The Derivation and Justification of a CORDIC Implemented Artificial Neuron.

by

Martine Bruce Wedlake
BaSC, Portland State University, 1992

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF APPLIED SCIENCE
in the
Department of Electrical and Computer Engineering

We accept this thesis as conforming
to the required standard

[redacted]

Dr. H. L. L. Kwok, Supervisor(Dept. of Electrical and Computer Engineering)

[redacted]

Dr. K. Li, Departmental Member(Dept. of Electrical and Computer Engineering)

[redacted]

Dr. N. Horspool, Outside Member(Dept. of Computer Science)

[redacted]

Dr. D. Shpak, External Examiner(Dept. of Engg., Royal Roads Military College)

Supervisor: Dr. Harry L. L. Kwok

Abstract

Hopfield's network[31, 32], and Widrow and Hoff's[63] Adaptive Linear Element (Adaline) network demonstrated early on that artificial neural networks were versatile; however, the Hopfield network's limited information capacity and the Adaline's inability to train multiple layers limited these systems to simple networks. The field of artificial neural networks has blossomed in the past few years due, in part, to the pivotal results of Rumelhart's Back Propagation network[51]. With his training methodology we are able to construct meaningful neural networks and train them reliably. The Back Propagation network addresses these problems by extending the single layer of neurons into a multi–layer perceptron. This architecture improves information capacity and also allows for arbitrary mapping functions between the input and output spaces[40].

Digital neural networks are not well represented in the literature, as compared with software implementations. To fill in the gap, this work applies the CORDIC technique to the field of neural networks to construct digital neurons.

The resulting CORDIC neuron is an enabling technology; in of itself a neuron is barely interesting, but the CORDIC neuron enables full neural network systems to be constructed. The CORDIC method is particularly powerful for neural networks because of its ability to calculate the nonlinear activation function in a straightforward manner whereas most digital neural networks use ROMs for the activation function.

The CORDIC neuron provides a rich set of topics to examine. Of particular interest in this work is the sequence of CORDIC steps: the CORDIC schedule. Both static and dynamic CORDIC schedules are considered and evaluated. Static CORDIC neurons are made initially using emulation software, and later on the Xilinx XC4003 FPGA chip.

Examiners:

[redacted]

Dr. H. L. L. Kwok, Supervisor(Dept. of Electrical and Computer Engineering)

[redacted]

Dr. K. Li, Departmental Member(Dept. of Electrical and Computer Engineering)

[redacted]

Dr. N. Horspool, Outside Member(Dept. of Computer Science)

[redacted]

Dr. D. Shpak, External Examiner(Dept. of Engg., Royal Roads Military College)

Table of Contents

List of Tables

List of Figures

Acknowledgements

I wish to thank my parents who have given me the opportunity to pursue my ambitions.

I also give thanks to the teachers and professors who have inspired me; in particular, I thank Dr. Duncan MacFarlane for showing me that learning is an adventure.

I acknowledge the Micronet Centre of Excellence for their generous support.

I dedicate this work to Linda.

Chapter 1

Introduction

The feed forward artificial neural network described later in the chapter is based, in part, on models of biological neural networks. While the author does not intend to provide an in–depth analysis of biological neurons, some background is necessary to show the derivation of artificial neurons.

1.1 Biological Neural Networks

The field of artificial neural networks began as a mathematical model for neural biologists, psychologists and others studying the central nervous system, the brain, and even human psychology[43, 63, 50, 28, 35][1].

[1]The work of James[35] is astounding given the contemporary nature of his ideas even though his book was published over a hundred years ago.

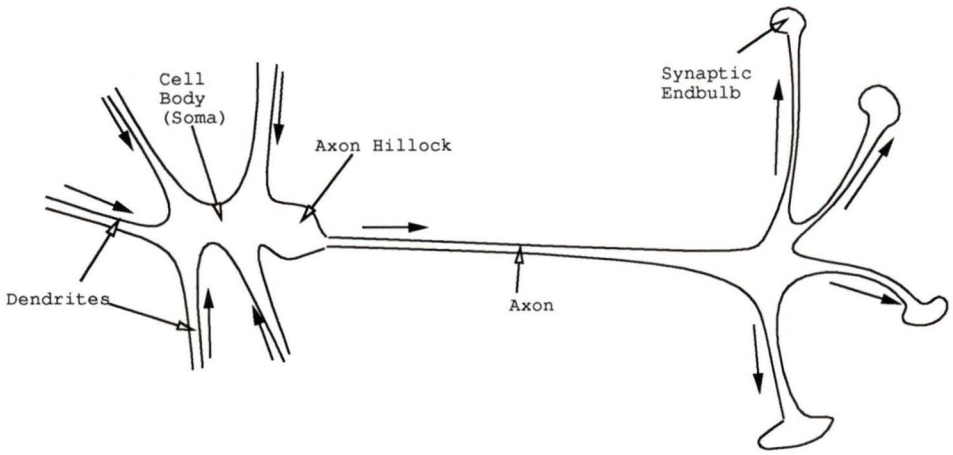


Figure 1.1: A Biological Neuron. The closed arrowheads show the direction of information flow from the dendrites, through the axon hillock, and out the synaptic endbulbs.

1.1.1 Neural Structures

The system level view of human beings has three primary components: receptors, effectors, and neurons. Receptors are special cells that convert observable phenomena (light, sound, touch, etc.) into electrical charges that propagate through the natural neural network. Effectors contract muscles in response to an electrical charge. They allow us to move about, secrete gastric fluids, and maintain hormonal levels in the blood stream — in general, effectors are the output devices that allow us to interact with the world around us.

A typical neuron is shown in Fig. 1.1. Information is transmitted by inducing charges along cell membranes. These charges move along the axon and dendrites, much like heat conducts through a metal rod[2]. These charges move through the

[2]This is in contrast to the way electricity move through metals. Electric charges moving through metals push valence electrons from one atom to another, while charges moving through a neuron induce potential differences that move along the length of the neural structure – much like a chain of capacitors[5, 6].

dendrites and accumulate in the axon hillock. When the accumulated charge reaches a critical amount (the action potential), the neuron fires. This sends a rush of charge down the axon to the synaptic endbulbs.

Since the electric charge leaks out of the cell just as a capacitor leaks current, the charges must arrive from the dendrites within a certain time window to be effective. If they arrive far apart from each other, the charge collected in the axon hillock would bleed off before the next charge arrives, thereby preventing the action potential from forming.

Therefore, neurons are not steady state devices – they operate in the frequency domain. The work of McCulloch and Pitts[43], Hopfield[31], and others describe neuron activity in terms of firing rate to simplify the modeling process. This work follows this convention.

A natural limit to the firing rate is the recharging time, or refractory period, for the neuron. The refractory period imposes an upper bound on the rate a neuron can fire, effectively squashing the output range. This effect is called the activation function in the neural net literature[5, 6, 13].

The synaptic endbulbs act as interfaces from one neuron to another. This interface can be strong or weak, excite or inhibit (meaning it can help the target neuron accumulate the action potential or try to prevent it from reaching the action potential). Evidence shows that the synaptic endbulbs change in response to the activity of the neuron[5, 6]. This effect was known as early as 1949, when Hebb postulated, "When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased."[28]

From the biological neuron, McCulloch and Pitts[43] devised a mathematical model of the neuron. This early work was extended by Widrow and Hoff[63] to

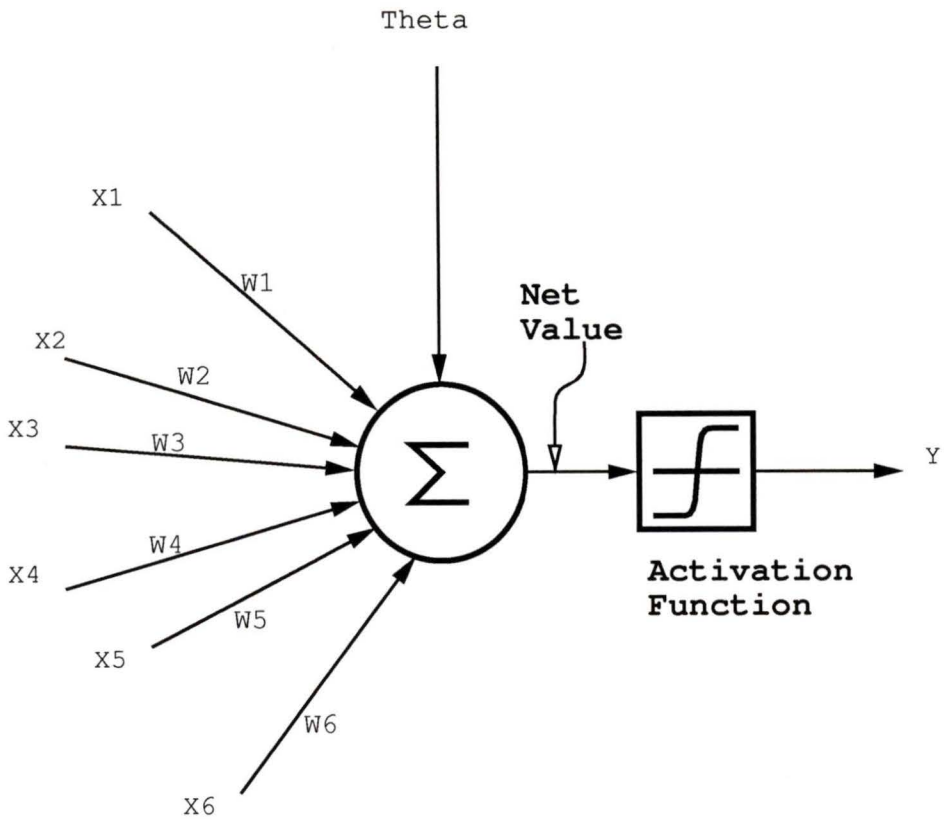


Figure 1.2: The Artificial Neuron. Please note that some authors associate the neuron bias with the activation function rather than the net function.

develop the Adaptive Linear Element (ADALINE) model of neural networks. My choice of artificial neuron stems from this work.

1.2 Modeling of Neural Structures

For the purposes of this thesis, the basic artificial neuron is described as in Fig. 1.2. This is Widrow and Hoff's ADALINE[63] (see also Minsky and Pappert [44]), but with a sigmoid activation function rather than a hard limiter. It is apparent from the figure that the behavior of a single neuron can be represented by a few simple

equations:

$$u = \sum_{j=1}^{N} w_j x_j + \theta \tag{1.1}$$

$$y = \psi(u) \tag{1.2}$$

where w_j represents the synaptic weight for the jth input, θ represents a neuron bias (think of it as a charge that exists in the axon hillock prior to synaptic activity), the net value u represents the accumulation of charge in the soma, y is the output of the neuron bounded by a maximum excitation and maximum inhibition, and finally $\psi(\cdot)$ is the activation function. The activation function decides the firing behavior for the neuron.

