The CORDIC Algorithm:
An Area-Efficient Technique for FPGA-Based Artificial Neural Networks

A Thesis
presented to the Faculty of
California Polytechnic State University
San Luis Obispo

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Electrical Engineering

by
Tim McLenegan
November 2006

Authorization for Reproduction of Master's Thesis

I grant permission for the reproduction of this thesis in its entirety or any of its parts, without further authorization from me.

_____

Signature (Timothy McLenegan)

_____

Date

Approval Page

Title: The CORDIC Algorithm: An Area-Efficient Technique for FPGA-Based Artificial Neural Networks

Author: Timothy McLenegan

Date Submitted: November 28, 2006

Dr. Lynne Slivovsky	_____
Advisor and Committee Chair	Signature
Dr. Albert Liddicoat	_____
Committee Member	Signature
Dr. Jane Zhang	_____
Committee Member	Signature

Abstract

An artificial neural network is a computing technique that allows problems to be solved that would otherwise require an overly complex procedural algorithm. By designing a large network of computing nodes based on the artificial neuron model, new solutions can be developed to problems relating fields such as image and speech recognition.

This thesis presents the CORDIC algorithm as a computing technique that is capable of supporting an artificial neural network in programmable hardware such as FPGAs. The algorithm is presented in depth, including a derivation and precision analysis. The designs of a parallel and bit-serial implementation are analyzed with respect to their ability to support a large neural network. Simulations demonstrating the operation of the unit follow a breakdown of the subcomponents in each design. It is shown that the small resource requirements of the CORDIC algorithm allow for many instances in an FPGA, allowing the parallelism inherent to an artificial neural network to be maintained.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

Traditional design techniques typically require a systems designer to formulate a mathematical model of the system, and then design an algorithm to operate on the system inputs in accordance with the model. In general, this approach works well, but there are some applications where either the mathematical model is difficult to derive, or the algorithm is too complex to be implemented in a cost-effective manner. Image and speech processing are two such problems that people intuitively understand but require complicated mathematical models that in turn require a good deal of computing power to operate [10].

In these situations, an artificial neural network can be employed. Rather than deriving an algorithm for analyzing the image or speech pattern, the parameters of the network are tweaked in a process called training until the desired outputs are obtained from the network. As will be shown in Chapter 2, networks are massively parallel networks of small computing nodes called "neurons." This parallelism makes neural networks naturally suited to hardware implementations. A software implementation running on a single microprocessor can only simulate parallelism; only in a hardware implementation can the real-time benefits of this parallelism be achieved.

Programmable hardware devices such as FPGAs are particularly suited as platforms for implementation, since the various parameters of the network occur during training. Programmable hardware devices also afford several other benefits to hardware designers by eliminating a lot of the difficulties that come with custom chip design. With a platform for these artificial neural networks chosen, a means of performing the computations in the interconnected nodes is needed. The number of these nodes can be large; the arithmetic

unit must be small enough so that it can be duplicated enough to maintain the parallelism inherent in the structure.

The CORDIC algorithm will be presented as a solution to that problem. The algorithm lends itself to a simple hardware implementation consisting only of basic addition and shifting operations. In addition to its small chip area footprint, the algorithm can compute a wide variety of functions ensuring that it can be used in a variety of ways within a system. In particular, the algorithm can compute the exponential function, e^x by summing together the values of $\sinh(x)$ and $\cosh(x)$, both of which are computed in parallel by the unit. The exponential function has an application as the transfer function in a specific type of neural network and is chosen as the function of study for this thesis.

Chapter 2 will describe neural networks in detail and describe how they operate. The structure of the network will be analyzed, as will the design of the neurons that comprise the network. Chapter 3 will study the design of programmable hardware devices—FPGAs in particular. The benefits they offer hardware designers will be presented, as will the basic structures common to most implementations. Chapter 4 introduces the CORDIC algorithm. A general-purpose definition of the algorithm is presented, and proofs of convergence and precision are presented. Expansion into multiple computation domains allows for the computation of sine, cosine, square root, multiplication, hyperbolic sine, and hyperbolic cosine.

Chapter 5 and Chapter 6 study two implementations of the CORDIC algorithm and study how the components of the CORDIC processing unit fit into a Xilinx Spartan-3 FPGA. Simulations performed using the ModelSim software are presented. Chapter 7

then compares these two designs and shows how they can be used in a neural network to facilitate their implementation in a programmable device.

Chapter 2 Artificial Neural Networks

As electronic devices increasingly become more a part of society, the desire for devices to solve more complex problems is growing. Some problems have no easy algorithmic solution, or the problems themselves are not completely understood. It is in these cases that it is occasionally easier to develop a learning machine that can be taught what to do in a situation without needing to understand the underlying processes.

Artificial neural networks are one such technique. They are designed to emulate the human brain's ability to learn. The same structure can be used to solve many different problems, depending on the training methods employed.

2.1 Background

The structure of an artificial neural network is similar to that of the human brain. The human brain is made up of billions of cells called *neurons*. The computational power of the brain is a result of the large volume of neurons available, coupled with learning ability.

The neurons themselves have several different classifications and subcomponents. The computation process comes as a result of the interconnection of these different types of neurons. However, unlike the digital realm in which artificial neural networks reside, the neurons in the brain "form a process which is not binary, not stable, and not synchronous" [10].

Though artificial neural networks try to mimic the way in which the human brain operates, the methodology is not designed to replace the brain, especially since so much

of it is not understood. Rather, artificial neural networks represent a new technique for engineers to solve problems.

2.2 Artificial Neuron Model

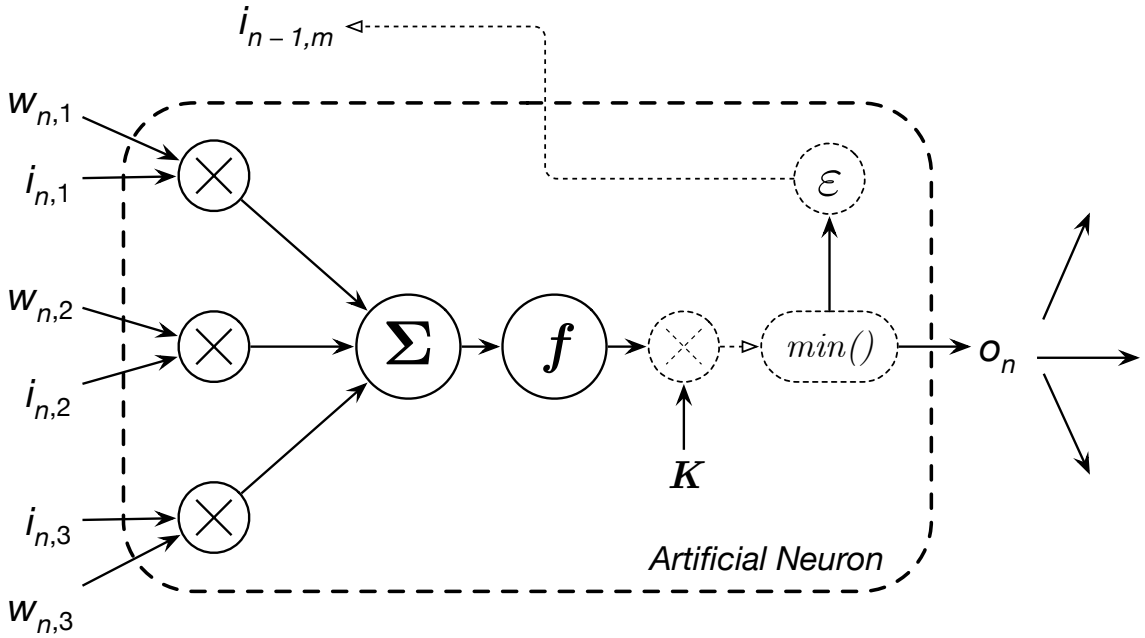

Figure 2.1—Components that comprise an artificial neuron, the basic building blocks of artificial neural networks.

Artificial neurons imitate the various features of biological neurons. Figure 2.1 shows a basic model of the various components that make up an artificial neuron. Each neuron has a set of inputs, labeled *i* in the figure, and a set of outputs denoted as *o* in the figure. The amount of these inputs and outputs can vary depending on the structure of the entire network. Outputs can be connected to multiple neurons.

2.2.1.1 Weighting Factors

In order to mimic the synaptic strengths of biological neurons, weighting factors (denoted *w* in Figure 2.1) are needed for the inputs to the artificial neuron. Each weight signifies the importance of their respective input in the processing function of the neuron. Inputs with larger weights will contribute more to the neural response than those with

lesser weights. The potential to learn is incorporated into the artificial neuron (and thus the network), by allowing the input weights to be adaptive coefficients. The adaptation process is performed in response to training sets of data, and depends on both the network's specific topology as well as the learning rule being applied.

2.2.2 Summation Function

The first step in the operation of an artificial neuron is the summation function. As the name implies, this is usually a summation of the weighted inputs to the neuron. The summation function can be more complex than a simple summation. Other functions used are minimum, maximum, majority, product, and other normalizing algorithms. Any function that operates on multiple inputs to produce a single output qualifies.

2.2.3 Transfer Function

After the inputs have passed through the summation function, they are then fed through a transfer function. This transfer function is usually a nonlinear function. One of the goals of artificial neural networks is to be able to provide nonlinear processing. However, the ability of a neural network to perform in a nonlinear fashion is dependant upon the transfer function of the individual neurons. By choosing a linear transfer function, the overall network would be limited to simple linear combinations of the inputs. Various transfer functions are typically used. A very common transfer function is the hyperbolic tangent. The hyperbolic tangent is a continuous function, as are its derivatives.

In a few cases, a uniform noise generator is added before the transfer function is applied. The output of this generator is referred to as the neuron's "temperature." The surrounding temperature can adversely affect the human brain, and by adding this

capability to artificial neural networks, the behavior of the network more closely emulates a human brain.

2.2.4 Scaling and Limiting

Implementation of this portion of the artificial neuron model is optional. The output of the transfer function is manipulated in order to lie within certain bounds. Scaling is performed first, followed by some sort of threshold function.

According to Anderson and McNeil [10], these components are usually used when explicitly simulating biological neuron models.

2.2.5 Error Function

The raw error of the network is the difference between the desired output and the actual output. The error function transforms this raw error to match the particular network architecture in use. If it is desired by the system architect to consider error in the network, then this component is included in the neuron model. In this case, propagation direction of this error is usually backwards through the network. The back-propagated value serves as the input to other neurons' learning functions.

2.2.6 Output Function

Outside the neuron model, but related to it is the output function. Normally the output of the neuron is equal to the output of the transfer function. When implemented, the output function allows for competition between the outputs of various neurons. Within a small "neighborhood" of neurons, a large output by one neuron will cause the output of a different neuron to diminish. In other words, the loudest neuron causes the other neurons to be quieter.

2.2.7 Learning Function

The learning function modifies the input weights of the neuron. Other names given to this function are the adaptation function, or learning mode. There are two main types of learning when dealing with neurons and neural networks. The first type, supervised learning, is a form of reinforcement learning and requires a teacher, usually in the form of training sets or an observer. Unsupervised learning is the other type, and is based upon internal criteria built into the network. The majority of neural networks utilize the supervised learning method, as unsupervised learning is currently more in the research realm. The processes used to train a network are discussed in 2.4.

2.3 Artificial Neural Network Structure

Artificial neural networks function as parallel distributed computing networks. Each node in the network is an artificial neuron. These neurons are connected together in various architectures for specific types of problems. It is important to note that the most basic function of any artificial neural network is its architecture. The architecture, along with the algorithm for updating the input weights of the individual neurons, determines the behavior of the artificial neural network. Neurons are typically organized into layers with connections between neurons existing across layers, but not within. Each neuron within each layer is often fully connected to all neurons in the associated layer. This can lead to a vast amount of connections existing within the network, even with relatively few neurons per layer. Figure 2.2 shows a simple neural network containing 3 layers. In this case, the layers are not fully connected.

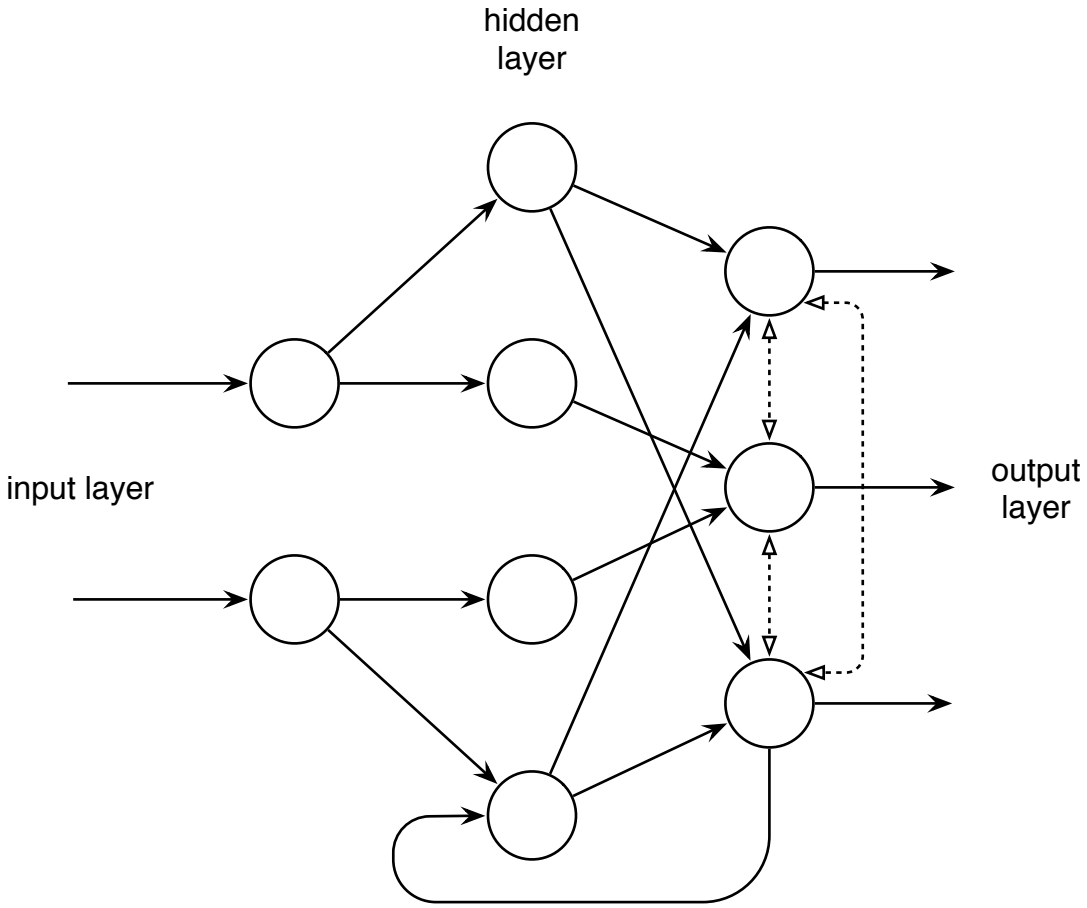

Figure 2.2—A basic artificial neural network.

2.3.1 Input Layer

Individual neurons are used for each input of an artificial neural network. These inputs could be collected data, or real world inputs from physical sensors. Pre-processing of the inputs can be done to speed up the learning process of the network. If the inputs are simply raw data, then the network will need to learn to process the data itself, as well as analyze it. This would require more time, and possibly an even larger network than with processed inputs.

2.3.2 Hidden Layers

The input layer is typically connected to a hidden layer. Multiple hidden layers may exist, with the inputs of each hidden layer's neurons often being fully connected to the

outputs of the previously layer's neurons. Hidden layers were given that name due to the fact that they do not see any real world inputs nor do they give any real world outputs. They are fed by the input layer's outputs, and feed the output layer's inputs. The number of neurons within each of the hidden layers, as well as the number of hidden layers themselves, determines the complexity of the system. Choosing the right amount for each is a major part of designing a working neural network. Figure 2.2 shows one hidden layer, but other network architectures can have multiple hidden layers.

2.3.3 Output Layer

Each neuron within the output layer receives the output of each neuron within the last hidden layer. The output layer provides real world outputs. These outputs could go to another computer process, a mechanical control system, or maybe saved into a file for analyzing. Like the output function of an individual neuron, the output layer may participate in some sort of competition between outputs. This lateral inhibition can be seen in Figure 2.2 as the dotted lines connecting the output neurons. In addition, the outputs may also be fed back into previous neurons to assist the learning process. In Figure 2.2, the result from an output neuron is fed back into a neuron in the hidden layer.

2.4 Learning Modes

A variety of different learning modes exist for determining how and when the input weights of the individual neurons are updated within a network. The types of learning are either supervised or unsupervised. As stated earlier, the unsupervised learning is currently the most unknown type of learning. The learning rate of a network drastically effects the its performance.

2.4.1 Supervised Learning

Learning in a supervised mode starts with a comparison of the network's generated outputs and the desired outputs. Input weights of each neuron are adjusted to minimize any differences found. This process is repeated until the network is deemed to be accurate enough. After the training phase, the neurons' weights are typically frozen, which allows the network to be used reliably. To better adapt to slight variations, the learning rate can be lowered. One of the most important things to do when training a network is to carefully choose the data used for training. Typically data is separated into a training set and a much smaller test set. The training set is used to train the network to perform a task. The test set is used to verify that the network is able to generalize what it has learned to slight variations. Without this separation of data sets, one would not be able to know if the network simply memorized the data set or not.

2.4.2 Unsupervised Learning

Unsupervised learning is performed without any form of external reinforcement. This learning mode represents a sort of end goal for systems designers. Using this method, the system teaches itself by itself. The network contains within itself a method of determining when its outputs are not what they should be. This method of learning is not nearly as well understood as the supervised method. It requires that the network learn online. Current work has been limited to self-organizing maps, which learn to classify incoming data. Further developments with this type of learning would have uses in many situations where adaptation to new inputs is required regularly.

2.4.3 Learning Rates

The learning rate of a network is determined by many factors. Network architecture, size and complexity play a big role in the speed at which the network learns. Another factor that affects the learning rate is the learning rule or rules employed. Slow and fast learning rates each have their pros and cons. A lower rate will obviously take longer to arrive at a minimum error at the output. A faster rate will arrive more quickly, but has a tendency to overshoot the minimum. Some learning rules use the best of both worlds, and start off with a high learning rate, and lower it gradually until a minimum is reached.

2.4.4 Common Learning Laws

Learning laws govern how the input weights of neurons within the network are modified. Typically the error at the output is propagated back through the various layers of the network, adjusting the weights as it goes. How the error is propagated back is the major difference. The following, all from [10], are some laws commonly used by network architects.

2.4.4.1 Hebb's Rule

Hebb's rule was the first general rule for updating weights. It states, "If a neuron receives an input from another neuron, and if both are highly active (mathematically have the same sign), the weight between the neurons should be strengthened." Hebb observed that biological neural pathways are strengthened each time they are used, and this rule is designed to simulate that effect. Most of the other rules build upon this fundamental law.

As an example of how this rule works, suppose a neural network is being trained to control the acceleration of a car. Suppose further that the network's inputs are the brake and gas pedal positions being as operated by a human driver. The acceleration and

deceleration of the car can be compared with the desired output of the driver. Now suppose the driver of the car wanted to slow down, and pushed the brake. If the output of the neural network was to decelerate, than any input weights, which a "decelerate" command was passed through, should be increased. This would positively reinforce the acceptable behavior of the network.

