The Dissertation Committee for Jason Todd Arbaugh certifies
that this is the approved version of the following dissertation:

Table Look-up CORDIC:
Effective Rotations Through Angle Partitioning

Committee:

_____
Earl E. Swartzlander, Jr., Supervisor

_____
Anthony P. Ambler

_____
John H. Davis

_____
Chang Yong Kang

_____
Nur A. Touba

Table Look-up CORDIC:
Effective Rotations Through Angle Partitioning

by

Jason Todd Arbaugh, B.S., M.S.

Dissertation

Presented to the Faculty of the Graduate School of
the University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the degree of

Doctor of Philosophy

The University of Texas at Austin
December 2004

This dissertation is dedicated to all of the teachers in my life: those who have taught me theory in my classrooms, application in my jobs in industry, hard work in my activities, integrity in my interactions, and balance in my life.

Acknowledgements

There are many people that deserve acknowledgement for their help in completing this dissertation. The help they provided, while not always technical in nature, was sorely needed at various times during my research, implementation, writing, and editing.

I would like to thank Dr. Earl E. Swartzlander, Jr. for all of his teaching, help, and encouragement in graduate school. From his class on High Speed Arithmetic to his advice as my supervisor, I have enjoyed his humor, imagination, knowledge, and support. I have, and will continue to appreciate his mentorship. And even when he insists that I call him Earl, I will *always* regard him as Dr. Swartzlander.

I am also thankful to the members of my dissertation committee Dr. Tony Ambler, Dr. John Davis, Dr. Chang Yong Kang, and Dr. Nur Touba. In addition to their time, which is so very valuable in today's hectic world, they have provided valuable insights and timely suggestions that have widened the scope and improved the content and quality of my dissertation.

Brian P. Klawinski, Esquire, provided seemingly instantaneous assistance in the development of several of the C++ programs used in the calculation, evaluation, and verification of the Table Look-up CORDIC algorithm.

Mr. Paul T. Muehr helped with the setup of the backend tools. Without his help with the environment, scripts for the synthesis, auto place and route, and static timing analysis of my Verilog, I would not have been able to meet my deadlines.

In addition to my committee and supervisor, Mr. Charles Pummill and Mrs. Sherri Staley assisted in reading, critiquing, and improving this dissertation. Their help on my dissertation has been greatly appreciated.

My close friend, Mr. Chris W. Mobley, helped keep me motivated and healthy through the many long days of sitting in front of the computer typing. The

games of racquetball, the workouts, the pep talks, the reality checks, and the social interaction have kept me from going completely insane.

I would like to thank Dr. Dhananjay Phatak, Associate Professor at the University of Maryland, Baltimore County, for providing my a copy of some of the C code used in simulating his Double Step Branching CORDIC algorithm. The C code was very helpful for figuring out some of the nuances of the algorithm that were not covered in the published paper.

And finally, I would like to thank my entire family for helping make me who I am today: my mother and father, Sherri E. Staley and Wayne L. Arbaugh, my stepparents, Dr. Leo G. Staley and Kathy Arbaugh, my brothers and step-sister, “Huck” Fenn J. Arbaugh, Dr. Jesse L. Arbaugh, and Kristi Bogans.

Table Look-up CORDIC:
Effective Rotations Through Angle Partitioning

Publication No. _____

Jason Todd Arbaugh, Ph.D.
The University of Texas at Austin, 2004

Supervisor: Earl E. Swartzlander, Jr.

This dissertation documents the development, derivation, verification, implementation, and evaluation of an improved version of the COordinate Rotation DIgital Computer (CORDIC) algorithm for calculating sine and cosine values. The CORDIC algorithm was originally developed to calculate trigonometric relationships in navigation systems using a family of linearly converging iteration equations. The CORDIC algorithm computes numerous elementary functions including powers, exponentials, logarithms, trigonometric and hyperbolic functions.

Many different versions of the classic CORDIC algorithm have been developed to enhance the performance of calculating these elementary functions. These alternative algorithms utilize methods that vary from using different number systems, to increasing the number of rotations performed in each iteration, to calculating the rotations using different Arc Tangent Radices. Even though each of

these methods improves the performance of the CORDIC calculations, they still require a significant number of iterations through the CORDIC equations to obtain the final answer.

The new CORDIC algorithm utilizes look-up tables and standard microprocessor arithmetic functional units to perform the calculations. The look-up tables employ either the traditional CORDIC or the new Parallel Arc Tangent Radix (ATR). The traditional CORDIC ATR combines multiple CORDIC iterations into a single effective rotation. The parallel ATR uses the exact angle value to perform the rotation rather than a summation of CORDIC ATR angles. Utilizing exact angles reduces the complexity of the decoders and permits parallel access to the ROMs.

The Table Look-up CORDIC (TLC) algorithm is shown to be correct through the development of a mathematical proof utilizing the polar form of the CORDIC iteration equations. The TLC algorithm and other versions of the CORDIC algorithm are implemented in MatLab and simulated. The results of these simulations are compared with the bit correct value calculated with MatLab's built in trigonometric functions to verify the correct operation.

The same CORDIC algorithms are then modeled in Verilog. The Verilog models are synthesized to gates, placed and routed, and statically timed. The auto place and route of these circuits allows area estimates to be obtained for the different algorithms. The static timing analysis allows the worst-case path to be timed for frequency and latency comparisons.

Table of Contents

List of Tables

List of Figures

List of Supplemental Files

Chapter 1
Introduction

Elementary functions are found in every area of mathematics, engineering, physics, and science. Due to the numerous ways in which they are used, elementary functions have been, and will continue to be, some of the most frequently evaluated functions in digital computations. Because elementary functions are frequently evaluated, their computation needs to be fast and accurate without requiring excessive system resources.

1.1 Elementary Functions

A function is an elementary function if it can be constructed from a finite combination of constant functions, field operations (addition, subtraction, multiplication, or division), algebraic (x^n), exponential (e^n), logarithmic ($\log_n(x)$) functions and their inverses [1]. Some of the most common elementary functions are the trigonometric ($\sin(x)$, $\cos(x)$, and $\tan(x)$) and the hyperbolic functions ($\sinh(x)$, $\cosh(x)$, and $\tanh(x)$) and their inverses.

Many current and future applications depend upon the accurate calculation of elementary functions. Trigonometric functions, such as sine and cosine, are especially important. Whether it is the ENIAC calculating shell trajectory tables [2], the HP35 hand held calculator [3] [4], Singular Value Decomposition (SVD) algorithms [5] [6], computer graphics applications [7] [8] [9], Digital Signal Processing (DSP) systems [10] [11] [12], digital communications [13] [14] [15], adaptive filters [16] [17], or robotic movement [18] [19] [20], trigonometric functions are repeatedly required during the course of their operation.

In order to correctly calculate their results, each of these applications requires accurate trigonometric values for every possible angle. Whether the functions are implemented in software or hardware, the calculation of trigonometric values is a complex and time consuming process. Often, the computation time for obtaining the trigonometric values can dominate the execution time of the algorithm.

System performance can also be degraded if the calculations following the trigonometric function use the value it returns. If this happens, the system will not be able to proceed until the trigonometric calculation is complete. Table 1.1 shows the latency and error for some of the elementary functions in Intel's IA-64 processor [21]. The cube root (cbrt), exponential (exp), and natural logarithm (ln) all have large latencies, but the latencies of the trigonometric functions (sin, cos, tan, and atan) have the highest latencies. This underscores the fact that reducing the latency of trigonometric calculations will significantly improve the performance of any system that calculates these functions.

Table 1.1 – *Latency and Error of IA-64 Elementary Functions*

FUNCTION NAME	**LATENCY (cycles)**	**ERROR (*ulp*s)**
cbrt	60	0.51
exp	60	0.51
ln	52	0.53
sin	70	0.51
cos	70	0.51
tan	72	0.51
atan	66	0.51

1.2 Previous Research

Since the advent of modern digital computers, machines such as the ENIAC have used trigonometric functions to produce their results. Each of these digital computers has had to develop an algorithm to calculate trigonometric functions for use in computations. The ENIAC had twenty registers, each ten decimal digits wide. For correct computations, a trigonometric function had to return a binary number with an accuracy of ten decimal digits.

Even though the ENIAC had a function table where constants could be input by setting switches, there was not enough room to store the many different values needed to implement a single bit accurate trigonometric function. Implementing all of the trigonometric functions and their inverses was equally impossible. To work around this problem, the original ENIAC algorithms were coded to use a single angle so only two constants would need to be stored, one for sine and one for cosine. This program was then run for each required angle. If the program could not be written in a way that only used a single angle, the trigonometric functions were implemented in software.

To calculate the sine or cosine of an angle in software, the ENIAC programmers could use the infinite sum or the infinite product formulas for these functions. These mathematical definitions of sine and cosine can be used to generate the correct values of sine and cosine out to any bit length, as long as enough terms are added. The infinite sums for calculating sine and cosine are shown in Equations 1.1 and 1.2. The infinite products for calculating sine and cosine are shown in Equations 1.3 and 1.4.

$$\sin(\theta) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{(2n-1)!} \theta^{2n-1} \tag{1.1}$$

$$\cos(\theta) = \sum_{n=0}^{\infty} \frac{(-1)^{n}}{(2n)!} \theta^{2n} \tag{1.2}$$

$$\sin(\theta) = \theta \prod_{n=1}^{\infty} \left[1 - \frac{\theta^2}{\pi^2 n^2} \right] \tag{1.3}$$

$$\cos(\theta) = \prod_{n=1}^{\infty} \left[1 - \frac{4\theta^2}{\pi^2 (2n-1)^2} \right] \tag{1.4}$$

An examination of these formulas shows the problems that are encountered when they are coded. The first problem is the complexity required to calculate each term within the series. In order to calculate each term, a power operation and a division must be performed. For the infinite series term, a factorial must be computed while the infinite product term requires a multiplication and a subtraction. Obtaining the correct value for a single term requires a large amount of computation. Calculating several terms and combining them requires an even more significant amount of computation.

The second problem is determining how many terms are required to calculate a bit correct answer. Unfortunately, there is not a deterministic way to make this evaluation. Some angle calculations, such as $\sin(0)$ or $\cos(0)$, require only a single term to obtain the correct answer from any of the formulas. Other angle calculations, such as $\sin(\pi/4)$ or $\cos(\pi/4)$, require an infinite number of terms to produce a bit correct answer.

To handle this problem, conditional loops are placed in the code. When a new term is calculated, it is added to or multiplied with, the previous terms in the series. This new value is compared against the previously calculated answer. If the difference between the two numbers is zero, the correct answer has been reached and the loop is exited. If the difference is not zero, the process is repeated.

Using this method, the ENIAC could calculate bit accurate trigonometric functions for use in other calculations. The problem with this method is that it requires a large number of complex and time-consuming calculations to obtain each

new term. Each new term then had to be combined with the previous terms before a new term could be calculated.

As technology has advanced, computers have added more registers and larger memories. In parallel with these technological advances, the number of binary bits used to represent a number has increased as well. These increases in bit width have outpaced technology's capability of providing a Read Only Memory (ROM) large enough to store all of the trigonometric values for all of the representable angles. Due to the fact that a ROM this size is impractical to build, this complex series of calculations had to be performed each time a trigonometric value was needed.

In 1624, Henry Briggs published his mathematical treatise, *Arithmetica Logarithmica*, in which he describes an algorithm to calculate sine and cosine using shift and addition [4], [22], [23]. This algorithm was implemented digitally when Jack Volder developed the COordinate Rotation DIgital Computer (CORDIC) algorithm in 1959 to calculate the trigonometric relationships of navigation equations [24]. Through careful selection of the scale factor, Arc Tangent Radix, and initial conditions, Volder was able to develop a family of iterative equations that only required shifts and adds to calculate the trigonometric functions in a deterministic number of operations.

Many versions of the CORDIC algorithm have been developed with the intention of improving the speed with which the algorithm can compute trigonometric results. The speed improvements to the CORDIC algorithm have been achieved in diverse and creative ways. Some variations improve the calculation speed by improving the hardware used to perform the iterative calculations. Other variations use different number representations or attempt to calculate multiple rotations in each iteration. Improvements have also been obtained by using new Arc Tangent Radices that eliminate the serial nature of the calculations. But no matter what improvement is applied to the CORDIC algorithm, it is still based on Jack Volder's original shift and add algorithm.

1.3 Table Look-up CORDIC

This dissertation documents the development, derivation, verification, implementation, and evaluation of an improved version of the COordinate Rotation DIgital Computer (CORDIC) algorithm for calculating sine and cosine.

The new CORDIC algorithm utilizes look-up tables and standard microprocessor arithmetic functional units to perform the calculations. The look-up tables implement either the traditional CORDIC or the new Parallel Arc Tangent Radix (ATR). Each entry in the traditional CORDIC ATR combines multiple CORDIC iterations into a single effective rotation. Combining these rotations divides the angle domain into separate partitions between the critical angles. All of the angles in an individual partition are rotated in the same direction in all of the iterations represented in the table.

The Parallel ATR improves upon the traditional CORDIC ATR by rotating the vector by the exact value of the angle. This provides several significant benefits for a designer implementing the Table Look-up CORDIC algorithm. First, the complexity of the ROM decoder is greatly simplified. Second, all ROM look-up tables can be accessed simultaneously without intermediate computations to determine residual angles. And finally, the number of computations required to obtain the final answer is reduced.

The Table Look-up CORDIC (TLC) algorithm is shown to be correct through the development of a mathematical proof utilizing the polar form of the CORDIC iteration equations. The TLC algorithm and other versions of the CORDIC algorithm are implemented in MATLAB and simulated. The results of these simulations are compared to verify the new algorithm's operation. The same CORDIC algorithms are then modeled in Verilog. The Verilog models are then synthesized into gates, routed, and statically timed. The auto place and route of these circuits allowed area estimates to be obtained for the different algorithms. The auto

place and route allows silicon areas to be compared while the static timing analysis allows the worst-case path to be timed for frequency comparisons.

The organization of this dissertation is as follows: Chapter 2 gives an overview of some classes of algorithms that can be used to calculate Elementary functions. Chapter 3 details the development of the classic CORDIC algorithm and highlights the consequences of the design decisions. In Chapter 4, some of the previous work to enhance the CORDIC algorithm is described. The development of the Table Look-up CORDIC algorithm and a mathematical proof of its correctness are described in Chapter 5. The generation of the tables used in the Table Look-up CORDIC algorithm is contained in Chapter 6. The modeling of these CORDIC algorithms in MATLAB and Verilog are covered in Chapter 7 and Chapter 8 respectively. Chapter 9 details the process of logic synthesis and auto place and route for the different Verilog models. The dissertation ends in Chapter 10 with a discussion of conclusions and areas of future research.

Chapter 2
Algorithm Classes

There are a large number of algorithms that can be used to calculate the various trigonometric functions. These algorithms can be classified by the manner in which they perform their computations. Four classes of algorithms will be discussed in the following sections. These classes are the polynomial approximation algorithms, rational approximation algorithms, linear convergence algorithms, and quadratic convergence algorithms. Although this dissertation only examines variations of the CORDIC algorithm, a linear convergence algorithm, it is important to understand the implementation and computation time of the other classes of algorithms so that valid performance and architectural comparisons can be made between the algorithms.

2.1 Polynomial Approximation

A polynomial approximation, $P(x)$, is a degree-*n* polynomial of the form shown in Equation 2.1 [25]. This polynomial is used to approximate a function over the interval of interest. The degree of the polynomial, n, depends upon the amount of error that can be allowed in the calculation. Polynomials of higher degrees generate less error, but they obtain this precision at the expense of computation time.

$$f(x) \approx P(x) = p_n x^n + p_{n-1} x^{n-1} + \ldots + p_1 x + p_0 \tag{2.1}$$

The coefficients of each term of the polynomial are selected to minimize the average error between the polynomial and the actual function. Normally, a standard least squares or Chebyshev approximation is used to calculate the coefficients of the

polynomial. Sometime the function being approximated has regions where there are sharp changes in its slope. These regions are more difficult to accurately model than regions with small changes or no change in the slope. In order to model these intervals without significantly increasing the degree of the polynomial, weighting functions are used to ensure a more precise match. Two of the more common weighting functions in use are Legendre and Jacobi functions [26].

The polynomial approximation class of algorithms is one of the easiest to implement within a digital system. Once the degree and coefficients of the polynomial have been selected, the values of the coefficients are stored in a ROM. These coefficients are used to calculate the given function any time it is needed.

It is possible to reduce the required order of the polynomial by breaking the function into several intervals. Although this will increase the number of ROM tables needed to hold the coefficients, it will reduce the total number of calculations that must be performed. The increased speed of the implementation is obtained at the expense of increased silicon area.

Even with intelligently subdivided intervals, the approximation will still be a degree-m polynomial, where $m \le n$. It is possible to rearrange the polynomial to minimize the number of multiplications that must be performed. Applying Horner's scheme [26] to the polynomial, $P(x)$, it can be re-written as shown in Equation 2.2. Using Horner's scheme, a degree-n polynomial requires n multiplications and n additions. The total computation time required by this calculation is $n \cdot t_{mult} + n \cdot t_{add}$ where n is the degree of the polynomial, t_{mult} is the time to perform a multiplication, and t_{add} is the time to perform an addition.

$$P(x) = ((((p_n x + p_{n-1})x + p_{n-2})\ldots)x + p_1)x + p_0 \tag{2.2}$$

In addition to Horner's scheme, there are several other computation minimization techniques that can be applied to polynomial approximations. Two of these techniques are the E-Method [27] and Estrin's Method [28]. Some techniques are only useful for polynomials of specific degrees or polynomials that only contain even or odd powers.

Koren and Zinaty researched the degrees and coefficients required by rational approximations in order to achieve errors less than 1 *ulp*, unit in the last position, in 32-bit binary numbers [29]. Even though the research has not been published on the requirements for polynomial approximations, it is possible to draw some conclusions by examining papers that have been published. AMD's K5 microprocessor implements the elementary functions through the use of polynomial approximations stored in ROM tables [30]. Unfortunately, no mention is made of the degree of these polynomials or their coefficients.

Intel's IA-64 architecture also uses polynomial approximations for calculating elementary functions [21]. The number of polynomials and the degrees of the polynomials are given within the descriptions of each of the functions presented. The calculation of sine or cosine requires the calculation of two polynomials, one with a degree of eight and the other with a degree of nine. These two numbers are multiplied by separate coefficients and then combined using a formula that requires several additions.

2.2 Rational Approximation

A rational approximation, $R(x)$, is the ratio of two polynomials, $P(x)$ and $Q(x)$, of degree-*n* and degree-*m* respectively. The general form of a rational approximation is shown in Equation 2.3 [25]. This ratio is used to approximate the function over the interval of interest. With the addition of the second polynomial, $Q(x)$, higher accuracy can be achieved with lower degree polynomials. This reduces

the number of multiplications and additions required to obtain the answer, but it introduces a division operation.

$$f(x) \approx R(x) = \frac{P(x)}{Q(x)} = \frac{p_n x^n + p_{n-1} x^{n-1} + \ldots + p_1 x + p_0}{q_m x^m + q_{m-1} x^{m-1} + \ldots + q_1 x + q_0} \tag{2.3}$$

Equation 2.3 can be rearranged using Horner's scheme to minimize the number of multiplications and additions that must be performed. The result of applying Horner's scheme to the polynomials $P(x)$ and $Q(x)$ in Equation 2.3 can be seen in Equation 2.4. The total computation time required for this calculation is $(n+m)\cdot t_{mult} + (n+m)\cdot t_{add} + t_{div}$ where n is the degree of the polynomial $P(x)$, m is the degree of the polynomial $Q(x)$, t_{mult} is the time to perform a multiplication, t_{add} is the time to perform an addition, and t_{div} is the time to perform a division.

$$R(x) = \frac{P(x)}{Q(x)} = \frac{((((p_n x + p_{n-1})x + p_{n-2})\ldots)x + p_1)x + p_0}{((((q_n x + q_{n-1})x + q_{n-2})\ldots)x + q_1)x + q_0} \tag{2.4}$$

The selection of coefficients for rational approximations is performed in the same manner as for polynomial approximations. Weighting functions can be applied to achieve greater accuracy on certain intervals of the function if greater accuracy is required. Rational approximations are also straightforward to implement in digital systems with ROM tables and temporary storage. As with polynomial approximations, the degree of $P(x)$ and $Q(x)$ can be further reduced by breaking the range into smaller intervals. Breaking the polynomials into intervals increases the size of the look-up table just as it does for the polynomial approximation.

The biggest drawback for implementing rational approximations is the division that must be performed once $P(x)$ and $Q(x)$ have been calculated. Division

operations are one of the most time consuming instructions in any computational hardware [31]. A careful analysis should be performed between the savings obtained by reducing the degree of the polynomials and the increased execution time of the division operation.

Research into the use of rational approximations for calculating many of the elementary functions has indicated that good results can be obtained for a 32-bit number by using two fifth-degree polynomials [29]. The generalized form of this rational approximation, written out using Horner's method, is shown below in Equation 2.5. Using this implementation, the calculation requires ten multiplications, ten additions, and one division. The computational time can be expressed as $10 \cdot t_{mult} + 10 \cdot t_{add} + t_{div}$, where t_{mult} is the time an ALU takes to perform a multiplication, t_{add} is the time it takes and ALU to perform an addition, and t_{div} is the time it takes an ALU to perform a division.

$$R(x) = \frac{((((p_5 x + p_4)x + p_3)x + p_2)x + p_1)x + p_0}{((((q_5 x + q_4)x + q_3)x + q_2)x + q_1)x + q_0} \tag{2.5}$$

The choice between using polynomial or rational approximations is not a clear-cut decision. Functions that have poles, finite limits at $\pm\infty$, or infinite derivatives will not be accurately represented by polynomial approximations [26]. This does not automatically mean that using a rational approximation would be the best solution. Due to the large latency times of division, significantly higher order polynomial approximations can be computed in the same execution time.

It has been argued that the division function is so seldom used that it does not make sense to implement fast division algorithms. Oberman and Flynn have argued that the results of waiting for a slow division to complete are so catastrophic to overall performance that fast division must be implemented [32]. If Oberman and

Flynn are correct, fast division units could significantly improve overall system performance and might be provided in future generations of microprocessors.

2.3 Linear Convergence

A linear convergence algorithm is a family of iteration equations where the next value for each variable in the equation is based upon the current value of the variables. A single iteration through the family of equations refines the accuracy of the variables as the equations linearly converge upon the correct answer. An example of a family of iterations can be seen in Equations 2.6, 2.7, and 2.8. At least two independent equations are required.

$$x_{i+1} = f(x_i, y_i, z_i) \tag{2.6}$$

$$y_{i+1} = g(x_i, y_i, z_i) \tag{2.7}$$

$$z_{i+1} = h(x_i, y_i, z_i) \tag{2.8}$$

The biggest problem with linear convergence algorithms is their speed of convergence. As the name suggests, the convergence speed of this category of algorithm is linear. The time to compute the correct answer is a linear function of the number of bits of precision required by the digital system.

The CORDIC algorithm is an example of a linear convergence algorithm. In order to obtain an accurate answer for an *n*-bit binary number, *n* iterations of the equations must be performed. Even though this might not sound very time consuming, it can have a significant performance impact on an algorithm that requires the trigonometric results. Consider any program that requires the use of double precision floating point numbers such as a drafting tool for high precision parts, a simulation of high-energy physics, or even a high definition computer

animation for a feature film. Each of these programs uses one or more trigonometric functions to model a chamfer, particle trajectory, or lighting effect.

The format of the double precision floating-point number that the program uses is defined by IEEE specification 754 [33]. This specification dictates that fifty-two bits will be used to represent the mantissa of the double precision floating-point number. These are the fifty-two bits following the first binary 1 in the number. This means that there are actually fifty-three bits in the mantissa of a double precision floating-point number, fifty-two bits that are stored and the one hidden bit that is understood to be there.

Because there are fifty-three bits in the mantissa, the CORDIC algorithm requires fifty-three iterations to obtain an accurate answer. In reality, several more iterations are required to ensure that the final answer is bit correct after rounding. From this example, it is obvious that if many trigonometric functions are required, a large number of iterations and time will be required to obtain the correct answer.

2.4 Quadratic Convergence

Just like a linear convergence algorithm, a quadratic convergence algorithm is a family of iteration equations where the next value for each variable in the equation is based upon the current value of the variables. An example of a quadratic family of iterations can be seen in Equations 2.9, 2.10, and 2.11.

$$x_{i+1} = j(x_i, y_i, z_i) \tag{2.9}$$

$$y_{i+1} = k(x_i, y_i, z_i) \tag{2.10}$$

$$z_{i+1} = l(x_i, y_i, z_i) \tag{2.11}$$

The difference between a linear convergence algorithm and a quadratic convergence algorithm is the speed with which they converge upon the correct

answer. As the name suggests, this category of algorithms converges upon the correct answer quadratically. The time to compute the correct answer for this category of algorithm is a logarithmic function of the number of bits of precision required by the digital system. Even though quadratic convergence equations only require $\lceil \log_2 n \rceil$ iterations to converge upon the correct answer, this can still represent a significant number of iterations if *n* is large.

Unfortunately, many quadratic convergence equations are made up of complex operations that require significant amounts of computation time to calculate. In 1976, Richard Brent published a paper describing quadratic convergence algorithms for many different elementary functions [34]. The quadratic convergence algorithm for computing $\tan^{-1}(\theta)$ is shown in Figure 2.1.

$S = 2^{\frac{-n}{2}};$
$V = \theta / \left(1 + \sqrt{1+\theta^2}\right);$
$Q = 1;$

while $(1-S) > 2^{-n}$ **do**
$Q = 2Q/(1+S);$
$W = 2SV/(1+V^2);$
$W = W/\left(1+\sqrt{1-W^2}\right);$
$W = (V+W)/(1-VW);$
$V = W/\left(1+\sqrt{1+W^2}\right);$
$S = 2\sqrt{S}/(1+S);$
end

return $Q \cdot \log((1+V)/(1-V));$

Figure 2.1 – *Quadratic Convergence Algorithm*

Brent's algorithm requires two shift operations, nine additions, five multiplications, six divisions, and three square roots to be performed during each

iteration of this arctangent algorithm. In addition to these operations, another shift, two additions, one division, and one square root have to be performed during the initialization of the algorithm. The calculation of the final answer also requires two more additions, one multiplication, one division, and one logarithm. This makes Brent's quadratic convergence algorithm for arctangent a very costly implementation in terms of system resources and time. Even using a ROM table to provide an initial approximation to the answer in order to reduce the number of iterations required for convergence, the algorithm still requires significant computation time to converge upon the final answer.

2.5 Research Opportunities

Polynomial and rational approximation algorithms have been studied in detail by many researchers. With the advances in computer processing power, the selection of the degree of the polynomials and the appropriate coefficients has been automated by mathematical problem solving tools like MAPLE [26]. By providing the function to approximate, the range of interest, and the desired error bound, MAPLE can determine the appropriate degrees for the polynomials and the coefficients that will produce suitably accurate approximations. Research opportunities in polynomial and rational approximations are in the domain of coefficient selection or selectively combining polynomials to approximate a given function, but not how the actual computations are performed.

Linear convergence algorithms provide many opportunities to enhance operation through the modification of the basic algorithms. Techniques for combining the iterative equations or modifying the underlying assumptions to improve performance are continually being developed. The proliferation of research into the CORDIC algorithm demonstrates that there are still a large number of improvements to be investigated.

Quadratic convergence algorithms have not been fully developed due to the complexity of the operations required to implement them. Though there is ample room for improvement, the primary improvements that must be made are in the theoretical realm. If a family of iterative equations is developed with relatively simple operations, multiple implementations will quickly follow.

Chapter 3
Classic CORDIC Algorithm

The COordinate Rotations DIgital Computer (CORDIC) algorithm was originally developed to replace the limited accuracy analog driven navigation system of the B-58 bomber [35]. This replacement was necessary because the analog methods could not provide accurate results for flights near the North Pole and were too slow in providing solutions for star fixing and radar ground sightings. The trigonometric algorithms being used by the analog system were too slow to meet the B-58's real-time requirements. The CORDIC algorithm was developed to provide a purely digital solution to these navigation problems. The CORDIC algorithm is an iterative family of equations that is used to calculate vectors or angles, depending on the mode in which they are used. The CORDIC algorithm is classified as a linear convergence algorithm, requiring *n*-iterations for *n*-bits of accuracy.

Obtaining the correct value of a trigonometric function is always important. When dealing with navigation, it is imperative to obtain the correct answer. Because of the distances that can be involved, even an error of only a single *ulp* of a trigonometric function can cause catastrophic errors in positioning. The best way to prevent an error in calculating a trigonometric function is to have a ROM table with entries for all possible angles. Each entry in the table is bit correct to less than 0.5 *ulp*. If a small number of angles are required, the trigonometric function can be implemented as a ROM table. Due to the number of angles required by navigation systems, a ROM table for all of the angles is not practical using today's technology.

In 1959, Jack Volder published the definitive paper on the COordinate Rotation DIgital Computer (CORDIC) algorithm. The CORDIC algorithm allows for the calculation of the sine and cosine functions using its rotation mode. These trigonometric functions, as well as others, can be precisely calculated to any bit

length that is required as long as the equations are iterated through enough times and the adder is wide enough to provide a guard band for correct rounding.

Volder's original paper on the CORDIC algorithm [24] explains its operation and highlights several of the major design decisions that were required to make the algorithm possible. Volder's paper on the birth of CORDIC [35] emphasizes the importance of the selection of the appropriate Arc Tangent Radix that makes it possible. Even though neither of Volder's papers provides the full explanation of the algorithms original development or a detailed mathematical derivation of the CORDIC algorithm, this chapter attempts to show its development as a series of logical design tradeoffs.

3.1 The Unit Circle

A vector is a line segment that represents a magnitude and a direction. If the magnitude of the vector is equal to a unit length, it is known as a unit vector. The unit vector is used in many areas of science, but its most common use is to define coordinate systems. Within the field of mathematics, one of its uses is defining the unit circle. If the tail of the unit vector is located at the origin of the *x-y* plane and the unit vector is rotated through every angle from $-\pi$ to π, the path of the head of the unit vector inscribes the unit circle.

The unit circle is used to define the trigonometric or circular functions over all real numbers. A unit vector from the origin $(0,0)$ to the point $(1,0)$ is defined to have a rotation angle of zero. All positive angles are found by rotating the unit vector counterclockwise, while all negative angles are found by rotating the unit vector clockwise. A full revolution in either direction requires a rotation of 2π.

Examining a generic rotation can show the utility of using the unit vector and unit circle for calculating trigonometric functions. Figure 3.1 shows a unit vector with a rotation of θ radians. The head of the unit vector intersects the unit circle at

point (x, y). Using the unit vector as the hypotenuse, a right triangle can be constructed inside the unit circle. A line parallel to the y – axis from the point (x, y) to the x – axis creates one side of the right triangle. This side of the right triangle is called the opposite side because it is opposite the rotation angle, θ. A line on top of the x – axis from the origin $(0,0)$ to the location where the opposite side intersects the x – axis creates the other side of the right triangle. This side is known as the adjacent side because it is adjacent to the rotation angle, θ.

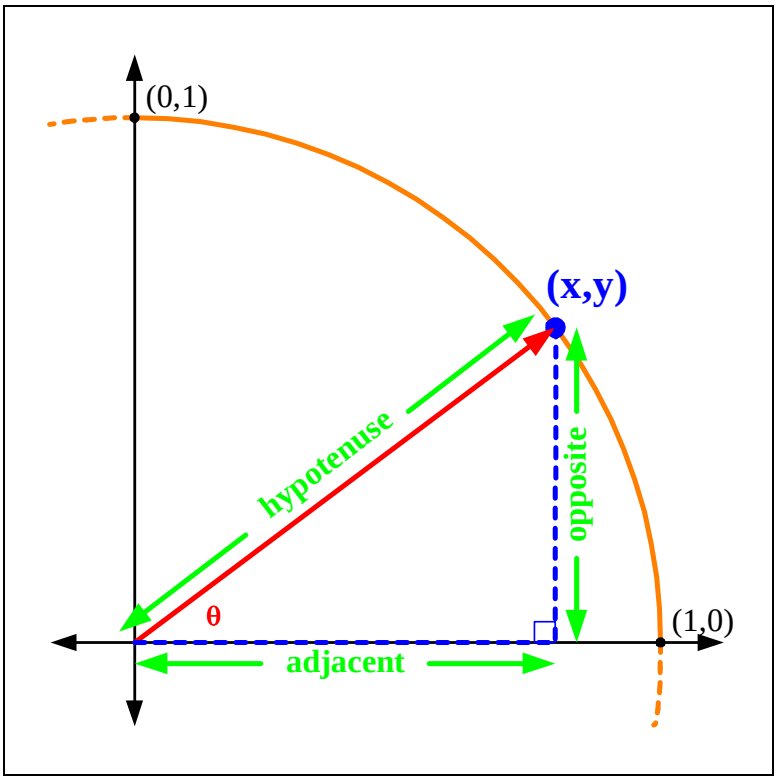


Figure 3.1 – *The Unit Circle*

Using Figure 3.1, the well-known formulas for cosine and sine can be developed. The cosine of angle θ is defined as the ratio of the adjacent side to the

hypotenuse while the sine of angle θ is defined as the ration of the opposite side to the hypotenuse. Because the hypotenuse of this right triangle is the unit vector, the length of the hypotenuse is one. Using this identity in the ratios simplifies the definitions of cosine and sine as shown in Equations 3.1 and 3.2.

$$\cos(\theta) = \frac{adjacent}{hypotenuse} = \frac{adjacent}{1} = adjacent \tag{3.1}$$

$$\sin(\theta) = \frac{opposite}{hypotenuse} = \frac{opposite}{1} = opposite \tag{3.2}$$

Equations 3.1 and 3.2 show that when using the unit circle, the cosine of an arbitrary angle θ is the length of the adjacent side while the sine of the arbitrary angle θ is the length of the opposite side. If the lengths of the sides of the right triangle are known, then values of the sine and cosine of angle θ are also known. Because the unit vector intersects the unit circle at point (x, y), it can be shown that the length of the adjacent side of the right triangle is x and that the length of the opposite side of the right triangle is y. Substituting these values into Equations 3.1 and 3.2 produces the identities of $\cos(\theta) = x$ and $\sin(\theta) = y$ for the unit circle.

3.2 Calculation by Rotation

Using the unit vector and unit circle as a model, the equation for a generic rotation can be developed. Figure 3.2 shows a random unit vector that has an initial rotation of α. The tail of the unit vector is located at the origin, while the head is located at point (x_i, y_i). Using a unit circle and the well known identities for cosine and sine derived in the previous section, the position of the head of the unit vector can be expressed in terms of $\cos(\alpha)$ and $\sin(\alpha)$ as shown in Equations 3.3 and 3.4.

Even though Figure 3.2 shows the unit vector at angle α in the first quadrant of the Cartesian coordinate system, it can be located in any quadrant of the unit circle.

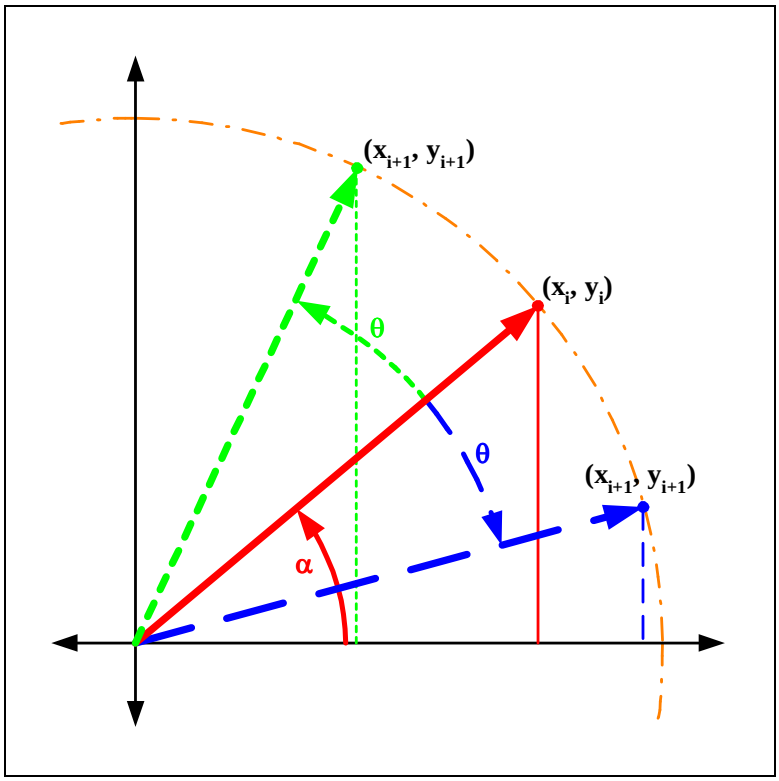


Figure 3.2 – *Generic Unit Vector Rotation*

$$\cos(\alpha) = \frac{x_i}{1} = x_i \tag{3.3}$$

$$\sin(\alpha) = \frac{y_i}{1} = y_i \tag{3.4}$$

Rotating the unit vector by angle θ will move the head of the vector to a new location on the unit circle. If the unit vector is rotated in a positive direction, the location of the head of the vector will be at the summation of angles α and θ. If the unit vector is rotated in a negative direction, the location of the head of the unit

vector will be at the difference of angles α and θ. In order to develop a generalized equation, the possibility of rotating in both directions must be taken into account, as shown in Equations 3.5 and 3.6.

$$\cos(\alpha \pm \theta) = \frac{x_{i+1}}{1} = x_{i+1} \tag{3.5}$$

$$\sin(\alpha \pm \theta) = \frac{y_{i+1}}{1} = y_{i+1} \tag{3.6}$$

Using the additive angle formulas for cosine and sine, Equations 3.5 and 3.6 can be rewritten as Equations 3.7 and 3.8, respectively. These forms also allow for the possibility of rotating the unit vector in either direction.

$$x_{i+1} = \cos(\alpha \pm \theta) = \cos(\alpha)\cos(\theta) \mp \sin(\alpha)\sin(\theta) \tag{3.7}$$

$$y_{i+1} = \sin(\alpha \pm \theta) = \sin(\alpha)\cos(\theta) \pm \cos(\alpha)\sin(\theta) \tag{3.8}$$

Utilizing the identities for x_i and y_i found in Equations 3.3 and 3.4, Equations 3.7 and 3.8 can be simplified as shown in Equations 3.9 and 3.10.

$$x_{i+1} = \cos(\theta)x_i \mp \sin(\theta)y_i \tag{3.9}$$

$$y_{i+1} = \cos(\theta)y_i \pm \sin(\theta)x_i \tag{3.10}$$

Equations 3.9 and 3.10 can be written as a matrix multiplication as shown in Equation 3.11. This is the standard 2-Dimensional rotation equation derived in any computer graphics textbook that discusses translations and rotations [7], [8], [9]. This equation can be used to rotate any vector or group of vectors, with any initial rotation, α, about the z-axis by any desired angle, θ.

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \mp\sin(\theta) \\ \pm\sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{3.11}$$

Equation 3.11 provides an excellent starting point for explaining the development of the CORDIC algorithm. The new location of the head of a rotated unit vector is located at the point (x_{i+1}, y_{i+1}). This location is calculated from the old point (x_i, y_i) by multiplying the x and the y locations by the appropriate values of $\sin(\theta)$ and $\cos(\theta)$. At first glance, this means that in order to calculate the cosine of an angle, the values for the cosine and sine of that angle must already be calculated. If it were possible to store all of these values, a ROM table would be implemented rather than developing an algorithm to perform the calculations.

Fortunately, there is a simple solution to this problem. When a unit vector is rotated, the rotation can be in the positive direction or the negative direction. If the rotation is performed in the positive direction, the unit vector is rotated by the angle θ. If the rotation is performed in the negative direction, the unit vector is rotated by the angle $(\theta - 2\pi)$. Both rotations place the head of the unit vector at the correct point on the unit circle, (x_{i+1}, y_{i+1}).

This means that it does not matter what path around the unit circle a unit vector takes, as long as the head of the unit vector ends up at the correct location. Taking this to the next logical step, an angle can be calculated by performing a series of rotations that place the head of the unit vector at its final location. Figure 3.3 provides a graphical example of performing an effective rotation of angle θ by rotating the unit vector by angles θ_1, θ_2, and θ_3. Even though the three angles shown in this example are positive, the angles can be positive or negative, as long as their summation is equivalent to the effective rotation angle of θ.

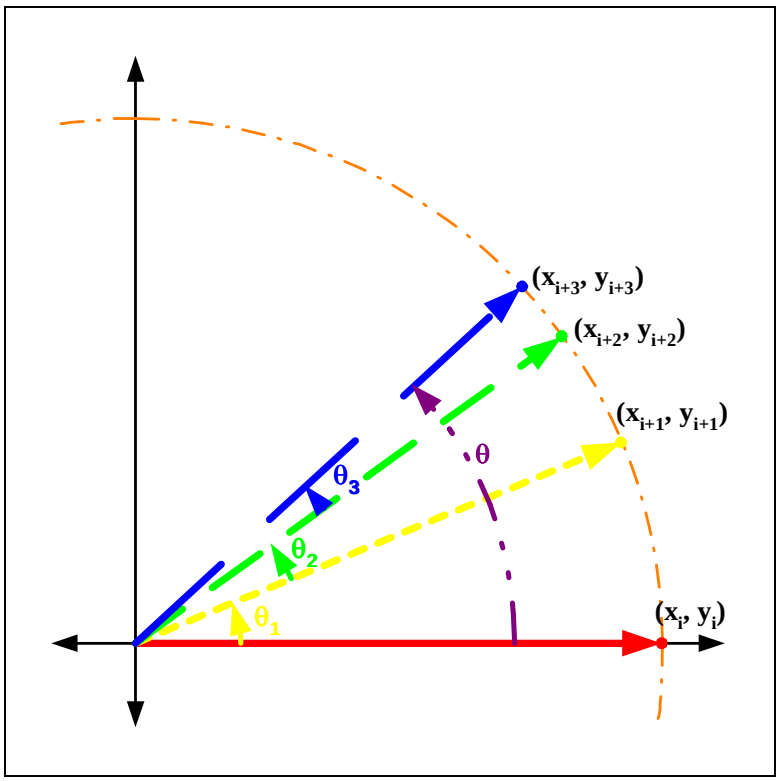


Figure 3.3 – *Multiple Unit Vector Rotations*

Using this property of unit circle rotations, the calculation of the trigonometric functions can be accomplished by performing a series of rotations of the unit vector. Using a finite set of angles, θ_1 through θ_n, the rotation equation can be rewritten as shown in Equation 3.12. The storage required is reduced to a ROM table with n entries, each containing a sine and a cosine value. This is a significant reduction in storage space and makes it possible to implement the algorithm.

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} \cos(\theta_n) & \mp\sin(\theta_n) \\ \pm\sin(\theta_n) & \cos(\theta_n) \end{bmatrix} \cdots \begin{bmatrix} \cos(\theta_1) & \mp\sin(\theta_1) \\ \pm\sin(\theta_1) & \cos(\theta_1) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{3.12}$$

Now that storage has been eliminated as an implementation barrier, other problems with the current rotation equation need to be addressed. One problem is that four multiplications and two additions are required for each sub-rotation. The two additions require a small amount of execution time and area when compared to the four multiplications. Rearranging the Equation 3.12 might be able to reduce the number of operations required for each sub-rotation. Factoring the cosine term out of each multiplication matrix produces Equation 3.13. This reduces the number of multiplications for each sub-rotation to two.