Activation functions are decision functions; they drive the output of the neuron high or low[3]. In the current literature there is a wide variety of activation functions[38, 13, 40] to choose from; for example, hard limiter functions decide between high and low on a knife edge, while the saturation limiter ramps linearly between low to high, and the sigmoid (unipolar and bipolar) functions decide on a smooth squashing–like curve.

For my work, the activation function is defined to be the hyperbolic tangent function because: (1) the hyperbolic tangent function is frequently used in the neural network community due to its smooth and positive derivative, and (2) sufficient challenges exist to design a hardware algorithm for implementation. As will be shown soon, the backprop learning method relies on the steepest descent optimization technique. In order to produce incremental changes in the weights, the activation function's derivative must be well defined and positive. No learning takes place when the derivative of the activation function is zero. The hyperbolic tangent function meets these criteria.

[3]Since some activation functions drive the output to either one or zero, but some others drive between minus one and positive one the generic terms "high" and "low" are used.

1.2.1 Single and Multi–layer Perceptrons

A multi–layer perceptron[50, 40, 38, 13], also known as a feed forward neural network, is made up from layers of neurons transmitting information from the input layer toward the output layer. Figure 1.3 shows a two–layer perceptron with twelve inputs, and three outputs. The boxes represent neurons and the arcs are the dendrites/synaptic–bulb connection (and, the associated weights). One can think of the input vector (x_i) as receptors, and the output vector (y_i) as effectors from the biological neural network section.

Multi-layer perceptrons are made from distinct neural layers with connections between adjacent layers. There are no connections flowing backward (feedback), or bypassing layers. Furthermore, the neurons have the same net function and activation function. This interpretation is not meant to exclude other possibilities, but rather to limit the scope of this work.

The neural network literature is full of other interpretations; for example, Recurrent Neural Networks allow feedback paths, many types of perceptrons allow connections to bypass neural layers, and the Hopfield network allows an arbitrary connection between neurons (no layers at all). It is not feasible for me to define all the various forms of artificial neural networks, however the main ideas in this work also apply well to alternative neural architectures.

Furthermore, the simple multi–layer perceptron defined above is the most popular form of artificial neural network today — even with the stiff competition from other architectures. Multi-layer perceptrons are used for speech recognition, computer vision, process control, and handwriting recognition to name a few examples[40].

The network and activation equations, Eqs. 1.1 and 1.2, show the operations of a

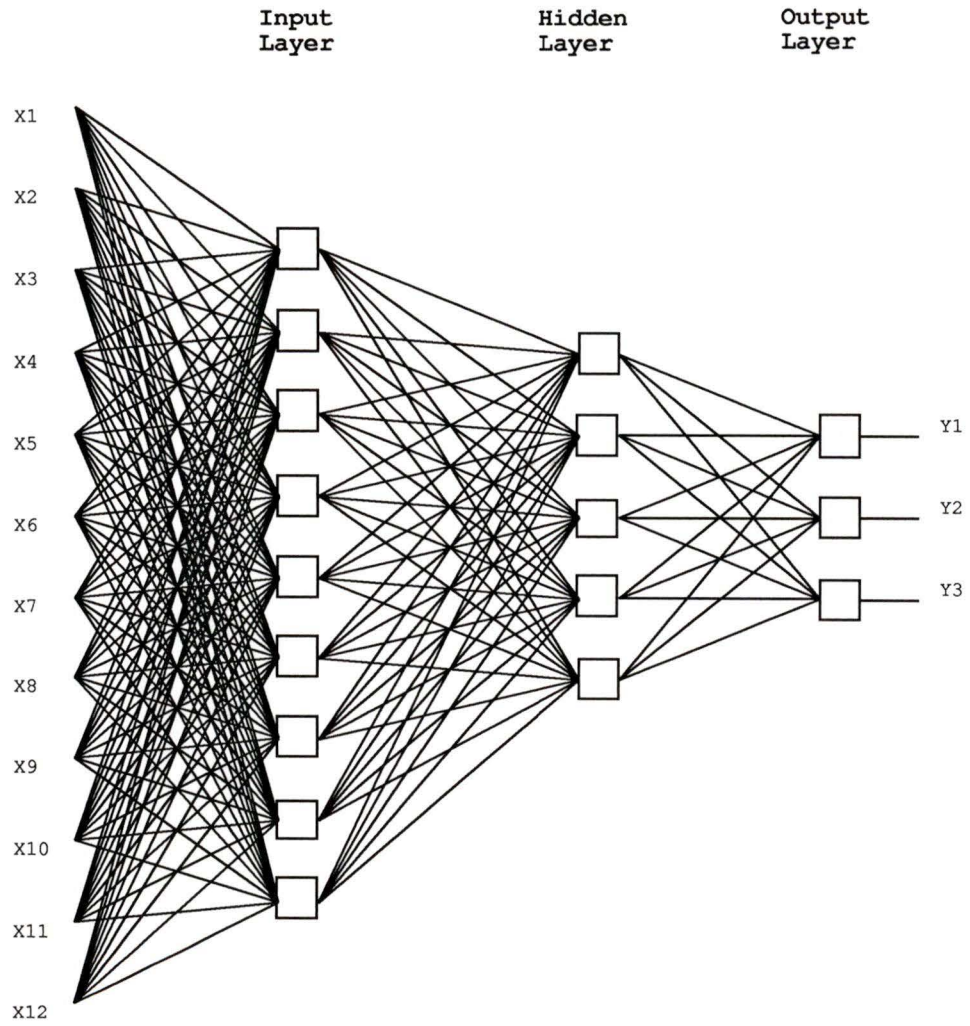


Figure 1.3: The Multi–Layer Perceptron.

single neuron. The equations are extended to include layers of neurons:

$$u_i^{[\alpha]} = \sum_{j=1}^{N} W_{ij}^{[\alpha]} x_j^{[\alpha]} + \theta_i^{[\alpha]} \tag{1.3}$$

$$y_i^{[\alpha]} = \psi(u_i^{[\alpha]}) \tag{1.4}$$

or,

$$\mathbf{u}^{[\alpha]} = W^{[\alpha]}\mathbf{x}^{[\alpha]} + \mathbf{\Theta}^{[\alpha]} \tag{1.5}$$

$$\mathbf{y}^{[\alpha]} = \Psi(\mathbf{u}^{[\alpha]}) \tag{1.6}$$

See Eqs. 1.3 and 1.4 for the standard form and, Eqs. 1.5 and 1.6 for a shorter matrix–vector form. The first form is more prevalent in the neural network community, however the second form is better suited for mathematical analysis software such as Matlab, Maple, and Mathematica.

The index, i, on the net value u and output y identifies distinct neurons within a single layer. In the matrix–vector notation, a vector is used for the entire layer of outputs (thereby making the index implicit). The weight vector of Eq. 1.1 extends to a matrix where the element W_{ij} is the weight of the synaptic connection between neuron i and input j[4]. Another way to look at W is to realize that the rows (indexed by i) are the weight vectors of Eq. 1.1.

The parameter, $[\alpha]$ is introduced into the equations to identify distinct layers. The values for α range from one (the input layer), to $\mathcal{N}$ the last layer (typically three or less)[5]. This notation implies that $\mathbf{y}^{[\alpha]} \equiv \mathbf{x}^{[\alpha+1]}$.

[4]The input can either be a direct input (similar to a receptor), or another neuron. The author chose to simply call it an **input** since it does not matter where the input comes from.

[5]Note: layers are defined in terms of synaptic layers, not neuron layers.

1.2.2 Determining the Synaptic Weights of a Multi–Layer Perceptron

Building the topology of the artificial neural network is only half the work required to make a working artificial neural network. The synaptic weights are also needed prior to using the network.

One method to obtain the weights is by training the network on a set of known data. This is called the supervised learning method because a 'teacher' alters the weights of the network, enabling it to learn a particular training set of input–output vector pairs.

Training on a large enough training set makes the network generalize to inputs outside of the original training set. It's the ability to learn classifications outside the scope of the original training set, and the ability to handle input noise that sets artificial neural networks apart from more traditional pattern recognition approaches[56].

This methodology does have some disadvantages: (1) there must be a moderate to large training set, (2) due to the iterative nature of the algorithm and the size of the training set the training time can be lengthy, and (3) many supervised training algorithms do not handle new training vectors gracefully – if you wish to add new vectors, then you must retrain the network using the larger training set.

The Backward Propagation of Error Learning Algorithm

Because of its simple–to–implement and practical nature, the Backward Propagation of Error algorithm (Backprop), introduced by Rumelhart[51], is the most popular training algorithm today.

The steepest descent technique minimizes the network error (defined below) in the Backprop algorithm. The slow convergence of the steepest descent optimization

technique[23, 46] has prompted the development of faster optimization techniques such as the conjugate gradient and quasi–Newton methods. See Kung[38] pp. 152–167 for an overview of the various training schemes. These methods require much more computational power than the Backprop algorithm and also require more knowledge of the error function's derivatives (gradient for conjugate gradient and the Hessian matrix for the quasi–Newton methods). These downsides tend to offset the advantage of a faster convergence rate. Furthermore, steepest descent algorithms are guaranteed to converge to a local minimum[23][6], while there do not exist similar proofs for quasi-Newton and conjugate gradient techniques[23]. Since the error surface is unknown (it depends on the weights which are changing during each iteration), it is comforting to use a proven algorithm such as the Backprop.

Rumelhart's[51] Back Propagation of Error algorithm defines the error function for the pth pattern on the output layer ($\alpha = \mathcal{N}$) as the following:

$$E_p = \frac{1}{2}\sum_{j=1}^{n_\mathcal{N}} (d_{ip} - y_{ip}^{[\mathcal{N}]})^2 = \frac{1}{2}\sum_{i=1}^{n_\mathcal{N}} e_{ip}^2 \tag{1.7}$$

where d_{ip} is the desired output for the i^{th} neuron of the p^{th} pattern.

For each input pattern, a different error surface is obtained. The approach takes a small step in the direction of the steepest descent for each pattern and after many iterations through the input patterns the global minimum for the network[32] is found.

