used for both the redundant and non-redundant CORDIC algorithm. There have been numerous publications that use higher order radix representations (radix-4, 8) [24] [25] [26] [27] and very-high radix (radix-512) [28]. These representations allow for more bits of the result to be calculated in a single iteration and hence, reduce the required number of iterations. A higher order radix r is always a power of two. The Arc Tangent Radix set is thus extended such that $\alpha_i\left[\sigma_i\right] = \arctan(\sigma_i r^{-i})$, where $\sigma_i \in \{-r/2, \ldots, 0, \ldots, r/2\}$. Such a representation gives a redundant ATR representation of angles. For example, the radix-4 CORDIC pseudo-rotation equations would be rewritten as:

$$x_{i+1} = x_i - \sigma_i y_i 4^{-i} \tag{2.72}$$

$$y_{i+1} = y_i + \sigma_i x_i 4^{-i} \tag{2.73}$$

$$z_{i+1} = z_i - \sigma_i \arctan 4^{-i} \tag{2.74}$$

where $\sigma_i \in \{-2, -1, 0, 1, 2\}$. In a radix-4 implementation, the number of iterations required for the CORDIC algorithm are reduced by half. Two problems arise from the use of this new radix representation. First, the direction selection function for both the rotation and vectoring mode become quite complex because of the extended σ_i set of ATR digits. Second, this digit set will lead to a variable scaling factor.

Several solutions have been devised to both these problems, and can be found in [24] [25] [26] [27] [28].

2.4 Convergence

When considering a particular variation of the CORDIC algorithm, it is important to ensure the correctness of the result and determine its limitations in terms of precision. Obviously, the precision of the result is determined by the width of the internal data paths. In the case of circular CORDIC, precision is determined by the maximum shift parameter $S(N-1)$ and the smallest rotation angle $\alpha_{N-1} = \arctan(2^{-S(N-1)})$.

Convergence in both the rotation mode and the vectoring mode must also be ensured in order to obtain the correct result. In rotation mode, the z data path is being driven to zero, and there has to be enough angles remaining in subsequent iterations in order to bring the value of z within α_{N-1} of zero. Thus, there exists a set $\{\sigma_j\}$ where $j = i, i+1, \ldots, N-1$ such that:

$$\left| z_i - \sum_{j=i}^{N-1} \sigma_j \alpha_j \right| \leq \alpha_{N-1} \tag{2.75}$$

From this, we define the *range of convergence* which determines the angles that the CORDIC algorithm will be able to process in order to produce correct results. From Equation 2.75 it is easily shown that the range of convergence is given by:

$$|z_{in}| \leq \sum_{i=0}^{N-1} \alpha_i + \alpha_{N-1} \tag{2.76}$$

When using conventional CORDIC, the shift sequence is $\{S(i) = i\}$, that is $i = 0, 1, \ldots, N-1$. The range of convergence is roughly $[-1.7433, +1.7433]$ for $N = 16$. Two possible solutions exist in order to expand the range to $[-\pi, +\pi]$. The first solution is to perform pre-rotation by an angle of $\frac{\pi}{2}$. In this case, the pre-rotation equations are given by:

$$x_0' = -\sigma x_0 \tag{2.77}$$

$$y_0' = \sigma y_0 \tag{2.78}$$

$$z_0' = z_0 + \sigma\frac{\pi}{2} \tag{2.79}$$

where $\sigma = +1$ if $y < 0$, -1 otherwise. The second solution is to perform an initial rotation of either π or 0, as in the following equations:

$$x_0' = \sigma x_0 \tag{2.80}$$

$$y_0' = \sigma y_0 \tag{2.81}$$

$$z_0' = \begin{cases} z_0 & \text{if } \sigma = 1, \\ z_0 - \pi & \text{if } \sigma = -1 \end{cases} \tag{2.82}$$

where $\sigma = +1$ if $x < 0$, -1 otherwise. From Equation 2.76, it is now possible to derive the *convergence criterion* for the CORDIC process. This criterion imposes a specific order on the angle shift sequence, and is given by:

$$|\alpha_i| \leq \sum_{j=i+1}^{N-1} \alpha_j + \alpha_{N-1} \tag{2.83}$$

Similarly, the range of convergence in vectoring mode is related to the x and y input and is given by:

$$\left|\arctan\left(\frac{y_{in}}{x_{in}}\right)\right| \leq \sum_{i=0}^{N-1} \alpha_i + \alpha_{N-1} \tag{2.84}$$

2.5 Error Analysis

The finite nature of the CORDIC algorithm leads to errors in terms of precision when performing rotation and vectoring operations on the input data. The ideal CORDIC process data paths would have an infinite number of bits and the number of iterations would also be infinite. It is useful to use this theoretical model in order to derive the error bounds for a particular hardware implementation. An in-depth discussion of the various errors related to the CORDIC algorithm can be found in [29] and [30], and only a brief overview will be provided. There are four main sources of error in the CORDIC process:

1. Input value rounding error
2. Angle approximation error
3. Iteration rounding error
4. Scaling factor compensation error

The input value rounding error occurs when the data path width is less than the input data width. Since the CORDIC process induces magnitude growth in the vector being rotated, the input value rounding error is also scaled as it propagates to the outputs.

The second type of error is due to the fact that the total rotation angle θ is approximated by a linear combination of a limited set of ATR angles such that:

$$\theta = \sum_{i=0}^{N-1} \sigma_i \alpha_i + \delta \tag{2.85}$$

where δ is the error due to the angle approximation. In order to minimize this error, it is important to make α_{N-1} as small as possible. However, the width of the data path impose a lower bound on the value of $\alpha_{(N-1)}$. Since $\alpha_{(N-1)} = \arctan 2^{(N-1)} \approx 2^{(N-1)}$, this angle will be equal to the smallest representable value in the z data path.

The third type of error is due to the limited precision arithmetic of the shift and add operations within the pseudo-rotation equations for the x and y data paths. Rounding errors will occur every time a shifted operand is added (or subtracted) from a non-shifted operand. Once again, the errors due to rounding will incur growth from the moment they appear up to the end of the CORDIC process. Defining ϵ as the machine accuracy and b as the number of bits for the x and y data paths, we have

$\epsilon = 2^{-b-1}$ in the case of rounding and $\epsilon = 2^{-b}$ in the case of truncation. Thus, for n iterations, the accumulated error $f(n)$ due to rounding is bounded by:

$$|f(n)| \leq n\epsilon \tag{2.86}$$

Note that the scaling factor contribution has been neglected in this case.

The final type of error is due to the post-scaling multiplication of the inverse scaling factor with the final x and y values. Since the inverse scaling factor is also rounded, it has an associated error ΔK^{-1}. Multiplication by the inverse scaling factor will be of the form:

$$x' + \Delta x' = (x + \Delta x) \cdot (K^{-1} + \Delta K^{-1}) \tag{2.87}$$

Using simple averaging errors, this error may be represented by:

$$\Delta x' = x\Delta K^{-1} + K^{-1}\Delta x \tag{2.88}$$

2.6 Implementations on FPGA

In recent years, there has been increasing interest in FPGA implementations of the CORDIC algorithm. This is due in part to the trend back to hardware for Digital Signal Processing applications and the increase in logic density of modern programmable devices. Andraka [1] was one of the first to provide a practical overview of CORDIC algorithms specifically for FPGAs. The architectures discussed in the paper examined two different aspects for implementing the algorithm. In the first, an architecture could be either *iterative* or *pipelined*. The second aspect considers *bit-serial* and *bit-parallel* implementations. These different approaches may be combined together in a number of ways, and selecting the proper architecture for a particular application is a trade off between speed and area. The paper also discusses how these architectures are implemented at the Control Logic Block level for some specific devices. In order to optimize the performance of the CORDIC algorithm on an FPGA, a designer must carefully decide how to use the available resources in the most efficient way; this is a very different approach compared to ASIC design. There have also been other similar studies [31] [32] [33] [34] . Some of these studies examined high-radix implementations [32] [34] and the use of Redundant Number System representations (RNS) [31] [33]. It is interesting to note that in [33], the authors conclude that an RNS implementation is not suitable for FPGA design. They stipulate that operators needed for the RNS implementation require a 4 to 5 times larger area than their conventional counterpart. Furthermore the speed advantages that are gained in a full custom design are lost in the FPGA design due to longer routing delays.

2.7 Summary

This chapter presented the basic CORDIC algorithm, along with the unified CORDIC algorithm capable of performing vector rotations for the circular, linear and hyberbolic coordinate systems. An overview of the most important ideas in order to speed up and/or simplify the algorithm were also presented. In the next chapter, the characteristics of the target FPGA will be examined in order to determine which type of CORDIC variant will be suitable for implementation on the chosen platform.

CHAPTER 3: DESIGN CONSIDERATIONS

In order to design CORDIC processors optimized for a certain type of application, an understanding of the physical characteristics of the target platform is essential. In this particular case, the Field Programmable Gate Array (FPGA) will be used for the various processor implementations. The available resources will determine what type of algorithm and ultimately, what type of architecture, are most suitable for implementation on this particular platform.

3.1 FPGA Architecture

The basic Field Programmable Gate Array (FPGA) is essentially an array of Configurable Logic Blocks (CLB). Each CLB contains programmable function generators and associated memory elements. They may be combined via wiring resources called the *routing channel.* The routing channel contains wires that run in both horizontal and vertical directions. Programmable elements allow for wire interconnects. Input/Output (I/O) ports are also accessible through the routing channel. In addition to these basic elements, an FPGA might contain other dedicated resources such as large blocks of RAM or built-in multipliers. For this project, the target platform is the Xilinx Virtex-II. Although a specific device was chosen, the proposed CORDIC processors are also adaptable for other programmable logic devices.

Following is a list of dedicated resources that are available on the Virtex-II:

- Programmable I/Os
- Digital Clock Manager blocks (DCMs)
- Dual Port Block Select RAM (18Kbit)

- Embedded 18x18 Multiplier blocks
- Control Logic Blocks (CLBs)

The CLB is the top level logic component and is made up of four slices. Each slice contains two logic cells, for a total of eight distinct logic cells per CLB. The logic elements available in a slice is of particular interest, since they will become the basic building blocks of an optimized design. Each Virtex-II slice contains the following:

- 2 Function Generators (FG)
- 2 Storage Elements
- Arithmetic Logic Gates
- Large Multiplexers
- Fast Carry Look-Ahead chain
- Horizontal Cascade Chain (OR gate)

Figure 3.1 shows further details for a single logic cell and shows how these different elements are connected together. The most important logic element available in a cell is the Function Generator (FG). It can be configured as a Look-Up Table (LUT) that implements a 4:1 logic function. A second option is to use the FG as a 16-bit Variable Tap Shift Register. Finally, it can also be configured as 16-bit SelectRAM memory. It is also important to note that two 16-bit RAM resources can be combined together in order to provide dual port 16-bit RAM with one dedicated read/write port and a second read-only port.

3.2 Number System and Arithmetic for FPGA

As described in Chapter 2, considerable research has been conducted on the use of high-radix and redundant number systems for the CORDIC algorithm. The

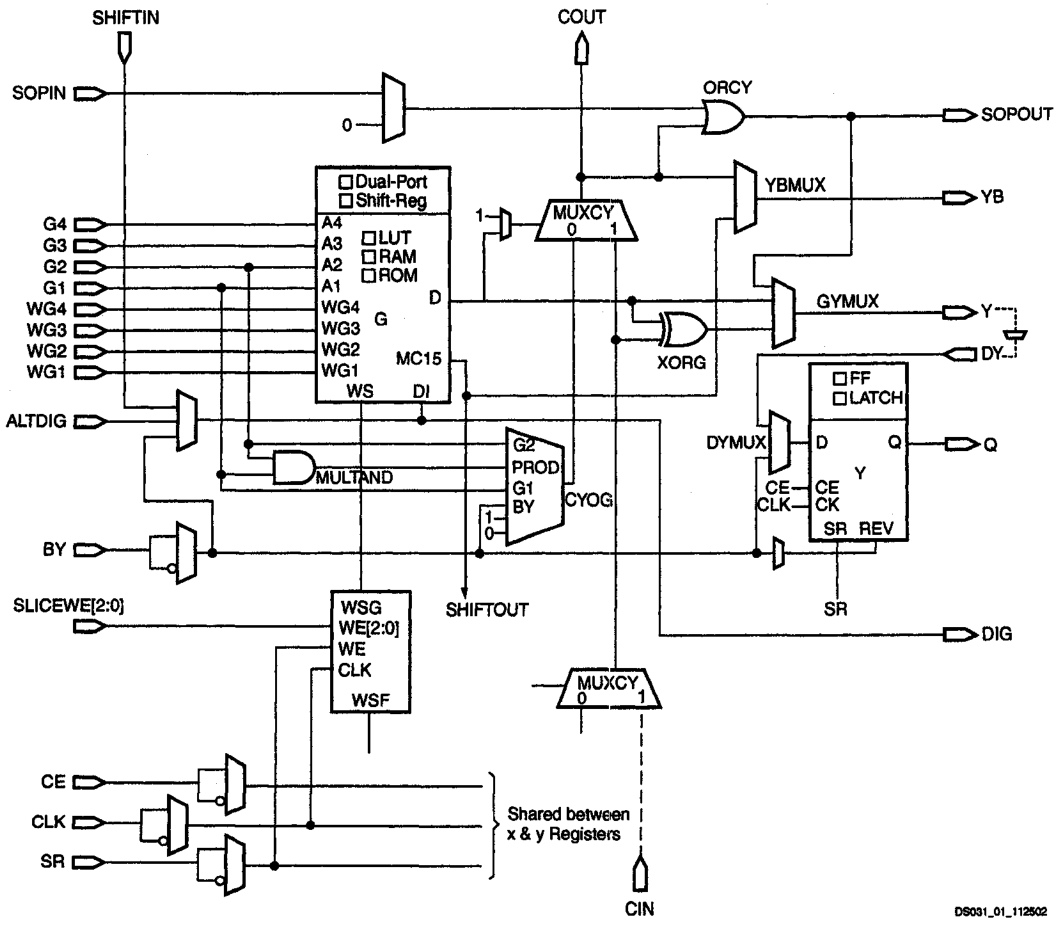


Figure 3.1: Virtex II Logic Cell

motivation for using number systems other than two's complement is to reduce the carry propagation time of additions in order to increase throughput. In a signed digit representation, every digit requires two bits, effectively doubling the data path. Furthermore, use of these non-conventional systems leads to a non-constant scaling factor. Studies have shown that advantages gained in an ASIC design by using redundant number systems are lost when implemented on an FPGA because of the increase in number of logic blocks and routing delays [31] [33]. Use of higher-radix number systems was also found unsuitable for FPGA implementations [34].

The Fast Carry Logic available on the Virtex-II is intended for the two's complement number system, and allows for very fast arithmetic operations. These operations are carried out in a bit-parallel fashion, meaning that all the bits of the result are calculated at once by allowing the carry signal to propagate from the LSB to the MSB. Bit-serial arithmetic is an alternative way of performing such operations. Each operation is carried-out one bit at a time by reusing the same hardware. Although this is not as efficient in terms of speed, it does provide the potential for significant savings in terms of both logic cells and routing resources. Consider for example the storage elements included in a slice on the Virtex-II device. When a bit-parallel approach is used, only the 2 memory elements are usable for storage, meaning that a 16-bit register will require 8 slices for storage. In the bit-serial case, the entire register can fit in one LUT configured as a 16-bit shift register, requiring only half a slice. The 16 bit register requires virtually no wiring: only the input, output and clock lines are required. The drawback however is that a controller is required in order to perform bit-serial operations. Also note that in lower grade programmable devices that do not have fast carry-logic, the gap in performance between a bit-serial design and a bit-parallel design will be even smaller.

3.3 Design Goals

The main focus for this project is to provide CORDIC implementations suitable for System on Chip (SoC) that will fit on a single FPGA. As already mentioned, many applications will require a number of CORDIC processing elements. Thus, the CORDIC implementations are required to be compact and of low complexity. The fully pipelined bit-parallel implementation will obviously have the highest throughput compared to other approaches, but will require the most logic elements and routing resources. In applications such as audio processing that do not require extremely high data rates, such an approach will take up unnecessary real-estate. Providing alternative architectures and characterizing their area/performance trade offs provides designers with a wider range of options that will allow for the tightest possible integration of their design. Following are some of the important characteristics to examine when determining the most suitable CORDIC implementation for a particular application.

3.3.1 Latency

Latency is the amount of time required for the input to propagate to the output. Ideally, this amount of time is to be kept to a minimum. An iterative algorithm such as CORDIC has an inherently high latency as opposed to other methods such as look-up tables for calculating values of trigonometric functions. Some possible solutions to reducing latency are simplification of the algorithm and exploiting parallelism wherever possible.

3.3.2 Throughput

Throughput is the amount of data produced per unit of time. Pipelining is the most common way of increasing throughput. In order to have high throughput, the maximum delay between pipeline stages must be kept to an absolute minimum. In an iterative architecture, throughput is determined by the latency of the algorithm, as well as the delay required for each iteration. Insuring the algorithm can complete in as few iterations as possible will maximize throughput in the case of an iterative architecture.

3.3.3 *Accuracy and Error*

Accuracy and allowable error are highly dependent on the application being targeted. Obviously, the automatic generation of a CORDIC PE architecture must allow for a wide range of accuracy requirements. Floating point implementations also allow for a greater dynamic range than fixed point implementations. For the CORDIC algorithm, one extra bit of accuracy is obtained for every iteration.

3.3.4 *Area*

The area occupied by a CORDIC PE becomes a crucial design consideration if the PE is to be used as part of a larger design, such as systolic array or a system on chip (SoC). Architectures that use the minimum number of logic blocks and minimize the number of required interconnects will ensure that the PE does not take up too much valuable real estate on the FPGA.

3.3.5 *Power Consumption*

With the increase in integration, larger FPGAs and portable systems, power consumption is becoming an important issue. Obviously, implementation must be as simple as possible in order to reduce unnecessary power consumption. Cutting down on long interconnects and minimizing fanout are crucial in bringing down power consumption in an FPGA. This is usually accomplished by inserting memory elements to break up the interconnect and reduce the fanout. Reducing unnecessary glitches with the use of appropriate state-machine memory elements will also help to reduce power dissipation. Finally, power consumption can be reduced by preventing unnecessary switching and turn off parts of a circuit when they are not required.

3.3.6 Scalability

Scalability is the ability of a system to be extended in order to meet increasing requirements while minimizing impacts on performance. In the case of the CORDIC PE, scaling will imply that various bitwidths must be supported in order to be suitable for a wide range of applications. It is important to note that the performance and complexity of the CORDIC algorithm is directly proportional to its required accuracy.

3.4 Architectural Styles

The internal structure of FPGAs in general lends itself to a certain number of architectural styles. For this work, four broad categories will be examined, namely bit-parallel iterative, bit-parallel pipelined, bit-serial iterative and bit-serial pipelined.