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \cos(\theta_n) \begin{bmatrix} 1 & \mp \tan(\theta_n) \\ \pm \tan(\theta_n) & 1 \end{bmatrix} \cdots \cos(\theta_1) \begin{bmatrix} 1 & \mp \tan(\theta_1) \\ \pm \tan(\theta_1) & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{3.13}$$

3.3 Angle Selection

The next issue that must be addressed is the selection of the subset of angles, θ_1 through θ_n, to use in each sub-rotation. Because the tangent of these angles will be multiplied by the x_i and y_i terms in each sub-rotation, the angles should be selected to minimize that calculation. The only multiplication that is quick and easy to perform in a binary computation is a multiplication by a power of two. Multiplications by a power of two can be accomplished by shifting the binary number the correct number of positions to the left or the right. If we set the tangent of the angle θ equal to 2^x and then take the inverse tangent of both sides we obtain Equation 3.14. So if all of the angles are arctangents of a power of two, the multiplications for each sub-rotation reduce to simple shift operations.

$$\theta = \tan^{-1}\left(2^{x}\right) \tag{3.14}$$

Substituting these angles into Equation 3.13 reduces its complexity. The new equation is shown in Equation 3.14. This substitution reduces the computation complexity and improves the performance of calculating the trigonometric function.

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \cos\left(\tan^{-1}\left(2^{x_n}\right)\right)\begin{bmatrix} 1 & \mp 2^{x_n} \\ \pm 2^{x_n} & 1 \end{bmatrix}\cdots \\ \cos\left(\tan^{-1}\left(2^{x_1}\right)\right)\begin{bmatrix} 1 & \mp 2^{x_1} \\ \pm 2^{x_1} & 1 \end{bmatrix}\begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{3.14}$$

The next issue to tackle is the powers of two to use in the arctangent angle formulas. The angles selected should start out being large and then decrease in order to provide finer degrees of rotation so that accurate rotations can be modeled. The arctangent of increasing powers of two quickly approaches π and does not generate the small angles required to precisely model any given rotation. This implies that decreasing powers of two should be used in the arctangent angle formula. Starting with 2^{0} and decreasing to $2^{-(n-1)}$ provides n angles of decreasing size that can be used to accurately model the rotations. Substituting these values into the previous equations generates Equation 3.15.

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \cos\left(\tan^{-1}\left(2^{-(n-1)}\right)\right)\begin{bmatrix} 1 & \mp 2^{-(n-1)} \\ \pm 2^{-(n-1)} & 1 \end{bmatrix}\cdots \\ \cos\left(\tan^{-1}\left(2^{0}\right)\right)\begin{bmatrix} 1 & \mp 2^{0} \\ \pm 2^{0} & 1 \end{bmatrix}\begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{3.15}$$

3.4 Rotation Direction

Equation 3.15 maintains the ability to rotate the unit vector in either a positive or negative direction. The $\pm$ and the $\mp$ signs in the matrix multiplications represent this functionality. In order to use these equations as an algorithm, these rotations need to be implemented as a variable that will select the rotation direction. The variable σ_i is used to represent the sign of the current angle. If the angle is positive, the rotation should be in the negative direction. If the angle is negative, the rotation should be in the positive direction. Implementing these requirements generates Equation 3.16. The possible values for σ_i are $\{-1,0,1\}$.

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \cos\left(\tan^{-1}\left(2^{-n}\right)\right)\begin{bmatrix} 1 & -\sigma_n 2^{-n} \\ \sigma_n 2^{-n} & 1 \end{bmatrix}\cdots \cos\left(\tan^{-1}\left(2^{-1}\right)\right)\begin{bmatrix} 1 & -\sigma_1 2^{-1} \\ \sigma_1 2^{-1} & 1 \end{bmatrix}\begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{3.16}$$

3.5 Scale Factor

Substituting the variable K_i into the equation for the term $\cos\left(\tan^{-1}\left(2^{-i}\right)\right)$ simplifies the equation. After all of the substitutions are performed, the K_i terms are collected together as shown in Equation 3.17. Because the angles that are used are predetermined, each of the K_i terms is a constant value and can be considered a scale factor. Rotation angles with σ_i values of ± 1 have a scale factor of K_i. Rotation angles with σ_i values of 0 have a scale factor of 1.

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = K_n \cdots K_1 \begin{bmatrix} 1 & -\sigma_n 2^{-n} \\ \sigma_n 2^{-n} & 1 \end{bmatrix}\cdots\begin{bmatrix} 1 & -\sigma_1 2^{-1} \\ \sigma_1 2^{-1} & 1 \end{bmatrix}\begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{3.17}$$

Because zero is a possible value for σ_i, the final value of the scale factor depends upon which rotations are performed. If the possible values for σ_i are limited to the set $\{-1,1\}$, a rotation will be performed for each sub-rotation angle. Even though this requires additional rotations to be performed, it creates a constant scale factor K, which is the product of all of the scale factors as shown in Equation 3.18. The additional rotations are performed with shift and add operations that can be completed quickly. A variable scale factor requires a series of multiplications to compute. Forcing a rotation to be performed for each sub-rotation angle reduces the overall computation time for the algorithm. In addition, because the scale factor is constant, the x and y variables can be initialized with values pre-scaled by $1/K$. When the rotations are completed, the range of the outputs is correct and no post calculation normalizations are required.

$$K = K_0 K_1 \cdots K_{n-2} K_{n-1} \tag{3.18}$$

3.6 Angle Criteria

Because a rotation is taken for each sub-rotation angle, it is possible for the residual angle to increase. If the previous residual angle is close to zero, the next sub-angle rotation can rotate the unit vector further from zero if the magnitude of the rotation angle is greater than twice the residual angle. To ensure that the unit vector can still converge upon zero after one of these rotations, the summation of the remaining angles must be large enough converge upon zero. This condition is expressed in Equation 3.19 and must be satisfied to ensure convergence.

$$\forall i, \alpha_i \leq \sum_{k=i+1}^{\infty} \alpha_k \tag{3.19}$$

3.7 Iteration Equations

The CORDIC algorithm uses the system of equations shown in Equations 3.20, 3.21, and 3.22 to iteratively calculate the vector or angle in question. As the equations demonstrate, the multiplication is by a power of two and can be accomplished through a simple shift operation rather than a real multiplication. The only arithmetic operation required to calculate each new value of these equations is a simple addition or subtraction.

$$X_{i+1} = X_i - \sigma_i 2^{-i} Y_i \tag{3.20}$$

$$Y_{i+1} = Y_i + \sigma_i 2^{-i} X_i \tag{3.21}$$

$$Z_{i+1} = Z_i - \sigma_i \tan^{-1}\left(2^{-i}\right) \tag{3.22}$$

The additions cannot be performed until the value of σ_i has been determined for each residual angle. Table 3.1 shows how σ_i is selected during each iteration of the equations. If the angle is positive, the unit vector is rotated in a negative direction, the X variable is reduced by a fraction of the Y variable, and Y variable is incremented by a fraction of the X variable. If the angle is negative, the opposite operation is performed for each variable. Because the sign of the next residual angle can not be determined until the current operation has been performed, the CORDIC iteration equations are inherently serial in nature.

Table 3.1 – *Classic CORDIC Sign Bit Selection*

ANGLE	**SIGN**
$Z_i \geq 0$	$\sigma_i = 1$
$Z_i < 0$	$\sigma_i = -1$

3.8 Pseudo Rotations

Examining a single iteration of the CORDIC equations shows the importance of Volder's modification to the original rotation equations. A standard rotation changes the rotation angle of the unit vector but does not affect the length of the unit vector. Volder's CORDIC rotation of the vector changes its rotation angle the same amount as a standard rotation but lengthens the vector by a factor of K_i in each iteration. These rotations are known as pseudo-rotations because they do not maintain the length of the vector. Because a rotation is taken in each iteration of the equations, the change in the length of the vector is a constant and can be corrected using pre or post normalization. Examples of a rotation and a pseudo-rotation are shown in Figure 3.4.

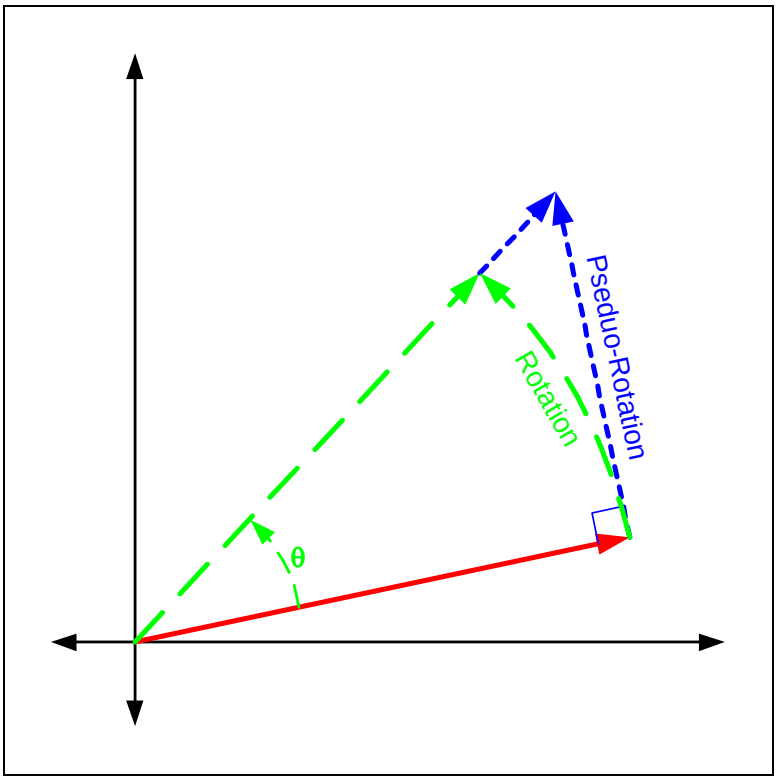


Figure 3.4 – *Classic CORDIC Pseudo-Rotations*

Chapter 4
Previous Work

Many variations of the CORDIC algorithm have been developed to improve the algorithm's performance. Variations that allow the sign of the angle, σ_i, to take on the value of zero at the cost of a variable scale factor [36] [37], K, or correct the scale factor in parallel with the selection have been developed [38]. The CORDIC iteration equations have been implemented after applying a Householder transform in order to improve performance [39] [40]. Multiplier recoding techniques have been applied to the iteration equations to improve performance and reduce latency [41] [42]. Hardware reduction has been achieved by carefully pairing rotation iterations to reduce the number of shifters required [43]. Control theory has been applied to the equations to eliminate the over damped response of the Classic CORDIC algorithm [44].

Due to the number and types of variations of CORDIC algorithms available, the subset of the algorithms that will be examined, implemented, evaluated, and compared in this dissertation needs to be selected carefully. Because normalization and the method by which it is implemented can have a large impact on the performance of a CORDIC algorithm, only algorithms with a constant scale factor, K, will be considered. In addition, CORDIC algorithms that maintain a constant scale factor through the use of additional correction rotations will not be considered in order to keep comparisons equivalent. The CORDIC algorithms that were selected and will be discussed in the following sections are the Unified, Step Branching, Double Step Branching, and Hybrid CORDIC algorithms.

4.1 Unified CORDIC

Not only can the Classic CORDIC algorithm calculate trigonometric functions, but there are also variations of the algorithm that can compute hyperbolic functions, exponentials, logarithms, and multiplication and division. The Unified CORDIC algorithm, developed by J.S. Walther in 1971 [45], merges all of these functions into a single algorithm. This algorithm consists of a set of iteration equations that can calculate trigonometric and hyperbolic functions, exponentials, logarithms, multiplications, and divisions using the same hardware and simply setting a single bit to choose the mode of operation.

4.1.1 Iteration Equations

The Unified CORDIC algorithm iteration equations are shown in Equations 4.1, 4.2, and 4.3. There are minimal differences between the Classic CORDIC and the Unified CORDIC iteration equations. The first difference is the insertion of the μ parameter into the X_{i+1} equation. The μ parameter determines whether the hardware will perform trigonometric, hyperbolic, or linear functions. The allowable values of μ and their corresponding operational modes are shown in Table 4.1.

$$X_{i+1} = X_i - \mu\sigma_i 2^{-i} Y_i \tag{4.1}$$

$$Y_{i+1} = Y_i + \sigma_i 2^{-i} X_i \tag{4.2}$$

$$Z_{i+1} = Z_i - \sigma_i e(i) \tag{4.3}$$

The second difference is found in the Z_{i+1} equation. Instead of rotating the vector by $\arctan(2^{-i})$, the generalized function $e(i)$ is used. The function $e(i)$ is used because the rotation function is different for each of the Unified CORDIC algorithm's modes of operation. A list of the operational modes and its rotation

function, $e(i)$, is shown in Table 4.2. The selection of the sign bit, σ_i, remains the same as the Classic CORDIC algorithm and can be found in Table 3.1.

Table 4.1 – *Unified CORDIC Operational Modes*

μ	ROTATION TYPE
1	Circular Rotations (sin, cos, etc.)
0	Linear Rotations (multiplication, division)
-1	Hyperbolic Rotations (sinh, cosh, etc.)

Table 4.2 – *Unified CORDIC Rotation Functions*

ROTATION TYPE	*e(i)*
Circular Rotations	$\tan^{-1}\left(2^{-i}\right)$
Linear Rotations	2^{-i}
Hyperbolic Rotations	$\tanh^{-1}\left(2^{-i}\right)$

4.2 Step-Branching CORDIC

The Step-Branching CORDIC algorithm, developed by Duprat and Muller in 1993, improves the performance of the algorithm by using the Binary Signed Digit (BSD) number system for representing all of the equation variables [46]. The BSD number system, first studied by Avizienis in 1961, is a redundant number system that uses the digit set $\{-1,0,1\}$ [47]. The BSD number system has the beneficial property of very short carry chains. The sum of the bit in position i only depends on results from bit position $i-1$ and bit position $i-2$. By removing the carry propagation

delay inherent in a Ripple Carry Adder, the time to perform an addition is greatly reduced in the Step-Branching CORDIC algorithm.

4.2.1 Iteration Equations

The iteration equations used by the Step-Branching CORDIC algorithm are the same as those used by the Unified CORDIC algorithm and can be seen in Equations 4.1, 4.2, and 4.3. The only difference between the two algorithms is that the additions in the Step-Branching CORIDC algorithm are performed using redundant binary adders and the results are stored as a redundant numbers.

Although the use of the BSD number system improves the performance of the additions, it introduces a different problem. At each iteration of the algorithm, the sign of the current angle, Z_i, must be determined before the next iteration can begin. The sign of any BSD number is the same as the sign of its most significant non-zero digit. To determine the sign of Z_i, the most significant non-zero digit must be found, and then its sign must be determined. If the angle Z_i is close to zero, almost every single digit would need to be examined. This delay could eliminate much of the gain obtained by using the BSD number system.

To avoid examining every digit in Z_i, a subset of digits is examined to determine the sign of the current angle. Examining the subset can be performed in a constant time in order to preserve the benefits of using a redundant number system. The drawback of examining a subset of digits is that the selection of $\sigma_i = 0$ must be allowed. This means that the scale factor, K, is no longer a constant.

In 1990, Ercegovac and Lang implemented an On-Line CORDIC algorithm that calculated the scale factor in parallel with the rotations [48]. The results are normalized once all of the rotations are complete. Even though this algorithm achieves the potential gains of carry-free addition, it does so at the expense of additional complexity and computation. A multiplier for calculating the scale factor

and a divider for performing the normalization must be designed and implemented. Even though the multiplications may be computed in parallel with the rotations, the division to normalize the numbers will occur after the rotations have completed. Due to the complexity of the division, a large portion of the speedup obtained through the use of the redundant number system may be lost.

Takagi, Asada, and Yajima proposed a double rotation method in 1987 [49] and a correcting rotation method in 1991 [50] that use redundant number systems to speed the computations of each rotation. Both of these methods preserve a constant scale factor. Although these methods eliminate the additional complexity required in calculating a variable scale factor, they still require additional double rotations or correcting rotations. Even though these operations do not require as much time as a division, these additional rotations prevent these methods from achieving the full potential speedup offered by the carry-free addition.

The Step-Branching CORDIC algorithm takes a different approach to solve this problem. If the examination of the subset of digits is sufficient to show that Z_i is positive, then σ_i is selected to be 1. If the examination of the subset of digits is sufficient to show that Z_i is negative, then σ_i is selected to be -1. If the examination of these bits is inconclusive, then two computations are performed in parallel in two separate hardware implementations. One computation, with $\sigma_i = 1$, is performed in the "positive" hardware branch, and the other computation, with $\sigma_i = -1$, is performed in the "negative" hardware branch. This covers both possibilities for the value of the residual angle Z_i.

When this occurs, the algorithm is considered to be in branch mode. Each possible set of calculations continues in parallel until another branch is reached. Once this next branch is reached, the correct set of calculations can be determined from the signs of the angles. The appropriate branch according to σ_i^+ and σ_i^- is given in Table 4.3. The correct branch is reloaded into each hardware branch and the

calculations split into parallel computations again. This process repeats until all of the required iterations have been performed.

Table 4.3 – *Step Branching Calculation Selection*

σ_i^+	σ_i^-	**Correct Branch**
0	*	Positive Branch
1	*	Positive Branch
-1	0	Negative Branch
-1	1	Negative Branch

Because the two calculations are performed in parallel, the full benefit of carry-free addition from the redundant number system is realized. In addition, by only allowing the sign bit, σ_i, to take on the values of ± 1, the Step Branching CORDIC algorithm produces a constant scale factor, so no post-iteration normalization is required. The cost of this speed-up is the addition of three extra addition/subtraction units for the second set of branch hardware. Compared to the addition of a multiplier and divider, the area required by the additional adders is a small price to pay for the speed-up the algorithm achieves.

4.3 Double Step-Branching CORDIC

The Double Step Branching CORDIC algorithm, described by Dhananjay Phatak in 1998 [51], takes the advances of the Branching algorithm a step further. The Step Branching CORDIC algorithm computes the same equations in both functional units until it reaches a branch. These computations are wasted and do not improve the performance of the algorithm until a branch is reached. The Double

Step Branching CORDIC algorithm uses the second functional unit to perform useful calculations during every iteration of the equations.

4.3.1 Iteration Equations

The Double Step Branching algorithm uses the same equations as the Unified CORDIC algorithm. Those equations are shown in Equations 4.1, 4.2 and 4.3. The difference between the algorithms is that the Double Step Branching algorithm performs two iterations of the equations during each step. If the first few bits of the current angle Z_i can positively be determined to be positive, then Equations 4.4 and 4.5 are used to calculate the two possible next angles. Angle Z_{i+1}^{α} represents the calculation from the first arithmetic unit while Z_{i+1}^{β} represents the calculation from the second arithmetic unit. In this way, the duplicated hardware of the Branching algorithm is put to a constructive use. If the first few bits of Z_{i+1}^{α} show that the number is positive or close to zero after this calculation, then its value is the correct answer. This value is loaded into both arithmetic units for the next calculation. If the first few bits of Z_{i+1}^{α} show that the number is negative, then the first few bits of Z_{i+1}^{β} must be examined. If these bits show that Z_{i+1}^{β} is negative or close to zero, then Z_{i+1}^{β} is the correct answer. This value is then loaded into both arithmetic units for the next calculation. If the first few bits of Z_{i+1}^{β} show that it is a positive number, then the Double Step Branching algorithm enters into the branching mode just like the Branching algorithm.

$$Z_{i+1}^{\alpha} = Z_i - \tan^{-1}\left(2^{-(2i)}\right) - \tan^{-1}\left(2^{-(2i+1)}\right) \tag{4.4}$$

$$Z_{i+1}^{\beta} = Z_i - \tan^{-1}\left(2^{-(2i)}\right) + \tan^{-1}\left(2^{-(2i+1)}\right) \tag{4.5}$$

If the first few bits of the current angle Z_i can positively be determined to be negative, then Equations 4.6 and 4.7 are used to calculate the two possible next angles. If the first few bits of Z_{i+1}^{α} show that the number is negative or close to zero after this calculation, then its value is the correct answer. This value is loaded into both arithmetic units for the next calculation. If the first few bits of Z_{i+1}^{α} show that the number is positive, then the first few bits of Z_{i+1}^{β} must be examined. If these bits show that Z_{i+1}^{β} is positive or close to zero, then Z_{i+1}^{β} is the correct answer. This value is then loaded into both arithmetic units for the next calculation. If the first few bits of Z_{i+1}^{β} show that it is a negative number, then the Double Step Branching algorithm enters into the branching mode just like the Branching algorithm.

$$Z_{i+1}^{\alpha} = Z_i + \tan^{-1}\left(2^{-(2i)}\right) + \tan^{-1}\left(2^{-(2i+1)}\right) \tag{4.6}$$

$$Z_{i+1}^{\beta} = Z_i + \tan^{-1}\left(2^{-(2i)}\right) - \tan^{-1}\left(2^{-(2i+1)}\right) \tag{4.7}$$

If the first few bits of the current angle Z_i can determine that the angle is close to zero, then Equations 4.8 and 4.9 are used to calculate the two possible next angles. If either Z_{i+1}^{α} or Z_{i+1}^{β} are equal to zero, then that branch has the correct answer. This value is loaded into both arithmetic units for the next calculation. Otherwise, the Double Step Branching algorithm enters into the branching mode just like the Step Branching algorithm.

$$Z_{i+1}^{\alpha} = Z_i + \tan^{-1}\left(2^{-(2i)}\right) - \tan^{-1}\left(2^{-(2i+1)}\right) \tag{4.8}$$

$$Z_{i+1}^{\beta} = Z_i - \tan^{-1}\left(2^{-(2i)}\right) + \tan^{-1}\left(2^{-(2i+1)}\right) \tag{4.9}$$

The Double Step Branching CORDIC algorithm has several benefits. The first is that it performs two angle rotations in each iteration through the equations, reducing the number of iterations that must be performed. The second is that it always selects the sign of the angle as ± 1, retaining a constant scale factor that does not need to be calculated and then used for post-iteration normalization. The drawbacks are the increased complexity of the control logic required to decide which hardware branch contains the correct answer and the increased complexity of the computations.

4.4 Hybrid CORDIC

Where the Step Branching CORDIC algorithm and Double Step Branching CORDIC algorithm obtain performance improvements by utilizing a redundant number system, the Hybrid CORDIC algorithm, presented in 1997 [52], uses a different Arc Tangent Radix (ATR). The use of this different ATR allows a significant portion of the iterations to be performed in parallel. This parallelism is possible, because the new radix allows the sign bit, σ_i, for the current iteration and all future iterations to be calculated directly from the digits in the current angle, Z_i.

4.4.1 Iteration Equations

The Hybrid CORDIC algorithm is based upon the same iteration equations that have been used by the other algorithms, shown in Equations 4.1, 4.2, and 4.3. The difference between the Hybrid CORDIC algorithm and these other algorithms is the choice of the Arc Tangent Radix, α_i. Rather than selecting $\alpha_i = \tan^{-1}\left(2^{-i}\right)$ for all angles, only the first rotations that require a higher accuracy use these angles. The later rotations use the constant angle set $\alpha_i = 2^{-i}$. As *i* increases, the difference between $\tan^{-1}\left(2^{-i}\right)$ and 2^{-i} decreases. The choice of where to separate the use of

$\alpha_i = \tan^{-1}\left(2^{-i}\right)$ and $\alpha_i = 2^{-i}$ was made by ensuring that the accuracy of the two calculations remained the same. Using the Taylor series expansion of the error term, it was determined that performing approximately a third of the iterations with $\alpha_i = \tan^{-1}\left(2^{-i}\right)$ and then using $\alpha_i = 2^{-i}$ for the rest of the iterations would provide a result with the same error. The exact number of iterations required is shown in Equation 4.7.

$$n = \left\lceil \frac{N - \log_2 3}{3} \right\rceil \tag{4.7}$$

Approximately the first third of the iterations are performed on traditional CORDIC hardware in a serial manner. Once these calculations have been performed, the last two thirds can be calculated in parallel. By looking at the residue rotation angle Z_i, each bit with a value of one represents a positive rotation while each bit with a value of zero represents a negative rotation. These rotations can be combined into a single calculation with the appropriate hardware. In addition, by requiring a rotation for all ones and zeros, the magnitude will change the same for each angle, resulting in a constant expansion factor K.

Chapter 5
Table Look-up CORDIC Algorithm

The original impetus of this research was to determine if were possible to improve the performance of the Hybrid CORDIC algorithm. If there were a way to calculate the first third of the CORDIC iterations in a single step, it could be used in conjunction with the Hybrid CORDIC algorithm to perform the calculations in a two-step process. This chapter discusses the original development of the Table Look-up CORDIC (TLC) algorithm for use with the Hybrid CORDIC algorithm, the development of a mathematical proof showing its correctness, the extended capabilities that were identified, and the design trade offs that can be made between performance and area.

5.1 Effective Rotations

Look-up tables have been used extensively to implement both simple and complex functions when a bit accurate answer is needed for the algorithm [53] [54]. It is easy to calculate the desired function to the appropriate number of bits, place that answer in a ROM look-up table, and then retrieve the answer when it is needed. In addition, ROM look-up table implementations are very fast when compared with the amount of time it may take the hardware to calculate the same answer.

The problem with ROM look-up tables is the rate at which the number of entries grows when the number of bits increases. For every bit that is added, the number of entries doubles. In addition to the exponential growth of the number of table entries, the amount of storage for each entry increases by the number of additional bits that are added. Table 5.1 shows the number of table entries that are required and the amount of storage needed to implement a ROM table with the given

number of bits. As the table clearly shows, the storage space required for a ROM can quickly go from reasonable, in the case of a twelve bit number, to very difficult, in the case of a twenty bit number, to impossible, in the case of a twenty-four bit number. If the range of floating-point numbers is considered, the number of entries and amount of storage required is radically increased.

Table 5.1 – *ROM Storage Requirements*

Bits	**Table Entries**	**MBytes**
8	256	0.0002
12	4,096	0.0059
16	65,536	0.125
20	1,048,576	2.5
24	16,777,216	48
28	268,435,456	896
32	4,294,967,296	16,385

ROM look-up tables have not only been used to provide the final answer, but they have also been used to provide a starting approximation for the calculation. By providing a good approximation, a quadratic convergence algorithm can start its doubling process on multiple bits, rather than a single bit [55]. Because the accuracy of the algorithm starts doubling on multiple bits, the number of iterations is reduced; therefore the execution time of the calculation is reduced. An additional benefit is that the number provided by the ROM look-up table can be a rough approximation to the answer. This appreciably reduces the number of bits required in each entry and reduces the number of entries in the ROM. This reduction results in a smaller look-

up table that is practical to implement in silicon while still providing a significant performance enhancement to the algorithm.

ROM look-up tables have also been used to implement functions by partitioning the calculations in a way that they can be stored in separate tables and then combined using simple arithmetic operations [56] [57]. By partitioning the calculation in this manner, the storage required in the ROM table is greatly reduced, saving valuable silicon area.

5.1.1 Table Look-up Technique

Unfortunately, the CORDIC algorithms are not quadratic convergence algorithms. Therefore, an approximation to the answer is not good enough to allow the algorithm to converge upon the correct answer. Any offset in the approximation will result in an offset in the final answer. Because the amount of offset in the answer is not a linear combination of the initial angle, there is not an easy method for correcting the final answer. The use of a ROM look-up table with the Hybrid CORDIC algorithm requires that all answers be bit accurate to the number of bits in the final answer.

The Hybrid CORDIC algorithm provides an impetus to examine a look-up table method for determining part of the answer. Because the Hybrid CORDIC algorithm only requires that the first third of the iterations be calculated by the serial iteration method, it is possible to use a ROM table for providing the results of these initial unit vector rotations.

The Hybrid CORDIC algorithm requires that the variables X_n, Y_n, and Z_n be bit accurate when input to the Hybrid CORDIC processor. Implementing a traditional CORDIC processor as a direct mapped ROM look-up table requires an entry for every possible angle representation. Using a n-bit number to represent the angle, 2^n table entries are required for the direct mapped ROM. Each of the 2^n

table entries contains two n-bit rotation coefficients. For large values of n, this is impractical. Even though it is impractical to implement a direct mapped ROM look-up table, it is possible to use a ROM look-up table to speed up the calculation of the first third of the Hybrid CORDIC iterations. Looking at the first three iterations of the CORDIC algorithm provides the necessary understanding of how this implementation works.

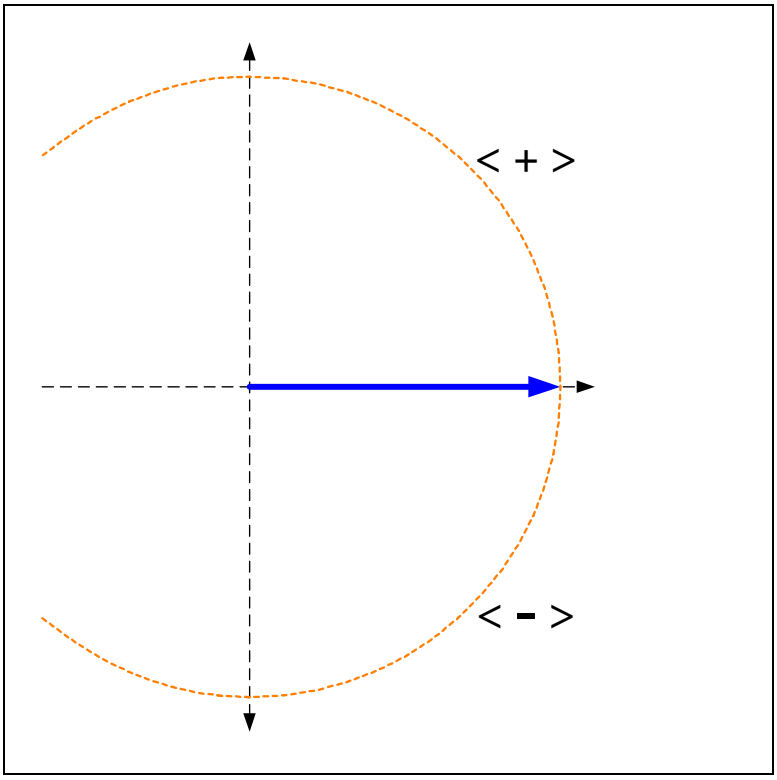


Figure 5.1 – *The First Iteration Angle Partitions*

In the first CORDIC iteration, unit vectors with initial angles greater than or equal to zero will be rotated by a negative $\tan^{-1}\left(2^{-i}\right)$ or -45°, while unit vectors with initial angles less than zero will be rotated by a positive $\tan^{-1}\left(2^{-i}\right)$ or 45°. This selection can be seen graphically in Figure 5.1. The $\langle + \rangle$ sign in Figure 5.1

represents the unit vectors with positive angles, Z_o, that will be rotated in the negative direction. The $\langle - \rangle$ sign in Figure 5.1 represents the unit vectors with negative angles, Z_o, that will be rotated in the positive direction.

The two angle partitions generated by this subdivision are [90°, 0°] and (0°, -90°]. All unit vectors in the [90°, 0°] partition are rotated by -45° while all unit vectors in the (0°, -90°] partition are rotated by 45°. The angle partitions are created by taking the original angle partition, [90°, -90°], and dividing it along the critical angles. A critical angles is an initial angle, Z_0, that can have a residual angle, Z_i, equal to zero. For the first iteration, the initial angle is the only residual angle. Therefore, the only critical angle for the first rotation is 0°.

In the second CORDIC iteration, unit vectors with residual angles, Z_1, that are greater than or equal to zero will be rotated by a negative $\tan^{-1}\left(2^{-i}\right)$ or -26.565°, while unit vectors with residual angles less than zero will be rotated by a positive $\tan^{-1}\left(2^{-i}\right)$ or 26.565°. If the initial angle, Z_o, is greater than or equal 45°, then both of the unit vector's rotations will be in a negative direction. This can be seen in Figure 5.2 and is represented by the $\langle +,+ \rangle$. If the unit vector's initial angle was greater than or equal to 0° but less than 45°, then its first rotation is in the negative direction and its second rotation is in the positive direction. The $\langle +,- \rangle$ in Figure 5.2 represents this rotation. If the unit vector's initial angle was less than 0° but greater than or equal to -45°, then its first rotation will be in the positive direction and its second rotation will be in the negative direction. The $\langle -,+ \rangle$ in Figure 5.2 represents this rotation. Finally, if the unit vector's initial angle was less than -45°, then both of the unit vector's rotations are in the positive direction. The $\langle -,- \rangle$ in Figure 5.2 represents this rotation.

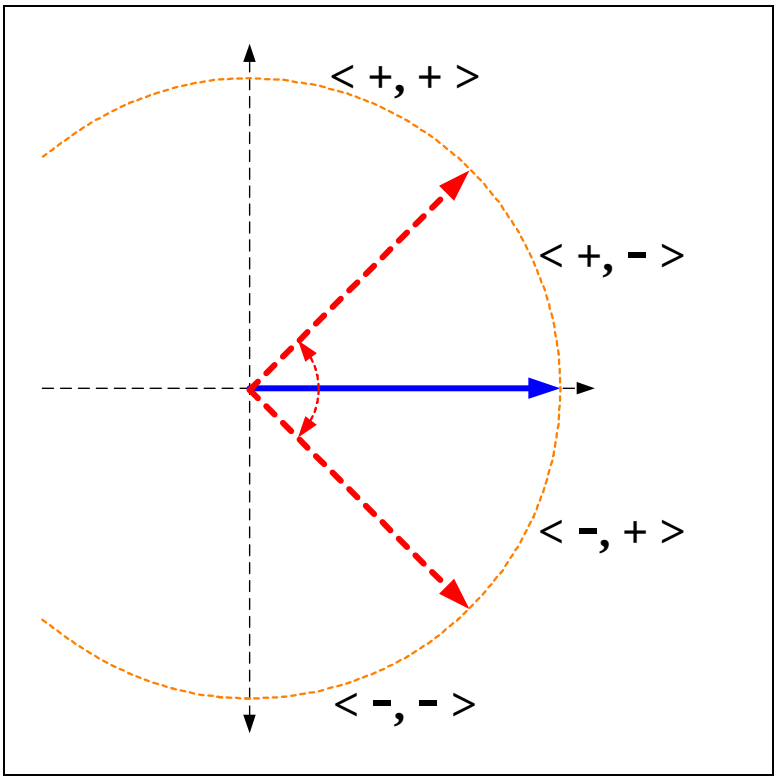


Figure 5.2 – *The Second Iteration Angle Partitions*

The four angle partitions generated by this subdivision are [90°, 45°], (45°, 0°], (0°, -45°], and (-45°, -90°]. The critical angles for the second iteration are 0° from the first iteration and ±45° from the second iteration. These are the three angles that can generate a residual angle of zero in the first two iterations.

The results of the third iteration are shown graphically in Figure 5.3. If the unit vector's residual angle, Z_2, is greater than or equal to zero, the unit vector is rotated in the negative direction. If the unit vector's residual angle, Z_2, is less than zero, the unit vector is rotated in the positive direction. This rotation generates four additional angle partitions by splitting each of the previous partitions. These areas are represented by the $\langle +,+,+ \rangle$, $\langle +,+,- \rangle$, $\langle +,-,+ \rangle$, $\langle +,-,- \rangle$, $\langle -,+,+ \rangle$, $\langle -,+,- \rangle$, $\langle -,-,+ \rangle$, and $\langle -,-,- \rangle$ shown in Figure 5.3.

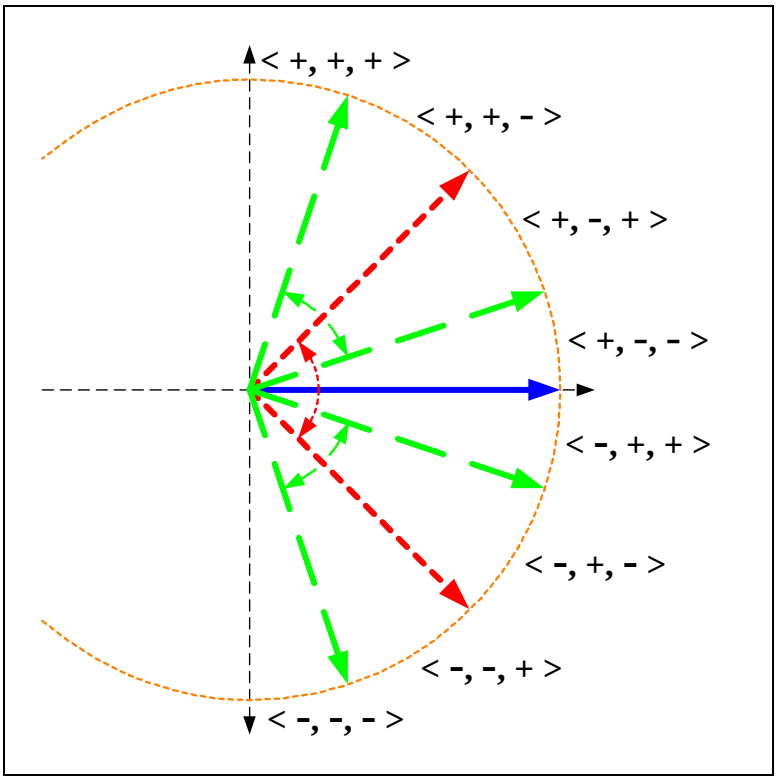


Figure 5.3 – *The Third Iteration Angle Partitions*

The eight angle partitions generated by this subdivision are [90°, 71.565°], (71.565°, 45°], (45°, 18.435°], (18.435°, 0°], (0°, -18.435°], (-18.435°, -45°], (-45°, -71.565°], and (-71.565°, -90°]. The critical angles for the third iteration are 0° from the first iteration, ±45° from the second iteration, and ±71,565° and ±18.435° from the third iteration. These are the seven angles that can generate a residual angle of zero in the first three iterations.

These three iterations exhibit characteristics that are inherent in this partitioning algorithm. First, the number of partitions doubles with every iteration through the CORDIC equations. Therefore, at any given iteration step, i, there are 2^i partitions. Second, each of these partitions can be represented by an i-bit code of plusses and minuses separated by commas. The location of each plus or minus

within the i-bit code represents the sign of the unit vector's residual angle at that iteration step. The rotation that was performed during that iteration is in the opposite direction of the sign of the residual angle. Third, the unit vectors that are located within each angle partition are rotated in the same direction in each iteration. This means that the effective rotation for a given angle partition can be calculated and placed in a look-up table.

5.1.2 Critical Angles

Calculating the critical angles is a vital step in partitioning the range of the CORDIC algorithm. Iteration i will have $2^i - 1$ critical angles. These critical angles, along with the initial range limits of ±90° are used to form the 2^i angle partitions. The critical angles can be calculated using Equation 5.1. Calculating all possibilities, where σ_i can take on any value in the set $\{-1,0,1\}$ will produce all of the critical angles for the given iteration.

$$\sum_{i=0}^{n-1} \sigma_i \tan^{-1}\left(2^{-i}\right) \tag{5.1}$$

5.1.3 Iteration Calculations

Having obtained a graphical understanding of the angle partitioning, it is important to examine the iteration equations to see how the calculations are performed. Equations 5.2, 5.3, 5.4, and 5.5 show the *X* variable for the first four iterations of the CORDIC algorithm. Equations 5.6, 5.7, 5.8, and 5.9 show the *Y* variable for the first four iterations of the CORDIC algorithm.

$$X_{i+1} = X_i - \sigma_i 2^{-i} Y_i \tag{5.2}$$

$$X_{i+2} = X_{i+1} - \sigma_{i+1} 2^{-(i+1)} Y_{i+1} \tag{5.3}$$

$$X_{i+3} = X_{i+2} - \sigma_{i+2} 2^{-(i+2)} Y_{i+2} \tag{5.4}$$

$$X_{i+4} = X_{i+3} - \sigma_{i+3} 2^{-(i+3)} Y_{i+3} \tag{5.5}$$

$$Y_{i+1} = Y_i + \sigma_i 2^{-i} X_i \tag{5.6}$$

$$Y_{i+2} = Y_{i+1} + \sigma_{i+1} 2^{-(i+1)} X_{i+1} \tag{5.7}$$

$$Y_{i+3} = Y_{i+2} + \sigma_{i+2} 2^{-(i+2)} X_{i+2} \tag{5.8}$$

$$Y_{i+4} = Y_{i+3} + \sigma_{i+3} 2^{-(i+3)} X_{i+3} \tag{5.9}$$

Starting with the equations for X_{i+4} and Y_{i+4}, shown in Equations 5.4 and 5.8, the values of X_{i+3} and Y_{i+3} from Equations 5.3 and 5.7 can be substituted into them to obtain the values of X_{i+4} and Y_{i+4} all in terms of X_{i+2} and Y_{i+2}. This substitution process can be performed until the values of X_{i+4} and Y_{i+4} are expressed completely in terms of X_i and Y_i. Once these substitutions have been performed, all of the terms for X_i and Y_i can be grouped together as shown in Equations 5.10 and 5.11.

$$\begin{aligned} X_{i+4} = \; & X_i[1 - \sigma_{i+1}\sigma_i 2^{-(i+1)-i} - \sigma_{i+2}\sigma_i 2^{-(i+2)-i} - \sigma_{i+2}\sigma_{i+1} 2^{-(i+2)-(i+1)} - \sigma_{i+3}\sigma_i 2^{-(i+3)-i} \\ & - \sigma_{i+3}\sigma_{i+1} 2^{-(i+3)-(i+1)} - \sigma_{i+3}\sigma_{i+2} 2^{-(i+3)-(i+2)} + \sigma_{i+3}\sigma_{i+2}\sigma_{i+1}\sigma_i 2^{-(i+3)-(i+2)-(i+1)-i}] \\ & + Y_i[-\sigma_i 2^{-i} - \sigma_{i+1} 2^{-(i+1)} - \sigma_{i+2} 2^{-(i+2)} + \sigma_{i+2}\sigma_{i+1}\sigma_i 2^{-(i+2)-(i+1)-i} - \sigma_{i+3} 2^{-(i+3)} \\ & + \sigma_{i+3}\sigma_{i+1}\sigma_i 2^{-(i+3)-(i+1)-i} + \sigma_{i+3}\sigma_{i+2}\sigma_{i+1} 2^{-(i+3)-(i+2)-i} + \sigma_{i+3}\sigma_{i+2}\sigma_{i+1} 2^{-(i+3)-(i+2)-(i+1)}] \end{aligned} \tag{5.10}$$

$$
\begin{aligned}
Y_{i+4} = {} & X_i[\sigma_i 2^{-i} + \sigma_{i+1}2^{-(i+1)} + \sigma_{i+2}2^{-(i+2)} - \sigma_{i+2}\sigma_{i+1}\sigma_i 2^{-(i+2)-(i+1)-i} + \sigma_{i+3}2^{-(i+3)} \\
& - \sigma_{i+3}\sigma_{i+1}\sigma_i 2^{-(i+3)-(i+1)-i} - \sigma_{i+3}\sigma_{i+2}\sigma_i 2^{-(i+3)-(i+2)-i} - \sigma_{i+3}\sigma_{i+2}\sigma_{i+1}2^{-(i+3)-(i+2)-(i+1)}] \\
& + Y_i[1 - \sigma_{i+1}\sigma_i 2^{-(i+1)-i} - \sigma_{i+2}\sigma_i 2^{-(i+2)-i} - \sigma_{i+2}\sigma_{i+1}2^{-(i+2)-(i+1)} - \sigma_{i+3}\sigma_i 2^{-(i+3)-i} \\
& - \sigma_{i+3}\sigma_{i+1}2^{-(i+3)-(i+1)} - \sigma_{i+3}\sigma_{i+2}2^{-(i+3)-(i+2)} + \sigma_{i+3}\sigma_{i+2}\sigma_{i+1}\sigma_i 2^{-(i+3)-(i+2)-(i+1)-i}]
\end{aligned} \tag{5.11}
$$

Equations 5.9 and 5.10 represent four consecutive iterations of the CORDIC algorithm starting with iteration i. Any value can be substituted into i to obtain the correct formula for the rotations performed from the i to the $i+4$ iterations. To simplify the equations for this example, 0 is substituted into the equations for i. This substitution results in Equations 5.12 and 5.13.

$$
\begin{aligned}
X_4 = {} & X_0\left[1 - \frac{\sigma_1\sigma_0}{2} - \frac{\sigma_2\sigma_0}{4} - \frac{\sigma_2\sigma_1}{8} - \frac{\sigma_3\sigma_0}{8} - \frac{\sigma_3\sigma_1}{16} - \frac{\sigma_3\sigma_2}{32} + \frac{\sigma_3\sigma_2\sigma_1\sigma_0}{64}\right] + \\
& Y_0\left[-\sigma_0 - \frac{\sigma_1}{2} - \frac{\sigma_2}{4} + \frac{\sigma_2\sigma_1\sigma_0}{8} - \frac{\sigma_3}{8} + \frac{\sigma_3\sigma_1\sigma_0}{16} + \frac{\sigma_3\sigma_2\sigma_1}{32} + \frac{\sigma_3\sigma_2\sigma_1}{64}\right]
\end{aligned} \tag{5.12}
$$

$$
\begin{aligned}
Y_4 = {} & X_0\left[\sigma_0 + \frac{\sigma_1}{2} + \frac{\sigma_2}{4} - \frac{\sigma_2\sigma_1\sigma_0}{8} + \frac{\sigma_3}{8} - \frac{\sigma_3\sigma_1\sigma_0}{16} - \frac{\sigma_3\sigma_2\sigma_0}{32} - \frac{\sigma_3\sigma_2\sigma_1}{64}\right] + \\
& Y_0\left[1 - \frac{\sigma_1\sigma_0}{2} - \frac{\sigma_2\sigma_0}{4} - \frac{\sigma_2\sigma_1}{8} - \frac{\sigma_3\sigma_0}{8} - \frac{\sigma_3\sigma_1}{16} - \frac{\sigma_3\sigma_2}{32} + \frac{\sigma_3\sigma_2\sigma_1\sigma_0}{64}\right]
\end{aligned} \tag{5.13}
$$

Each of these equations has sixteen terms, which equals the number of angle partitions that four iterations of the CORDIC algorithm generate. No term with the same denominator has the same combination of residual angle sign bits in the numerator. This shows that each of the terms produces an independent result. Substituting the sixteen possible combinations of sign bits into Equations 5.11 and 5.12 provides the coefficients that can be multiplied by X_0 and Y_0 to obtain the precise value of the fourth iteration of the CORDIC equations. These coefficients

are shown in Table 5.2. These coefficients can be used for any unit vector within the angle partition to obtain a bit accurate representation of the fourth iteration of the CORDIC equations.