The Backprop algorithm is quoted in many books on neural networks[38, 13]; the equations here are from Cichocki[13], slightly altered to fit Eqs. 1.3 and 1.4[7].

[6]It is important to emphasize that finding a local minimum does not always mean that a good network has been found. Sometimes, the Backprop algorithm will find a local minimum that is not close to the global minimum, therefore the network performance is poor. The guarantee of locating the local minimum is a useful property because it ensures that the algorithm will not oscillate, or jump spontaneously to unexpected regions within the weight space.

[7]The author chose not to remap the Backprop algorithm into the matrix–vector notation because Eqs. 1.8–1.11 are well known and remapping them would do little but confuse the reader.

For the output layer, $\alpha = \mathcal{N}$.

$$\Delta w_{ij}^{[\alpha]} = \eta \delta_i^{[\mathcal{N}]} y_i^{[\mathcal{N}-1]} \tag{1.8}$$

$$\delta_i^{[\mathcal{N}]} = e_{ip} \frac{\partial \psi_i(u_i^{[\mathcal{N}]})}{\partial u_i^{[\mathcal{N}]}} \tag{1.9}$$

And, for the hidden layers, $1 \leq \alpha < \mathcal{N}$.

$$\Delta w_{ij}^{[\alpha]} = \eta \delta_i^{[\alpha]} y_i^{[\alpha-1]} \tag{1.10}$$

$$\delta_i^{[\alpha]} = \frac{\partial \psi_i(u_i^{[\alpha]})}{\partial u_i^{[\alpha]}} \sum_{j=1}^{n_\alpha} \delta_j^{[\alpha]} w_{ji}^{[\alpha]} \tag{1.11}$$

Recall that $y_i^{[0]} \equiv x_i^{[1]}$, therefore Eq. 1.11 is also used for the input layer.

The learning rate, η, regulates the step size taken in each iteration. If this value is too large, the changes to the weights are too severe and the network may take longer to converge or may even oscillate; however, a small learning rate limits the speed of convergence. Typical values rest between 0.5 and 1.0.

1.3 Network Design and Training

With Eqs. 1.1–1.2 defining the individual neuron and Eqs. 1.8–1.11 describing the Backprop algorithm for determining the synaptic weights, there is enough background to consider a methodology for constructing neural network applications. Figure 1.4 shows an example of a neural network in operation. The neural network must be designed before it can be used in an application. The number of neurons in the input layer, hidden layer(s), and output layer have to be determined. The input and output layer are usually easy to find since the dimensions of the input and output vectors interfacing to system one and system two are known. However, the hidden layers are not as simple to deal with. In general, there are two ways to build the hidden layers:

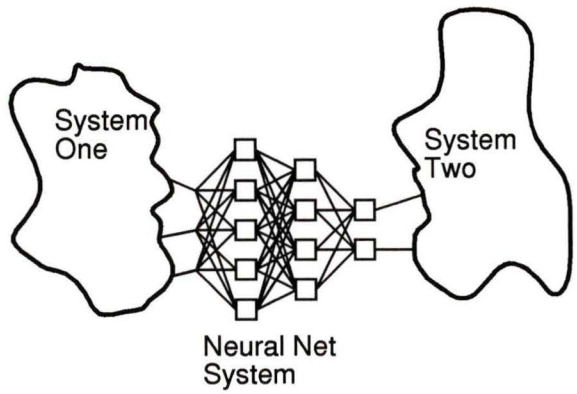


Figure 1.4: An Example Application for a Neural Network. In this example, the network classifies data presented from system one and presents the classes to system two.

(1) design by trial and error, and (2) use an algorithm to automatically build a neural network.

The trial and error design approach manually varies the number of neurons in the hidden layer as well as the number of hidden layers until a satisfactory result is found. Generally this is done within a neural network simulator to speed up the process of evaluating networks. The main advantage of this technique is the flexibility offered to the neural network designer; the designer is free to use any network topology that makes sense given the constraints at hand.

There are many algorithm techniques in the literature, two of them are: pruning methods, and Cascade Correlation algorithm. Pruning methods iteratively remove neurons from the hidden layer(s) that are not needed (see Reed[48] for a survey of pruning algorithms), while Cascade Correlation iteratively adds neurons into the hidden layer(s) to increase network performance (see Fahlman[21]). The algorithm techniques work very well for novice neural network designers; the designer does not require intimate knowledge of the number of layers, or the number of neurons per layer in the network. The largest drawback to algorithm techniques is the loss of

flexibility when designing neural networks. The networks are constructed outside human hands and therefore may not meet the designer's particular constraints.

As will be seen, neural networks intended for the CORDIC hardware have constraints on the number of neurons per layer. Typically, the CORDIC hardware offers powers of two (8, 16, 32, etc.) neurons per layer. The neural simulator used by the author does not have the ability to constrain the number of neurons per layer. Therefore the author builds up networks by trial and error until a suitable network is found.

The Stuttgart Neural Network Simulator (SNNS)[65] is used to draw, train, and evaluate neural networks. After a network is trained, the synaptic weight matrices and neuron bias vectors are available to be programmed into a digital neural network for physical implementation. The SNNS tool is freely available through ftp (see the bibliography reference for the ftp address).

1.4 Summary

This chapter introduced the biological neuron with emphasis on black–box functionality. Based on the firing rate of a biological neuron, the author examined the artificial neuron as shown in Fig. 1.2 and described by Eqs. 1.1 and 1.2.

The multi–layer perceptron was also discussed with enough detail for the reader to construct his/her own artificial neural networks if desired. In addition, neural network simulators are considered as a tool to draw, train, and evaluate neural networks.

The remainder of this thesis focuses on the the electronic hardware required to execute the neural networks designed with the neural simulator. A short chapter summary of the main chapters follows:

Ch. 2 This chapter examines several examples of electronic hardware to implement neural networks. Both analogue and digital implementations are considered. It is noted that there are relatively few dedicated digital implementations of neural networks.

Ch. 3 This chapter evaluates several potential designs for dedicated neural networks. Of particular interest is the concept of modifying the VLSI complexity metric, AT^2, to account for hardware algorithms that do not give exact answers. The primary issue is how to compare functions that converge to the answer over time with those that give their result after a delay.

Ch. 4 With the a priori evaluation indicating that the CORDIC method is suitable for neural networks, this chapter examines the details of a CORDIC implemented neuron: the CORDIC step, convergence requirements, and constructing neurons from CORDIC operations. A novel part of this chapter is the concept of a CORDIC schedule. Typically the sequence of CORDIC steps is static over time, however it need not be so. It is quite possible, in fact, to construct a dynamic CORDIC schedule that automatically guarantees convergence.

Chapter 2

Case Studies of Hardware Neural Networks

Soon after the mathematical development of neural networks, analogue implementations began appearing in the literature. The initial work was done with analogue hardware because the digital computer was still in its infancy.

Two examples of these early works are the Adaptive Linear Neuron (later renamed the Adaptive Linear Element; shortened to ADALINE) of Widrow and Hoff[63], and Rosenblatt's perceptron[50] implemented by Block[8].

In 1969, Marvin Minsky and Seymour Pappert published a book devoted to the study of perceptrons[44]. Their strong, analytic style held much influence amongst the connectionists.

Minsky and Pappert's critical account of single layer perceptrons nearly halted work in the field. The hardest blow to neural networks came in the form of a challenge [44, pp. 231–232]:

> The perceptron has shown itself worthy of study despite (and even be-

> cause of!) its severe limitations. It has many features to attract attention: its linearity; its intriguing learning theorem; its clear paradigmatic simplicity as a kind of parallel computation. There is no reason to suppose that any of these virtues carry over to the many-layered version. Nevertheless, we consider it to be an important research problem to elucidate (or reject) our intuitive judgement that the extension is sterile. Perhaps some powerful convergence theorem will be discovered, or some profound reason for the failure to produce an interesting "learning theorem" for the multi–layered machine will be found.

At that time no practical algorithms existed to train multi–layer networks. The challenge went unanswered for close to 20 years.

The effect on the neural network community was quick and devastating. Very little work continued under the neural network banner. In fact, major funding groups grew so disaffected with neural networks that researchers like Bernard Widrow changed the name of their work to disassociate themselves from the neural network community (the ADALINE started life as the Adaptive Linear Neuron, and then changed to the Adaptive Linear Element). Therefore, few if any networks were implemented after 1969 (until the backprop revolution).

In the mid 80s, Rumelhart, Hinton and Williams[51] answered Minsky and Pappert's challenge. They developed the back propagation of error algorithm described in chapter 1. This revitalized the neural network community. For the first time, practical multi–layer networks could be trained. This allowed designers to construct real applications for their neural networks.

A major distinction between the backprop revolution and the initial work done in the 60s, is the proliferation of digital computers. The advent of the digital computer enabled researchers to develop and test algorithms painlessly – new algorithms could

Section Name	No. Articles	% of Total
Applications	77	12
Robotics and Control	64	10
Pattern Recognition	57	8
Associative Memory	48	7
Supervised Learning	48	7
Neurodynamics	48	7
Signal Processing	43	6
Unsupervised Learning	42	6
Machine Vision	41	6
Intelligent Neural Systems	38	6
Electro–Optical Implementations	28	4
Speech and Language	26	4
Biological Vision	24	4
Cognitive Neuroscience	22	3
Neural Fuzzy Systems	19	3
Biological Sensory–Motor Control	16	2
Manufacturing Track	13	2
Local Circuit Neurobiology	9	1
IFSA/INNS Track	5	1
Total	668	99

Table 2.1: The number of articles published in the 1993 World Congress on Neural Networks organized by subject area.

be tested in the matter of hours. The effect is still felt today – software simulation is the most popular environment to develop neural networks.

To illustrate this point, Table 2.1 shows the number of articles published under various topics for the 1993 World Congress on Neural Networks[34]. Note that there are only 28 articles (4% of total) published regarding physical implementation (electrical or optical) of neural networks while the supervised and unsupervised learning alone contribute 90 articles (13% of total).

However, there is evidence of a bias toward software implementations of neural networks. There are reasons to use hardware implementations: (1) hardware implementations are faster, (2) general purpose computers are not an efficient engine to

run neural network algorithms, (3) software algorithms are not practical for small system (embedded) applications. Software implementations are generally used whenever special purpose neuro–computers are not available. Even hardware implementations can benefit from software tools; for example, the SNNS tool has proved invaluable for designing neural networks to download to the neural hardware.

Given the author's natural bias towards hardware and the gap in hardware implementations, the remainder of this chapter focuses on hardware implementations of neural networks.