3.4.1 Bit-Parallel Iterative Architecture

The bit-parallel iterative structure stems directly from the CORDIC pseudo-rotation equations, and is shown in Figure 3.2. Initial values are loaded into the appropriate registers via the input multiplexers. Two barrel shifters are required in order to provide the shifted values for $x2^{-i}$ and $y2^{-i}$ for the cross-additions. The ATR constants for the z data path are stored in ROM and the outputs of the x, y and z data paths are fed back into their respective registers. The direction of rotation σ_i is determined by the sign of the z register in rotation mode or the sign of the y register in vectoring mode. A controller (omitted on the figure) is required in order to provide the ROM addresses and and shift parameters to the barrel shifters.

This design is relatively compact and straight forward. Implementing efficient barrel-shifters is an important issue for this architecture in order to maximize the use of available logic elements. For bit-parallel implementations, the latency is equal to the number of iterations N_{iter}. If F_{clock} is the maximum operating frequency of the

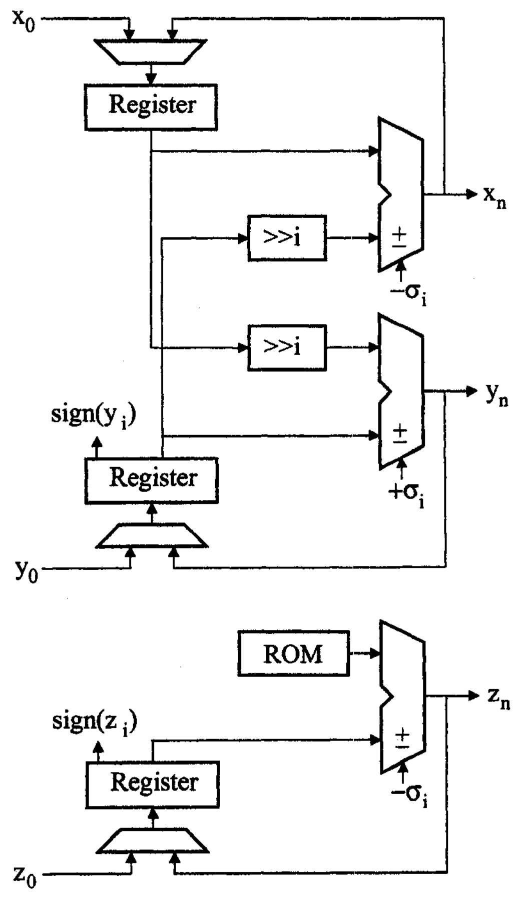


Figure 3.2: Bit-Parallel Iterative CORDIC Structure [1]

iterative processor, then the throughput R will be:

$$R = \frac{F_{clock}}{N_{iter}} \tag{3.1}$$

3.4.2 Bit-Parallel Pipelined Architecture

By unrolling the iterative architecture, a purely combinational structure as depicted in Figure 3.3 can be obtained. This type of structure does not require barrel shifters since the shift for the cross adders are provided by the wiring. The

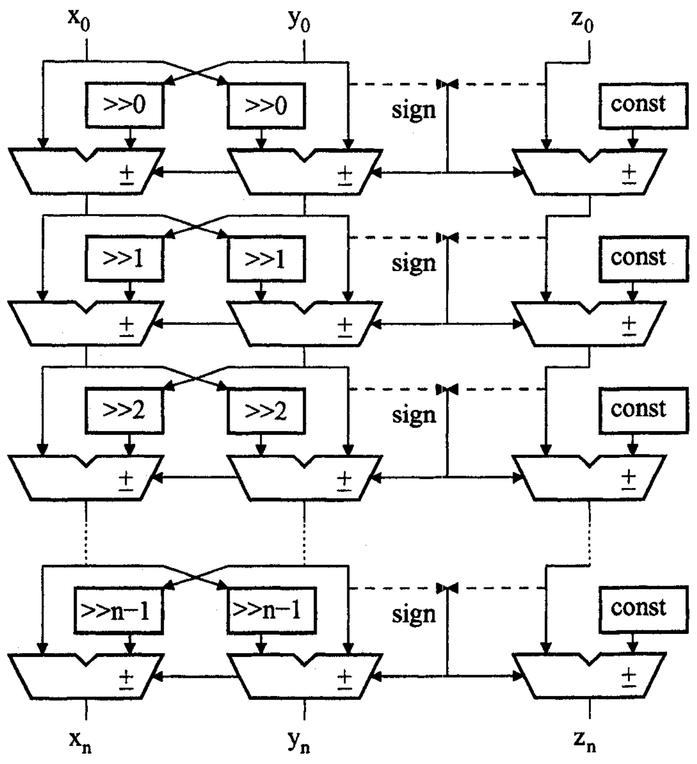


Figure 3.3: Unrolled Bit-Parallel CORDIC Processor [1]

ROM is also eliminated, as each ATR constant required in the z data path can now be hardwired. The suppression of the barrel shifters and ROM also leads to the elimination of the controller. The design is easily pipelined by adding registers after each adder/subtracter. In an FPGA, each logic cell has an associated memory element. If only the logic cell are used, the memory elements are lost. Thus, pipelining in an FPGA comes at no extra cost, and should be exploited wherever possible. In this case, the throughput R is:

$$R = F_{clock} \tag{3.2}$$

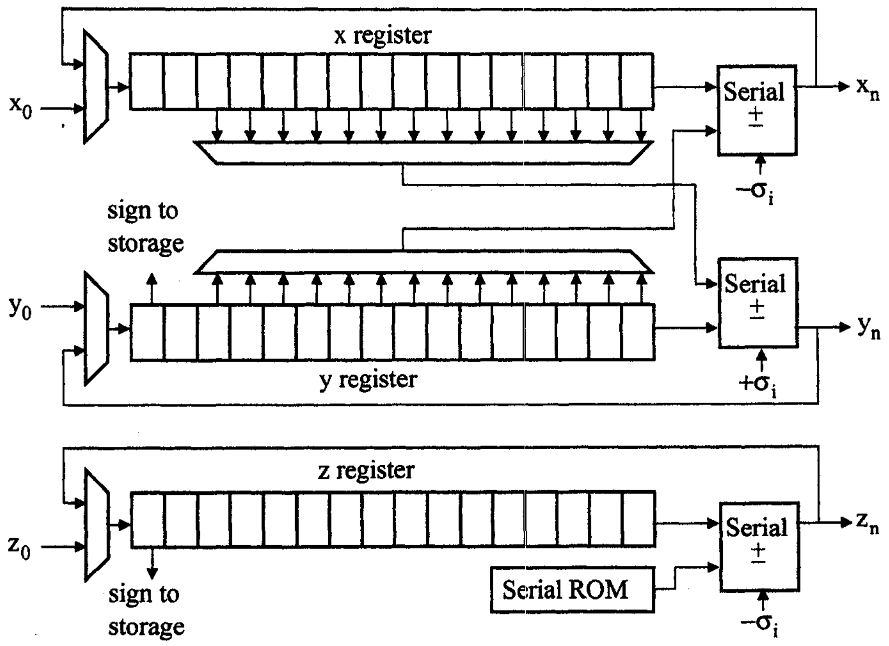


Figure 3.4: Bit-Serial Iterative CORDIC [1]

3.4.3 Bit-Serial Iterative Architecture

The bit-serial iterative architecture is depicted in Figure 3.4. By processing the bits one at a time, there is no delay associated with carry propagation and a lot less hardware is required for the adders. In this manner, the processor can have a higher clock rate. However, the wide multiplexer can present problems for implementations in certain FPGAs. The controller required for such an architecture is more complex than for a bit-parallel design, since the bit-serial addition has to be properly sequenced. This usually requires insertion of a few delay memory elements in order to properly align the control signals and reset the serial adder carry storage. Denoting B_{eff} as the number of effective bits in the register and B_{delay} as the number of delay bits, the throughput of the bit-serial processor will be:

$$R = \frac{F_{clock}}{N_{iter} \cdot (B_{eff} + B_{delay})} \tag{3.3}$$

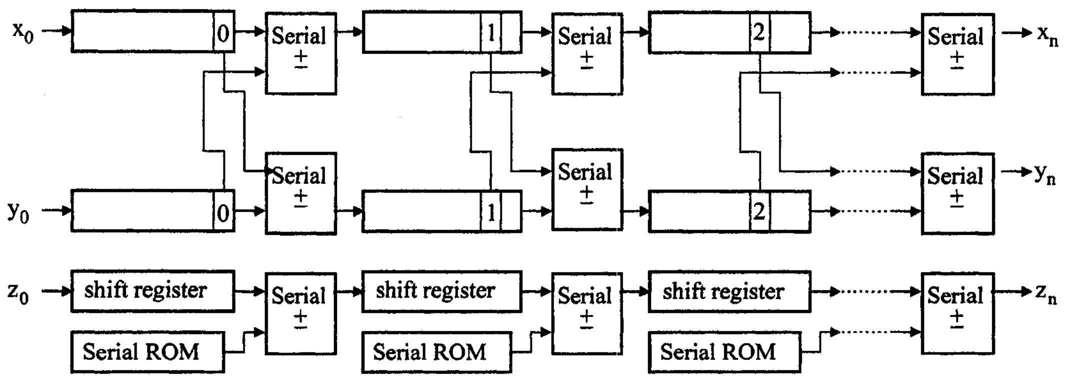


Figure 3.5: Bit-Serial Pipelined CORDIC

3.4.4 Bit-Serial Pipelined Architecture

The bit-serial iterative structure can also be unrolled into a pipelined architecture, as shown in Figure 3.5. The serial ROMs will become one-dimensional as opposed to the two-dimensional ROM required for the iterative architecture. Also, the wide multiplexer is no longer required since all registers contain a tap for the corresponding shift value. As the bits are fed continuously into the next register, some mechanism for sign recording and sign extension is also required. Some form of sequencing is necessary in order to handle sign recording, sign extension, serial ROM addressing and serial addition. The throughput for such an architecture will be:

$$R = \frac{F_{clock}}{(B_{eff} + B_{delay})} \tag{3.4}$$

3.5 Xilinx Intellectual Property Core

Xilinx already provides an Intellectual Property (IP) CORDIC implementation as part of their LogiCORE product line. These cores can be built automatically through the Xilinx Core Generator interface according to user supplied parameters. The implementations are specially tailored to Xilinx FPGAs and utilize the available logic elements in an extremely efficient way. The version used during the course of this project was the CORDIC v2.0. In this version, only the bit-parallel iterative and pipelined architectures can be generated. These implementations use the original CORDIC equations with a very fast and area efficient constant coefficient multiplier (CCM) for scaling factor compensation. The generated CORDIC cores will be used as benchmarks for the processing elements proposed in this work.

3.6 Summary

This chapter outlined the key characteristics of the Xilinx Virtex-II, which will be the target platform used for this project. The design goals for the CORDIC Processing Elements were also outlined. Finally, the four broad architectural styles that will form the basis for the CORDIC PEs have been examined, namely the bit-parallel iterative, bit-parallel pipelined, bit-serial iterative, and bit-serial pipelined architectures. In the next chapter, these design considerations will be used in order to select a suitable CORDIC algorithm for implementation on the Virtex-II device.

CHAPTER 4: ALGORITHM SELECTION

The original algorithm proposed by Volder [2] is the most common way of implementing CORDIC in hardware. Since each pseudo-rotation is dependent on the previous result, the rotation stages offer little parallelism opportunities. Hybrid radix representation and rotation prediction provide an alternative, but are only valid in the case of rotation. Decreasing latency in a pipelined architecture thus becomes a matter of implementing an extremely efficient Constant Coefficient Multiplier, as is the case for the Xilinx CORDIC IP Core. However, the iterative structure and bit-serial architectures have different characteristics, and the most efficient approach for a parallel pipelined architecture might not be suitable in these other circumstances. In this chapter, two potential candidates will be compared and the most suitable algorithm will be selected in order to implement the CORDIC PE on an FPGA.

4.1 Double Shift CORDIC

The double shift CORDIC technique was first discussed in [35]. Instead of performing a single shift on the x_i and y_i variables, two different shift values are obtained and are either added or subtracted. This allows added flexibility in the angles that are used for each iteration, and angle shift parameters are chosen in such a way that the scaling factor is steered toward a simple sum of signed powers of two. Originally, parameter optimization was considered in order to produce a general purpose ALU capable of performing rotations in all three coordinate systems, i.e. linear, circular and hyperbolic. Similar results were also presented in [36]. In [37], sets of shift parameters are presented that are optimized only for the circular case. Furthermore, [36] and [37] consider scaling factors that are simple sums (or differences) of signed powers of two containing 2 terms, whereas [35] only considers a scaling factor of 0.5 for the circular coordinate system.

The modified iteration equations for the Double Shift CORDIC algorithm are as follows:

$$x_{i+1} = x_i - \sigma_i(2^{-S(i)} + \eta(i)2^{-S'(i)})y_i \tag{4.1}$$

$$y_{i+1} = y_i + \sigma_i(2^{-S(i)} + \eta(i)2^{-S'(i)})x_i \tag{4.2}$$

$$z_{i+1} = z_i - \sigma_i\alpha_i \tag{4.3}$$

where:

$$\alpha_i = \arctan(2^{-S(i)} + \eta(i)2^{-S'(i)}) \tag{4.4}$$

Notice that each angle is now described by three parameters, $S(i)$, $S'(i)$ and $\eta(i)$. The parameter $S'(i)$ is a second shift factor, and $\eta(i) \in \{-1, 0, 1\}$. Thus, the equations for the x and y data paths now contain two shifting operations and two addition operations. The z data path is unaffected since the α_i constants are pre-calculated. The following constraints are also imposed on the shift parameters:

1. Angle set:

$$\alpha_i = \arctan(2^{-S(i)} + \eta_i 2^{-S'(i)}) \tag{4.5}$$

   where:

$$\eta_i \in \{-1, 0, 1\}$$
$$S(i), S'(i) \in \{0, 1, \ldots, N\}$$
$$S'(i) > S(i)$$

2. Representation range:

$$\sum_{i=0}^{N} \alpha_i \geq \frac{\pi}{2} \tag{4.6}$$

3. Angle convergence:

$$\alpha_i \leq \alpha_N + \sum_{j=i+1}^{N} \alpha_j \text{ for } i = 0, \ldots, N-1 \tag{4.7}$$

4. Angle precision:

$$\alpha_N = \arctan(2^{-N}) \tag{4.8}$$

5. Inverse scaling factor composition:

$$K^{-1} = \prod_{i=0}^{N} \cos\alpha_i \approx \sum_{j=0}^{D-1} \lambda_j 2^{-T_j} \tag{4.9}$$

$$\lambda_j \in \{-1, 0, +1\} \tag{4.10}$$

$$T_j \in \{0, \ldots, N\} \tag{4.11}$$

The resulting scaling factor is given by:

$$K = \prod_{i=0}^{N-1} \sqrt{1 + 2^{-2S(i)} + \eta_i 2^{-S(i)-S'(i)+1} + \eta_i^2 2^{-2S'(i)}} \tag{4.12}$$

The derivation of the scaling factor is given in Appendix A.

In (4.9), D indicates the maximum number of signed digits. The above constraints are required in order to preserve the convergence properties and the accuracy of the CORDIC process. Finding shift parameters that will yield a simple inverse scaling factor is an optimization problem where the total number of iterations has to be kept to a minimum and the number of iterations where $\eta_i \neq 0$ also has to be minimum. Furthermore, limits have to be imposed on how many double shift iterations and how many total iterations that can be used as the ATR. Note also that according to [38], this problem is NP-complete.

In order to further investigate the properties of the Double Shift CORDIC and its usefulness for hardware implementations, a study was conducted as part of this thesis and published in [39]. In this study, a more flexible solution in terms of optimization is proposed for both recursive and pipelined architectures. Particularly, the use of modified CORDIC ATR sequences was carefully considered against the basic CORDIC algorithm originally proposed by Volder [2] and Double Shift CORDIC shift sequences presented in [40] in order to ensure that benefits are achieved in terms of area, power, and throughput. The proposed approach was to allow more flexibility in the number of terms that can appear in the scaling factor. A search algorithm was developed in Matlab in order to find double shift CORDIC sequences that would yield scaling factórs with a specified number of signed digits. In order to restrict the search space, repetition of the same angles in the angle set is not allowed. However, the search space stills remains quite overwhelming, containing 3.373825×10^{49} different possibilities for the 32 bit precision ATR. The input to the search function is the maximum shift value, the maximum number of double shift stages allowed, the depth of the search and the maximum number of signed digits that can be present in the scaling factor. The depth of the search takes advantage of the fact that angles with smaller shift values will have the most impact on changing the value of the scaling factor. In contrast to the previous approaches where a series of nested "for loops" were used, this program uses modulo counters to perform the loops. In this manner, the program does not have to be rewritten for different bit sizes. For each candidate angle sequence, the convergence requirements are first tested. If the angle set meets the convergence criterion, its corresponding scaling factor is recoded into Canonical Signed Digit (CSD) format, and the program checks whether it contains the allowable number of signed digits or fewer. If the angle set meets all these requirements, it is added to the list of acceptable angle sequences.

Hardware requirements for the original CORDIC shift sequence, the double shift sequences found in [40] and the proposed modified double shift CORDIC were tabulated and compared for bit widths of 12, 16, 18, 24 and 32 bits. The study clearly showed that no benefits were derived in terms of latency and amount of hardware for the shift sequences found in [40] when compared to the original CORDIC shift sequence. It was also shown that the modified double shift CORDIC did outperform the original CORDIC and the shift sequences found in [40].

One major disadvantage of the Double Shift approach is the complexity of the search algorithm required to produce valid and efficient ATR shift sequences. Furthermore, this search process has to be conducted each time we want to generate an ATR sequence for a different data width. Note that for a fixed bit width, this ATR sequence does not change. The second disadvantage is that two different shifted operands have to be generated for each pseudo-rotation equation for the x and y data path. This leads to the requirement for either four barrel-shifters for an iterative architecture or to split the double-shift iterations into two with an accumulator and increased control logic. In a bit-serial design, this means that we need double tap registers in a pipelined implementation or two wide multiplexers for each data path in an iterative implementation.