Applying this concept to the calculation of a 64-bit representation of a trigonometric function shows its usefulness. Implementing the CORDIC algorithm as a direct mapped ROM table would require 18,446,744,073,709,551,616 table entries with four sixty-four-bit coefficients per entry. A Hybrid CORDIC processor used in conjunction with a Table Look-up CORDIC ROM would only require 2,097,152 table entries with four sixty-four-bit coefficients per entry. This reduced number of entries can be implemented in a table and used in conjunction with the Hybrid CORDIC hardware to implement the trigonometric functions.

Examining Table 5.2 reveals some interesting characteristic of the table coefficients. One of the first observations is that some of the columns are identical. The X variable in the X_0 column and the Y variable in the Y_0 column are exactly the same. An examination of the equations for these terms in Equations 5.10 and 5.11 show them to be identical. The X variable in the Y_0 column and the Y variable in the X_0 column are the same except for the sign. Again, an examination of the equations for these terms in Equations 5.10 and 5.11, show them to be identical except for the signs. Finally, if the signs of the coefficients are ignored, the bottom half of the table is a reflection of the top.

After reviewing the original definition of a rotation given in Equation 3.11, these observations make sense. Two of the rotation terms are the same, $\cos(\theta)$, and two of the terms are the same except for the sign, $\sin(\theta)$. Understanding these properties allows the ROM look-up table to be implemented in one fourth of the space originally calculated. Instead of requiring 2,097,152 table entries with four coefficients per entry to implement a TLC ROM, it would only require 1,048,576 table entries with two coefficients per entry. Some control logic would be required

to select the appropriate sign and table entry, but it would be minimal compared to the space saved by reducing the table.

Table 5.2 – *Multiplication Coefficients for X_{i+4} and Y_{i+4}*

Sign Bits				X_4		Y_4	
σ_0	σ_1	σ_2	σ_3	X_0	Y_0	X_0	Y_0
1	1	1	1	-0.078125	-1.640625	1.640625	-0.078125
1	1	1	-1	0.328125	-1.609375	1.609375	0.328125
1	1	-1	1	0.703125	-1.484375	1.484375	0.703125
1	1	-1	-1	1.046875	-1.265625	1.265625	1.046875
1	-1	1	1	1.265625	-1.046875	1.046875	1.265625
1	-1	1	-1	1.484375	-0.703125	0.703125	1.484375
1	-1	-1	1	1.609375	-0.328125	0.328125	1.609375
1	-1	-1	-1	1.640625	0.078125	-0.078125	1.640625
-1	1	1	1	1.640625	-0.078125	0.078125	1.640625
-1	1	1	-1	1.609375	0.328125	-0.328125	1.609375
-1	1	-1	1	1.484375	0.703125	-0.703125	1.484375
-1	1	-1	-1	1.265625	1.046875	-1.046875	1.265625
-1	-1	1	1	1.046875	1.265625	-1.265625	1.046875
-1	-1	1	-1	0.703125	1.484375	-1.484375	0.703125
-1	-1	-1	1	0.328125	1.609375	-1.609375	0.328125
-1	-1	-1	-1	-0.078125	1.640625	-1.640625	-0.078125

As larger numbers of iterations are implemented in the ROM look-up table, another important characteristic becomes apparent. Due to the requirement that the summation of all subsequent angles must be greater than or equal to the current angle, expressed in Equation 3.3, the summation of all possible rotations is greater than 90°. Because the range of the initial angle has been reduced to the range

$\left[-90^{\circ},90^{\circ}\right]$, table entries with effective rotations larger than 90° do not need to be implemented in the ROM table. In addition, there is an overlap of angle coverage around the 45°, 0°, and -45° critical angels. These duplicate angles do not need to be implemented in the ROM table either. This reduces the total number of entries required to implement the TLC ROM.

Once the look-up table has been implemented, standard arithmetic units can be used to complete the calculations. Appropriate registers and control logic are required to control the data flow, but are minimal to implement.

5.2 Table Look-up CORDIC Proof

Even though it is possible to understand how this algorithm works from the graphical representations and numerical calculations, it is important to prove that this implementation is mathematically identical to the CORDIC iterations. The Classic CORDIC iterative equations can be written using complex numbers to obtain the equation shown in Equation 5.14 [29].

$$\left(X_{i+1} + j \cdot Y_{i+1}\right) = \left(X_i + j \cdot Y_i\right)\left(1 + j \cdot \sigma_i \cdot 2^{-i}\right) \tag{5.14}$$

Setting $W_i = \left(X_i + j \cdot Y_i\right)$ and substituting it back into Equation 5.11 in the appropriate locations generates Equation 5.15.

$$W_{i+1} = W_i \cdot \left(1 + j \cdot \sigma_i \cdot 2^{-i}\right) \tag{5.15}$$

Equation 5.15 can be converted into the polar form of complex numbers to produce Equation 5.16.

$$W_{i+1} = W_i \cdot \sqrt{1 + \left(\sigma_i \cdot 2^{-i}\right)^2} \cdot e^{j \cdot \theta_i} \tag{5.16}$$

Setting $K_i = \sqrt{1 + \left(\sigma_i \cdot 2^{-i}\right)^2}$ and $\theta_i = \tan^{-1}\left(\sigma_i \cdot 2^{-i}\right)$, these values can be substituted back into Equation 5.16 to produce Equation 5.17, which is similar to the original CORDIC equation.

$$W_{i+1} = W_i \cdot K_i \cdot e^{j \cdot \theta_i} \tag{5.17}$$

Starting with $i = 0$ and performing n iterations of Equation 5.17, generates Equation 5.18.

$$W_n = W_0 K_0 K_1 \cdots K_{n-2} K_{n-1} \cdot e^{j\theta_0} e^{j\theta_1} \cdots e^{j\theta_{n-2}} e^{j\theta_{n-1}} \tag{5.18}$$

In order to simplify Equation 5.18, the variables K and θ that are defined in Equations 5.19 and 5.20 respectively, are substituted back into Equation 5.18 to generate Equation 5.21.

$$K = \prod_{i=0}^{n-1} K_i = \prod_{i=0}^{n-1} \sqrt{1 + \left(\sigma_i \cdot 2^{-i}\right)^2} = \prod_{i=0}^{n-1} \sqrt{1 + 2^{-2i}} \tag{5.19}$$

$$\theta = \sum_{i=0}^{n-1} \theta_i = \sum_{i=0}^{n-1} \tan^{-1}\left(\sigma_i \cdot 2^{-i}\right) \tag{5.20}$$

$$W_n = W_0 \cdot K \cdot e^{j \cdot \theta} \tag{5.21}$$

By selecting the appropriate initial conditions for these equations ($X_0 = 1/K$ and $Y_0 = 0$) and ensuring that the appropriate σ_i is selected at every iteration so that

Equation 5.20 approaches θ_0, Equation 5.21 will produce the sine and cosine of θ as required.

Using the definitions in Equations 5.22, 5.23, 5.24, and 5.25:

$$K_T = \prod_{i=0}^{\left\lceil \frac{N-\log_2 3}{3} \right\rceil} K_i = \prod_{i=0}^{\left\lceil \frac{N-\log_2 3}{3} \right\rceil} \sqrt{1+\left(\sigma_i \cdot 2^{-i}\right)^2} = \prod_{i=0}^{\left\lceil \frac{N-\log_2 3}{3} \right\rceil} \sqrt{1+2^{-2i}} \tag{5.22}$$

$$K_{HC} = \prod_{i=\left\lceil \frac{N-\log_2 3}{3} \right\rceil+1}^{n-1} K_i = \prod_{i=\left\lceil \frac{N-\log_2 3}{3} \right\rceil+1}^{n-1} \sqrt{1+\left(\sigma_i \cdot 2^{-i}\right)^2} = \prod_{i=\left\lceil \frac{N-\log_2 3}{3} \right\rceil+1}^{n-1} \sqrt{1+2^{-2i}} \tag{5.23}$$

$$\theta_T = \sum_{i=0}^{\left\lceil \frac{N-\log_2 3}{3} \right\rceil} \theta_i = \sum_{i=0}^{\left\lceil \frac{N-\log_2 3}{3} \right\rceil} \tan^1\left(\sigma_i \cdot 2^{-i}\right) \tag{5.24}$$

$$\theta_{HC} = \sum_{i=\left\lceil \frac{N-\log_2 3}{3} \right\rceil+1}^{n-1} \theta_i = \sum_{i=\left\lceil \frac{N-\log_2 3}{3} \right\rceil+1}^{n-1} \tan^{-1}\left(\sigma_i \cdot 2^{-i}\right) \tag{5.25}$$

Equation 5.21 can be rewritten as Equation 5.26.

$$W_n = W_0 \cdot K_T \cdot K_{HC} \cdot e^{j\cdot\theta_T} \cdot e^{j\cdot\theta_{HC}} \tag{5.26}$$

Because the multiplication in Equation 5.26 is both associative and commutative, the terms can be rearranged to produce Equation 5.27.

$$W_n = W_0 \cdot \left(K_T \cdot e^{j\cdot\theta_T}\right) \cdot \left(K_{HC} \cdot e^{j\cdot\theta_{HC}}\right) \tag{5.27}$$

Equation 5.27 clearly shows that a look-up table implementing the first portion of the CORDIC iterations can be combined with the Hybrid CORDIC algorithm to calculate accurate results.

5.3 Table Look-up CORDIC Capabilities

The proceeding proof shows that CORDIC iterations can be replaced with ROM look-up tables without a loss of accuracy in the Hybrid CORDIC algorithm. The next issue that must be addressed is whether the TLC algorithm can be used for providing an initial solution for other CORDIC algorithms in order to reduce the number of iterations they must perform. Examining Equation 5.27 shows that the TLC algorithm can be used in conjunction with any other CORDIC algorithm. As long as the look-up table contains bit accurate results for the correct number of iterations, the table's results can be used as a starting point for any of these algorithms. In addition, the table can be pre-scaled to the appropriate value required by any specific algorithm implementation.

The next question that must be answered is if there is a limit to the number of look-up tables that can be used in conjunction for calculating answers. The answer to this question is no. Taking the substitutions used to generate Equations 5.26 and 5.27 a step further, it is possible to decompose the Table Look-up CORDIC equation into a series of multiplications of sub-rotations as shown in Equation 5.28. These sub-rotations can be implemented as look-up tables following the procedures outlined in Section 5.1.2.

$$W_n = W_0 \cdot \left(K_{T1} \cdot e^{j \cdot \theta_{T1}}\right) \cdot \left(K_{T2} \cdot e^{j \cdot \theta_{T2}}\right) \cdots \left(K_{TX-1} \cdot e^{j \cdot \theta_{TX-1}}\right) \cdot \left(K_{TX} \cdot e^{j \cdot \theta_{TX}}\right) \qquad (5.28)$$

5.3.1 Traditional CORDIC Arc Tangent Radix

When calculating the TLC algorithm tables, there are two Arc Tangent Radices that can be used in calculations. The first method is to use the traditional CORDIC ATR of $\tan^{-1}\left(2^{-i}\right)$. The angle partitions generated from this ATR selection correspond with the partitions described in Section 5.1.1. The boundaries of these angle partitions are the critical angles calculated from summations of the CORDIC ATR. An example of these critical angles and their 8 and 16-bit representations for the iterations from 0 to 3 is shown in Table 5.3.

Table 5.3 – *Critical Angles for the First Three Iterations*

Critical Angle	**8-Bit Representation**	**16-Bit Representation**
71.5651	01010000	0100111111110000
45	00110010	0011001001000100
18.4349	00010101	0001010010011000
0	00000000	0000000000000000
-18.4349	11101011	1110101101101000
-45	11001110	1100110110111100
-71.5651	10110000	1011000000010000

This table highlights one of the major problems with using the traditional CORDIC ATR for calculating TLC look-up tables. The binary representations of the angles are used as the index to the ROM tables. In order to access the correct coefficients, a decoder has to take the angle and select the correct ROM look-up table entry. Examining the 8-bit angle representations, it is possible to determine that all eight of the bits are required for the decoding process. Examining the 16-bit

angle representations, it is possible to determine that fourteen of the bits are required for the decoder. As the number of bits grows and the iterations increase, the design of the decoder quickly becomes an impossible task.

Examining other methods of decoding the input angle shows them to be problematic in different ways. Content Addressable Memory (CAM) allows the contents of the memory to be used as the index. For some applications, this allows a relatively small memory to be used. For the CORDIC application, this is not a practical solution. The critical angles would be found in the CAM memory, but angles falling in between them would not have a match.

A variation of the CAM memory could be used to index the ROM look-up table. Rather than signaling whether the entry exactly match indexing angle, it could signal whether it was greater than the indexing angle. By appropriately arranging the table entries, the first entry to be less than or equal to the indexing angle would be the correct entry to use. There are several problems with this indexing scheme. The first is ensuring that that table has been properly arranged to ensure that the angles are in a decreasing order. The second is the amount of hardware required to determine if the entry is greater than the angle index. The final problem is the storage space required. Not only do the rotation angle and trigonometric coefficients need to be stored, but the critical angle must also be stored.

The easiest way to implement the traditional CORDIC ROM is to have an entry for each angle that will be used in the algorithm. The index would then be the angle and the outputs would be the rotation angle and trigonometric coefficients. For a system using n-bits to represent a number would require $3 \times n \times 2^n$ bits of storage. As the number of bits used to represent a number, n, increases, a amount of silicon area is required for the ROM look-up table quickly becomes too large to fabricate. In addition, many of the entries in the ROM look-up table would be duplicated.

5.3.2 Parallel CORDIC Arc Tangent Radix

Fortunately, the TLC algorithm does not require that the traditional CORDIC ATR be used for calculating the look-up tables. The original rotation equation shown in Equation 3.11 indicates that any angle can be used to perform the rotation. The reason the CORDIC ATR was originally used was to reduce the complexity of the computation by reducing the multiplications to simple shift operations. The ability to use any angle for the rotations in the Parallel ATR allows the designer to select angles that simplify the decoding for the ROM table entries or eliminates the need for certain arithmetic operations to be performed.

Although any angle can be selected for the rotations, the convergence property shown in Equation 3.17 must still be satisfied. The angle used to perform the rotation must produce a residual angle that is less than the maximum rotation angle of the next table. This is to ensure that each of the following tables can provide rotations that will allow the residual angle of the unit vector to converge upon zero.

Figure 5.4 shows an example of the parallel ATR being used to calculate the trigonometric functions for a n-bit number. Four tables are used to perform the calculations. To minimize the silicon area, all four tables contain one fourth of the iterations required to complete the calculations. The first $n/4$ bits of the input angle are used to index the first table. The table coefficients are the exact values of sine and cosine for the angle represented by those bits. When the residual angle is calculated, the first $n/4$ bits will all be zero because that exact angle was used to rotate the unit vector.

The second group of $n/4$ bits is then used to index the second ROM table. Again the rotation that is performed is the exact angle of the number represented by the $n/4$ bits. When the residual angle is calculated, the first $n/2$ bits will be zeros.

When this same procedure is followed for the following two groups of $n/4$ bits, each of their residual angles will be zeroed as well.

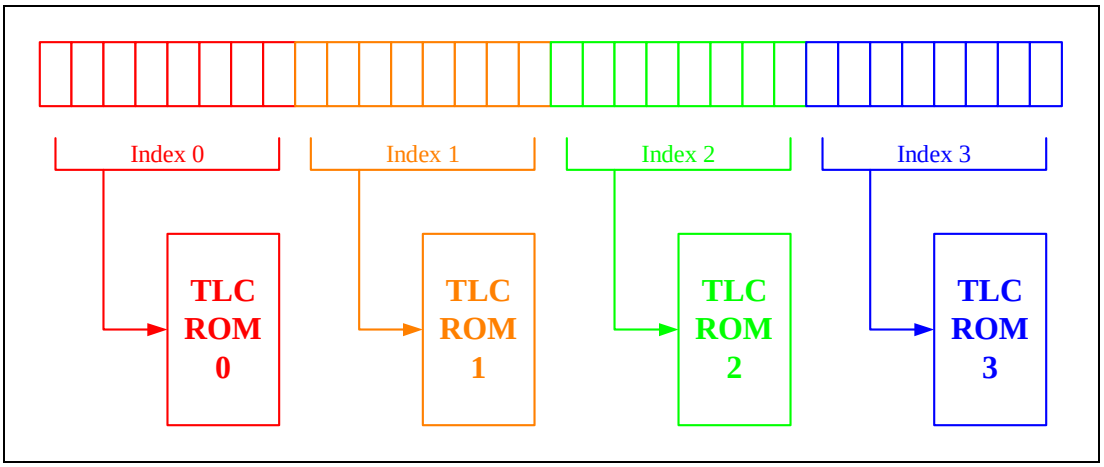


Figure 5.4 – *Parallel Arc Tangent Radix Example*

This example demonstrates three important points of the Parallel ATR. The first point is that the sign of the residual angles will always be the same. Because the rotation angle is always the exact value represented by the bits, the calculation of the residual angle will always produce zeros in those locations. The second point is that the residual angle produced by each rotation will always be smaller than the maximum rotation of the next able. Because the upper bits are all zeros, the remaining angles will all be represented in the next table. The third point is that the calculation of the residual angle can be eliminated completely. Because the calculation only eliminates the bits used to access each ROM, the others will remain the same for each operation. This also means that all of the ROM tables can be accessed in parallel because the exact angles are used; hence the name Parallel ATR.

5.4 Design Trade Offs

One of the benefits of the Table Look-up CORDIC algorithm is the amount of flexibility it provides designers. If an application requires very high speed and area is not a primary design consideration, the trigonometric functions can be implemented in two ROM look-up tables. If an application is more area sensitive and does not require the highest performance, six or more ROM look-up tables can be used. In order to assist the designer in selecting the implementation point that provides the appropriate performance without requiring an excessive silicon area, graphs of the number of operations and the amount of storage required are provided and explained in this section.

The number of operations required to perform a calculation versus the number of ROM tables is shown in Figure 5.5. The first ROM table access is essentially a look-up and does not require any arithmetic operations. This is the result of the X variable being initialized to 1 and the Y variable being initialized to 0. Because multiplication by one and zero are identities, multiplication is not required to calculate the X and the Y variables for the first ROM table. Addition is not required to combine the X and the Y variables, because addition by zero, which is the value of the Y variable, is also an identity.

ROM table accesses between the first and the last access require four multiplications and three additions. The first addition is required to determine the residual angle from the previous rotation. The residual angle is used as the index to the current table and has to be calculated first. The four multiplications are required to form the terms $X\cos\theta$, $Y\sin\theta$, $X\sin\theta$, and $Y\cos\theta$. The remaining two additions are required to form the terms $X\cos\theta - Y\sin\theta$, and $X\sin\theta - Y\cos\theta$, which are equal to the new X and Y variables respectively.

The final ROM table access requires two multiplications and two additions to calculate the answer. The first addition is again required to determine the residual angle from the previous rotation. Once this angle is determined, the appropriate

ROM table entry can be accessed. Because this is the last ROM table, only one of the two variables needs to be computed. If cosine is being calculated, the two multiplications and the addition are used to calculate the X variable from the term $X\cos\theta - Y\sin\theta$. If sine is being calculated, the two multiplications and the addition are used to calculate the Y variable from the term $X\sin\theta - Y\cos\theta$.

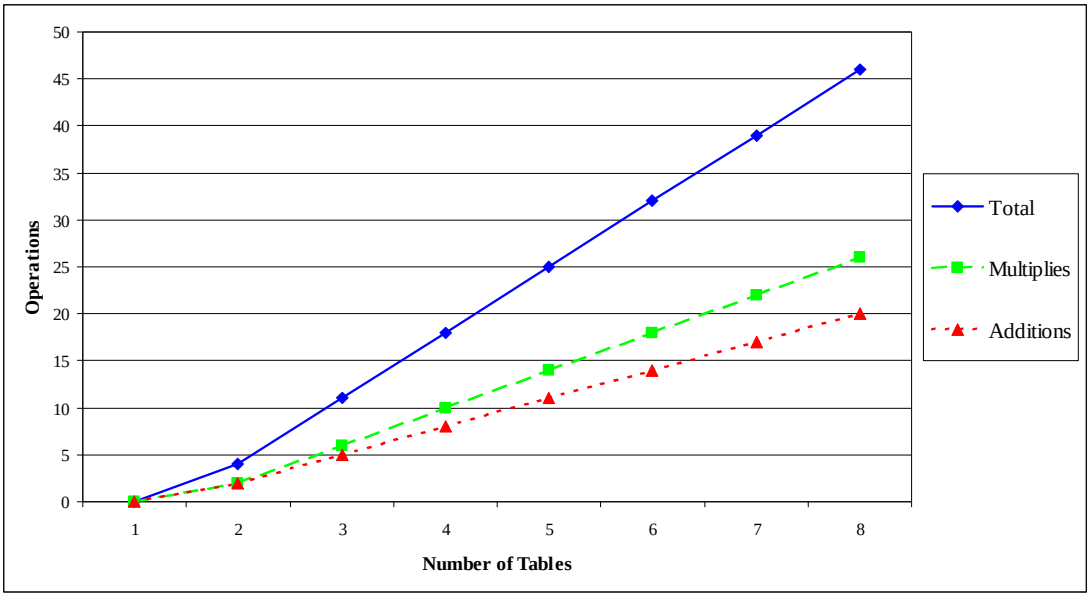


Figure 5.5 – *Operations Required to Calculate Sine or Cosine*

It is important to note the total number of operations required for the calculations is a maximum. There are several reasons why the number of operations could be lower. One reason relates to the ROM entry values of the higher iteration tables. As the rotations become smaller and small, the value of cosine approaches 1. Because the values in the tables are represented by a finite number of bits, the value stored in the entry may be 1 for the entire table. Knowing this would eliminate two of the multiplications required for those tables' accesses. Additionally, if the TLC ATR is used, the additions required in each step would be eliminated. It is always

important for the designer to understand the implementation and choose a design trade-off that benefits their application.

The storage space required for various bit widths versus the number of tables is shown in Figure 5.6. The storage size for a given table is calculated by multiplying the number of bytes required to store the coefficients by the number of entries in the ROM tables. Because there are two coefficients and a rotation angle to store in each entry, the number of bytes is $3n/8$, where n is the bit width. The number of entries depends on the number of tables being used to calculate the functions. In order to minimize the number of entries, the number of entries in each table should be equal. ROM tables where the number of entries is a power of two are easier to implement than others. In order to maintain this design advantage, the size of the tables cannot be equal when the number of tables is odd. In these instances, a portion of the tables will be twice as large as the others. The total number of entries in these tables can be calculated from Equation 5.29, where n is the bit width and t is the number of tables.

$$\frac{1}{2}\left(\left\lceil\frac{t}{2}\right\rceil\times 2^{\left\lceil\frac{n}{t}\right\rceil}+\left\lfloor\frac{t}{2}\right\rfloor\times 2^{\left\lfloor\frac{n}{t}\right\rfloor}\right) \tag{5.29}$$

As with the previous graph, the storage size requirements are a maximum. One reason for this is that the first table does not have to be fully populated. The angles to be calculated with any of the CORDIC algorithms are first range reduced to $[-\pi/2,\pi/2]$ or approximately $[-1.5708,1.5708]$. The range of representable numbers is almost $[-2,2]$, meaning that approximately a quarter of the representable angles in the first table will never be used. In addition, the higher order tables that have cosine values of one do not require a coefficient for every number. The cosine column can output a one for every entry and only require a single coefficient to be

stored. This would reduce the storage required by those tables by half. It is up to the designer to understand the application that will use these calculations and minimize storage requirements as needed.

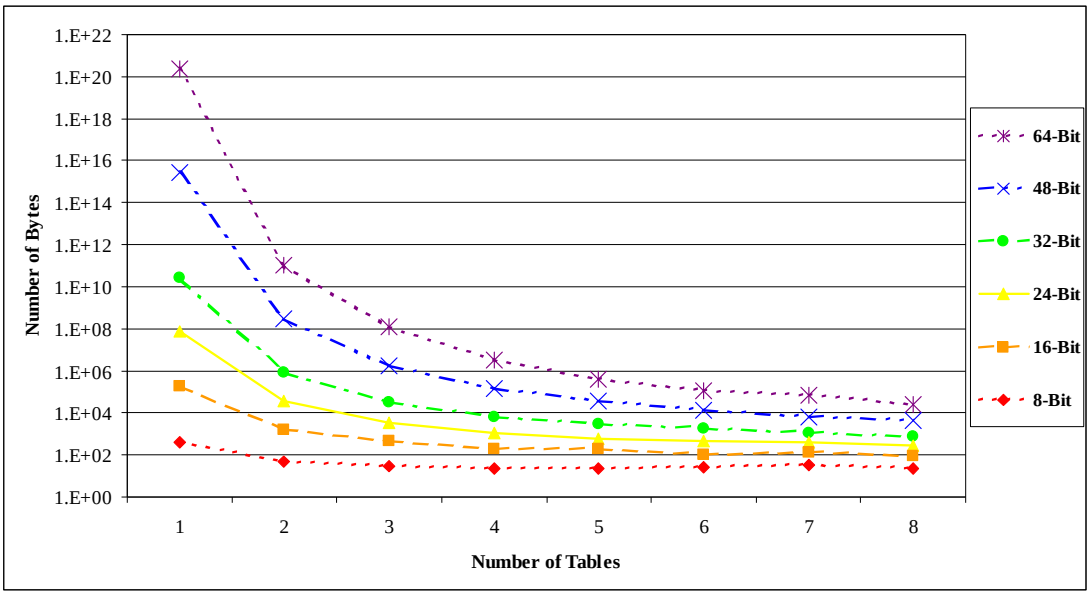


Figure 5.6 – *Bytes Required for Coefficient Storage*

Chapter 6
Software Development

Symbolically iterating through the CORDIC equations creates very complex equations. Each iteration doubles the number of terms in both the X_i and Y_i equations and adds residual angle signs, σ_i, and multiplication components, 2^{-i}, to each term. In order to reduce the computation time and ensure reliable answers, the computation process was coded in C++. This chapter outlines the data structure considerations, code implementation, and verification of the results.

6.1 Data Structures

When preparing to implement any algorithm in software, it is important to understand the calculations that will be performed in order to determine what data structures will most readily facilitate their calculation. In the case of the CORDIC iterative equations, each variable, X_i or Y_i, will be replaced by two variables, an X_{i-1} and a Y_{i-1}. These two new variables will need to retain the previous variable's coefficients, add any new coefficients (σ_{i-1} and $2^{-(i-1)}$) from the current iteration, and have the ability to change the sign of the term. Adding elements to any location should be quick and easy to implement. The two data structures considered for this implementation were a linked list structure and a bit array. Each of these data structures is discussed in the following sections.

6.1.1 Linked Lists

A linked list is a simple structure to implement in C++. Because of the nature of pointers, it is easy to add additional components to the list if the structure

has been carefully planned and implemented. Linked lists are easy to traverse with simple algorithms but do not allow immediate access to any given piece of data. Due to the iterative behavior of this algorithm, immediate access to any specific element is not a requirement.

6.1.1.1 Implementation

A linked list structure is initialized with a head pointer that points to a data structure representing the variable being calculated. This data structure is set to the upper iteration limit that is being calculated by the program. The program then calculates down to the lower iteration limit to obtain the appropriate symbolic representation. The upper and lower limits for the algorithm can be any positive integer numbers as long as the upper limit is greater than or equal to the lower limit. This allows any set of CORDIC iterations to be calculated for implementation as a ROM look-up table.

To simplify the data structures used in this example, the program calculates the iterations from 2 to 0. The head pointers for the X and Y equations and the associated initial data structures can be seen in Figure 6.1. Each of the initial data structures is given the upper iteration value of two.

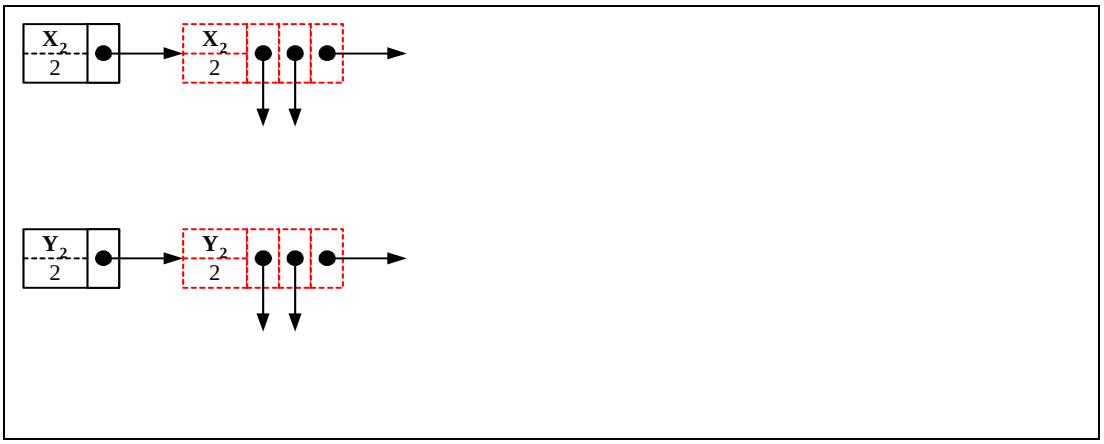


Figure 6.1 – *Initial Linked List Representation*

While the iteration count of the data structures within the linked list does not equal the lower iteration limit, the program will replace the data structures with the appropriate representation of the variable one iteration below it. In this case, the iteration equations indicate that $X_2 = X_1 - \sigma_1 2^{-1} Y_1$ and $Y_2 = Y_1 - \sigma_1 2^{-1} X_1$. Every X_2 data structure is decremented to a X_1 data structure. In addition, a Y_1 data structure with a σ_1 and a 2^{-1} component is added. The sign of this new Y_1 term is negative. Every Y_2 data structure is decremented to a Y_1 data structure and a X_1 data structure with a σ_1 and a 2^{-1} component is added. The sign of this new X_1 term is positive. Since neither the X_2 nor the Y_2 data structure had any modifying components such as σ_i or 2^{-i} (i.e., its pointers are NULL), nothing needs to be copied to the new data structures. This can be seen in graphically in Figure 6.2.

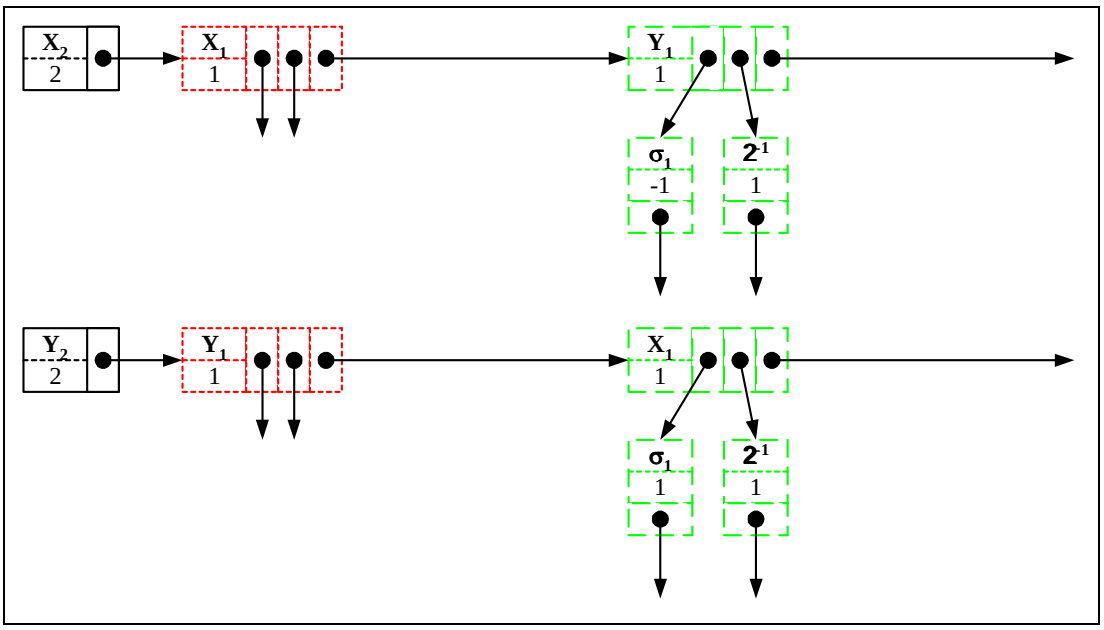


Figure 6.2 – *Linked List Representation After First Iteration*

Because the data structures within the linked list are not equal to the lower iteration limit of zero, the program replaces all of the X_1 data structures and the Y_1 data structures with X_0 and Y_0 data structures and the appropriate modifiers. In this case, the iteration equations indicate that $X_1 = X_0 - \sigma_0 2^{-0} Y_0$ and $Y_1 = Y_0 + \sigma_0 2^{-0} X_0$. All X_1 data structures are decremented to X_0 data structures. A Y_0 data structure with the appropriate modifying components is also added for each X_0. Because the X_1 data structure does not have any modifying components, no modifying components are copied to the new data structures.

All Y_1 data structures are decremented to Y_0 data structures. A X_0 data structure with the appropriate modifying components is also added. Because the Y_1 data structure has modifying components, these modifying components are copied to the new data structure as can be seen in Figure 6.3. The same reductions and replacements are required for the X_1 and Y_1 data structures in the linked list representing the Y_2 equation.

6.1.1.2 Problems

Although a linked list structure is very easy to code, it has the major drawback of memory usage. Although the storage size is dependant on the operating system and particular machine used to compile and run the algorithm, several specific problems can be identified that are independent of the platform.

Pointer variables will contain at least the same number of bits as the address space of the processor. If the algorithm is compiled and run on a 32-bit machine, each pointer will consist of at least four bytes of memory. With each element containing from one to three pointers, the amount of memory used for the pointers themselves will be significant.

Another problem is the sign of the operation for each variable. In reality, only a single bit is required for its representation. Because memory can only be allocated on byte boundaries, at least a full byte will be used to represent the sign. In addition, the integer value for the power of two in the modifier component will require at least two bytes even if the smallest integer type is used.

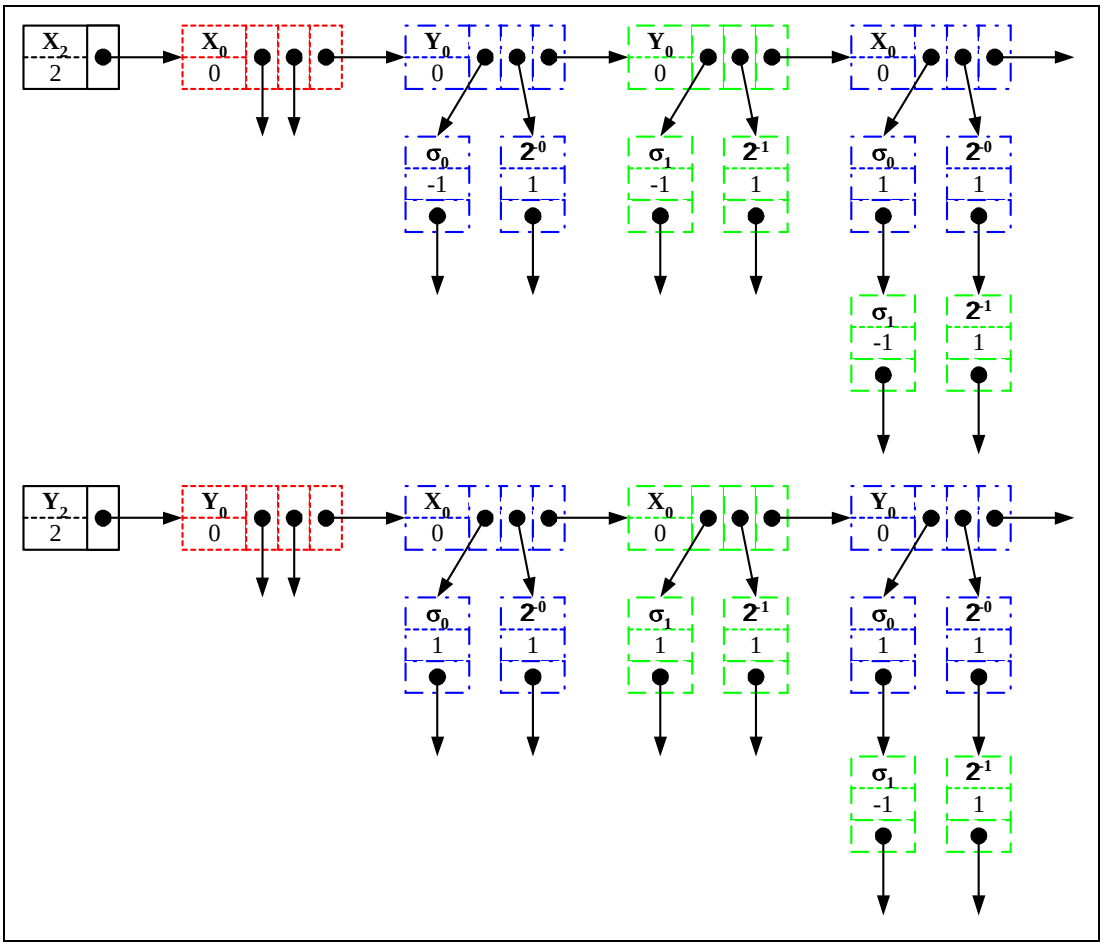


Figure 6.3 – *Linked List Representation After Second Iterations*

Because the number of terms in the iteration equations doubles with each pass through them, the number of terms increases exponentially. With each term requiring several bytes to represent, the amount of memory required to represent

both of the symbolic iteration equations can quickly increase into the hundreds of Mbytes. If the amount of memory exceeds the physical memory of the system, the operating system will start swapping memory pages to disk. This will decrease the performance of the algorithm and can possibly lead to a system crash. For this reason, the bit array was investigated as an alternative for representing the equations.

6.1.2 Bit Array

The bit array representation is investigated as an option to reduce the total memory usage needed for computing the CORDIC iterations. By representing the variable type (X_i or Y_i), the sign of the term (+ or -), the sign of the residual angle (σ_i), and power (2^{-i}) as individual bits, the amount of storage required is minimized. In addition, recognizing that all sign bits, σ_i, and powers, 2^{-i}, are found as pairs, allows the data structure to use a single bit to represent them both. The number of rows in the equation arrays is equal to the number of iterations plus two. The two additional rows are used to represent the variable type (X_i or Y_i) and the sign of the term (+ or -). The number of columns in the calculation matrices is equal to the number of terms in the final equation.

6.1.2.1 Implementation

Using the same iteration sequence that was used to demonstrate the linked list structure allows the benefits of the Bit Array structure to be clearly seen. In order to understand how the Bit Array data structure is used, the generalized CORDIC equations ($X_{i+1} = X_i - \sigma_i 2^{-i} Y_i$ and $Y_{i+1} = Y_i + \sigma_i 2^{-i} X_i$) need to be examined. When dealing with an X variable, the X_{i+1} term is replaced with an X_i and a Y_i term. The X_i term retains all of its current modifiers and does not add any new ones. The Y_i term receives all of the X_{i+1} term's modifiers and adds an additional

sign bit and power (σ_i and 2^{-i}) component. The sign of the newly added Y_i term is set to the opposite of the sign of the X_{i+1} term.

When dealing with a Y variable, the Y_{i+1} term is replaced with a Y_i and an X_i term. The Y_i term retains all of its current modifiers and does not add any additional ones. The X_i term receives all of the Y_{i+1} term's modifiers and adds an additional sign bit and multiplier (σ_i and 2^{-i}) component. The sign of the new X_i term is the same as the sign of the Y_{i+1} term.

The initial condition of the X and Y arrays can be seen in Figure 6.4. Because $X_2 = X_2$ and $Y_2 = Y_2$, the first term of the array representing the X equation is set to one in order to indicate that the variable is X. The first term of the array representing the Y matrix is set to zero in order to indicate that the variable is Y. Because both terms are positive and have no modifiers, the sign bit and the modifier bits are set to zero for these terms in both bit arrays.

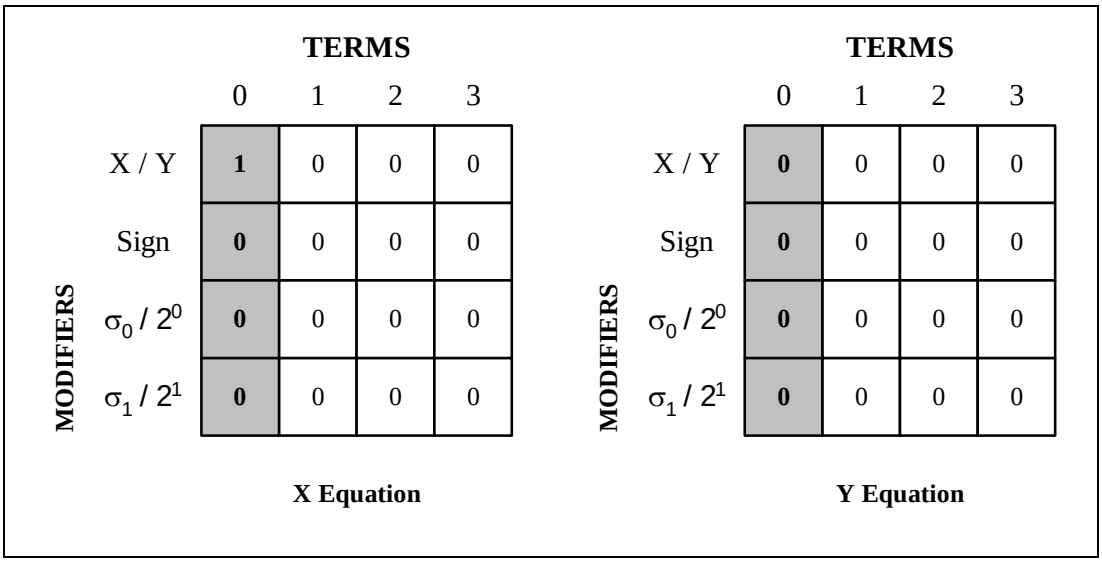


Figure 6.4 – *Initial Bit Array Representation*

Once the initial conditions for each of the equations are placed into the bit arrays, the calculation algorithm can start replacing the initial terms with their equivalents expressed in terms of lower iterations variables. The number of loops through the replacement code is determined by the iteration limits. The number of loops is equal to the difference of the upper iteration limit and the lower iteration limit. In this example, the difference in the upper and lower iteration limits is two, so the replacement code will be executed twice.