2.1 Analogue Designs

While analogue implementations have been around for over 30 years, they have not grown as quickly as digital implementations. Specifically, the growth of digital VLSI techniques enables designers to build fast and cheap circuits with high density. However, some designers are still attracted to the speed and controllability of analogue designs.

As with all analogue systems, analogue neural networks suffer problems related to power fluctuations, crosstalk, and noise. Furthermore, although efforts have been made to build analogue CAD tools[66, 11, 37, 15, 17] they are not yet powerful enough to compare with digital design automation tools. In general, analogue VLSI designers must still design the circuits by hand while digital designers have the luxury of choosing from a large selection of CAD tools (e.g. automatic generation from hardware description languages, VHDL and Verilog are the two main alternatives, schematic capture, hardware simulation, fault analysis, etc.).

Long term weight storage is another problem facing analogue neural network designers[45, p. 1541]. Current technology offers limited solutions for storing weight

values over long periods:

- Capacitors store voltages, but charge leakage limits this storage method to short term storage only. To offset this, the weight storage nodes of Furman and Abidi[24] use cryogenic temperatures to limit the charge leakage problem.

- Floating gate transistors (see Fig. 2.1) act like programmable resistors; the preset floating gate bias sets the resistance within the transistor[33, 62, 14, 54, 9].

  Floating gate transistors are slow, have limited lifespan due to wear out properties[14], are difficult to manufacture with uniform threshold voltages over large spans of area, and have variations between transistors with regard to programming voltage and time required to program.

- Charge Coupled Devices offer another option for weight storage[2, 14, 52]. They perform well as long term storage, however charge transfer efficiency limits refreshing[14]. Refreshing is critical for feed forward neural networks.

- Digital to analogue converters[14] provide another approach to the weight storage problem. This method suffers problems related to weight accuracy limited by digital circuitry, hardware complexity due to extra converters, the negative impact on speed because of extra circuitry required for digital to analogue conversion. Variations of the method include analogue refreshing circuitry that samples the voltage on a capacitor and restores it by passing the voltage through an A/D converter and back through a D/A converter[39].

Analogue neural networks have several advantages over digital networks: (1) they tend to be quicker (i.e. instantaneous output vs. sequential output), (2) they enjoy a simpler design with respect to the digital control circuitry (e.g. clocks and enable lines) that is (possibly) no longer required, (3) they require one line for signal data

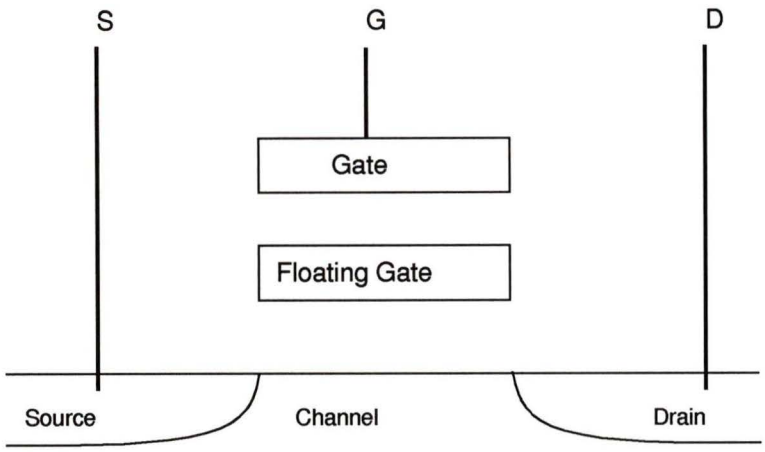


Figure 2.1: Idealized floating gate transistor. Charges are induced into the floating gate by biasing the source, drain, and gate pins to induce electron tunneling into (or out of) the floating gate.

instead of complete data buses[1], (4) analogue designs are truly asynchronous not sequential — this offers different computation models than clocked designs, and (5) the net function and activation function are much simpler to implement in analogue (the net function can be the summation of currents on a node, and the activation function can be implemented as the nonlinearity of a simple operational amplifier).

2.1.1 Examples of Analogue Neural Networks

Typical analogue designs use the current summing on a node principle with some form of non–linear saturation to act as an activation function. There are numerous examples of this technique in the literature[12, 19]; only one will be considered here[2].

[1]The author recognizes that data can be clocked through a single bit serially. This, however, involves an obvious speed penalty.

[2]It is really quite unfair of me to give the impression that analogue neural networks are devoid of character and variety. In fact, quite the opposite is true; there is a wide selection of analogue neural network designs including switched capacitor techniques[20], switched resistor techniques[49], Charge Coupled Device techniques[2].

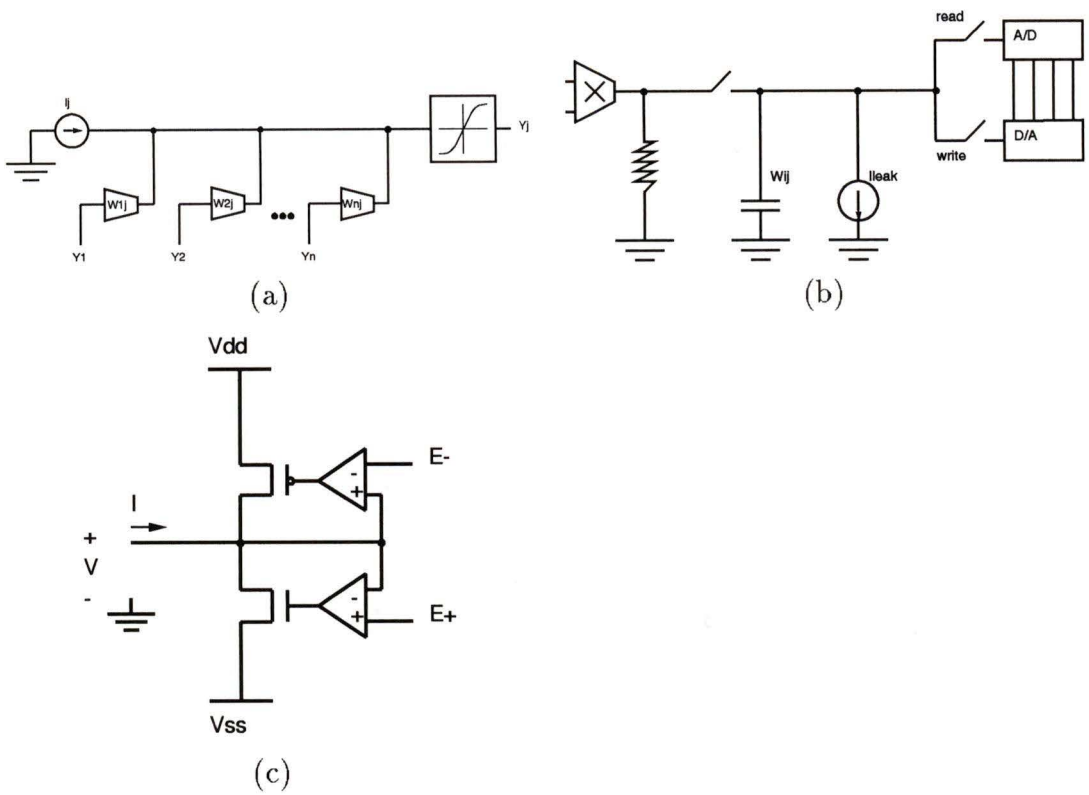


Figure 2.2: Linares–Barranco's analogue neuron[39]: (a) the neuron built up from transconductance amplifiers (and a non–linear activation circuit; implemented as a non–linear resistive shunt), (b) Analogue Weight Refreshing Circuit, (c) Analogue Activation Circuit (non–linear resistor).

A typical analogue design is shown in figure 2.2. This represents the analogue neuron, weight refreshing circuit, and the non–linear saturation circuitry for Linares–Barranco's analogue VLSI neural network[39].

The refresh circuitry acts very much like the refresh circuitry for DRAMs, but uses an A/D and D/A circuit to set the reference signal in the capacitor. The A/D and D/A reference signal generator is not replicated for each weight, but rather the pair is shared by all the synapses in the chip. The refresh cycle moves sequentially through the weights. The net effect of this mechanism is that the weights are quantized to the word length in the A/D and D/A converters.

2.2 Digital Designs

Digital designs offer many advantages over their analogue counterparts: (1) it is much simpler to change a digital design than an analogue design, due both to the mature CAD environment and the simpler design rules, (2) the design is easier to expand in both size (i.e. number of devices) and accuracy (i.e. number of fractional bits), (3) it is much easier to construct a working digital circuit than a working analogue circuit[3], (4) digital designs do not suffer the same problems with regard to crosstalk, and noise, (5) the performance of the network is reasonably easy to predict in terms of output error and clock speed (this makes feasibility studies much easier). The downside of digital designs is that they are often synchronous (which means they can be slow), have problems with clock skew, and can use lots of chip area for weight storage.

Most commercial digital implementations are specially built parallel computers where the algorithms are mapped to them. The advantage of this approach is that you can change the algorithm at will. In fact, some proponents of such systems argue that the neural network algorithms are not stable enough to etch in silicon[1] and that a reprogrammable device provides a very nice alternative.

The remainder of this section is devoted to digital implementations.

2.2.1 Adaptive Solutions, Inc.'s CNAPS neuro–computer

Adaptive Solutions, Inc.[1] provides an entire range of products for developing neural network applications, both hardware (many sizes of neuro–computers) and software (compilers, simulators, libraries, etc.).

[3]Even large scale digital circuits work first time out of the foundry more times than not. The black art of digital circuit design is not to make the circuit work, but to make it work *fast*.

The CNAPS neuro–computer top level architecture may be found in Fig. 2.3(a). As can be seen from the figure, the CNAPS system is constructed as a linear array of one to eight CNAPS–1064 processor chips. Each CNAPS–1064 chip contains 64 processing nodes arranged as shown in Fig. 2.3(b).

The linear system structure of Figs. 2.3(a)-(b) allows for a configurable multi–processor computer system with between 64 and 512 processing nodes.

System level I/O flows through three busses: (1) the control processor communicates through a system bus to the CP Bus Interface (see Fig. 2.3(a)), (2) the CNAPS Sequencer Chip has an external data out bus, and (3) each CNAPS–1064 chip has an external I/O bus for off board communication. The system bus is used mainly for system level control between the host computer and the CNAPS neuro–computer, however data can also flow through this bus. The external data out bus is the main data bus for results, while the direct I/O bus allows for much faster processing node local communication.