2.4.4.2 The Delta Rule

The delta rule is one of the most common learning rules, and is a variation of Hebb's Rule. It is also known by several other names, including the Widrow-Hoff Learning Rule and the Least Mean Square Learning Rule. It works by transforming the error at the output by the derivative of the transfer function. The result of this transformation is used to adjust the input weights associated with the previous layers outputs. The transformed error result is propagated back through all of the layers. Feed-forward, back-propagation networks use this method of learning.

2.4.4.3 The Gradient Descent Rule

The Gradient Descent rule is similar to the Delta Rule in that the derivative of transfer function modifies the output error. An additional proportional constant related to the learning rate is added to the modifying factor before the weights are adjusted. This method in its basic for is known to have a slow rate of convergence. Using a varying learning rate as was mentioned before can mitigate this.

2.4.4.4 Kohonen's Learning Law

This learning law is used for unsupervised networks. Teuvo Kohonen was inspired by learning in biological systems, and thus came up with the law. With this law, neurons compete for the opportunity to learn. The neuron with the largest output is the winner,

and gets to update its weights and possibly some of its neighbors. Usually the neighborhoods start large, and shrink as training progresses.

2.5 FPGA Implementations

Currently, most neural network research and testing is done using software simulators. Software implementations allow for easy analysis of network behaviors. Additionally, prediction-style problems in many cases do not require an embedded hardware application. However, hardware neural networks can offer many advantages.

An optimized hardware implementation can yield better performance than a software configuration running on a standard microprocessor. Additionally, a hardware implementation can allow for applications that would be difficult to achieve in a software set-up, such as with remote sensing applications. Designing for an FPGA also offers many advantages, as discussed in Chapter 3.

The ability of FPGAs to be re-programmed offers several specific advantages. Various techniques in the category of "density enhancement" aim to increase the amount of "effective circuit functionality per unit circuit area" [12]. One way to accomplish this is to separate out the various stages of the network's learning algorithm onto different regions of the FPGA. Another way is to use optimized constant-coefficient multipliers to handle the weighting calculations of the neuron's inputs. These calculations will be fast. In a training situation, these constant weights can be changed in under 69 μs [12].

FGPAs also allow for the hardware implementation of artificial neural networks that can dynamically adjust their topology. This allows for more complex learning algorithms to adjust the topology, resulting in a more sophisticated final network.

Hardware implementations raise new issues and considerations that are not present in software designs. A key consideration involves the numerical representation. Fundamentally, there are two ways that a number can be represented in any hardware device: fixed-point, and floating-point. As discussed in 4.7.1, CORDIC uses fixed-point notation, which is simply a scaled integer. Floating-point is the most common representation used in computing hardware due to the wide range of values that can be represented. In a hardware setting, especially a setting where the arithmetic units support a much more complex hardware architecture (e.g. an artificial neural network), the chip area requirements of a floating-point unit are prohibitive. Nichols showed in [15] that it is impractical to implement a neural network in an FPGA using floating-point weights. Fixed-point units are much more attractive in that they allow for more chip area to be devoted to the actual neural network. There is a large body of ongoing research devoted to the use fixed-point weights in neural networks, which eliminate the need for floating-point hardware.

Along with choosing a numerical representation comes the decision of how much precision the weights should have. As with any system, greater precision results in increased computation time, greater chip area requirements and power consumption. For any problem, in hopes of eliminating some of the problems associated with a greater precision, the so-called “minimum precision” must be determined. For some applications [16], it may be as few as 16 bits.

Draghici [17] performed a more extensive study of precision, wherein the range and precision of the neuron weights were treated as variables. The aim was to derive a general formula that designers could use to find the optimal weight format for their systems. It

was determined the minimum number of bits to represent one fixed-point weight is $\lceil \log(2p+1) \rceil$, where the values for the weights is in the range $[-p, p]$.

2.6 CORDIC

Any neural network implementation needs an arithmetic unit to perform the various calculations necessary in a neural network. Neural networks are large and rely on the parallel computing power of their neurons for efficiency. The large quantity of neurons means that neural networks have significant hardware resource requirements. Consequently, only very small arithmetic units can be used in a hardware implementation.

The CORDIC algorithm satisfies that requirement. CORDIC is capable of computing many of functions that are used in neural networks. The fact that it can switch between function sets so easily means that it can be used in a network that utilizes different transfer functions in different layers of the network.

The most common transfer functions used in neural networks are sigmoid functions such as the hyperbolic tangent. When in the proper computation mode, CORDIC can be used to compute this function with the help of a binary divider, using the identity $\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$. CORDIC is also capable of performing this division when in linear vectoring mode, but the division is slow and would double computation time. CORDIC can also perform multiplication when in the linear rotation mode, which can compute the weights for the neuron inputs, though multiplication units optimized for multiplication by a constant are better suited when the weights have already been determined.

This thesis studies the implementation of the exponential function in an FPGA using CORDIC. Though not as common as hyperbolic tangent, there are applications where the exponential function is used as a transfer function. Wu and Batalama used the exponential function as a transfer function in their implementation of a neural network that acted as an associative memory. [18] Similarly, Halgamuge used the exponential function in the implementation of a defuzzication approximator. [19].

Chapter 3 FPGAs

Any designer faces a number of different onto which the system can be implemented. Each platform comes with a set of benefits and tradeoffs. The nature of the system in addition to its environment need to be taken into account when choosing a platform.

3.1 Design Decisions

One approach is to realize the design as a software program and implement it using a microprocessor of some kind. There are numerous architectures that can be chosen, each with its own benefits. It can be easy to find one that is suited for the design. This approach does not have any special manufacturing costs associated with it; the only cost comes for the purchase of the processors. However, this technique is best suited for applications that are mostly sequential in nature. For applications that are parallel in nature, a custom hardware-based design can offer much in the way of performance benefits.

Once the designer has elected to use a custom hardware implementation, the type of implementation must then be chosen. At the lowest level, a fully custom integrated circuit can be designed at the transistor level. This offers the designer the most amount of flexibility, however design costs can be higher due to the complexity of the implementation and fabrication of the design. An Application-Specific Integrated Circuit (ASIC) can alternatively be employed to realize the design. Rather than designing at the transistor level, the designer is presented with a library of logic gates with different performance and power characteristics that can all be used in the design. Since the design

process is simpler, the cost of implementing the design can be less than a fully custom solution.

As an alternative to the static hardware solutions are a number of programmable logic devices that can be used. Programmable logic offers much of the benefits of using a microprocessor. Devices come already fabricated, eliminating the cost and time associated with fabricating an ASIC or other custom design. Also, since the devices are programmable, a new design can be implemented on the device without having to re-fabricate new chips. This makes these devices excellent choices for design prototyping, and for applications where it is desirable to update to the design after it has been deployed.

There are various architectures for programmable devices, that allow for designs of varying sizes to be implemented. At the smallest level are simple programmable logic devices (PLDs) that allow the implementation of two-level logic functions. Complex PLDs (CPLDs) were developed to allow for the implementation of larger logic functions. These consist of a collection of PLD blocks using a programmable interconnect to allow deep functions to be implemented.

PLDs are still very limited in the number of logic gate equivalents that can be made. Gate arrays offer significantly more hardware resources than PLDs, and often come with other hardware blocks, such as RAM or a processor to complement a hardware design. Gate arrays can be implemented statically with the design frozen on silicon or in a programmable package. Field programmable gate arrays (FPGAs) allow for programming after the chip has been fabricated.

3.2 FPGA Structure

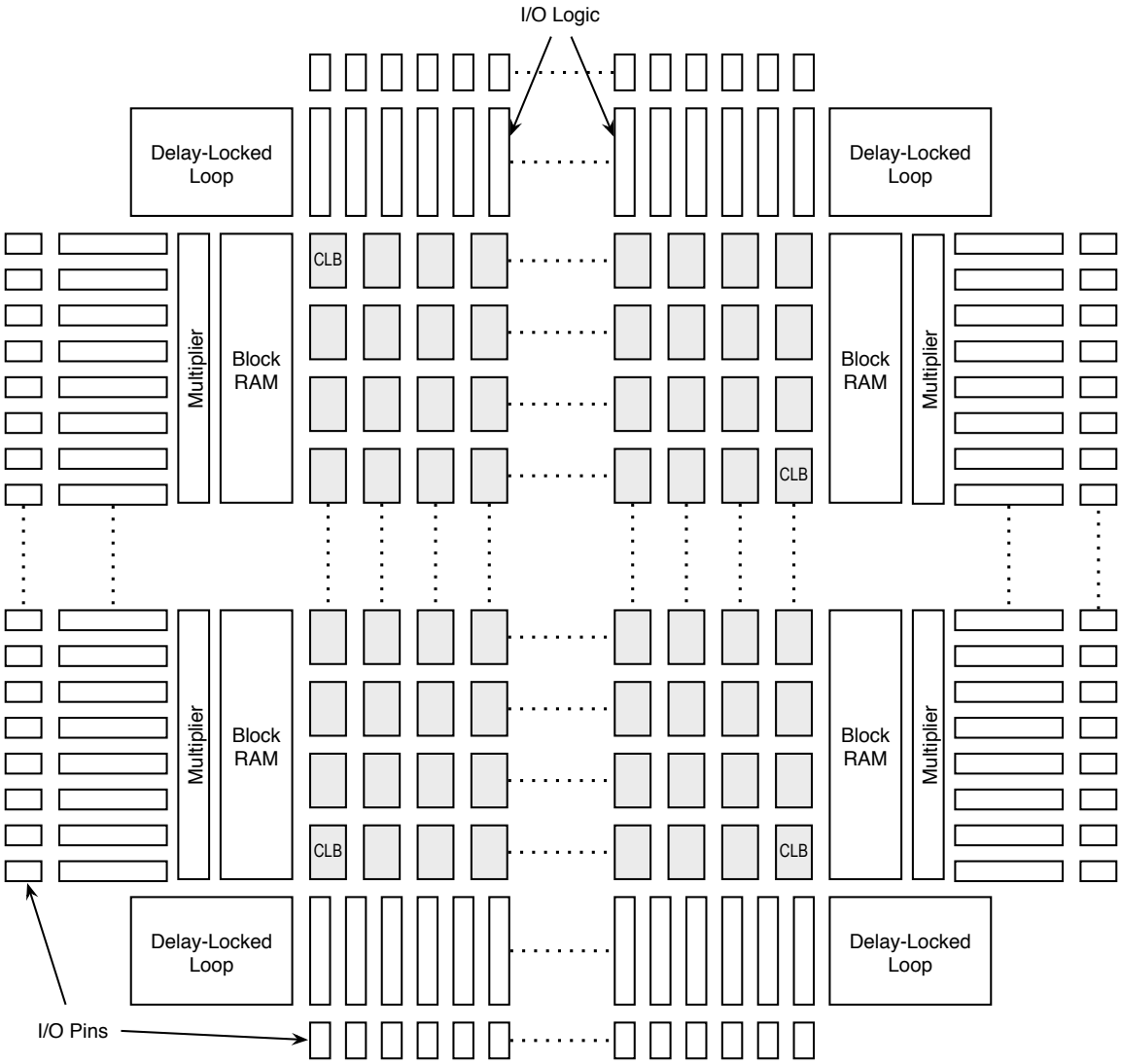

Figure 3.1—Internal structure of a Xilinx Spartan-3 FPGA [8].

The standard FPGA has three main components: configurable logic blocks, I/O blocks, and a programmable interconnect. Each component plays a role in allowing for the wide variety of hardware designs that can be implemented in FPGAs. Figure 3.1 shows a block-level diagram of the various components that make up an FPGA.

The configurable logic blocks (CLBs) are where the actual logic functions specified by the designer are implemented. In most FPGA designs, the configurable logic blocks

are made up of a number of lookup tables (LUTs). LUTs can be implemented as a multiplexer and can implement any *n*-variable Boolean function. The *n* inputs to the function can be used as the control lines on the multiplexer, and the 2^n outputs for the function are data inputs to the multiplexer. These inputs are generally implemented as one-bit SRAM cells and are what make the logic blocks programmable. Each LUT describes the complete truth table for the function, rather than the relationship between the inputs and outputs in normal Boolean circuits.

A configurable logic block will contain multiple LUTs with multiplexers selecting between the outputs of the inner LUTs. The output of the final multiplexer becomes the output of the logic block. This allows each configurable logic block to implement a function of more variables than is allowed by the individual LUTs themselves.

FPGAs contain a variety of interconnections. Usually, there is a fast interconnect that connects adjacent CLBs. This allows functions that require multiple CLBs to perform more efficiently by placing the subfunctions in adjacent CLBs. For chip-wide routing, FPGAs include a programmable interconnect that is usually implemented as a switching matrix using six pass transistors that all possible connections vertically, horizontally, and diagonally.

The I/O (input/output) blocks (IOBs) provide the interface between the hardware design and the outside world. Each IOB contains logic specific the input and output of signals. Usually, a three-state buffer is used to configure the pin that the block is connected to as an input pin, and output pin, or a bidirectional pin. Additionally, the block contains logic that specifies the current state of the block (i.e. whether it is an input or an output pin). The input logic is responsible for buffering the input signal and

allowing the signal to be properly captured so it is presentable to the rest of the FGPA. The output logic allows for optional buffering in a flip-flop, or the signal can be directly sent to the pin.

Before any signal leaves or enters the chip, it passes through an electronic interface component that determines the electrical characteristics of the signal and control the sampling of any input value.

Besides these three basic components, many newer FPGAs provide additional facilities to make the design process simpler. An early addition was large arrays of block SRAM that can be used for data storage. Chaining together CLBs for simple data storage can be cumbersome and area inefficient, so using the block SRAM can leave more chip area for the hardware design. Fast carry logic is often available which allows fast adder designs such as carry lookahead adders to be implemented. Some designers are taking that idea a step forward and implementing static multipliers or other units dedicated to performing simple arithmetic operations. Newer designs can include processors such as the PowerPC, allowing software and hardware designs to exist on the same chip.

3.3 Xilinx Spartan-3

The Xilinx Spartan-3 is the platform used for the designs discussed in the following chapters. The Spartan-3 uses SRAM cells to store the programming information. Since the SRAM cells lose their state when the power supply is turned off, they must be reprogrammed each time the chip is powered on. This requires the program to be stored in a nonvolatile format that can be translated onto the FPGA.

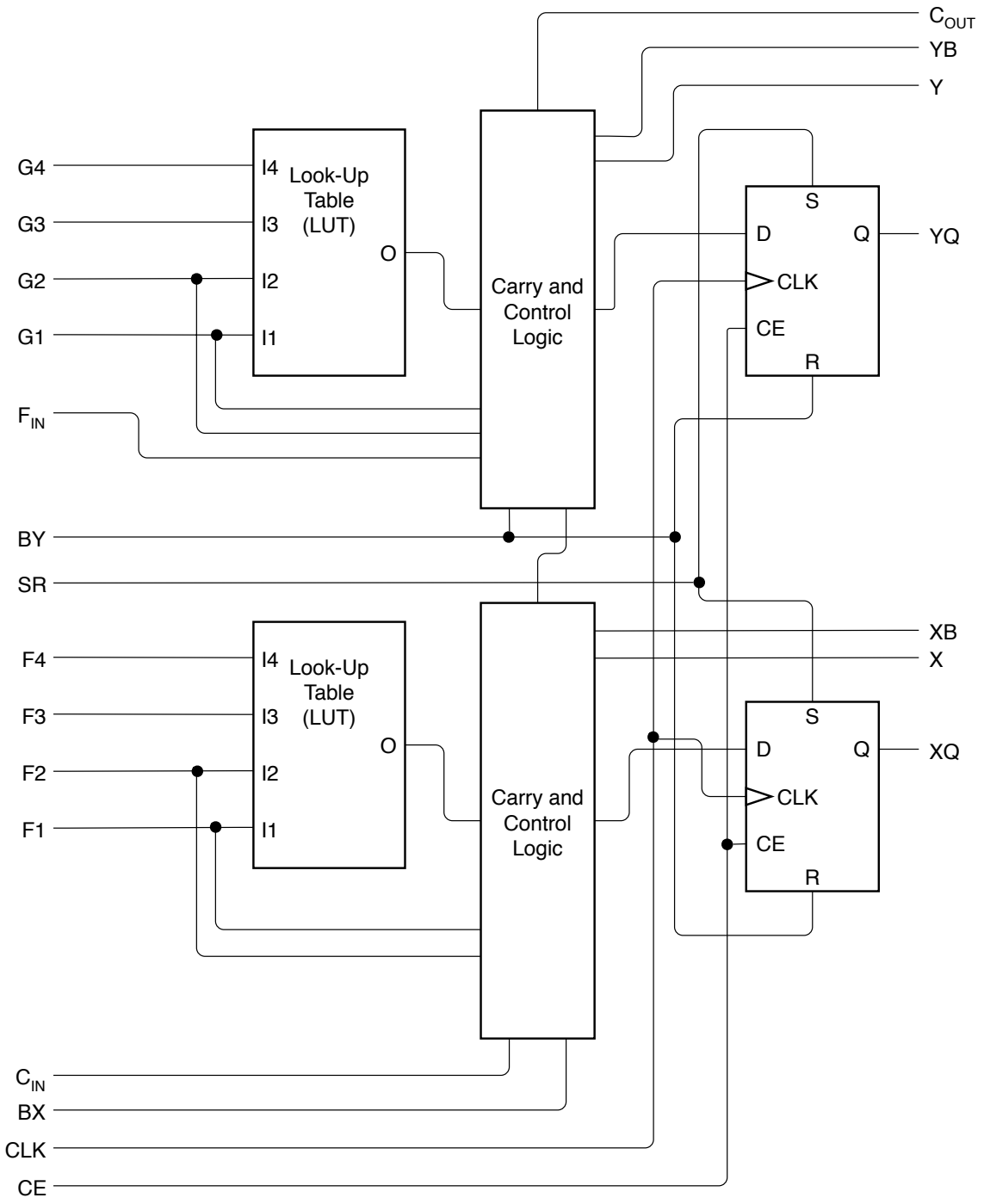

Figure 3.2—Block diagram of a slice in a Spartan-3.

The Spartan-3 CLBs are broken up into two *slices*. Figure 3.2 shows what a slice looks like. Each slice contains two two-variable LUTs and a multiplexer that selects between the two. The output of each slice can be any function of five variables, or for some restricted functions, as many as nine variables can be used. An additional

multiplexer is used to select between the outputs of the two slices. This enables each CLB to operate on any function of six variables, or a restricted class of functions of up to 19 variables. Each slice also contains carry and control logic that facilitate the implementation of arithmetic functions. Figure 3.3 shows how the multiplexers are used to select between the two inner slices. Also, the CLB contains two storage elements that can be configured either as latches or flip-flops.