4.2 Compensated CORDIC

The compensated CORDIC algorithm is based on a two step process where simple correction iterations are inserted in order to force the scaling factor to 2. First, the conventional CORDIC iteration is performed:

$$x'_{i+1} = x_i - \sigma_i y_i 2^{-i} \tag{4.13}$$

$$y'_{i+1} = y_i + \sigma_i x_i 2^{-i} \tag{4.14}$$

$$z'_{i+1} = z_i - \sigma_i \alpha_i \tag{4.15}$$

$$k'_{i+1} = k_i(1 + 2^{-2i})^{\frac{1}{2}} \tag{4.16}$$

where $\sigma_i = -1$ if $z_i < 0, +1$ otherwise (Rotation Mode)

followed by a magnitude correction iteration which does not perform any rotation of the vector:

$$x_{i+1} = x'_{i+1} - \eta_i x'_{i+1} 2^{-i} \tag{4.17}$$

$$y_{i+1} = y'_{i+1} + \eta_i y'_{i+1} 2^{-i} \tag{4.18}$$

$$z_{i+1} = z'_{i+1} \tag{4.19}$$

$$k_{i+1} = k'_{i+1}\sqrt{1 + \eta_i 2^{-i+1} + \eta_i^2 2^{-2i}} \tag{4.20}$$

where $\eta_i = +1$ if $k_i < 0, -1$ otherwise

The rotation and compensation can be combined into a single set of equations:

$$x_{i+1} = x_i - \sigma_i y_i 2^{-i} + \eta_i x_i 2^{-i} + \sigma_i \eta_i y_i 2^{-2i} \tag{4.21}$$

$$y_{i+1} = y_i + \sigma_i x_i 2^{-i} + \eta_i y_i 2^{-i} + \sigma_i \eta_i x_i 2^{-2i} \tag{4.22}$$

With these compensated pseudo-rotations, the values for η_i are selected at each step in order to steer the scaling factor to 2. However, these equations do not bring about any real benefits in terms of reducing the amount of processing required for the CORDIC algorithm, since they contain 4 operands with different shift values. In order to simplify the compensated pseudo-rotations, the last term in each equation can be dropped, leading to the following equations:

$$x_{i+1} = x_i - \sigma_i y_i 2^{-S(i)} + \eta_i x_i 2^{-S(i)} \tag{4.23}$$

$$y_{i+1} = y_i + \sigma_i x_i 2^{-S(i)} + \eta_i y_i 2^{-S(i)} \tag{4.24}$$

$$z_{i+1} = z_i - \sigma_i \alpha_i \tag{4.25}$$

Note that the shift factors 2^{-i} have been replaced by $2^{-S(i)}$. The ATR angles are thus given by:

$$\alpha_i = \arctan\left(\frac{1}{\eta_i + 2^{S(i)}}\right) \tag{4.26}$$

and the scaling factor is given by:

$$K = \sum_{i=0}^{N-1} \sqrt{1 + 2^{-2S(i)} + \eta_i 2^{-S(i)+1} + \eta_i^2 2^{-2S(i)}} \tag{4.27}$$

Appendix A contains all the details for the derivation of the ATR angles and scaling factor. Significant benefits are gained from these modified compensated CORDIC iterations. The equations now contain only 3 operands, just like the double shift CORDIC. However, the two same shifted terms (i.e. $x_i 2^{-S(i)}$ and $y_i 2^{-S(i)}$) appear in both equations, whereas the double shift algorithm required four different shift terms.

While these equations are simpler, the ATR radix has been modified and convergence is no longer guaranteed for $S(i) = 0, 1, 2, 3, ..., N$. Certain shift parameters

$S(i)$ will need to be used more than once in order to ensure convergence. A heuristic search was used in [5] in order to determine the appropriate shift sequence and corresponding values for η_i. The search algorithm was developed in order to minimize the number of operations required to insure both angle convergence to zero and scaling factor convergence to a value of 2. The results for a maximum shift of 31 was given in [5]. Extension to a maximum shift value of 34 was trivial, and the resulting shift sequence is presented in table 4.1. The Compensated CORDIC leads to two other significant advantages over its Double Shift counterpart: 1) the search algorithm is much simpler and 2) the shift sequence obtained from the search can be used for the entire range of data width contained in the list. For example, if a maximum shift of 20 is required, iterations 0 through 22 are used.

i	$S(i)$	η_i	K^{-1}	$\log_2 E$	i	$S(i)$	η_i	K^{-1}	$\log_2 E$
0	0	0	0.70710678118655	-2.27155	19	18	1	0.50000031318151	-21.60650
1	1	0	0.63245553203368	-2.91642	20	18	0	0.50000031317788	-21.60651
2	2	0	0.61357199107790	-3.13832	21	19	0	0.50000031317697	-21.60652
3	3	1	0.54206153065211	-4.57135	22	20	0	0.50000031317674	-21.60652
4	4	1	0.50929518563202	-6.74930	23	21	1	0.50000007475807	-23.67320
5	5	0	0.50904668833221	-6.78839	24	22	0	0.50000007475805	-23.67320
6	6	1	0.50115589593831	-9.75677	25	23	1	0.50000001515340	-25.97578
7	6	0	0.50109473088091	-9.83521	26	24	0	0.50000001515340	-25.97578
8	7	0	0.50107943938330	-9.85550	27	25	1	0.50000000025224	-31.88447
9	8	0	0.50107561649432	-9.86062	28	26	0	0.50000000025224	-31.88447
10	9	1	0.50009791076158	-13.31817	29	27	0	0.50000000025224	-31.88447
11	10	0	0.50009767229648	-13.32169	30	28	0	0.50000000025224	-31.88447
12	11	0	0.50009761268020	-13.32257	31	29	0	0.50000000025224	-31.88447
13	12	0	0.50009759777613	-13.32279	32	30	0	0.50000000025224	-31.88447
14	13	1	0.50003655443256	-14.73959	33	31	1	0.50000000001941	-35.58433
15	14	1	0.50000603555478	-17.33808	34	32	0	0.50000000001941	-35.58433
16	15	0	0.50000603532195	-17.33814	35	33	0	0.50000000001941	-35.58433
17	16	0	0.50000603526374	-17.33815	36	34	0	0.50000000001941	-35.58433
18	17	1	0.50000222053498	-18.78066					

Table 4.1: Compensated CORDIC Shift Sequence and Properties

4.3 Algorithm Selection

The compensated CORDIC algorithm provides some clear advantages over the double shift algorithm, and will be the algorithm used for the FPGA implementations. Following are the 3 main reasons why the compensated CORDIC is well suited to FPGA implementation:

1. Simple Inverse Scaling Factor ($K^{-1} = 0.5$)
2. Occurrence of same shift factors in both x and y data path
3. No complicated search algorithm for ATR sequence

Having a simple scaling factor that only requires a shift operation will become crucial in a bit-serial implementation, where implementation of a bit-serial multiplier is awkward: a bit-serial multiplier requires twice as many clock cycles as a bit-serial adder, since it produces twice as many bits. Occurrence of the same shift factors will greatly simplify hardware, particularly in the iterative bit-parallel and both types of bit-serial structures. If a three operand adder is used in the bit-parallel iterative structure, only two barrel shifters will be required in the case of the compensated algorithm, whereas four barrel shifters would have been required in the case of the double shift algorithm. For bit-serial arithmetic, single-tap shift registers will be used instead of double tap shift registers. This also has an effect on the controller, since only a single sign extension signal will be required instead of two different signals.

4.4 Error Analysis

4.4.1 Angular Error Analysis

The first source of error to consider is the one introduced by the finite set of ATR angles. As such, the angle precision will be determined by the smallest ATR angle. The worst case angular error will occur when the second last iteration produces a vector with the exact angle of rotation required (or an angle of zero in the case of vectoring). The last iteration will then rotate the vector by the final angle α_{N-1}, inducing an error in the x and y component of the vector. Appendix B provides the details on the effects of this error. The result is that for b bit precision, the shift value for the smallest angle will have to be $S(N-1) = b$.

Table 4.2 shows how precision is improved with each subsequent iteration when considering angular error for 16-bit precision. Another interesting fact is that if the CORDIC algorithm is used only for vector magnitude calculation (i.e. vectoring without the need for accurate angular displacement), the number of iterations can be cut in half since the x data path has an improvement of 2 bits per iterations (except in cases where angles are repeated).

i	arctan(α_i)	x err(%)	affected	y err(%)	affected	z err	affected
		$1-\cos\alpha$	x bits	$\sin\alpha$	y bits		z bits
0	0.785398	29.289322	15	70.710678	16	0.785398	0
1	0.463648	10.557281	13	44.721360	15	0.463648	-1
2	0.244979	2.985750	11	24.253563	14	0.244979	-2
3	0.110657	0.611627	9	11.043153	13	0.110657	-3
4	0.058756	0.172563	7	5.872202	12	0.058756	-4
5	0.031240	0.048792	5	3.123475	11	0.031240	-5
6	0.015383	0.011832	3	1.538280	10	0.015383	-6
7	0.015624	0.012205	3	1.562309	10	0.015624	-6
8	0.007812	0.003052	1	0.781226	9	0.007812	-7
9	0.003906	0.000763	-1	0.390622	8	0.003906	-8
10	0.001949	0.000190	-3	0.194931	7	0.001949	-9
11	0.000977	0.000048	-5	0.097656	6	0.000977	-10
12	0.000488	0.000012	-7	0.048828	5	0.000488	-11
13	0.000244	0.000003	-9	0.024414	4	0.000244	-12
14	0.000122	0.000001	-11	0.012206	3	0.000122	-13
15	0.000061	0.000000	-13	0.006103	2	0.000061	-14
16	0.000031	0.000000	-15	0.003052	1	0.000031	-15
17	0.000015	0.000000	-17	0.001526	0	0.000015	-16

Table 4.2: Angular Error Analysis - 16 bit Inputs

4.4.2 Truncation Error Analysis

Since shifted operands are used with finite word length, rounding will produce further errors during the CORDIC process. In order to keep the hardware as simple as possible in the present implementations, it was decided to use truncation instead of rounding. The error introduced is given by the number of 1's that have been truncated. Thus, the maximum error is found by adding all the weights for the truncated bits. In compensated CORDIC, stages where $\eta_i \neq 0$ have two shifted operands, and will introduce more errors than stages that have only a single shifted operand. It is possible to pad the data path with an appropriate number of bits in order to allow for error accumulation. If E is the maximum accumulated error for the CORDIC process, then $log_2(E)$ bits will need to be padded. Tables B.1 and B.2 in Appendix B show truncation error details as well as the number of pad bits required for different numbers of iterations for both the original CORDIC algorithm and the compensated CORDIC.

4.4.3 Overflow Error Analysis

The x and y data path of the pseudo-rotation stages of the CORDIC PE perform simple fixed point shift and add operations. Because of this, the width of the data paths has to be adjusted in order to handle possible overflow conditions. Overflow conditions can result from the following sources: 1) growth from 2D vector rotations and 2) growth in magnitude induced by the pseudo-rotations themselves (i.e. the scaling factor). The worst case growth will occur when both x and y are at their absolute maximum, and the desired angle of rotation is $\pm\pi/4$. For example:

$$x' = x\cos\left(\frac{\pi}{4}\right) - y\sin\left(\frac{\pi}{4}\right) = 0 \tag{4.28}$$

$$y' = y\cos\left(\frac{\pi}{4}\right) - x\sin\left(\frac{\pi}{4}\right) \approx 1.4142 \cdot y \tag{4.29}$$

Growth resulting from the scaling factor occurs at every iteration of the compensated CORDIC algorithm, and reaches $K \approx 2$. Thus, the x and y data paths must be padded with a total of 2 extra bits in the MSB positions.

4.5 Data Format

The internal data path format is shown in Figure 4.1 for b bit inputs. Note that if a binary point is present in the inputs, it must be at the same position for both the x and y input. Contrary to the x and y data paths, the z data path does not

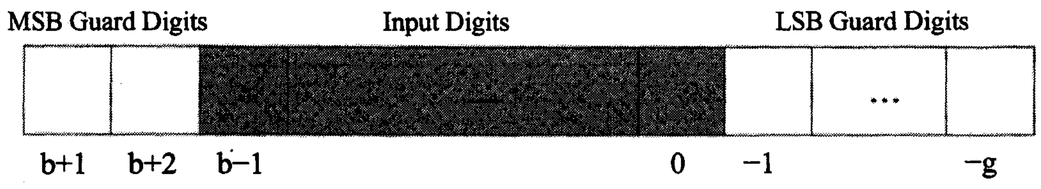


Figure 4.1: x and y Datapath Format

require MSB or LSB guard bits, since it deals with simple add and subtract operations without any shifting or induced scaling. It has a standardized fixed-point format in order to represent angles in the range of $[-\pi/2, +\pi/2]$. In order to represent this range, the z datapath will have 2 whole bits and $b-2$ fractional bits, which will allow for an effective range of $[-2, +2 - 2^{-b+2}]$.

4.6 Summary

The Compensated CORDIC algorithm was selected for the hardware implementation of the CORDIC PE. It provides a simple inverse scaling factor, has the same shift factors for both the x and y data paths and requires no complicated search algorithm for determining the ATR shift sequence. The characteristics of the Compensated CORDIC lead to significant savings in terms of hardware. In the next chapter, FPGA implementations of the Compensated CORDIC will be examined for the four different types of architectural styles.

CHAPTER 5: IMPLEMENTATIONS

The compensated CORDIC algorithm was chosen for the FPGA implementation of the CORDIC PE. There is a broad range of options when attempting to map the CORDIC algorithm into hardware. The architectural styles presented in Chapter 3 will be used and arithmetic operations will be adapted in such a way as to derive maximum benefits either in terms of speed or minimum area. One major advantage of the compensated CORDIC is that no multiplier is required for scaling factor compensation. Hence, the design of normal CORDIC stages and compensated stages, in addition to controllers, is all that is required to be examined.

5.1 Bit-Parallel Pipelined Implementation

The bit-parallel pipelined implementation has two different types of pipeline stages: 1) the regular CORDIC iteration stage ($\eta_i = 0$) and 2) the compensated CORDIC stage ($\eta_i = 1$). The regular CORDIC stage is shown in Figure 5.1 and is the one that was used in describing the basic architectural style in Chapter 3. In the

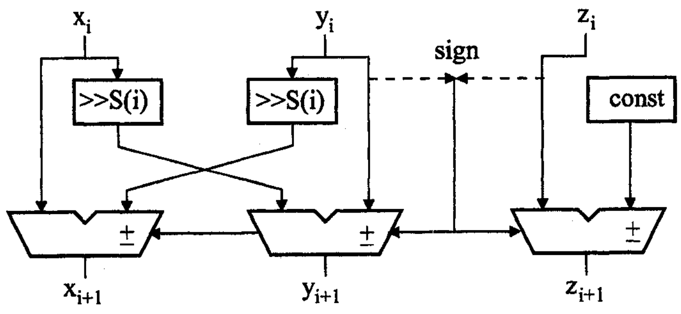


Figure 5.1: Regular CORDIC Pipeline Stage

case of the compensated stage, it is required to perform an operation such as:

$$R = A \pm B + C \tag{5.1}$$

There are two different options for performing this type of operation in an FPGA such as the Virtex-II. The first is to perform the operation in two steps, which will require one adder/subtracter followed by an adder (Figure 5.2). Typically, each bit-slice of a 2 operand adder requires 1 LUT and associated carry-logic. Thus, by cascading two w-bit adders, $2w$ LUTs (and associated carry-logic) will be required. The delay of the operation will be twice the delay of a w bit adder. The second

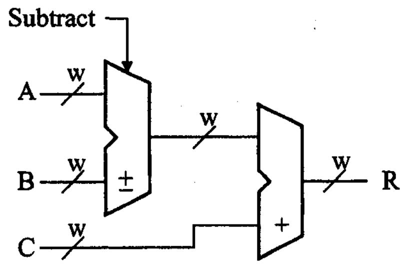


Figure 5.2: 3 Operand Adder/Subtracter

option, and the one that was used for the pipelined implementation, is to use a 3:2 compressor structure followed by an adder, as depicted in Figure 5.3. The compressor stage will have a delay equal to a single LUT, since there is no carry propagation. Following are the Boolean equations for calculating the *sum* and *carry* bits for both the sum and carry vectors from the a, b and c bits of the three operands:

$$d = b \oplus subtract \tag{5.2}$$

$$sum = a \oplus d \oplus c \tag{5.3}$$

$$carry = a \cdot d + a \cdot c + d \cdot c \tag{5.4}$$

Note that the equations for each *sum* and *carry* bit is dependant on 4 inputs, namely a, b, c and *subtract*, and these functions can be implemented in a single LUT. Once the sum and carry vectors have been generated after a delay of 1 LUT, these vectors are then used as inputs to a w-bit adder. The total delay is thus 1 LUT plus the delay of 1 w-bit adder. The drawback to this approach is that 2 LUT for each bit-slice is required (one for the sum, and one for the carry). Thus, the resources required to implement such a structure will be $3w$ LUTs, or 1/3 more hardware than the cascaded approach. Considering that the compensated stages on average make up less than 30% of the total number of stages, this increase is a small price to pay for the added performance gain. The resulting compensated pipelined stage is shown in Figure 5.4.

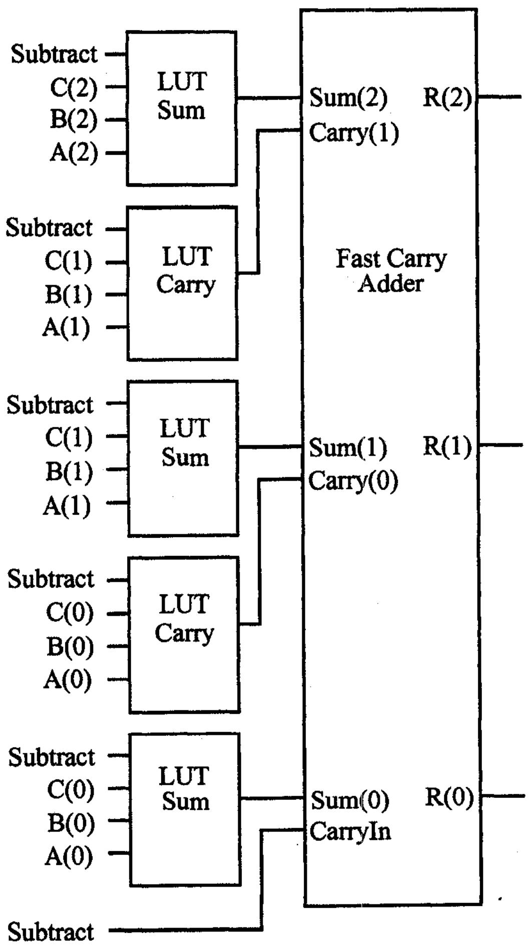


Figure 5.3: 3:2 Compressor and Adder

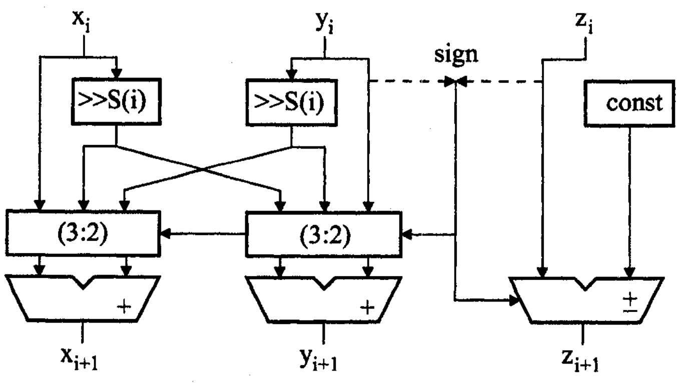


Figure 5.4: Compensated Pipeline Stage

5.2 Bit-Parallel Iterative Implementation

The bit-parallel iterative implementation of the CORDIC PE can be subdivided into three main parts (Figure 3.2), namely the controller, the ROM and the data paths. The data path is slightly different than the one used in a compensated CORDIC stage of the pipelined implementation, and the processing element can be seen in Figure 5.5. A cascaded arrangement of one adder and one adder/subtracter is used instead of the 3:2 compressor structure of the pipelined implementation. The reason for this is that there is now an extra decision variable, η_i, which prevents the logic function from fitting nicely into a 4 input LUT. The decision variable η_i and the shift values $S(i)$ are provided by the controller. The direction of rotation variable σ_i is equal to the sign bit of the z register for rotation mode or the complement of the y register sign bit in vectoring mode. The controller also provides the appropriate address for the ATR constants stored in the ROM. The Block RAM resources available on the Virtex-II were used in order to implement the ROM. Note also that the values of the x and y output registers are shifted one bit position to the right in order to correct for the scaling factor. When all iterations have been completed, the x, y and z output registers contain the unscaled values for the rotated vector.