The file memory module is a block of dynamic RAM used to store data locally in the CNAPS architecture. This is considerably faster than feeding data through the system bus. The data is available to the processing nodes and to the controlling processor through the control processor bus interface. This memory can also hold input or output data files for fast batch mode operation.

The program memory is a block of static RAM holding the 64–bit instructions — 32 bits required by the CNAPS Sequencing Chip and 32–bits broadcast to all processing nodes. Therefore, the CNAPS neuro–computer is a SIMD multi–processor computer system. The SIMD model has proven to be efficient for many neural network algorithms because the neurons operate the same instructions[1, 42] and also typically broadcast their outputs to many other neurons.

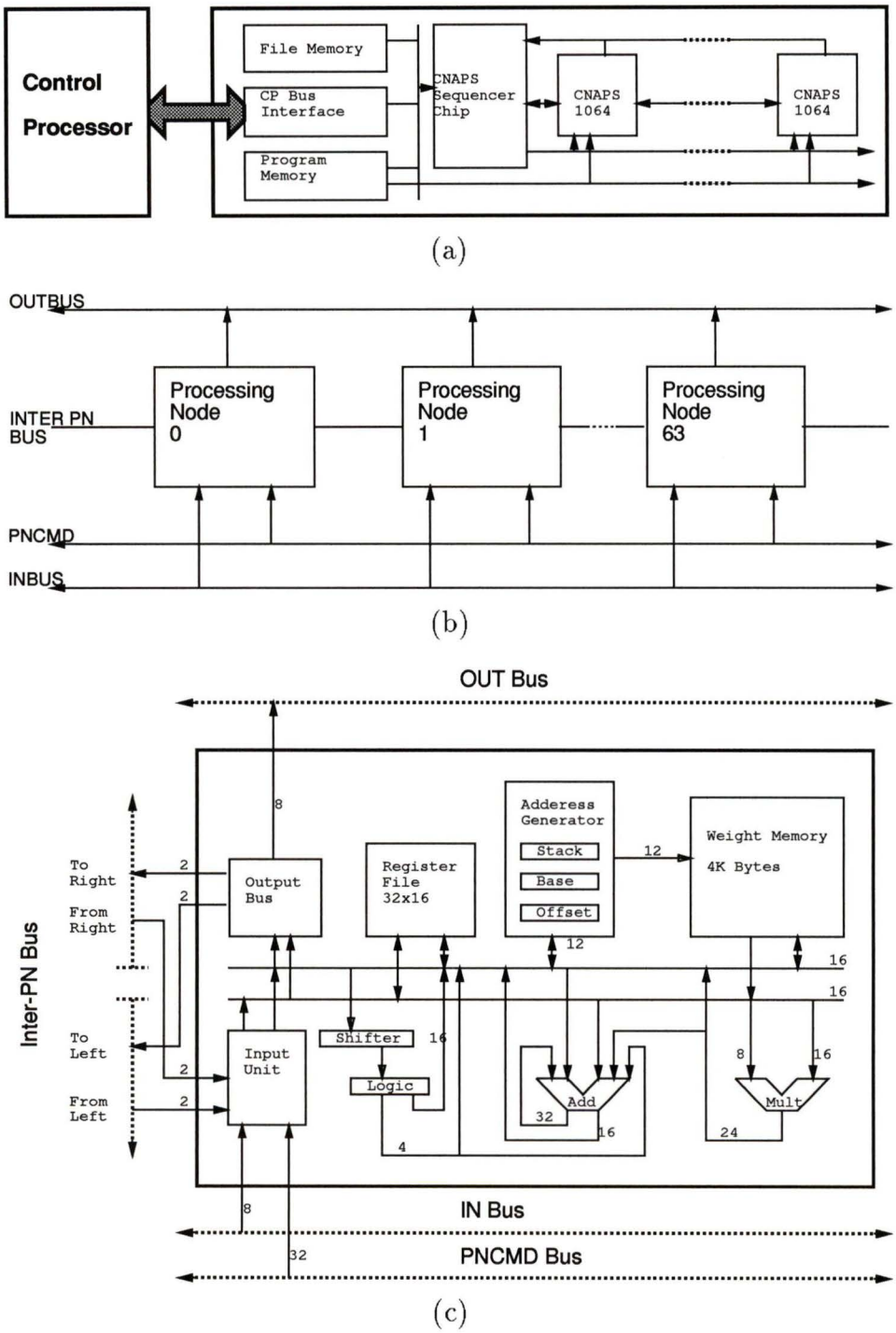


Figure 2.3: The CNAPS neuro–computer: (a) System level, (b) CNAPS–1064 chip level, (c) Processing Node level.

The Processing Node Architecture

The block diagram for the processing node can be found in Fig.2.3(c). It is evident from the diagram that there are three global busses: the output bus (OUTBUS), the input bus (INBUS), the control bus (PNCMD), and two local busses for transferring node operations (ALU, memory, etc.).

The output bus is looped back into the input bus through the CNAPS Sequencer Chip. This allows processing nodes to broadcast data to all the other processing nodes. All the processing nodes can read the data in the input bus simultaneously, but only one processing node can output at a time.

In addition to the local bus optimizing internal processing node communication, there are many paths between the ALU units to reduce the time required to perform multiple ALU operations. For example, the multiplier unit feeds into the adder unit for efficient multiply/accumulate operations (used for the net function, recall Eq. 1.1).

The input unit decodes the instructions placed on the control bus. The output unit contains bus arbitration circuitry to ensure data integrity on the output bus.

The adder unit takes 32–bit inputs and produces a two's complement 32–bit result. On overflow the results saturate to the largest positive or negative number depending on the sign of the result.

The memory unit is a four kilobyte memory accessed using either 8–bit or 16–bit modes. A special hardware feature called 'virtual zeros' allows groups of contiguous zeros to be collected into one location. This addresses the characteristics that neural networks often exhibit sparse connectivity matrices[1].

From the processing node description, it is apparent that this architecture is not just for neural network applications. In fact, any computer algorithm using SIMD control and broadcast output maps well to this architecture. The flexibility offered

Characteristic	CNAPS/64	CNAPS/128	CNAPS/256	CNAPS/512
Number of PNs	64	128	256	512
Multiply/Accumulates	0.96 Billion	1.92 Billion	3.84 Billion	7.68 Billion
Feed forward (CPS)	0.87 Billion	1.70 Billion	3.16 Billion	5.70 Billion
Learning (CPS)	220 Million	429 Million	813 Million	1.46 Billion

Table 2.2: Published Results for the CNAPS–1064 Neuro–Computer.

by the general purpose neuro–computer design allows this architecture to simulate a wide variety of neural network algorithms currently used (e.g. feed forward, Hopfield, Adaptive Resonance Theory).

Table 2.2 shows the published results for the the CNAPS server line of neuro–computers. These are stand alone cabinet units connected to a host computer through Ethernet. The control processor is a Motorolla 68030 based microprocessor.

2.2.2 The SYNAPSE-1 Neuro–computer

The SYNAPSE-1 (derived from Synthesis of Neural Algorithms on a Parallel Systolic Engine) neuro–computer of Ramacher[47] is another parallel neuro–computer at the system level (four boards connected via a VMEbus). The top level system level architecture is shown in Fig. 2.4[47, p. IV–776].

The SYNAPSE–1 neuro–computer is a combined hardware and software project. Since my concern is with hardware only, the implications of the software (C++ compiler, host workstation communications coding, etc.) are ignored.

The hardware portion is a two dimensional array (two rows by four columns) of MA16 neural signal processors. Each MA16 processor contains local memory and a linear systolic array of four processing modules.

Each processing module is broken into the input section and the output section. The input section calculates functions listed in Table 2.3, while the output section

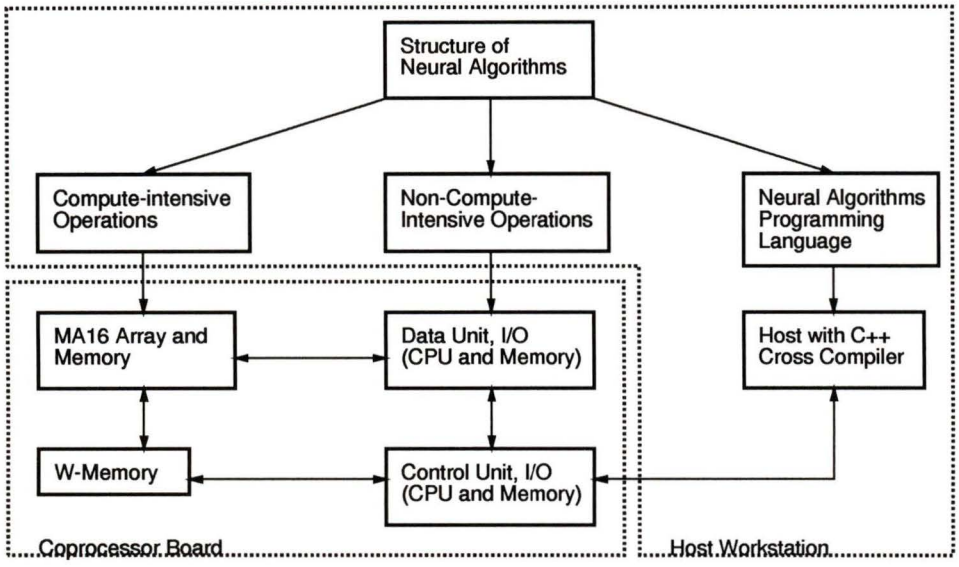


Figure 2.4: System Level View of the SYNAPSE–1 Neuro–Computer.

performs overflow detection, min/max searches, block floating–point calculations, and specialized operations for neural networks.

The controller unit issues control signals to the input and output sections to direct the flow of the calculations according to a stored program written in C++. The control unit comes in two pieces: the communications are run by a MC68040 CPU while the MA16 array and its dependents (Memory, FIFOs, backplane transceivers,

s_{ik}	$=$	$\alpha_i \sum_j x_{ij} y_{jk}$
s_{ij}	$=$	$\pm x_{ij} \pm y_{ij}$
s_i	$=$	$\sum_j (x_{ij} - y_{ij})^2$
s_i	$=$	$\sum_j \lvert x_{ij} - y_{ij} \rvert$
s_i	$=$	$\sum_j \lvert x_{ij} \rvert$
s_i	$=$	$\sum_j x_{ij}^2$
s_i	$=$	$\sum_j \lvert y_{ij} \rvert$
s_i	$=$	$\sum_j y_{ij}^2$

Table 2.3: List of Functions Available in the Input Section of the SYNAPSE–1 Neuro–Computer.