The number of terms that must be replaced starts with one term and doubles each time through the loop. If the loop variable is started at zero and incremented each time through the replacement process, the number of terms that need to be replaced can be expressed as two raised to the power of the loop variable. This expression can be used as the upper limit of a secondary loop that steps through the current terms and performs the replacements. In this example, the current replacement loop variable is equal to zero. This means that there is one term to be replaced in each bit array.

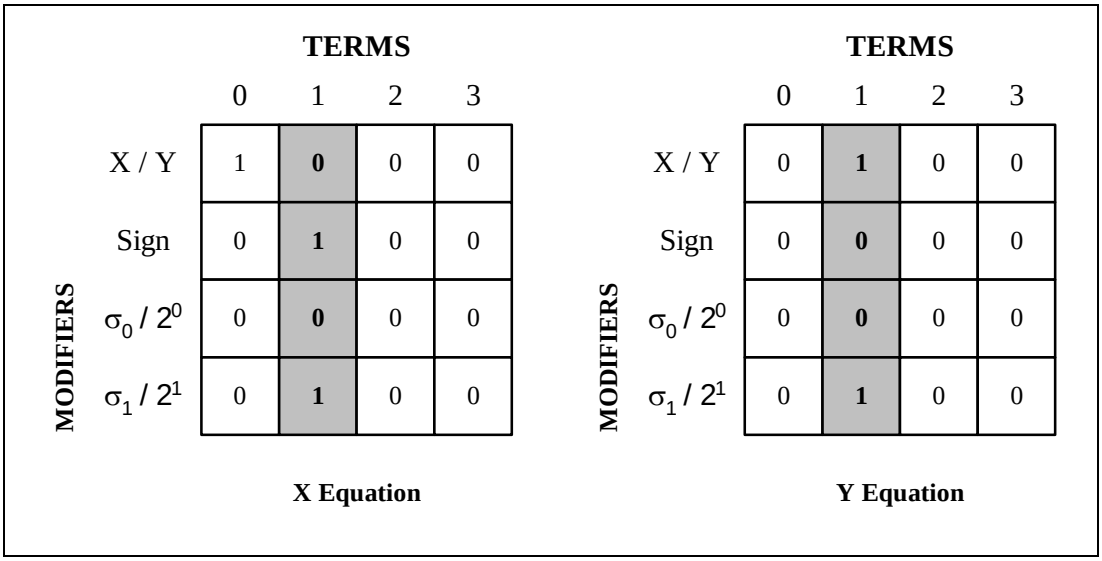


Figure 6.5 – *Bit Array Representation After First Iteration*

As has been discussed, all X_2 terms are replaced with an X_1 term and a Y_1 term with the appropriate sign and modifiers for this iteration. The only difference between the X_2 term and the X_1 term is the iteration index. Because this notation is tracked by the loop variable, nothing needs to be changed in the first column that represents the original X_2 term. The second column represents the new Y_1 term. The bit in the last row that is set to 1 represents the sign of the residual angle at this iteration step and the power while the bit in the second row that is set to 1 represents that the new Y_1 term is negative. All Y_2 terms are replaced with a Y_1 term and an X_1 term with the appropriate sign and modifiers for this iteration. A graphical representation of this operation can be seen in Figure 6.5.

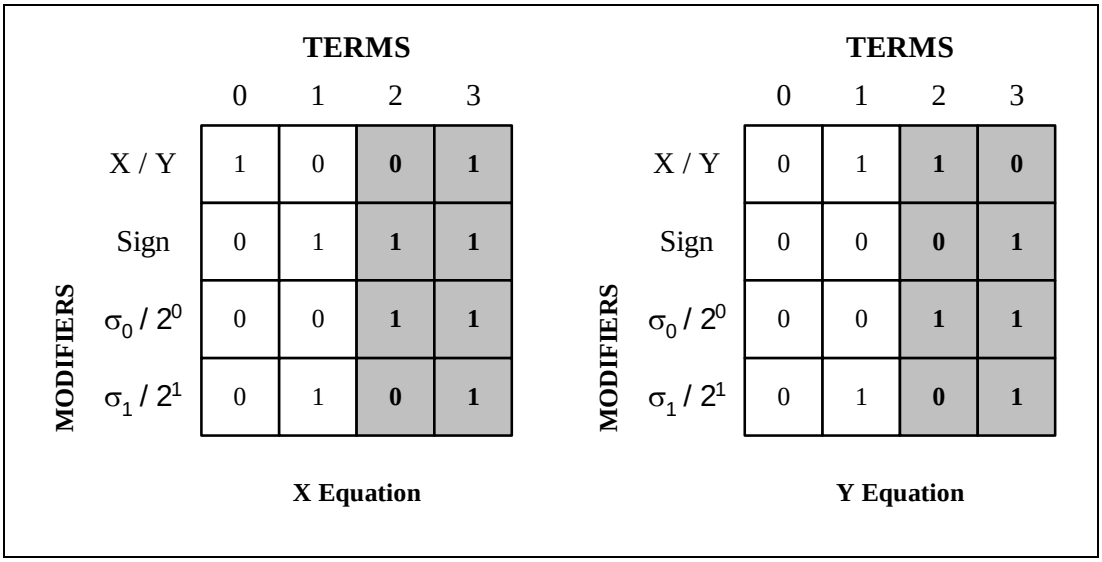


Figure 6.6 – *Bit Array Representation After Second Iterations*

Once these replacements are performed, the loop index is incremented and another series of replacements is performed. In this example, the loop index is equal to one and the number of terms to be replaced is equal to $2^1 = 2$. The X_1 and Y_1 terms in the first two columns do not have to be modified because the change in their

iteration index is handled by the loop variable. The third column represents the new Y_0 term generated from replacing the X_1 term. It receives the opposite sign and modifiers from its parent X_1 term and a new modifier from this iteration. The fourth column represents the new X_0 term generated from replacing the Y_1 term. It receives the same sign and modifiers of its parent Y_1 term and a new modifier from this iteration. The same procedure is followed to produce the new terms in the bit array representing the Y equation. This can be seen graphically in Figure 6.6.

6.1.2.2 Problems

The problem with implementing a bit array storage structure is the complexity of the code for referencing the individual bits within the data structure. The masks required for reading and writing bits to the array and the variables required to calculate the position of the bits quickly renders the code incomprehensible to most programmers. To eliminate this problem, the bit array access mechanisms can be hidden by implementing them as functions. If the functions are appropriately named, they can assist in documenting the code by making it easier to read.

6.2 C++ Coding

A C++ class object and four C++ programs were written to assist in calculating CORDIC iteration tables. The class is the Bit Array Class, which is used to hide all access details into the bit array and provide data security. The four programs were written to calculate CORDIC Iteration tables, merge multiple tables into higher order tables, verify the higher order tables, and calculate the tables faster. The data class and each of the programs are discussed in the following sections.

6.2.1 Bit Array Class

The `BitArray` class was written to improve the readability of the code by hiding the bit accesses to the array. In addition to hiding the bit accessed, the class provides data security by making the bit arrays local structures that can only be accessed through friend classes. This reduces the possibility of corrupting the data through inappropriate access to the data structure.

The `BitArray` class is made up of the Private Data Structures, Resize Function, Size Function, Clear Function, Zero All Function, One All Function, Get At Function, and Set At Function. The C++ code for this class and all of these functions is located in Appendix A.

The `BitArray::Resize` function takes a long integer as an argument and returns no arguments. The long integer passed to the Resize function is the number of elements need in the array. The Resize function calculates the number of bytes required for that number of elements and then allocates the storage space.

The `BitArray::Size` function takes no arguments and returns an unsigned integer. The unsigned integer that is returned is the number of elements, rather than the number of bytes, in the Bit Array.

The `BitArray::Clear` function takes no arguments and returns no arguments. Its purpose is to free up any previously allocated memory by resizing the array to zero elements.

The `BitArray::ZeroAll` function takes no arguments and returns no arguments. When called, this function will initialize the array elements to zeros if they have been allocated. If the size of the array is zero, nothing is initialized.

The `BitArray::OneAll` function takes no arguments and returns no arguments. It is the complement to the Zero All function. Instead of initializing the array elements to zeros, it initializes the array elements to ones.

The `BitArray::GetAt` function takes a long integer as an argument and returns a Boolean value. The long integer argument passed to the Get At function is

the array element to read from the array. The correct byte in the Bit Array and appropriate mask are selected for accessing that element. If the array element is set, a one, then TRUE is returned. Otherwise, FALSE is returned.

The `BitArray::SetAt` function takes a long integer and an integer value as arguments and returns a Boolean value. The long integer argument passed to the Set At function is the array element to set in the Bit Array. The integer value passed to the function is the value to set in the Bit Array. If the programming of the Bit Array is successful, a TRUE is returned. If the programming is unsuccessful, a FALSE is returned and an error message is printed.

6.2.2 CORDIC Iterations Program

Using the `BitArray` class, the CORDIC iterations can be programmed with a focus on the algorithm rather than the underlying data structure. The complete code for the C++ CORDIC Iterations program is located in Appendix B.

6.2.2.1 Iteration Calculations

The CORDIC Iterations program starts by querying the user for the iteration at which to start the calculations and the iteration at which to stop them. From this information, the number of iterations, the final number of terms in the equations, and the storage requirements are calculated. The bit arrays are allocated and initialized to the appropriate values.

Once the initialization is complete, the symbolic equation generation is initiated. As discussed in Section 6.1.2, each X_{i+1} term is replaced with an X_i and a Y_i term of the next lower iteration. The new X_i term retains the same sign and modifiers as the original X_{i+1} term. The new Y_i term retains the same modifiers as the original X_{i+1} term but receives the opposite sign. In addition, the new Y_i term

receives an additional modifier for the current CORDIC iteration. Each Y_{i+1} term is replaced with a Y_i and an X_i term. The new Y_i term retains the same sign and modifiers as the original Y_{i+1} term. The new X_i term retains the same sign and modifiers as the original Y_{i+1} term. In addition, the new X_i term receives an additional modifier for the current CORDIC iteration.

Once all of the terms for both the X and the Y equations have been calculated, the symbolic equations are output to a file. The file name is in the form *IterationEquation*`ll`*to*`uu`*.txt*, where `ll` is the lower iteration limit and `uu` is the upper iteration limit. The X equation is output first, followed by the Y equation. Each equation groups all X terms together and all of the Y terms together. This requires two iterations through the X bit array and two iterations through the Y bit array. The first iteration prints out all of the terms and modifiers of X terms. The second iteration prints out all of the terms and modifiers of Y terms. An example of the symbolic equations for four iterations can be seen in Figure 6.7.

```
X4 = X0 ( 1 - s3s2*2(-5) - s3s1*2(-4) - s2s1*2(-3) - s3s0*2(-3)
          - s2s0*2(-2) - s1s0*2(-1) + s3s2s1s0*2(-6) )
   + Y0 ( - s3*2(-3) - s2*2(-2) - s1*2(-1) + s3s2s1*2(-6)
          - s0*2(-0) + s3s2s0*2(-5) + s3s1s0*2(-4) + s2s1s0*2(-3) )

Y4 = X0 ( s3*2(-3) + s2*2(-2) + s1*2(-1) - s3s2s1*2(-6)
          + s0*2(-0) - s3s2s0*2(-5) - s3s1s0*2(-4) - s2s1s0*2(-3) )
   + Y0 ( 1 - s3s2*2(-5) - s3s1*2(-4) - s2s1*2(-3) - s3s0*2(-3)
          - s2s0*2(-2) - s1s0*2(-1) + s3s2s1s0*2(-6) )
```

Figure 6.7 – *Symbolic Equation Output for 4 Iterations*

Once the X and the Y symbolic equations have been output to the file, the expansion factor, K, and the rotation coefficients are calculated. With the coefficients and the expansion factor, the standard CORDIC iteration table and a normalized version of the table can be output to files. The file name for the standard

CORDIC iteration table is in the form *IterationTable*`ll`*to*uu*.txt* and the file name for the normalized iteration table is in the form *IterationNormalized*`ll`*to*uu*.txt*. As with the symbolic equations, `ll` is the lower and uu is the upper iteration limit.

In order to calculate all of the CORDIC rotations, coefficients, and effective rotation angles, an array is created to represent the signs of the residual angles. It is initialized to all zeros, representing all positive angles. Using a positive one in all of the σ_i locations, the coefficients and effective rotation angle are calculated and output to the files. The array representing the signs of the residual angles is incremented by one, with one representing a negative angle. The calculations are repeated using ones and zeros in the appropriate σ_i locations and the results are output to the files. This is repeated for all combinations of positive and negative residual angles have been calculated. An example of the standard iteration table for four iterations is show in Figure 6.8 while the normalized iteration table for four iterations is shown in Figure 6.9. The entries have been truncated to fit the page.

	XX0	XY0	YX0	YY0	Zi+1
0	-0.0781250	-1.6406250	1.6406250	-0.0781250	92.7263110
1	0.3281250	-1.6093750	1.6093750	0.3281250	78.4762782
2	0.7031250	-1.4843750	1.4843750	0.7031250	64.6538241
3	1.0468750	-1.2656250	1.2656250	1.0468750	50.4037914
4	1.2656250	-1.0468750	1.0468750	1.2656250	39.5962086
5	1.4843750	-0.7031250	0.7031250	1.4843750	25.3461759
6	1.6093750	-0.3281250	0.3281250	1.6093750	11.5237217
7	1.6406250	0.0781250	-0.0781250	1.6406250	-2.7263110
8	1.6406250	-0.0781250	0.0781250	1.6406250	2.7263110
9	1.6093750	0.3281250	-0.3281250	1.6093750	-11.5237217
10	1.4843750	0.7031250	-0.7031250	1.4843750	-25.3461759
11	1.2656250	1.0468750	-1.0468750	1.2656250	-39.5962086
12	1.0468750	1.2656250	-1.2656250	1.0468750	-50.4037914
13	0.7031250	1.4843750	-1.4843750	0.7031250	-64.6538241
14	0.3281250	1.6093750	-1.6093750	0.3281250	-78.4762783
15	-0.0781250	1.6406250	-1.6406250	-0.0781250	-92.7263110

Figure 6.8 – *Scaled Look-up Table Multiplication Coefficients*

	XX0	XY0	YX0	YY0	Zi+1
0	-0.0475651	-0.9988681	0.9988681	-0.0475651	92.7263110
1	0.1997736	-0.9798421	0.9798421	0.1997736	78.4762783
2	0.4280863	-0.9037378	0.9037378	0.4280863	64.6538241
3	0.6373730	-0.7705554	0.7705554	0.6373730	50.4037914
4	0.7705554	-0.6373730	0.6373730	0.7705554	39.5962086
5	0.9037378	-0.4280863	0.4280863	0.9037378	25.3461759
6	0.9798421	-0.1997736	0.1997736	0.9798421	11.5237217
7	0.9988681	0.0475651	-0.0475651	0.9988681	-2.7263110
8	0.9988681	-0.0475651	0.0475651	0.9988681	2.7263110
9	0.9798421	0.1997736	-0.1997736	0.9798421	-11.5237217
10	0.9037378	0.4280863	-0.4280863	0.9037378	-25.3461759
11	0.7705554	0.6373730	-0.6373730	0.7705554	-39.5962086
12	0.6373730	0.7705554	-0.7705554	0.6373730	-50.4037914
13	0.4280863	0.9037378	-0.9037378	0.4280863	-64.6538241
14	0.1997736	0.9798421	-0.9798421	0.1997736	-78.4762783
15	-0.0475651	0.9988681	-0.9988681	-0.0475651	-92.7263110

Figure 6.9 – *Normalized Look-up Table Multiplication Coefficients*

Because the CORDIC equations are represented symbolically in the Bit Arrays, the CORDIC Iterations program performs multiple passes through the arrays to calculate a single coefficient. As the number of iterations increases, the number of passes increases exponentially, taking significant time to calculate the iteration tables. The CORDIC Iterations program required in excess of 136 hours to calculate the iterations from 0 to 18.

6.2.3 CORDIC Table Merger

In order to reduce the calculation time for the iteration tables, the Table Merger program was written. Using the same method of combining multiple rotation tables as the TLC algorithm, the computation time can be significantly reduced. Tables that require large numbers of iterations can be partitioned into smaller tables

and then combined to form an equivalent table. Equation 6.1 shows the matrix multiplication required to merge two rotations by different angles.

$$\begin{bmatrix} X_{i+2} & Y_{i+2} \end{bmatrix} = \begin{bmatrix} \cos(\theta_{i+1}) & -\sin(\theta_{i+1}) \\ \sin(\theta_{i+1}) & \cos(\theta_{i+1}) \end{bmatrix} \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i) \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \end{bmatrix} \tag{6.1}$$

Performing the matrix multiplication shown in Equation 6.1 provides the calculations that must be performed to obtain each term for the new table as shown in Equation 6.2. The magnitudes of the cross terms are equivalent and do not have to be individually calculated. The code for the Table Merger program is shown in Appendix C.

$$\begin{bmatrix} X_{i+2} & Y_{i+2} \end{bmatrix} = \begin{bmatrix} \cos(\theta_{i+1})\cos(\theta_i) - \sin(\theta_{i+1})\sin(\theta_i) & -\cos(\theta_{i+1})\sin(\theta_i) - \sin(\theta_{i+1})\cos(\theta_i) \\ \sin(\theta_{i+1})\cos(\theta_i) + \cos(\theta_{i+1})\sin(\theta_i) & -\sin(\theta_{i+1})\sin(\theta_i) + \cos(\theta_{i+1})\cos(\theta_i) \end{bmatrix} \begin{bmatrix} X_i \\ Y_i \end{bmatrix} \tag{6.2}$$

6.2.4 CORDIC Table Checker

Anytime intermediate calculations are made in an attempt to reproduce other results, it is prudent to develop a method for verifying that they are the same. The Table Checker program was written to perform this function. Using the original rotation equation shown in Equation 3.11, the actual rotation coefficients can be checked by taking the sine and cosine of the effective rotation angle. This value is checked against the calculated value to ensure it is within the appropriate error bound required by the user. The complete code for the Table Checker algorithm is shown in Appendix D.

6.2.5 Fast CORDIC Iterations Program

Once the Table Checker program was written, it was possible to use the rotation equation show in Equation 3.11 to actually calculate the table coefficients directly. This eliminates the multiple loops through the symbolic equation while still producing exact results. The `BitArray` class is still used to calculate and output the symbolic equations for the iterations. Once they are completed, the Fast CORDIC Iterations program calculates the critical angles from all of the possible angle combinations. The sines and cosines of these critical angles are then calculated and output to the iteration tables with the appropriate scaling. The code for the Fast CORDIC Iterations program is shown in Appendix D.

6.3 Performance Improvement

The CORDIC Iterations and the Fast CORDIC Iterations programs include code that can calculate how much CPU time was used for each of the algorithms. These algorithms were run for multiple iterations to determine the performance enhancement from using the rotation equation to calculate the table entries. The programs were run on a 500 MHz Celeron with 256 Mbytes of memory. During the calculations of the larger iterations, the memory usage and swapping statistics were tracked to ensure that the performance of the CORDIC Iterations program was not limited by running out of memory. A comparison of the performance times is shown in Figure 6.10.

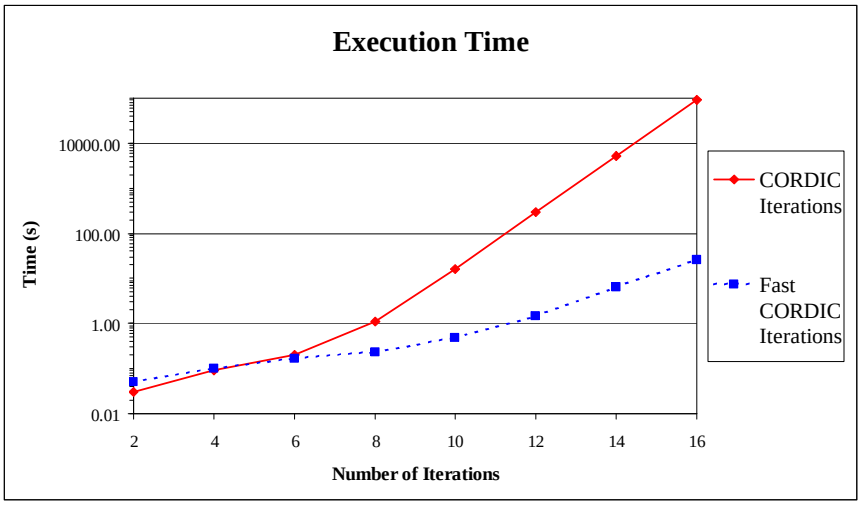


Figure 6.10 – *Execution Time of CORDIC and Fast CORDIC Programs*

Chapter 7
MATLAB Development

One of the most rapid ways to verify the functionality of a new algorithm is to model it in software. All of the algorithmic development for the Table Look-up CORDIC and other CORDIC algorithms was performed in MatLab 6.5 Release 13. This chapter details the number system representations used for the algorithms, the basic functional units developed for performing binary operations, and the top-level implementation of each CORDIC algorithm.

7.1 Number Systems

The selection of the number system has a significant impact on the capabilities of an algorithm. The number system defines the arithmetic operations that can be performed with the numbers, which in turn impacts the performance. The number system also determines how many bits of storage are required to represent each digit and the inherent accuracy of a number. The same number systems implemented by the inventors in their papers are used in these MatLab models. The Classic, Hybrid, and Table Look-up CORDIC algorithms use the two's complement number system. The Step Branching and Double Step Branching CORDIC algorithms use the Binary Signed Digit (BSD) number system.

7.1.1 Fixed Point Determination

With the number system selected by the inventors, the location of the fixed point for the two's complement and Binary Signed Digit number systems must be decided. For each of the CORDIC algorithms, there are three numbers that must be represented, the X variable, the Y variable, and the Z variable. Ideally, the

location of the fixed point would be the same for all three variables. This eliminates the need for specialized hardware for converting the numbers into the correct format, aligning the numbers for arithmetic operations, or interpreting the numbers once the calculations are complete.

7.1.1.1 Variable Ranges

The first two numbers that must be represented are the X and Y variables, which represent the (x, y) location of the head of the vector being rotated. At the start of the algorithm, the X variable is initialized to one and the Y variable is initialized to zero. As each iteration is performed, the values of X and Y change to represent the new location of the head of the vector. In addition, the magnitude of the vector being rotated increases by a factor of K_i as shown in Equation 3.17. For a 64-bit number, the value of K is approximately 1.646760. Therefore the range for the X variable is $[0,1.646760]$ and the range of the Y variable is $[-1.646760,1.646760]$.

If pre-scaling is used to eliminate the post rotation normalization calculation, the X variable is initialized with $1/K$ and the Y variable is initialized with $0/K$. Even though each rotation increases the magnitude of the vector by K_i, the variables will exceed the value of 1.000000 on the last rotation. Therefore the range for the X variable is $[0,1.000000]$ and the range of the Y variable is $[-1.000000,1.000000]$ when pre-scaling is performed.

The third number that must be represented is the rotation angle of the vector being rotated. Although any real number can be used as an angle for circular functions, all of these CORDIC algorithms assume that range reduction has been performed to limit the range of the angle to the range $[-90,90]$ degrees or $[-\pi/2, \pi/2]$ radians. Depending on whether the angle is represented in degrees or

radians, the fixed-point representation must be able to represent either the range $[-90,90]$ or the range $[-1.570796,1.570796]$, respectively.

7.1.1.2 Fixed Point Implementations

The X variable has two possible ranges: $[0,1.646760]$ for post-normalized implementations and $[0,1.000000]$ for pre-normalized implementations. In the two's complement number system, the most significant digit always represents a negative value. Both of these ranges require two fixed-point integers to properly represent the ranges. In the Binary Signed Digit number system, each digit can be positive, negative, or zero. Theoretically, both of these ranges can be represented with a single fixed-point integer in its representation. Realistically, the range of the X variable can extend into the negative domain due to the nature of the Arc Tangent Radix. To avoid possible problems, the representation would require two fixed-point integers to properly represent the ranges. These results are shown in Table 7.1.

The Y variable also has two possible ranges: $[-1.646760,1.646760]$ for post-normalized implementations and $[-1.000000,1.000000]$ for pre-normalized implementations. With respect to the two's complement number system, both of these ranges would require two fixed-point integers to properly represent the ranges. With respect to the Binary Signed Digit number system, the post-normalized implementations would require two fixed-point integers while the pre-normalized implementations would only require a single fixed-point integer. These results are summarized in Table 7.1

The Z variable has two possible ranges: $[-90,90]$ for implementations using degrees and $[-1.570796,1.570796]$ for implementations using radians. With respect to the two's complement number system, the degrees implementation would require seven fixed point integers while the radians implementation would require two fixed-point integers. With respect to the Binary Signed Digit number system, the degrees

implementation would require six fixed-point integers while the radians implementation would require a single fixed-point integer. These results are recapped in Table 7.1.

Table 7.1 – *Fixed Point Integers Required by X, Y, and Z Variables*

	***X* Variable**		***Y* Variable**		***Z* Variable**	
	Post	**Pre**	**Post**	**Pre**	**Deg.**	**Rad.**
Two's Complement	2	2	2	2	7	2
Binary Signed Digit	2	2	2	1	6	1

These results are the minimum number of fixed-point integer bits required to correctly represent the variables' ranges. It is possible to use more fixed-point integer bits in the representation, but this results in a loss of precision.

7.1.2 Two's Complement

The selection of the angle representation and the number of fixed-point integer bits is intertwined. The selection of the angle representation determines the resolution that can be obtained for residual angles. It also determines the number of fixed-point integer bits that must be used to correctly represent the Z variable. Because the resolution is similar with both angle representations, the radians representation is used so that all three variables will use the same representation.

The algorithms utilizing the two's complement number system represent the Z variable in radians and use two fixed-point integers in the representation of all of the variables. The remaining bits in the number representations are used as fractional bits. The fixed bit representation of a two's complement number can be

seen in Figure 7.1. The range of numbers that can be represented and their resolution are shown in Table 7.2.

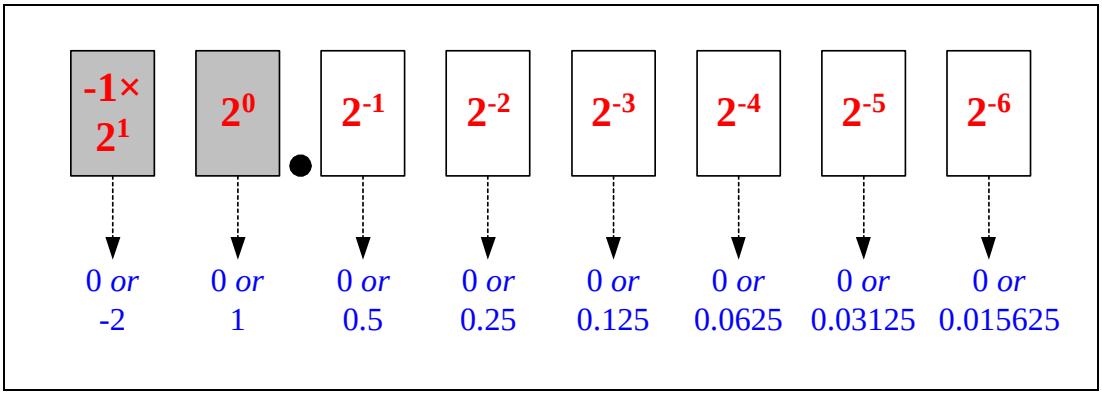


Figure 7.1 – *Fixed Point Two's Complement Representation*

Table 7.2 – *Two's Complement Range of Representable Numbers*

	Bit Width					
	8	16	24	32	48	64
Minimum	-2	-2	-2	-2	-2	-2
Maximum	$2-2^{-6}$	$2-2^{-14}$	$2-2^{-22}$	$2-2^{-30}$	$2-2^{-46}$	$2-2^{-62}$
Resolution	2^{-6}	2^{-14}	2^{-22}	2^{-30}	2^{-46}	2^{-62}

7.1.3 Binary Signed Digit (BSD)

In order to maintain consistency between the MatLab models, radians are used to represent the angles and rotations. With the selection of radians, the only remaining decision is the number of fixed-point integers to use in the representation. Even though two of the variables only require a single fixed-point integer, there are two reasons that two are used. The first reason is that it reduces software and

hardware complexity by maintaining the same representation for all variables. The second reason is that all of the numbers in both representations will have the same resolution. This will provide a more accurate comparison between the five different algorithms and their implementations.

The algorithms utilizing the Binary Signed Digit number systems represent the Z variable in radians and use two fixed-point integers in the representation of all of the variables. The remaining bits in the number representations are used as fractional bits. The fixed bit representation of a Binary Signed Digit number can be seen in Figure 7.2. The range of numbers that can be represented and their resolution are shown in Table 7.3.

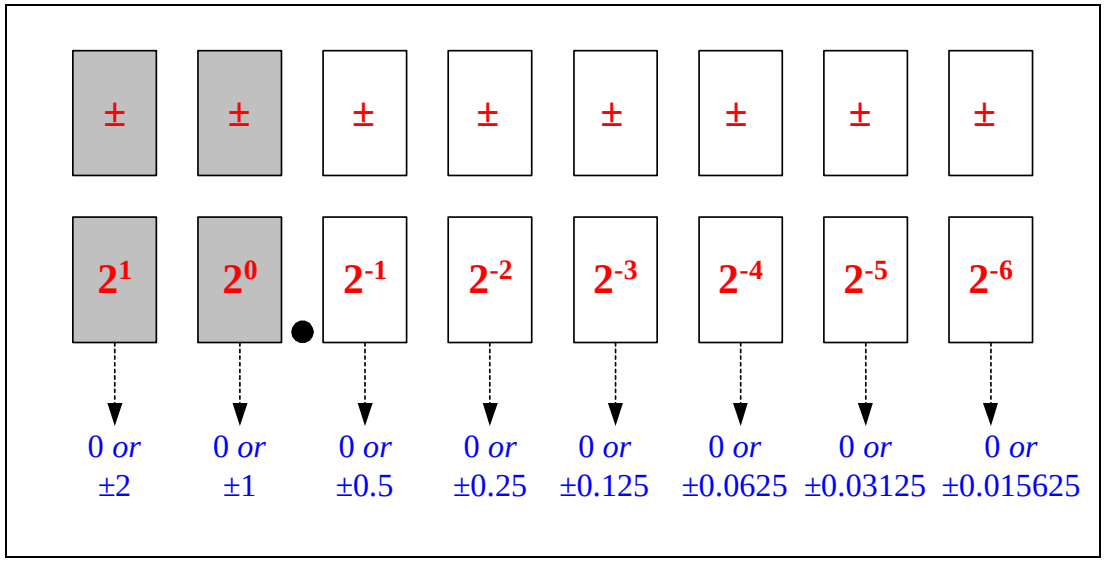


Figure 7.2 – *Fixed Point Binary Signed Digit Representation*

7.1.3.1 BSD Encoding

Because each individual digit of a Binary Signed Digit number can take on three values (positive, zero, or negative), two bits are required to encode each digit. There are actually multiple representations that can be selected. Robertson showed

that there are only nine ways to represent three values with two bits [58]. These representations can be seen in Table 7.4. For this implementation of BSD, the second permutation was selected. This representation corresponds to the first bit being the sign bit and the second bit being the digit bit was selected. When writing BSD numbers as a single digit, a 0 represents a zero, 1 represents a one, and $\bar{1}$ represents a negative one.

Table 7.3 – *Binary Signed Digit Range of Representable Numbers*

	Bit Width					
	8	16	24	32	48	64
Minimum	$-4+2^{-6}$	$-4+2^{-14}$	$-4+2^{-22}$	$-4+2^{-30}$	$-4+2^{-46}$	$-4+2^{-62}$
Maximum	$4-2^{-6}$	$4-2^{-14}$	$4-2^{-22}$	$4-2^{-30}$	$4-2^{-46}$	$4-2^{-62}$
Resolution	2^{-6}	2^{-14}	2^{-22}	2^{-30}	2^{-46}	2^{-62}

Table 7.4 – *Nine Permutation and Negation Combinations*

Digit		**Permutations**								
Ds	**Dv**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	0	0	0	0	U	0	0	1	$\bar{1}$
0	1	1	1	1	1	1	1	1	1	1
1	0	$\bar{1}$	U	0	$\bar{1}$	0	1	$\bar{1}$	0	0
1	1	0	$\bar{1}$	$\bar{1}$	U	$\bar{1}$	$\bar{1}$	$\bar{1}$	$\bar{1}$	$\bar{1}$

7.2 Functional Units

The MatLab modeling of these five CORDIC algorithms is used to accomplish three goals. The first goal is to verify that the algorithms function correctly as described in the literature. The second goal is to compare the results generated by the algorithms with the correct sine and cosine values. The third goal is to determine the accuracy of each algorithm in terms of *ulp* error and compare the performance of the established CORDIC algorithms with the new TLC algorithm.

In order to correctly determine the *ulp* error of each algorithm, the calculations need to be performed on a binary representation of the variables. Using real numbers allows higher accuracies than can be represented by a fixed bit width number representation. Unfortunately, MatLab does not provide binary variables or binary operations. MatLab stores all variables as double precision floating point numbers, including integers and character strings.

In order to correctly model the CORDIC algorithms and determine their *ulp* error, MatLab functions were written to convert decimal numbers to and from two's complement and BSD binary representations and to perform all binary arithmetic operations required by the algorithms. Each function is written so that it can perform its operation on numbers of any bit width. This requires that the number of bits being used in the calculation be provided to the function when it is called. The conversion routines and arithmetic functions are discussed in the following sections.

7.2.1 Decimal to Two's Complement (DtoB)

Before any calculations are performed in the CORDIC algorithms, the decimal numbers from MatLab are converted into a binary array representation. The `DtoB` function takes a real number and an integer as arguments and returns an array. The real number is the decimal to be converted into a binary representation. The integer is the number of bits to use in the binary representation. The array that is

returned is the two's complement binary representation of the decimal number as shown in Figure 7.1. The number of elements in the array is equal to the number of bits of accuracy passed to the function in the integer. If the decimal number cannot be exactly represented, the binary number returned is the one closest to the representation. Numbers exactly between two representations are rounded up to the next large binary representation. The MatLab code for the decimal to two's complement binary conversion is located in Appendix F.

7.2.2 Two's Complement to Decimal (BtoD)

The output of the CORDIC algorithms is easier to read and comprehend in a decimal format, therefore a function to convert two's complement binary numbers into decimals is provided. The `BtoD` function takes an array and an integer as arguments and returns a real number. The array is the two's complement binary representation of the number to convert to a decimal. The integer is the number of bits, or elements, in the array. The decimal number is the bitwise conversion of the array into a real number. The MatLab code for the binary two's complement to decimal conversion is located in Appendix G.

7.2.3 Two's Complement Adder (BinAdd)

Addition is a fundamental operation required for implementing any of the CORDIC algorithms. The logic equation for calculating the sum for a given bit position from its augend, addend, and carry input is shown in Equation 7.1. The logic equation for calculating the carry out for a given bit position from the augend, addend, and carry input is shown in Equation 7.2.

$$S_i = A_i \oplus B_i \oplus C_i \tag{7.1}$$

$$C_{i+1} = A_i B_i + A_i C_i + B_i C_i \tag{7.2}$$

The `BinAdd` function takes two arrays and two integers as arguments and returns an array and an integer. The first array represents the augend while the second array represents the addend. The first integer is the number of bits, or elements, in the arrays. The second integer is the value of the carry input. If the value of the carry input is not provided to the function, it is assumed to zero. The array returned from the function is the summation of the addend, the augend, and the carry input. The returned integer is the value of the overflow calculation. If an overflow occurs, this integer is set to one. Otherwise the integer is zero. The MatLab code for the two's complement adder is shown in Appendix H.

7.2.4 Two's Complement Subtractor (BinSub)

Subtraction can be defined in terms of addition. If the subtrahend of an operation is negated and then added to the minuend of the operation, the result is the difference between the two numbers. To negate a two's complement number, all of the bits are inverted and then a *ulp* is added to the inverted bits. The result of this operation is the negation of the original two's complement number. Using this technique, a two's complement subtraction can be implemented by adding the minuend, the bitwise inversion of the subtrahend, and a carry in of one.

The `BinSub` function takes two arrays and an integer as arguments and returns an array and an integer. The first array represents the minuend while the second array represents the subtrahend. The integer is the number of bits or elements in the array. The array that is returned is the differences of the minuend and subtrahend arrays. The integer that is returned is the value of the overflow calculation. If an overflow occurs, this integer is set to one. Otherwise the integer is zero. The MatLab code for the two's complement subtractor is shown in Appendix I. As the code clearly shows, the subtraction algorithm functions by complementing the subtrahend and then adding it to the minuend.

7.2.5 Two's Complement Shift (RShift)

In any number representation, multiplication and division by the radix can be implemented through shift operations. Shifts to the left are equivalent to multiplication while shifts to the right are equivalent to division. The CORDIC algorithm requires a multiplication by a power of two, which can be implemented as a shift in a binary representation. Because the exponent of the radix is negative, the operation is a multiplication by a fraction. Multiplication by a fraction is equivalent to a division, so the shifts are to the right.

The `RShift` function takes an array and two integers as arguments and returns an array. The input array is the two's complement representation of the number to divide by shifting it to the right. The first integer is the number of places to shift the two's complement representation while the second integer is the number of bits, or elements, in the array. The returned array is the shifted version of the input array. For two's complement numbers, it is important that all bits shifted into the array from the left have the same value as the most significant bit (msb) of the original array. This ensures that the value and sign of the shifted number is correct. The MatLab code for the two's complement shifter is shown in Appendix J.

7.2.6 Decimal to Redundant Binary (DtoRB)

Just like the `DtoB` function, the `DtoRB` function takes a real number and an integer as arguments and returns an array. The real number is the decimal that is converted into a redundant binary representation. The integer is the number of bits to use in the representation. The array that is returned is the Binary Signed Digit representation of the decimal number as shown in Figure 7.2. The number of elements in the array is equal to the number of bits of accuracy passed to the function in the integer. If the decimal number cannot be exactly represented, the binary number returned is the one closest to the representation. Numbers exactly between

two representations are rounded up to the next large binary representation. The MatLab code for the decimal to redundant binary conversion is in Appendix K.

7.2.7 Redundant Binary to Decimal (RBtoD)

As with the two's complement numbers, a function is provided to convert redundant binary numbers into decimals. The `RBtoD` function takes an array and an integer as arguments and returns a real number. The array is the Binary Signed Digit representation of the number to convert to a decimal. The integer is the number of bits, or elements, in the array. The real number that is returned is the bitwise conversion of the array into a decimal number. The MatLab code for the redundant binary to decimal conversion is shown in Appendix L.

7.2.8 Redundant Binary Adder (RBinAdd)

Avizienis' mathematical description of redundant binary addition was first turned into a logic design by Borovec in 1968 [59]. Borovec's paper describing the implementation of redundant addition and development of the logic equations overlooked one of the possible variations. In 1978, Chow and Robertson published a paper on the implementation of the overlooked variation and showed that it produced simpler logic equations than those developed by Borovec [60]. Equations 7.3, 7.4, 7.5, 7.6 and 7.7 show the logic equations used to code the redundant binary addition operation for the MatLab `RBinAdd` function. A_{si} represents the sign of the augend while A_{di} represents the digit of the augend. B_{si} represents the sign of the addend while B_{di} represents the digit of the addend. S_{si} represents the sign of the sum of A and B, while S_{di} represents the digit of the sum of A and B.

$$m_{i+1} = \overline{A}_{si}\,\overline{B}_{si}\left(A_{di} + B_{di}\right) \tag{7.3}$$

$$b_{i+1} = \overline{m}_i\,\overline{A}_{di}B_{di} + \overline{m}_i A_{di}\,\overline{B}_{di} + A_{si}B_{si} \tag{7.4}$$

$$d_i = m_i \oplus A_{di} \oplus B_{di} \tag{7.5}$$

$$S_{si} = \overline{d}_i b_i \tag{7.6}$$

$$S_{di} = d_i \oplus b_i \tag{7.7}$$

The `RBinAdd` function takes two arrays and an integer as arguments and returns an array and an integer. The first input array is the augend while the second input array is the addend. The integer is the number of bit, or elements, in the array. The returned array is the Signed Binary Digit representation of the summation of the two input numbers. The integer that is returned is the value of the overflow calculation. If an overflow occurs, this integer is set to one. Otherwise the integer is zero. The MatLab code for the redundant binary adder is shown in Appendix M.

7.2.9 Redundant Binary Subtractor (RBinSub)

In addition to developing the appropriate addition logic equations, Chow and Robertson developed subtraction logic equations for redundant number representations [60]. Equations 7.8, 7.9, 7.10, 7.11, and 7.12 show the equations used to code the redundant binary subtraction operation for the MatLab `RBinSub` function. A_{si} represents the sign of the minuend while A_{di} represents the digit of the minuend. B_{si} represents the sign of the subtrahend while B_{di} represents the digit of the subtrahend. S_{si} represents the sign of the difference of A and B, while S_{di} represents the digit of the difference of A and B.

$$m_{i+1} = B_{si} + \overline{A}_{si} A_{di} \overline{B}_{di} \tag{7.8}$$

$$b_{i+1} = \overline{m}_i \overline{A}_{di} B_{di} + \overline{m}_i A_{di} \overline{B}_{di} + A_{si} B_{di} \tag{7.9}$$

$$d_i = m_i \oplus A_{di} \oplus B_{di} \tag{7.10}$$

$$S_{si} = \overline{d}_i b_i \tag{7.11}$$

$$S_{di} = d_i \oplus b_i \tag{7.12}$$

The `RBinSub` function takes two arrays and an integer as arguments and returns an array and an integer. The first input array is the minuend while the second input array is the subtrahend. The input integer is the number of bits, or elements, in the arrays. The returned array is the Binary Signed Digit representation of the difference between the two input numbers. The integer that is returned is the value of the overflow calculation. If an overflow occurs, this integer is set to one. Otherwise the integer is zero. The MatLab code for the redundant binary subtractor is shown in Appendix N.

7.2.10 Redundant Binary Shifter (RRShift)

As with the two's complement numbers, a shift function is provided to perform the multiplication by a power of two operation. The `RRShift` function takes an array and two integers as arguments and returns an array. The input array is the Binary Signed Digit representation of the number to divide by shifting it to the right. One integer is the number of places to shift the BSD representation while the other is the number of bits, or elements, in the array. The returned array is the shifted version of the input array. Because each digit retains its own sign, the bits shifted into the array from the left are all zeros. This ensures that the value and sign of the shifted number is correct. The MatLab code for the two's complement shifter is shown in Appendix O.

7.2.11 Rotation Operations (SBC_Rotate)

The example code for the Step Branching algorithm described in [46] was written to focus on the new Step Branching technique. The rotation of the unit vector and the residual angle calculations were implemented as the `updatez` function. The Pascal code implementing the `updatez` function did not even update the X or Y variables. The description of these updates was discussed in the following section. Using this realization as a template, the rotations are implemented in a separate function in MatLab. This not only follows the original coding, but improves the readability of the code as well.

The `SBC_Rotate` function takes six arrays and four integers as arguments and returns six arrays. The first input array represents the X_p variable. The second input array represents the X_n array. The third input array represents the Y_p variable. The fourth input array represents the Y_n variable. The fifth input array represents the Z_p variable. And the sixth input array represents the Z_n variable. First two integers represent the sign of the residual angles Z_p and Z_n, respectively. The third integer represents the current iterative rotation. The fourth integer represents the number of bits, or elements, in the arrays. The output arrays are the newly calculated X_p, X_n, Y_p, Y_n, Z_p, Z_n variables, respectively, in Signed Binary Digit representation. The MatLab code for the Step Branching rotation is shown in Appendix P.

7.2.12 Redundant Angle Evaluation (SBC_AngleEval)

In addition to coding the angle rotations as the `updatez` function [46], the evaluation of the sign of the residual angles Z_p and Z_n were also coded as a

separate function, `eval`. As before, this coding style has been retained to improve readability and allow better comparisons between the Pascal and MatLab code.