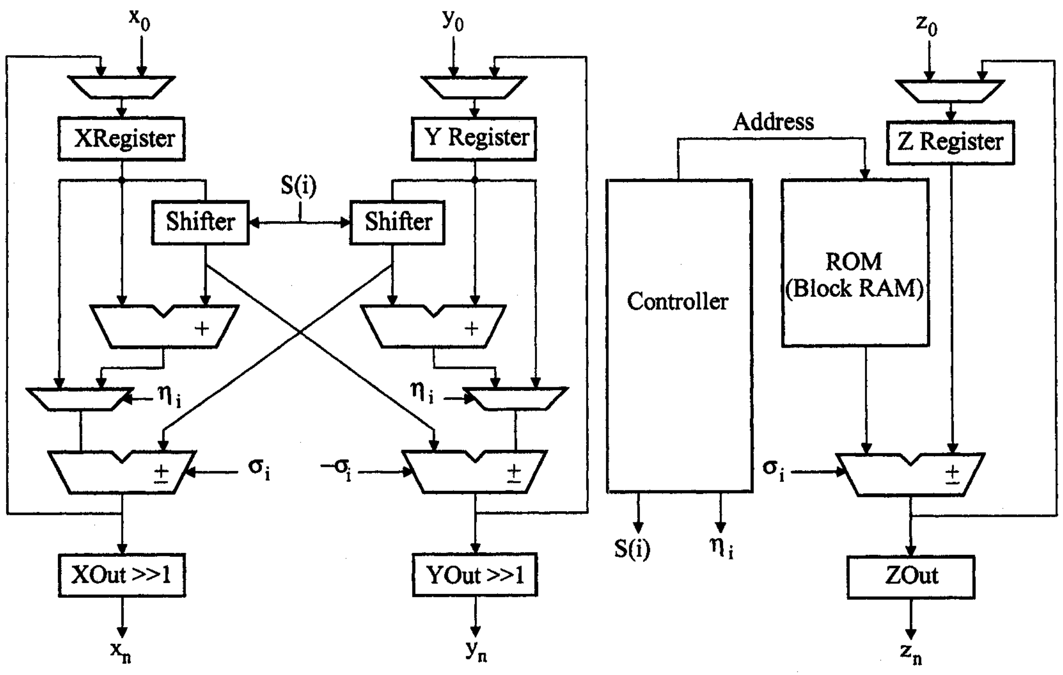


Figure 5.5: Bit-Parallel Iterative Implementation

5.3 Bit-Serial Pipelined Implementation

In a bit-parallel architecture, the adder/subtracter delay is equal to the the carry propagation time. The Virtex-II provides fast carry logic within each slice in order to speed up these operations. Parallel arithmetic requires the same amount of logic to perform operations as there are bits in the operands. This means that for each bit of a register, one logic cell is required (recall that there is 1 memory element per cell). One way to significantly reduce the amount of hardware required to perform arithmetic operations is to perform bit-serial arithmetic. Since each LUT in the Virtex-II can also be configured as a 16-bit shift register, registers will take up less logic and wiring resources on the device. This does not amount to a reduction in area of 16:1 however, since arithmetic operations have to be controlled by a sequencer. Bit-serial arithmetic can also prove very useful when no fast carry logic is available on a device. Bit-serial implementations will be much slower than bit-parallel implementations, but the reduction in area and routing makes this solution particularly attractive for FPGAs, especially for System On Chip applications.

The first consideration when attempting to map the compensated CORDIC algorithm in a bit-serial architecture is the implementation of a tapped shift register. An implementation example of a 34-bit tapped register with LSB output is shown in Figure 5.6. Each LUT in the Virtex-II can be used as a 16-bit variable tap shift register, denoted as *SRL16* in Figure 5.6. The values on the address lines (*A0* through *A3*) indicate which bit of the register will appear on the Q output. In order to implement a shift register with two outputs (one for the tap and one for the LSB), at least two shift registers are needed with the appropriate addresses for the Q outputs. The synthesizer also uses the available flip-flop element since it will be lost if it is not used.

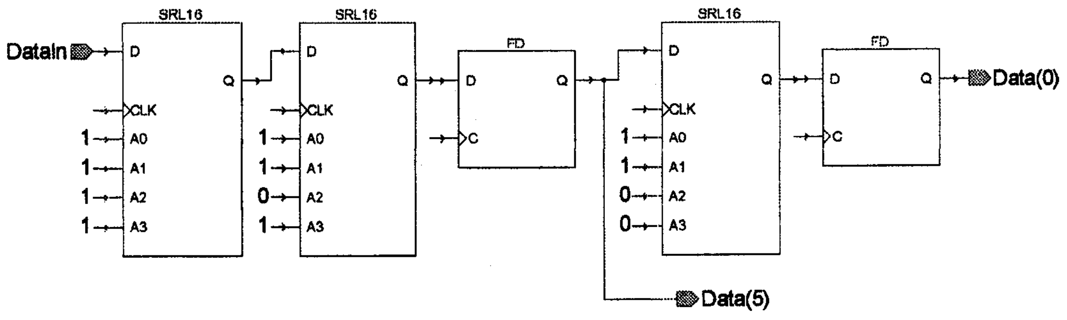


Figure 5.6: 34-bit Tapped Shift Register with LSB Output

Next, a serial adder/subtracter element is required for the iteration equations. An effort was made in order to maximize the use of available logic resources. The resulting element is shown in Figure 5.7 and requires one LUT for the sum (*LUT1*), one LUT for the carry or borrow (*LUT2*) and one D Flip-Flop for storage of the carry or borrow (*DFF1*). The *Init* signal resets the carry/borrow storage element when starting a new operation. For a compensated CORDIC stage, an extra addition is required, and the resulting compensated adder is shown in Figure 5.8. The structure of this element is similar to the serial adder/subtracter element, except that the second operation is always an addition. Also, the compensated element will have a delay equivalent to that of two LUTs.

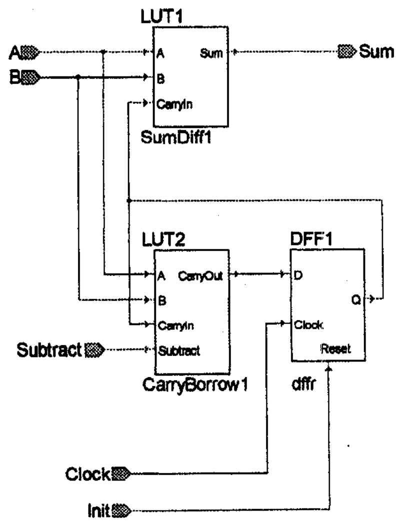


Figure 5.7: Bit-Serial Add/Subtract Element

The tapped shift registers and serial adders are then used in the data path pipe sections. A pipe section for the x and y data path is depicted in Figure 5.9. A storage element is present in order to store the MSB of the x and y registers when the *RecordSign* signal is high. This sign is used for sign extension of the shifted operand, which is controlled by the *ExtendSign* signal. The LSB and tap outputs are fed to a serial adder/subtracter in the case of a regular stage or a compensated serial adder/subtracter in the case of a compensated stage. Once again, the σ_i signals are generated from the sign of either the y or z sign storage element depending on the mode of operation. The z data path is shown in Figure 5.10. The ATR constants are stored in one or more LUTs, and are addressed using a Linear Feedback Shift Register (LFSR). An LFSR is used since it has a simpler hardware implementation than other types of counters. Because of this, the ATR digits have to be recoded in order to correspond to the count sequence of the LFSR. Also note that all z pipe sections only require a simple serial adder/subtracter.

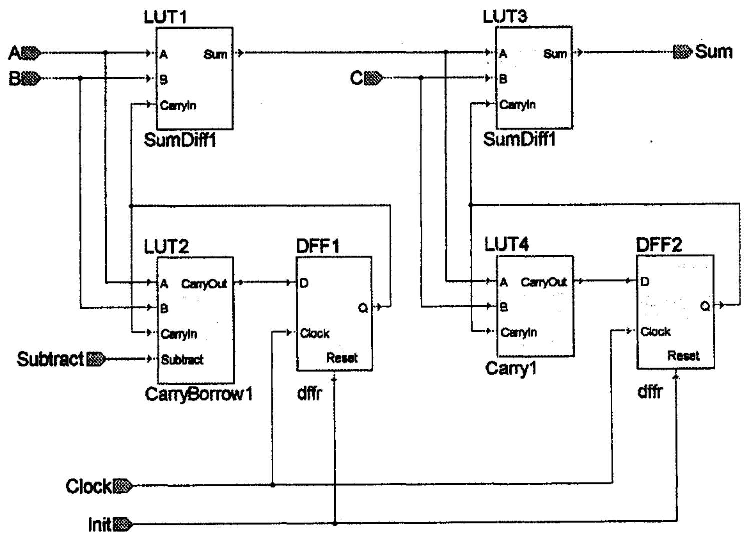


Figure 5.8: Bit-Serial Compensated Element

In order to minimize long wire delays and increase performance, control pipes as shown in Figure 5.11 are used in order to propagate control signals injected by a sequencer at the front of the pipeline. Figure 5.12 shows an example of a timeline that was used in order to determine the control signal dependencies for each stage. Control signals are divided into internal and external control signals. The *Subtract* signal is the only internal control signal, and is used as a control signal for the adder/subtracters of the x, y and z datapaths, where a value of 1 indicates a subtract operation, and

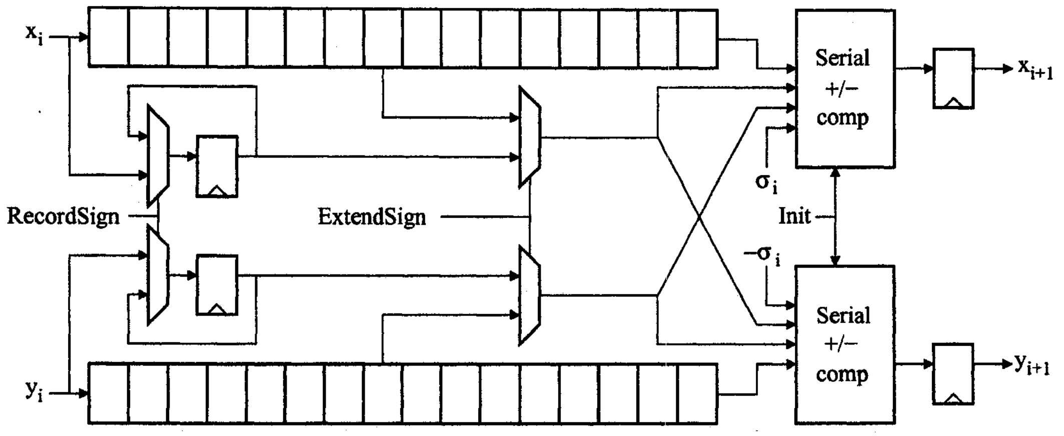


Figure 5.9: Pipe Section - x and y Data path

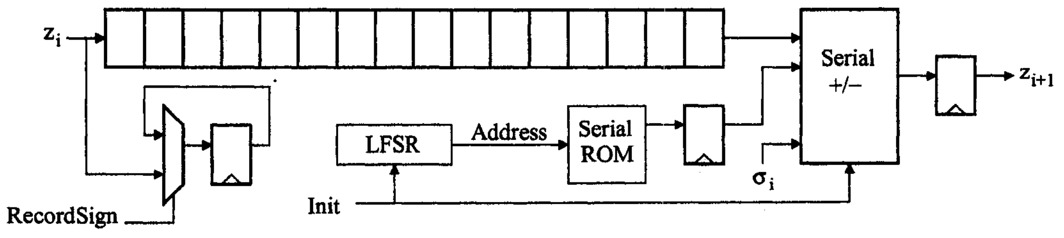


Figure 5.10: Pipe Section - z Data path

a value of 0 indicates an add operation. In the case of the x and z datapaths, the *Subtract* signal is equal to the σ_i signal. In the case of the y datapath, the *Subtract* signal is equal to $-\sigma_i$. External control signals are those that have to be generated by the control pipeline, and include the *Init*, *RecordSign* and *ExtendSign* signals. As can be seen in Figure 5.12, there is a delay of two clock cycles between successive pipeline stages. This delay is required in order to properly align the external control signals generated by the control pipeline. One of these delays is a flip-flop inserted after the serial adder/subtracter of each stage of Figure 5.10 and a second delay is included in the shift registers by increasing their size by one extra bit, for a total of two clock cycles. The first clock cycle delay is required for the *Init* signal, which initializes the carry/borrow storage element of the serial adders. This signal must be active at the

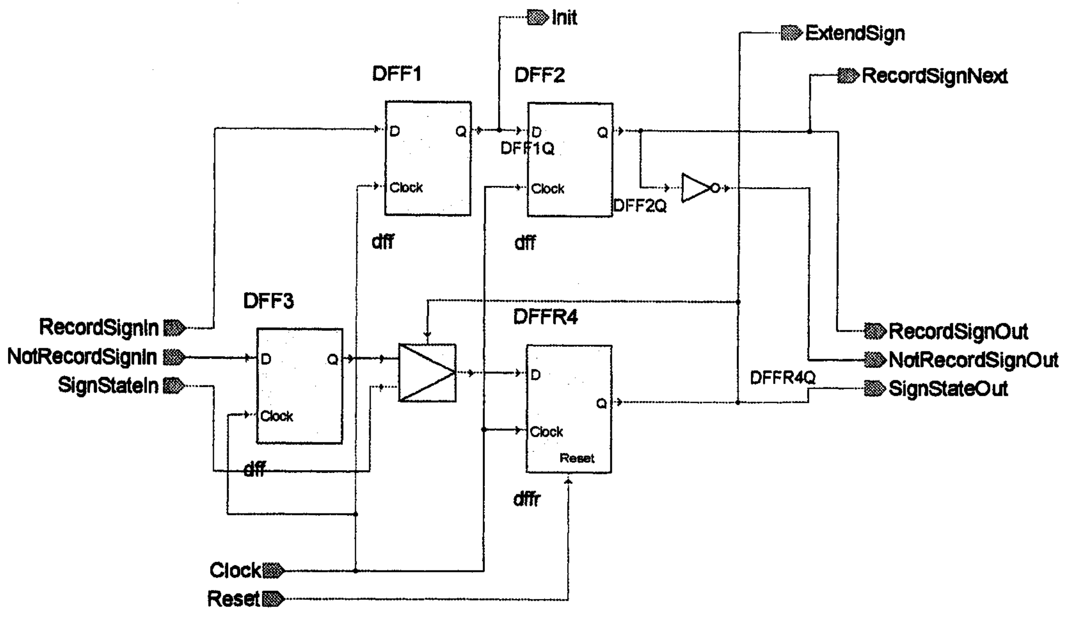
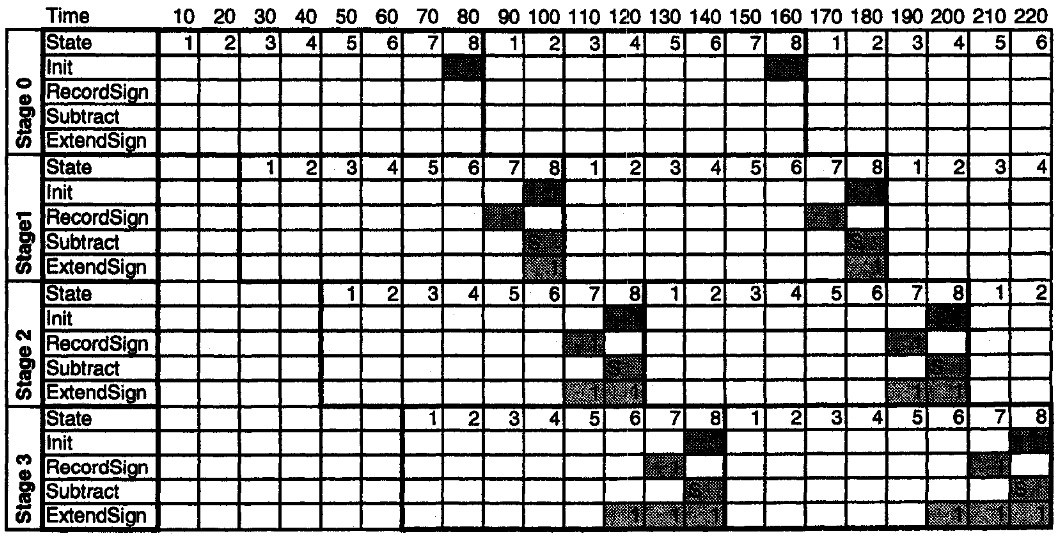


Figure 5.11: External Control Pipe Section



	Time	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	160	170	180	190	200	210	220
Stage 0	State	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4	5	6
	Init								▨								▨						
	RecordSign																						
	Subtract																						
	ExtendSign																						
Stage1	State			1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4
	Init										▨								▨				
	RecordSign									▨ 1								▨ 1					
	Subtract										▨ S								▨ S				
	ExtendSign										▨ 1								▨ 1				
Stage 2	State					1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2
	Init												▨								▨		
	RecordSign											▨ 1								▨ 1			
	Subtract												▨ S								▨ S		
	ExtendSign											▨ 1	▨ 1							▨ 1	▨ 1		
Stage 3	State							1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
	Init														▨								▨
	RecordSign													▨ 1								▨ 1	
	Subtract														▨ S								▨ S
	ExtendSign												▨ 1	▨ 1	▨ 1						▨ 1	▨ 1	▨ 1

Figure 5.12: External Control Timing (8 Bits, 4 Iterations)

same time as the last bit enters the LSB of the shift register. The other clock cycle delay is required in order to properly align the signals in the control pipeline. The

RecordSign signal needs to be generated one clock cycle after the *Init* signal of the previous stage, and is generated one clock cycle before the *Init* signal of the next stage. In order to clarify this, the signal on the control pipe of Figure 5.11 is labeled *RecordSignNext* instead of *RecordSign*. Finally, the *ExtendSign* signal of stage $i+1$ is initiated immediately after the initiation of the *ExtendSign* signal of the previous stage i, and is terminated one clock cycle after the *RecordSign* signal of the present stage $i+1$. Note that the *Subtract* signal on Figure 5.12 indicates the moment at which this internal control signal becomes valid.