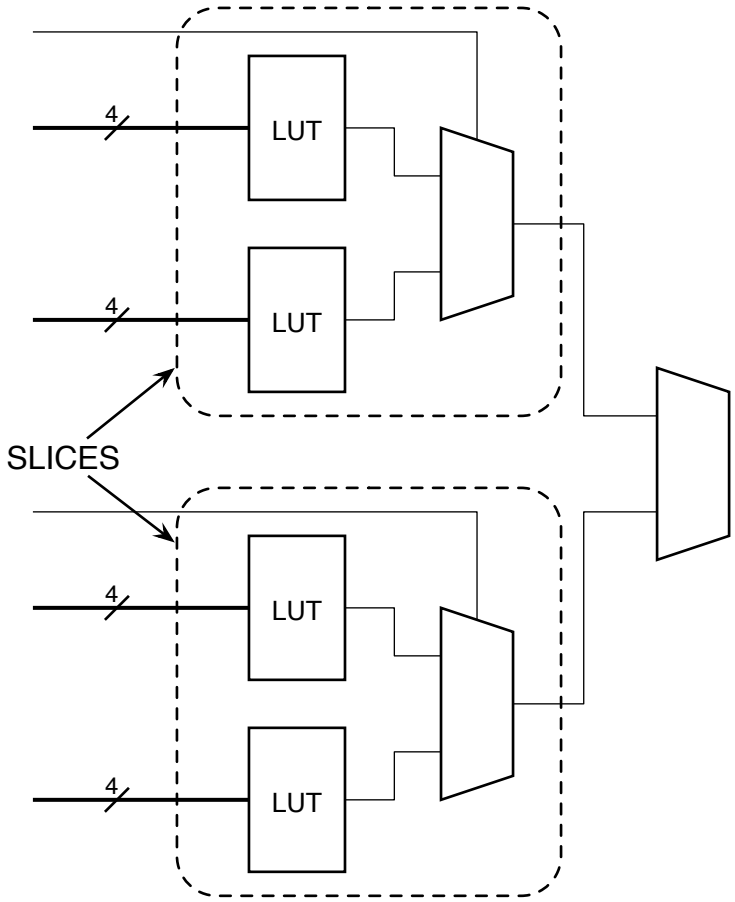

Figure 3.3—Block diagram of a Complex Logic Block (CLB) in a Spartan-3.

The Spartan-3 contains a number of block RAMs designed to efficiently store relatively large amounts of data. Each block contains 16K of data storage, and another 2K for optional parity checking. Each block uses SRAM cells for storage. Each block can

have a variety of organizations for any combination of data word size and parity option. Parity is done at the byte level, so data paths using 4 bytes will have 4 parity bits.

Adjacent to each block RAM is an 18×18 multiplier. The multipliers can be cascaded to support operands larger than 18 bits wide or to operate on more than 3 operands. When synthesizing a multiplication operation, the synthesizer can use either an asynchronous multiplier or a synchronous multiplier that includes a register. Each multiplier outputs a 36-bit wide product.

The Spartan-3 has four different levels of interconnection. The first type of interconnect is the Direct Line that connects adjacent CLBs. This includes connections that go inside the CLBs to connect the LUTs as well as fast connections between the CLBs to the immediate left and right. These lines are most frequently used to a longer interconnect, which in turn connects to a different direct line to the destination CLB. Long lines connect to every sixth CLB and have low capacitance to facilitate the transfer of high-frequency signals. Between the Direct Lines and the Long Lines in terms of capability are the Hex Lines and Double Lines. Hex Lines connect to every third CLB and Double Lines connect to alternating CLBs. Both types of interconnect offer a compromise between connectivity and high-frequency characteristics.

Chapter 4 The CORDIC Algorithm

The COrdinate Rotation DIgital Computer (CORDIC) algorithm is a technique that can be used to compute the value of trigonometric functions. CORDIC differs from other methods of computation because it can be easily implemented using only addition, subtraction and bit shift operations. It is not as fast as table-based methods, but it can use significantly less chip area, making it desirable for application where area is more important than performance.

4.1 Background

The CORDIC algorithm was first proposed by Volder in his paper "The CORDIC Trigonometric Computing Technique," published in 1959 [2]. It was originally intended to be used for real-time airborne computation, but has since found other applications. In 1971, Walther demonstrated that the algorithm's domain could also be expanded beyond computing trigonometric equations to also include hyperbolic and linear (multiply-add-fused, etc.) functions.

Walther presented a possible hardware implementation using three *n*-bit registers, three *n*-bit adder/subtractors, three shifters, and a small look-up table. Volder presented a serial design containing three *n*-bit registers, three 1-bit adder/subtractors, and a number of shift gates that specified which bits from the registers get fed into the adder/subtractors. This thesis will explore both implementations and analyze the capability of this algorithm to be used in an artificial neural network.

4.2 The Algorithm

The algorithm operates in one of two modes: Rotation or Vectoring. The two modes determine which set of functions can be computed using the algorithm. In Rotation mode, the *x*- and *y*- components of the starting vector are input, as well as an angle of rotation. The hardware then iteratively computes the *x*- and *y*-components of the vector after it has been rotated by the specified angle of rotation.

In Vectoring mode, the two components are input, and the magnitude and angle of the original vector are computed. This is accomplished by rotating the input vector until it is aligned with the *x*-axis. By recording the angle of rotation to achieve this alignment, we know the angle of the original vector. Once the algorithm is complete, the *x*-component of the vector is equal to the magnitude of the starting vector.

The CORDIC algorithm accomplishes its computations by using only adds, subtracts, and shifts. This is done by iteratively rotating the input vector and slowly converging on the final rotated vector. This is done in a series of well-defined steps. The first rotation step sees the vector rotated $\frac{\pi}{4}$ radians:

$$y_1 = \pm x_0 = R_0 \sin\left(\theta_0 \pm \tfrac{\pi}{4}\right)$$
$$x_1 = \mp y_0 = R_0 \cos\left(\theta_0 \pm \tfrac{\pi}{4}\right) \tag{4.1}$$

Where x_0 and y_0 represent the input vector aligned at the origin with magnitude R_0 and angle θ_0:

$$x_i = R_i \cos\theta_i$$
$$y_i = R_i \sin\theta_i \tag{4.2}$$

In each subsequent step, a new angle of rotation α_i is determined such that:

(4.3) $$\alpha_i = \tan^{-1}\left(2^{-i}\right), \text{ where } i > 0$$

This restriction is crucial in allowing the rotation calculations performed in each step to be accomplished using only an add (or subtract) and a shift.

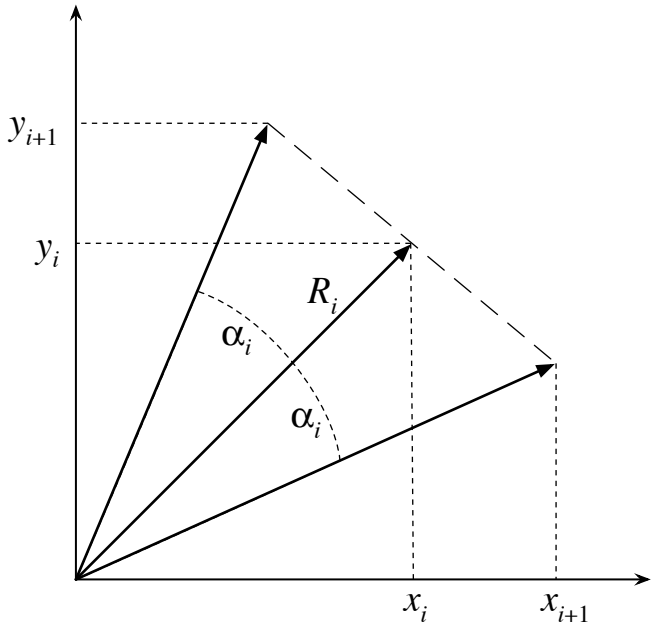

Figure 4.1—Step *i* in the CORDIC algorithm.

Figure 4.1 shows what each step in the CORDIC algorithm looks like. At each step, a decision is made whether to rotate the vector by $+\alpha_i$ or $-\alpha_i$. The outcome of both of these decisions is shown in the figure. The expression for the rotated vector in the (*i*+1)th step is:

(4.4) $$\begin{aligned} x_{i+1} &= \sqrt{1+2^{-2i}}\cos\left(\theta_i \pm \alpha_i\right) \\ y_{i+1} &= \sqrt{1+2^{-2i}}\sin\left(\theta_i \pm \alpha_i\right) \end{aligned}$$

When applying the restriction in (4.3), the shifting and adding become evident:

$$
\begin{aligned}
x_{i+1} &= \frac{1}{K_i}\left(x_i - d_i 2^{-i} y_i\right) \\
y_{i+1} &= \frac{1}{K_i}\left(y_i + d_i 2^{-i} x_i\right) \\
&\text{where } K_i = \sqrt{1+2^{-2i}},\ d_i = \pm 1
\end{aligned} \tag{4.5}
$$

K_i is the magnitude error term, and d_i corresponds to the rotation direction (+1 corresponds to a rotation away from the *x*-axis). The 2^{-i} terms in the top and bottom equations correspond to a left shift of y_i and x_i, respectively (when operating in base 2). This shifted value is then added (or subtracted) to the current value of the component. Volder referred this operation *cross-addition*. It is this cross-addition that enables the algorithm to be used effectively in digital hardware.

As illustrated in Figure 4.1, all rotation steps effect an increase in the magnitude of the input vector by a factor of $\sqrt{1-2^{-2i}}$ with each rotation. This error is introduced as a consequence of the algorithm's derivation from the Givens transform, which rotates a vector by a specified angle:

$$
\begin{aligned}
x' &= x\cos\phi - y\sin\phi \\
y' &= y\cos\phi + x\sin\phi
\end{aligned} \tag{4.6}
$$

These terms can be rearranged using the basic trigonometric identity $\tan\alpha = \dfrac{\sin\alpha}{\cos\alpha}$:

$$
\begin{aligned}
x' &= \cos\phi\left(x - y\tan\phi\right) \\
y' &= \cos\phi\left(y + x\tan\phi\right)
\end{aligned} \tag{4.7}
$$

Using the same restriction in (4.3), we get the same relationships as in (4.5). The error term is from the presence of the cosine term in (4.7), which is independent of the rotation direction, since cosine is symmetric about the rotation direction, since cosine is

symmetric about the *y*-axis. Moreover, this error accumulates with each step, so if the number of iterations is set, the total error in the algorithm is independent of the input angle.

If the first rotation is given by

$$\begin{aligned} x_1 &= \sqrt{1+2^{-2(1)}}R_0\cos(\theta_0 + d_1\alpha_1) \\ y_1 &= \sqrt{1+2^{-2(1)}}R_0\sin(\theta_0 + d_1\alpha_1) \end{aligned} \tag{4.8}$$

Then the second rotation is given by

$$\begin{aligned} x_2 &= \sqrt{1+2^{-2(1)}}\sqrt{1+2^{-2(2)}}R_0\cos(\theta + d_0\alpha_0 + d_1\alpha_1) \\ y_2 &= \sqrt{1+2^{-2(1)}}\sqrt{1+2^{-2(2)}}R_0\sin(\theta + d_0\alpha_0 + d_1\alpha_1) \end{aligned} \tag{4.9}$$

The *n*-th rotation can be extended and a general definition derived:

$$\begin{aligned} x_{n+1} &= \left[\sqrt{1+2^{-2(1)}}\sqrt{1+2^{-2(2)}}\cdots\sqrt{1+2^{-2n}}\right]R_0\cos(\theta + d_0\alpha_0 + d_1\alpha_1 + \cdots + d_n\alpha_n) \\ y_{n+1} &= \left[\sqrt{1+2^{-2(1)}}\sqrt{1+2^{-2(2)}}\cdots\sqrt{1+2^{-2n}}\right]R_0\sin(\theta + d_0\alpha_0 + d_1\alpha_1 + \cdots + d_n\alpha_n) \end{aligned} \tag{4.10}$$

The total increase in magnitude can be specified as $K_n = \prod_n \sqrt{1+2^{-2k}}$. This increase must be accounted for when performing calculations using this algorithm.

4.3 Accumulator Registers

The CORDIC design calls for three accumulator registers: the X register, the Y register, and the angle accumulator (Z). The X and Y registers hold the present *x*- and *y*-components of the vector as it is being rotated. The angle accumulator holds the total rotation amount completed at the current iteration.

The angle accumulator stores the arguments to the sine and cosine terms in Equation (4.10): $Z_n = d_0\alpha_0 + d_1\alpha_1 + \cdots + d_n\alpha_n$. However, since this term is equivalent to the desired rotation angle—constrained to $\alpha_i = \tan^{-1}\left(2^{-i}\right)$—the Z register must always contain the expression

$$\mathrm{Z}_i = \sum_{n=1}^{i} d_n \tan^{-1}\left(2^{-i}\right) \tag{4.11}$$

The arctangent terms can be stored in a small lookup table. When this is done, only an addition or a subtraction is required to compute the next value in the Z register, since $d = \pm 1$. This table is very small, requiring only one row for each iteration of the algorithm. Since the iteration count can be fixed in hardware, the size of the table is constant.

The accumulator registers are also used to determine the direction of rotation. In Rotation mode, the sign of the Z register determines the direction. In Vectoring mode, the Y register determines the direction. These modes are further elaborated in the next section.

4.4 Computation Modes

The CORDIC algorithm operates in one of two modes: Rotation and Vectoring. The mode of operation determines which set of functions can be computed, and how the values in the X, Y, and Z registers change each iteration.

4.4.1 Rotation Mode

i	*X*	*Y*	*Z*
—	**1.0000**	**0.0000**	**30.0000°**
0	1.0000	1.0000	–15.0000°
1	1.5000	0.5000	11.5651°
2	1.3750	0.8750	–2.4712°
3	1.4844	0.7031	4.6538°
4	1.4404	0.7959	1.0775°
5	1.4156	0.8409	–0.7124°
6	1.4287	0.8188	0.1828°
7	1.4223	0.8300	–0.2649°
8	1.4255	0.8244	–0.0411°
9	1.4272	0.8216	0.0709°
10	**1.4264**	**0.8230**	**0.0149°**

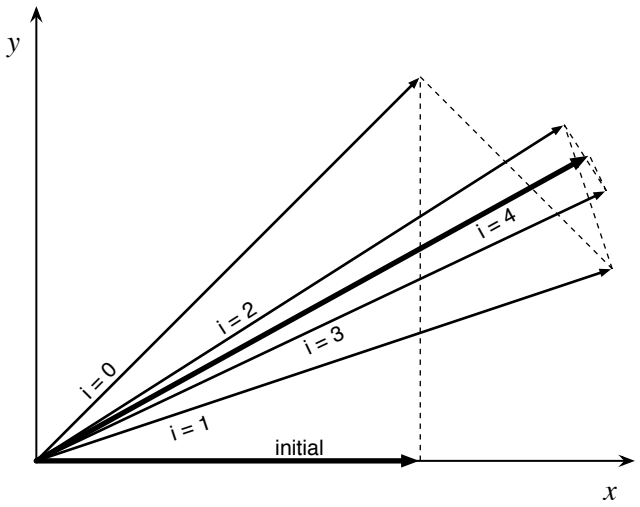

Figure 4.2—The CORDIC Rotation mode.

In Rotation mode, the input vector is rotated over a specified angle. The input vector is specified as the initial value the X and Y registers. The rotation amount is input into the Z register. In order to rotate the input vector over the input angle, the goal of each iteration should be to reduce the value in the angle accumulator to 0. Since the arctangent values for each iteration are fixed, the only way that the value in the Z register can be controlled is through the *d* values. In Rotation Mode, the decision values are defined to be:

$$d_i = \begin{cases} +1, \text{ if } \mathrm{Z}_i \geq 0 \\ -1, \text{ if } \mathrm{Z}_i < 0 \end{cases} \tag{4.12}$$

With this definition of the decision function, after *n* iterations of the algorithms, it is known what the values in each of the registers will be:

$$\begin{aligned} \mathrm{X}_n &= K_n\left[\mathrm{X}_0 \cos(\mathrm{Z}_0) - \mathrm{Y}_0 \sin(\mathrm{Z}_0)\right] \\ \mathrm{Y}_n &= K_n\left[Y_0 \cos(\mathrm{Z}_0) + X_0 \sin(\mathrm{Z}_0)\right] \\ \mathrm{Z}_n &= 0 \\ K_n &= \prod_n \sqrt{1 + 2^{-2n}} \end{aligned} \tag{4.13}$$

Figure 4.2 shows what happens to the input vector after each iteration in the algorithm. In this example, the vector is initially aligned with the *x*-axis. The magnitude of the vector increases with each step, as the angle of the vector converges on 30°. The values of the X, Y, and Z registers are also shown.

In the following sections, descriptions of how various functions can be computed using Rotation mode will be discussed.

4.4.1.1 Sine, Cosine and Polar-Cartesian Transformation

The computation of the sine and cosine functions is intrinsic to the Rotation mode, and can easily be derived from (4.13). All that is necessary is to initialize the Y register with 0, and the X register with the desired scaling factor. If there were no gain in the magnitude of the input vector, then the X register could be initialized to 1, and values of sine and cosine could be read directly out of the Y and X registers, respectively. However, the gain means that some scaling must be performed before computation. After *n* iterations of the algorithm, the contents of the registers will be:

$$\begin{aligned} \mathrm{X}_n &= K_n X_0 \cos(Z_0) \\ \mathrm{Y}_n &= K_n Y_0 \sin(Z_0) \\ \mathrm{Z}_n &\approx 0 \end{aligned} \tag{4.14}$$

Therefore, in order to compute the sine or cosine of an angle θ, then the Z register will be initialized with θ, Y with 0, and the X register with K_n, to account for the gain. Figure 4.2 shows the CORDIC process to compute sine and cosine. The initial vector is aligned, as required. The X register corresponds to the scaled cosine value, and the Y register corresponds to the scaled sine value. We can determine the unscaled value by

dividing the final values by $K_{10} = 1.6468$. For example, after 11 iterations, the algorithm yields

$$\sin(30°) = \frac{X_{10}}{K_{10}} = \frac{0.8230}{1.6468} = 0.4998$$

The method to compute sine and cosine is the same for Polar-to-Cartesian coordinate transformation. Since the transformation is defined to be

$$x = r\cos\theta \qquad (4.15)$$
$$y = r\sin\theta$$

All that is necessary to perform the transformation is to once again load θ into Z and load X with rK_n, and Y with 0.

4.4.2 Vectoring Mode

i	*X*	*Y*	*Z*
—	**1.0000**	**1.7321**	**0.0000°**
0	2.7321	0.7321	45.0000°
1	3.0981	–0.6340	71.5651°
2	3.2566	0.1405	57.5288°
3	3.2741	–0.2665	64.6538°
4	3.2908	–0.0619	61.0775°
5	3.2927	0.0409	59.2876°
6	3.2934	–0.0105	60.1828°
7	3.2935	0.0152	59.7351°
8	3.2935	0.0024	59.9589°
9	3.2935	–0.0041	60.0709°
10	**3.2935**	**–0.0009**	**60.0149°**

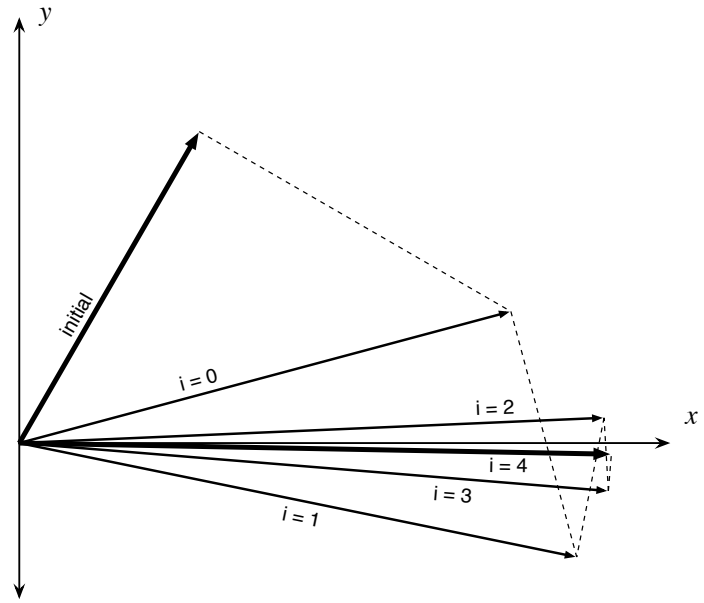

Figure 4.3—CORDIC Vectoring Mode

In Vectoring mode, the equations used to update the registers remain the same, but the function that determines the rotation direction changes. In vectoring mode, the

algorithm tries to align the input vector with the *x* axis, which means that the value in the Y register should converge on zero. The decision function in rotating mode is:

$$d_i = \begin{cases} +1, \text{ if } \mathrm{Y}_i < 0 \\ -1, \text{ if } \mathrm{Y}_i \geq 0 \end{cases} \tag{4.16}$$

Figure 4.3 shows how the input vector converges on the *x*-axis with each iteration of the algorithm. As in the rotation mode, the magnitude of the vector increases with every iteration of the algorithm. The sign of Y is used to determine which direction to rotate, with the goal of bringing the value to 0. In this example, the angle of the input vector with respect to the *x*-axis is computed and found in the Z register. Since the angle does not scale, the result is the correct angle. Using the new decision function, after *n* iterations of the algorithm, the registers will contain:

$$\begin{aligned} \mathrm{X}_n &= K_n \sqrt{\mathrm{X}_0{}^2 + \mathrm{Y}_0{}^2} \\ \mathrm{Y}_n &\approx 0 \\ \mathrm{Z}_n &= Z_0 + \tan^{-1}\left(\frac{\mathrm{Y}_0}{\mathrm{X}_0}\right) \\ K_n &= \prod_n \sqrt{1 + 2^{-2i}} \end{aligned} \tag{4.17}$$

The following sections discuss which functions can be computed in vectoring mode.