The `SBC_AngleEval` function takes two arrays and two integers as arguments and returns two integers. The two input arrays are the residual angles Z_p and Z_n, respectively, of the previous rotation. The first input integer is current iterative rotation while the second input integer is the number of bits, or elements, in the arrays. The two returned integers are the signs of the residual angles S_p and S_n, respectively. The MatLab code for the Step Branching angle evaluation is shown in Appendix Q.

7.2.13 Double Step Branching Rotate (DSBC_Rotate)

In order to focus on the Double Step Branching algorithm described in [51] arithmetic operations that are performed multiple times are implemented as function in the MatLab model. This improves the readability of the code and focuses on the algorithm and not the calculations. The `DSBC_Rotate` function takes three arrays and four integers as arguments and returns three arrays. The three input arrays represent the X, Y, and Z variables, respectively, for either the alpha or the beta hardware. The first two input integers are the residual angles for the rotations that are performed in this iterative rotation. The rotations will be in the opposite direction of the sign of the residual angle. The third input integer is the current value of the iterative rotation. The fourth input integer is the number of bits, or elements, in the arrays. The three returned arrays are the newly calculated values for the X, Y, and Z variables, respectively. The MatLab code for the Double Step Branching rotation is shown in Appendix R.

7.2.14 Double Step Branching Angle Evaluation (DSBC_AngleEval)

The `DSBC_AngleEval` function takes an array and two integers as arguments and returns two integers. The input array is the residual angle Z from either the alpha or the beta hardware. The first input integer is the current iterative rotation while the second input integer is the number of bits, or elements, in the array. The first returned integer is the sign of the residual angle, Z, while the second returned integer is a flag variable used in determining whether the alpha or the beta hardware has the correct answer. The MatLab code for the Double Step branching angle evaluation is shown in Appendix S.

7.2.15 Hybrid Content Addressable Memory (HCCAM)

The `HCCAM` function takes an array and an integer as arguments and returns an integer. The input array represents the angle to calculate, Z. The input integer is the number of bits, or elements, in the array. The output integer represents index of the table that contains the rotation parameters to initialize the Hybrid CORDIC algorithm. The MatLab code for the Hybrid CAM is shown in Appendix T.

7.2.16 Hybrid Pipeline (HCPipe)

The `HCPipe` function takes three arrays and three integers as arguments and returns three arrays. The three input arrays represent the X, Y, and Z variables, respectively. The first and second input integers represent the start and stop iterations for the Hybrid Pipeline to use in its calculations. The third input integer is the number of bits, or elements, in the arrays. The three output arrays are the newly calculated X, Y, and Z variables, respectively, in two's complement representation. The MatLab code for the Hybrid calculation pipeline is shown in Appendix U.

7.2.17 Two's Complement Multiplication (BinMult)

The `BinMult` function takes two arrays and an integer as arguments and returns an array. The first input array is the multiplier while the second input array is the multiplicand. The input integer is the number of bits, or elements, in the array. The returned array is the product of the two input arrays in Signed Binary Digit representation. The MatLab code for the two's complement multiplier is shown in Appendix V.

7.2.18 Binary to Index (BtoIndex)

The `BtoIndex` function takes an array and two integers as arguments and returns an integer. The input array is the angle being calculated by the TLC algorithm. The first input integer is the most significant bit (MSB) of the Parallel ATR while the second input integer is the least significant bit (LSB) of the Parallel ATR. The returned integer is the value to use as the index to the TLC ROM look-up table. The MatLab code for the two's complement shifter is shown in Appendix W.

7.3 Algorithm Implementation

The following sections discuss the CORDIC algorithms that are modeled in MatLab and simulated. Each section discusses the implementation of the algorithm and any differences between the implementation and the algorithm as its inventor describes it.

When examining the MatLab code in the appendices, it is important to pay attention to the loop indices. Many of the CORDIC algorithms iterate from 0 to $n-1$ during the course of the calculations. In MatLab, all array indexes must be positive integer numbers. Due to this limitation, the CORDIC algorithms in the MatLab models iterate from 1 to n. The appropriate powers, loop variables, and arctangent calculation formulas are adjusted to compensate for this difference.

7.3.1 Classic CORDIC (CCordic)

The Classic CORDIC algorithm is a simple implementation of the original algorithm presented in [24]. No modifications are required to implement the algorithm in MatLab. The `CCordic` function takes a real number and two integers as arguments and returns three arrays and three real numbers. The input real number is the angle, in degrees, to calculate. The first input integer is the number of iterations to perform while the second input integer is the number of bits to use in the binary representation of the answer. The returned arrays are the X, Y, and Z variables, respectively, calculated by the `CCordic` function in the two's complement representation. The three returned real numbers are the X, Y, and Z variables, respectively, of the returned arrays as decimal numbers. The MatLab code for the Classic CORDIC algorithm is shown in Appendix X.

7.3.2 Step Branching CORDIC (SBCordic)

The Step Branching CORDIC is a close implementation of the Pascal algorithm presented in [46]. The only difference is in the number of iterations performed. The Pascal program is written as an infinite loop. There is no capacity for performing a certain number of iterations and then exiting the program. For the comparisons between the different CORDIC algorithms to be equitable, each algorithm should perform the same number of iterations. In the MatLab models, each of the loops is qualified by an iterations variable to ensure that only a given number of iterations are performed.

This modification introduced a problem. Because the MatLab model can exit from any of the loops, it is possible to reach the upper iteration limit while the algorithm is in a branch. The correct answer is determined by performing a comparison of the residual angles of both hardware units. The hardware with the

smallest residual angle contains the correct answer and is output. This requires an extra subtraction and control logic.

The `SBCordic` function takes a real number and two integers as arguments and returns three arrays and three real numbers. The input real number is the angle, in degrees, to calculate. The first input integer is the number of iterations to perform while the second input integer is the number of bits to use in the binary representation of the answer. The returned arrays are the X, Y, and Z variables, respectively, calculated by the `SBCordic` function in the Binary Signed Digit representation. The three returned real numbers are the X, Y, and Z variables, respectively, of the returned arrays as decimal numbers. The MatLab code for the Step Branching CORDIC algorithm is shown in Appendix Y.

7.3.3 Double Step Branching CORDIC (DSBCordic)

The code in the Double step branching algorithm uses $\lceil (n+3)/2 \rceil$ iterations to ensure the number is accurate the tighter bound. This limit was changed to $\lceil n/2 \rceil$ iterations to match other algorithms modeled in MatLab. The Double Step Branching algorithm also has the problem of being in a branch when the algorithm terminates. The difference between the two residual angles is used to determine the correct hardware.

The `DSBCordic` function takes a real number and two integers as arguments and returns three arrays and three real numbers. The input real number is the angle, in degrees, to calculate. The first input integer is the number of iterations to perform while the second input integer is the number of bits to use in the binary representation of the answer. The returned arrays are the X, Y, and Z variables, respectively, calculated by the `DSBCordic` function in the Signed Binary Digit representation. The three returned real numbers are the X, Y, and Z variables,

respectively, of the returned arrays as decimal numbers. The MatLab code for the two's complement shifter is shown in Appendix Z.

7.3.4 Hybrid CORDIC (HCordic)

The `HCordic` function takes a real number and two integers as arguments and returns three arrays and three real numbers. The input real number is the angle, in degrees, to calculate. The first input integer is the number of iterations to perform while the second input integer is the number of bits to use in the binary representation of the answer. The returned arrays are the X, Y, and Z variables, respectively, calculated by the `HCordic` function in the two's complement representation. The three returned real numbers are the X, Y, and Z variables, respectively, of the returned arrays as decimal numbers. The MatLab code for the Hybrid CORDIC algorithm is shown in Appendix AA.

7.3.5 Table Look-up CORDIC

The `TLCordic` function takes a real number and two integers as arguments and returns three arrays and three real numbers. The input real number is the angle, in degrees, to calculate. The first input integer is the number of iterations to perform while the second input integer is the number of bits to use in the binary representation of the answer. The returned arrays are the X, Y, and Z variables, respectively, calculated by the `TLCordic` function in the two's complement representation. The three returned real numbers are the X, Y, and Z variables, respectively, of the returned arrays as decimal numbers. The MatLab code for the Table Look-up CORDIC algorithm is shown in Appendix BB.

7.4 Results

Before results from any simulation are analyzed it is important to understand the types of errors that can affect the calculations. Fixed width arithmetic units, number representations, rounding, noise, and other factors can affect the accuracy of simulation results. Understand the source of these errors and how they affect the results allows bounds to be calculated for the error. This enables the results to be compared to the error bound limits in order to ensure that the numbers obtained from the calculations are correct.

Because the CORDIC algorithms perform multiple additions on shifted operands, there are segments of the numbers that are not being added into the final answer due to the fixed width of the adders. These bits can add up to a significant amount of error. Figure 7.3 shows the error that can accumulate through the rotation operations. This error can be quantified. Equation 7.13 shows several versions of the formula that can be used to calculate the amount of error in terms of *ulp*.

$$\sum_{i=1}^{n}(n-i)2^{-i} = \sum_{i=1}^{n}\frac{n-i}{2^{i}} = \sum_{i=0}^{n-1}(n-1-i)2^{-(i+1)} = \sum_{i=0}^{n-1}\frac{n-1-i}{2^{i+1}} \tag{7.13}$$

There are several implementations that can be used to reduce the error introduced by fixed width arithmetic units. One method, described by Eric King [63], involves using the carry input to adders as an additional input bit. This simple modification reduces the error by more than fifty percent. Another method described in [63] involves intelligently adding or subtracting a bias from the final calculation to move the answer in the appropriate direction. Even though both of these methods offer improved performance, neither of them is implemented in any of the arithmetic modules used in the MatLab simulations or the Verilog coding.

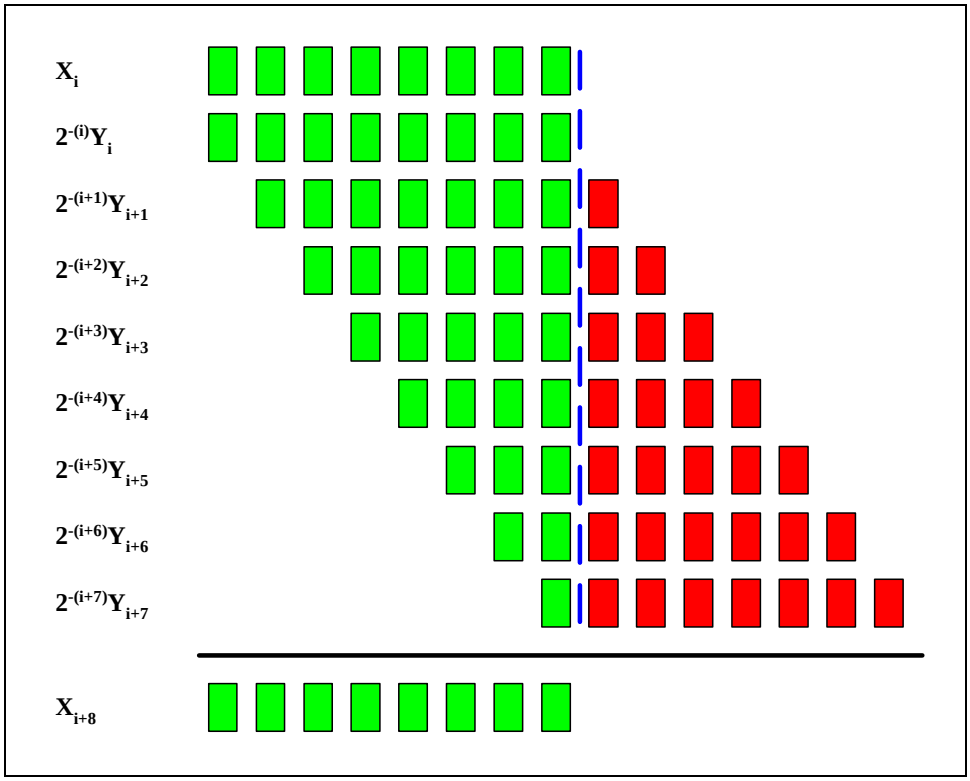


Figure 7.3 – *Addition Error Due to Fixed Width Operations*

Each algorithm was simulated with two different angle increments. The first calculated 10,000 angles between –90 and 90, or every 0.018 degrees. The second simulation calculated every 0.05 degrees. The results from these simulations were converted into *ulp* error. Negative values indicate that the number is smaller than the correct value while positive numbers indicate that the number is larger than the correct answer.

7.4.1 Classic CORDIC

The Classic CORDIC algorithm simulations show an error less than the maximum possible error that can occur according to Equation 7.13. This is because no angle will always be rotated in the same direction for all its rotations. Because it

is not rotating in the same direction, some of the error will cancel itself out, thus reducing the maximum error. The results from the MatLab simulations of the Classic CORDIC algorithm are shown in Table 7.5.

Table 7.5 – *Classic CORDIC Calculation Error in ulps*

		Bit Width					
		8	16	24	32	48	64
Theory	Upper Limit	-6	-14	-22	-30	-46	-62
	Lower Limit	6	14	22	30	46	62
Cosine	Upper Limit	-3	-9	-12	-15	-15	-856
	Lower Limit	3	8	12	12	19	299
Sine	Upper Limit	-5	-10	-13	-15	-17	-946
	Lower Limit	4	9	11	12	18	944

7.4.2 Step Branching CORDIC

The Step Branching algorithm produced similar results to the Classic CORDIC algorithm, but slightly worse to the fact that it can select the angle further from zero for a branching rotation. The results from the MatLab simulations of the Step Branching CORDIC algorithm are shown in Table 7.6.

7.4.3 Double Step Branching CORDIC

Due to the change in the algorithm, the results are better than expected. The comparison of residual angles at each step always selects the angle closer to zero, providing results almost as good as the Classic CORDIC algorithm. The results from

the MatLab simulations of the Double Step Branching CORDIC algorithm are shown in Table 7.7.

Table 7.6 – *Step Branching CORDIC Calculation Error in ulps*

		Bit Width					
		8	16	24	32	48	64
Theory	Upper Limit	-6	-14	-22	-30	-46	-62
	Lower Limit	6	14	22	30	46	62
Cosine	Upper Limit	-3	-4	-7	-8	-9	-852
	Lower Limit	2	4	6	7	9	292
Sine	Upper Limit	-2	-5	-7	-8	-11	-944
	Lower Limit	3	5	8	7	10	940

Table 7.7 – *Double Step Branching CORDIC Calculation Error in ulps*

		Bit Width					
		8	16	24	32	48	64
Theory	Upper Limit	-6	-14	-22	-30	-46	-62
	Lower Limit	6	14	22	30	46	62
Cosine	Upper Limit	-2	-4	-6	-8	-10	-854
	Lower Limit	2	5	6	7	9	292
Sine	Upper Limit	-3	-4	-7	-8	-10	-943
	Lower Limit	3	6	6	7	8	943

7.4.4 Hybrid CORDIC

The Hybrid CORDIC algorithm requires that at least a third of the calculations be performed by traditional CORDIC methods. In order to minimize the hardware implementation the number of pipelined Hybrid CORDIC iterations that are performed as a group is four. To keep from redesigning the hardware, the actual number of CORDIC iterations that were used is shown in Table 7.8.

Table 7.8 – *Hybrid CORDIC ROM Table Requirements*

	Bit Width					
	8	16	24	32	48	64
Minimum CORDIC Iterations	3	5	8	11	16	21
Implemented CORDIC Iterations	4	8	8	12	16	24

The results from the Hybrid CORDIC simulations are shown in Table 7.9. The N/A in the table indicates that the calculations could not be performed. Due to the size of the ROM tables needed to simulate the Hybrid algorithm, MatLab ran out of memory. The *ulp* error for the smaller simulations showed excellent results.

7.4.5 Table Look-up CORDIC

The Table Look-up CORDIC algorithm should have excellent results. For the smaller bit widths, the calculations a pure table look-ups that are bit accurate. For the bit widths that require a calculation, a full multiplier is used. This lowers the introduced error to a single *ulp* per multiplication and addition. The lower error bound is shown in Table 7.10. The results for the TLC algorithm produces the expected results for 8 and 16 bit widths, and even better results for higher bit widths.

Table 7.9 – *Hybrid CORDIC Calculation Error in ulps*

		Bit Width					
		8	16	24	32	48	64
Theory	Upper Limit	-4	-8	-16	-20	-32	-40
	Lower Limit	4	8	16	20	32	40
Cosine	Upper Limit	-3	-4	-6	-7	-9	N/A
	Lower Limit	2	3	5	7	9	N/A
Sine	Upper Limit	-2	-4	-6	-8	-10	N/A
	Lower Limit	2	4	6	7	9	N/A

Table 7.10 – *Table Look-up CORDIC Calculation Error in ulps*

		Bit Width					
		8	16	24	32	48	64
Theory	Upper Limit	0	0	-2	-4	-6	-8
	Lower Limit	0	0	2	4	6	8
Cosine	Upper Limit	0	0	-1	-2	-2	-740
	Lower Limit	0	0	1	2	2	729
Sine	Upper Limit	0	0	-1	-2	-2	-781
	Lower Limit	0	0	1	2	2	762

Chapter 8
Verilog Development

The MatLab models were converted to Verilog models in order to prepare the algorithms for hardware realization. In order to ensure that the correct arithmetic units are selected, all of the arithmetic units are implemented in Verilog as structural code. This eliminates the possibility of the synthesizer selecting inappropriate circuitry for critical components of the algorithm. Non-arithmetic units, such as the control circuitry, are implemented behaviorally.

8.1 Major Functional Units

The structural Verilog code for the major functional units is discussed in the following sections. The target library for this implementation is a 0.18 μm CMOS process. Because CMOS standard cells are by their very nature inverting, the functional units are implemented using negative logic. These implementations provide the benefits of reduced gate count and improved speed performance.

8.1.1 Ripple Carry Adder

The ripple carry adder used in these designs implement the logic described in Equations 7.1 and 7.2. These equations have been implemented with negative logic and the standard nine-gate ripple carry adder shown in Figure 8.1 is implemented in Verilog structural code. Each of the *n*-bit wide adders is constructed using *n* of these nine gate ripple carry adder modules. The C_{out} output of the n-1^{th} module is connected to the C_{in} input of the n^{th} module. The C_{in} input of the first module is used as the carry-in for the entire adder while the C_{out} output of the last module is used as the carry-out for the entire adder. For a general-purpose processor, an overflow

signal is required to ensure proper operation of the adder. For this particular application, these will be no overflow during the course of the calculations, so it can be eliminated in order to save hardware.

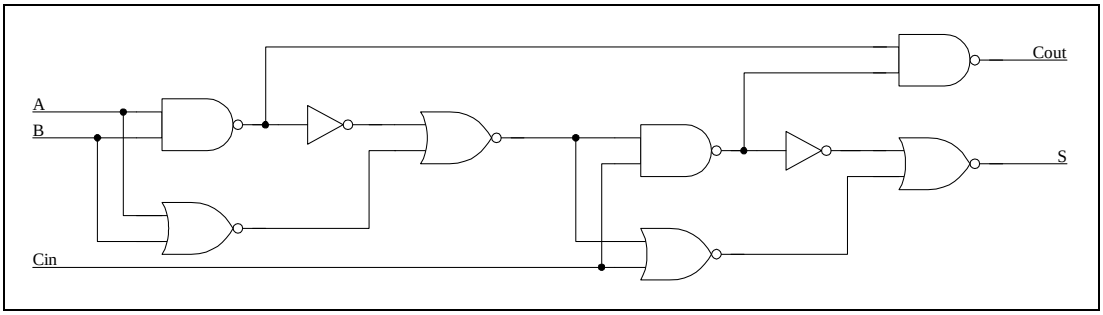


Figure 8.1 – *Single Bit Ripple Carry Adder*

8.1.2 Ripple Carry Subtractor

It is possible to perform subtraction using an adder if the complement of the second operand, which will become the subtrahend, can easily be generated. Forming the complement of a two's complement number is a simple operation. The first step is to invert all of the bits in the two's complement number. The second step is to add a single bit to the least significant bit of the inverted number. This will generate the complement of a two's complement number.

Designing hardware to implement the subtraction function is very simple, especially if a two's complement adder already exists. The bits of the first operand, which is the augend for addition and the minuend for subtraction, are input directly to the adder. The bits of the second operand, which is the addend for addition and the subtrahend for subtraction, along with their inverses, are input into a multiplexer as shown in Figure 8.2. If addition is being performed, the non-inverted bits are selected. If subtraction is being performed, the inverted bits are selected. For addition operations, that is all that is required. For subtraction operations, the additional one's bit must be added to the number. This is handled by applying a one

to the C_{in} input of the ripple carry adder during subtraction operations. This can be implemented using the circuitry show in Figure 8.2 and tying the input to ground.

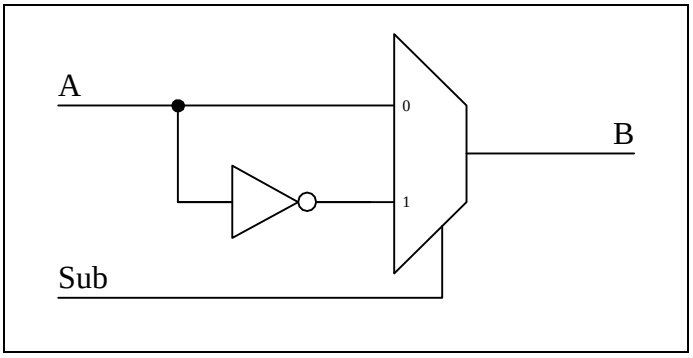


Figure 8.2 – *Single Bit Two's Complement Operand Complement Circuitry*

8.1.3 Carry Lookahead Adder

Because the ripple carry adder is inherently slow, the Classic CORDIC algorithm's performance will be significantly slower than the other implementations. In order to more accurately characterize the performance of the Classic CORDIC algorithm, a fast adder design is also encoded. This provides an upper and a lower bound for the performance of the original algorithm that can be more accurately compared with the newer versions.

The carry lookahead is used as the fast adder design for the second Classic CORDIC implementation. The sum of each carry lookahead adder bit is generated in the same manner as the ripple carry adder. The difference is in the generation of the carry signals. Rather than generating a carry-out signal, a generate and a propagate signal are produced. The generate signal indicates that the two bits will always generate a carry-out. This occurs when $A_n = 1$ and $B_n = 1$. The propagate signal indicates that the two bits will propagate a carry-in signal to the carry-out signal. This occurs when $A_n = 1$ or $B_n = 1$. The formulas for the sum (S_i),

propagate (P_i), generate (G_i) and signals are shown in Equations 8.1, 8.2, and 8.3. These formulas are structurally coded in Verilog using negative logic to implement the nine-gate carry lookahead adder shown in Figure 8.3.

$$S_i = A_i \oplus B_i \oplus C_i \tag{8.1}$$

$$P_i = A_i + B_i \tag{8.2}$$

$$G_i = A_i B_i \tag{8.3}$$

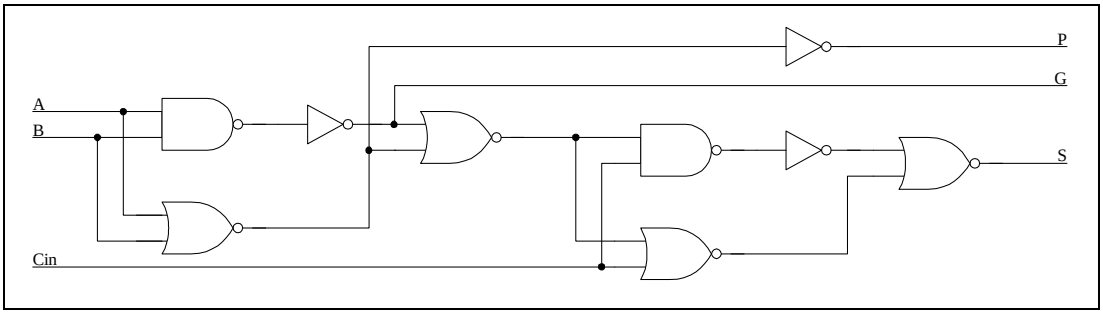


Figure 8.3 – *Single Bit Carry Lookahead Adder*

The first carry, C_1, is asserted if the previous two bits generate a carry or if the previous two bits can propagate a carry and the carry-in is asserted. This is shown in Equation 8.4. The second carry, C_2, is asserted if the previous two bits generate a carry or if the previous two bits can propagate a carry and C_1 is asserted. This is shown in Equation 8.4. The third carry, C_3, is asserted in the same manner and is shown in Equation 8.6.

$$C_1 = G_0 + P_0 C_{in} \tag{8.4}$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_{in} \tag{8.5}$$

$$C_3 = G_2 + P_2 C_2 = G_2 + G_1 P_2 + G_0 P_2 P_1 + P_2 P_1 P_0 C_{in} \tag{8.6}$$

Even though this process can be continued for any number of carries that are required, it is not practical. The addition of each carry requires an additional bit to be combined to calculate the next value. As the number of bits increase, either an extremely wide, and extremely slow, OR gate, or an OR tree, with a large number of gates is required. Rather than implement these wide OR gates, it is possible to create group propagate (G_P) and group generate (G_G) signals. These signals can be combined with other group signals to determine the appropriate carry inputs. The group will propagate if every bit in the group propagates a carry. This is shown in Equation 8.7. The group will generate if a pair of bits in the group generates and the appropriate bit pairs propagate. This is shown in Equation 8.8. The implementation of these equations in negative logic is shown in Figure 8.4.

$$G_P = P_3 P_2 P_1 P_0 \tag{8.5}$$

$$G_G = G_3 + G_2 P_3 + G_1 P_3 P_2 + G_0 P_3 P_2 P_1 \tag{8.6}$$

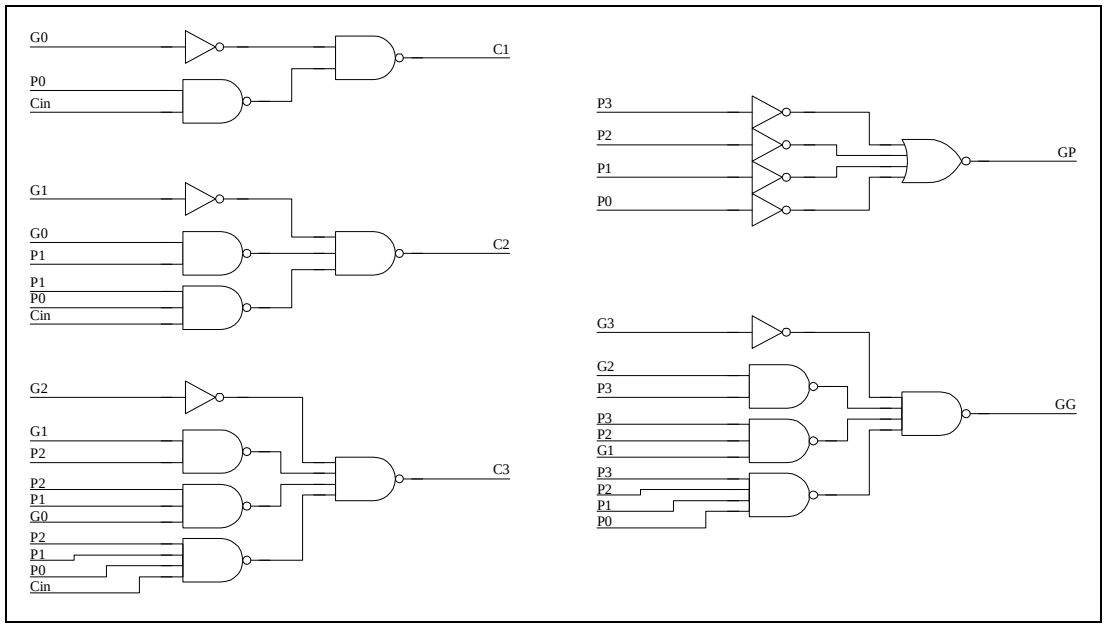


Figure 8.4 – *Four Bit Carry Lookahead Generation Module*

8.1.4 Signed Digit Adder

The signed digit adder used in these designs implement the logic described in Equations 7.3, 7.4, 7.5 and 7.6. These equations have been implemented with negative logic and are shown in Figure 8.5. Each of the *n*-bit wide adders is constructed using *n* of these Verilog adder modules. The M_{i+1} and S_{i+1} outputs of the n-1^{th} module are connected to the M_i and S_i inputs of the n^{th} module. The M_i and S_i inputs of the first module are connected to ground while the M_{i+1} and S_{i+1} outputs of the last module are discarded.

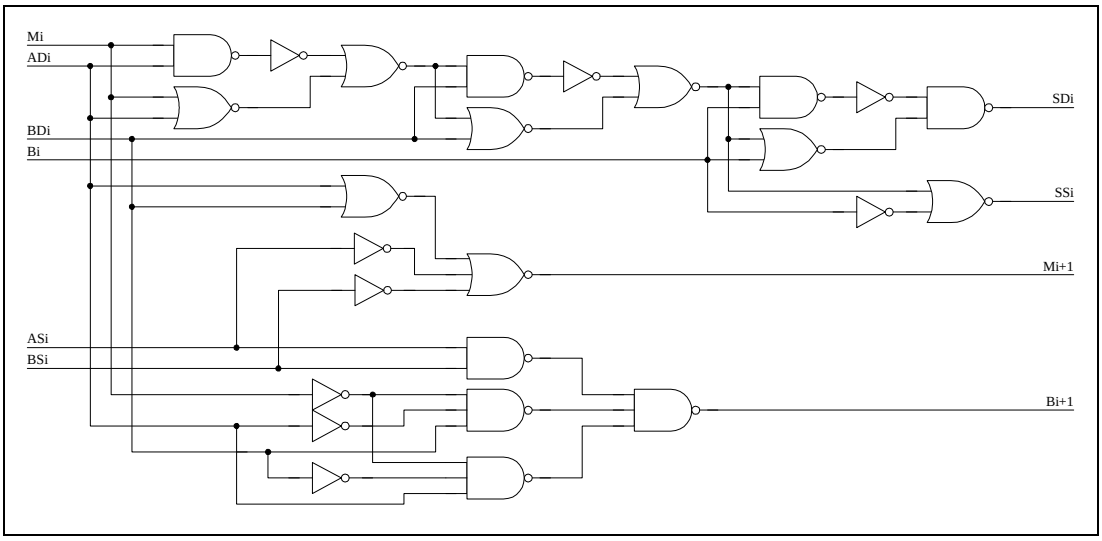


Figure 8.5 – *Single Bit Binary Signed Digit Adder*

8.1.5 Signed Digit Subtractor

The signed digit subtractor can be implemented using the logic functions described in Equations 7.8, 7.9. 7.10, 7.11, and 7.12. Implementing these equations in negative logic produces a subtractor with one more gate than a signed digit adder and an additional gate delay in its critical path. For optimum performance, both modules perform the calculations and the appropriate answer is selected by muxing

the outputs. Even though Chow and Robertson developed these subtraction equations for redundant number representations [60], an examination of the alternatives leads to a more optimal solution.

Because subtraction can be performed using an adder if the complement of the second operand can easily be generated, the difficulty of generating the complement of a BSD number is determined. Because the representation selected in Section 7.1.3.1uses one bit as the sign of the digit and the other as the magnitude, all that is required to form the complement is to change the sign of any digit that has a magnitude equal to one. This quickly and easily generates the complement of a BSD number using this encoding representation. The BSD complementing circuitry coded in structural Verilog is shown in Figure 8.6. Even though this implementation does not improve the speed of the circuit, it eliminates approximately half of the gates required to implement the addition/subtraction module.

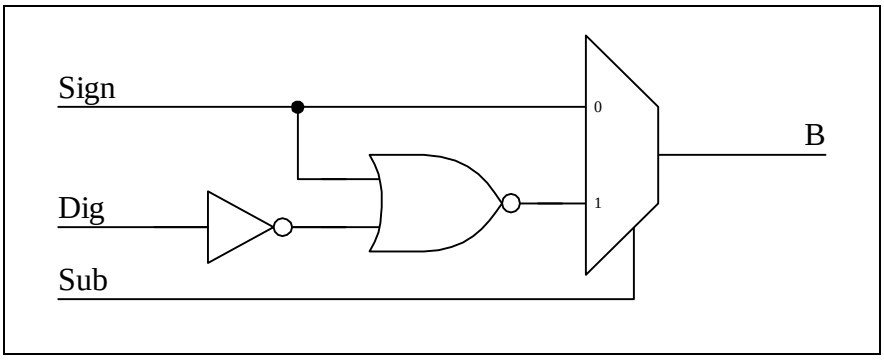
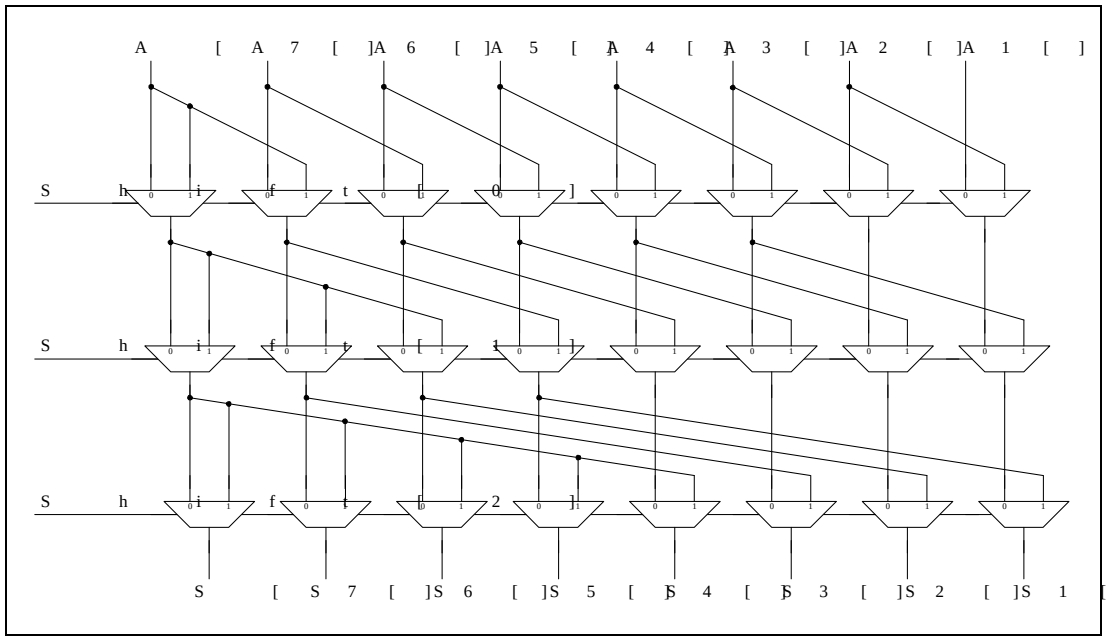


Figure 8.6 – *Single Bit Binary Signed Digit Operand Complement Circuitry*

8.1.6 Shifter

The shifter is a generic shifter implemented using multiplexors as described by Katz [64]. Each row *n* of muxes, including the 0^{th} row, either keeps the operand the same or shifts it by 2^n locations to the right. This corresponds to division or multiplication by a fraction. To achieve *k* shifts, $\lceil \log_2(k+1) \rceil$ rows are required.

Each row will contain the same number of muxes as the bit width of the shifter. Figure 8.7 shows an eight bit wide shifter capable of shifting up to seven bits. This particular shifter is for shifting two's complement numbers because it shifts in copies of the upper bit into the new locations. Shifters for signed digit numbers would have shift zeros into the new locations.



F 8.7 – *An Eight Bit, Seven-Position Shifter* e

8.1.7 Multiplier

The Wallace multiplier was coded in Verilog for this implementation of the Table Look-up CORDIC algorithm. Even though the Wallace multiplier requires more logic, it was selected because it is slightly faster than the Dadda multiplier. Other multiplier implementations were examined. These versions were not used due to the complexity of their design or the need for converting between other number systems to obtain the final answers.

8.2 Algorithm Implementations

The CORDIC algorithms can be implemented using the functional units described in the previous section, behavioral code that handles the control flow, and register arrays that model the ROM tables. Even though only the Hybrid and Table Look-up CORDIC algorithms use ROM tables extensively, every CORDIC algorithm contains a small ROM table to store the arctangent values. These ROM tables are instantiated as top-level components so that they can be removed from the synthesis process.

8.2.1 Ripple Carry Classic CORDIC

The Ripple Carry Adder version of the Classic CORDIC algorithm uses the ripple carry adders described in Section 8.1.1 and the shifters described in Section 8.1.6. The rest of the design is composed of the signal multiplexing to implement the subtraction operation and the control logic to select whether and addition or subtraction is performed for the X, Y, and Z functional units.

The control logic for the ripple carry adder implementation is very simple. A counter is used to keep track of the number of iterations that have been performed. The output of the counter is used to control the shifters and to select the new arctangent value from the ROM table. The sign of the residual angle, sigma, is determined by examining the most significant bit of the two's complement sign variable. If the bit is zero, then the angle is positive. The X functional unit subtracts the shifted Y variable, the Y functional unit adds the shifted X variable, and the Z functional unit subtracts the new arctangent value. If the bit is one, the each of the functional units performs the opposite action.

8.2.2 Carry Look Ahead Classic CORDIC

The Carry Look Ahead Adder version of the Classic CORDIC algorithm uses the carry look ahead adders described in Section 8.1.3 and the shifters described in Section 8.1.6. The design of the control logic, signal multiplexing, and subtraction functionality is the same as the ripple carry adder implementation. The only difference is the use of the faster adder blocks.

8.2.3 Step Branching CORDIC

The Step Branching CORDIC algorithm uses the signed digit adder described in Section 8.1.4, the signed digit subtractor described in Section 8.1.5, and the shifters described in Section 8.1.6. The rest of the design is composed of the control logic that determines the sign of the residual angle from a subset of the bits, the control logic that selects which values should be stored after each iteration of the algorithm, the control logic that selects the shift amount for the X, Y, and Z functional units.

The control logic that determines the sign of the residual angle receives three bits from the Z registers. These values are shifted each time so that the appropriate bits are examined.

A simple counter is used to keep track of the current iteration, index the arctangent ROM table, and control the shift amount to the functional units.

8.2.4 Double Step Branching CORDIC

The Double Step Branching CORDIC algorithm uses the signed digit adder described in Section 8.1.4, the signed digit subtractor described in Section 8.1.5, and the shifters described in Section 8.1.6. The rest of the design is composed of the control logic that determines the sign of the residual angle from a subset of the bits, the control logic that selects which values should be stored after each iteration of the

algorithm, the control logic that selects the shift amount for the X, Y, and Z functional units.

The control logic that determines the sign of the residual angle receives six bits from the Z registers. These values are shifted each time so that the appropriate bits are examined.

8.2.5 Hybrid CORDIC

The Hybrid CORDIC algorithm uses the carry look ahead adder described in Section 8.1.3 and the shifters described in Section 8.1.6. The rest of the design is composed of the adder that calculates the residual angle that determines the sign of each of the following operations, the control logic to implement the subtraction operation, the functional units that implement the Hybrid CORDIC Pipe, a counter to determine how many iterations through the pipe should occur.

The control logic that determines the sign of the residual angle receives three bits from the Z registers. These values are shifted each time so that the appropriate bits are examined.

8.2.6 Table Look-up CORDIC

The Table Look-up CORDIC algorithm uses the carry look ahead adders described in Section 8.1.3, the shifters described in Section 8.1.6, and the multiplier described in Section 8.1.7. The rest of the design is composed of the control logic for indexing the ROM tables, multiplying the appropriate coefficients, and outputting the results.

The control logic that determines the sign of the residual angle receives three bits from the Z registers. These values are shifted each time so that the appropriate bits are examined.

Chapter 9
Back End Tools

To compare the CORDIC algorithms in a real world setting requires the Verilog code to be synthesized into gates, placed and routed, and timed. Performing these steps provides a valid comparison in execution times, latency, and silicon area.

9.1 Synthesis

Synthesis of the Verilog designs was performed using Design Compiler, a part of the Galaxy Design Platform from Synopsys. Design Compiler is synthesis tool capable of providing small and fast logical representations of behavioral code. This tool can address the design goals of timing, testability, power, and datapath optimization while adhering to technology constraints that include hold time, slew rate, and fanout. Design Compiler obtains its results by allowing the user to select trade offs between speed and area.

9.1.1 ROM Tables

Because ROM tables are not synthesizable modules, they are discussed separately from the synthesizable Verilog code. Each of the CORDIC algorithm implementations requires some version of a ROM table. The Classic (ripple carry adder), Classic (carry lookahead adder), and Step Branching CORDIC algorithms require standard ROM tables as described by Volder [24]. The Double Step Branching CORDIC algorithm uses a ROM table that stores the sum and difference of the angles as described by Phatak [51]. The Hybrid CORDIC algorithm only requires ROM tables for approximately the first third of the ATR angles. The Table Look-up CORDIC algorithm requires the different tables as described in this

dissertation. The number of bytes required for the ROMs of each implementation is shown in Table 9.1. As the table shows, the Table Look-up CORDIC algorithm requires more ROM storage than the other implementations. If the area required for these ROM tables is too large, the designer can choose to use more tables. This will exponentially decrease the size of the tables, but will increase the latency for calculating the answers.

Table 9.1 – *ROM Table Requirements for the CORDIC Algorithms*

	Bit Width					
	8	**16**	**24**	**32**	**48**	**64**
Classic (Ripple)	8	32	72	128	288	512
Classic (CLA)	8	32	72	128	288	512
Step	8	32	72	128	288	512
Double Step	8	32	72	128	288	512
Hybrid	4	16	72	48	96	192
TLC	128	65,536	98,688	262,144	589,284	1,048,576

9.1.2 Ripple Carry Adder Classic CORDIC

The ripple carry adder implementation of the CORDIC algorithm is the benchmark against which all the other algorithms are compared. The major components consist of three ripple carry adders, three shifters, three groups of registers, and simple control logic. Because the most of the modules are coded structurally, there is really very little synthesis that will be performed. The only module that is coded behaviorally is the control module. The gate count and

transistor count from the synthesis of the ripple carry adder implementation of the Classic CORDIC algorithm are shown in Table 9.2.

Table 9.2 – *Ripple Carry Classic CORDIC Gate Usage*

	Bit Width					
	8	**16**	**24**	**32**	**48**	**64**
Gate Count	1,156	2,236	3,508	4,588	7,324	9,676
Transistor Count	3,502	6,892	10,894	14,254	22,990	30,382

9.1.3 Carry Lookahead Classic CORDIC

The major components of the carry lookahead adder implementation consist of three carry lookahead carry adders, three shifters, three groups of registers, and simple control logic. Just as with the ripple carry implementation, there is very little synthesis that will be performed. The only module that is coded behaviorally is the control module, which is the same one used in the ripple carry adder implementation. The gate count and transistor count from the synthesis of the carry lookahead carry adder implementation of the Classic CORDIC algorithm are shown in Table 9.3.

Table 9.3 – *Carry Look Ahead Classic CORDIC Gate Usage*

	Bit Width					
	8	**16**	**24**	**32**	**48**	**64**
Gate Count	1,354	2,566	4,102	5,314	8,380	11,062
Transistor Count	4,354	8,266	13,450	17,362	27,502	36,298

9.1.4 Step Branching CORDIC

The major components of the Step Branching implementation consist of six signed digit adders, six shifters, six groups of registers, and moderate control logic. Just as with the other implementations, there is very little synthesis that will be performed. The only module that is coded behaviorally is the control module. The gate count and transistor count from the synthesis of the Step Branching CORDIC algorithm are shown in Table 9.4.

Table 9.4 – *Step Branching CORDIC Gate Usage*

	Bit Width					
	8	**16**	**24**	**32**	**48**	**64**
Gate Count	3,679	6,655	10,015	12,991	20,095	26,431
Transistor Count	11,862	21,556	32,598	42,294	65,718	86,454

9.1.5 Double Step Branching CORDIC

The major components of the Double Step Branching implementation consist of six signed digit adders, six shifters, six groups of registers, and complex control logic. There is no synthesis performed on the arithmetic modules of the Double Step Branching implementation. The only module that is coded behaviorally is the control module, which is highly complex. The control module actually consists of several internal modules: the zeroing module, the rotator module, and the flow control module. These modules are used to simplify the overall design readability and match the description of the original implementation described in [51]. The gate count and transistor count from the synthesis of the Double Step Branching implementation are shown in Table 9.5.