5.4 Bit Serial Iterative Implementation

One of the challenges in implementing the bit-serial iterative architecture is to efficiently implement the very wide multiplexers for the shifted values of the x and y data path. As suggested in [1], the tapped shift registers can be implemented by configuring the Virtex-II slices as dual port RAM. As stated in Chapter 3, the Virtex-II function generators may be configured as 16x1-bit RAM, as shown in Figure 5.13. Any type of RAM which is implemented within the Control Logic Blocks is refered to as CLB RAM, as opposed to the dedicated dual port block select RAM (18Kbit) resources.

In the case of the Virtex-II, CLB RAM can also be easily configured as dual-port RAM shown in Figure 5.14 with one synchronous write port *D* and two asynchronous read ports *DPO* and *SPO*. The *A[3:0]* address is used for both the write port and the *SPO* read port, and the *DPRA[3:0]* address is used for the *DPO* output. In this configuration, a shift register can be implemented by using the CLB RAM and incrementing the *A[3:0]* address after each access. The second read port *DPO* of the dual port RAM can be used for the register tap, and by properly sequencing the *DPRA[3:0]* address on this second port, sign extension can be accomplished at no extra hardware cost.

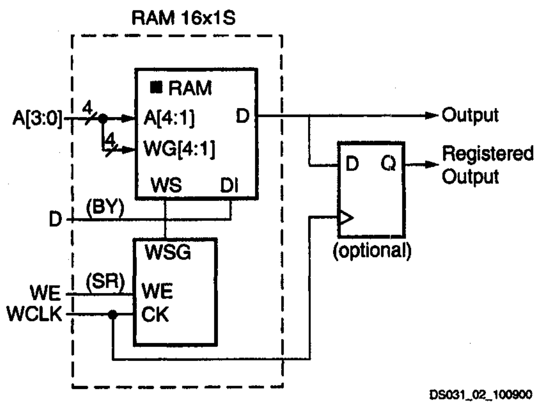


Figure 5.13: Virtex-II CLB RAM

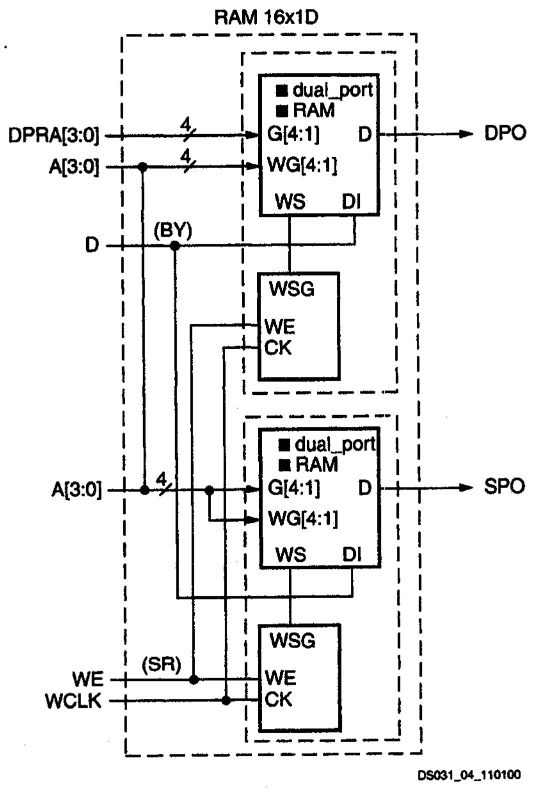


Figure 5.14: Virtex-II Dual Port RAM

The basic CLB RAM unit can implement 16x1-bit Dual Port RAM by using a single slice (two logic cells). By cascading 16x1-bit CLB RAM with multiplexers, it is possible to implement the 32x1-bit and 64x1-bit RAM required for the wider data path widths. The addresses for the CLB RAM are generated by the controller, as shown in figure 5.15. One set of lines is for the bit count (*BitCount*), and seperate address lines are for the tap count (*TapCount*).

Input multiplexers (*MX*, *MY* and *MZ*) are required for choosing whether to load input data or select data from the internal data paths according to the *Load* signal. The x and y data path are comprised of a dual port sync RAM and a compensated adder/subtracter element. This element takes the unshifted data from the corresponding read/write port (x_i and y_i for the x and y datapaths respectively) and the shifted outputs from both read only ports ($x_i 2^{-S(i)}$ and $y_i 2^{-S(i)}$). The subtract signal S is stored in a flip flop, and is fed to all add/sub elements. The compensated add/sub elements also have an extra input η, which indicates if compensation is required for the current iteration. This η signal and the initialize signals (labeled I) are generated by the controller. Note that the add/sub element of the z data path is a simple adder subtracter where one of the inputs originates from the z Single Port RAM and the second input originates from the ROM that stores the ATR constants.

In order to keep the controller as simple as possible, the serial add/sub elements of Figure 5.15 are asynchronous so they don't introduce any clock cycle delays; the result of the operation is directly fed into the input multiplexers. This means that there will be a delay of two LUTs (one for the first add/sub operation and one for the compensated addition) plus one multiplexer delay between the output of the dual port CLB RAM to its input. This allows for the simple controller design shown in Figure 5.16. It is essentially made up of three counters: the bit counter, the tap counter and the iteration counter. The first two counters work together in order to provide the proper addresses for the bit count and the tap count. For a b-bit width

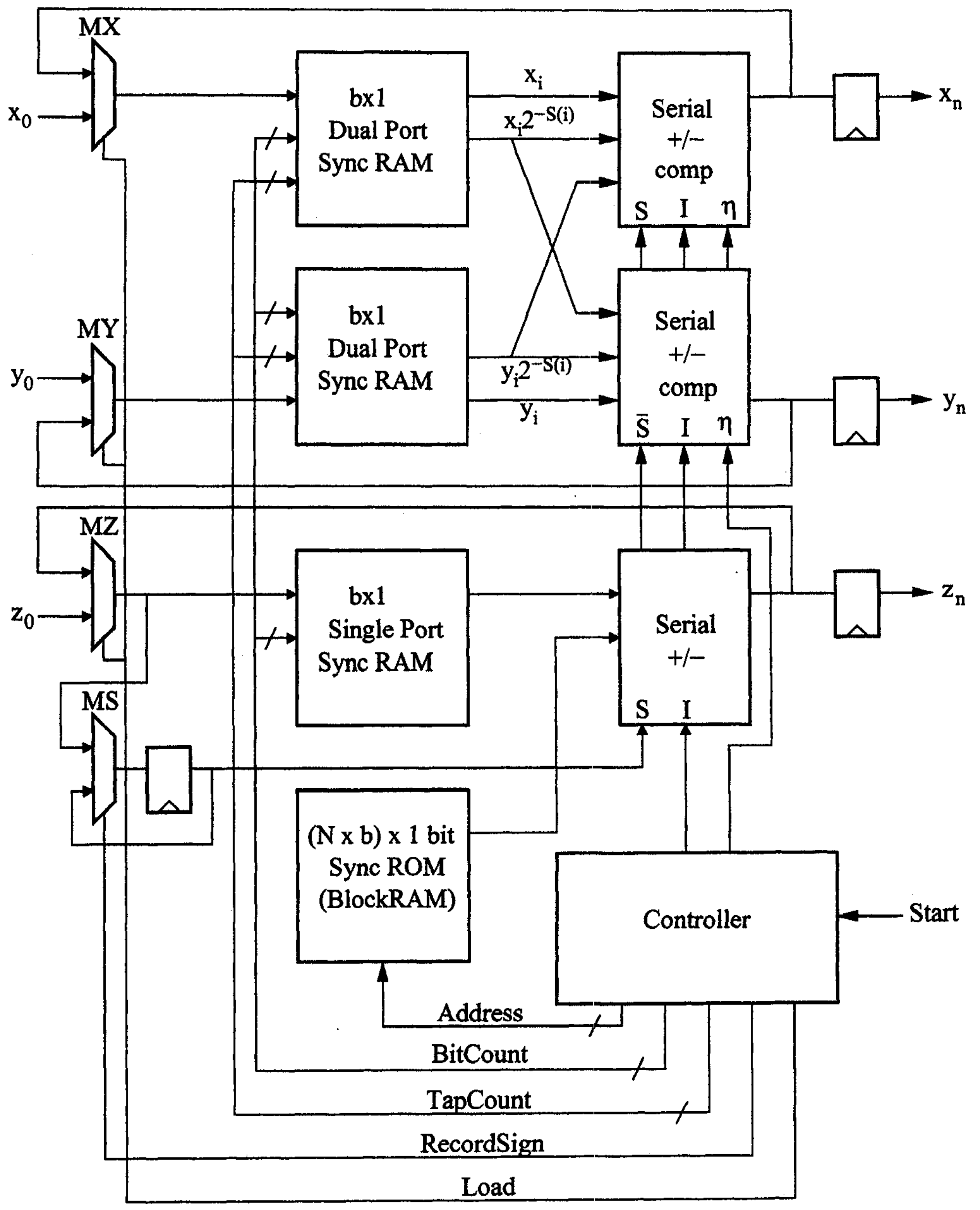


Figure 5.15: Bit-Serial Iterative Architecture

processor, the bit counter will always count from 0 to $b - 1$. In contrast, the tap counter must start with the current shift value index, and count up to $b - 1$. The $b - 1$ count must be kept until the bit counter has finished counting; this has the effect of performing sign extension on the shifted output.

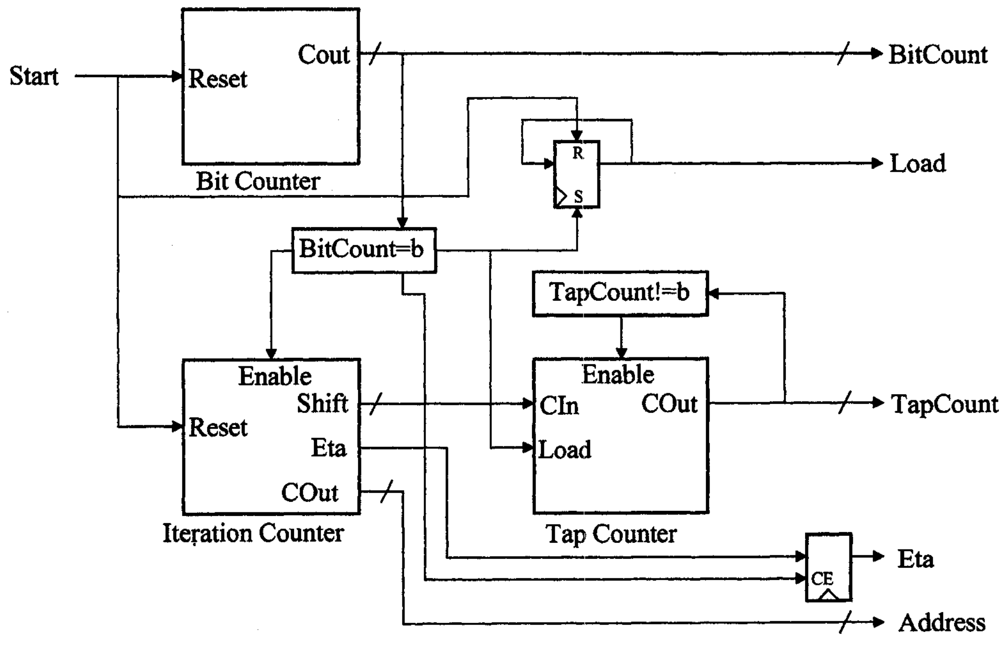


Figure 5.16: Bit-Serial Iterative Controller

Another key element in the synchronization of the entire process is the iteration counter. Recall that the compensated CORDIC shift sequence contains two distinct variables, $S(i)$ and η_i, as listed in table 4.1 of Chapter 4. However, since there are repetitions in the $S(i)$ sequence, this variable cannot be used as the count sequence. In order to keep the hardware as simple as possible, it was decided to use the combination of $S(i)$ and η_i as the count sequence. In this manner, the $S(i)$ shift factors and η_i factors are immediately available from the controller, and the count sequence is used as an address for the ROM. Note that using the concatenation of $S(i)$ and η_i leads to a non-linear count sequence, and there will be gaps in the ROM addresses. Also, the fact that the tap counter needs to be loaded with the value of the $S(i)$ parameter presents a synchronization problem. First, the iteration and bit counters are reset to 0 when the Start signal is activated. Second, in order for the tap counter to be loaded with the proper starting value, the iteration value can only be incremented *after* the tap counter has latched in the proper value. The easiest solution to this problem was to ensure that the iteration counter has a shift value of $S(i+1)$. A similar approach is also required in order to properly latch the value of the *Eta* signal, which is accomplished using a Flip Flop and Clock Enable signal which acts as the "Load" signal.

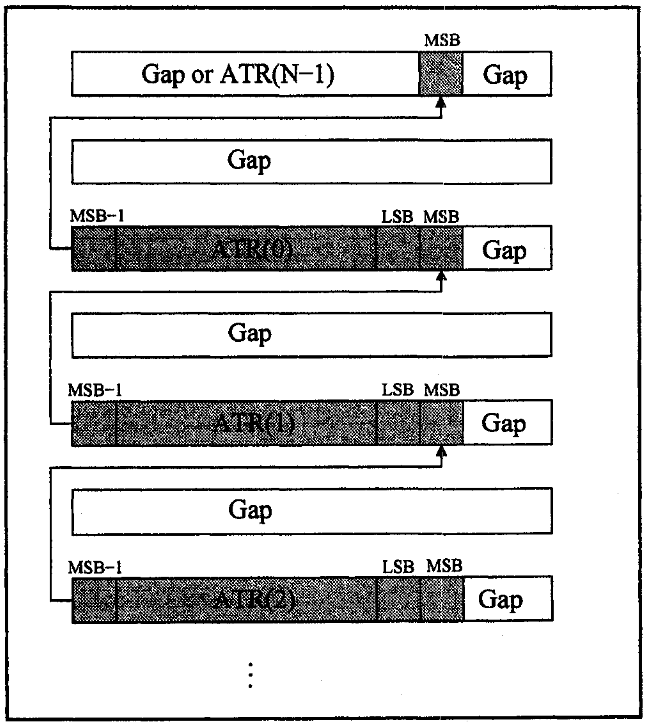


Figure 5.17: ATR ROM Memory Layout

Synchronization issues and a non-linear counting sequence will also affect the ATR ROM memory layout. The amount of data required to be stored in the ROM is $(b \times N) \times 1$ bits. The first consideration is in the generation of the addresses for the ROM. The ROM addresses is split into two parts: the Address High (AH) are formed from the concatenation of $S(i)$ and η_i, and the Address Low (AL) bits are formed from the *BitCount* address bus. Since the AH address count is non-linear (it skips over certain values), it will introduce gaps in between the ATR constants

(Figure 5.17). Since the AL address bits are taken from the bit count, there is a possibility that the AL address will not cover all the possible values for the AL bus. This will introduce small trailing gaps right next to the ATR constants. As already discussed, the shift values of the iteration counter (and hence, the AH address bits), are always +1 with respect to the *current* iteration value. In order to compensate for this, there will be an offset in the memory layout. This means that ATR(0) will have to be stored at AH = x0002 instead of AH = x0000. All ATR constants will thus have to be moved accordingly in memory. Since we are using sync ROM, there is also a clock cycle delay between the availability of the address, and the resulting output. In order to account for this, the MSB of each ATR has to be shifted up in memory as depicted in Figure 5.17.

5.5 Summary

In this chapter, the proposed FPGA implementations of the compensated CORDIC algorithm for the four different architectural styles were presented. From this discussion, it can be seen that the four implementations take advantage of different FPGA resources, and they will require quite different descriptions when using a hardware description language. Furthermore, these hardware descriptions have to be parameterized in order to be able to generate a CORDIC PE of a given data width and precision. In the next chapter, a new approach to VHDL code generation will be examined, along with the proposed design flow for the compensated CORDIC processing elements.

CHAPTER 6: VHDL CODE GENERATION AND SYNTHESIS

The compensated CORDIC PEs need to be scalable in terms of data width and precision. Some of the implementations of the CORDIC PE contain elements that are not easily parameterizable using standard VHDL constructs, such as LFSRs and specialized ROMs that need a very specific memory layout. In this chapter, a new approach to the generation of synthesizeable VHDL code will be examined, along with the proposed design flow and automatic code generation tool built as part of this project.

6.1 Code Generation With PHP Hypertext Preprocessor

Code generation is the process of producing program source files in some automatic manner. In a more general sense, it essentially amounts to "writing code that writes code". In this particular case, the target language is VHDL, or to be more precise, *synthesizeable* VHDL. Synthesizeable VHDL code will allow for efficient and correct translation of a high-level hardware description into a netlist that can be implemented on an FPGA. VHDL contains a host of features that make it a powerful language for the description and simulation of hardware systems. However, the nature of synthesis precludes the use of a wide range of VHDL constructs: some abstractions simply do not have hardware equivalents. The subset of VHDL constructs appropriate for synthesis is often referred to as *Register Transfer Logic* (RTL). It is a coding style that essentially describes hardware systems containing memory elements (such as registers) and the logic present between these elements. It is thus vital for the designer to properly partition the design in order to obtain predictable results. The best way to achieve this is to subdivide the design into smaller components and to synthesize them individually. In this manner, the VHDL description can be adjusted in order to obtain the desired result. Sometimes, the best way to describe a particular component will be in a purely "structural" manner.

Parameterization is also an important consideration when describing hardware. Parameterization will allow for the reuse of existing VHDL code in order to instantiate components of different word sizes and internal architecture. VHDL does provide some basic constructs in order to allow for parameterized designs, such as generics and the `generate` statement. However, these statements are rather inflexible, and only allow for limited parameterization. Features such as including or removing input/output ports and renaming entities according to user supplied parameters cannot be easily done. Furthermore, arithmetic functions such as `ceil` and `log2` used to calculate generics need to be carefully coded in order to make them understandable for the synthesis tool. Another area of concern is in automatically generating the values for a ROM of given depth and width. Clearly, VHDL lacks flexibility in its ability to completely parameterize a design. Also, the restricted VHDL subset allowed by the synthesis tool further restricts this flexibility by providing inference templates. In contrast, code generation provides unlimited flexibility in parameterizing a synthesizeable design and is a much more efficient way of achieving code reuse.

The language used for code generation in this case is PHP, which is a recursive acronym for *PHP: Hypertext Preprocessor.* It is a widely popular open-source programming language used primarily for server-side applications and developing dynamic web content. PHP has also benefited from many enhancements over the past few years, most notably CLI (Command Line Interface) and PHP-GTK. The CLI allows PHP to be used as a general purpose scripting language from the command line. PHP-GTK is an object-oriented interface to the Gimp Toolkit (GTK+) classes and functions for writing client-side GUI applications. PHP's syntax is borrowed from Perl, C, C++ and Java. The most peculiar characteristic of this language is that it can be literally embedded inside another source file. As its name implies, it is primarily used in conjunction with HTML in order to dynamically generate a Web page on the server before transmitting it to the client.

It is much more than a simple preprocessor or macro language, and is an extremely powerful tool for code generation (i.e. writing code to write code). The PHP code is embedded within starting <? and ending ?> tags. Furthermore, any number of code fragments can be inserted within a PHP file, and all text outside these fragments is considered as literal `print` statements, conserving the exact format including all white spaces and carriage returns.