4.4.2.1 Arctangent

As seen in equation (4.17), the arctangent function is intrinsically computed in the Z register when in Vectoring mode. In order to compute the arctangent of an angle α, then the Z register must be initialized to 0, so that the Z_0 term is eliminated in (4.17), and the angle α must be expressed as a ratio of the two values in the X and Y registers. It is

possible to initialize X with 1.0, and Y with α, as is done in Figure 4.3. The result of $\arctan\left(\sqrt{3}\right)$ is correctly computed to be 60°.

$$\begin{aligned} Z_n &= Z_0 + \tan^{-1}\left(\frac{Y_0}{X_0}\right) \\ Z_n &= 0 + \tan^{-1}\left(\sqrt{3}\right) \\ Z_n &= 60^\circ \end{aligned} \tag{4.18}$$

4.4.2.2 Vector Magnitude and Cartesian-Polar Transformation

As mentioned earlier, the final value in the X register contains the scaled magnitude of the input vector. This property, combined with the intrinsic computation of arctangent at the same time means that the CORDIC vectoring mode automatically does a Cartesian to Polar coordinate transformation, just as the Rotation mode does a Polar to Cartesian conversion. Recall the equation for the Cartesian-Polar transformation:

$$\begin{aligned} r &= \sqrt{x^2 + y^2} \\ \theta &= \tan^{-1}\frac{y}{x} \end{aligned} \tag{4.19}$$

The value for *r* is computed in the X register, and θ in the Z register.

4.5 Expanding the Computation Domain

The CORDIC algorithm as presented thus far can only compute functions based on the sine and cosine functions. This is a consequence of the circular rotations performed in each step. The algorithm is capable of performing linear and hyperbolic rotations as well, which expands the set of functions that the algorithm can compute. To allow for these new domains, a new factor is introduced to the set of CORDIC equations. Its value

determines in which coordinate system the algorithm will operate. This factor is defined as

$$m \in \{-1,0,1\} \tag{4.20}$$

An m value of –1 corresponds to the hyperbolic domain, 0 to the linear domain, and +1 to the circular domain. When this factor is applied to Equation (4.5), the following general-purpose definition of the CORDIC algorithm is obtained (assuming n iterations):

$$\begin{aligned} X_n &= K_n\left[X_0 \cos\left(\alpha_n\sqrt{m}\right) + Y_0\sqrt{m}\sin\left(\alpha_n\sqrt{m}\right)\right] \\ Y_n &= K_n\left[Y_0 \cos\left(\alpha_n\sqrt{m}\right) - X_0\sqrt{m}\sin\left(\alpha_n\sqrt{m}\right)\right] \\ Z_n &= Z_0 + \alpha_n \end{aligned} \tag{4.21}$$

As in 4.2, α is the elementary rotation performed each step, with α_n being the sum of the rotations performed. This rotation is defined so that the operation performed for each step in the algorithm reduces to a shift and adds:

$$\alpha_i = \begin{cases} \tan^{-1}\left(2^{-i}\right), & m = +1 \\ 2^{-i}, & m = 0 \\ \tanh^{-1}\left(2^{-i}\right), & m = -1 \end{cases} \tag{4.22}$$

With these conditions, the algorithm reduces to:

$$\begin{aligned} x_{i+1} &= \frac{1}{K_i}\left(x_i - m d_i 2^{-i} y_i\right) \\ y_{i+1} &= \frac{1}{K_i}\left(y_i + d_i 2^{-i} x_i\right) \\ z_{i+1} &= z_i - d_i \alpha_i \\ &\text{where } K_i = \sqrt{1 + m2^{-2i}},\ d_i = \pm 1,\ m \in \{-1,0,+1\} \end{aligned} \tag{4.23}$$

Domain	Growth Factor, K_i	**Constant After**
Circular $(m=1)$	$\prod_i \sqrt{1+1\cdot 2^{-i}} = 1.166707425$	15 iterations
Linear $(m=0)$	$\prod_i \sqrt{1+0\cdot 2^{-i}} = 1.000000000$	—
Hyperbolic $(m=-1)$	$\prod_i \sqrt{1+(-1)\cdot 2^{-i}} = 0.828159361$	15 iterations

Table 4.1—CORDIC growth factors for each computation domain.

The values of the CORDIC growth factors are shown in Table 4.1. The values are rounded to nine decimal places. The Constant After column shows how many iterations it takes until the displayed value remains constant. For all but the linear domain, which always has a factor of 1, the displayed value can be used for any implementation with 15 or more iterations.

As before, the x and y components are stored in the X and Y registers. The definition of Z depends on the computation mode, with hyperbolic tangent being used instead of the standard tan function when in the hyperbolic domain, for example:

$$z_{i+1} = \begin{cases} z_i - d_i \tan^{-1}\left(2^{-i}\right), & m = 1 \\ z_i - d_i 2^{-i}, & m = 0 \\ z_i - d_i \tanh^{-1}\left(2^{-1}\right), & m = -1 \end{cases} \tag{4.24}$$

The following table summarizes what functions can be computed and in which mode and domain they can be computed:

	Computation Mode	
Domain	**Rotation**	**Vectoring**
Circular $(m=1)$	$X = K_n X_0 \cos(Z_0)$ $Y = K_n X_0 \sin(Z_0)$ $\tan Z_0 = \frac{Y}{X}$	$X = K_n \sqrt{X_0^{\,2} + Y_0^{\,2}}$ $Z = Z_0 + \tan^{-1}\left(\frac{Y_0}{X_0}\right)$
Linear $(m=0)$	$Y = Y_0 + X_0 Z_0$	$Z = Z_0 + \frac{Y_0}{X_0}$
Hyperbolic $(m=-1)$	$X = K_n X_0 \cosh(Z_0)$ $Y = K_n X_0 \sinh(Z_0)$ $\tanh Z_0 = \frac{Y}{X}$ $e^{Z_0} = X + Y$	$X = K_n \sqrt{X_0^{\,2} - Y_0^{\,2}}$ $Z = Z_0 + \tanh^{-1}\left(\frac{Y_0}{X_0}\right)$ $\ln(\lambda) = 2Z,\ \text{with}\ X_0 = \lambda + 1,\ Y_0 = \lambda - 1$ $\sqrt{\lambda} = \frac{1}{K_n} X,\ \text{with}\ X_0 = \lambda + \frac{1}{4},\ Y_0 = \lambda - \frac{1}{4}$

Table 4.2—Functions that can be computed using the CORDIC algorithm.

In addition to the addition of the *m* domain selector, a couple other modifications must be made to the algorithm when functioning in other modes.

In the circular domain, the algorithm starts with the registers in their initialized states and the algorithm begins with a sequence of *i* values of the form 0, 1, 2, 3, 4, … and so on until the desired number of iterations has been completed. In the first step, no shift is performed as a result of *i* being equal to 0. In the linear domain, the sequence goes 1, 2, 3, 4, 5, … and so on, with a shift happening in the first step. Finally, as explained in 4.6, the hyperbolic domain's *i* sequence is further complicated by the necessity to repeat iterations. The *i* sequence in this domain is 1, 2, 3, 4, 4, 5, … with every $3k + 1$ iteration being repeated.

4.6 Convergence

The algorithm is now fully defined, and it will now be demonstrated that the algorithm converges on the previously noted functions. The specific region of convergence will also be shown.

4.6.1 The Convergence Condition

Assume that the CORDIC algorithm is in vectoring mode. Let ϕ_i be the angle of the input vector after the *i*th iteration of the algorithm. The algorithm tries to reduce the angle of the input vector. Therefore, after step $i+1$, the angle of the vector will change such that

$$|\phi_{i+1}| = ||\phi_i| - \alpha_i| \tag{4.25}$$

where α_i is the rotation performed in the *i*th iteration of the algorithm. In order for the algorithm to converge in *n* iterations, then all subsequent iterations must bring ϕ to within α_{n-1} of zero. If this is the case, then when the *n*th iteration completes, the input angle will be zero. Since the rotations accumulate, the following condition is derived:

$$\alpha_i - \sum_{k=i+1}^{n-1} \alpha_k < \alpha_{n-1} \tag{4.26}$$

In order for the algorithm to converge, the condition in (4.26) must hold when the algorithm first begins:

$$|\phi_0| - \sum_{k=0}^{n-1} \alpha_k < \alpha_{n-1} \tag{4.27}$$

To find the domain of initial values for which the algorithm converges, the above inequality is solved:

(4.28) $$\max\left(\left|\phi_0\right|\right) = \alpha_{n-1} + \sum_{k=0}^{n-1} \alpha_k$$

Since the definition of α_i depends only on *i*, the domain of convergence can be computed. The domains of convergence for the three computation domains are shown in Table 4.3 by using (4.28). By evaluating the limit of the α terms as *i* approaches ∞, and observing that $\alpha_{n-1} \to 0$ as $i \to \infty$, the α_{n-1} term can be dropped from (4.28).

Computation Domain	Approximate Domain of Convergence
Circular $(m=1)$ $\alpha_i = \tan^{-1} 2^{-i}$	$\sum_{i=1}^{\infty} \tan^{-1}\left(2^{-i}\right) \approx 1.74329$
Linear $(m=0)$ $\alpha_i = 2^{-i}$	$\sum_{i=1}^{\infty} 2^{-i} = 1$
Hyperbolic $(m=-1)$ $\alpha_i = \tanh^{-1} 2^{-i}$	$\sum_{i} \tanh^{-1}\left(2^{-i}\right) \approx 1.11817$ $i = \{1, 2, 3, 4, 4, 5, \ldots\}$

Table 4.3—Domains of convergence for the CORDIC algorithm.

Note that the condition (4.26) is not met when in the hyperbolic domain if the standard non-repeating sequence (e.g. 1, 2, 3,…) is used. In order for the algorithm to converge, certain iterations must be repeated. Specifically, it is necessary for steps {4, 13, 40, … , $3k+1$} to be repeated. This comes as a consequence of the following: [1]

(4.29) $$\alpha_i - \left(\sum_{k=i+1}^{n-1} \alpha_k\right) - \alpha_{3i+1} < \alpha_{n-1}$$

4.6.2 Proof of Convergence Criteria

In order to demonstrate that ϕ_0 will converge to at most $\left|\alpha_{n-1}\right|$, it will first be proven by induction that the following is true:

(4.30) $$\left|\phi_i\right| < \alpha_{n-1} + \sum_{k=1}^{n-1} \alpha_k$$

First, (4.30) is true for $i = 0$, as a result of (4.27). To prove that (4.30) is true for $i + 1$, α_i is subtracted from (4.30) and (4.26) is applied to the left side. This yields:

$$-\left[\alpha_{n-1} + \sum_{k=i+1}^{n-1} \alpha_k\right] < -\alpha_i < \left|\phi_i\right| \tag{4.31}$$
$$-\alpha_i < \left[\alpha_{n-1} + \sum_{k=i+1}^{n-1} \alpha_k\right]$$

When (4.25) is applied, the following results:

$$\left|\phi_{i+1}\right| < \alpha_{n-1} + \sum_{k=i+1}^{n-1} \alpha_k \tag{4.32}$$

Therefore, by induction, it is proven that (4.30) is true for all $i \leq 0$. If $i = n$, then

$$\left|\phi_n\right| < \alpha_{n-1} \tag{4.33}$$

which proves that the CORDIC algorithm converges if the input angle is within the domain of convergence defined in (4.28) is satisfied.

4.6.3 Convergence in Rotation Mode

The domain of convergence definition and proof in 4.6.1 and 4.6.2 assumed that CORDIC is operating in vectoring mode. If z is substituted for ϕ in the equations in those sections, then the domain of convergence can be found and proven for the rotation mode. In particular, (4.28) shows that z has the same domain of convergence as ϕ:

$$\max\left(\left|z_0\right|\right) = \max\left(\left|\phi_0\right|\right) \tag{4.34}$$

4.7 Accuracy and Error

With the algorithm now fully defined, and its convergence proven, the accuracy and precision of the algorithm will now be discussed. Generally speaking, for each iteration of the algorithm, an additional bit of accuracy is obtained.

Error creeps into the results from many sources. Naturally, since there is not an infinite number of bits available to any real hardware device, rounding errors are inherent to any implementation. Additionally, since there are a finite number of iterations to any real-world implementation of the algorithm, the desired rotation angle can never be fully realized. This results in an angle approximation error.

4.7.1 Numerical Representation

The CORDIC algorithm operates using bit-shifts, the natural representation for any number used in the algorithm is going to be a fixed-point representation rather than floating-point. A fixed point number is a scaled integer, where the binary point is implied to be n positions to the left of the LSB. As a result, the integer is 2^n times larger than the number it is representing. This is similar to storing 2.64 as 264. This scale, combined with the number of bits available completely determines the range and precision available to all numbers in the algorithm. The scale determines how many bits are available to the left and right of the binary point.

As the binary point moves to the right, greater ranges of numbers are available, but the scale becomes coarser, with larger gaps between adjacent numbers. Likewise, as the binary point moves to the left, the scale is finer, but the range of possible numbers is smaller.

Guard digits can also be employed at both ends of the binary word to enhance accuracy. The guard digits are not used in any input values and are reserved for bits appearing in final values. For example, at least 2 bits are necessary in circular mode, because the K_n factor is grater than 1, which means that the magnitude of the values will grow as the algorithm works. Guard digits on the least-significant side are employed to reduce rounding error.

4.7.2 Rounding Error

When the number of bits required to represent a value exceeds the number of bits available in the system, the designer has two options: round or truncate. In a truncation, the excess bits are discarded. In a round, the representation is altered depending on the excess bits. Rounding works better in the CORDIC algorithm, because the maximum rounding error is only half as large as the error resulting from truncation.

Rounding errors are introduced in the course of completing each iteration of the algorithm. These errors accumulate, which has the effect of further offsetting the final result. However, Walther pointed out in [1] that the rounding of the results in each of N iterations never results in more than $\log_2 N$ bits of error. In order to negate this effect, the system can use $N + \log_2 N$ bits of storage for all intermediate results.

4.7.3 Angle Approximation Error

It has been shown that the algorithm will always converge on an accurate result if given infinite precision. However, for some values, it may take an infinite number of iterations to arrive at the accurate result. As a result, the actual rotation of the input vector is only an approximation of the desired angle. This angle approximation error is the

desired rotation angle minus the actual rotation angle (which is the sum of all the intermediate angles):

(4.35) $\Delta\theta = \theta - \sum_{i=0}^{n-1} d_i \alpha_i$, where θ is the desired rotation angle

Since this error decreases with the number of iterations, an obvious way to improve the accuracy of the algorithm is to increase the number of iterations performed. However, there are some limits to how many iterations can be added and still improve accuracy. First, the rounding error discussed in 4.7.2 increases with the number of iterations, so this effect must be considered when adding to the iteration count. Further, since the angles are quantized, the accurate angle may never fit into the bit width provided. If *B* bits are used for storage, and there are *P* bits to the right of the binary point, then the value of the least-significant bit in storage is given by $2^{-B}2^{P}$. This is the smallest number that can be accurately specified in the system. Therefore the last rotation angle (the smallest) must be able to fit in the available space. This yields the following condition:

(4.36) $$\alpha_{n-1} \geq 2^{-B}2^{P}$$

If this does not hold, then the angle actually stored will be 0, which will result in no further rotations of the input vector.

Chapter 5 The Parallel Implementation

This thesis studies two different implementations of the CORDIC algorithm. Both designs are synthesized for the Xilinx Spartan-3 XC3S1500 using the Xilinx ISE version 7.1i and simulated using ModelSim version 6.0a. This chapter studies a straightforward, parallel design consisting of two shift registers, a standard register, four adder/subtractors, a lookup table, and a control unit. The area and timing requirements are discussed, and simulations of the design are presented.

The design is configured to be in hyperbolic rotation mode in order to compute the exponential function, but all components are present for a completely general-purpose CORDIC unit. Only minor changes to the control unit would be necessary to implement the other modes. This design also includes the hardware necessary to perform the repeat iterations required in the hyperbolic domain. The control unit can be modified to allow for more a sophisticated iteration repetition scheme.

This implementation uses 32 bits of precision, with the binary point fixed such that there is one sign bit and three integer bits to the left of the binary point, and 28 bits representing the fractional part of the values. The design uses 2's complement arithmetic, giving the registers in the design a range of values from –8.0 to 7.0. The position of the binary point is implicit, and not specified anywhere in the design. The values in the lookup table are generated using the specified binary point position, and in order for correct results to be achieved, all inputs must be scaled accordingly. The only change necessary to allow for a different binary point position would be in the values stored in the lookup table. No changes to the design would be required.

5.1 Design

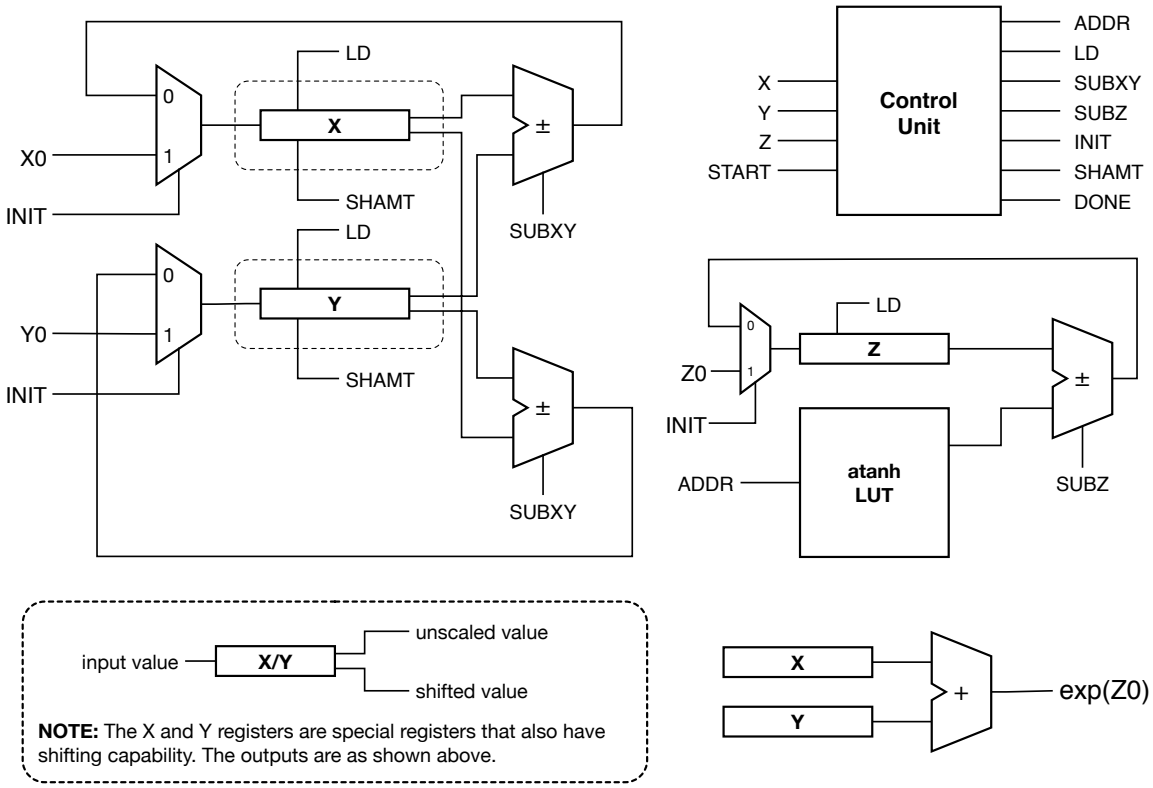

Figure 5.1—Block-level diagram of the complete parallel CORDIC unit.