Table 9.5 – *Double Step Branching CORDIC Gate Usage*

	Bit Width					
	8	**16**	**24**	**32**	**48**	**64**
Gate Count	4,205	7,181	10,541	13,517	20,621	26,957
Transistor Count	13,556	23,252	34,292	43,988	67,412	88,148

9.1.6 Hybrid CORDIC

The major components of the Hybrid implementation consist of twelve carry lookahead carry adders, twelve shifters, three groups of registers, and control logic. The only module that is coded behaviorally is the control module, which is a straightforward implementation of the unrolled CORDIC algorithm described in [61]. The gate count and transistor count from the synthesis of the Hybrid CORDIC implementation are shown in Table 9.6.

Table 9.6 – *Hybrid CORDIC Gate Usage*

	Bit Width					
	8	**16**	**24**	**32**	**48**	**64**
Gate Count	4,876	8,932	14,284	18,340	29,020	38,164
Transistor Count	17,238	31,302	50,454	64,516	101,910	133,926

9.1.7 Table Look-up CORDIC

The major components of the Table Look-up implementation consist of two carry lookahead adders, two multipliers, three groups of registers, and control logic. The control module is the only module that is coded behaviorally and follows the

procedures outlined in section 5.3.2. The gate count and transistor count from the synthesis of the Hybrid CORDIC implementation are shown in Table 9.7. The gate counts from the eight and sixteen bit implementations are not applicable with this implementation of the TLC algorithm. Each of these bit widths is implemented as a full look-up table and do not require any arithmetic or control modules.

Table 9.7 – *Table Look-up CORDIC Gate Usage*

	Bit Width					
	8	**16**	**24**	**32**	**48**	**64**
Gate Count	N/A	N/A	20,774	29,550	54,146	87,958
Transistor Count	N/A	N/A	69,134	99,341	184,757	302,941

9.1.8 Synthesis Analysis

The results for four of the six implementations are fairly predictable. The results for the ripple carry and carry lookahead adder implementation of the Classic, along with the Step Branching and Double Step Branching CORDIC algorithms appear fairly linear. The growth in the number of gates, and therefore the number of transistors, for these CORDIC designs is linear with the increase in the bit width. This is derived from the fact that none of the modules used to implement these designs (adders nor shifters) grows exponentially in size with the number of bits.

The results for the Hybrid CORIDC algorithm are misleading if they are not considered carefully. At first glance, the number of gates and transistors appear disproportionate with the Step and Step Branching algorithms. Those two algorithms use the signed digit adder that has a large gate count per bit while the Hybrid algorithm uses smaller carry lookahead adders. The number of gates and

transistors should be large because there are four times as many arithmetic units because the iterative loop has been unrolled.

The Table Look-up also looks significantly larger than the other implementations. Because there are no results for the eight and sixteen implementations, the growth in gate and transistor count almost looks linear. In reality, the TLC algorithm utilizes multipliers, which increase exponentially with an increase in bit width. Because the Wallace multiplier used in this design grows slower than the square in the increase in bits, the growth is not as quickly recognized and therefore, misinterpreted.

9.2 Place and Route

Place and route of the synthesized RTL designs was performed using Astro, a part of the Galaxy Design Platform from Synopsys. Astro is an advanced physical design system for optimization, placement and routing, uses a specialized architecture to concurrently account for physical effects while optimizing the resulting design. This tool integrates physical optimization, extraction, and analysis throughout place and route stages with high efficiency and precise correlation. The place and route was performed with a 0.18 μm, six metal layer technology with 9-track standard cells.

9.2.1 ROM Tables

The results shown in the following sections do not contain area requirements for the ROM tables required by the different algorithms. Although there are tools for developing and auto generating ROMs, none were freely available for use in the development of these implementations. For all of the previously developed CORDIC algorithms, the area required for the ROMs is insignificant when compared to the size of the arithmetic and control blocks. The ROM tables required for the

Table Look-up CORDIC algorithm are fairly large when compared with its arithmetic and control blocks. Although an estimate could be made for the area required, it would not be accurate and is therefore not included. But it is important to note that the areas provided for all of these algorithms do not include the ROM tables and should be evaluated accordingly.

9.2.2 Ripple Carry Adder Classic CORDIC

The ripple carry adder implementation of the Classic CORDIC algorithm is used the benchmark for are performance. The area, along with the speed, of this algorithm allows designers to appropriately evaluate the design tradeoffs between improved performance and silicon area. In addition to providing the required layout area, the density, or coverage area, of the circuit is provided as well. This metric assists in determining how well the place and route algorithm performed. The results of the place and route for the ripple carry adder implementation of the Classic CORDIC algorithm is shown in Table 9.8.

Table 9.8 – *Ripple Carry Classic CORDIC Place and Route Results*

	Bit Width					
	8	**16**	**24**	**32**	**48**	**64**
Area (μm^2)	9,294	21,328	36,295	47,650	81,413	109,792
Density (%)	83	81	80	81	79	78

9.2.3 Carry Lookahead Classic CORDIC

The results of the place and route for the carry lookahead adder implementation of the Classic CORDIC algorithm are shown in Table 9.9.

Table 9.9 – *Carry Lookahead Classic CORDIC Place and Route Results*

	Bit Width					
	8	**16**	**24**	**32**	**48**	**64**
Area (μm^2)	13,548	28,705	49,134	65,609	107,420	144,483
Density (%)	81	79	79	77	76	75

9.2.4 Step Branching CORDIC

The results of the place and route for the Step Branching CORDIC algorithm are shown in Table 9.10.

Table 9.10 – *Step Branching CORDIC Place and Route Results*

	Bit Width					
	8	**16**	**24**	**32**	**48**	**64**
Area (μm^2)	36,334	80,405	133,001	179,404	296,789	401,296
Density (%)	73	70	68	67	65	64

9.2.5 Double Step Branching CORDIC

The results of the place and route for the Double Step Branching CORDIC algorithm are shown in Table 9.11.

9.2.6 Hybrid CORDIC

The results of the place and route for the Hybrid CORDIC algorithm are shown in Table 9.12.

Table 9.11 – *Double Step Branching CORDIC Place and Route Results*

	Bit Width					
	8	**16**	**24**	**32**	**48**	**64**
Area (μm^2)	40,157	85,855	144,162	197,674	322,156	435,950
Density (%)	67	66	63	61	60	59

Table 9.12 – *Hybrid CORDIC Place and Route Results*

	Bit Width					
	8	**16**	**24**	**32**	**48**	**64**
Area (μm^2)	60,918	129,378	228,409	301,596	511,035	700,426
Density (%)	71	69	67	66	63	61

9.2.7 Table Look-up CORDIC

The results of the place and route for the Table Look-up CORDIC algorithm are shown in Table 9.13. It is important to note that the area required to implement the ROM tables for the TLC algorithm are not included in these numbers as discussed at the beginning of this section.

9.2.8 Place and Route Analysis

The silicon area required by these CORDIC algorithm implementations follows the same growth patterns identified in the analysis of their synthesis results. Both of the Classic algorithms, as well as the Step Branching, Double Step Branching, and Hybrid CORDIC algorithms have a linear increase in their area. The

Table Look-up CORDIC algorithm's area increases exponentially with its bit size. These area results are not misleading or difficult to understand.

The density of the designs is disappointing. None of the densities exceeds eighty-five percent. One complex large bit width designs, a low density is understandable. The large amount of interconnect on complex designs can significantly affect the placement of standard cells. But a low density on a small, eight bit design is disappointing and indicates a problem. The two possible sources for this problem are the RTL generated from synthesis and the place and route tool configuration. A better understanding of both tools would lead to a better RTL implementation of a better set of optimization parameters for the place and route tool. In either case, better place and route results could be obtained, but are outside the scope if this dissertation.

Table 9.13 – *Table Look-up CORDIC Place and Route Results*

	Bit Width					
	8	**16**	**24**	**32**	**48**	**64**
Area (μm^2)	N/A	N/A	217,073	371,048	812,935	1,499,641
Density (%)	N/A	N/A	69	67	65	62

9.3 Static Timing Analyses

Static timing analysis was performed using PrimeTime, a part of the Galaxy Design Platform from Synopsys. PrimeTime is a gate level static timing analyzer capable of targeting 100 million gate designs. This tool can analyze multi-frequency designs that combine synthesized logic, embedded memories, and microprocessor cores. PrimeTime obtains its results by exhaustively analyzing all of the logic design's critical paths. Not only is the delay of the worst-case path reported, but the

latency for obtaining the result is provided as well. The latency is determined by taking the worst-case path delay and multiplying it by the number of iterations that must be performed in order to calculate the results.

9.3.1 Ripple Carry Adder Classic CORDIC

The results of the static timing analysis for the ripple carry adder implementation of the Classic CORDIC algorithm are shown in Table 9.14.

Table 9.14 – *Ripple Carry Classic CORDIC Static Timing Results*

	Bit Width					
	8	**16**	**24**	**32**	**48**	**64**
Delay (ns)	3.43	5.55	7.66	9.54	13.54	17.30
Latency (ns)	27.44	88.72	183.84	305.28	649.68	1,106.8

9.3.2 Carry Lookahead Classic CORDIC

The results of the static timing analysis for the carry lookahead adder implementation of the Classic CORDIC algorithm are shown in Table 9.15

Table 9.15 –*Carry Look Ahead Classic CORDIC Static Timing Results*

	Bit Width					
	8	**16**	**24**	**32**	**48**	**64**
Delay (ns)	2.72	3.13	3.84	4.01	4.32	4.42
Latency (ns)	21.78	50.12	92.10	128.40	207.48	283.04

9.3.3 Step Branching CORDIC

The results of the static timing analysis for the Step Branching CORDIC algorithm are shown in Table 9.16.

Table 9.16 – *Step Branching CORDIC Static Timing Results*

	Bit Width					
	8	**16**	**24**	**32**	**48**	**64**
Delay (ns)	2.60	2.85	3.09	3.16	3.34	3.67
Latency (ns)	20.83	45.57	74.22	98.97	160.18	213.57

9.3.4 Double Step Branching CORDIC

The results of the static timing analysis for the Double Step Branching CORDIC algorithm are shown in Table 9.17.

Table 9.17 – *Double Step Branching CORDIC Static Timing Results*

	Bit Width					
	8	**16**	**24**	**32**	**48**	**64**
Delay (ns)	3.21	3.22	3.23	3.25	3.27	3.34
Latency (ns)	38.47	77.26	116.38	156.15	235.70	320.81

9.3.5 Hybrid CORDIC

The results of the static timing analysis for the Hybrid CORDIC algorithm are shown in Table 9.18.

Table 9.18 – *Hybrid CORDIC Static Timing Results*

	Bit Width					
	8	**16**	**24**	**32**	**48**	**64**
Delay (ns)	10.02	11.53	14.12	14.77	15.91	16.28
Latency (ns)	10.02	23.06	56.49	73.83	127.25	162.75

9.3.6 Table Look-up CORDIC

The results of the static timing analysis for the Table Look-up CORDIC algorithm are shown in Table 9.19. It is important to note that the timing information for the eight and sixteen bit TLC implementations is not given. There are two reasons that these numbers are left out. The first is that the eight and sixteen bit implementations do not require any arithmetic operations because they are implemented as pure function tables. For this reason, no code was written that could be statically timed. The second reason is that most local memory accesses can be made very quickly, even for large memories. Because the access time for the ROMs only has to be faster than the operating frequency of the algorithm, it can be made arbitrarily fast, especially for small tables.

Table 9.19 – *Table Look-up CORDIC Static Timing Results*

	Bit Width					
	8	**16**	**24**	**32**	**48**	**64**
Delay (ns)	N/A	N/A	12.36	13.49	14.34	15.26
Latency (ns)	N/A	N/A	12.36	13.49	43.02	76.29

9.3.7 Static Timing Analysis

The first impression of the static timing results is that they are faster than expected for these arithmetic operations. It is important to remember that these timing results do not include extracted parasitic capacitances. The tools for performing extraction from layout were not freely available and therefore not used in calculating these results. Because these parasitic capacitances play a large role in the speed performance of small geometry designs, the exceptional speed of these operations is easily explained.

Many of the timing results appear as expected. The carry lookahead adder implementation is comparable to the ripple carry adder version at low bit widths, but quickly outperforms it as the bit width increases. The delay of the Step branching and Double Step Branching implementations are fast than the carry lookahead adder as should be expected because of their short carry propagation paths.

The difference between the delay times of the Step Branching and Double Step Branching algorithms is unexpected at first. Because the Double Step Branching algorithm performs the shifts at the same time as the additions, the delay time should be approximately half of that of the Step Branching algorithm. Upon further investigation, this difference is due to the more complex control logic required to operate the Double Step Branching implementation. When the multiple iterations are factored into the latency, the Double Step Branching performs slower than its counterpart.

Chapter 10
Conclusion

This dissertation presents the development, derivation, verification, implementation, and evaluation of an improved version of the COordinate Rotation DIgital Computer algorithm for calculating sine and cosine. The Table Look-up CORDIC algorithm utilizes look-up tables and standard microprocessor arithmetic functional units to calculate the results.

The TLC algorithm is shown to be correct through the development of a mathematical proof utilizing the polar form of the CORDIC iteration equations. The TLC algorithm and other versions of the CORDIC algorithm are implemented in MATLAB and simulated. The results of these simulations were compared to verify the new algorithm's operation.

The same CORDIC algorithms are then modeled in Verilog. The Verilog models are then synthesized to gates, routed, and statically timed. The auto place and route of these circuits allowed area estimates to be obtained for the different algorithms. The auto place and route allows silicon areas to be compared while the static timing analysis allows the worst-case path to be timed for frequency comparisons.

10.1 Results

The latency of the Table Look-up CORDIC algorithm is far superior to any of the other CORDIC algorithms examined in this dissertation. The final trigonometric values are available for use in a fraction of the time that it takes the others to produce them. The speed of the Table Look-up implementation can be

improved through pipelining the stages of the multiplier so that a processor implementation of this algorithm can run at faster speeds.

The gate count, transistor count, and area are larger than that required by the other CORDIC algorithms examined in this dissertation. The particular implementation of the TLC algorithm chose performance over area in the design criteria. Due to the flexibility of the TLC algorithm, the gate count, transistor count, and area can be reduced by using more tables in the calculation of the trigonometric values. The reduced area is obtained at the expense of an increased latency for the calculations. Depending on the performance required, this may be an acceptable design versus performance tradeoff.

10.2 Future Research

Even though the CORDIC algorithm has been in active use for over forty-five years, there are still many areas of research that left to be explored. The modeling and implementation of these six variations of the CORDIC algorithm highlighted areas in both the algorithms and the modeling tools that would benefit from a closer inspection of their operation.

The first area of research is into number representation within the MatLab modeling environment. The standard MatLab assignment routines store their results as double precision floating point numbers using the IEEE – 754 format. This limits the accuracy of any calculations performed in MatLab to fifty-three bits of precision due to the size of the mantissa. Even though MatLab does provide the capability for users to develop their own tool extensions, the storage classes cannot be created and modified to the user's need. In order to achieve accurate results for large bit widths, either new modeling tools need to be used or a C++ modeling tool should be developed that allows for user defined accuracy.

Another area of research is the modeling of bit operations within the MatLab environment. Currently no bit manipulation operations are included in the standard

MatLab libraries. In order to model binary operations in a MatLab environment, the user must develop routines that emulate a binary number through the use of an array. Each element of the array represents a single bit in the binary number. Because each -array element is represented by a double precision floating-point number, eight bytes are used to represent each bit in the binary number. This severely limits the size of the arrays that can be used in binary modeling. A simulation tool developed in C++ might provide a better solution and faster execution times for binary modeling.

The Double Step Branching CORDIC algorithm has several areas of research that could significantly improve its performance. A better method for determining the sign of the Binary Signed-Digit residual angle would reduce the complexity of the algorithm and improve performance. Investigation into higher radix implementations might be able to reduce the three additions required per iterations to one addition. This would significantly reduce the computation time and possibly allow more CORDIC iterations to be performed in parallel.

The Hybrid CORDIC algorithm significantly reduced the computation time by selecting a new ATR that allows operations to be performed in parallel without a loss of accuracy. Several implementation options for the Hybrid CORDIC processor have been proposed, and an analysis of their complexity has been performed [] and []. In addition to the analysis on these implementations, the feasibility of using higher radix number systems and combining multiple rotations to reduce the number of stages should be investigated.

The Table Look-up CORDIC algorithm has several areas of interesting research as well. The implementations in this dissertation focus on the calculation of certain trigonometric functions, namely sine and cosine. Calculation of the inverse trigonometric functions, as well as the hyperbolic functions and their inverses should be pursued as well. The possibility of combining functions into unified tables and reducing table sizes is also an important area to investigate.

Appendix A

BitArray.cpp

```
#include "BitArray.h"

using namespace std;


static unsigned char reg_masks[8] = { 0x01, 0x02, 0x04, 0x08, 0x10, 0x20,
   0x40, 0x80 };
static unsigned char inv_masks[8] = { 0xFE, 0xFD, 0xFB, 0xF7, 0xEF, 0xDF,
   0xBF, 0x7F };


KBitArray::KBitArray()  :  m_nElements(0) { }


KBitArray::~KBitArray() { }


void KBitArray::resize(long nElements)
{
   m_nElements = nElements;
   m_arrBytes.resize( (m_nElements + 7) / 8 );
}


unsigned long KBitArray::size()
{
   return m_nElements;
}


void KBitArray::clear()
{
   resize(0);
}


void KBitArray::zero_all()
{
   // make sure there is a vector to zero
   if (m_arrBytes.size() == 0)
      return;

   for (vector<unsigned char>::iterator it = m_arrBytes.begin();
   it != m_arrBytes.end(); it++)
   {
      (*it) = 0;
   }
}
```

```
void KBitArray::one_all()
{
   // make sure there is a vector to zero
   if (m_arrBytes.size() == 0)
      return;

   for (vector<unsigned char>::iterator it = m_arrBytes.begin();
   it != m_arrBytes.end(); it++)
   {
      (*it) = 0xFF;
   }
}


bool KBitArray::GetAt(long nIndex)
{  // [ element 0   ]   [ element 1   ]   [ element 2   ]
   // [   bits0     ]   [   bits1     ]   [   bits2     ]
   // 0 1 2 3 4 5 6 7   0 1 2 3 4 5 6 7   0 1 2 3 4 5 6 7

   unsigned long nElement = nIndex / 8;
   unsigned long nOffset  = nIndex % 8;

   return (m_arrBytes[nElement] & reg_masks[nOffset]) > 0;
}


bool KBitArray::SetAt(long nIndex, int bValue)
{
   // verify it is a valid index
   if (nIndex < 0 || (unsigned long)nIndex >= m_nElements)
   {
      printf( "ERROR : -> Index is less than 0 or greater
than nElements\n" );
      return false;
   }

   // [ element 0   ]   [ element 1   ]   [ element 2   ]
   // [   bits0     ]   [   bits1     ]   [   bits2     ]
   // 0 1 2 3 4 5 6 7   0 1 2 3 4 5 6 7   0 1 2 3 4 5 6 7

   unsigned long nElement = nIndex / 8;
   unsigned long nOffset  = nIndex % 8;

   // different things are in order for setting 1 and 0
   if (bValue)
      m_arrBytes[nElement] |= reg_masks[nOffset];
   else
      m_arrBytes[nElement] &= inv_masks[nOffset];
   return true;
}
```

Appendix B

Iterations.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>

#include "BitArray.h"


int main(int argc, char* argv[])
{
   int    i, j, k,            // Integer variables for loops
          StartIterations,    // Start of the CORDIC Iterations
          StopIterations,     // End of CORDIC Iterations
          NumberOfIterations, // Number of Iterations to perform
          NumberOfTerms,      // Number of Terms in the Equation
          OldVarX,            // Variable expanded from the X Equation
          OldSignX,           // Sign of the variable from the X Equation
          OldSigmaX,          // Sigma associated with the variable
          OldVarY,            // Variable expanded from the Y Equation
          OldSignY,           // Sign of the variable from the X Equation
          OldSigmaY,          // Sigma associated with the variable
          PSize,              // Variable number of terms at the i^th step
          ExpSize,            // Value of the exponent of the current term
          SignBit,            // Sign of the current variable
          XSignBit,           // Sign of the X Equation term
          YSignBit,           // Sign of the Y Equation term
          CarryBit;           // Variable used to calculate rotation signs

   double CalculationTime,    // Time taken to perform calculations
          XX0, XY0, YX0, YY0, // Coefficients for the of X and Y Equations
          Z0,                 // Effective rotation angle
          K;                  // Expansion coefficient

   char   FormFileName[64],   // File name for Symbolic Iteration Formula
          IterFileName[64],   // File name for Standard Iteration Table
          NormFileName[64];   // File name for Normalized Iteration Table

   FILE   *FormFile,          // Symbolic Iteration Formula File pointer
          *IterFile,          // Standard Iteration Table File pointer
          *NormFile;          // Normalized Iteration Table File pointer

   KbitArray           SArray;// Array for holding the rotations signs
   vector<KBitArray>   XArray,// Array for holding the X Equation
                       YArray;// Array for holding the Y Equation

   clock_t BeginCalculations, // Clocks at the start of the calculations
             EndCalculations; // Clocks at the end of the calculations
```

```
printf( "At what CORDIC Iteration do you wish to start? : " );
scanf ( "%d", &StartIterations);

printf( "At what CORDIC Iteration do you wish to stop?  : " );
scanf ( "%d", &StopIterations);

// Start timing here to eliminate input time
BeginCalculations = clock();

sprintf(FormFileName, "IterationEquation%dto%d.txt", StartIterations,
   StopIterations);
sprintf(IterFileName, "IterationTable%dto%d.txt", StartIterations,
   StopIterations);
sprintf(NormFileName, "IterationNormalized%dto%d.txt", StartIterations,
   StopIterations);

FormFile = fopen(FormFileName, "w" );
IterFile = fopen(IterFileName, "w" );
NormFile = fopen(NormFileName, "w" );

if ( (FormFile == NULL) || (IterFile == NULL) || (NormFile == NULL ) )
{
   printf( "\nOUTPUT ERROR:\n" );
   printf( "\tAn error occurred in opening the required output
      files.\n\n" );
   exit(-1);
}

// Determine number of Iterations
NumberOfIterations = StopIterations - StartIterations;

// Allocate space for X Array
XArray.resize(NumberOfIterations + 2);

// Allocate space for Y Array
YArray.resize(NumberOfIterations + 2);

// Determine number of Terms
NumberOfTerms = (int) pow(2, NumberOfIterations);

// Initialize the X Array
for (vector<KBitArray>::iterator xit = XArray.begin();
xit != XArray.end(); xit++)
{
   (*xit).resize(NumberOfTerms);
   (*xit).zero_all();
}

// Initialize the Y Array
for (vector<KBitArray>::iterator yit = YArray.begin();
yit != YArray.end(); yit++)
{
   (*yit).resize(NumberOfTerms);
   (*yit).zero_all();
}
```

```
   // Initialize the Sign Array
   SArray.resize(NumberOfIterations);
   SArray.zero_all();

/*************************************************************************\
*                                                                         *
* The following section of code iterates through the CORDIC equations from*
* StartIterations to StopIterations.  Because each iteration only has     *
* 2^i terms, rather than 2^(StopIterations - StartIterations), PSize is   *
* used to reduce the number of loops through each stage of the iteratiosn *
* equations and minimize calculation time.                                *
*                                                                         *
* During each iteration, the current "X" variables are replaced by an "X" *
* variable with the same "sigma" and "power of 2" modifiers and a "Y"     *
* varaible with those modifiers and an additional "sigma" / "power of 2"  *
* modifier for that iteration.  The sign of the "Y" variable is negated as*
* well.  This corresponds with the the CORDIC equation for X as:          *
*                                                                         *
*      Xi+1 = Xi - Sigma*2^-i*Yi                                          *
*                                                                         *
* During each iteration, the current "Y" variables are replaced by a "Y"  *
* variable with the same "sigma" and "power of 2" modifiers and an "X"    *
* varaible with those modifiers and an additional "sigma" / "power of 2"  *
* modifier for that iteration.  This corresponds with the the CORDIC      *
* equation for Y as:                                                      *
*                                                                         *
*      Yi+1 = Yi + Sigma*2^-i*Xi                                          *
*                                                                         *
* The representation used by the XArray and YArray are shown below.  This *
* is vital to understand in order to understand the code below.  The      *
* number of rows is equal to the number of iterations plus 2.  The first  *
* additional row is for representing whether this is an X or a Y variable.*
* If the bit is set ( == 1), then it is an X variable.  Otherwise it is a *
* Y variable.  Because the arrays are initialized to zeros, all variables *
* are Ys unless they are intentionally set.                               *
*                                                                         *
* The second extra row is for the sign of the term.  If the bit is set, it*
* means taht this term should be subtracted from the equation. (i.e. -1)  *
* Because the arrays are initialized to all zeros, all varaibles are      *
* positive unless they are intentionally set.                             *
*                                                                         *
* The number of columns is equal to the number of terms in the final      *
* equation (i.e. one to represent each of the terms)                      *
*                                                                         *
*                                TERMS                                    *
*                                                                         *
*            0   1   2   3   4   5   6   7                    2^N         *
*          +---+---+---+---+---+---+---+---+-.............-+---+---+     *
*  X / Y   |   |   |   |   |   |   |   |   |               |   |   |     *
*          +---+---+---+---+---+---+---+---+-.............-+---+---+     *
*  Sign    |   |   |   |   |   |   |   |   |               |   |   |     *
*          +---+---+---+---+---+---+---+---+-.............-+---+---+     *
*  Sigma0  |   |   |   |   |   |   |   |   |               |   |   |     *
*          +---+---+---+---+---+---+---+---+-.............-+---+---+     *
*  Sigma1  |   |   |   |   |   |   |   |   |               |   |   |     *
*          +---+---+---+---+---+---+---+---+-.............-+---+---+     *
*          .   .   .   .   .   .   .   .   .               .   .   .     *
```

```
*           .   .   .   .   .   .   .   .   .               .   .   .     *
*           .   .   .   .   .   .   .   .   .               .   .   .     *
*           +---+---+---+---+---+---+---+---+-.............-+---+---+     *
*  SigmaN-1 |   |   |   |   |   |   |   |   |               |   |   |     *
*           +---+---+---+---+---+---+---+---+-.............-+---+---+     *
*  SigmaN   |   |   |   |   |   |   |   |   |               |   |   |     *
*           +---+---+---+---+---+---+---+---+-.............-+---+---+     *
*                                                                         *
\*************************************************************************/

   // Place the appropriate initial representations in the X & Y Arrays
   XArray[0].SetAt(0,0);
   XArray[1].SetAt(0,0);
   YArray[0].SetAt(0,1);
   YArray[1].SetAt(0,0);

   for ( i = 0 ; i < (StopIterations - StartIterations) ; i++ )
   {
      // The number of terms for this iteration step
      PSize = (int) pow(2,i);

      for ( j = 0 ; j < PSize ; j++)
      {
         // Determine if the new term is an X or Y
         OldVarX  = XArray[0].GetAt(j);
         if (OldVarX == 0)                      // If it was a Y....
         {
            XArray[0].SetAt(j + PSize, 1);      // Intentionally set an X
         }

         OldVarY  = YArray[0].GetAt(j );
         if (OldVarY == 0)                      // If it was a Y
         {
            YArray[0].SetAt(j + PSize, 1);      // Intentionally set an X
         }

         //Determine if the new term is a + or -
         OldSignX = XArray[1].GetAt(j);
         if ((OldVarX == 0) && (OldSignX == 0))
         {
            XArray[1].SetAt(j + PSize, 1);
         }
         if ((OldVarX == 1) && (OldSignX == 1))
         {
            XArray[1].SetAt(j + PSize, 1);
         }

         OldSignY = YArray[1].GetAt(j);
         if ((OldVarY == 0) && (OldSignY == 0))
         {
            YArray[1].SetAt(j + PSize, 1);
         }
         if ((OldVarY == 1) && (OldSignY == 1))
         {
            YArray[1].SetAt(j + PSize, 1);
         }
```

```
      // Copy the old sigma values into the new term
      for ( k = NumberOfIterations + 1 ; k > NumberOfIterations + 1 – I
         ; k-- )
      {
         OldSigmaX = XArray[k].GetAt(j);
         if (OldSigmaX == 1)
         {
            XArray[k].SetAt(j + PSize, 1);
         }

         OldSigmaY = YArray[k].GetAt(j);
         if (OldSigmaY == 1)
         {
            YArray[k].SetAt(j + PSize, 1);
         }
      }

      //Add a sigma for the new term
      XArray[k].SetAt(j + PSize, 1);
      YArray[k].SetAt(j + PSize, 1);
   }
}

// Print out the X terms of the X equation
fprintf(FormFile, "X%d = X%d * ( ", StopIterations, StartIterations );
for ( i = 0 ; i < NumberOfTerms ; i++ )
{
   if ( XArray[0].GetAt(i) == 0 )
   {
      if ( XArray[1].GetAt(i) == 0 )
         fprintf(FormFile, "+ " );
      else
         fprintf(FormFile, "- " );

      if ( i == 0)
         fprintf(FormFile, "1 " );
      else
      {
         for ( j = 0 ; j < NumberOfIterations ; j++ )
         {
            if ( XArray[NumberOfIterations + 1 - j].GetAt(i) == 1 )
               fprintf(FormFile, "s%d", StopIterations - j - 1 );
         }
         fprintf(FormFile, "*" );
         ExpSize = 0;
         for ( j = 0 ; j < NumberOfIterations ; j++ )
         {
            if ( XArray[NumberOfIterations + 1 - j].GetAt(i) == 1 )
               ExpSize = ExpSize + StopIterations - j - 1;
         }
         fprintf(FormFile, "2(-%d) ", ExpSize );
      }
   }
}

fprintf(FormFile, " ) +\n" );
```

```
// Print out the Y terms of the X equation
fprintf(FormFile, "     Y%d * ( ", StartIterations);
for ( i = 0 ; i < NumberOfTerms ; i++ )
{
   if ( NumberOfTerms == 1 )
      fprintf(FormFile, "0 " );

   if ( XArray[0].GetAt(i) == 1 )
   {
      if ( XArray[1].GetAt(i) == 0 )
         fprintf(FormFile, "+ " );
      else
         fprintf(FormFile, "- " );

      for ( j = 0 ; j < NumberOfIterations ; j++ )
      {
         if ( XArray[NumberOfIterations + 1 - j].GetAt(i) == 1 )
            fprintf(FormFile, "s%d", StopIterations - j - 1 );
      }
      fprintf(FormFile, "*" );
      ExpSize = 0;
      for ( j = 0 ; j < NumberOfIterations ; j++ )
      {
         if ( XArray[NumberOfIterations + 1 - j].GetAt(i) == 1 )
            ExpSize = ExpSize + StopIterations - j - 1;
      }
      fprintf(FormFile, "2(-%d) ", ExpSize);
   }
}
fprintf(FormFile, " )\n\n\n" );

// Print out the X terms of the Y equation
fprintf(FormFile, "Y%d = X%d * ( ", StopIterations, StartIterations );
for ( i = 0 ; i < NumberOfTerms ; i++ )
{
   if ( NumberOfTerms == 1 )
      fprintf(FormFile, "0 " );

   if ( YArray[0].GetAt(i) == 0 )
   {
      if ( YArray[1].GetAt(i) == 0 )
         fprintf(FormFile, "+ " );
      else
         fprintf(FormFile, "- " );

      for ( j = 0 ; j < NumberOfIterations ; j++ )
      {
         if ( YArray[NumberOfIterations + 1 - j].GetAt(i) == 1 )
            fprintf(FormFile, "s%d", StopIterations - j - 1 );
      }
      fprintf(FormFile, "*" );
      ExpSize = 0;
      for ( j = 0 ; j < NumberOfIterations ; j++ )
      {
         if ( YArray[NumberOfIterations + 1 - j].GetAt(i) == 1 )
            ExpSize = ExpSize + StopIterations - j - 1;
```

```
      }
      fprintf(FormFile, "2(-%d) ", ExpSize);
   }
}
fprintf(FormFile, " ) +\n");

// Print out the Y terms of the Y equation
fprintf(FormFile, "     Y%d * ( ", StartIterations );
for ( i = 0 ; i < NumberOfTerms ; i++ )
{
   if ( YArray[0].GetAt(i) == 1 )
   {
      if ( YArray[1].GetAt(i) == 0 )
         fprintf(FormFile, "+ ");
      else
         fprintf(FormFile, "- ");

      if ( i == 0 )
         fprintf(FormFile, "1 ");
      else
      {
         for ( j = 0 ; j < NumberOfIterations ; j++ )
         {
            if ( YArray[NumberOfIterations + 1 - j].GetAt(i) == 1 )
               fprintf(FormFile, "s%d", StopIterations - j - 1 );
         }
         fprintf(FormFile, "*" );
         ExpSize = 0;
         for ( j = 0 ; j < NumberOfIterations ; j++ )
         {
            if ( YArray[NumberOfIterations + 1 - j].GetAt(i) == 1 )
               ExpSize = ExpSize + StopIterations - j - 1;
         }
         fprintf(FormFile, "2(-%d) ", ExpSize);
      }
   }
}
fprintf(FormFile, " )\n\n\n");

fclose(FormFile);

// Calculate the expansion factor for outputting the Normalized Tables
K = 1.0;
for ( i = StartIterations ; i < StopIterations ; i++ )
{
   K = K * sqrt(1.0 + pow(2,-2*i) );
}

for ( i = 0 ; i < NumberOfTerms ; i++ )
{
   XX0 = 0.0;
   XY0 = 0.0;
   YX0 = 0.0;
   YY0 = 0.0;
   Z0  = 0.0;
```

```
// Calculate Coefficient Values for each combination of Sigma values
for ( j = 0 ; j < NumberOfTerms ; j++ )
{
   SignBit = 1;
   ExpSize = 0;
   for ( k = 0 ; k < NumberOfIterations ; k++ )
   {
      if ( XArray[NumberOfIterations + 1 - k].GetAt(j) == 1 )
      {
         ExpSize = ExpSize + StopIterations - k - 1;
         if ( SArray.GetAt(k) )
            SignBit *= -1;
      }
   }

   if ( XArray[1].GetAt(j) )
      XSignBit = -1;
   else
      XSignBit = 1;

   if ( YArray[1].GetAt(j) )
      YSignBit = -1;
   else
      YSignBit = 1;

   if ( XArray[0].GetAt(j) )
      XY0 = XY0 + XSignBit*SignBit*pow(2,-1*ExpSize);
   else
      XX0 = XX0 + XSignBit*SignBit*pow(2,-1*ExpSize);

   if ( YArray[0].GetAt(j) )
      YY0 = YY0 + YSignBit*SignBit*pow(2,-1*ExpSize);
   else
      YX0 = YX0 + YSignBit*SignBit*pow(2,-1*ExpSize);
}

// Calculate the effective rotation angle
for ( j = 0 ; j < NumberOfIterations ; j++ )
{
   if ( SArray.GetAt(NumberOfIterations - 1 - j) )
      Z0 = Z0 - atan(pow(2,-1*(j + StartIterations)));
   else
      Z0 = Z0 + atan(pow(2,-1*(j + StartIterations)));
}

Z0 = Z0 * 57.295779513082320876798154814105;

// Output the values to the Iterations Table and the Normalized Table
fprintf(IterFile, "%d\t %26.22f\t %26.22f\t %26.22f\t %26.22f\t
%26.22f\n", i, XX0,   XY0,   YX0,   YY0,   Z0);

fprintf(NormFile, "%d\t %26.22f\t %26.22f\t %26.22f\t %26.22f\t
%26.22f\n", i, XX0/K, XY0/K, YX0/K, YY0/K, Z0);

// Calculate the next series of sign values for all of the Sigmas
CarryBit = 1;
for ( j = 0 ; j < NumberOfIterations ; j++ )
```

```
        {
            if ( CarryBit )
            {
                if ( SArray.GetAt(j) )
                    SArray.SetAt(j,0);
                else
                {
                    SArray.SetAt(j,1);
                    CarryBit = 0;
                }
            }
        }
    }

    fclose(IterFile);
    fclose(NormFile);

    EndCalculations = clock();

    CalculationTime = (double)(EndCalculations - BeginCalculations)/CLK_TCK;

    printf( "The elapsed calculation time was %f seconds.\n",
    CalculationTime);

    return (1);
}
```

Appendix C

Merger.cpp

```
#include "stdio.h"
#include "math.h"
#include <vector>

using namespace std;


int main(int argc, char* argv[])
{
   int              i, j, k,         // Loop Variables
                    NumberOfTables,  // Number of tables to combine
                    Table1Entries,   // Number of entries in 1st table
                    Table2Entries,   // Number of entries in 2nd table
                    Iteration1,      // Iteration number from 1st table
                    Iteration2;      // Iteration number from 2nd table


   float            IXX, IXY,        // Coefficients for the XX, XY, YX,
                    IYX, IYY,        //      and YY values read in from
                    IZ,              //      the 1st iteration table
                    NXX, NXY,        // Coefficients for the XX, XY, YX,
                    NYX, NYY,        //      and YY values read in from
                    NZ;              //      the 1st normalized table

   vector<int>      StartIteration,  // Vector of table start iterations
                    StopIteration;   // Vector of table end iterations


   vector<float>    IXX0, IXY0,      // Vector of coefficients for the
                    IYX0, IYY0,      //      XX, XY, YX, and YY values
                    IZ0,             //      read from 2nd iteration table
                    NXX0, NXY0,      // Vector of coefficients for the
                    NYX0, NYY0,      //      XX, XY, YX, and YY values
                    NZ0;             //      read from 2nd normalized table

   char   InputTable1IterName[64];  // Name of the 1st Iteration Table
   char   InputTable1NormName[64];  // Name of the 1st Normalized Table
   char   InputTable2IterName[64];  // Name of the 2nd Iteration Table
   char   InputTable2NormName[64];  // Name of the 2nd Normalized Table
   char   OutputTableIterName[64];  // Name of the merged Iteration Table
   char   OutputTableNormName[64];  // Name of the merged Normalized Table

   FILE   *InTable1Iter;            // Pointer to the 1st Iteration Table
   FILE   *InTable1Norm;            // Pointer to the 1st Normalized Table
   FILE   *InTable2Iter;            // Pointer to the 2nd Iteration Table
   FILE   *InTable2Norm;            // Pointer to the 2nd Normalized Table
   FILE   *OutTableIter;            // Pointer to merged Iteration Table
   FILE   *OutTableNorm;            // Pointer to merged Normalized Table
```

```
   // Variable initialization to eliminate MS warnings
   Iteration1 = 0;
   Iteration2 = 0;
   IXX        = 0;
   IXY        = 0;
   IYX        = 0;
   IYY        = 0;
   IZ         = 0;
   NXX        = 0;
   NXY        = 0;
   NYX        = 0;
   NYY        = 0;
   NZ         = 0;

   // Determine how many tables to merge
   printf( "How many CORDIC tables do you want to combine: " );
   scanf ( "%d", &NumberOfTables);
   printf( "\n" );

   StartIteration.resize(NumberOfTables);
   StopIteration.resize(NumberOfTables);

/*************************************************************************\
*                                                                         *
*  Determine the start and stop iterations for each table to be merged.   *
*  In order for the table to merge properly, the start iteration of the   *
*  current table must be the same as the end iteration of the previous    *
*  table.  For that reason, the user is not allowed to input the start    *
*  iterations for the 2nd or higher tables.                               *
*                                                                         *
\*************************************************************************/
   for ( i = 0 ; i < NumberOfTables ; i++ )
   {
      if ( i == 0 )
      {
         printf( "Table 1 starts at iteration: " );
         scanf ( "%d", &StartIteration[i] );
      }
      else
      {
         StartIteration[i] = StopIteration[i-1];
         printf( "Table %d starts at iteration: %d\n", i+1,
            StartIteration[i]);
      }
      printf( "        & ends at iteration: " );
      scanf ( "%d", &StopIteration[i] );
      printf( "\n" );
   }

/*************************************************************************\
*                                                                         *
*  Determine the name of the first table to open.  If this is the second  *
*  time through the loop, then an intermediate output table has been      *
*  generated and needs to be opened.  The standard naming convention is   *
*  used to generate the intermediate table with a prefix of "Gen".        *
*                                                                         *
\*************************************************************************/
```

```
for ( i = 1 ; i < NumberOfTables ; i++ )
{
   if ( i == 1 )
   {
      sprintf(InputTable1IterName, "IterationTable%dto%d.txt",
                  StartIteration[0], StopIteration[0]);
      sprintf(InputTable1NormName, "IterationNormalized%dto%d.txt",
                  StartIteration[0], StopIteration[0]);
   }
   else
   {
      sprintf(InputTable1IterName, "GenIterationTable%dto%d.txt",
                  StartIteration[0], StopIteration[i-1]);
      sprintf(InputTable1NormName, "GenIterationNormalized%dto%d.txt",
                  StartIteration[0], StopIteration[i-1]);
   }

   // Determine the name of the 2nd table to open.
   sprintf(InputTable2IterName, "IterationTable%dto%d.txt",
               StartIteration[i], StopIteration[i]);
   sprintf(InputTable2NormName, "IterationNormalized%dto%d.txt",
               StartIteration[i], StopIteration[i]);

   // Determine the name of the output table.
   sprintf(OutputTableIterName, "GenIterationTable%dto%d.txt",
                StartIteration[0], StopIteration[i]);
   sprintf(OutputTableNormName, "GenIterationNormalized%dto%d.txt",
                StartIteration[0], StopIteration[i]);

   // Open current tables.
   InTable1Iter = fopen(InputTable1IterName, "r" );
   InTable1Norm = fopen(InputTable1NormName, "r" );

   InTable2Iter = fopen(InputTable2IterName, "r" );
   InTable2Norm = fopen(InputTable2NormName, "r" );

   OutTableIter = fopen(OutputTableIterName, "w" );
   OutTableNorm = fopen(OutputTableNormName, "w" );

   // Exit program if unable to read from tables or output to the
   //      merged tables for intermiate results.
   if ( (InTable1Iter == NULL) || (InTable1Norm == NULL) ||
        (InTable2Iter == NULL) || (InTable2Norm == NULL) ||
        (OutTableIter == NULL) || (OutTableNorm == NULL) )
   {
      printf( "\nOUTPUT ERROR:\n" );
      printf( "\tAn error occurred in opening the required
         I/O files.\n\n" );
      exit(-1);
   }

   // Calculate the number of entries per table
   Table1Entries = (int) pow(2, (StopIteration[i-1] –
      StartIteration[0]));
   Table2Entries = (int) pow(2, (StopIteration[i] –
      StartIteration[i]));
```

```
/*************************************************************************\
*                                                                         *
*  Size the vectors to hold the correct number of values from the tables. *
*  The second table is always read into memory.  This is done for two     *
*  reasons.  The first is the this order will generate the outputs in the *
*  same order as the original program would if it were run for a larger   *
*  number of terms.  This keeps the output format the same as what would  *
*  be expected.  The second reason is that the intermediate tables could  *
*  get significantly large as they are merged.  In order to reduce system *
*  memory usage, the smaller tables has all of its values read into the   *
*  arrays.  It is possible not to store any of the values in memory and   *
*  run completely from the hard drive, but this would increase the run    *
*  time and produce unacceptable delays.                                  *
*                                                                         *
\*************************************************************************/

      IXX0.resize(Table2Entries);
      IXY0.resize(Table2Entries);
      IYX0.resize(Table2Entries);
      IYY0.resize(Table2Entries);
      IZ0.resize(Table2Entries);

      NXX0.resize(Table2Entries);
      NXY0.resize(Table2Entries);
      NYX0.resize(Table2Entries);
      NYY0.resize(Table2Entries);
      NZ0.resize(Table2Entries);