In order to illustrate VHDL code generation with PHP, consider the code fragment of Listing 6.1 describing the entity for a maximum length LFSR.

Listing 6.1: Maximum Length LFSR in PHP (Entity)

```
<? php
/* Maximum length LFSR for different bit widths */
$xnor_taps = array(
 3 => array (3,2),  4 => array (4,3),   5 => array (5,3),
 6 => array (6,5),  7 => array (7,6),   8 => array (8,6,5,4),
 9 => array (9,5), 10 => array (10,7), 11 => array (11,9),
12 => array (12,6,4,1));
$lfsr_width = $argv[1];
php ?>
library ieee;
use ieee.STD_LOGIC_1164.all;
entity Lfsr<?print $lfsr_width?> is
    port(
        Clock : in STD_LOGIC;
        Reset : in STD_LOGIC;
        CountOut : out STD_LOGIC_VECTOR(<?php print $lfsr_width-1 ?> downto 0)
    );
end Lfsr<?print $lfsr_width?>;
...
```

By typing php `lfsr.php 4 > Lfsr4.vhd` on the command line, the embedded code will be processed and the output file `Lfsr4.vhd` will contain the result of the output. The example in Listing 6.1 demonstrates the basic idea of code generation using the PHP language. Lines 2 to 7 describe the `$xnor_taps` array structure which contains the taps required for maximum length LFSRs of width 3 to width 12. Line 8 parses the first command line argument, in this case, `$argv[1] = 4`. The value of `$lfsr_width` is then used in lines 12 and 18 in order to give the entity a descriptive name. In this particular case, the entity will be named `Lfsr4` in the resulting output file. Finally, `$lfsr_width` is used in line 16 in order to provide the upper bound of the `CountOut` vector. The resulting output after preprocessing is shown in Listing 6.2.

Listing 6.2: Maximum Length LFSR in VHDL (Entity)

```
library ieee;
use ieee.STD_LOGIC_1164.all;

entity Lfsr4 is
    port(
        Clock : in STD_LOGIC;
        Reset : in STD_LOGIC;
        CountOut : out STD_LOGIC_VECTOR(3 downto 0)
    );
end Lfsr4;
...
```

The corresponding architecture portion of the PHP file is shown in Listing 6.3. Once again, the `$lfsr_width` variable is used in order to construct the proper entity name (`Lfsr4`) on line 21 and as an upper bound for the register `Q` on line 23. Lines 25 through 34 contain the PHP code that prints out the feedback equation in VHDL syntax for the LFSR, and uses the `$xnor_taps` array structure defined in Listing 6.1. The resulting output is shown in Listing 6.4.

Since any text that is not within the PHP begin <? and end ?> tags is interpreted as one big print statement, conditional output of entire blocks of text is greatly simplified. An example of conditional output is shown in Listing 6.5. In this

Listing 6.3: Maximum Length LFSR in PHP (Architecture)

```
...
architecture Behavioral of Lfsr<?print $lfsr_width?> is
signal FeedBack : STD_LOGIC;
signal Q: STD_LOGIC_VECTOR(1 to <?php print $lfsr_width?>);
begin
        FeedBack <= <?php
                $taps = $xnor_taps[$lfsr_width];
                $count = count($taps);
                for ($i = 0; $i < $count; $i++) {
                        print "Q($taps[$i])";
                        if ($i < $count-1)
                                print " xnor ";
                }
                print ";\n";
    ?>
...
```

Listing 6.4: Maximum Length LFSR in VHDL (Architecture)

```
...
architecture Behavioral of Lfsr4 is
signal FeedBack : STD_LOGIC;
signal Q: STD_LOGIC_VECTOR(1 to 4);
begin
        FeedBack <= Q(4) xnor Q(3);
...
```

listing, it can be seen that the `if-else` construct is formed by lines 2, 9 and 11, and the two blocks of text are on lines 3-9 and line 10.

Embedding PHP within the VHDL code does have the disadvantage of having to deal with two different languages. Surprisingly, VHDL with embedded PHP is not more complicated than the equivalent parameterized VHDL code. The reason for this is that the VHDL code present in the PHP file describes a pure hardware architecture, whereas the PHP provides all the support for parameterization. Finally, the VHDL code resulting from code generation can be made much more understandable since

Listing 6.5: Conditional Block in PHP

```
...
<? if ($is_buffered) { ?>
DFF1 : dff
    port map(
        Clock => Clock,
        D => NET001,
        Q => DataOut
    );
<? } else { ?>
DataOut <= NET001;
<? } ?>
...
```

the hardware description isn't clouded by extraneous parameterization constructs.

Traditionally, languages such as C, C++, Java and Matlab have been used in order to automatically generate VHDL code. Since these languages cannot be embedded in the source code, they yield heavier source files that are not as intuitive to work with. Every time a line has to be output to a file, one must use `printf` style statements. This leads to a lot of repetitive typing and does not provide a clear view of the structure of the output file. Another language that has some nice features for automatic code generation is PERL, which includes heredoc statements that allow to print entire formatted blocks of text. However, PERL cannot be embedded within the target source code.

6.2 Design Flow

The Design flow for the CORDIC Generation tool is depicted in Figure 6.1. A GUI front-end allows the user to input the parameters for the generation of the CORDIC processor. The tool then performs preprocessing of the `.php` files which are VHDL source files with embedded PHP code. After preprocessing, all the required VHDL files are now available for the CORDIC processor. These VHDL files can be used for functional simulation, synthesis using commercial design tools, or inclusion into a larger design. Implementation on the FPGA is the last step in the process.

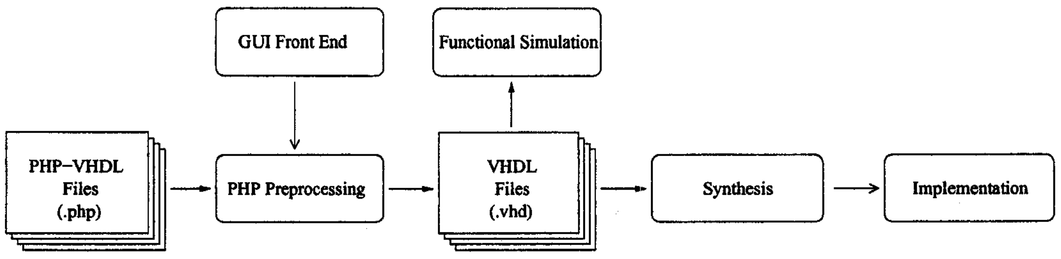
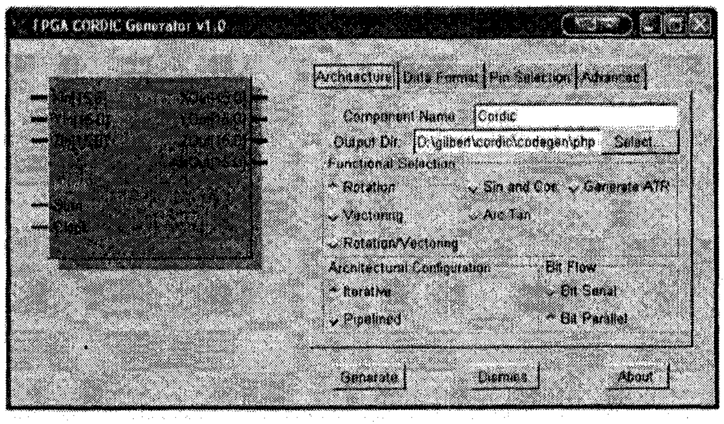


Figure 6.1: CORDIC Generation Tool Design Flow

6.3 CORDIC Generation Tool

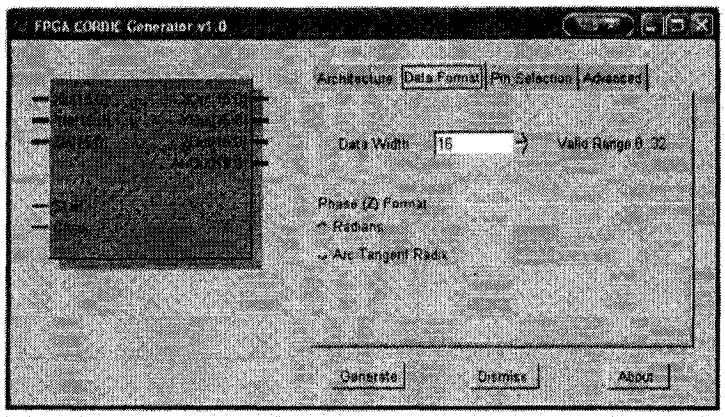
The CORDIC Generation Tool is a GUI front end for the automatic code generation of the various CORDIC architectures described in this thesis. It was programmed using the PHP-GTK framework and provides a user friendly interface that allows the designer to control the generation of the CORDIC implementations. It contains a block diagram of the generated architecture that contains all input and output ports along with their characteristics, and is updated in real-time. The interface is very similar to the one used for the Xilinx LogiCORE library. The Architecture page (Figure 6.2) allows the designer to assign the component name and the desired output directory for the generated VHDL files. It also has options for the functional selection, architectural configuration (iterative or pipelined) and what type of bit-flow to use (bit-serial or bit-parallel arithmetic). Note that currently, only the first three functional selection options (rotation, vectoring and rotation/vectoring) have been implemented. The second page is for the Data Format of the processing element (Figure 6.3). It allows the user to enter the width of the data path. In order to simplify the CORDIC PE designs, the x, y and z data paths all have the same width. Another option is the phase format for the z data path. Currently, only the



Component Name: Cordic

Output Dir: D:\gilbert\cordic\codegen\php

Figure 6.2: CORDIC Generation Tool - Architecture



Data Width: 16 Valid Range 8..32

Figure 6.3: CORDIC Generation Tool - Data Format

radians format is supported. The third page contains the Pin Selection sheet (Figure 6.4). It allows the user to select which input/output pins will appear in the final VHDL entity. Note that certain pins are optional, and others are not. This depends on the functional selection the user made on the architecture page. The last page (Figure 6.5) is for other options, such as the number of iterations, pre-rotation by $\pi/2$ and whether or not to perform scale factor compensation. Obviously, scaling factor compensation is accomplished using the compensated CORDIC algorithm. Note also that at this point, pre-rotation by $\pi/2$ has not been implemented.

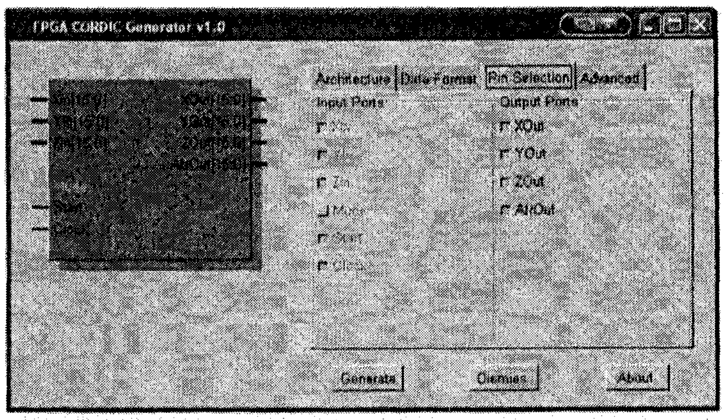
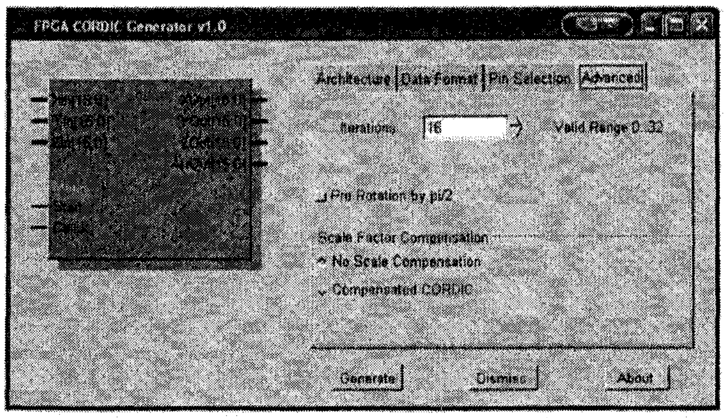


Figure 6.4: CORDIC Generation Tool - Pin Selection



Iterations 16 Valid Range 0..32

Pre Rotation by pi/2

Scale Factor Compensation
No Scale Compensation
Compensated CORDIC

Figure 6.5: CORDIC Generation Tool - Advanced

6.4 Summary

In this chapter, the automatic code generation method using the PHP language was described, along with the proposed design flow for the parameterized generation of the compensated CORDIC PEs. It was shown that PHP is superior to other languages such as C, C++, Java and Perl because of its ability to be embedded in VHDL code. The next chapter contains the synthesis results of the CORDIC PEs generated by the automatic code generated tool for the four basic architectural styles.

CHAPTER 7: RESULTS

The proposed CORDIC PE implementations provide the designer with a wide variety of options in order to provide a module capable of performing general vector rotation. Some or the architectures were designed for maximum speed and low latency, while others attempt to use the least amount of logic resources possible, leading to the tighter integration necessary for SoC applications. Providing characterisation data for the proposed architectures is crucial in order to allow designers to have a realistic view of what is achievable in terms of performance and area when implementing the CORDIC algorithm on an FPGA. This chapter outlines the performance characteristics of various synthesized versions of the proposed CORDIC PE implementations built by the automatic code generation tool.

7.1 Characterization Environment

The FPGA platform used in this work was the Xilinx Virtex-II, device number xc2v2000, package number ff896 and speed grade -6. The synthesis tool used is XST G. 31a, Release 6.2.03i. Default parameters for the synthesis tool were used, with the exception of optimization options. The optimization goal was set to "speed" and the optimization effort to a level of 2.

The CORDIC PEs were produced by the automatic code generation tool, and the resulting descriptions were output in VHDL. The compilation and simulation of the PEs were carried out using ActiveHDL 6.2. A Matlab script was used to provide the necessary data for functional simulation of the compensated CORDIC algorithm. To obtain a good number of samples, compensated CORDIC PEs for 12, 16, 24 and 32-bit precision for pipelined (bit-parallel and bit-serial) and for iterative (bit-parallel and bit-serial) were generated. Since vectoring and rotation have the same hardware complexity, only the rotation mode of operation was used in generating the CORDIC PEs. Pre-rotation was not used in order to generate the simplest possible processing elements for comparison purposes. MSB bits for vector magnitude growth were included in the data format, but extra LSB bits for error accumulation were excluded and truncation is used instead of rounding. A summary of the CORDIC PE data path characteristics is shown in table 7.1.

Precision	Datapath	Maximum	Number of
	Width	Shift	Iterations
12	14	12	14
16	18	16	18
24	26	24	27
32	34	32	35

Table 7.1: CORDIC PE Data Characteristics

The generated CORDIC processing elements were benchmarked against the Xilinx CORDIC LogiCORE v2.0. The Xilinx LogiCORE is able to generate only the bit-parallel iterative and pipelined architectures. Once again, the LogiCORE CORDIC implementations have no pre-rotation, are generated for rotation mode only, have the same data path characteristics as those outlined in Table 7.1 and perform truncation instead of rounding. Other settings specific to the LogiCORE CORDIC include maximum pipelining (only valid for the pipelined implementation), creation of Relationally Placed Macro set to "off", and scaling compensation is accomplished using a specialized LUT based Constant Coefficient Multiplier. Obviously, the latency of the Xilinx CORDIC cores will be different than the number of iterations outlined in Table 7.1 for the compensated CORDIC. According to the Xilinx CORDIC LogiCORE v2.0 data sheet, the LogiCORE is an implementation of the original CORDIC algorithm when using the circular coordinate system. Scaling factor compensation is thus accomplished by a post-scaling multiplication.

For the benchmarks, 8 different Xilinx CORDIC LogiCOREs were generated, including 4 different bit-parallel pipelined versions and 4 different bit-parallel iterative versions (12, 16, 24 and 32 bits). These benchmarks were compared to 16 different compensated CORDIC PEs produced by the automatic code generation tool, including 4 bit-parallel pipelined, 4 bit-serial pipelined, 4 bit-parallel iterative and 4 bit-serial iterative (12, 16, 24 and 32 bits). The results have been divided into two categories, which are pipelined architectures and iterative architectures. For the pipelined architectures, the main areas of comparison were throughput, latency and number of slices. The iterative architectures have a slightly different comparison basis. First of all, latency is only a measure of how many iterations must be performed, and is used in determining the throughput. Secondly, the compensated CORDIC architectures differ slightly from the Xilinx implementation in that they use Block RAM for the ROM implementation. It was still deemed useful to include the amount of reduction in the number of slices in order to see what effect using block RAM can have on slice count reduction.

7.2 Results for Pipelined Architectures

The results for the Xilinx, bit-parallel and bit-serial pipelined architectures can be found in Tables 7.2, 7.3, 7.4 and 7.5. The first characteristic to be examined will be the clock frequency of the different implementations. It is interesting to note that the clock rate of the bit-parallel compensated CORDIC is higher than the Xilinx implementation. At first glance, this should not be the case, since the original CORDIC has a maximum delay of one adder between pipelined stages, and the compensated cordic, which uses 3:2 compression, has roughly one extra LUT delay. However, the synthesis reports show that the critical path for the Xilinx implementation also has an extra LUT delay. Since no source code is available for the Xilinx implementation, it can only be concluded that some form of compression must

	12-bit		
	Xilinx	Bit-Parallel	Bit-Serial
Clock Rate (MHz)	214.247	228.050	336.757
Number of Slices	501	496	176
Latency	18	14	14
Throughput (MHz)	214.25	228.05	21.05
Throughput Increase %	N/A	6.44	-90.18
Area Reduction %	N/A	1.01	64.87
Latency Reduction %	N/A	22.22	22.22

Table 7.2: Results for 12-bit Pipelined Architectures

	16-bit		
	Xilinx	Bit-Parallel	Bit-Serial
Clock Rate (MHz)	204.290	215.808	312.061
Number of Slices	822	912	303
Latency	23	18	18
Throughput (MHz)	204.29	215.81	15.60
Throughput Increase %	N/A	5.64	-92.36
Area Reduction %	N/A	-9.87	63.14
Latency Reduction %	N/A	21.74	21.74

Table 7.3: Results for 16-bit Pipelined Architectures

occur in the post-scaling multiplication. However, this fact alone does not explain why the compensated CORDIC is faster. There are actually three significant differences in the data paths of the Xilinx and the parallel compensated implementations. First, the compensated implementation uses the active low output of the second LUT, whereas the Xilinx implementation uses the active high output of the second LUT.