All major components of the CORDIC processing unit are shown in Figure 5.1. The X, Y, and Z registers contain the current values of the X, Y, and Z components of the CORDIC algorithm. They each have parallel load capabilities controlled by the LD input signal. 2:1 multiplexers control the input values to each of the registers. An INIT output by the control unit allows initial values to be loaded before computation begins. When INIT is asserted (active high), then the multiplexers pass the initial values into the register. During the computation phase, the INIT signal is not asserted, allowing the output of the adders to be fed back into the registers.

The control unit uses the sign of the Z register to generate the subtract signals that are fed into the adder/subtractors. With the appropriate minor modifications to the control unit, the generation of the SUBXY and SUBZ signals can be modified so that the unit

operates in vectoring mode. The design for the control unit is discussed further in 5.1.1. The values of the X and Y registers are input into a final adder to generate the exponential function. After the algorithm is complete, the result will be e^{Z_0}. The unit can also compute e^{-Z_0} by changing the adder into a subtractor.

Rather than compute the arc-hyperbolic tangent, a small lookup table is used to store the pre-computed values of this function. The table has one row per iteration of the algorithm. If a general-purpose CORDIC algorithm were to be designed, an additional lookup table would be necessary to store the necessary arctangent values. For the linear domain, an additional shift register would also be necessary. A multiplexer would then be used to select the value from the appropriate table. The control unit provides the address used to access the data from the table.

	Used	Available	Utilization
Slices	329	13,312	2.5%
Slice Flip-Flops	123	26,624	0.5%
LUTs	607	26,624	2.3%
IOBs	132	221	59.7%
BRAMs	1	32	3.1%
GCLKs	1	8	12.5%

Table 5.1—Overall device utilization for the parallel design.

The device utilization for entire design is summarized in Table 5.1. The design uses 2.4% of the XC3S1500's available slices, leaving ample space for usage by a neural network. The Xilinx synthesizer also reported that the design can handle a theoretical maximum clock frequency of 75.0 MHz. Naturally, testing is necessary to determine the true timing requirements for the design. The following sections will discuss the design and area requirements forf the major components of the unit.

5.1.1 The Control Unit

The control unit is responsible for regulating the flow of data through the CORDIC unit. The unit is a finite state machine having three states, as shown in Figure 5.2. The default state for the unit is the IDLE state. In this state, the registers are not loading, and the unit is stays in this state until the START signal is asserted. When the START signal is asserted, the unit moves into the PRELOAD state.

In the PRELOAD state, the register load signal is asserted, and the INIT signal is also asserted, causing the initial values to be loaded into the registers (these values are represented as X0, Y0, and Z0 in Figure 5.1). The unit then advances to the COMPUTE state. The register load signals remain asserted, but the INIT signal is not asserted. As a result, the registers are loaded with the values from the corresponding adders, rather than the input values. The unit maintains an internal iteration count, and when these values are equal, the control unit moves back into the IDLE state, with the DONE signal asserted, signifying that the values output by the registers are valid. Table 5.2 shows the unconditional outputs for each state. Other signals, such as the look-up table address, SHAMT, and SUB signals vary and are continuously updated while in the COMPUTE state. The logic behind the state transitions is shown in Figure 5.3.

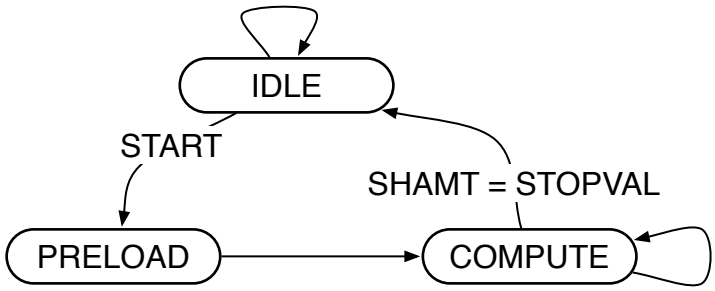

Figure 5.2—State diagram for the parallel CORDIC unit.

	IDLE	PRELOAD	COMPUTE
INIT	0	1	0
DONE	1	0	0
LD	0	1	1

Table 5.2—Unconditional control unit outputs for each state in the parallel design.

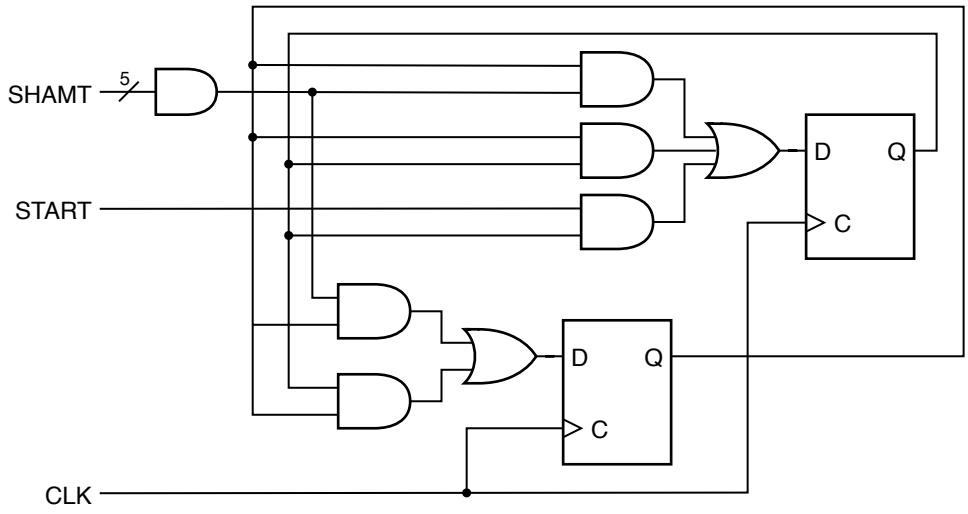

Figure 5.3—State transition logic for the parallel CORDIC unit.

The generation of the SHAMT signal is complicated by the need to perform repeat iterations, and the need to wait one clock cycle to read hyperbolic tangent data from the LUT. Figure 5.4 shows the logic behind the SHAMT signal, as well as the other signals that are state independent (SUBXY, SUBZ, ADDR, and DONE). The subtract signals SUBXY and SUBZ are determined by the sign of the Z register, since the CORDIC unit is operating in rotation mode. Therefore, the sign bit of the Z register is connected directly to the subtract inputs of the corresponding adders.

The SHAMT register contains the shift amount, which also corresponds to the current iteration count, which is then connected to the X and Y registers. Since data from the look-up table takes one clock cycle to be retrieved, SHAMT is required to lag one cycle behind the ADDR signal, which is fed to the lookup-table. This is shown as a direct connection between the output of the ADDR register and the input of the SHAMT

register. It is the contents of the ADDR register that determine when a repeat iteration is necessary.

The current ADDR is compared against the value of the NXTRPT register. The NXTRPT register contains the next iteration that must be repeated. It is initialized to 4. In hyperbolic mode, every $3k+1$ iteration must be repeated (where k is the previously repeated iteration). This computation can be performed using a single adder. $3k$ is computed by inputting NXTRPT into the first input of the adder and $2 \cdot \text{NXTRPT}$ (accomplished with a bit shift). By asserting the carry-in input, $3 \cdot \text{NXTRPT} + 1$ is computed. When the current value of ADDR and NXTRPT match, then NXTRPT is updated, and ADDR gets loaded with its current value, rather than the output of the adder that increments its value.

When SHAMT matches the constant STOPVAL, then the DONE signal is asserted, and the unit returns to the IDLE state.

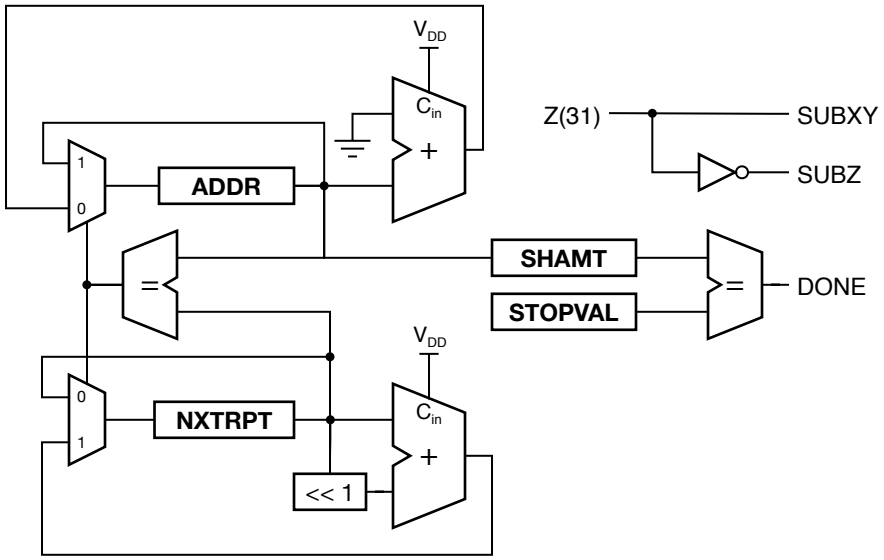

Figure 5.4—Logic behind the state-independent outputs of the parallel control unit.

	Used	Available	Utilization
Slices	39	13,312	0.3%
Slice Flip-Flops	29	26,624	0.1%
LUTs	71	26,624	0.3%
IOBs	113	221	51.1%
BRAMs	0	32	0.0%
GCLKs	1	8	12.5%

Table 5.3—Device utilization for the parallel control unit.

The area requirements for this component of the CORDIC unit are detailed in Table 5.3. The control unit accounts for 12% of the slices used by the entire design.

5.1.2 The Shift Registers

The X and Y registers require shift capability. Since the shift amount increases with each iteration of the algorithm, a standard single-bit shift register cannot be used. A register capable of shifting a variable amount is necessary. Fundamentally, there are two approaches to designing such a register. The easiest and most area-conscious is to perform a single one-bit shift per clock cycle and repeat as necessary until the value has been shifted by the desired amount. The single bit shift can be hard-wired, minimizing area requirements. This approach requires the computation to pause until the shifting is complete. This is used as part of the area optimization gained from the bit-serial implementation discussed in Chapter 6.

It is preferred that the shift be performed in one clock cycle. This can be accomplished using multiplexers, and is illustrated for a 4-bit register in Figure 5.5. For this design, which uses 32-bit registers, 32 32×5 multiplexers are used for each shifting unit. The registers have two outputs, the unshifted output direct from the register, and the shifted output, which is the output from the multiplexers. The SHAMT signal controls the amount by which the output is shifted. The shift register performs an arithmetic shift

operation to ensure proper execution when negative values are used. Consequently, the MSB (the sign bit) is connected to the shifted outputs of the multiplexers to allow for a sign extension to occur when shifting.

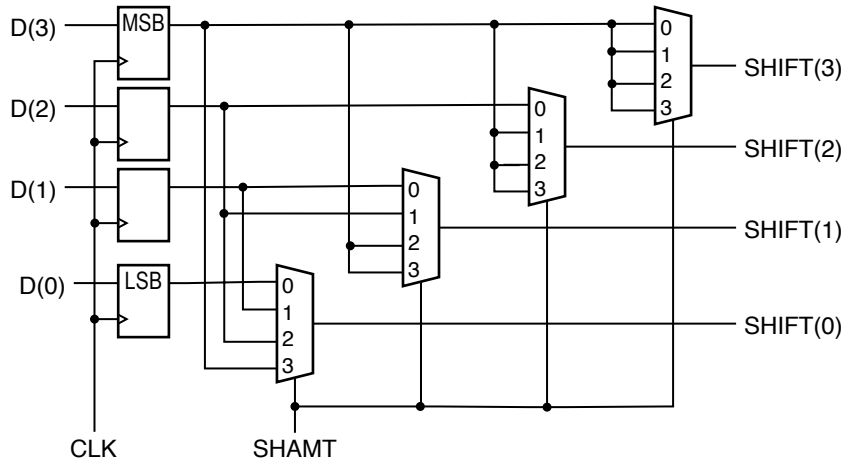

Figure 5.5—Design of the variable shift register.

	Used	Available	Utilization
Slices	90	13,312	0.7%
Slice Flip-Flops	65	26,624	0.2%
LUTs	162	26,624	0.6%
IOBs	104	221	47.1%
BRAMs	0	32	0.0%
GCLKs	1	8	12.5%

Table 5.4—Device utilization for the parallel shift register.

The shift registers use a significant portion of the device area, relative to the rest of the design, as shown in Table 5.4. Note that the table only covers a single register, and there are two shift registers in the total design. The Z register is a standard register without shifting functionality.

5.1.3 The Lookup Table

The lookup table contains the values of the arc hyperbolic tangent necessary for each iteration of the algorithm. A C program written for that purpose generates the VHDL for the table. The program uses the C math library functions in conjunction with a fixed-

point conversion function to output a VHDL file containing a description of the lookup table in fixed-point notation.

	Used	Available	Utilization
Slices	0	13,312	0.0%
Slice Flip-Flops	0	26,624	0.0%
LUTs	0	26,624	0.0%
IOBs	38	221	17.2%
BRAMs	1	32	3.1%
GCLKs	1	8	12.5%

Table 5.5—Device utilization for the parallel lookup table.

The presence of the lookup table has essentially zero impact on the area requirements for the unit, since the entire table can be synthesized into a single BRAM, as shown in Table 5.5.

5.1.4 Design Summary

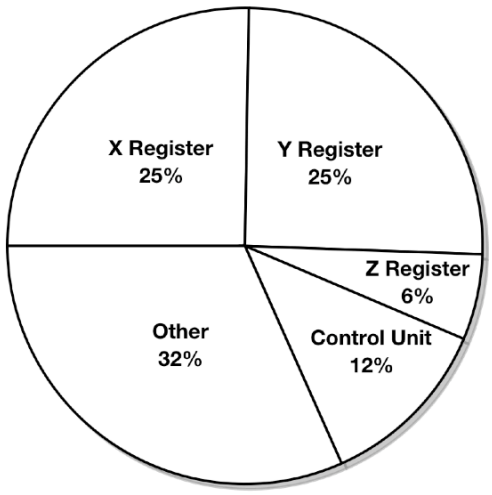

Figure 5.6—Slices used by the various components of the parallel CORDIC unit.

The entire design uses very little of the FPGA's resources, and Figure 5.6 shows that much of the FPGA's slices are used up by the X and Y shift registers. The large multiplexers required by the design result in massive resource requirements. Chapter 6

presents an alternate design that aims to further reduce the area requirements, but at the cost of speed.

With only 2.4% of the Spartan-3's slices used by this design, there is still plenty of room left for a system such as a neural network that uses the CORDIC unit. If necessary, the resource requirements could be further reduced by reducing the precision of the unit (to 16 bits, for example).

At 31 iterations (with 2 repeats), plus a PRELOAD cycle, the algorithm will take 34 clock cycles to complete. At the theoretical maximum clock frequency of 75.0 MHz, the unit will take 0.45 μs to compute the final value.

5.2 Simulation Results

This section presents the results of simulations performed using the design specified in 5.1. The design uses 32 bits to represent each word, with 4 bits for the integer part of the number, and 28 bits for the fractional part of the number. Thus the values output by the X, Y, Z, and the adder that computed the exponential function are effectively scaled up by a factor of 2^{28}. The simulated clock runs at 100 MHz.

5.2.1 Simulation 1: Computing *e* With CORDIC Growth Error

The first simulation performed demonstrates the effect of the CORDIC gain K_n. To compute the exponential function e^z, the X register must be initialized with 1, the Y register with 0, and the Z register with the desired argument to the exponential function. This simulation attempts to compute the value of e, so an initial value of 1 is placed into the Z register.

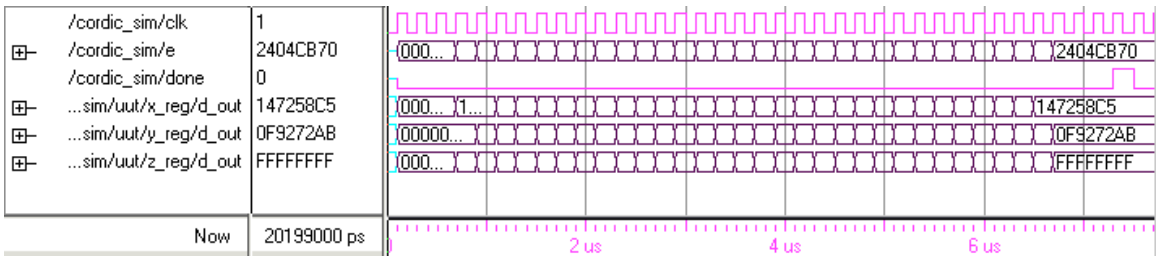

Figure 5.7—Results of a ModelSim simulation attempting to compute the value of *e*. As a result of CORDIC growth, the final computed value is inaccurate.

The results of this simulation are shown in Figure 5.7. The algorithm iterated for a total of 34 times (including the PRELOAD state and the two repeat iterations required in the hyperbolic domain). The final hexadecimal values in the X and Y registers are $147258C5_{16}$ and $0F9272AB_{16}$ respectively. When converted to decimal and scaled accordingly, they become 1.2779_{10} for the X register and 0.9733_{10} for the Y register. The value of the exponential function is the sum of these two numbers: 2.2512_{10}, which matches the output of the E signal.

The actual value of *e* to four decimal places is 2.7183. This simulation demonstrates an error in computation by 0.4671. This is a result of the CORDIC magnitude error factor K_i discussed in 4.2 and illustrated in Figure 4.2. In order to produce accurate results, the X register needs to be pre-scaled by the gain factor *K*.

5.2.2 Simulation 2: Computing *e* Compensating for CORDIC Growth

In order to compute an accurate value of *e*, the growth observed in 4.2 must be accounted for. To produce an unscaled final value, the initial value must be divided by the growth factor. The formula used to compute the amount of growth is shown in (4.23). A precomputed value is shown in Table 4.1. For this simulation rather than initializing X with 1, X will be initialized with the hexadecimal value $1351E872_{16}$, which equals 1.2075_{10}. Y is again initialized with 0 and Z with 1.