Theoretical Clock Speed	40 MHz
Peak Performance	1×10^9 connections per second
Number of Boards	4

Table 2.4: Published Results for the SYNAPSE–1 Neuro–Computer.

etc.) are controlled by a microprogram sequencer.

The published results are shown in Table 2.4.

2.2.3 The MIND-1024 Neuro–computer

The MIND–1024 neuro–computer is a digital hardware implementation of a fully connected 1024 neuron Hopfield network[25][4] In addition, fully connected networks allow for building arbitrary networks from the 1024 available neurons.

There are three units to the MIND–1024 neuro–computer: (1) a 1024 fully connected binary neural network, (2) a 64 + 1 MIMD processor array to train the network, and (3) a host processor for user interface and control.

(1) The neurons of the binary neural network describe states:

$$S_i = \text{sign}(f(\sum_{j=1}^{N} W_{ij} S_j - \theta_i + \eta_i)) \tag{2.1}$$

This is similar to Eqs. 1.1 and 1.2 with a few modifications: $f(\cdot) \leftarrow \psi(\cdot)$, $S_j \leftarrow x_j$, the sign of θ_i is reversed, and η_i is inserted into the expression[5]. The activation function, $f(\cdot)$, is implemented as a ROM lookup table. The η_i parameter is a random

[4]Hopfield networks[31, 32] implement fully connected networks (each neuron broadcasts its output to each other neuron) with a hard limiter activation function. They are often used for optimization (traveling salesman problem) and auto–associative memories. Auto–associative memories are associative memories where the retrieval keys are the storage patterns themselves; they retrieve the closest stored pattern to the given input pattern.

[5]The $\leftarrow$ operator is a substitution operator. The left side is substituted by the right side of the expression.

variable whose distribution has width T. According to the authors, the random variable enables the MIND–1024 to handle noisy neurons. A pseudo–random generator provides addresses to where the user has stored random values according to the desired distribution. The noise generator is fully programmable and can deliver more than a million random numbers without any observable correlation each second.

(2) The learning unit comprises 64 Intel 80C186 microprocessors running at 12.5 MHz. Each microprocessor implements 16 neurons and their corresponding 1024 outputs. One additional microprocessor maintains the neuron thresholds. The processor array is MIMD in design, but functions in a SIMD mode since each processor runs the same program on local memory.

(3) The supervising unit synchronizes the other microprocessor's activities and controls the machine interface to the outside world. The supervising unit controls machine resources by sending commands via the VME bus. The supervising unit is controlled via a standard library available for C, C++, or Pascal programming languages. The supervisor is an Intel 80386 based PC running at 20 MHz with a VME controller.

The published results are found in Table 2.5.

2.2.4 User Configurable Boolean Neural Network Chip

Boolean neural networks operate like normal neurons (Eqs. 1.1 and 1.2), but with some modifications. Boolean neural networks get their name because the inputs and outputs are Boolean (0 or 1 only). Furthermore, the weights are restricted to -1, 0, or +1. The activation is simply a hard limiter with a threshold.

The user configurable Boolean neural network chip (BNNC) of Bhide[7] implements a Boolean neural network with up to 64 inputs.

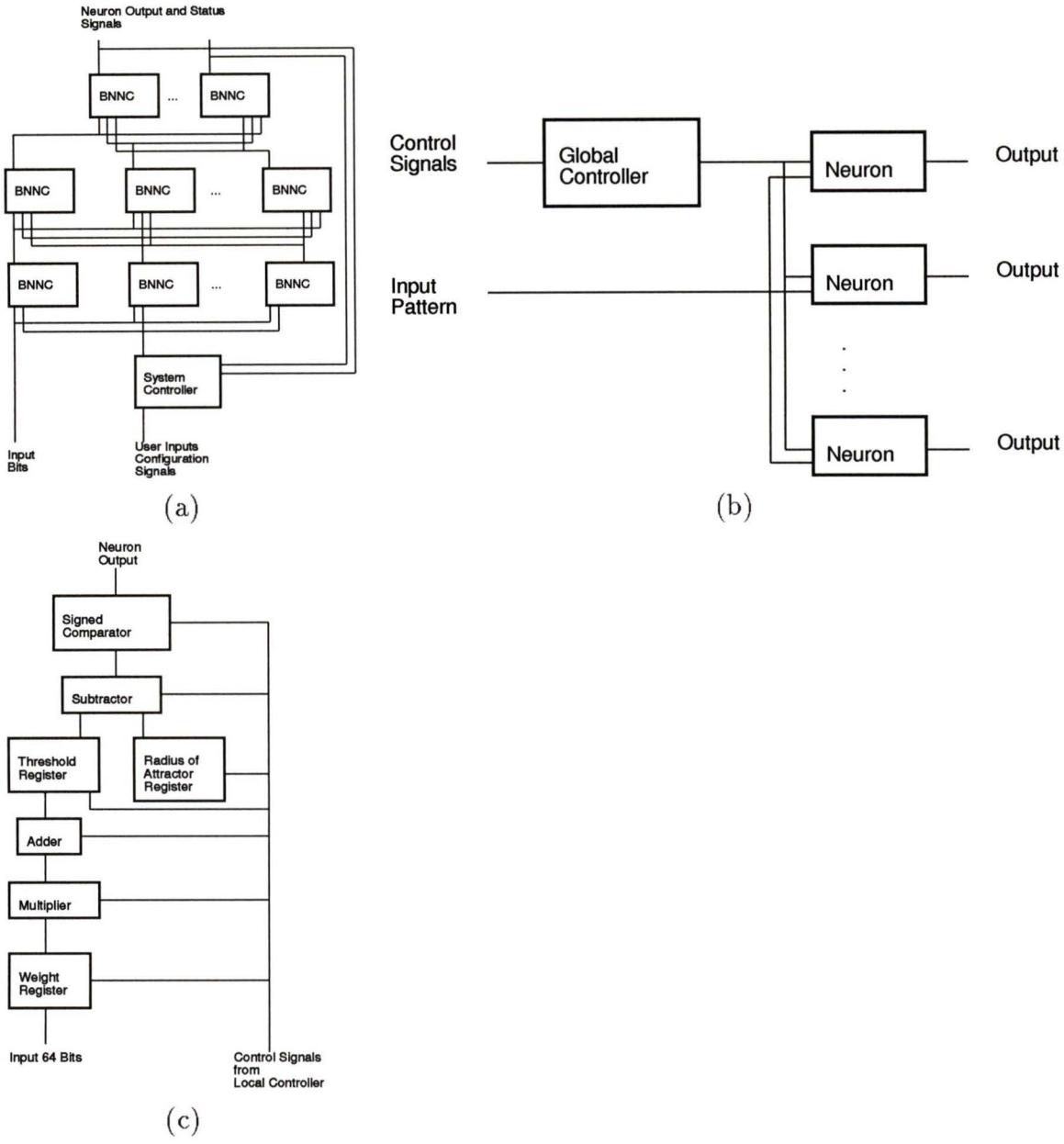


Figure 2.5: The User Configurable Boolean Neural Network design of S. Bhide[7]: (a) is the system level view of a Boolean Neural Network designed using BNN chips, (b) shows the BNN chip itself, and (c) shows the BNN neuron.

Number of binary neurons	1024
Number of synapses	1,048,576 (full connectivity)
Precision of the synapses	16 bits (learning) / 8 bits (recalling)
Precision of the thresholds	32 bits (learning) / 18 bits (recalling)
Precision of noise variable	18 bits
Type of noise distribution	Any (Gaussian, thermal and white built–in)
Cycle length (updating time)	910 ns.
Neurons updating scheme	Asynchronous, synchronous or mixed
Number of learning processors	64 for the synapses + 1 for the thresholds
Learning processor type	Intel 80C186 at 12.5 MHz
Learning rule	Any one programmed in C, C++, or Pascal
Recalling speed	1.1×10^9 Synapses/second
Learning speed	20×10^6 Synapses update/sec (Hebb in C)
Input/Output of neuron states	Available in real–time on a dedicated bus

Table 2.5: Published Results for the MIND-1024.

This design demonstrates how a neural network may be decomposed into a multi–chip design, with each chip handling a subset of the neurons required for the entire network. In this case the chips are cascaded both layer to layer as well as within the same layer. This is significantly different from the approach taken by Adaptive Solutions, Inc. where the addition of CNAPS–1024 chips only provides more power to the neuro–computer. Here, the addition of BNN chips form the structure and function of the neural network itself.

Figure 2.5(a) shows a system level architecture using BNN chips. The system controller issues control signals to the network to configure the individual BNN chips, thus building the desired network topology. The input bits feed directly into the input layer of BNN chips, which in turn pass the results to the next layer, until the last layer. The outputs of the last layer feed into the system controller so that the proper network weights may be calculated for learning. The system is implemented at the board level.

The BNN chips , Fig. 2.5(b), contain the global controller and neuron hardware.

Clock Speed	10 MHz
Classification Rate	143,000 patterns per second
No. of Inputs	64
Input Precision	1 bit
Synaptic Weight Precision	2 bits
Threshold Precision	8 bits

Table 2.6: Published Results for the User Configurable Boolean Neural Network Chip.

The global controller gets configuration and training signals from the system controller and passes on instructions to the individual neurons. As you can see from the figure, the BNN chip operates in the SIMD mode.

The neuron may be found in Fig. 2.5(c). The weights are represented as a 128–bit register (two bits per input). The weights are passed to the multiplier where each of the 64 Boolean inputs is multiplied by a ternary weight simultaneously using a combinational circuit. The adder sums the negative and positive contributions of the 64 multiplications. The threshold register and ROA register are used for testing and specialized networks. The signed comparator emits a 1 if the result is positive, and 0 if negative.

The published results are given in Table 2.6.

2.3 Hybrid Designs

Hybrid designs offer a compromise between the analogue and digital implementations. Typically hybrid designs use analogue circuits to perform the net function and activation function (Eqs. 1.1 and 1.2), while digital circuitry gives long term weight storage, off–chip communication, and network training if desired.