	24-bit		
	Xilinx	Bit-Parallel	Bit-Serial
Clock Rate (MHz)	189.215	203.479	312.061
Number of Slices	1624	1973	454
Latency	31	27	27
Throughput (MHz)	189.22	203.48	12.00
Throughput Increase %	N/A	7.54	-93.66
Area Reduction %	N/A	-17.69	72.04
Latency Reduction %	N/A	12.90	12.90

Table 7.4: Results for 24-bit Pipelined Architectures

	32-bit		
	Xilinx	Bit-Parallel	Bit-Serial
Clock Rate (MHz)	176.940	187.723	312.061
Number of Slices	2710	3413	737
Latency	40	35	35
Throughput (MHz)	176.94	187.72	8.67
Throughput Increase %	N/A	6.09	-95.10
Area Reduction %	N/A	-20.60	72.80
Latency Reduction %	N/A	12.50	12.50

Table 7.5: Results for 32-bit Pipelined Architectures

According to the synthesis report, this amounts to a difference of 0.347 ns. Secondly, the critical path of the compensated implementation has one less fast carry logic multiplexer delay from carry in to output, labelled as MUXCY:CI->O, but has one extra MUXCY multiplexer delay from S input to output, labelled MUXCY:S->O of 0.235 ns. The MUXCY:CI->O path used in the fast carry logic has an associated gate delay of 0.042 ns. Finally, there are slightly higher fanouts for certain gates in the Xilinx design, adding to the Net Delay for those gates. The net effect is that the critical path of the compensated implementation is lower than the one for the Xilinx implementation, which is a reduction of 6.06% for the 12 bit implementation and 5.75% for the 32 bit implementation.

The compensated serial implementation's critical path is mostly determined by the clock to output delay of the 16-bit shift register. Recall that a LUT in the Virtex-II can be configured as a 16 bit shift register, labeled as SRL16E. The clock to output delay of an SRL16E is 2.294 ns, which is 6.6 times higher than the delay of a LUT (0.347 ns). Since the Virtex-II has MUXCY elements for fast-carry logic (delay of 0.042 ns), the benefits of using bit-serial arithmetic is greatly diminished because of the overwhelming speed of the carry logic. In a bit-parallel CORDIC implementation, the factors that will affect the efficiency of the adder will be the clock to output delay of the memory elements, the fan-out of these memory elements, the delay of the LUT used for the sum and the entire fast-carry logic chain delay. The fast-carry logic chain delay is a function of how many bits make up a word. The maximum fan-out will occur in the last CORDIC stage, where the sign must be extended to $b-1$ bits for a b bit data path. These two delays are eliminated when bit-serial arithmetic is used, and is the reason why a faster clock rate can be achieved. If fast carry logic is not available in a programmable device, then the difference in clock speed between bit-parallel and bit-serial arithmetic would be much higher.

It is interesting to note that the clock rate for the 12-bit serial implementation is lower than the clock rate for the other 3 bit widths. The reason for this is that in the 12-bit case, the critical path has the output of the SRL16E chained to its corresponding flip-flop, which is in the same logic cell. For the other three cases, the registers are much wider, and the synthesizer produced a path form the output from an SRL16E to the input of the next SRL16E which resides in another logic element. This results in a difference of 0.235 ns, and is the reason why the 12-bit implementation has a higher clock rate than the 16, 24 and 32 bit implementations. Also, the throughput in a serial design is equal to the clock rate divided by the number of bits to be processed in a word. Two bits also have to be added in the pipelined designs in order to account for synchronization bit delays as explained in Chapter 5.

From this it can be seen that the compensated bit-parallel implementation had an average Throughput Increase of 6.43% over the Xilinx design, with a minimum of a 5.64% increase and a maximum of 7.54%. On the other hand, the average decrease in terms of throughput for the compensated bit-serial designs was 92.82%. It is important to mention however that even for a 32 bit serial implementation, a throughput of 8.62 MHz was achieved which is still quite impressive considering how little logic resources and more importantly, how few wiring resources were required.

As a rule of thumb in digital design, faster designs will usually have a larger area than their slower counterparts. As it can be seen from Tables 7.2, 7.3, 7.4 and 7.5, this is certainly the case here. In the case of the bit-parallel implementation, there is a steady increase in area compared to the Xilinx implementation. The main reason for this is the amount of compensated stages that contain 3:2 compressors, which is roughly 1/3 of all iterations for compensated CORDIC. As expected the bit-serial implementation has significant area reductions, with an average of 68.21% (maximum of 72.80% and minimum of 64.87%).

In a pipelined implementation, latency is another important aspect, since a system is "on-line" when the pipeline has filled and valid results are starting to be produced. By using the 3:2 compression technique for the compensated stages, it was possible to significantly decrease the latency, with a minimum gain of 12.5% and a maximum gain of 22.22%.

	12-bit		
	Xilinx	Bit-Parallel	Bit-Serial
Clock Rate (MHz)	141.29	117.77	253.97
Number of Slices	326	155	33
Number of BRAMs	0	1	1
Number of Iterations (per word)	20	14	14
Throughput (per word) MHz	7.06	8.41	1.51
Throughput Increase %	N/A	19.07	-78.60
Slice Reduction %	N/A	52.45	89.88

Table 7.6: Results for 12-bit Iterative Architectures

	16-bit		
	Xilinx	Bit-Parallel	Bit-Serial
Clock Rate (MHz)	140.83	108.50	226.68
Number of Slices	482	203	43
Number of BRAMs	0	1	1
Number of Iterations (per word)	25	18	18
Throughput (per word) MHz	5.63	6.03	0.79
Throughput Increase %	N/A	7.00	-86.03
Slice Reduction %	N/A	57.88	91.08

Table 7.7: Results for 16-bit Iterative Architectures

7.3 Results for Iterative Architectures

The results for the Xilinx, bit-parallel and bit-serial iterative architectures can be found in Tables 7.6, 7.7, 7.8 and 7.9. As can be seen in these tables, the clock rate of the Xilinx implementation will always be higher than the bit-parallel implementation, since the bit-parallel implementation requires two cascaded add/subtract operations

	24-bit		
	Xilinx	Bit-Parallel	Bit-Serial
Clock Rate (MHz)	107.98	104.51	226.682
Number of Slices	873	304	49
Number of BRAMs	0	1	1
Number of Iterations (per word)	33	27	27
Throughput (per word) MHz	3.27	3.87	0.35
Throughput Increase %	N/A	18.30	-89.31
Slice Reduction %	N/A	65.18	94.39

Table 7.8: Results for 24-bit Iterative Architectures

	32-bit		
	Xilinx	Bit-Parallel	Bit-Serial
Clock Rate (MHz)	103.39	91.63	203.48
Number of Slices	1352	426	68
Number of BRAMs	0	1	1
Number of Iterations (per word)	42	35	35
Throughput (per word) MHz	2.46	2.62	0.24
Throughput Increase %	N/A	6.34	-90.16
Slice Reduction %	N/A	68.49	94.97

Table 7.9: Results for 32-bit Iterative Architectures

and the Xilinx implementation presumably uses only one add/subtract operation for the x and y data path, as can be deduced from the synthesis log files. It is important to note that the clock rates for the bit serial design is the same in the case of the 16 and 24 bit implementations. This is to be expected, since the actual data width for both these implementations is 18 and 26 bit respectively, in order to account for growth in the rotated vectors. Thus, the 12-bit serial implementation requires 16-bit sync RAM, the 16 and 24 bit serial implementations require 32-bit sync RAM, and the 32 bit implementation (with an actual width of 34 bits) will require 64-bit RAM. The critical path in the case of the 12, 16 and 24 bit serial implementations is mainly due to the Sync RAM and compensated adders. Note that the basic building block for Dual Port Sync RAM is two logic elements configured as 16-bit RAMs. In order to build bigger RAMs, these basic building blocks are cascaded using multiplexing logic. It is interesting to note that in the case of the 32-bit serial implementation, the critical path is actually not the 64-bit Dual Port Sync RAM, but the state machine associated with the Tap Counter.

The key factor in speeding up the CORDIC process for the iterative design is to have the lowest possible number of iterations in order to complete the process. Even though the clock rate of the bit-parallel implementations was lower than the Xilinx implementations, the fact that fewer iterations were required resulted in a higher throughput for the compensated implementations. An interesting side effect of this reduction in clock frequency is that there will be a reduction in switching power dissipation. The average switching power dissipation is defined as:

$$P_{avg} = \alpha C_L V_{DD}^2 f_{clk} \tag{7.1}$$

where α is the node transition factor, C_L is the capacitive load being driven, V_{DD} is the power supply voltage and f_{clk} is the clock frequency. From this equation, it is obvious that reducing the clock frequency will directly reduce power consumption. The maximum increase in speed for the compensated CORDIC PE as opposed to the Xilinx implementation was 19.07 %, and the minimum increase was 6.34%. As expected, the decrease in throughput for the bit-serial compensated design was much lower than the Xilinx design, ranging between 78.60 and 90.16 %. However, the throughput of the 32-bit serial design was still around 240 KHz, which is still an acceptable data rate for many types of applications in audio processing.

Early on in the design process, it was determined that the ROM elements required for the compensated CORDIC might pose a problem in terms of performance. In order to simplify the design process, it was decided to use Block RAM in order to implement the ROM. Because of this, there was a considerable reduction in the number of slices required to implement both the bit-serial and bit-parallel compensated CORDIC algorithms. To get a precise idea of the relative amount of logic required to implement the bit-parallel vs the Xilinx implementation, an attempt would have to made in order to implement the ROM into standard slices. As far as the bit-serial implementation is concerned, it is almost certain that this design will be much smaller than its Xilinx counterpart, considering the very simple data paths it contains. Nonetheless, comparison of the slice reduction is still very useful, since it gives us an idea of what types of reduction in slice count can be achieved by using the built in Block RAM. Maximum slice reduction were 94.97% and 68.49% for the bit-serial and bit-pipelined implementations respectively.

CHAPTER 8: SUMMARY AND CONCLUSIONS

8.1 Conclusions

This thesis presented the development of four different types of CORDIC processing elements suitable for a wide range of applications. An in depth survey of a wide range of CORDIC variants was investigated and the most suitable algorithms for implementation on an FPGA platform were identified and evaluated. An evaluation of the suitability of using redundant number systems and high radix number systems for arithmetic operations on the FPGA was also conducted. It was found that the logic resources available on the FPGA are geared toward conventional two's complement arithmetic, and that the benefits that could be gained from alternate number representations on an FPGA would be lost due to the increase in logic resources required to implement them. The goal of this thesis was to develop different architectures that could be used for general vector rotation; i.e. the CORDIC processing elements must be able to perform in both rotation and vectoring modes of operation. Because of this, certain short-cuts specifically designed for use in rotation mode only were discarded, such as rotation prediction and the use of hybrid radix systems. However, for certain applications that require only the rotation mode (such as trigonometric function evaluation), use of these types of short-cuts is the most efficient way of implementing the CORDIC algorithm. For general vector rotation on FPGA architectures, it was found that the compensated CORDIC algorithm provides an efficient alternative to the conventional circular CORDIC algorithm first presented by Volder.

A novel technique for VHDL pre-processing using the PHP scripting language was also developed and incorporated into a GUI-based application for automatic code generation. This tool allows the user to quickly and efficiently produce synthesizeable VHDL code for the four different types of CORDIC architectures. One major feature of this tool is that it allows for the entire range of data width from 12 bits to 32 bits, and allows much freedom in tweaking the number of LSB guard bits and number of iterations in order for the user to control performance-area-precision trade offs.

The compensated CORDIC architectures were found to be extremely efficient and provide several implementation options that were unavailable in the LogiCORE tool provided by Xilinx. Both the bit-parallel pipelined and iterative implementations outperformed the Xilinx implementations at little or no extra cost in terms of logic resources. Additionally, the bit-parallel compensated architecture had a significant decrease in latency, ranging from 12.5% to 22.22% over the Xilinx implementation. The bit-serial architectures are extremely useful additions to the CORDIC implementations, since they provide acceptable performance for a fraction of the cost in terms of both logic and wiring resources.

8.2 Recommendations for Future Work

Although much work has gone into developing efficient architectures, there are still possible improvements that could be made to the architectures. In particular, the controllers of the iterative architectures could be implemented differently, using either Johnson or LFSR counters; this would require a different approach in order to provide the ATR constants, the shift coefficients and the compensated η_i coefficients. Also, since the iterative architectures were implemented using Block RAM, efficient LUT based memory structures could be developed in order for the compensated CORDIC implementations to be independent of these logic resources. Another enhancement for the bit-serial pipelined architecture would be to use delayed shift registers for the smaller ATR angles instead of using serial ROM structures.

This thesis examined general vector rotation for both the rotation mode of operation and vectoring. Short-cuts specific to rotation only, especially for trigonometric function evaluation, could be investigated and implemented for FPGAs. Also, extending the architectures in order to be able to use and generate the ATR representation for angles would be very useful, in particular for use in systolic array architectures were the output of one CORDIC processor is fed into the input of another. These changes could be incorporated into the GUI tool to provide even more flexibility for the designer. An efficient floating point version for all four CORDIC architectures could also be quickly developed, since only pre and post processing blocks would be required in order to align the input mantissas and finally perform post-normalization of the floating point values. A further enhancement would be to extend the compensated CORDIC algorithm to data widths up to 64 bits. This would require the development of a heuristic search in order to find $S(i)$ and η_i factors that have an inverse scaling factor that is close to 2^{-1}. All these improvements could be incorporated in the GUI tool to provide even more flexibility for the different CORDIC implementations.

The previous project carried out at RMC concerning floating point CORDIC used a rounding technique in the cross-addition stages of the CORDIC algorithm. An in-depth study of the impact of rounding on precision, amount of hardware and speed could be carried out in order to determine whether rounding or truncation is the most efficient way of implementing the CORDIC algorithm on an FPGA. Another very important area for future development is to use the compensated CORDIC architectures in actual applications such as SoC, image/audio processing, ALUs, SVD or 3D computer graphics, to name only a few. Implementing these algorithms in other types of programmable logic devices from Xilinx or other companies (such as Altera) could also be carried out. Finally, a similar approach for steering the scaling factor to a simple value could be implemented for the hyperbolic coordinate system.

LIST OF REFERENCES

[1] R. Andraka, "A survey of cordic algorithms for fpga based computers," in *Proc. Of the 1998 CM/SIGDA Sixth International Symposium on FPGAs,* (ACM, ed.), pp. 191–200, February 1998.

[2] J. E. Volder, "The cordic trigonometric computing technique," *IRE Transactions on Electronic Computers*, vol. EC-8, pp. 330–334, September 1959.

[3] J. Walther, "A unified algorithm for elementary functions," in *Proceeding, Spring Joint Computer Conference*, vol. 38, pp. 379–385, 1971.

[4] S. Yu and E. S. Jr, "A scaled dct architecture with the cordic algorithm," *IEEE Transactions on Signal Processing*, vol. 50, pp. 160–167, January 2002.

[5] A. M. Despain, "Fourier transform computers using cordic iterations," *IEEE Transactions on Computers*, vol. C-23, pp. 993–1001, October 1974.

[6] N. Hemkumar and J. Cavallaro, "Efficient complex matrix transformations with cordic," *Proceedings of the 11th Symposium on Computer Arithmetic*, pp. 122–129, 29 June-2 July 1993.

[7] S. Wang, V. Piuri, and E. E. Swartzlander, "Hybrid cordic algorithms," *IEEE Transactions on Computers*, vol. 46, pp. 1202–1207, November 1997.

[8] J.-H. Kwak and E. E. Swartzalander, "A new scheme for prediction of rotation directions in cordic processing," in *42nd Midwest Symposium on Circuits and Systems*, vol. 2, pp. 870–873, August 8-11 1999.

[9] S.-F. Hsiao, Y.-H. Hu, and T.-B. Juang;, "A memory-efficient and high-speed sine/cosine generator based on parallel cordic rotations," *IEEE Signal Processing Letters*, vol. 11, pp. 152–155, February 2004.

[10] C.-F. Lin and S.-G. Chen, "A cordic algorithm with fast rotation prediction and small iteration number," in *Proceedings of the IEEE International Symposium on Circuits and Systems,*, vol. 5, pp. 229–232, IEEE, May 31 - June 3 1998.

[11] D. Timmermann, H. Hahn, and B. J. Hosticka, "Low latency time cordic algorithms," *IEEE Transactions on Computers*, vol. 41, pp. 1010–1015, August 1992.

[12] A.-Y. Wu and C.-S. Wu, "A unified view for vector rotational cordic algorithms and architectures based on angle quantization approach," *IEEE Transactions on Circuits and Systems - I: Fundamental Theory and Applications*, vol. 49, pp. 1442–1456, October 2002.

[13] J.-C. Chih and S.-G. Chen, "A fast cordic algorithm based on a novel angle recoding scheme," in *IEEE International Symposium on Circuits and Systems*, pp. IV621–IV624, May 28-31 2000.

[14] N. Takagi, T. Asada, and S. Yajima, "Redundant cordic methods with constant scale factor for sine and cosine computation," *IEEE Transactions on Computers*, vol. 40, pp. 989–995, September 1991.

[15] B. Gisuthan and T. Srikanthan, "Flat cordic: A unified architecture for high-speed generation of trigonometric and hyperbolic functions," in *IEEE Midwest Symposium on Circuits and Systems*, pp. 1414–1417, August 8-11 2000.

[16] S. Wang and E. E. Swartzlander, "Merged cordic algorithm," in *IEEE International Symposium on Circuits and Systems*, vol. 3, pp. 1988–1991, IEEE, April 20 - May 3 1995.

[17] G. Haviland and A. Tuszynski, "A cordic arithmetic processor chip," *IEEE Transactions on Computers*, vol. C-29, pp. 68–79, February 1980.

[18] W. Xingjun, C. Hongyi, and S. Yihe, "Jacobi-type svd and its floating-point realization based on fast rotations," in *Proceedings of the 5th International Conference on Signal Processing*, vol. 1, pp. 583–586, August 21-25 2000.

[19] G. J. Hekstra and E. F. Deprettere, "Floating point cordic," in *Proceedings of the 11th Symposium on Computer Arithmetic*, pp. 130–137, June 1993.

[20] J. R. Cavallero and F. T. Luk, "Floating point cordic for matrix computations," in *IEEE International Conference on Computer Design*, pp. 40–42, 1988.

[21] Z. Liu, K. Dickson, and J. McCanny, "A floating-point cordic based svd processor," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, June 24-26 2003.

[22] J. Duprat and J.-M. Muller, "The cordic algorithm: new results for fast vlsi implementation," *IEEE Transactions on Computers*, vol. 42, pp. 168–178, February 1993.

[23] D. Phatak, "Double step branching cordic: a new algorithm for fast sine and cosine generation," *IEEE Transactions on Computers*, vol. 47, pp. 1037–1040, September 1998.

[24] E. Antelo, J. D. Bruguera, and E. L. Zapata, "Unified mixed radix 2-4 redundant cordic processor," *IEEE Transactions on Computers*, vol. 45, pp. 1068–1073, September 1996.

[25] E. Antelo, J. Villalba, J. D. Bruguera, and E. L. Zapata, "High performance rotation architectures based on the radix-4 cordic algorithm," *IEEE Transactions on Computers*, vol. 46, pp. 855–870, August 1997.

[26] T. Aoki, H. Nogi, and T. Higuchi, "High-radix cordic algorithms for vlsi signal processing," in *IEEE Workshop on Design and Implementation of Signal Processing Systems*, pp. 183–192, November 3-5 1997.