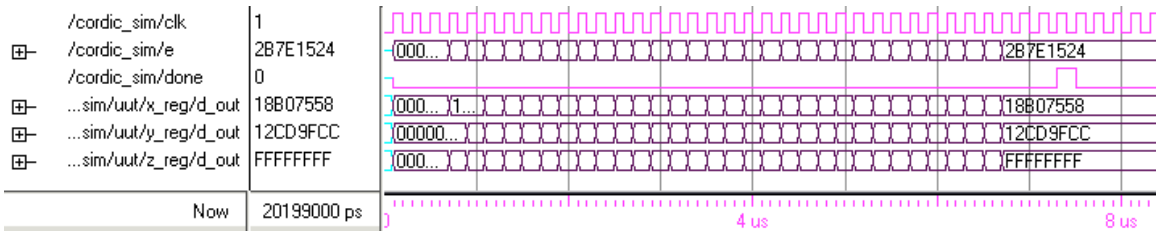

Figure 5.8—Results of a ModelSim simulation computing the value of *e*. By accounting for CORDIC growth, the final computed value is accurate.

The results of this simulation are shown in Figure 5.8. The final computed value of *e* is output on the E bus. Its hexadecimal value $2B7E1524_{16}$ converts to 2.7183_{10}, which matches the actual value of *e*. When taken to 30 decimal places the value output in E is $2.718281880021100000000000000000_{10}$, which differs from the real value of *e* by only $0.000000051562050185083300_{10}$. The only way for this error to be reduced would be to use more precision in the registers. Widening the registers, or shifting the binary point to the left can achieve this. In addition to allowing more precision in the results, it also allows greater precision for the values stored in the arc hyperbolic tangent lookup table.

5.2.3 Simulation 3: Computing e^{-1}

It is also important for the unit to compute the value of the exponential function for negative arguments. Since the domain of convergence for the algorithm is symmetric about 0, half of the possible arguments are negative. This simulation computes the value of e^{-1}. X is initialized with 1.2075_{10} to ensure an unscaled result, and Y is initialized with 0. The Z register contains the argument for the exponential function and is thus initialized to -1_{10} ($F0000000_{16}$ in 2's complement notation).

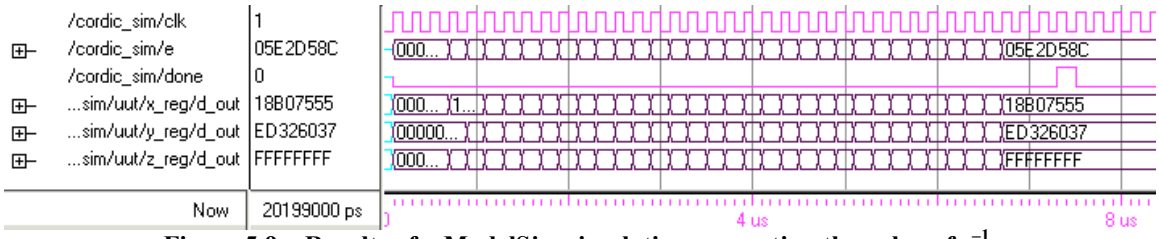

Figure 5.9—Results of a ModelSim simulation computing the value of e^{-1}.

The simulation results presented in Figure 5.9 show that the CORDIC unit is able to handle the negative argument. The final value in the E register ($05E2D58C_{16}$) corresponds to 0.3678794354_{210}, which is of the same accuracy as the computation for e^1.

5.2.4 Simulation 4: Computing Outside the Domain of Convergence

As discussed in 4.6, the CORDIC algorithm does not converge on an accurate answer for all input values. Table 4.3 shows the domains of convergence for all the computation domains supported by the CORDIC algorithm. For the rotation mode, which this design uses, the domain of convergence is a limiter on the magnitude of the initial value of Z. If Z were initialized with a value whose magnitude is greater than 1.11817_{10}, then the final computed value would be incorrect. For this simulation, the X register is again initialized with 1.2075_{10}, and Y is again initialized to 0, but Z is initialized to 5, so that the unit will attempt to compute e^5.

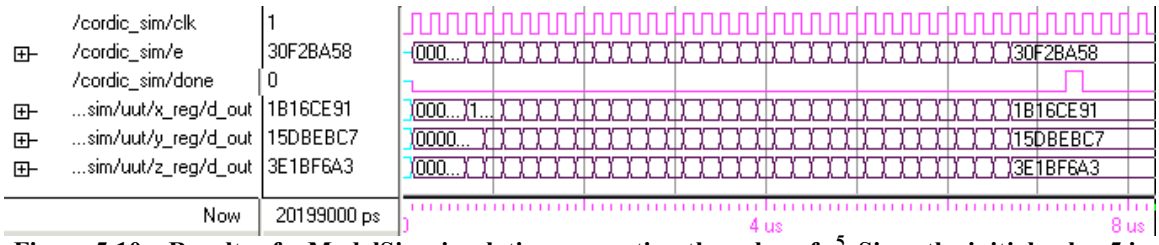

Figure 5.10—Results of a ModelSim simulation computing the value of e^5. Since the initial value 5 is outside the domain of convergence, the computed value is incorrect.

The simulation results shown in Figure 5.10 demonstrate that the CORDIC unit is unable to compute e^5. The final value output by E is 3.0593_{10}, which is nowhere near the correct value (148.4132_{10}). 3.0593_{10} acts is an asymptote for the computed values of E. Once the initial values of Z move outside the domain of convergence, the computed E value rapidly approaches 3.0593_{10}, and remains at that value no matter how large a value

Z is initialized with. A similar asymptote exists for when Z is initialized with values that are too negative. In this case, the E values approach 0.3269_{10}.

5.3 Summary

The parallel design is a straightforward implementation of the CORDIC equations. Even with the large register sizes, the design has a small footprint inside the Spartan-3. With iterations lasting one clock cycle, the performance is consistent. Performance can be improved by using fewer iterations, which usually entails using a smaller word size. The simulations demonstrate acceptable accuracy, especially given the area requirements for the design. For systems that require an even smaller footprint, Chapter 6 presents a smaller, slower bit-serial implementation that further aims to reduce area requirements by using 1-bit adder/subtractors and simple shift registers.

Chapter 6 The Serial Implementation

A straightforward implementation of the CORDIC algorithm is discussed in Chapter 5. While that design required very little of the FPGA's resources, there might be some applications where further area reduction is necessary. It was observed in 5.1.4 that the two variable-width shift registers accounted for half the resource requirements for the design. This chapter presents an alternate design that uses bit-serial arithmetic along with one-bit adders to execute the CORDIC algorithm. It will be shown that this design requires far less of the FPGA's resources, but will require more time to execute.

6.1 Design

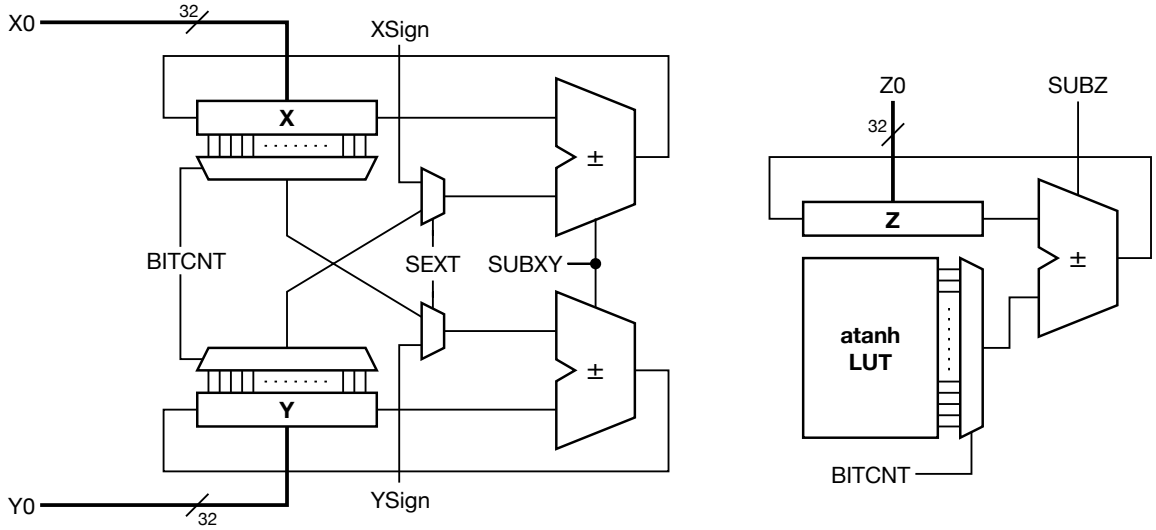

Figure 6.1—Block diagram for the serial implementation of the CORDIC algorithm.

As shown in Figure 6.1, the overall the design remains relatively unchanged when compared to the parallel design shown in Figure 5.1. All the changes were made to the individual components of the design. The X, Y, and Z registers are now standard one-bit shift registers with parallel load and parallel output capability. This keeps their internal logic simple and reduces area requirements. The adder/subtractors are now reduced to operating on single-bit operands, which also reduces their complexity. The adders also

contain a flip-flop that saves the carry output so that it can be applied to the next pair of operands. This allows the 32-bit addition to be performed one bit at a time.

Each iteration requires the X and Y cross-values to be shifted by the current iteration count. The two multiplexers attached to the X and Y registers allow the correct bit to be selected for the corresponding register. Since the results of the addition or subtraction are fed back as the serial inputs to the registers, an additional SEXT control signal and 2:1 multiplexer is needed to choose either the bit selected from the register or the corresponding sign bit. Without this signal, bits computed previously in the iteration would be fed into the adders. The sign of each register is saved prior to the execution of each iteration, allowing for an arithmetic shift of the values being fed into the adders.

A multiplexer was also used to allow the correct bit to be fed from the arc hyperbolic tangent lookup table to the Z-adder. The multiplexer allows the lookup table to be placed in BRAM and prevents any additional clock cycle delays from being introduced into the design, as would be the case if the value were loaded into a shift register. As before, the control unit provides the address into the lookup table.

This design uses a 32-bit word size, with 4 bits for the whole part of the number, and 28 bits for the fractional part of the number.

	Used	Available	Utilization
Slices	139	13,312	1.0%
Slice Flip-Flops	143	26,624	0.5%
LUTs	256	26,624	1.0%
IOBs	132	221	59.7%
BRAMs	1	32	3.1%
GCLKs	1	8	12.5%

Table 6.1—Overall device utilization for the serial design.

The device utilization for the serial design is shown in Table 6.1. This design uses only 139 of the XC3S1500's available slices, as opposed to the 329 required by the parallel design. This is a reduction of 57.8%. In addition, the Xilinx synthesizer reported that a maximum clock frequency of 134.8 MHz can be used. The parallel design tops out at 75.0 MHz. The simplicity of this design results in less combinational logic delay which in turn allows for faster clock speeds.

6.1.1 Control Unit

The parallel design allowed each iteration to be completed in one clock cycle. This reduced the complexity of the control unit. The switch to a serial design means that the number of clock cycles required to complete each iteration depends on the word size, since all values are computed one bit at a time. Previously, the subtract control signals that determine whether each adder is adding or subtracting could be evaluated each clock cycle. In the serial design, these values need to be determined once at the beginning of the iteration and saved until the current value is fully computed. To accomplish this, a new state is added to the state machine of the control unit, as shown in Figure 6.2.

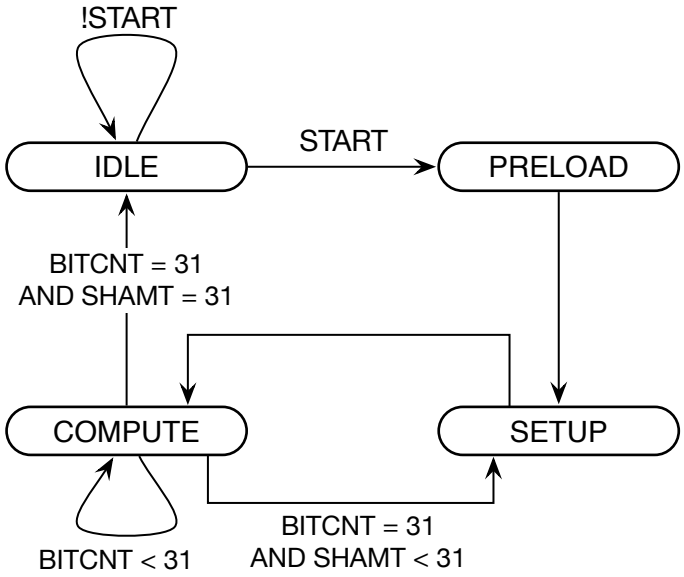

Figure 6.2—State diagram for the serial implementation of the CORDIC algorithm.

The IDLE and PRELOAD states remain unchanged. The IDLE state is active when the algorithm is not being executed. The PRELOAD state is used to load the initial values into the X, Y, and Z registers. The SETUP state is the new state and is only active the very first clock cycle of each iteration. In the SETUP state, the values in the X, Y, and Z registers are evaluated to determine the subtract and sign signals. These signals are saved into registers for use during the remaining clock cycles of the current iteration. The COMPUTE state is active until all 32 bits have been computed. Once this happens, IDLE becomes the active state, otherwise if more iterations remain, the machine returns to the SETUP state to begin execution of the next iteration.

A new internal control value is also added, named BITCNT. Whereas SHAMT holds the current iteration count, BITCNT holds the number of bits that have been processed within the current iteration. When this counter rolls over to zero, then the active iteration is completed.

	IDLE	PRELOAD	SETUP	COMPUTE
INIT	0	1	0	0
DONE	1	0	0	0
PLD	0	1	0	0
SHIFT	0	0	1	1

Table 6.2—State outputs for the serial control unit.

To manage the new shift registers, a new SHIFT signal is also added. When asserted, the SHIFT signal causes the shift registers to load the input bit into the MSB position, and shift out the LSB. The old LD signal has been changed to cause a parallel load operation in the shift registers. This allows execution of the algorithm to begin sooner. Otherwise, without the parallel load, the initial values would need to be shifted into the register. The values of these signals in each of the four states are shown in Table 6.2.

	Used	Available	Utilization
Slices	38	13,312	0.3%
Slice Flip-Flops	33	26,624	0.1%
LUTs	67	26,624	0.3%
IOBs	126	221	57.0%
BRAMs	0	32	0.0%
GCLKs	1	8	12.5%

Table 6.3—Device utilization for the serial control unit.

The device utilization for the control unit is shown in Table 6.3. The parallel control unit uses 39 slices, and this control unit uses 38 slices, making the area requirements for the two units essentially the same. This unit required more slice flip-flops (33) than the parallel unit's 29 flip-flops.

6.1.2 The Shift Registers

The X, Y, and Z registers in the serial design are all identical shift registers with parallel load and parallel output capability. In addition to the clock signal, the registers have two control inputs, two data inputs, and two data outputs. The data inputs are

labeled as S_IN and P. S_IN is the bit to be shifted in to the MSB position of the register and P is the parallel data input. The two control signals are PLD and SHIFT. When SHIFT is asserted, then the contents of the register shift to the right every clock cycle, with S_IN becoming the new MSB. When P_LD is asserted, then the parallel data in P is loaded into the register at the next rising edge of the clock.

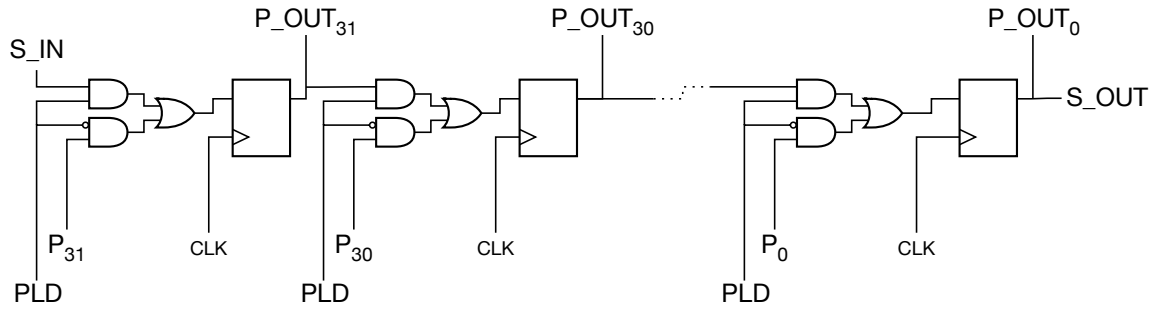

Figure 6.3—Design of the serial shift registers.

The basic design is shown in Figure 6.3. When PLD is not asserted, then the data inputs for each flip-flop are the data outputs of the adjacent flip-flip to the left. When PLD is asserted, then the bits in P become the inputs for each of the flip-flops.

Not shown in the figure are the clock enable inputs for the flip-flops. After the algorithm is complete, the values should be held constant; that is, neither shifting nor loading. The SHIFT and PLD inputs are connected through an OR gate to the clock enable inputs of each of the flip-flops. As a result, the register will only shift if SHIFT is asserted and will only do a parallel load if PLD is asserted. In any case, the parallel outputs are available as the P_OUT signal, and serial output is available as the S_OUT signal, which always corresponds to the LSB of the register.

	Used	Available	Utilization
Slices	29	13,312	0.2%
Slice Flip-Flops	32	26,624	0.1%
LUTs	49	26,624	0.2%
IOBs	44	221	19.9%
BRAMs	0	32	0.0%
GCLKs	1	8	12.5%

Table 6.4—Device utilization for a single serial shift register (including 32:1 multiplexer).

Since this design does not have the large amount of multiplexers required by the variable shift register described in 5.1.2, the resource requirements of this register are significantly reduced. The device utilization for one shift register is shown in Table 6.4. The table includes the bit-select multiplexer shown in Figure 6.1 that is used to perform the bit-shift operation. Only 29 slices are required for this design, compared to 90 for the parallel shift register. The parallel shift registers are each 210% larger than the serial registers. Since the parallel shift registers were used twice in the parallel design, this area reduction is the key to the small area requirements of the entire serial design.

6.1.3 The Lookup Table

The lookup table remains essentially unchanged. In the serial design, a multiplexer is added to choose the individual bits to be fed into the serial adder.

	Used	Available	Utilization
Slices	8	13,312	0.1%
Slice Flip-Flops	0	26,624	0.0%
LUTs	16	26,624	0.1%
IOBs	12	221	5.4%
BRAMs	1	32	3.1%
GCLKs	1	8	12.5%

Table 6.5—Device utilization for the serial lookup table.

The parallel lookup table did not require any slices. The entire design was implemented in one BRAM. Since the serial design requires a 32:1 multiplexer, this

implementation of the lookup table requires 8 slices. This small increase in area is more than offset by the savings in the registers and adders.

6.1.4 The Serial Adders

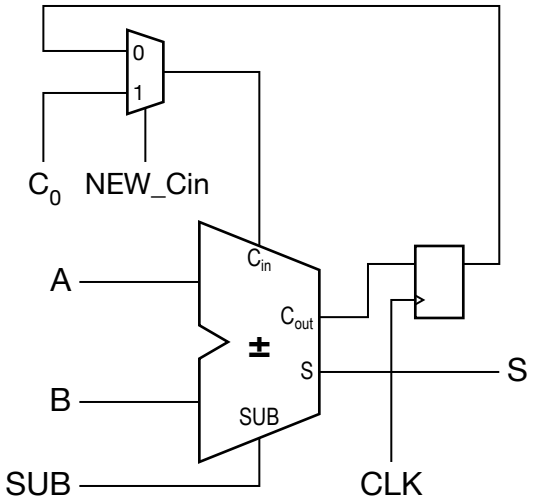

Figure 6.4—Design of the serial adder.