      // Initialize the vectors to eliminate warnings from MS
       for ( j = 0 ; j < Table2Entries ; j++ )
      {
         IXX0[j] = 0;
         IXY0[j] = 0;
         IYX0[j] = 0;
         IYY0[j] = 0;
         IZ0[j]  = 0;
         NXX0[j] = 0;
         NXY0[j] = 0;
         NYX0[j] = 0;
         NYY0[j] = 0;
         NZ0[j]  = 0;
      }

      // Read in the values from the Iteration and Normalized Tables
      for ( j = 0 ; j < Table2Entries ; j++ )
      {
         fscanf(InTable2Iter, "%d%f%f%f%f%f", &Iteration2, &IXX0[j],
            &IXY0[j], &IYX0[j], &IYY0[j], &IZ0[j]);
         fscanf(InTable2Norm, "%d%f%f%f%f%f", &Iteration2, &NXX0[j],
            &NXY0[j], &NYX0[j], &NYY0[j], &NZ0[j]);
      }

      fclose(InTable2Iter);
      fclose(InTable2Norm);
```

```
        // Combine the results from the two tables
        for ( j = 0 ; j < Table1Entries ; j++ )
        {
            fscanf(InTable1Iter, "%d%f%f%f%f%f\n", &Iteration1, &IXX, &IXY,
                &IYX, &IYY, &IZ);

            fscanf(InTable1Norm, "%d%f%f%f%f%f\n", &Iteration1, &NXX, &NXY,
                &NYX, &NYY, &NZ);

            for ( k = 0 ; k < Table2Entries ; k++ )
            {

                fprintf(OutTableIter, "%d\t %26.22f\t %26.22f\t %26.22f\t
                    %26.22f\t %26.22f\n", Iteration1*Table2Entries+k,
                    IXX*IXX0[k] + IXY*IYX0[k], IXX*IXY0[k] + IXY*IYY0[k],
                    IYX*IXX0[k] + IYY*IYX0[k], IYX*IXY0[k] + IYY*IYY0[k],
                    IZ + IZ0[k] );

                fprintf(OutTableNorm, "%d\t %26.22f\t %26.22f\t %26.22f\t
                    %26.22f\t %26.22f\n", Iteration1*Table2Entries+k,
                    NXX*NXX0[k] + NXY*NYX0[k], NXX*NXY0[k] + NXY*NYY0[k],
                    NYX*NXX0[k] + NYY*NYX0[k], NYX*NXY0[k] + NYY*NYY0[k],
                    NZ + NZ0[k] );
            }
        }

        fclose(InTable1Iter);
        fclose(InTable1Norm);
        fclose(OutTableIter);
        fclose(OutTableNorm);

    }

    return(1);

}
```

Appendix D

Checker.cpp

```
#include "stdio.h"
#include "math.h"

int main(int argc, char* argv[])
{
   int    i,                     // Loop variable
          StartIteration,        // Start Iteration of the table to check
          StopIteration,         // Stop Iteration of the table to check
          TableEntries,          // Number of entries in the table
          Bits,                  // Number of bits the final answer has
          IterationI,            // Iteration number for Iteration Table
          IterationN;            // Iteration number for Normalized Table

   float  IXX, IXY,              // Coefficients for the Iteration Table
          IYX, IYY,
          IZ,
          NXX, NXY,              // Coefficients for the Normalized Table
          NYX, NYY,
          NZ;


   double IXXd, IXYd,            // Difference between the coefficient read
          IYXd, IYYd,            //    and the coefficient calculated
          NXXd, NXYd,            // Difference between the coefficient read
          NYXd, NYYd,            //    and the coefficient calculated
          K, Error,              // Expansion factor, Error Limit
          Cosine, Sine;          // Computed Cosine and Sine values

   char   IterFileName[64],      // File name for Iteration Table
          NormFileName[64];      // File name for Normalized Table

   FILE   *IterFile,             // File pointer for Iteration Table
          *NormFile;             // File pointer for Normalized Table

   // Initialization of variables to eliminate warnings from MS
   StartIteration = 0;
   StopIteration  = 0;

   printf( "This program will verify the contents of Generated
      Tables.\n\n" );

   // Determine which table to examine by getting start and stop iterations
   printf( "The Generated Table starts at iteration: " );
   scanf ( "%d", &StartIteration);

   printf( "                 and stops at iteration: " );
   scanf ( "%d", &StopIteration);

   printf( "\n" );
```

```
   printf( "How many binary bits are being calculated: " );
   scanf ( "%d", &Bits);

   // Determine the names of the files to open for verification
   sprintf(IterFileName, "GenIterationTable%dto%d.txt",
      StartIteration, StopIteration);
   sprintf(NormFileName, "GenIterationNormalized%dto%d.txt",
      StartIteration, StopIteration);

   // Open the files for verification
   IterFile = fopen(IterFileName, "r" );
   NormFile = fopen(NormFileName, "r" );

   if ( (IterFile == NULL) | (NormFile == NULL ) )
   {
      printf( "\nOUTPUT ERROR:\n" );
      printf( "\tAn error occurred in opening the required input
         files.\n\n" );
      exit(-1);
   }

   // Determine the number of entries
   TableEntries = (int) pow(2, (StopIteration - StartIteration));

   Error = pow(2, (-1 - Bits) );

   // Calculate the expansion factor for comparing Iteration values
   K = 1.0;
   for ( i = StartIteration ; i < StopIteration ; i++ )
   {
      K = K * sqrt(1.0 + pow(2,-2*i) );
   }

/*************************************************************************\
*                                                                         *
*  Read in a line from the iteration table and the normalized table.      *
*  Then calculate the correct coefficient values from the rotation angle. *
*  Find the difference between the coefficients from the table and those  *
*  from the calculation.  Print out an error for any values that exceed   *
*  the error limit needed for the merged table.                           *
*                                                                         *
\*************************************************************************/
   for ( i = 0 ; i < TableEntries ; i++ )
   {
      fscanf(IterFile, "%d%f%f%f%f%f", &IterationI, &IXX, &IXY, &IYX,
         &IYY, &IZ);

      fscanf(NormFile, "%d%f%f%f%f%f", &IterationN, &NXX, &NXY, &NYX,
         &NYY, &NZ);

      Cosine = cos(NZ/57.295779513082320876798154814105);
      Sine   = sin(NZ/57.295779513082320876798154814105);

      IXXd = IXX - Cosine * K;
      IXYd = IXY + Sine   * K;
      IYXd = IYX - Sine   * K;
      IYYd = IYY - Cosine * K;
```

```
      NXXd = NXX - Cosine;
      NXYd = NXY + Sine;
      NYXd = NYX - Sine;
      NYYd = NYY - Cosine;

      if ( ( IXXd > Error ) || ( IXYd > Error ) ||
           ( IYXd > Error ) || ( IYYd > Error) )
      {
         printf( "%d\t%f\t%f\t%f\t%f\t%f\n", IterationI, IXXd, IXYd,
            IYXd, IYYd, IZ);
      }

      if ( ( NXXd > Error ) || ( NXYd > Error ) ||
           ( NYXd > Error ) || ( NYYd > Error) )
      {
         printf( "%d\t%f\t%f\t%f\t%f\t%f\n", IterationN, NXXd, NXYd,
            NYXd, NYYd, IZ);
      }
   }

   fclose(IterFile);
   fclose(NormFile);

   return(1);

}
```

Appendix E

FastIterations.cpp

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "BitArray.h"


int main(int argc, char* argv[])
{
   int    i, j, k,            // Integer variables for loops
          index,              // Additional loop variable
          StartIterations,    // Start of the CORDIC Iterations
          StopIterations,     // End of CORDIC Iterations
          NumberOfIterations, // Number of Iterations to perform
          NumberOfTerms,      // Number of Terms in the Equation
          OldVarX,            // Variable expanded from the X Equation
          OldSignX,           // Sign of the variable from the X Equation
          OldSigmaX,          // Sigma associated with the variable
          OldVarY,            // Variable expanded from the Y Equation
          OldSignY,           // Sign of the variable from the X Equation
          OldSigmaY,          // Sigma associated with the variable
          PSize,              // Variable number of terms at the i^th step
          ExpSize;            // Value of the exponent of the current term

   double ArcTangentRadix,    // Value of atan(2^-i) for each iteration
          CalculationTime,    // Time taken to perform calculations
          XX0, YX0,           // Coefficients for the of X and Y Equations
          Z0,                 // Effective rotation angle
          K;                  // Expansion coefficient

   char   FormFileName[64],   // File name for Symbolic Iteration Formula
          IterFileName[64],   // File name for Standard Iteration Table
          NormFileName[64];   // File name for Normalized Iteration Table

   FILE   *FormFile,          // Symbolic Iteration Formula File pointer
          *IterFile,          // Standard Iteration Table File pointer
          *NormFile;          // Normalized Iteration Table File pointer

   vector<double> Z_Partition;// Vector to hold the CORDIC rotation angles

   vector<KBitArray>   XArray,// Array for holding the X Equation
                       YArray;// Array for holding the Y Equation

   clock_t BeginCalculations, // Clocks at the start of the calculations
           EndCalculations;   // Clocks at the end of the calculations
```

```
printf( "At what CORDIC Iteration do you wish to start? : " );
scanf ( "%d", &StartIterations);

printf( "At what CORDIC Iteration do you wish to stop?  : " );
scanf ( "%d", &StopIterations);

// Start timing here to eliminate input time
BeginCalculations = clock();

sprintf(FormFileName, "FastIterationEquation%dto%d.txt",
   StartIterations, StopIterations);
sprintf(IterFileName, "FastIterationTable%dto%d.txt",
   StartIterations, StopIterations);
sprintf(NormFileName, "FastIterationNormalized%dto%d.txt",
   StartIterations, StopIterations);

FormFile = fopen(FormFileName, "w" );
IterFile = fopen(IterFileName, "w" );
NormFile = fopen(NormFileName, "w" );

if ( (FormFile == NULL) || (IterFile == NULL) || (NormFile == NULL ) )
{
   printf( "\nOUTPUT ERROR:\n" );
   printf( "\tAn error occurred in opening the output files.\n\n" );
   exit(-1);
}

// Determine number of Iterations
NumberOfIterations = StopIterations - StartIterations;

// Allocate space for X Array
XArray.resize(NumberOfIterations + 2);

// Allocate space for Y Array
YArray.resize(NumberOfIterations + 2);

// Determine number of Terms
NumberOfTerms = (int) pow(2, NumberOfIterations);

// Set aside storage for the effective rotation angles
Z_Partition.resize(NumberOfTerms);

// Initialize the X Array
for (vector<KBitArray>::iterator xit = XArray.begin();
xit != XArray.end(); xit++)
{
   (*xit).resize(NumberOfTerms);
   (*xit).zero_all();
}

// Initialize the Y Array
for (vector<KBitArray>::iterator yit = YArray.begin();
yit != YArray.end(); yit++)
{
   (*yit).resize(NumberOfTerms);
   (*yit).zero_all();
}
```

```
/*************************************************************************\
*                                                                         *
* The following section of code iterates through the CORDIC equations from*
* StartIterations to StopIterations.  Because each iteration only has     *
* 2^i terms, rather than 2^(StopIterations - StartIterations), PSize is   *
* used to reduce the number of loops through each stage of the iteratiosn *
* equations and minimize calculation time.                                *
*                                                                         *
* During each iteration, the current "X" variables are replaced by an "X" *
* variable with the same "sigma" and "power of 2" modifiers and a "Y"     *
* variable with those modifiers and an additional "sigma" / "power of 2"  *
* modifier for that iteration.  The sign of the "Y" variable is negated as*
* well.  This corresponds with the CORDIC equation for X as:              *
*                                                                         *
*      Xi+1 = Xi - Sigma*2^-i*Yi                                          *
*                                                                         *
* During each iteration, the current "Y" variables are replaced by a "Y"  *
* variable with the same "sigma" and "power of 2" modifiers and an "X"    *
* variable with those modifiers and an additional "sigma" / "power of 2"  *
* modifier for that iteration.  This corresponds with the CORDIC          *
* equation for Y as:                                                      *
*                                                                         *
*      Yi+1 = Yi + Sigma*2^-i*Xi                                          *
*                                                                         *
* The representation used by the XArray and YArray are shown below.  This *
* is vital to understand in order to understand the code below.  The      *
* number of rows is equal to the number of iterations plus 2.  The first  *
* additional row is for representing whether this is an X or a Y variable.*
* If the bit is set ( == 1), then it is an X variable.  Otherwise it is a *
* Y variable.  Because the arrays are initialized to zeros, all variables *
* are Ys unless they are intentionally set.                               *
*                                                                         *
* The second extra row is for the sign of the term.  If the bit is set, it*
* means taht this term should be subtracted from the equation. (i.e. -1)  *
* Because the arrays are initialized to all zeros, all varaibles are      *
* positive unless they are intentionally set.                             *
*                                                                         *
* The number of columns is equal to the number of terms in the final      *
* equation (i.e. one to represent each of the terms)                      *
*                                                                         *
*                                TERMS                                    *
*                                                                         *
*             0   1   2   3   4   5   6   7                    2^N        *
*           +---+---+---+---+---+---+---+---+-.............+---+---+      *
*  X / Y    |   |   |   |   |   |   |   |   |              |   |   |      *
*           +---+---+---+---+---+---+---+---+-.............+---+---+      *
*  Sign     |   |   |   |   |   |   |   |   |              |   |   |      *
*           +---+---+---+---+---+---+---+---+-.............+---+---+      *
*  Sigma0   |   |   |   |   |   |   |   |   |              |   |   |      *
*           +---+---+---+---+---+---+---+---+-.............+---+---+      *
*  Sigma1   |   |   |   |   |   |   |   |   |              |   |   |      *
*           +---+---+---+---+---+---+---+---+-.............+---+---+      *
*           .   .   .   .   .   .   .   .   .              .   .   .      *
*           .   .   .   .   .   .   .   .   .              .   .   .      *
*           .   .   .   .   .   .   .   .   .              .   .   .      *
*           +---+---+---+---+---+---+---+---+-.............+---+---+      *
```

```
*  SigmaN-1 |   |   |   |   |   |   |   |   |              |   |   |     *
*           +---+---+---+---+---+---+---+---+-.............-+---+---+     *
*  SigmaN   |   |   |   |   |   |   |   |   |              |   |   |     *
*           +---+---+---+---+---+---+---+---+-.............-+---+---+     *
*                                                                         *
\*************************************************************************/

   // Place the appropriate initial representations in the X & Y Arrays
   XArray[0].SetAt(0,0);
   XArray[1].SetAt(0,0);
   YArray[0].SetAt(0,1);
   YArray[1].SetAt(0,0);

   for ( i = 0 ; i < (StopIterations - StartIterations) ; i++ )
   {
      // The number of terms for this iteration step
      PSize = (int) pow(2,i);

      for ( j = 0 ; j < PSize ; j++)
      {
         // Determine if the new term is an X or Y
         OldVarX  = XArray[0].GetAt(j);
         if (OldVarX == 0)                      // If it was a Y....
         {
            XArray[0].SetAt(j + PSize, 1);      // Intentionally set an X
         }

         OldVarY  = YArray[0].GetAt(j );
         if (OldVarY == 0)                      // If it was a Y
         {
            YArray[0].SetAt(j + PSize, 1);      // Intentionally set an X
         }

         //Determine if the new term is a + or -
         OldSignX = XArray[1].GetAt(j);
         if ((OldVarX == 0) && (OldSignX == 0))
         {
            XArray[1].SetAt(j + PSize, 1);
         }
         if ((OldVarX == 1) && (OldSignX == 1))
         {
            XArray[1].SetAt(j + PSize, 1);
         }

         OldSignY = YArray[1].GetAt(j);
         if ((OldVarY == 0) && (OldSignY == 0))
         {
            YArray[1].SetAt(j + PSize, 1);
         }
         if ((OldVarY == 1) && (OldSignY == 1))
         {
            YArray[1].SetAt(j + PSize, 1);
         }

         // Copy the old sigma values into the new term
         for ( k = NumberOfIterations + 1 ; k > NumberOfIterations + 1 – I
            ; k-- )
```

```
      {
         OldSigmaX = XArray[k].GetAt(j);
         if (OldSigmaX == 1)
         {
            XArray[k].SetAt(j + PSize, 1);
         }

         OldSigmaY = YArray[k].GetAt(j);
         if (OldSigmaY == 1)
         {
            YArray[k].SetAt(j + PSize, 1);
         }
      }

      //Add a sigma for the new term
      XArray[k].SetAt(j + PSize, 1);
      YArray[k].SetAt(j + PSize, 1);
   }
}

// Print out the X terms of the X equation
fprintf(FormFile, "X%d = X%d * ( ", StopIterations, StartIterations );
for ( i = 0 ; i < NumberOfTerms ; i++ )
{
   if ( XArray[0].GetAt(i) == 0 )
   {
      if ( XArray[1].GetAt(i) == 0 )
         fprintf(FormFile, "+ " );
      else
         fprintf(FormFile, "- " );

      if ( i == 0)
         fprintf(FormFile, "1 " );
      else
      {
         for ( j = 0 ; j < NumberOfIterations ; j++ )
         {
            if ( XArray[NumberOfIterations + 1 - j].GetAt(i) == 1 )
               fprintf(FormFile, "s%d", StopIterations - j - 1 );
         }
         fprintf(FormFile, "*" );
         ExpSize = 0;
         for ( j = 0 ; j < NumberOfIterations ; j++ )
         {
            if ( XArray[NumberOfIterations + 1 - j].GetAt(i) == 1 )
               ExpSize = ExpSize + StopIterations - j - 1;
         }
         fprintf(FormFile, "2(-%d) ", ExpSize );
      }
   }
}
fprintf(FormFile, " ) +\n" );

// Print out the Y terms of the X equation
fprintf(FormFile, "     Y%d * ( ", StartIterations);
for ( i = 0 ; i < NumberOfTerms ; i++ )
{
```

```
   if ( NumberOfTerms == 1 )
      fprintf(FormFile, "0 " );

   if ( XArray[0].GetAt(i) == 1 )
   {
      if ( XArray[1].GetAt(i) == 0 )
         fprintf(FormFile, "+ " );
      else
         fprintf(FormFile, "- " );

      for ( j = 0 ; j < NumberOfIterations ; j++ )
      {
         if ( XArray[NumberOfIterations + 1 - j].GetAt(i) == 1 )
            fprintf(FormFile, "s%d", StopIterations - j - 1 );
      }
      fprintf(FormFile, "*" );
      ExpSize = 0;
      for ( j = 0 ; j < NumberOfIterations ; j++ )
      {
         if ( XArray[NumberOfIterations + 1 - j].GetAt(i) == 1 )
            ExpSize = ExpSize + StopIterations - j - 1;
      }
      fprintf(FormFile, "2(-%d) ", ExpSize);
   }
}
fprintf(FormFile, " )\n\n\n" );

// Print out the X terms of the Y equation
fprintf(FormFile, "Y%d = X%d * ( ", StopIterations, StartIterations );
for ( i = 0 ; i < NumberOfTerms ; i++ )
{
   if ( NumberOfTerms == 1 )
      fprintf(FormFile, "0 " );

   if ( YArray[0].GetAt(i) == 0 )
   {
      if ( YArray[1].GetAt(i) == 0 )
         fprintf(FormFile, "+ " );
      else
         fprintf(FormFile, "- " );

      for ( j = 0 ; j < NumberOfIterations ; j++ )
      {
         if ( YArray[NumberOfIterations + 1 - j].GetAt(i) == 1 )
            fprintf(FormFile, "s%d", StopIterations - j - 1 );
      }
      fprintf(FormFile, "*" );
      ExpSize = 0;
      for ( j = 0 ; j < NumberOfIterations ; j++ )
      {
         if ( YArray[NumberOfIterations + 1 - j].GetAt(i) == 1 )
            ExpSize = ExpSize + StopIterations - j - 1;
      }
      fprintf(FormFile, "2(-%d) ", ExpSize);
   }
}
fprintf(FormFile, " ) +\n");
```

```
// Print out the Y terms of the Y equation
fprintf(FormFile, "     Y%d * ( ", StartIterations );
for ( i = 0 ; i < NumberOfTerms ; i++ )
{
   if ( YArray[0].GetAt(i) == 1 )
   {
      if ( YArray[1].GetAt(i) == 0 )
         fprintf(FormFile, "+ ");
      else
         fprintf(FormFile, "- ");

      if ( i == 0 )
         fprintf(FormFile, "1 ");
      else
      {
         for ( j = 0 ; j < NumberOfIterations ; j++ )
         {
            if ( YArray[NumberOfIterations + 1 - j].GetAt(i) == 1 )
               fprintf(FormFile, "s%d", StopIterations - j - 1 );
         }
         fprintf(FormFile, "*" );
         ExpSize = 0;
         for ( j = 0 ; j < NumberOfIterations ; j++ )
         {
            if ( YArray[NumberOfIterations + 1 - j].GetAt(i) == 1 )
               ExpSize = ExpSize + StopIterations - j - 1;
         }
         fprintf(FormFile, "2(-%d) ", ExpSize);
      }
   }
}
fprintf(FormFile, " )\n\n\n");

fclose(FormFile);


// Calculate the expansion factor for the given set of iterations
K = 1.0;
for ( i = StartIterations ; i < StopIterations ; i++ )
{
   K = K * sqrt(1.0 + pow(2,-2*i) );
}

// Determine the effective CORDIC rotation angles that would be produced
//      for each combination of sigma values.
for ( i = 0 ; i < NumberOfIterations ; i++)
{
   index = 0;
   ArcTangentRadix = atan(pow(2,-1*(i + StartIterations)));

   for ( j = 0 ; j < pow(2,i) ; j++ )
   {
      for ( k = 0 ; k < pow(2,(NumberOfIterations - i - 1)) ; k++ )
      {
         Z_Partition[index] += ArcTangentRadix;
         index++;
```

```
            }
            for ( k = 0 ; k < pow(2,(NumberOfIterations - i - 1)) ; k++ )
            {
               Z_Partition[index] -= ArcTangentRadix;
               index++;
            }
         }
      }

      // Calculate the coefficients from the effective CORDIC rotation angle
      for ( i = 0 ; i < NumberOfTerms ; i++ )
      {
         XX0 = cos(Z_Partition[i]);
         YX0 = sin(Z_Partition[i]);
         Z0  = Z_Partition[i] * 57.295779513082320876798154814105;

         fprintf(IterFile, "%d\t %26.22f\t %26.22f\t %26.22f\t %26.22f\t
            %26.22f\n", i, XX0*K, -1*YX0*K, YX0*K, XX0*K, Z0);

         fprintf(NormFile, "%d\t %26.22f\t %26.22f\t %26.22f\t %26.22f\t
            %26.22f\n", i, XX0,   -1*YX0,   YX0,   XX0,   Z0);
      }

      fclose(IterFile);
      fclose(NormFile);

      EndCalculations = clock();

      CalculationTime = (double)(EndCalculations - BeginCalculations)/CLK_TCK;

      printf( "The elapsed calculation time was %f seconds.\n",
         CalculationTime);

      return (1);
   }
```

Appendix F

DtoB.m

```
function Binary = DtoB(Decimal, Bits)

%DtoB - Algorithm for converting Decimal numbers into Binary numbers
%
%     - Decimal -> The decimal number to be converted into binary
%     - Bits    -> The number of bits in the array
%
%     - Binary  -> The array representing the two's complement
%                     binary "number"
%
%     Copyrighted by
%     Jason T. Arbaugh
%     December 2004
%

if Decimal < 0
    Binary(1) = 1;
    Diff      = -2;
else
    Binary(1) = 0;
    Diff      = 0;
end

for i = 2:Bits
    if ( Decimal - Diff ) < pow2(2-i)
        Binary(i) = 0;
    else
        Binary(i) = 1;
        Diff      = Diff + pow2(2-i);
    end
end

if ( Decimal - Diff ) >= pow2(1-Bits)
    for i = 1:Bits-1
        OneArray(i) = 0;
    end
    OneArray(Bits) = 1;
    [Binary, Overflow] = BinAdd(Binary, OneArray, Bits);
end
```

Appendix G

BtoD.m

```
function Decimal = BtoD(Array, Bits)

%BtoD - Algorithm for converting Binary numbers into Decimal numbers
%
%     - Array   -> The array representing the binary "number"
%                  in two's complement
%     - Bits    -> The number of bits in the array
%
%     - Decimal -> The decimal number equivalent to the binary
%                  "number" represented in Array
%
%     Copyrighted by
%     Jason T. Arbaugh
%     December 2004
%

if Array(1) == 1
    Decimal = -2;
else
    Decimal = 0;
end

for i = 2:Bits
    Decimal = Decimal + Array(i)*pow2(2-i);
end
```

Appendix H

BinAdd.m

```
function [Sum, OF] = BinAdd(A, B, Bits, Cin)

%BinAdd - Algorithm for adding two's complement Binary numbers
%
%       - A    -> The two's complement representation of the augend
%                 in array format
%       - B    -> The two's complement representation of the addend
%                 in array format
%       - Bits -> The number of bits in the array
%       - Cin  -> Value of the Carry into the first bit position
%
%       - Sum  -> The summation of the augend (A) and the addend (B) in
%                 array format
%       - OF   -> The Overflow flag set if an overflow occurs
%
%     Copyrighted by
%     Jason T. Arbaugh
%     December 2004
%

if ( nargin == 3 )
    Cin = 0;
end

Carry(Bits+1) = Cin;

for i = Bits:-1:1

    if (xor(Carry(i+1), xor(A(i), B(i))))
        Sum(i) = 1;
    else
        Sum(i) = 0;
    end

    if ( ( A(i) & B(i) ) | ( A(i) & Carry(i+1) ) | ( B(i) & Carry(i+1) ) )
        Carry(i) = 1;
    else
        Carry(i) = 0;
    end
end

OF = xor(Carry(1), Carry(2));
```

Appendix I

BinSub.m

```
function [Diff, OF] = BinSub(A, B, Bits)

%BinAdd  - Algorithm for subtracting two's complement numbers in array
%             format
%
%        - A    -> The two's complement representation of the minuend
%                     in array format
%        - B    -> The two's complement representation of the subtrahend
%                     in array format
%        - Bits -> The number of bits in the array
%
%        - Diff -> The difference between the minuend (A) and subtrahend
%                     (B) in array format
%        - OF   -> The Overflow flag set if an overflow occurs
%
%     Copyrighted by
%     Jason T. Arbaugh
%     December 2004
%

for i = 1:Bits
    if ( B(i) == 1 )
        B(i) = 0;
    else
        B(i) = 1;
    end
end

[Diff, OF] = BinAdd(A, B, Bits, 1);
```

Appendix J

RShift.m

```
function Shifted = RShift(Array, Shift, Bits)

%RShift  - Algorithm for shifting two's complement numbers to the
%             right (i.e. divide by 2)
%
%        - Array   -> The array representing the two's complement
%                     binary "number"
%        - Shift   -> The number of shifts the perform
%        - Bits    -> The number of bits in the array
%
%        - Shifted -> The shifted array representing the
%                     binary "number" divided by two
%
%     Copyrighted by
%     Jason T. Arbaugh
%     December 2004
%

for i = Bits:-1:Shift+1
    Shifted(i) = Array(i-Shift);
end

for i = Shift:-1:1
    Shifted(i) = Array(1);
end
```

Appendix K

DtoRB.m

```
function Binary = DtoRB(Decimal, Bits)

%DtoRB - Algorithm for converting Decimal numbers into Redundant
%           Binary numbers
%
%      - Decimal -> The decimal number to be converted into binary
%      - Bits    -> The number of bits in the array
%
%      - Binary  -> The array representing the Signed Binary Digit (SBD)
%                     / redundant binary "number"
%
%     Copyrighted by
%     Jason T. Arbaugh
%     August 2004
%

Binary(1,1) = 0;
Binary(2,1) = 0;

if Decimal < 0
    Binary(1,2) = 1;
    Binary(2,2) = 1;
    Diff        = -2;
else
    Binary(1,2) = 0;
    Binary(2,2) = 0;
    Diff        = 0;
end

for i = 2:Bits
    if ( Decimal - Diff ) < pow2(2-i)
        Binary(1,i+1) = 0;
        Binary(2,i+1) = 0;
    else
        Binary(1,i+1) = 0;
        Binary(2,i+1) = 1;
        Diff        = Diff + pow2(2-i);
    end
end

if ( Decimal - Diff ) >= pow2(1-Bits)
    for i = 1:Bits
        OneArray(1,i) = 0;
        OneArray(2,i) = 0;
    end
    OneArray(1,Bits+1) = 0;
    OneArray(2,Bits+1) = 1;
    [Binary, OF] = RBinAdd(Binary, OneArray, Bits);
end
```

Appendix L

RBtoD.m

```
function Decimal = RBtoD(Array, Bits)

%RBtoD - Algorithm for converting Signed Binary Digit / redundant numbers
%           into Decimal numbers
%
%      - Array   -> The array representing the Signed Binary Digit /
%                       redundant number
%      - Bits    -> The number of bits in the array
%
%      - Decimal -> The decimal number equivalent to the redundant
%                      binary "number" represented in Array
%
%     Copyrighted by
%     Jason T. Arbaugh
%     December 2004
%

for i = 1:Bits+1
    if (Array(1,i) == 1)
        Array(2,i) = 0;
    end
end

[Diff, C0] = BinSub(Array(2,:), Array(1,:), Bits+1);

if Diff(2) == 1
    Decimal = -2;
else
    Decimal = 0;
end

for i = 3:Bits+1
    Decimal = Decimal + Diff(i)*pow2(3-i);
end
```

Appendix M

RBinAdd.m

```
function [Sum, OF] = RBinAdd(L, K, Bits)

%RBinAdd  - Algorithm for adding two Redundant Binary numbers in an
%              array format
%
%         - L     -> The Signed Binary Digit / redundant representation of
%                       the augend in array format
%         - K     -> The Signed Binary Digit / redundant representation of
%                       the addend in array format
%         - Bits  -> The number of bits in the array
%
%         - Sum   -> The summation of the augend (A) and the addend (B) in
%                       array format
%         - OF    -> The Overflow flag set if an overflow occurs
%
%     Copyrighted by
%     Jason T. Arbaugh
%     December 2004
%

K(1,Bits+2) = 0;
K(2,Bits+2) = 0;
L(1,Bits+2) = 0;
L(2,Bits+2) = 0;
M(Bits+2)   = 0;

for i = Bits+1:-1:1
    if ( ~L(1,i+1) & ~K(1,i+1) &
        ( L(2,i+1) | K(2,i+1) ) )

        M(i) = 1;
    else
        M(i) = 0;
    end

    if ( ( ~M(i+1) & ~L(2,i+1) & K(2,i+1) ) |
         ( ~M(i+1) & L(2,i+1) & ~K(2,i+1) ) |
         ( L(1,i+1) & K(1,i+1) ) )


        B(i) = 1;
    else
        B(i) = 0;
    end

    D(i)  = mod( ( M(i) + L(2,i) + K(2,i) ), 2 );
```

```
    if ( ~D(i) & B(i) )
        Sum(1,i) = 1;
    else
        Sum(1,i) = 0;
    end

    Sum(2,i) = mod( ( D(i) + B(i) ), 2 );

end

if ( ~L(1,1) & ~K(1,1) & ( L(2,1) | K(2,1) ) )
    MCarry = 1;
else
    MCarry = 0;
end

if ( ( ~M(1) & ~L(2,1) & K(2,1) ) |
     ( ~M(1) & L(2,1) & ~K(2,1) ) |
     ( L(1,1) & K(1,1) ) )


    BCarry = 1;
else
    BCarry = 0;
end

OF = mod( (MCarry + BCarry), 2);
```

Appendix N

RBinSub.m

```
function [Diff, UF] = RBinSub(L, K, Bits)

%RBinAdd  - Algorithm for subtracting two Signed Binary Digit / redundant
%              numbers in an array format
%
%         - L     -> The Signed Binary Digit / redundant minuend
%                       in array format
%         - K     -> The Signed Binary Digit / redundant subtrahend
%                       in array format
%         - Bits  -> The number of bits in the array
%
%         - Diff  -> The difference between the minuend (L) and subtrahend
%                        (K) in array format
%         - UF    -> The Underflow flag set if an overflow occurs
%
%     Copyrighted by
%     Jason T. Arbaugh
%     December 2004
%

K(1,Bits+2) = 0;
K(2,Bits+2) = 0;
L(1,Bits+2) = 0;
L(2,Bits+2) = 0;
M(Bits+2)   = 0;

for i = Bits+1:-1:1
    if ( K(1,i+1) | ( ~L(1,i+1) & L(2,i+1) & ~K(2,i+1) ) )
        M(i) = 1;
    else
        M(i) = 0;
    end

    if ( ( ~M(i+1) & ~L(2,i+1) & K(2,i+1) ) |
         ( ~M(i+1) & L(2,i+1) & ~K(2,i+1) ) |
         ( L(1,i+1) & K(2,i+1) ) )
        B(i) = 1;
    else
        B(i) = 0;
    end

    D(i)  = mod( ( M(i) + L(2,i) + K(2,i) ), 2 );

    if ( ~D(i) & B(i) )
        Diff(1,i) = 1;
    else
        Diff(1,i) = 0;
    end
```

```
    Diff(2,i) = mod( ( D(i) + B(i) ), 2 );

end

if ( K(1,1) | ( ~L(1,1) & L(2,1) & ~K(2,1) ) )
    MCarry = 1;
else
    MCarry = 0;
end

if ( ( ~M(1) & ~L(2,1) & K(2,1) ) |
     ( ~M(1) & L(2,1) & ~K(2,1) ) |
     ( L(1,1) & K(2,1) ) )
    BCarry = 1;
else
    BCarry = 0;
end

UF = mod( (MCarry + BCarry), 2);
```

Appendix O

RRShift.m

```
function Shifted = RRShift(Array, Shift, Bits)

%RRShift - Algorithm for shifting Signed Binary Digit / redundant numbers
%             to the right (i.e. divide by 2)
%
%        - Array   -> The array representing the Signed Binary Digit /
%                        redundant binary "number"
%        - Shift   -> The number of shifts to perform
%        - Bits    -> The number of bits in the array
%
%        - Shifted -> The shifted array representing the Signed Binary
%                        Digit / redundant "number" divided by two
%
%     Copyrighted by
%     Jason T. Arbaugh
%     December 2004
%

for i = Bits+1:-1:Shift+1
    Shifted(1,i) = Array(1,i-Shift);
    Shifted(2,i) = Array(2,i-Shift);
end

for i = Shift:-1:1
    Shifted(1,i) = 0;
    Shifted(2,i) = 0;
end
```

Appendix P

SBC_Rotate.m

```
function [X1p, X1n, Y1p, Y1n, Z1p, Z1n] =
                SBC_Rotate(X0p, X0n, Y0p, Y0n, Z0p, Z0n, S0p, S0n, i, Bits)

%SBC_Rotate - Algorithm for performing a Step Branching rotation
%
%           - X0p/X0n  -> The initial X value of the "+" / "-" hardware
%           - Y0p/Y0n  -> The initial Y value of the "+" / "-" hardware
%           - Z0p/X0n  -> The initial Z value of the "+" / "-" hardware
%           - S0p/S0n  -> The sign of the residual angle Z in the
%                         "+" / "-" hardware
%           - i        -> The current iteration of the Step Branching
%                         CORDIC algorithm
%           - Bits     -> The number of bits in the array
%
%           - X1p/X1n  -> The rotated X value of the "+" / "-" hardware
%           - Y1p/Y1n  -> The rotated Y value of the "+" / "-" hardware
%           - Z1p/Z1n  -> The residual angle of the "+" / "-" hardware
%
%     Copyrighted by
%     Jason T. Arbaugh
%     December 2004
%

X0pShift = RRShift(X0p, i-1, Bits);
X0nShift = RRShift(X0n, i-1, Bits);
Y0pShift = RRShift(Y0p, i-1, Bits);
Y0nShift = RRShift(Y0n, i-1, Bits);
ArcTan   = DtoRB(atan(2^(-1*(i-1))), Bits);

if S0p == 1
    [X1p, Out] = RBinSub(X0p, Y0pShift, Bits);
    [Y1p, Out] = RBinAdd(Y0p, X0pShift, Bits);
    [Z1p, Out] = RBinSub(Z0p, ArcTan,   Bits);
else
    [X1p, Out] = RBinAdd(X0p, Y0pShift, Bits);
    [Y1p, Out] = RBinSub(Y0p, X0pShift, Bits);
    [Z1p, Out] = RBinAdd(Z0p, ArcTan,   Bits);
end

if S0n == 1
    [X1n, Out] = RBinSub(X0n, Y0nShift, Bits);
    [Y1n, Out] = RBinAdd(Y0n, X0nShift, Bits);
    [Z1n, Out] = RBinSub(Z0n, ArcTan,   Bits);
else
    [X1n, Out] = RBinAdd(X0n, Y0nShift, Bits);
    [Y1n, Out] = RBinSub(Y0n, X0nShift, Bits);
    [Z1n, Out] = RBinAdd(Z0n, ArcTan,   Bits);
end
```

Appendix Q

SBC_AngleEval.m

```
function [S0p, S0n] = SBC_AngleEval(Z0p, Z0n, i, Bits)

%SBC_AngleEval  - Algorithm determining the sign of redundant angles
%
%         - Z0p   -> The Signed Binary Digit / redundant representation of
%                       the residual angle of the "+" hardware
%         - Z0n   -> The Signed Binary Digit / redundant representation of
%                       the residual angle of the "-" hardware
%         - i     -> The current start of the floating angle evaluation
%                       window which is equal to the current iteration
%         - Bits  -> The number of bits in the array
%
%         - S0p   -> The sign of the angle from the "+" hardware
%         - S0n   -> The sign of the angle from the "-" hardware
%
%     Copyrighted by
%     Jason T. Arbaugh
%     December 2004
%

Pad = [ 0; 0];

PAngle = cat(2, Z0p, Pad);
NAngle = cat(2, Z0n, Pad);

PValue =   4*PAngle(2,i) + 2*PAngle(2,i+1) + PAngle(2,i+2)
         - 8*PAngle(1,i) - 4*PAngle(1,i+1) - 2*PAngle(1,i+2);


NValue =   4*NAngle(2,i) + 2*NAngle(2,i+1) + NAngle(2,i+2)
         - 8*NAngle(1,i) - 4*NAngle(1,i+1) - 2*NAngle(1,i+2);

if ( mod(PValue, 8) == 0 )
    S0p = 0;
elseif ( ( mod(PValue, 8) ) < 4 )
    S0p = 1;
else
    S0p = -1;
end

if ( mod(NValue, 8) == 0 )
    S0n = 0;
elseif ( ( mod(NValue, 8) ) < 4 )
    S0n = 1;
else
    S0n = -1;
end
```

Appendix R

DSBC_Rotate.m

```
function [X1, Y1, Z1] = DSBC_Rotate(X0, Y0, Z0, S1, S2, i, Bits)


%RBinAdd  - Algorithm for adding two Redundant Binary numbers in an array
format
%         - KS    -> The sign bit of the first number to be added
%         - KD    -> The digit of the first number to be added
%         - LS    -> The sign bit of the first number to be added
%         - LD    -> The digit of the first number to be added
%         - Bits  -> The number of bits in the arrays
%         - Carry -> Value of the Carry into the first bit position
%
% $Revision: 1.0 $
%
% Copyrighted by
% Jason T. Arbaugh
% December 2004

if ( S1 == S2 )
    [X1Temp1, Out] = RBinAdd(Y0, RRShift(Y0, 1, Bits), Bits);
    [Y1Temp1, Out] = RBinAdd(X0, RRShift(X0, 1, Bits), Bits);
else
    [X1Temp1, Out] = RBinSub(Y0, RRShift(Y0, 1, Bits), Bits);
    [Y1Temp1, Out] = RBinSub(X0, RRShift(X0, 1, Bits), Bits);
end

X1Shift1 = RRShift(X0, 4*i-3, Bits);
Y1Shift1 = RRShift(Y0, 4*i-3, Bits);

if ( S1 == S2 )
    [X1Temp2, Out] = RBinSub(X0, X1Shift1, Bits);
    [Y1Temp2, Out] = RBinSub(Y0, Y1Shift1, Bits);
else
    [X1Temp2, Out] = RBinAdd(X0, X1Shift1, Bits);
    [Y1Temp2, Out] = RBinAdd(Y0, Y1Shift1, Bits);
end

X1Shift2 = RRShift(X1Temp1, 2*i-2, Bits);
Y1Shift2 = RRShift(Y1Temp1, 2*i-2, Bits);

if ( S1 == 1 )
    [X1, Out] = RBinSub(X1Temp2, X1Shift2, Bits);
    [Y1, Out] = RBinAdd(Y1Temp2, Y1Shift2, Bits);
else
    [X1, Out] = RBinAdd(X1Temp2, X1Shift2, Bits);
    [Y1, Out] = RBinSub(Y1Temp2, Y1Shift2, Bits);
end
```

```
ArcTan1  = DtoRB(atan(2^(-1*(2*i-2))), Bits);
ArcTan2  = DtoRB(atan(2^(-1*(2*i-1))), Bits);

if ( S1 == -1 )
    [Z1Temp, Out] = RBinAdd(Z0, ArcTan1, Bits);
else
    [Z1Temp, Out] = RBinSub(Z0, ArcTan1, Bits);
end

if ( S2 == -1 )
    [Z1, Out] = RBinAdd(Z1Temp, ArcTan2, Bits);
else
    [Z1, Out] = RBinSub(Z1Temp, ArcTan2, Bits);
end
```

Appendix S

DSBC_AngleEval.m

```
function [Sign, Flag] = DSBC_AngleEval(Z0, i, Bits)


%DSBC_AngleEval  - Algorithm for adding two Redundant Binary numbers in an
array format
%                - Z0    -> The sign bit of the first number to be added
%                - i     -> The sign bit of the first number to be added
%                - Bits  -> The number of bits in the arrays
%
% $Revision: 1.0 $
%
% Copyrighted by
% Jason T. Arbaugh
% 2004

Pad = [ 0 0 0 0; 0 0 0 0];

Angle = cat(2, Z0, Pad);

ValueU3 = 4*Angle(2,2*i)   + 2*Angle(2,2*i+1) + Angle(2,2*i+2) –
          8*Angle(1,2*i)   - 4*Angle(1,2*i+1) - 2*Angle(1,2*i+2);

ModOfU3 = mod(ValueU3, 8);


ValueL3 = 4*Angle(2,2*i+3) + 2*Angle(2,2*i+4) + Angle(2,2*i+5) –
          8*Angle(1,2*i+3) - 4*Angle(1,2*i+4) - 2*Angle(1,2*i+5);

MagOfL3 = abs(ValueL3);

if ( ModOfU3 == 0 )
    Sign1   = 0;
    MagOfU3 = 0;
elseif ( ModOfU3 < 4 )
    Sign1   = 1;
    MagOfU3 = ModOfU3;
else
    Sign1   = -1;
    MagOfU3 = 8 - ModOfU3;
end

if ( ValueL3 > 0 )
    Sign2 = 1;
elseif ( ValueL3 == 0 )
    Sign2 = 0;
else
    Sign2 = -1;
end
```

```
if ( Sign1 == 0 )
    Sign = Sign2;
else
    Sign = Sign1;
end

Flag = ((Sign1 == 0) & (MagOfL3 <= 6)) |
       ((MagOfU3 == 1) & (MagOfL3 >= 2) & (Sign2 == (-1*Sign1)));
```