Figure 2.6 shows the design for a single layer perceptron presented by Nosratinia[45]. The circuit implements a full layer of neurons by serially operating N analogue mul-

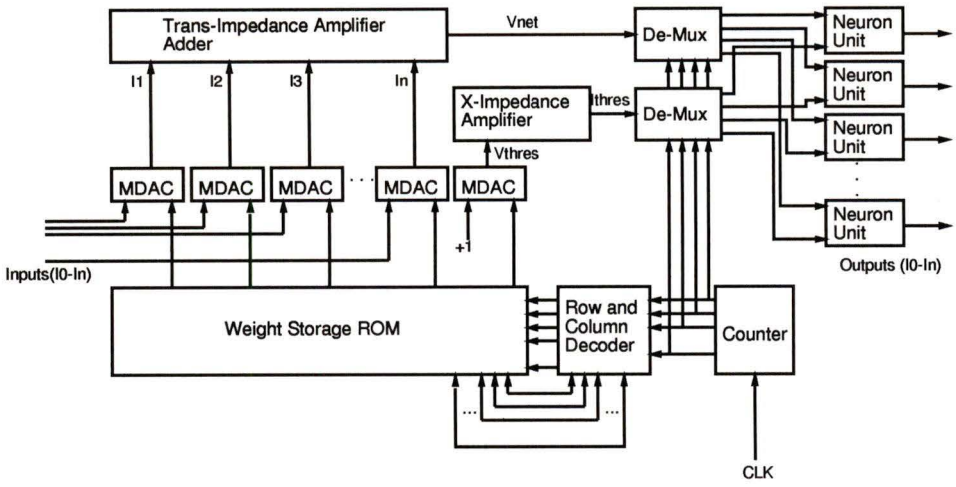


Figure 2.6: Block Diagram for Nosratinia's Single Layer Perceptron. This is a hybrid design (i.e. both digital and analogue) where the weight storage is in digital and the net and activation computations are performed in analogue.

tipliers and a single analogue adder. The results are output to independent output channels.

Input capacitors in the neuron blocks (not shown) stabilize the charge during the time between input patterns. Thus, the circuit must be refreshed regularly in order to keep the outputs valid. Nosratinia reports that the minimum clock speed to keep the capacitors charged is dependent on capacitive leakage and temperature, and is no larger than a few hundred kHz[45, p. 1543].

The multiplier in the figure is a multiplying digital to analogue converter (MDAC) where the weight is fed from the ROM as a digital signal and the input is an analogue signal fed from the previous layer. The result of the MDAC module is a current which is summed on a node and fed into a trans–impedance amplifier to convert back into a voltage. The resulting voltage is the hardware equivalent of the net value of Eq. 1.1. The net value is fed into a de–multiplexor to select which neuron unit receives the net value.

Clock Speed	6 MHz
Number of Layers	3
Number of Neurons	20, 15, 10
Multiplications per Second	426×10^6

Table 2.7: Published Results of a Test Run for Nosratinia's[45] Hybrid Neural Network.

The rightmost column of the weight storage ROM contains the neuron thresholds. This column is also multiplied via a MDAC (by unity) and passes through a trans–impedance amplifier and de–multiplexor before reaching the neuron unit. The extra circuitry (MDAC and trans–impedance amplifier) ensures that both the threshold and the net value arrive at the neuron unit at the same instant.

The neuron unit acts like the activation function. The input capacitors on the neuron unit keep the output valid until the next clock cycle.

A counter selects rows in the weight ROM to feed into the multiplying circuitry, as well as the destination neuron unit, for the resulting net value and threshold.

The published results are found in Table 2.7.

2.4 Summary

This chapter began with the historical development of artificial neural network hardware from Widrow's ADALINE to present. The interesting point of issue is how the development of neural networks is linked with current technology. The original implementations were fabricated using analogue computers while today neural network implementations are churned out by the dozens on highly sophisticated UNIX based workstations. It is apparent from the literature that work on neural networks has moved from the hardware regime to the software domain.

Even with this shift, there is still work being done on hardware implementations. As illustrated by Block[8, p. 142], there is a long standing rivalry between digital and analogue design approaches:

> *Construction of an actual machine.* This has an enormous advantage in speed over the digital computer, since essentially all the action goes on in parallel simultaneously and the response appears almost immediately, while in the digital simulation all computations are done in sequence. While an actual machine enjoys certain types of flexibility, such as the ease with which the experimenter can vary the stimulus patterns, it is a serious task to change the wiring diagram (in the digital computer this can be generated quickly by a suitable program) and it is impossible to alter certain basic features of the network.

The majority of digital implementations are highly complex, blazingly fast, and terribly expensive neuro–computers. These systems fill a similar role as the Intel Hypercube did a few years ago. They provide a specialized hardware environment to experiment with massively parallel (in this case neural) algorithms.

The author feels that neural algorithms can also be used for many small to medium sized applications that do not require the speed/cost/size of the neuro-computer approach, for example: rudimentary speech recognition, character recognition, and image processing can all be performed using neural networks with fewer than 64 neurons per layer[38].

The embedded ANN application is also particularly interesting. The term 'embedded' means both embedded chip design (small scale neuro–processor on board level designs) as well as embedded system design (having an ANN application inserted as a subtask of a larger system).

While there are examples of small scale dedicated neural networks in the literature (the Boolean Neural Network is particularly interesting), none can operate as multi–layer perceptrons (the BNN is limited to binary inputs and ternary weights).

The target system should work as a backprop neural network, but does not have the complexity the neuro–computer designs have (i.e. multiple boards, host machine, etc.). Dean Collins[14] describes a set of criteria for acceptable neural network chips:

- $\approx$100 neurons per chip
- Programmable interconnections
- At least 4 bits of resolution
- Stable
- Reproducible
- Convenient I/O
- Extensible
- Able to do many of the current neural network algorithms

The author objects to only one criterion: the last one. Not all neural network systems (particularly embedded systems) require the power of executing most (if not all) of the current neural network algorithms. With the diversity of the algorithms to choose from, it is hopeless to construct a pure hardware implementation to achieve this.

To many, however, the last criterion is very important. In fact, it has been argued that there is little point to construct dedicated hardware designs until the neural network algorithms become stable. This limits the hardware neural network designs to only the neuro–computer model. There is, however, a counter point to be made: given the proliferation of Field Programmable Gate Array (FPGA) technology[64] it is quite feasible to design neural network algorithms in hardware and change the design when necessary to implement a new neural algorithm.

Chapter 3

A Priori Evaluation of Implementations

There are many techniques available to design digital neural components. While most of these techniques are from the field of Digital Signal Processing (DSP), the application of these techniques to neural networks is a fairly straightforward task. However, the author would like to mention that Fahmi et al's SPIPP[22] technique is relatively new, and the introduction of the CORDIC method to neural networks is not currently represented in the neural network literature.

This chapter is devoted to the quantitative analysis of some of these designs, specifically: (1) Baugh Wooley multiplier, Serial–Parallel Inner Product Processor, and the CORDIC method for network functions; and (2) hard limiter, ROM, CORDIC, seventh order Chebyshev series, and three segment linear approximation techniques for the activation function. Only the arithmetic operations of these designs are important to this chapter; other considerations (e.g. I/O and bus throughput) are tangential to the purposes of this chapter.

Before the different approaches can be compared, a cost function to quantitatively measure the relative merits of the designs must be developed. Designs are chosen that minimize the cost, $\mathcal{C}(n, N)$ (where n represents the number of bits in the result and N represents the number of inputs to the neuron). It is difficult to evaluate designs based on the algebraic expression for the cost function, therefore the results are graphed to determine the merits of the designs.

3.1 Merit Function for Arithmetic Algorithms

Current VLSI literature often compares implementations using the VLSI complexity bound[58, 53], Eq. 3.1, to determine how closely the design comes to the theoretical limit of VLSI complexity for the problem at hand. The literature exhibits lower bounds for common problems such as integer multiplication, matrix multiplication, Fourier transforms[58, p. 79].

$$AT^2 < \mathcal{B} \tag{3.1}$$

where A is the circuit area, T is the propagation delay, and $\mathcal{B}$ is the complexity bound for the problem at hand.

From the circuit designer point of view, it is desirable to know the complexity bound for the design at hand prior to implementation. Unfortunately, the complexity bound shifts by many orders of magnitude depending on the technology used (minimum wire size, number of poly–silicon and metal layers), the skill of the layout designer (or layout CAD program efficiency)[41]. Furthermore, the difficulty in calculating accurate measures for the area and time parameters prior to circuit design adds yet more uncertainty to the complexity bound.

In general, it is very difficult to construct an a priori bound to determine how close the design is to the theoretical optimum. However, one may consider the complexity

bound as a merit function to compare designs as long as designs are compared on an even footing. It would be inappropriate to compare a Xilinx implementation to its CMOS4S Standard Cell VLSI counterpart.

There is an additional issue, independent of the problems stated above; namely, "How does one use the VLSI complexity metric to account for arithmetic designs containing errors in the output?" While some errors are inherent in the technology (bit error), there is also the potential for an *acceptable* error introduced because the hardware approximates the target function. In order to compare approximation designs, the merit function must be extended to include error.

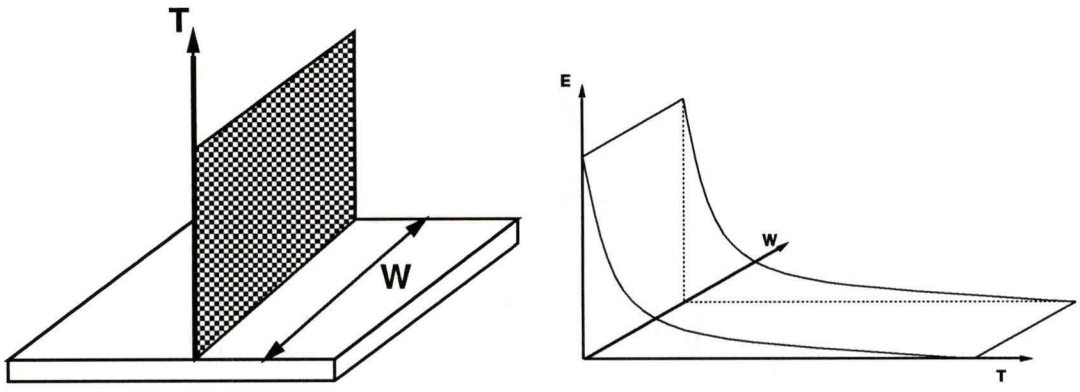
3.1.1 Extended Merit Function

The derivation of Eq. 3.1 is based on a simple model of circuit communication. Figure 3.1(a) shows the surface of a square chip bisected into two regions that are said to be communicating to each other along the cut.

For square chips, the minimum cut length, W, is $\sqrt{A}$. Since the regions communicate along the entire length of the cut, $\sqrt{A}$ will be named the communication factor for an instant in time.