The adders that compute the X, Y, and Z values in the serial CORDIC unit are simple one-bit adder/subtractors with additional hardware to support serial arithmetic. The design of these adders is shown in Figure 6.4. The additional hardware is necessary to save the carry value from one stage and then to feed it to the next. Without this logic, the final sum would be incorrect since there would be no carry propagation.

The C_{out} output of the adder is connected to the input of a D flip-flop. The data output of the flip-flop is in turn fed into a 2:1 multiplexer. When a subtraction is being performed, it is necessary to have an initial carry of 1 to allow for the 2s complement of the B input. The NEW_Cin signal is generated by the control unit and is asserted only in the SETUP state and when a subtraction is called for. In the COMPUTE state, NEW_Cin is not asserted, allowing the carry out of the previous stage to be passed through to the carry in of the current stage.

The adder that computes the exponential value is a 32-bit parallel adder that uses the parallel outputs of the X and Y registers.

	Serial	Parallel	Available	Serial Utilization
Slices	4	16	13,312	0.0%
Slice Flip-Flops	1	0	26,624	0.0%
LUTs	7	32	26,624	0.0%
IOBs	7	97	221	3.2%
BRAMs	0	0	32	0.0%
GCLKs	1	1	8	12.5%

Table 6.6—Device utilization for the serial and parallel adders.

As the design would imply, the resource requirements for this component are quite small. Table 6.6's serial column shows that only 4 slices are required to implement this design. The parallel column shows the device utilization for the adder that computes the exponential function.

6.1.5 Design Summary

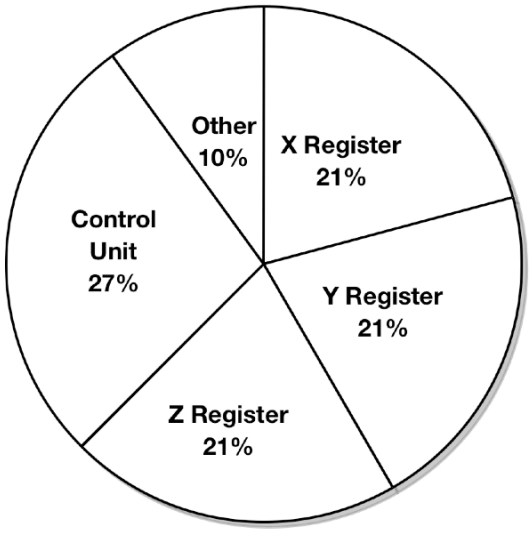

Figure 6.5—Slices used by the various components of the serial CORDIC unit.

This design is even more efficient at using the FPGA's resources, requiring much less of the chip's area than the parallel design. As Figure 6.5 shows, the distribution of

the chips resources to the components is very balanced, with a near even balance between the 3 registers and the control unit. Included in the "Other" category are the three serial adders, the 32-bit adder, and the arc hyperbolic tangent lookup table. With only 1% of the Spartan-3's slices used in this design, even larger applications can be supported. The serial nature of the design means that extra precision can be added to the system without having a large increase in area.

While the area requirements for the design are small, the timing requirements are not. For a system using n iterations (and r repeats) with a word size of w bits, then the total execution time of the unit will be $w(n+r)+1$ clock cycles. Each iteration will require w cycles to complete as the values are passed serially through the adders, and $n + r$ total iterations are completed after the algorithm finishes. The PRELOAD state of the control unit adds another clock cycle of execution time. The added clock cycles are partially offset by an increase in clock frequency. The linear convergence of the CORDIC algorithm means that the number of iterations should be close to the number of bits in the registers. A halving of the word size (and a corresponding halving of the iteration count) will reduce the execution time by a factor of 4.

This design, which performs 31 iterations, 2 repeated, with a word size of 32 bits will take 1,057 clock cycles to complete. At the theoretical maximum clock rate of 134.8 MHz, the unit will take 7.84 µs to compute the final value.

6.2 Simulation Results

This section presents the results of simulations performed using the design specified in 6.1. The design uses 32 bits to represent each word, with 4 bits for the integer portion, and 28 bits for the fractional portion of the value. The same simulations are performed

here as are done in 5.2, in order to demonstrate that this unit produces identical results, but at the cost of additional clock cycles.

6.2.1 Simulation 1: Computing *e* With CORDIC Growth Error

The first simulation performed demonstrates the effect of the CORDIC gain K_n. To compute the exponential function e^z, the X register must be initialized with 1, the Y register with 0, and the Z register with the desired argument to the exponential function. This simulation attempts to compute the value of e, so an initial value of 1 is placed into the Z register.

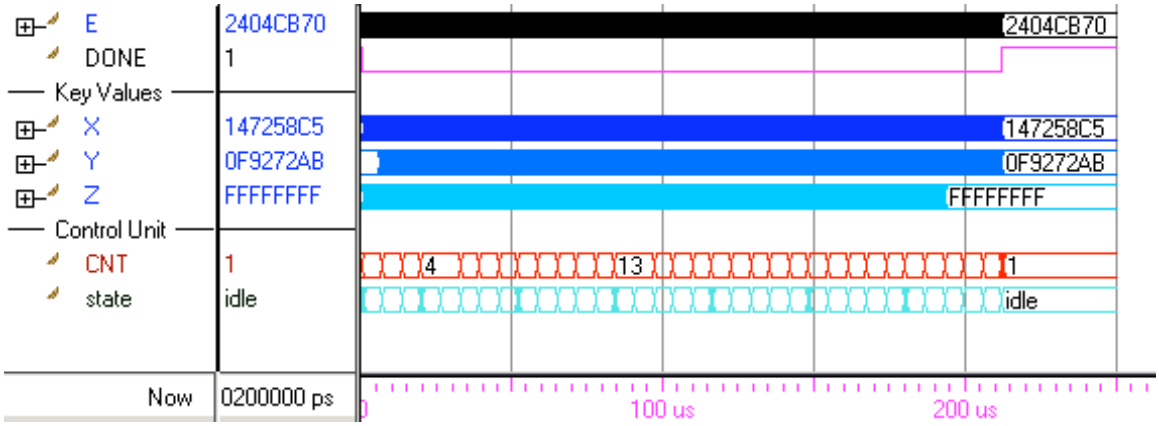

Figure 6.6—Results of a ModelSim simulation of the serial CORDIC unit attempting to compute the value of *e*. As a result of CORDIC growth, the final computed value is inaccurate.

The results of this simulation are shown in Figure 6.6. The algorithm iterated for a total of 34 times (including the two repeat iterations required in the hyperbolic domain). The final hexadecimal values in the X and Y registers are $147258C5_{16}$ and $0F9272AB_{16}$ respectively. When converted to decimal and scaled accordingly, they become 1.2779 for the X register and 0.9733 for the Y register. The value of the exponential function is the sum of these two numbers: 2.2512, corresponding to the value of the E signal.

These results correspond precisely to the values produced in 5.2.1. Whereas the parallel unit completes in under 8 µs, the serial unit takes just over 200 µs to produce the final value.

6.2.2 Simulation 2: Computing *e* Compensating for CORDIC Growth

In order to compute an accurate value of *e*, the growth observed in 4.2 must be accounted for. To produce an unscaled final value, the initial value must be divided by the growth factor. The formula used to compute the amount of growth is shown in (4.23). A precomputed value is shown in Table 4.1. For this simulation, rather than initializing X with 1, X will be initialized with the hexadecimal value $1351E872_{16}$, which equals 1.2075_{10}. Y is again initialized with 0 and Z with 1_{10}.

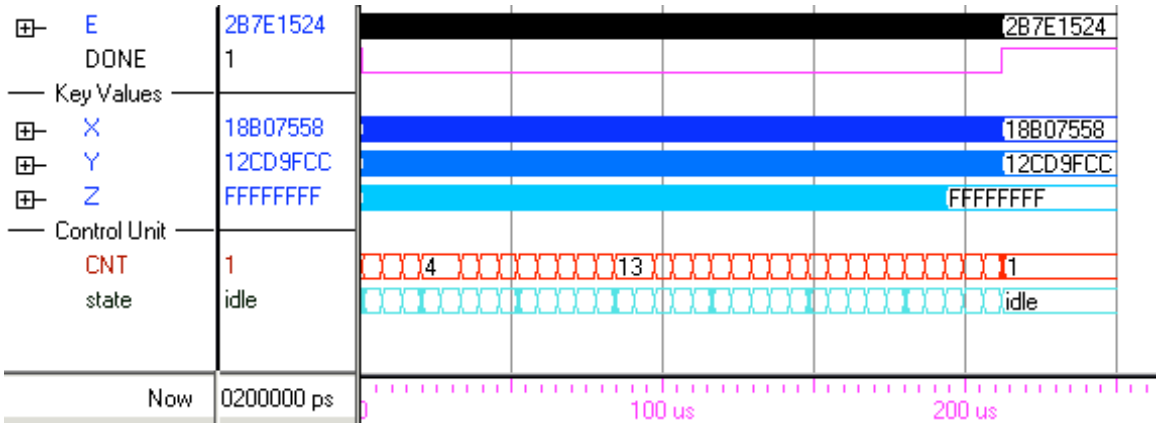

Figure 6.7—Results of a ModelSim simulation of the serial CORDIC unit computing the value of *e*. By accounting for CORDIC growth, the final computed value is accurate.

The results of this simulation are shown in Figure 6.7. The final computed value of *e* is output on the E bus. Its hexadecimal value $2B7E1524_{16}$ converts to 2.7183_{10}, which matches the actual value of *e*. When taken to 30 decimal places the value output in E is 2.718281880021100000000000000000, which differs from the real value of *e* by only 0.000000051562050185083300 0. These values are the same as the values produced by the parallel unit. As before, the only way to increase the precision is to increase the word

size and iteration count, which is easier to achieve in the serial design, but comes at the cost of longer execution time.

6.2.3 Simulation 3: Computing e^{-1}

It is also important for the unit to compute the value of the exponential function for negative arguments. Since the domain of convergence for the algorithm is symmetric about 0, half of the possible arguments are negative. This simulation computes the value of e^{-1}. X is initialized with 1.2075_{10} to ensure an unscaled result, and Y is initialized with 0. The Z register contains the argument for the exponential function and is thus initialized to –1 ($F0000000_{16}$ in 2's complement notation).

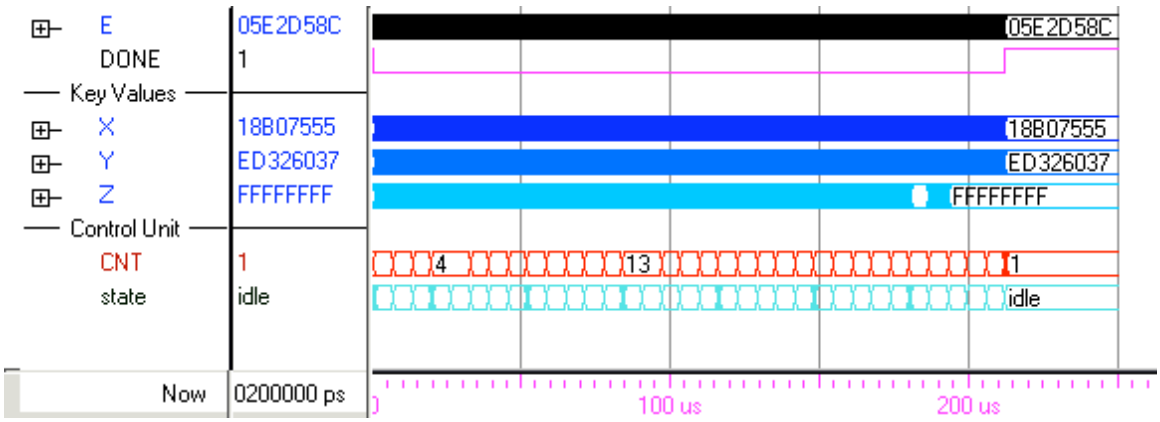

Figure 6.8—Results of a ModelSim simulation of the serial CORDIC unit computing the value of e^{-1}.

The simulation results presented in Figure 6.8 show that the CORDIC unit is able to handle the negative argument. The final value of the E signal ($05E2D58C_{16}$) corresponds to 0.36787943542_{10}, which is of the same accuracy as the computation for e^{1}.

6.2.4 Simulation 4: Computing Outside the Domain of Convergence

As discussed in 4.6, the CORDIC algorithm does not converge on an accurate answer for all input values. Table 4.3 shows the domains of convergence for all the

computation domains supported by the CORDIC algorithm. For the rotation mode, which this design uses, the domain of convergence is a limiter on the magnitude of the initial value of Z. If Z were initialized with a value whose magnitude is greater than 1.11817, then the final computed value would be incorrect. For this simulation, the X register is again initialized with 1.2075_{10}, and Y is again initialized to 0, but Z is initialized to 5_{10}, so that the unit will attempt to compute e^5.

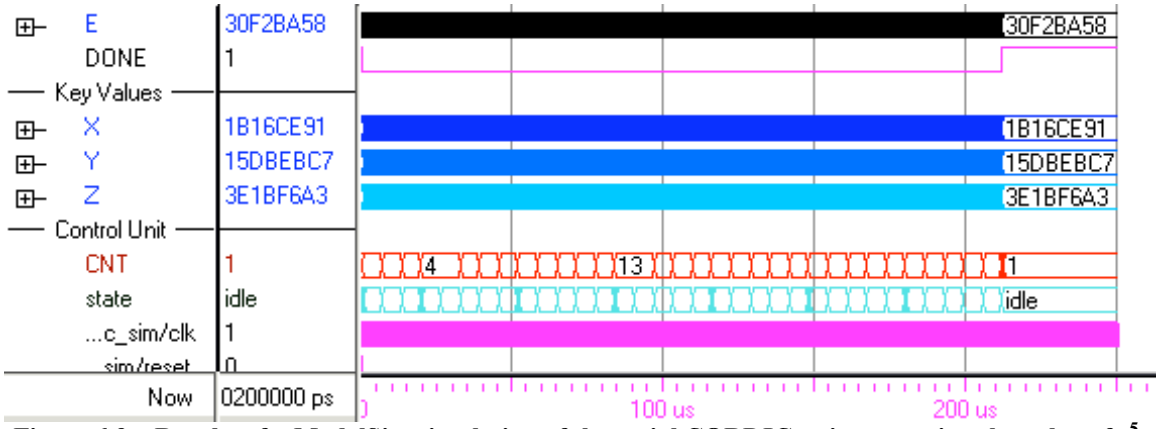

Figure 6.9—Results of a ModelSim simulation of the serial CORDIC unit computing the value of e^5. Since the initial value 5 is outside the domain of convergence, the computed value is incorrect.

The simulation results shown in Figure 6.9 demonstrate that the CORDIC unit is unable to compute e^5. The final value output by E is 3.0593_{10}, which is nowhere near the correct value (148.4132). 3.0593 acts is an asymptote for the computed values of E. Once the initial values of Z move outside the domain of convergence, the computed E value rapidly approaches 3.0593, and remains at that value no matter how large a value Z is initialized with. A similar asymptote exists for when Z is initialized with values that are too negative. In this case, the E values approach 0.3269.

6.3 Summary

For large applications where FPGA resources are scarce, the serial CORDIC unit described in this chapter offers many advantages over the parallel unit described in

Chapter 5. By operating only on single bits in a serial fashion, the complexity of the shift registers is reduced significantly. The size of the adders used in the design is also reduced since they only need to handle one bit at a time. Precision and accuracy are not sacrificed to achieve the reduction in resource utilization. However, the added execution time may prevent this design from being used in more time-sensitive applications. For those, the parallel design or a faster table-based approach would be better suited.

Chapter 7 Conclusion and Future Work

Artificial neural networks have the potential to become invaluable tools for systems designers looking to implement artificial intelligence into their work. The ability of these networks to be trained for a specific problem and to operate autonomously opens up a door to new ways of data analysis and other applications where it is difficult to design straightforward algorithms to produce the desired results.

FPGAs are likewise becoming increasingly useful for designers. The ability of the chips to be reprogrammed speeds prototyping and simplifies the design process for larger systems. The training procedure for artificial neural networks requires changes to internal weights of the network. The reprogrammability of the FPGA means that these changes can be made easily. The ability to place an artificial neural network in an embedded environment with real-world inputs further expands the number of applications available. Previously, it was very difficult to build a neural network in an FPGA because of the difficulty of fitting the arithmetic hardware on the same chip as the network.

Neural networks require complex mathematical support in order to facilitate the computation of the summation, weighting, and transfer functions. Most arithmetic algorithms are optimized for speed and require large amounts of chip area when implemented. Neural networks are inherently parallel computer systems and in order to use a large arithmetic unit, all the computations would have to be pipelined through one or several of these units. This would lessen the parallelism of the network, which also results in longer computation times.

The CORDIC implementations presented are compact enough that they potentially could be duplicated in the FPGA, allowing each neuron to have its own CORDIC unit. This maintains the parallelism of the network. Two alternate designs of CORDIC units have been presented, a parallel design and a much smaller serial design that takes longer to execute. Either of these could be used in a neural network at the neuron level.

7.1 Comparing the Designs

The parallel design fits in 329 of the Spartan-3's slices and can run at a theoretical maximum clock frequency of 75 MHz. The serial design is much smaller, needing only 139 slices, and can also run at almost twice the clock frequency: 134.8 MHz.

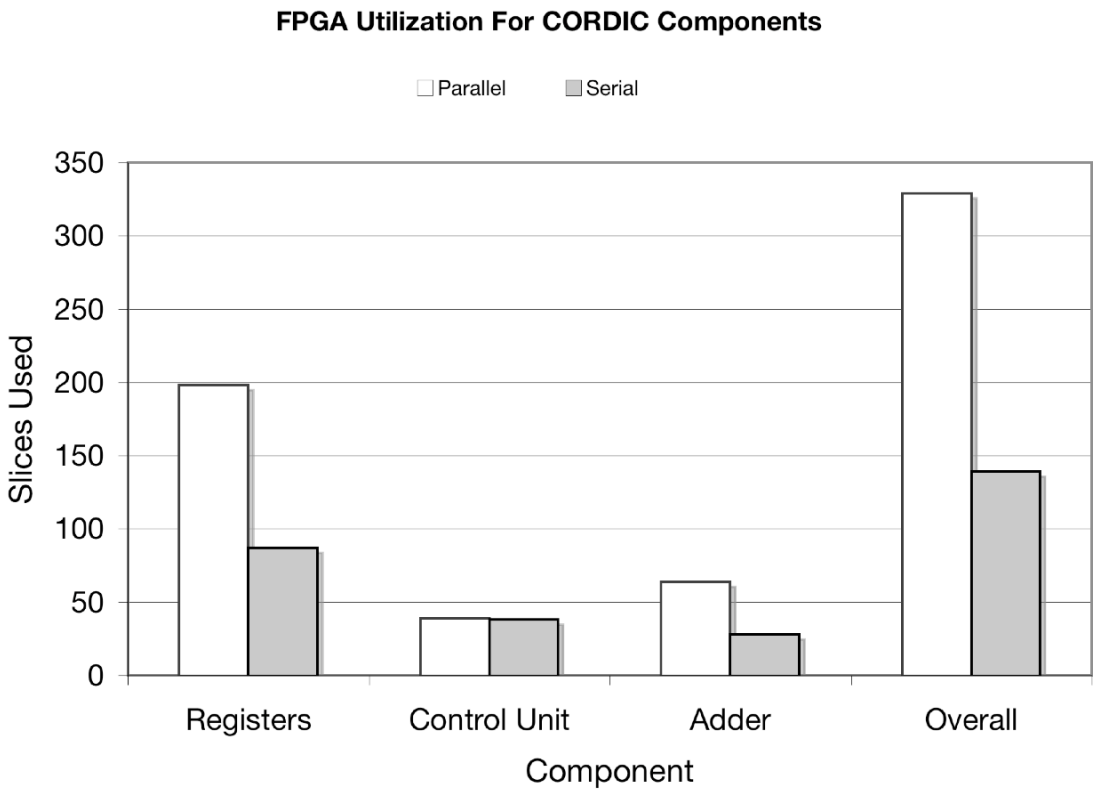

Figure 7.1—Slices required by the various components of the two CORDIC designs.