Appendix T

HCCAM.m

```
function [X0, Y0, Z0, XN, YN, ZN] = HCCAM(Z0, Bits)


%HCCAM   - Hybrid CORDIC Iteration for calculating sine and cosine
%        - Z0   -> The angle being used to access the ROM
%        - Bits -> Number of bits of precision to keep
%
% $Revision: 2.0 $
%
% Copyrighted by
% Jason T. Arbaugh
% 2004

global TABLE00to04_08B;
global TABLE00to08_16B;
global TABLE00to08_24B;
global TABLE00to12_32B;
global TABLE00to16_48B;


if ( Bits == 8 )
    Index = 1;
    for i = 1:Bits
        ZTemp(i) = TABLE00to04_08B(Index,i);
    end
    while ( BtoD(ZTemp, Bits) > BtoD(Z0, Bits) )
        Index = Index + 1;
        for i = 1:Bits
            ZTemp(i) = TABLE00to04_08B(Index,i);
        end
    end
end


if ( Bits == 16 )
    Index = 1;
    for i = 1:Bits
        ZTemp(i) = TABLE00to08_16B(Index,i);
    end
    while ( BtoD(ZTemp, Bits) > BtoD(Z0, Bits) )
        Index = Index + 1;
        for i = 1:Bits
            ZTemp(i) = TABLE00to08_16B(Index,i);
        end
    end
end
```

```
if ( Bits == 24 )
    Index = 1;
    for i = 1:Bits
        ZTemp(i) = TABLE00to08_24B(Index,i);
    end
    while ( BtoD(ZTemp, Bits) > BtoD(Z0, Bits) )
        Index = Index + 1;
        for i = 1:Bits
            ZTemp(i) = TABLE00to08_24B(Index,i);
        end
    end
end


if ( Bits == 32 )
    Index = 1;
    for i = 1:Bits
        ZTemp(i) = TABLE00to12_32B(Index,i);
    end
    while ( BtoD(ZTemp, Bits) > BtoD(Z0, Bits) )
        Index = Index + 1;
        for i = 1:Bits
            ZTemp(i) = TABLE00to12_32B(Index,i);
        end
    end
end


if ( Bits == 48 )
    Index = 1;
    for i = 1:Bits
        ZTemp(i) = TABLE00to16_48B(Index,i);
    end
    while ( BtoD(ZTemp, Bits) > BtoD(Z0, Bits) )
        Index = Index + 1;
        for i = 1:Bits
            ZTemp(i) = TABLE00to16_48B(Index,i);
        end
    end
end
```

Appendix U

HCPipe.m

```
function [XF, YF, ZF] = HCPipe(X0, Y0, Z0, Start, Finish, Bits)


%HCORDIC - Hybrid CORDIC Iteration for calculating sine and cosine
%        - Angle -> The angle being calculated
%        - Bits  -> Number of bits of precision to keep
%
% $Revision: 2.0 $
%
% Copyrighted by
% Jason T. Arbaugh
% 2004


ZA = Z0;

if ( Z0(4*(Start-1)+1) == 0 )
    ZA(4*(Start-1)+1) = 1;
else
    ZA(4*(Start-1)+1) = 0;
end


for i = Start:Finish

    if ( ZA(4*(i-1)+1) == 1 )
        [X1, OF] = BinSub(X0, RShift(Y0, 4*(i-1), Bits), Bits);
        [Y1, OF] = BinAdd(Y0, RShift(X0, 4*(i-1), Bits), Bits);
        [Z1, OF] = BinSub(Z0, DtoB(2^(-1*(4*(i-1)-1)), Bits), Bits);
    else
        [X1, OF] = BinAdd(X0, RShift(Y0, 4*(i-1), Bits), Bits);
        [Y1, OF] = BinSub(Y0, RShift(X0, 4*(i-1), Bits), Bits);
        [Z1, OF] = BinAdd(Z0, DtoB(2^(-1*(4*(i-1))), Bits), Bits);
    end


    if ( ZA(4*(i-1)+2) == 1 )
        [X2, OF] = BinSub(X1, RShift(Y0, 4*(i-1)+1, Bits), Bits);
        [Y2, OF] = BinAdd(Y1, RShift(X0, 4*(i-1)+1, Bits), Bits);
        [Z2, OF] = BinSub(Z1, DtoB(2^(-1*(4*(i-1))), Bits), Bits);
    else
        [X2, OF] = BinAdd(X1, RShift(Y0, 4*(i-1)+1, Bits), Bits);
        [Y2, OF] = BinSub(Y1, RShift(X0, 4*(i-1)+1, Bits), Bits);
        [Z2, OF] = BinAdd(Z1, DtoB(2^(-1*(4*(i-1)+1)), Bits), Bits);
    end
```

```
    if ( ZA(4*(i-1)+3) == 1 )
        [X3, OF] = BinSub(X2, RShift(Y0, 4*(i-1)+2, Bits), Bits);
        [Y3, OF] = BinAdd(Y2, RShift(X0, 4*(i-1)+2, Bits), Bits);
        [Z3, OF] = BinSub(Z2, DtoB(2^(-1*(4*(i-1)+1)), Bits), Bits);
    else
        [X3, OF] = BinAdd(X2, RShift(Y0, 4*(i-1)+2, Bits), Bits);
        [Y3, OF] = BinSub(Y2, RShift(X0, 4*(i-1)+2, Bits), Bits);
        [Z3, OF] = BinAdd(Z2, DtoB(2^(-1*(4*(i-1)+2)), Bits), Bits);
    end


    if ( ZA(4*(i-1)+4) == 1 )
        [X4, OF] = BinSub(X3, RShift(Y0, 4*(i-1)+3, Bits), Bits);
        [Y4, OF] = BinAdd(Y3, RShift(X0, 4*(i-1)+3, Bits), Bits);
        [Z4, OF] = BinSub(Z3, DtoB(2^(-1*(4*(i-1)+2)), Bits), Bits);
    else
        [X4, OF] = BinAdd(X3, RShift(Y0, 4*(i-1)+3, Bits), Bits);
        [Y4, OF] = BinSub(Y3, RShift(X0, 4*(i-1)+3, Bits), Bits);
        [Z4, OF] = BinAdd(Z3, DtoB(2^(-1*(4*(i-1)+3)), Bits), Bits);
    end


    X0 = X4;
    Y0 = Y4;
    Z0 = Z4;


end

XF = X4;
YF = Y4;
ZF = Z4;
```

Appendix V

BinMult.m

```
function Prod = BinMult(A, B, Bits)

%BinMult - Algorithm for multiplying two's complement Binary numbers
%
%         - A    -> The two's complement representation of the
%                   multiplier in array format
%         - B    -> The two's complement representation of the
%                   multiplicand in array format
%         - Bits -> The number of bits in the array
%
%         - Prod -> The product of multiplying the multiplier (A)
%                   by the multiplicand (B) in array format
%
%     Copyrighted by
%     Jason T. Arbaugh
%     December 2004
%

for i = 1:Bits
    M(i) = 0;
end

for i = Bits:-1:1
    NBits = 2*Bits - i;
    PP    = B(i) .* A;

    for j = Bits + 1 : NBits
        PP(j) = 0;
    end

    [M, Out]  = BinAdd( M, PP, NBits );

    for j = NBits + 1 : -1 : 2
        M(j) = M(j-1);
    end
    M(1) = Out;
end

for i = 1 : Bits
    Prod(i) = M(i+2);
end

if ( M(Bits+3) == 1 )
    for i = 1:Bits-1
        OneArray(i) = 0;
    end
    OneArray(Bits) = 1;
    [Prod, OF] = BinAdd(Prod, OneArray, Bits);
end
```

Appendix W

BtoIndex.m

```
function Index = BtoIndex(Array, Start, Finish)

%BtoIndex - Algorithm for converting two's complement binary numbers into
%              Parallel ATR ROM table indexes
%
%         - Array  -> The Binary Array to be converted into Decimal
%         - Start  -> The element to start at when calculating the index
%         - Finish -> The element to stop at when calculating the index
%
%         - Index  -> Integer equivalent of the bits from "Start" to
%                         "Finish" that can be used to index a TLC ROM table
%
%     Copyrighted by
%     Jason T. Arbaugh
%     December 2004
%

Index = 0;

for i = Finish:-1:Start
    Index = Index + Array(i)*pow2(Finish - i);
end
```

Appendix X

CCordic.m

```
function [X0, Y0, Z0, XD, YD, ZD] = CCordic(Angle, Iters, Bits)

%CCORDIC - Classic CORDIC Iteration for calculating sine and cosine
%
%        - Angle -> The angle (in degrees) being calculated
%        - Iters -> The number of iterations to perform
%        - Bits  -> The number of bits in the array
%
%        - X0    -> The final X value (Binary array) representing cosine
%                      in two's complement
%        - Y0    -> The final Y value (Binary array) representing sine
%                      in two's complement
%        - Z0    -> The final Z value (Binary array) representing the
%                      residual angle in radians in two's complement
%        - XD    -> The final X value (decimal) representing cosine
%        - YD    -> The final Y value (decimal) representing sine
%        - ZD    -> The final Z value (decimal) representing the
%                      residual angle in degrees
%
%     Copyrighted by
%     Jason T. Arbaugh
%     December 2004
%

K0 = 1.0;

for i = 0:(Iters-1)
    K0 = K0 * sqrt(1 + 2^(-2*i));
end

X0  = DtoB(1.0/K0      , Bits);
Y0  = DtoB(0.0         , Bits);
Z0  = DtoB(Angle*pi/180, Bits);

for i = 0:(Iters-1)

   XShift = RShift(X0, i, Bits);
   YShift = RShift(Y0, i, Bits);
   ArcTan = DtoB(atan(2^(-1*i)), Bits);

   if Z0(1) == 0
       X1 = BinSub(X0, YShift, Bits);
       Y1 = BinAdd(Y0, XShift, Bits);
       Z1 = BinSub(Z0, ArcTan, Bits);
   else
       X1 = BinAdd(X0, YShift, Bits);
       Y1 = BinSub(Y0, XShift, Bits);
       Z1 = BinAdd(Z0, ArcTan, Bits);
   end
```

```
   X0 = X1;
   Y0 = Y1;
   Z0 = Z1;

end

XD = BtoD(X0, Bits);
YD = BtoD(Y0, Bits);
ZD = BtoD(Z0, Bits)*180/pi;
```

Appendix Y

SBCordic.m

```
function [X0, Y0, Z0, XD, YD, ZD] = SBCordic(Angle, Iters, Bits)

%SBCORDIC - Branching CORDIC Iteration for calculating sine and cosine
%
%         - Angle -> The angle (in degrees) being calculated
%         - Iters -> The number of iterations to perform
%         - Bits  -> The number of bits in the array
%
%         - X0    -> The final X value (Binary array) representing cosine
%                       in two's complement
%         - Y0    -> The final Y value (Binary array) representing sine
%                       in two's complement
%         - Z0    -> The final Z value (Binary array) representing the
%                       residual angle in radians in two's complement
%         - XD    -> The final X value (decimal) representing cosine
%         - YD    -> The final Y value (decimal) representing sine
%         - ZD    -> The final Z value (decimal) representing the
%                         residual angle in degrees
%
%     Copyrighted by
%     Jason T. Arbaugh
%     December 2004
%

K0 = 1.0;

for i = 0:(Iters-1)
    K0 = K0 * sqrt(1 + 2^(-2*i));
end

X0p = DtoRB(1.0/K0, Bits);
X0n = X0p;

Y0p = DtoRB(0.0, Bits);
Y0n = Y0p;

Z0p = DtoRB(Angle*pi/180, Bits);
Z0n = Z0p;

i   = 1;
b   = 0;

[S0p, S0n] = SBC_AngleEval(Z0p, Z0n, i, Bits);

while i <= Iters

    while ((S0p ~= 0) & (S0n ~= 0) & (i <= Iters))

        if b == 0
```

```
        [X0p, X0n, Y0p, Y0n, Z0p, Z0n] = SBC_Rotate(X0p, X0n, Y0p, Y0n,
              Z0p, Z0n, S0p, S0n, i, Bits);

        [S0p, S0n] = SBC_AngleEval(Z0p, Z0n, i, Bits);

        i = i + 1;

    end
end

if i <= Iters

    S0p = 1;
    S0n = -1;

    [X0p, X0n, Y0p, Y0n, Z0p, Z0n] = SBC_Rotate(X0p, X0n, Y0p, Y0n,
          Z0p, Z0n, S0p, S0n, i, Bits);

    [S0p, S0n] = SBC_AngleEval(Z0p, Z0n, i, Bits);

    i = i + 1;

end

while ((S0p == -1) & (S0n == 1) & (i <= Iters))

    [X0p, X0n, Y0p, Y0n, Z0p, Z0n] = SBC_Rotate(X0p, X0n, Y0p, Y0n,
          Z0p, Z0n, S0p, S0n, i, Bits);

    [S0p, S0n] = SBC_AngleEval(Z0p, Z0n, i, Bits);

    i = i + 1;

end

if S0p == 0

    X0n = X0p;
    Y0n = Y0p;
    Z0n = Z0p;
    b   = 1;

elseif S0p == 1

   Z0n = Z0p;
    X0n = X0p;
    Y0n = Y0p;
    S0n = S0p;
    b   = 0;

elseif S0n == 0

    X0p = X0n;
    Y0p = Y0n;
    Z0p = Z0n;
    b   = 1;
```

```
    elseif S0n == -1

        X0p = X0n;
        Y0p = Y0n;
        Z0p = Z0n;
        S0p = S0n;
        b   = 0;

    end
end

[ZBDiff, CO] = RBinAdd(Z0p, Z0n, Bits);

ZDDiff = RBtoD(ZBDiff, Bits);

if ZDDiff < 0

    X0 = X0p;
    Y0 = Y0p;
    Z0 = Z0p;

else

    X0 = X0n;
    Y0 = Y0n;
    Z0 = Z0n;

end

XD = RBtoD(X0, Bits);
YD = RBtoD(Y0, Bits);
ZD = RBtoD(Z0, Bits)*180/pi;
```

Appendix Z

DSBCordic.m

```
function [X0, Y0, Z0, XN, YN, ZN] = DSBCordic(Angle, Iters, Bits)


%DSBCORDIC - Double Step Branching CORDIC Iteration for calculating sine
and cosine
%         - Angle -> The angle being calculated
%         - Iters -> Number of Iterations to perform
%         - Bits  -> Number of Bits to use in the operation
%
% $Revision: 1.0 $
%
% Copyrighted by
% Jason T. Arbaugh
% 2004

K0 = 1.0;

for i = 0:(Iters-1)
    K0 = K0 * sqrt(1 + 2^(-2*i));
end

X0a = DtoRB(1.0/K0, Bits);
X0b = X0a;

Y0a = DtoRB(0.0, Bits);
Y0b = Y0a;

Z0a = DtoRB(Angle*pi/180, Bits);
Z0b = Z0a;

i           = 1;
Branching   = 0;

[SignZ, FlagZ] = DSBC_AngleEval(Z0a, i, Bits);
SignZa = SignZ;
SignZb = SignZ;

for i = 1:ceil( (Iters)/2 )
    ACorrect = 0;
    BCorrect = 0;

    if ( Branching == 0 )

        if ( ( SignZ == 1 ) | ( SignZ == -1 )

            if ( SignZ == 1 )

                [X0a, Y0a, Z0a] = DSBC_Rotate(X0a, Y0a, Z0a,  1,  1, i,
                                              Bits);
```

```
            [SignZa, Flaga] = DSBC_AngleEval(Z0a, i, Bits);

            [X0b, Y0b, Z0b] = DSBC_Rotate(X0b, Y0b, Z0b,  1, -1, i,
                                          Bits);
            [SignZb, Flagb] = DSBC_AngleEval(Z0b, i, Bits);

        else
            [X0a, Y0a, Z0a] = DSBC_Rotate(X0a, Y0a, Z0a, -1, -1, i,
                                          Bits);
            [SignZa, Flaga] = DSBC_AngleEval(Z0a, i, Bits);

            [X0b, Y0b, Z0b] = DSBC_Rotate(X0b, Y0b, Z0b, -1,  1, i,
                                          Bits);
            [SignZb, Flagb] = DSBC_AngleEval(Z0b, i, Bits);

        end

        if ( SignZa == SignZ )
            ACorrect = 1;
        elseif ( SignZa == 0 )
            ACorrect = 1;
        else
            if ( ( SignZb == -1*SignZ ) | ( SignZb == 0 ) )
                BCorrect = 1;
            else
                if ( Flaga )
                    ACorrect = 1;
                elseif ( Flagb )
                    BCorrect = 1;
                else
                    Branching = 1;
                end
            end
        end

        if ( Branching == 0 )
            if ( ACorrect )
                X0b    = X0a;
                Y0b    = Y0a;
                Z0b    = Z0a;
                SignZb = SignZa;
                SignZ  = SignZa;
            elseif ( BCorrect )
                X0a    = X0b;
                Y0a    = Y0b;
                Z0a    = Z0b;
                SignZa = SignZb;
                SignZ  = SignZb;
            end
        end
    else
        [X0a, Y0a, Z0a] = DSBC_Rotate(X0a, Y0a, Z0a, -1,  1, i, Bits);
        [SignZa, Flaga] = DSBC_AngleEval(Z0a, i, Bits);

        [X0b, Y0b, Z0b] = DSBC_Rotate(X0b, Y0b, Z0b,  1, -1, i, Bits);
        [SignZb, Flagb] = DSBC_AngleEval(Z0b, i, Bits);
```

```
        if ( SignZa == 0 )
            ACorrect = 1;
        elseif ( SignZb == 0 )
            BCorrect = 1;
        elseif ( Flaga )
            ACorrect = 1;
        elseif ( Flagb )
            BCorrect = 1;
        else
            Branching = 1;
        end

        if ( Branching == 0 )
            if ( ACorrect )
                X0b    = X0a;
                Y0b    = Y0a;
                Z0b    = Z0a;
                SignZb = SignZa;
                SignZ  = SignZa;
            elseif ( BCorrect )
                X0a    = X0b;
                Y0a    = Y0b;
                Z0a    = Z0b;
                SignZa = SignZb;
                SignZ  = SignZb;
            end
        end
    end
else
    OldSignZa = SignZa;
    OldSignZb = SignZb;

    if ( SignZa == 1 )
        [X0a, Y0a, Z0a] = DSBC_Rotate(X0a, Y0a, Z0a,  1,  1, i, Bits);
        [SignZa, Flaga] = DSBC_AngleEval(Z0a, i, Bits);

        [X0b, Y0b, Z0b] = DSBC_Rotate(X0b, Y0b, Z0b, -1, -1, i, Bits);
        [SignZb, Flagb] = DSBC_AngleEval(Z0b, i, Bits);

    else
        [X0a, Y0a, Z0a] = DSBC_Rotate(X0a, Y0a, Z0a, -1, -1, i, Bits);
        [SignZa, Flaga] = DSBC_AngleEval(Z0a, i, Bits);

        [X0b, Y0b, Z0b] = DSBC_Rotate(X0b, Y0b, Z0b,  1,  1, i, Bits);
        [SignZb, Flagb] = DSBC_AngleEval(Z0b, i, Bits);

    end

    Branching = 0;

    if ( SignZa == 0 )
        ACorrect = 1;
    elseif ( SignZb == 0 )
        BCorrect = 1;
    elseif ( SignZa == -1*OldSignZa )
        ACorrect = 1;
    elseif ( SignZb == -1*OldSignZb )
```

```
            BCorrect = 1;
        elseif ( Flaga )
            ACorrect = 1;
        elseif ( Flagb )
            BCorrect = 1;
        else
            Branching = 1;
        end

        if ( Branching == 0 )
            if ( ACorrect )
                X0b    = X0a;
                Y0b    = Y0a;
                Z0b    = Z0a;
                SignZb = SignZa;
                SignZ  = SignZa;
            elseif ( BCorrect )
                X0a    = X0b;
                Y0a    = Y0b;
                Z0a    = Z0b;
                SignZa = SignZb;
                SignZ  = SignZb;
            end
        end
    end
end

[ZBDiff, CO] = RBinAdd(Z0a, Z0b, Bits);

ZDDiff = RBtoD(ZBDiff, Bits);

if ZDDiff < 0
    X0 = X0a;
    Y0 = Y0a;
    Z0 = Z0a;
else
    X0 = X0b;
    Y0 = Y0b;
    Z0 = Z0b;
end

XN = RBtoD(X0, Bits);
YN = RBtoD(Y0, Bits);
ZN = RBtoD(Z0, Bits)*180/pi;
```

Appendix AA

HCordic.m

```
function [X0, Y0, Z0, XN, YN, ZN] = HCordic(Angle, Bits)


%HCORDIC - Hybrid CORDIC Iteration for calculating sine and cosine
%        - Angle -> The angle being calculated
%        - Bits  -> Number of bits of precision to keep
%
% $Revision: 2.0 $
%
% Copyrighted by
% Jason T. Arbaugh
% 2004

global TABLE00to04_08B;
global TABLE00to08_16B;
global TABLE00to08_24B;
global TABLE00to12_32B;
global TABLE00to16_48B;


Z0  = DtoB(Angle*pi/180, Bits);


if ( Bits == 8 )
    Index = HCCAM(Z0, Bits);
    for i = 1:Bits
        X0(i) = TABLE00to04_08B(Index, i + 2*Bits);
        Y0(i) = TABLE00to04_08B(Index, i + 3*Bits);
        ZT(i) = TABLE00to04_08B(Index, i + 1*Bits);
    end
    [Z0, OF]     = BinAdd(Z0, ZT, Bits);
    [X0, Y0, Z0] = HCPipe(X0, Y0, Z0, 2, 2, Bits);
end


if ( Bits == 16 )
    Index = HCCAM(Z0, Bits);
    for i = 1:Bits
        X0(i) = TABLE00to08_16B(Index, i + 2*Bits);
        Y0(i) = TABLE00to08_16B(Index, i + 3*Bits);
        ZT(i) = TABLE00to08_16B(Index, i + 1*Bits);
    end
    [Z0, OF]     = BinAdd(Z0, ZT, Bits);
    [X0, Y0, Z0] = HCPipe(X0, Y0, Z0, 3, 4, Bits);
end
```

```
if ( Bits == 24 )
    Index = HCCAM(Z0, Bits);
    for i = 1:Bits
        X0(i) = TABLE00to08_24B(Index, i + 2*Bits);
        Y0(i) = TABLE00to08_24B(Index, i + 3*Bits);
        ZT(i) = TABLE00to08_24B(Index, i + 1*Bits);
    end
    [Z0, OF]     = BinAdd(Z0, ZT, Bits);
    [X0, Y0, Z0] = HCPipe(X0, Y0, Z0, 3, 6, Bits);
end


if ( Bits == 32 )
    Index = HCCAM(Z0, Bits);
    for i = 1:Bits
        X0(i) = TABLE00to12_32B(Index, i + 2*Bits);
        Y0(i) = TABLE00to12_32B(Index, i + 3*Bits);
        ZT(i) = TABLE00to12_32B(Index, i + 1*Bits);
    end
    [Z0, OF]     = BinAdd(Z0, ZT, Bits);
    [X0, Y0, Z0] = HCPipe(X0, Y0, Z0, 4, 8, Bits);
end


if ( Bits == 48 )
    Index = HCCAM(Z0, Bits);
    for i = 1:Bits
        X0(i) = TABLE00to16_48B(Index, i + 2*Bits);
        Y0(i) = TABLE00to16_48B(Index, i + 3*Bits);
        ZT(i) = TABLE00to16_48B(Index, i + 1*Bits);
    end
    [Z0, OF]     = BinAdd(Z0, ZT, Bits);
    [X0, Y0, Z0] = HCPipe(X0, Y0, Z0, 5, 12, Bits);
end


XN = BtoD(X0, Bits);
YN = BtoD(Y0, Bits);
ZN = BtoD(Z0, Bits)*180/pi;
```

Appendix BB

TLCordic.m

```
function [X0, Y0, Z0, XD, YD, ZD] = TLCordic(Angle, Bits)

%TLCORDIC - Table Look-up CORDIC Iteration for calculating sine and cosine
%
%        - Angle -> The angle (in degrees) being calculated
%        - Bits  -> The number of bits in the array
%
%        - X0    -> The final X value (Binary array) representing cosine
%                       in two's complement
%        - Y0    -> The final Y value (Binary array) representing sine
%                       in two's complement
%        - Z0    -> The final Z value (Binary array) representing the
%                       residual angle in radians in two's complement
%        - XD    -> The final X value (decimal) representing cosine
%        - YD    -> The final Y value (decimal) representing sine
%        - ZD    -> The final Z value (decimal) representing the
%                       residual angle in degrees
%
%     Copyrighted by
%     Jason T. Arbaugh
%     December 2004
%

global TABLE00to08_08B;
global TABLE00to16_16B;
global TABLE00to16_24B;
global TABLE00to16_32B;
global TABLE00to16_48B;
global TABLE00to16_64B;
global TABLE16to24_24B;
global TABLE16to32_32B;
global TABLE16to32_48B;
global TABLE16to32_64B;
global TABLE32to48_48B;
global TABLE32to48_64B;
global TABLE48to64_64B;

if ( Angle < 0 )
    Sign = -1;
    Angle = abs(Angle);
else
    Sign = 1;
end

Z0  = DtoB(Angle*pi/180, Bits);

if ( Bits == 8 )
    Index = BtoIndex(Z0, 1, 8) + 1;
    for i = 1:Bits
```

```
        X0(i) = TABLE00to08_08B(Index, i       );
        Y0(i) = TABLE00to08_08B(Index, i + Bits);
    end
end

if ( Bits == 16 )
    Index = BtoIndex(Z0, 1, 16) + 1;
    for i = 1:Bits
        X0(i) = TABLE00to16_16B(Index, i       );
        Y0(i) = TABLE00to16_16B(Index, i + Bits);
    end
end

if ( Bits == 24 )
    Index = BtoIndex(Z0, 1, 16) + 1;
    for i = 1:Bits
        X0(i) = TABLE00to16_24B(Index, i       );
        Y0(i) = TABLE00to16_24B(Index, i + Bits);
    end

    Index = BtoIndex(Z0, 17, 24) + 1;
    for i = 1:Bits
        XLookUp(i) = TABLE16to24_24B(Index, i       );
        YLookUp(i) = TABLE16to24_24B(Index, i + Bits);
    end

    XCos = BinMult(X0, XLookUp, Bits);
    XSin = BinMult(X0, YLookUp, Bits);
    YCos = BinMult(Y0, XLookUp, Bits);
    YSin = BinMult(Y0, YLookUp, Bits);

    X0 = BinSub(XCos, YSin, Bits);
    Y0 = BinAdd(XSin, YCos, Bits);
end

if ( Bits == 32 )
    Index = BtoIndex(Z0, 1, 16) + 1;
    for i = 1:Bits
        X0(i) = TABLE00to16_32B(Index, i       );
        Y0(i) = TABLE00to16_32B(Index, i + Bits);
    end

    Index = BtoIndex(Z0, 17, 32) + 1;
    for i = 1:Bits
        XLookUp(i) = TABLE16to32_32B(Index, i       );
        YLookUp(i) = TABLE16to32_32B(Index, i + Bits);
    end

    XCos = BinMult(X0, XLookUp, Bits);
    XSin = BinMult(X0, YLookUp, Bits);
    YCos = BinMult(Y0, XLookUp, Bits);
    YSin = BinMult(Y0, YLookUp, Bits);

    X0 = BinSub(XCos, YSin, Bits);
    Y0 = BinAdd(XSin, YCos, Bits);
end
```

```
if ( Bits == 48 )
    Index = BtoIndex(Z0, 1, 16) + 1;
    for i = 1:Bits
        X0(i) = TABLE00to16_48B(Index, i       );
        Y0(i) = TABLE00to16_48B(Index, i + Bits);
    end

    Index = BtoIndex(Z0, 17, 32) + 1;
    for i = 1:Bits
        XLookUp(i) = TABLE16to32_48B(Index, i       );
        YLookUp(i) = TABLE16to32_48B(Index, i + Bits);
    end

    XCos = BinMult(X0, XLookUp, Bits);
    XSin = BinMult(X0, YLookUp, Bits);
    YCos = BinMult(Y0, XLookUp, Bits);
    YSin = BinMult(Y0, YLookUp, Bits);

    X0 = BinSub(XCos, YSin, Bits);
    Y0 = BinAdd(XSin, YCos, Bits);

    Index = BtoIndex(Z0, 33, 48) + 1;
    for i = 1:Bits
        XLookUp(i) = TABLE32to48_48B(Index, i       );
        YLookUp(i) = TABLE32to48_48B(Index, i + Bits);
    end

    XCos = BinMult(X0, XLookUp, Bits);
    XSin = BinMult(X0, YLookUp, Bits);
    YCos = BinMult(Y0, XLookUp, Bits);
    YSin = BinMult(Y0, YLookUp, Bits);

    X0 = BinSub(XCos, YSin, Bits);
    Y0 = BinAdd(XSin, YCos, Bits);
end

if ( Bits == 64 )
    Index = BtoIndex(Z0, 1, 16) + 1;
    for i = 1:Bits
        X0(i) = TABLE00to16_64B(Index, i       );
        Y0(i) = TABLE00to16_64B(Index, i + Bits);
    end

    Index = BtoIndex(Z0, 17,32) + 1;
    for i = 1:Bits
        XLookUp (i) = TABLE16to32_64B(Index, i       );
        YLookUp(i) = TABLE16to32_64B(Index, i + Bits);
    end

    XCos = BinMult(X0, XLookUp, Bits);
    XSin = BinMult(X0, YLookUp, Bits);
    YCos = BinMult(Y0, XLookUp, Bits);
    YSin = BinMult(Y0, YLookUp, Bits);

    X0 = BinSub(XCos, YSin, Bits);
    Y0 = BinAdd(XSin, YCos, Bits);
```

```
    Index = BtoIndex(Z0, 33,48) + 1;
    for i = 1:Bits
        XLookUp(i) = TABLE32to48_64B(Index, i       );
        YLookUp(i) = TABLE32to48_64B(Index, i + Bits);
    end

    XCos = BinMult(X0, XLookUp, Bits);
    XSin = BinMult(X0, YLookUp, Bits);
    YCos = BinMult(Y0, XLookUp, Bits);
    YSin = BinMult(Y0, YLookUp, Bits);

    X0 = BinSub(XCos, YSin, Bits);
    Y0 = BinAdd(XSin, YCos, Bits);

    Index = BtoIndex(Z0, 49,64) + 1;
    for i = 1:Bits
        XLookUp(i) = TABLE48to64_64B(Index, i       );
        YLookUp(i) = TABLE48to64_64B(Index, i + Bits);
    end

    XCos = BinMult(X0, XLookUp, Bits);
    XSin = BinMult(X0, YLookUp, Bits);
    YCos = BinMult(Y0, XLookUp, Bits);
    YSin = BinMult(Y0, YLookUp, Bits);

    X0 = BinSub(XCos, YSin, Bits);
    Y0 = BinAdd(XSin, YCos, Bits);
end

if ( Sign == -1 )
    for i = 1:Bits
        if ( Y0(i) == 1 )
            Y0(i) = 0;
        else
            Y0(i) = 1;
        end
    end

    for i = 1:Bits-1
        OneArray(i) = 0;
    end
    OneArray(Bits) = 1;

    [Y0, OF] = BinAdd(Y0, OneArray, Bits);
end

XD = BtoD(X0, Bits);
YD = BtoD(Y0, Bits);
ZD = BtoD(Z0, Bits)*180/pi;
```

Bibliography

[1] Chow, Timothy Y., "What is a Closed Form Number?," *The American Mathematical Monthly*, Vol. 106, pp. 440 – 448, 1999.

[2] McCartney, Scott, *ENIAC, the Triumphs and Tragedies of the World's First Computer*, New York: Walker, 1999.

[3] Cochran, D., "Algorithms and Accuracy in the HP35," *Hewlett Packard Journal*, Vol. 23, pp. 10 – 11, 1972.

[4] Rheinstein, John, "Simple Algorithms for Calculating Elementary Functions," BYTE, pp. 142 – 145, August 1977.

[5] Cavallaro, Joseph .R. and Franklin T. Luk, "CORDIC Arithmetic for a SVD Processor," *Proceedings of the 8th Symposium on Computer Arithmetic*, pp. 271 – 290 , 1987.

[6] Lin, Hai Xiang and Henk J. Sips, "On-Line CORDIC Algorithms," *IEEE Transactions on Computers*, Vol. 39, pp. 1038 – 1052, 1990.

[7] Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics – Principles and Practice: Second Edition in C*, Reading, MA: Addison-Welsey, 1995.

[8] Gallier, Jean, *Curves and Surfaces in Geometric Modeling – Theory and Algorithms*, San Francisco, CA: Morgan Kaufmann, 2000.

[9] Watt, Alan and Fabio Policarpo, *3D Games – Real-Time Rendering and Software Technology*, New York: Addison-Welsey, 2001.

[10] Despain, Alvin M., "Fourier Transform Computers Using CORDIC Iterations," *IEEE Transactions on Computers*, Vol. C-23, pp. 993 – 1001, 1984.

[11] Hu, Yu Hen, and S. Naganathan. "A Novel Implementation of Chirp-Z Transform Using a CORDIC Processor," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. 38, pp. 352 – 354, 1990.

[12] Hu, Yu Hen, "A Forward Angle Recoding CORDIC Algorithm and Pipelined Processor Array Structure for Digital Signal Processing," *Digital Signal Processing*, Vol. 3, pp. 2 – 15, 1993.

[13] Gibson, Jerry, *Principles of Digital and Analog Communications: Second Edition*, New York: Macmillan Publishing Company, 1993.

[14] Sklar, Bernard, *Digital Communications – Fundamentals and Applications*, Englewood Cliffs, NJ: Prentice Hall, 1988.

[15] Ziemer, Rodger E., Roger L. Peterson, *Introduction to Digital Communication: Second Edition*, Upper Saddle River, NJ: Prentice Hall, 2001.

[16] Terre, M. and M. Bellanger, "Systolic QRD-Based Algorithms for Adaptive Filtering and Its Implementation," *Proceedings 1993 IEEE International Conference on Acoustics, Speech, and Signal Processing*, Vol. III, pp. 296 – 298, 1993.

[17] Vaidyanathan, P.P., "A Unified Approach to Orthogonal Digital Filters and Wave Digital Filters Based on the LBR Two-Pair Extraction," *IEEE Transactions on Circuits and Systems*, Vol. 32, pp. 673 – 686, 1985.

[18] Cutkosky, Mark R., *Robotic Grasping and Fine Manipulation*, Boston, MA: Kluwer Academic Publishers, 1985.

[19] Schwartz, Jacob T., Micha Sharir, and John Hopcroft, *Planning, Geometry, and Complexity of Robot Motion*, Norwood, NJ: Ablex Publishing, 1987.

[20] Zomaya, Albert Y., *Modeling and Simulation of Robot Manipulators: A Parallel Processing Approach*, River Edge, NJ: World Scientific, 1992.

[21] Harrison, John, Ted Kubaska, Shane Story, and Ping Tak Peter Tang, "*The Computation of Transcendental Functions on the IA-64 Architecture*," Intel Technology Journal, Q4, 1999.

[22] Briggs, Henry, *Arithmetica Logarithmica*, Londini: Excudebat Gvlielmvs Iones, 1624.

[23] Bruce, Ian, http://www-gap.dcs.st-and.ac.uk/~history/Miscellaneous/Briggs/index.html.

[24] Volder, Jack E., "The CORDIC Trigonometric Computing Technique," *IRE Transactions on Electronic Computers*, Vol. , pp. 330 – 334, 1959.

[25] Parhami, Behrooz, *Computer Arithmetic – Algorithms and hardware Design*, New York: Oxford University Press, 2000.

[26] Muller, Jean-Michel, *Elementary Functions – Algorithms and Implementation*, Boston, MA: Birkhäuser, 1997.

[27] Ercegovac, Miloš D., "A General Hardware-Oriented Method for Evaluation of Functions and Computations in Digital Computers," *IEEE Transactions on Computers*, Vol. C-26, pp. 667 – 680, 1977.

[28] Knuth, Donald, *The Art of Computer Programming, Volume 2*, Reading, MA: Addison Wesley, 1973.

[29] Koren, Israel, and O. Zinaty, "Evaluating Elementary Functions in a Numerical Coprocessor Based on Rational Approximations," *IEEE Transactions on Computers*, Vol. 39, pp. 1030 – 1037, 1990.

[30] Lynch, Tom, Ashraf Ahmed, Mike Schulte, Tom Callaway, and Robert Tisdale, "The K5 Transcendental Functions," *Proceedings of the 12th Symposium on Computer Arithmetic*, July 1995.

[31] Ercegovac, Miloš D. and Tomás Lang, *Digital Arithmetic*, San Francisco, CA: Morgan Kaufmann, 2004.

[32] Oberman, Stuart and Michael Flynn, "Implementing Division and Other Floating-Point Operations: A System Perspective," *Proceedings of the International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics SCAN-95*, pp. 18 – 24, 1996.

[33] *IEEE Standard 754 for Binary Floating Point Arithmetic: IEEE Standard 754–1985*. IEEE, 1985.

[34] Brent, Richard P., "Fast Multiple-Precision Evaluation of Elementary Functions," *Journal of the Association for Computing Machinery*, Vol. 23, pp. 242 – 251, 1976.

[35] Volder, Jack, "The Birth of CORDIC," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, Vol. 25, pp. 101 – 105, June 2000.

[36] Hu, Yu Hen and S. Naganathan, "Angle Recoding Method for Efficient Implementation of the CORDIC Algorithm," *IEEE International Symposium on Circuits and Systems*, pp. 175 – 178, January 1989.

[37] Hu, Yu Hen and S. Naganathan, “An Angle Recoding Method for CORDIC Algorithm Implementation,” *IEEE Transactions on Computers*, Vol. 42, pp. 99 – 102, 1993.

[38] Timmermann, Dirk, Helmut Hhn, and Bedrich J. Hosticka, “Low Latency Time CORDIC Algorithms,” *IEEE Transactions on Computers*, Vol. 41, pp. 1010 – 1015, 1992.

[39] Hsiao, S.F. and J.M. Delosme, “Householder CORDIC Algorithms,” *IEEE Transactions on Computers*, Vol. 44, pp. 990 – 1001, 1995.

[40] Hsiao, Shen-Fu, “A High Speed Constant-Factor Redundant CORDIC Processor without Extra Correcting or Scaling Iterations,” *Proceedings of the 1999 IEEE International Symposium on Circuits and Systems*, Vol. 1, pp. 455 – 458, 1999.

[41] Hu, Yu Hen, “An Angle Recoding Method for CORDIC Algorithm Implementation,” *IEEE Transactions on Computers*, Vol. 42, pp. 99 – 102, 1993.

[42] Hu, Yu Hen and Homer H. M. Chern, “A Novel Implementation of CORDIC Algorithm Using Backward Angle Recoding (BAR),” *IEEE Transactions on Computers*, Vol. 45, pp. 1370 – 1378, 1996.

[43] Wang, Shaoyun and Earl E. Swartzlander, Jr., “Merged CORDIC Algorithm,” *IEEE International Symposium on Circuits And Systems*, Vol. 3, pp. 1988 – 1991, 1995.

[44] Wang, Shaoyun and Earl E. Swartzlander, Jr., “The Critically Damped CORDIC Algorithm for QR Decomposition,” *Conference Record of the*

Thirtieth Asilomar Conference on Signals, Systems and Computers, Vol. 2, pp. 908 – 911, 1996.

[45] Walther, J. S., "A Unified Algorithm for Elementary Functions," *Spring Joint Computer Conference Proceedings*, Vol. 38, pp. 379 – 385, 1971.

[46] Duprat, Jean and Jean-Michel Muller, "The CORDIC Algorithm: New Results for Fast VLSI Implementation," *IEEE Transactions on Computers*, Vol. 42, pp. 168 – 178, 1993.

[47] Avizienis, Algirdas, "Signed-Digit Number Representations for Fast Parallel Arithmetic," *IRE Transactions on Electronic Computers*, Vol. EC-10, p. 389, 1961.

[48] Ercegovac, Miloš D. and Tomás Lang, "Redundant and On-Line CORDIC: Application to Matrix Triangularization and SVD," *IEEE Transactions on Computers*, Vol. 39, pp. 725 – 740, 1990.

[49] Takagi, Naofumi, Tohru Asada, and Shuzo Yajima, "A Hardware Algorithm for Computing Sine and Cosine Using Redundant Binary Representation," *Transactions IECE Japan*, Vol. J69-D, pp. 841 – 847, 1986.

[50] Takagi, Naofumi, Tohru Asada, and Shuzo Yajima, "Redundant CORDIC Methods with a Constant Scale Factor for Sine and Cosine Computation," *IEEE Transactions on Computers*, Vol. 40, pp. 989 – 995, 1991.

[51] Phatak, Dhananjay S., "Double Step Branching CORDIC: A New Algorithm for Fast Sine and Cosine Generation," *IEEE Transactions on Computers*, Vol. 47, pp. 587 – 602, 1998.

[52] Wang, Shaoyun, Vincenzo Piuri, and Earl E. Swartzlander, "Hybrid CORDIC Algorithms," *IEEE Transactions on Computers*, Vol. 46, pp. 1202 – 1207, 1997.

[53] Tang, Ping Tak Peter, "Table-lookup Algorithms for Elementary Functions and Their Error Analysis," *Proceedings of the 10th Symposium on Computer Arithmetic*, pp. 232 – 236, June 1991.

[54] Priest, Douglas M., "Fast Table-Driven Algorithms for Interval Elementary Functions," *Proceedings of the 13th Symposium on Computer Arithmetic*, pp. 168 – 174, July 1997.

[55] Koren, Israel, *Computer Arithmetic Algorithms*, Englewood Cliffs, NJ: Prentice Hall, 1993.

[56] Hassler, Hannes and Naofumi Takagi, "Function Evaluation by Table Look-up and Addition", *Proceedings of the 12th Symposium on Computer Arithmetic*, pp. 10 – 16, July 1995.

[57] Schulte, Michael J., and James E. Stine, "Approximating Elementary Functions with Symmetric Bipartite Tables," *IEEE Transactions on Computers*, Vol. 48, pp. 842 – 847, 1999.

[58] Robertson, James E., "A Deterministic Procedure for the Design of Carry-Save Adders and Carry-Borrow Propagation Adders," Report 235, Department of Computer Science, University of Illinois, Urbana, 1967.

[59] Borovec, R.T., "The Logical Design of a Class of Limited Carry-Borrow Propagation Adders," Report 275, Department of Computer Science, University of Illinois, Urbana, 1968.

[60] Chow, Catherine Y. and James E. Robertson, "Logical Design of a Redundant Binary Adder," *Proceedings of the 4th Symposium on Computer Arithmetic*, pp. 109 – 114, 1978.

[61] Wang, Shaoyun, Vincenzo Piuri, and Earl E. Swartzlander, Jr., "A Unified View of CORDIC Processor Design," *IEEE 39th Midwest symposium on Circuits and Systems*, Vol. 2, pp. 852 – 855, 1996.

[62] Wang, Shaoyun, Vincenzo Piuri, and Earl E. Swartzlander, Jr., "The Hybrid CORDIC Algorithms," Internal Report no. 96-058, Department of Electronics and Information, Politecnice de Milano, Milano, Italy, 1996.

[63] King, Eric Jerome, *A Reduced Complexity Truncated Multiplier*, Master's Thesis, The University of Texas at Austin, 1998.

[64] Katz, Randy H., *Contemporary Logic Design*, Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1994.

Vita

Jason Todd Arbaugh, the son of Sharon Elizabeth Jarrell Staley and Wayne Lee Arbaugh, was born in Atlanta, Georgia on October 25, 1970. After completing his work at Midway High School, Hewitt, Texas in May of 1989, he entered Texas A&M University in College Station, Texas. During his undergraduate studies, he interned for Marathon Pipeline Company and Tandem Computers. He received the degrees of Bachelor of Science in Electrical Engineering and Bachelor of Science in Computer Engineering from Texas A&M University in May 1994 with support from the Jean and Bernard C. Richardson `40 President's Endowed Scholarship and a National Eagle Scout Association Scholarship. In June 1994, he started work at Advanced Micro Devices in Austin, Texas as a Product and Test Engineer. In June 1996, he entered the Graduate School of The University of Texas at Austin while working full time. He received the degree of Master of Science in Engineering from the University of Texas at Austin in May 1998. In September 1999, he was accepted into the doctoral program of the Electrical and Computer Engineering Department at the University of Texas at Austin. In January 2000 he left Advanced Micro Devices as a Senior Analog Designer in order to join Cirrus Logic as a Senior Systems Engineer in their Communications Division.

Permanent Address: Jason T. Arbaugh
20707 Henry Avenue
Lago Vista, TX 78645

Jason_Arbaugh@yahoo.com

This dissertation was typed by the author.