[27] P. Rao and I. Chakrabarti, "High-performance compensation technique for the radix-4 cordic algorithm," *IEE Proceedings on Computers and Digital Techniques*, vol. 149, pp. 219–228, September 2002.

[28] E. Antelo, T. Lang, and J. D. Bruguera, "Very-high radix circular cordic: vectoring and unified rotation/vectoring," *IEEE Transactions on Computers*, vol. 49, pp. 727 – 739, July 2000.

[29] Y. H. Hu, "The quantization effects of the cordic algorithm," *IEEE Transactions on Signal Processing*, vol. 40, pp. 834–844, April 1992.

[30] H. Xiaobo and S. C. Bass, "A neglected error source in the cordic algorithm," in *IEEE International Symposium on Circuits and Systems*, vol. 1, pp. 766–769, May 3-6 1993.

[31] T. Vladimirova and H. Tiggeler, "Fpga implementation of sine and cosine generators using the cordic algorithm," in *Military and Aerospace Applications of Programmable Devices and Technologies International Conference*, pp. 1–11, September 28-30 1999.

[32] V. Kantabutra, "High-radix cordic for vector rotation with pipelined fpga implementation," in *Proceedings of the 6th IEEE International Conference on Electronics, Circuits and Systems*, vol. 2, pp. 1131–1134, Sptember 5-8 1999.

[33] J. Valls, M. Kuhlmann, and K. K. Parhi, "Efficient mapping of cordic algorithms on fpga," in *IEEE Workshop on Signal Processing Systems*, pp. 336–345, Oct 11-13 2000.

[34] S. Vadlamani and W. Mahmoud, "Comparison of cordic algorithm implementations on fpga families," in *Proceedings of the Thirty-Fourth Southeastern Symposium on System Theory*, pp. 192–196, March 18-19 2002.

[35] E. Deprettere, P. Dewilde, and R. Udo, "Pipelined cordic architectures for fast vlsi filtering and array processing," in *IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 9, pp. 250–253, March 1984.

[36] G. Schmidt, D. Timmermann, J. Bohme, H. Hahn, B. Hosticka, and G. Zimmer, "Parameter optimization of the cordic algorithm and implementation in a cmos chip," in *Signal Processing III: Theories and Applications: Proceedings of the EUSIPCO-86 Third European Signal Processing Conference*, pp. 1219–1222, September 2-5 1986.

[37] G. Schmidt, D. Timmermann, H. Hahn, J. Bohme, B. Hosticka, and G. Zimmer, "Design of 16-bit fixed point recursive cordic processors and evaluation tools," in *Signal Processing IV: Theories and Applications Proceedings of EUSIPCO 88 Fourth European Signal Processing Conference*, pp. 1557–1560, September 5-8 1988.

[38] J. Bu, E. F. Deprettere, and F. de Lange, "On the optimization of pipelined silicon cordic algorithm," in *Signal Processing III: Theories and Applications: Proceedings of the EUSIPCO-86 Third European Signal Processing Conference*, vol. 2, pp. 1227–1230, 2-5 September 1986.

[39] G. Gilbert, D. Al-Khalili, and C. Rozon, "Optimized distributed processing of scaling factor in cordic," in *The 3rd International IEEE-NEWCAS Conference*, pp. 35–38, 19-22 June 2005.

[40] J. P. Costello, "Behavioural synthesis of low power floating point cordic processors," m.eng., Royal Military College of Canada, Kingston, ON, April 2000.

Appendix A

Mathematical Derivations

A.1 Double Shift Scaling Factor

The pseudo-rotation equations for the double shift CORDIC are as follows:

$$x_{i+1} = x_i - \sigma_i(2^{-S(i)} + \eta_i 2^{-S'(i)})y_i \tag{A.1}$$

$$y_{i+1} = y_i + \sigma_i(2^{-S(i)} + \eta_i 2^{-S'(i)})x_i \tag{A.2}$$

$$z_{i+1} = z_i - \sigma_i \alpha_i \tag{A.3}$$

where:

$$\alpha_i = \arctan(2^{-S(i)} + \eta_i 2^{-S'(i)}) \tag{A.4}$$

and the scaling factor is given by:

$$K = \prod_{i=0}^{N-1} \cos^{-1} \alpha_i \tag{A.5}$$

From A.4 we have:

$$\tan \alpha_i = \frac{y}{x} = \frac{2^{-S(i)} + \eta_i 2^{-S'(i)}}{1} \tag{A.6}$$

Using the pythagorean theorem, we have:

$$h = \sqrt{x^2 + y^2} = \sqrt{1 + 2^{-2S(i)} + \eta_i 2^{-S(i)-S'(i)+1} + \eta_i^2 2^{-2S'(i)}} \tag{A.7}$$

$$\cos \alpha_i = \frac{x_i}{h} = \frac{1}{\sqrt{1 + 2^{-2S(i)} + \eta_i 2^{-S(i)-S'(i)+1} + \eta_i^2 2^{-2S'(i)}}} \tag{A.8}$$

Finally,

$$K = \prod_{i=0}^{N-1} \sqrt{1 + 2^{-2S(i)} + \eta_i 2^{-S(i)-S'(i)+1} + \eta_i^2 2^{-2S'(i)}} \tag{A.9}$$

A.2 True Compensated CORDIC Scaling Factor

The pseudo-rotation equations for the true compensated CORDIC are as follows:

$$x'_{i+1} = x_i - \sigma_i y_i 2^{-i} \tag{A.10}$$

$$y'_{i+1} = y_i + \sigma_i x_i 2^{-i} \tag{A.11}$$

$$z'_{i+1} = z_i - \sigma_i \alpha_i \tag{A.12}$$

$$k'_{i+1} = k_i(1 + 2^{-2i})^{\frac{1}{2}} \tag{A.13}$$

where $\sigma_i = -1$ if $z_i < 0, +1$ otherwise (Rotation Mode)

followed by a magnitude correction iteration which does not perform any rotation of the vector:

$$x_{i+1} = x'_{i+1} - \eta_i x'_{i+1} 2^{-i} \tag{A.14}$$

$$y_{i+1} = y'_{i+1} + \eta_i y'_{i+1} 2^{-i} \tag{A.15}$$

$$z_{i+1} = z'_{i+1} \tag{A.16}$$

where $\eta_i = +1$ if $k_i < 0, -1$ otherwise

In order to determine the scaling factor after correction, the following relationship relating the compensated vector norm to the uncompensated vector norm is used:

$$k_{i+1} = k'_{i+1} \frac{\sqrt{{x_{i+1}}^2 + {y_{i+1}}^2}}{\sqrt{{x'_{i+1}}^2 + {y'_{i+1}}^2}} \tag{A.17}$$

$${k_{i+1}}^2 = {k'_{i+1}}^2 \left[\frac{{x_{i+1}}^2 + {y_{i+1}}^2}{{x'_{i+1}}^2 + {y'_{i+1}}^2} \right] \tag{A.18}$$

For clarity, let $x'_{i+1} = v$, $y'_{i+1} = w$ and $\eta_i = b$. Then we have:

$${k_{i+1}}^2 = {k'_{i+1}}^2 \left[\frac{(v + bv2^{-i})^2 + (w + bw2^{-i})^2}{v^2 + w^2} \right] \tag{A.19}$$

$${k_{i+1}}^2 = {k'_{i+1}}^2 \left[\frac{v^2 + bv^2 2^{-i+1} + b^2 v^2 2^{-2i} + w^2 + bw^2 2^{-i+1} + b^2 w^2 2^{-2i}}{v^2 + w^2} \right] \tag{A.20}$$

$$k_{i+1}^{2} = k'^{2}_{i+1}\left[\frac{(v^2+w^2)+(v^2+w^2)b2^{-i+1}+(v^2+w^2)b^2 2^{-2i}}{v^2+w^2}\right] \quad (A.21)$$

$$k_{i+1}^{2} = k'^{2}_{i+1}(1+b2^{-i+1}+b^2 2^{-2i}) \quad (A.22)$$

Since $b = \eta_i$, we finally have:

$$k_{i+1} = k'_{i+1}\sqrt{1+\eta_i 2^{-i+1}+\eta_i^2 2^{-2i}} \quad (A.23)$$

A.3 Modified Compensated CORDIC ATR Angles and Scaling Factor

Recall that the basic Given's Transform equations are given by:

$$x_{i+1} = x_i \cos\alpha_i - y_i \sin\alpha_i \quad (A.24)$$

$$y_{i+1} = y_i \cos\alpha_i + x_i \sin\alpha_i \quad (A.25)$$

and the CORDIC rotations (unscaled) can be rewritten as:

$$x_{i+1} = \cos\alpha_i (x_i - \sigma_i y_i 2^{-i}) \quad (A.26)$$

$$y_{i+1} = \cos\alpha_i (y_i + \sigma_i x_i 2^{-i}) \quad (A.27)$$

$$\alpha_i = \arctan 2^{-i} \quad (A.28)$$

In order to adapt the modified compensated CORDIC equations, we have the following relationships:

$$\frac{x_i - \sigma_i y_i 2^{-i} + \eta_i x_i 2^{-i}}{1+\eta_i 2^{-i}} = \left[x_i - \sigma_i y_i \left(\frac{2^{-i}}{1+\eta_i 2^{-i}}\right)\right] \quad (A.29)$$

$$\frac{y_i + \sigma_i x_i 2^{-i} + \eta_i y_i 2^{-i}}{1+\eta_i 2^{-i}} = \left[y_i + \sigma_i x_i \left(\frac{2^{-i}}{1+\eta_i 2^{-i}}\right)\right] \quad (A.30)$$

the Given's Transform equations for the modified compensated cordic become:

$$x_{i+1} = \cos\alpha_i \left[x_i - \sigma_i y_i \left(\frac{2^{-i}}{1+\eta_i 2^{-i}}\right)\right] \quad (A.31)$$

$$y_{i+1} = \cos\alpha_i \left[y_i + \sigma_i x_i \left(\frac{2^{-i}}{1+\eta_i 2^{-i}}\right)\right] \quad (A.32)$$

$$\alpha_i = \arctan\left(\frac{2^{-i}}{1+\eta_i 2^{-i}}\right) = \arctan\left(\frac{1}{\eta_i + 2^{i}}\right) \quad (A.33)$$

thus we have:

$$x_{i+1} = \frac{\cos\alpha_i}{1+\eta_i 2^{-i}}(x_i - \sigma_i y_i 2^{-i} + \eta_i x_i 2^{-i}) \tag{A.34}$$

$$y_{i+1} = \frac{\cos\alpha_i}{1+\eta_i 2^{-i}}(y_i + \sigma_i x_i 2^{-i} + \eta_i y_i 2^{-i}) \tag{A.35}$$

and

$$k_i^{-1} = \frac{\cos\alpha_i}{1+\eta_i 2^{-i}} \tag{A.36}$$

From A.33 we have:

$$\tan\alpha_i = \frac{y}{x} = \frac{2^{-i}}{1+\eta_i 2^{-i}} \tag{A.37}$$

Using the pythagorean theorem, we have:

$$h = \sqrt{x^2+y^2} = \sqrt{(1+\eta_i 2^{-i})^2 + 2^{-2i}} \tag{A.38}$$

$$\cos\alpha_i = \frac{x}{h} = \frac{1+\eta_i 2^{-i}}{\sqrt{1+\eta_i 2^{-i+1} + \eta_i^2 2^{-2i} + 2^{-2i}}} \tag{A.39}$$

thus:

$$k_i^{-1} = \frac{1}{\sqrt{1+\eta_i 2^{-i+1} + \eta_i^2 2^{-2i} + 2^{-2i}}} \tag{A.40}$$

The scaling factor is given by:

$$K = \prod_{i=0}^{N-1} k_i \tag{A.41}$$

and finally:

$$K = \prod_{i=0}^{N-1} \sqrt{1 + 2^{-2i} + \eta_i 2^{-i+1} + \eta_i^2 2^{-2i}} \tag{A.42}$$

Appendix B

Error Analysis Supplement

B.1 Angular Error

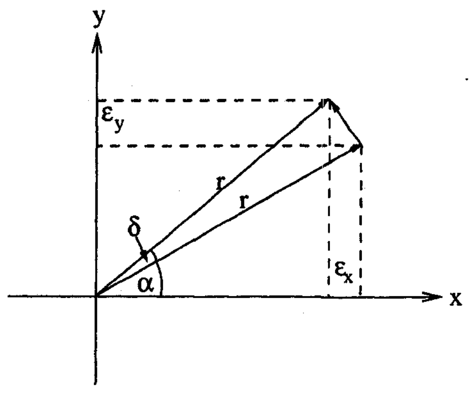


Figure B.1: Graphic Representation of Angular Error

$$\epsilon_x = r\cos\alpha - r\cos(\alpha+\delta) \tag{B.1}$$

$$\epsilon_x = r\cos\alpha - r\left(\cos\alpha\cos\delta + \sin\alpha\sin\delta\right) \tag{B.2}$$

$$\epsilon_x = r\cos\alpha(1-\cos\delta) + r\sin\alpha\sin\delta \tag{B.3}$$

The error will be maximum when $\cos\alpha = 1$ and $\sin\alpha = 0$. Thus, the maximum error is given by:

$$max(\epsilon_x) = r(1-\cos\delta) \tag{B.4}$$

Similarly, for the y component error we have:

$$\epsilon_y = r\sin(\alpha+\delta) - r\sin\alpha \tag{B.5}$$

$$\epsilon_y = r(\sin\alpha\cos\delta + r\cos\alpha\sin\delta) - r\sin\alpha \tag{B.6}$$

$$\epsilon_y = r\sin\alpha(\cos\delta - 1) + r\cos\alpha\sin\delta \tag{B.7}$$

Since we have $\cos\alpha = 1$ and $\sin\alpha = 0$, the resulting error in this case is:

$$\epsilon_y = r\sin\delta \tag{B.8}$$

Note that the situation could be reversed, and we could define the maximum error when $\sin\alpha = 1$ and $\cos\alpha = 0$. In that case, we would have:

$$max(\epsilon_y) = r(\cos\delta - 1) \tag{B.9}$$

$$\epsilon_x = r\sin\delta \tag{B.10}$$

The following proof will show that in order to have b bits of precision in the result, it is required to have the maximum shift $S(N-1) = b$.

We have:

$$1 - \cos\alpha \leq \sin\alpha \;\; \forall\alpha \in [0, +\pi/2] \tag{B.11}$$

Thus:

$$\log_2\left[(1-\cos\alpha)\cdot 2^b\right] \leq \log_2\left[\sin\alpha \cdot 2^b\right] \;\; \forall\alpha \in [0, +\pi/2] \tag{B.12}$$

For very small angles in the shift sequence, we have $\alpha_i = \arctan(2^{-S(i)}) \approx 2^{-S(i)}$. We will be able to ensure b bit precision iff:

$$\log_2(\sin\alpha_i \cdot 2^b) \leq 0 \tag{B.13}$$

Thus we have:

$$\log_2(2^{-S(i)} \cdot 2^b) \leq 0 \tag{B.14}$$

$$-S(i) + b \leq 0 \tag{B.15}$$

$$S(i) \geq b \tag{B.16}$$

i	$2^{-S(i)}$	Shifted Operands	Truncation Error	Accumulated Trunc Error	Guard Bits g(i)
0	1.000000	0	0.000000	0.000000	0
1	0.500000	1	0.500000	0.500000	1
2	0.250000	1	0.750000	1.250000	1
3	0.125000	1	0.875000	2.125000	2
4	0.062500	1	0.937500	3.062500	2
5	0.031250	1	0.968750	4.031250	3
6	0.015625	1	0.984375	5.015625	3
7	0.007813	1	0.992188	6.007813	3
8	0.003906	1	0.996094	7.003906	3
9	0.001953	1	0.998047	8.001953	4
10	0.000977	1	0.999023	9.000977	4
11	0.000488	1	0.999512	10.000488	4
12	0.000244	1	0.999756	11.000244	4
13	0.000122	1	0.999878	12.000122	4
14	0.000061	1	0.999939	13.000061	4
15	0.000031	1	0.999969	14.000031	4
16	0.000015	1	0.999985	15.000015	4
17	0.000008	1	0.999992	16.000008	5
18	0.000004	1	0.999996	17.000004	5
19	0.000002	1	0.999998	18.000002	5
20	0.000001	1	0.999999	19.000001	5
21	0.000000	1	1.000000	20.000000	5
22	0.000000	1	1.000000	21.000000	5
23	0.000000	1	1.000000	22.000000	5
24	0.000000	1	1.000000	23.000000	5
25	0.000000	1	1.000000	24.000000	5
26	0.000000	1	1.000000	25.000000	5
27	0.000000	1	1.000000	26.000000	5
28	0.000000	1	1.000000	27.000000	5
29	0.000000	1	1.000000	28.000000	5
30	0.000000	1	1.000000	29.000000	5
31	0.000000	1	1.000000	30.000000	5
32	0.000000	1	1.000000	31.000000	5
33	0.000000	1	1.000000	32.000000	6
34	0.000000	1	1.000000	33.000000	6

Table B.1: Truncation Error Analysis - Conventional CORDIC

i	$2^{-S(i)}$	Shifted	Truncation	Accumulated	Guard Bits
		Operands	Error	Trunc Error	g(i)
0	1.000000	0	0.000000	0.000000	0
1	0.500000	1	0.500000	0.500000	1
2	0.250000	1	0.750000	1.250000	1
3	0.125000	2	1.000000	2.250000	2
4	0.062500	2	1.125000	3.375000	2
5	0.031250	1	1.156250	4.531250	3
6	0.015625	2	1.187500	5.718750	3
7	0.015625	1	1.203125	6.921875	3
8	0.007813	1	1.210938	8.132813	4
9	0.003906	1	1.214844	9.347656	4
10	0.001953	2	1.218750	10.566406	4
11	0.000977	1	1.219727	11.786133	4
12	0.000488	1	1.220215	13.006348	4
13	0.000244	1	1.220459	14.226807	4
14	0.000122	2	1.220703	15.447510	4
15	0.000061	2	1.220825	16.668335	5
16	0.000031	1	1.220856	17.889191	5
17	0.000015	1	1.220871	19.110062	5
18	0.000008	2	1.220886	20.330948	5
19	0.000004	2	1.220894	21.551842	5
20	0.000004	1	1.220898	22.772739	5
21	0.000002	1	1.220900	23.993639	5
22	0.000001	1	1.220901	25.214540	5
23	0.000000	2	1.220901	26.435441	5
24	0.000000	1	1.220902	27.656343	5
25	0.000000	2	1.220902	28.877245	5
26	0.000000	1	1.220902	30.098147	5
27	0.000000	2	1.220902	31.319049	5
28	0.000000	1	1.220902	32.539951	6
29	0.000000	1	1.220902	33.760853	6
30	0.000000	1	1.220902	34.981755	6
31	0.000000	1	1.220902	36.202657	6
32	0.000000	1	1.220902	37.423559	6
33	0.000000	2	1.220902	38.644461	6
34	0.000000	1	1.220902	39.865364	6
35	0.000000	1	1.220902	41.086266	6
36	0.000000	1	1.220902	42.307168	6

Table B.2: Truncation Error Analysis - Compensated CORDIC