The parallel design requires a large amount of 32:1 multiplexers to accomplish the shifting. This is necessary because the amount by which the operands are shifted

increases each iteration of the algorithm. When switching to the serial approach, these multiplexers are eliminated resulting in significant resource savings. As seen in Figure 7.1, it is this savings that accounts for most of the efficiency of the design.

The effect of word size on the area requirement of the designs can be seen in Figure 7.2. The parallel design always requires more area than the serial design, and the number of slices required for the parallel design increases dramatically with each increase of the word size. The area requirement of the serial design increases roughly linearly, while the parallel design follows a sharp exponential curve.

The affect word size has on execution time is shown in Figure 7.3. The unit is assumed to be operating at the maximum clock frequency, as reported by the Xilinx synthesizer. This value is different for both the parallel and serial designs. The maximum clock frequency also decreases as word size increases, which further lengthens execution time with larger word sizes. The serial design is severely implacted by the increase in word size. Execution time increases exponentially with word size, while execution time for the parallel design follows a roughly linear track.

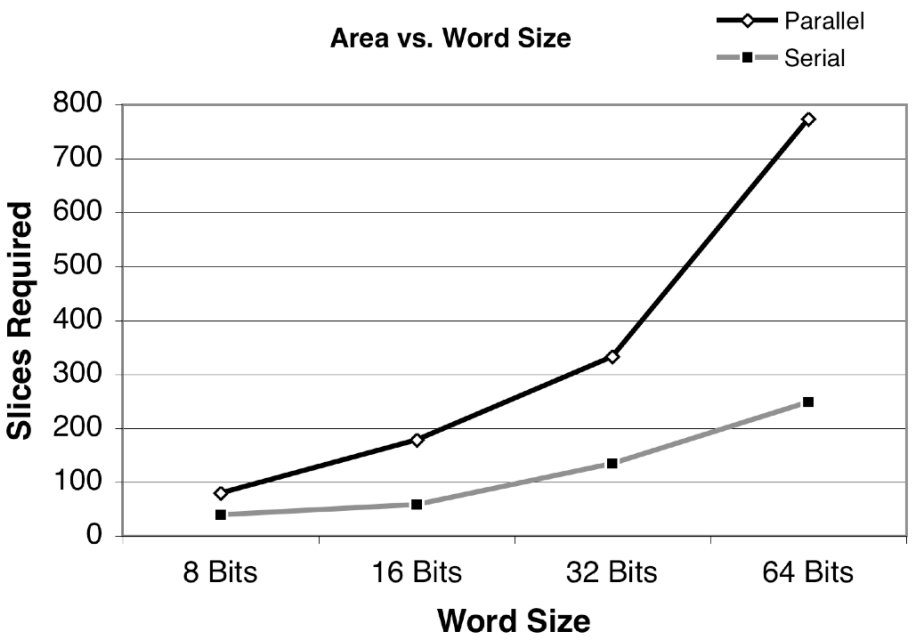

Figure 7.2—The effect of word size on the FPGA chip area requirements of the CORDIC unit.

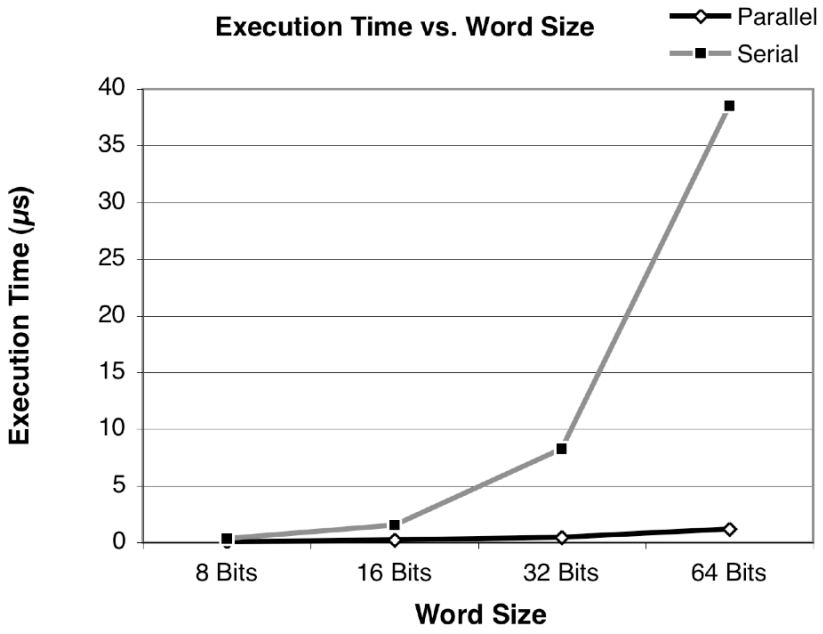

Figure 7.3—The effect of word size on the execution time of the CORDIC unit.

As can be seen by comparing the two figures, the area and timing requirements for these designs are inversely proportional. This is true for most designs. In general, when design changes are made that reduce the timing requirements for the system, these same changes result in an increased area requirement. The move to bit-serial arithmetic, which

optimized the algorithm for area, requires each iteration of the algorithm to take longer. Each bit in the operands will use one clock cycle to compute the new value. For these designs, which use the word sizes of 32 bits and 31 iterations (with 2 repeats), the 34 clock cycles that the paralle implementation requires are increased to 1,057 cycles. This is an increase of 3000%. This is offset somewhat by the increase in clock frequency, but the serial implementation still takes significantly longer to complete.

For applications that can tolerate less precision, further savings in both area and computation time can be achieved. Neural networks in particular are probabilistic in nature and may not need 32 bits of precision. Decreasing the word size to 16 bits will drastically reduce the area requirement for the parallel design, since its area requirement decreases exponentially with word size. The area requirement of the serial design depends only linearly on the word size, so the area savings would not be as significant. The execution time, however, would decrease exponentially.

Both implementations also have complex control units that are designed with flexibility in mind. It is possible that the clock frequency for both designs could be increased if the control unit could be simplified for a specific application.

7.2 Integration With Artificial Neural Networks

7.2.1 CORDIC Artificial Neuron

This section discusses the design of a basic artificial neuron that uses the CORDIC unit to compute its transfer function. Figure 7.4 shows the design of this neuron. This sample design uses 4 inputs, but the design can be scaled to accommodate any number of inputs.

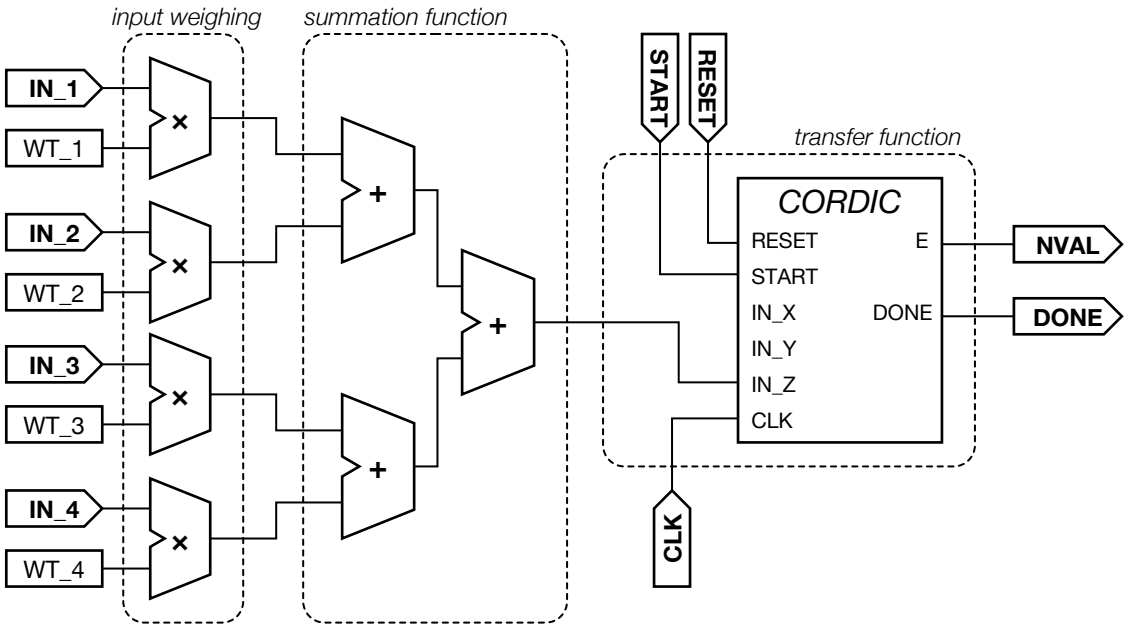

Figure 7.4—Block diagram of a basic artificial neuron utilizing the CORDIC unit to compute the transfer function e^x.

The 4 inputs are weighted by constants that are stored internally in the neuron. A separate multiplier computes the weighted value for each input in parallel. As per the artificial neuron model shown in Figure 2.1, these weighted values are then passed through a summation function. Three adders are cascaded to add the four values together. The CORDIC unit then computes the transfer function. The output of the final adder is used as the argument to the function. In this case, since the CORDIC unit has been designed to compute e^x, the sum is connected to the IN_Z input. IN_Y is initialized to 0, and IN_X is initialized to 1.2075 to ensure an unscaled result. The START and RESET signals may be provided externally or may be hard-wired to high or low. The E output of the CORDIC unit then becomes the output of the neuron (NVAL) and may be connected to any number of neurons in the next layer of the network.

The multipliers can be of any design, but since all inputs use a fixed point, the output must be shifted accordingly. The design used here has 28 bits reserved for the fractional

component of the values, so the final product must be shifted to the right 28 bits to ensure that it is in the format expected by the CORDIC unit.

	Parallel	Serial
Slices	661 (5.0%)	463 (3.5%)
Slice Flip-Flops	123 (0.5%)	141 (0.5%)
LUTs	1,273 (4.8%)	893 (3.4%)
IOBs	163 (73.8%)	163 (73.8%)
BRAMs	1 (3.1%)	1 (3.1%)
18×18 Multipliers	16 (50.0%)	16 (50.0%)
GCLKs	1 (12.5%)	1 (12.5%)

Table 7.1—Device resource requirements for the artificial neuron designs using both the parallel and serial CORDIC units.

The resource requirements of this design are shown in Table 7.1. The design was synthesized using both the parallel and serial CORDIC units. The neuron with the parallel CORDIC unit requires 661 of the Spartan's slices, while the serial design only needs 463. The added area requirement for both designs comes from the 3 adders for the summation function and the additional routing logic required to interconnect the multipliers and adders. The neuron is able to benefit from the Spartan's built-in multipliers. For larger networks, further chip area will be required to implement the multiplication function. The serial design is able to operate at a maximum theoretical clock frequency of 140.0 MHz, and the parallel design can operate at a theoretical maximum frequency of 77.9 MHz. The area/time tradeoff holds true for the neuron design just as it did for the CORDIC unit alone. The 32-bit adders present in the serial design are fast enough to allow the neuron to still operate at the faster clock rate.

It should be noted that the serial neuron still uses parallel multipliers to compute the weighted inputs, and serial adders to compute the summation function. It should be possible to use both serial adders and multipliers to gain additional area savings.

Depending on the designs used, modifications to the CORDIC design may also be required should a designer use this approach.

Results of a ModelSim simulation of the design are shown in Figure 7.5. The four imputs used in the simulation are 0.3 ($04CCCCCC_{16}$), 0.02 ($0051EB85_{16}$), 2.1 (21999999_{16}), and 0.65 ($0A666666_{16}$). Their respective weights are 1, 2, –0.5, and 0.25. The signals in1_w through in4_w show the results of multiplying the inputs by their weighting constants.

The signal sum_all shows the result of the summation function: the hex value $F73D70A3_{16}$ or –0.547500. The signal nval is the final output of the neuron and its hex value of $09411A07_{16}$ (0.578394(matches the result of $e^{-0.5475}$.

Since the implementation uses the Spartan's internal multipliers, the intermediate results of computing the weighted values are not available. The only delay in the simulation is from the CORDIC unit's execution. A design utilizing custom multipliers would have increased execution time.

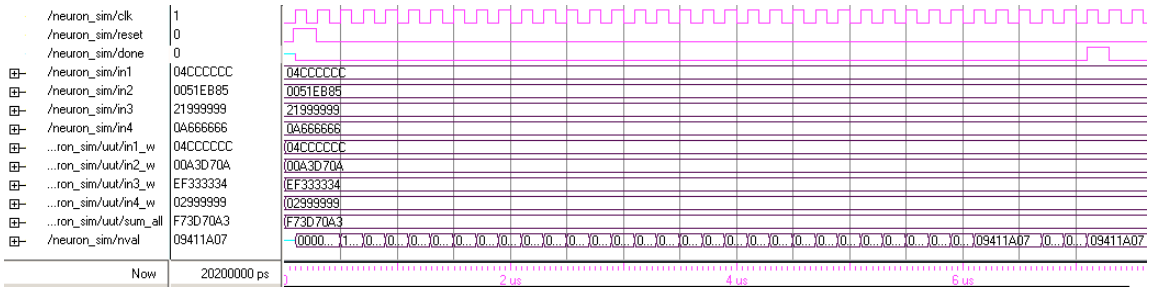

Figure 7.5—Simulation of an artificial neuron using the 4 inputs (0.3, 0.02, 2.1, 0.65) and the weights (1, 2, –0.5, 0.25).

7.2.2

For any implementation of an artificial neural network into a programmable device such as an FPGA, at least two possibilities exist for implementing the arithmetic

computations: a high-speed lookup table (either on-chip or off) or either of the CORDIC implementations discussed here.

Table-based implementations work similar to the logarithm tables found in the back of textbooks, requiring interpolations between rows. These designs can be very fast and accurate, but require large amounts of chip area. For large networks, it would require a much larger (and more expensive) FPGA to allow the lookup table to coexist with the network. Additionally, all arithmetic calculations would have to be queued, slowing down the operation of the entire network. If the table implementation is fast enough and the network small enough, then performance may be comparable to a CORDIC implementation.

The CORDIC algorithm is powerful enough and small enough to be able to support a neural network, ideally with a CORDIC unit in each neuron. This would maintain the parallelism that is key to the operation of the neural network. The longer computation time is not a major shortcoming, since the power of a neural network lies in its parallelism. The fact that all neurons can be computing in parallel and all neurons can finish processing their inputs at the same time allows for a natural progression of data through the network.

The flexibility of the algorithm with its two computation modes that can each operate in one of three domains means that the same unit can compute a wide variety of functions. This gives the network designer the flexibility to vary the transfer functions used with only minimal changes and no redesigning necessary. It would also be possible for the CORDIC unit to perform the multiplication of the weighting factors with the

inputs, though for neurons with large amounts of inputs, this would come with a severe performance penalty.

The systems designer would choose the proper implementation that best fits the size of the network that would be supported. The fast table-based approaches would be preferred because of their high performance, but only the smaller networks would be able to effectively utilize them. For medium-size networks, the parallel CORDIC unit is the best compromise between size and speed. And for large networks, the serial implementation may be the only choice.

7.3 Future Study

Further research can be done into the benefits that can be derived from using the CORDIC unit in a neural network. Study of a live, practical neural network will help in understanding how large a network could be supported and which implementation s best suited for networks of varying size.

Students may also wish to study ways that the CORDIC unit can be further optimized specifically for use in neural networks. The control unit is one component that is an easy target for optimization, but further area reduction may be possible when the unit is integrated into the structure of a neuron. This also includes determining the ideal word size for the application. The accuracy of the network needs to be taken into account when determining the precision of the underlying hardware.

The CORDIC unit presented here is hard-wired to the hyperbolic rotational mode. A truly complete general unit that is capable of operating in all three domains in both modes should be possible with only a little more hardware. A study of the resource requirements of this general-purpose unit could also be done.

References

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

[2] J.E. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Trans. Electronic Computers*, vol. EC-8, no. 3, pp. 330–334, Sept. 1959.

[3] H. Dawid and H. Meyr, "CORDIC Algorithms and Architectures," in *Digital Signal Processing for Multimedia Systems*, ed. by K.K. Parhi and T. Nishitani, Marcel Dekker, 1999, pp. 623–655.

[4] R. Andraka, "A survey of CORDIC algorithms for FPGA-based computers," in *International Symposium on Field Programmable Gate Arrays*, 1998.

[5] Y. H. Hu, "The Quantization Effects of the CORDIC Algorithm," *IEEE Trans. Signal Processing*, vol. 40, pp. 834–844, July 1992.

[6] IEEE Std. 754-1985, "Standard for Binary Floating Point Arithmetic," 1985.

[7] A. A. Liddicoat and L. A. Slivovsky. *Programmable logic*. 3rd Edition of EE Handbook. CRC Press.

[8] Xilinx. Spartan-3 FPGA family: Complete data sheet. Datasheet DS099, Xilinx.

[9] M.M. Mano and C.R. Kime, *VLSI Programmable Logic Devices,* Pearson Prentice Hall, Upper Saddle River, NJ, 3rd edition, 2004. Supplement to *Logic and Computer Design Fundamentals*.

[10] D. Anderson and G. McNeil, "Artificial Neural Networks Technology," DoD Data & Analysis Center for Software, August 1992.

[11] M. Skrbek, “New Neuochip Architecture,” Doctoral Thesis. 115 p.CTU, Faculty of Electrical Engineering, Prague, 2000.

[12] J. Zhu and P. Sutton, “FPGA Implementations of Neural Networks — a Survey of a Decade of Progress,” *Proceedings of 13th International Conference on Field Programmable Logic and Applications (FPL 2003)*, Lisbon, Sep 2003.

[13] M. Figueiredo and C. Gloster, “Implementation of a Probabilistic Neural Network for Multi-spectral Image Classification on an FPGA Based Custom Computing Machine,” *Proceedings of 5th Brazilian Symposium on Neural Networks*, Dec. 1998.

[14] S.L. Bade and B.L. Hutchings, “FPGA-Based Stochastic Neural Networks—Implementation,” *Proc. of IEEE Workshop on FPGAs*, 1994.

[15] K. Nichols and M. Moussa, “Feasibility of Floating-Point Arithmetic in FPGA based Artificial Neural Networks,” in *Proceedings of the 15th International Conference on Computer Applications in Industry and Engineering*, 2002.

[16] J.L. Holt and T.E. Baker, “Back propagation simulations using limited precision calculations,” in *Proceedings of International Joint Conference on Neural Networks*, 1991, pp 121–126 vol. 2.

[17] S. Draghici, “On the capabilities of neural networks using limited precision weights,” *Neural Networks*, 2002, 15: p. 395-414

[18] Y. Wu and S.N. Batalama, “An Efficient Learning Algorithm for Associative Memories,” *IEEE Trans. Neural Networks*, vol. 11, no. 5, pp. 1058–1066, Sept. 2000.

[19] S. Halgamuge, “A Trainable Transparent Universal Approximator for Defuzzification in Mamdani-Type Neuro-Fuzzy Controllers,” *IEEE Trans. Fuzzy Systems*, vol. 6, no. 2, pp. 304–314, May 1998.