Integrating the communication factor from $t = 0$ until the computation is complete yields $\sqrt{A}T$ — the area of the plane in Fig. 3.1(a). Since this figure of merit is somewhat clumsy to write, the relation is squared to give AT^2 as our final figure of merit.

As mentioned previously, this figure of merit neglects the effect of errors over the channel. This shortcoming prevents an accurate analysis of algorithms that approximate the final result. Since the hyperbolic tangent function is difficult to implement in digital hardware, approximation functions are often used — thereby requiring us



(a) (b)

Figure 3.1: Derivation of VLSI Complexity Metrics: (a) Shows the classic derivation of AT^2. The metric is the area of the plane $A^{\frac{1}{2}}T$, (b) Shows the introduction of the normalized error term, E, to the metric. See text for two possible metrics derived from this image.

to determine a new merit function.

Figure 3.1(b) shows an idealized plot of an error function decaying over time. The original plane is extended into a roughly wedge shaped volume. The choice of a decaying error function is purely aesthetic; there is no physical requirement that the approximation converge to some value close to the ideal result over time.

From Fig. 3.1(b) two merit functions may be constructed:

$$\mathcal{C}_1^{\frac{1}{2}}(n,N) = \int_0^T \sqrt{A}\mathcal{E}(n,N,t)\,dt \tag{3.2}$$

$$\mathcal{C}_2^{\frac{1}{2}}(n,N) = \int_0^T \sqrt{A}\mathcal{E}(n,N,T)\,dt \tag{3.3}$$

where $\mathcal{E}$ represents the error.

Equation 3.2 shows the total effect over time of the approximation function. This is useful for comparing designs that may have to be interrupted early. The integration is from the beginning of the calculation to some maximum time limit, T. The entire

volume is integrated to determine the cumulative cost of the convergent algorithm for all $t < T$.

Equation 3.3 compares the effect of the final state for the approximation function, given that the computation halts at time T. Most calculations fall into this category; the user usually waits until T for the result.

Since all the designs presented later in this chapter exhibit a fixed latency for the calculation, Eq. 3.3 will be used as the basis for our cost measure. A further enhancement is to normalize the error, $\mathcal{E}$, by the error produced by bit errors alone:

$$\mathcal{C}(n, N) = A(n, N)T^2(n, N)\frac{E^2(n, N)}{E_0^2(n)} \tag{3.4}$$

where $E(n, N)$ is the error in the computation and $E_0(n)$ is the error due to bit errors as defined below[1].

The error function, Eq. 3.5, reports the cumulative difference between the target function, $f(x)$, and the estimation function, $g_t(N, x)$ for single valued inputs, where t is explicitly represented to emphasize the possibility that the error may change over time. To model the effects of finite word length on fixed point numbers, the $\beta_n(\cdot)$ function rounds the input to a value obtainable with n fraction bits.

$$E^2(n, N) = \int_{-4}^{+4} \left[f(x) - \beta_n\left(g_t(N, x)\right)\right]^2 dx \tag{3.5}$$

$$E_0^2(n) = \int_{-4}^{+4} \left[f(x) - \beta_n\left(f(x)\right)\right]^2 dx \tag{3.6}$$

$$\beta_n(x) = \frac{\updownarrow 2^n x \updownarrow}{2^n} \tag{3.7}$$

where the $\updownarrow \cdot \updownarrow$ operator rounds the argument to the nearest integer.

[1]This formulation is chosen because it reverts back to the AT^2 bound under the condition that $E(n, N) = E_0(n)$. This allows us to compare "exact" arithmetic designs by an AT^2 basis just like before.

Multiple valued inputs require more work since the error must be integrated over each input[2]. This attention to detail is not required, however, since it is known beforehand that the net functions presented here are all "exact". Little is gained by introducing this extra complexity since only the single input activation function will be subjected to error analysis; therefore Eq. 3.5 is appropriate for our needs.

For physical implementations, the binary inputs are limited in both precision and magnitude. The range of integration reflects the limits in magnitude, while the $\beta_n(\cdot)$ accounts for the limits in precision.

In the CORDIC implementation chapter the maximum range of the net function is set to approximately -4 to +4. Therefore, for the purposes of symmetry, Eq. 3.5 uses this range. To justify this apparently arbitrary decision, the following point is offered: the extreme values of the activation function, $\tanh(\pm 4)$ are within 6.707×10^{-4} of the ± 1 boundary. This is very close to ± 1, and would take at least 11 fractional bits to represent the difference.

With the merit function, Eq. 3.4[3], the tools to perform a reasonable assessment of digital neural network designs are available.

The remainder of this chapter partitions the neural network algorithm, Eqs. 1.5 and 1.6, into the network function and the activation function. The various combinations of network and activation functions are put together in the section where the final results are given.

---

[2]In fact it may be formalized for our neural network system by integrating over **vectors** $\mathbf{w}$, and $\mathbf{x}$:

$$E^2(\mathbf{w}, \mathbf{x}, n) = \int_{\mathbf{W}} \int_{\mathbf{X}} \left[f(\mathbf{w}, \mathbf{x}) - \beta_n \left(g_t(\mathbf{w}, \mathbf{x}) \right) \right]^2 \, d\mathbf{x} \, d\mathbf{w} \tag{3.8}$$

where the parameter N is now implicit in the length of $\mathbf{w}$, and $\mathbf{x}$.

[3]With supporting functions Eqs. 3.5, 3.6 and 3.7.

3.2 Evaluation Technology

The technology used for analysis is based on the Canadian Microelectronics Corporation's CMOS4S standard cell chip process[10]. The area and delay numbers reflect the area and propagation delay for unit cells without the effects of parasitic capacitance due to interconnect, therefore these numbers reflect only relative merits of the respective design — the author does not expect that the design will turn out to have the exact area and delay reported here. Table 3.1 shows the values used in this work.

As another note, no attempt has been made to consider the most efficient modules available. There are two reasons for this: (1) since relatively high–level designs are being compared, the low–level modules must be comparable, (2) since this is an a priori assessment of the relative merits of several design approaches it makes little sense to provide as much detail during the assessment as would be done during an implementation – the assessments are meant to be simple and quick. For example, ripple carry adders are used instead of faster adders such as Manchester carry chain or carry lookahead adders.

3.3 Net Functions

The single neuron network functions, Eq. 1.1, are simply vector–vector inner product computations, comprising a multiplication step and an addition step (often combined together). There are three designs under consideration:

- Baugh–Wooley Multiplier with Accumulator. This is the standard design approach to compute inner products (i.e. the net function).
- Coordinate Rotation Digital Computer (CORDIC). The CORDIC method is an iterative method that takes approximately n clock cycles to calculate the multi-

Description		Area (μm^2)		Delay (ns)
Flip Flop	A_{FF}	3086.4	T_{FF}	0.89
And Gate	A_{And}	1900.8	T_{And}	0.44
Nand Gate	A_{Nand}	1267.2	T_{Nand}	0.29
Half Adder	A_{AHA}	3777.6	T_{HA}	0.62
And Half Adder	A_{AHA}	5678.4	T_{HA}	1.06
Nand Half Adder	A_{NFA}	5044.8	T_{NHA}	0.91
Full Adder	A_{AFA}	9619.2	T_{FA}	1.59
Nand Full Adder	A_{NFA}	10886.4	T_{NFA}	1.88
And Full Adder	A_{AFA}	11520.0	T_{AFA}	2.03
Exclusive–Or Gate	A_{Xor}	1876.8	T_{Xor}	0.62
Or Gate	A_{Or}	2064.0	T_{Or}	0.53
Nor Gate	A_{Nor}	1430.4	T_{Or}	0.38
Inverter Gate	A_{Inv}	633.6	T_{Inv}	0.15
Transmission Gate	A_{TG}	316.8	T_{TG}	0.15
2 input Multiplexor	A_{Mux}	1848.0	T_{Mux}	1.11

Table 3.1: Area and Delays for Cells from CMC's CMOS4S Standard Cell Library

ply/accumulation for two n–bit arguments. The real benefits of this method are seen in the section on Activation Functions where it is noted that the CORDIC function is able to calculate the hyperbolic tangent function.

- Serial–Parallel Inner Product Processor (SPIPP). This is a new approach that converts the Baugh–Wooley multiplier into a linear systolic array. In general terms one may say that this method is a streamlined version of the Baugh–Wooley multiplier with accumulator, but projected into a linear systolic array.

3.3.1 Baugh–Wooley

A common method to calculate inner products[22] is a parallel multiplier followed by an adder as shown in Fig. 3.2(a), where the parallel multiplier, shown in Fig. 3.2(b), is a Baugh–Wooley multiplier[4] and the adder is a simple ripple carry adder as was

discussed above[4].

When calculating the time to perform a single multiplication, please note that the AND gates and NAND gates for each cell of Fig. 3.2(b) operate in parallel. Therefore only the greater of these two delays needs to be counted:

$$\begin{aligned} T_{\text{mult}} &= T_{\text{Nand}} + \max(T_{\text{And}}, T_{\text{Nand}}) + T_{\text{HA}} + (2n-3)T_{\text{FA}} + T_{\text{Inv}} \\ &= 3.18n - 5.25\,\text{ns} \\ A_{\text{mult}} &= (n-1)(n-3)A_{\text{AFA}} + (n-1)A_{\text{Nand}} + (n-1)A_{\text{AFA}} + \\ &\quad (n-1)A_{\text{NFA}} + (n-1)A_{\text{AHA}}) + nA_{\text{And}} + A_{\text{Inv}} \\ &= 9619.2n^2 - 9124.8n + 2040\,\mu\text{m}^2 \end{aligned}$$

The time and area to perform the addition and store the result is:

$$\begin{aligned} T_{\text{acc}} &= nT_{\text{FA}} + T_{\text{FF}} \\ &= 1.59n + 0.89\,\text{ns} \\ A_{\text{acc}} &= nA_{\text{AFA}} + nA_{\text{FF}} \\ &= 12705.6n\,\mu\text{m}^2 \end{aligned}$$

Now the time and area required to perform an N element inner product can be calculated:

$$\begin{aligned} T_{\text{Baugh-Wooley}}(n, N) &= N(T_{\text{mult}} + T_{\text{acc}}) \\ &= N(4.77n - 4.36)\,\text{ns} \qquad (3.9) \\ A_{\text{Baugh-Wooley}}(n, N) &= A_{\text{mult}} + A_{\text{acc}} \\ &= 9619.2n^2 + 3580.8n + 2040\,\mu\text{m}^2 \qquad (3.10) \end{aligned}$$

[4]The selection of an 8–bit multiplier is for display purposes only: this work analyzes designs with different word lengths.

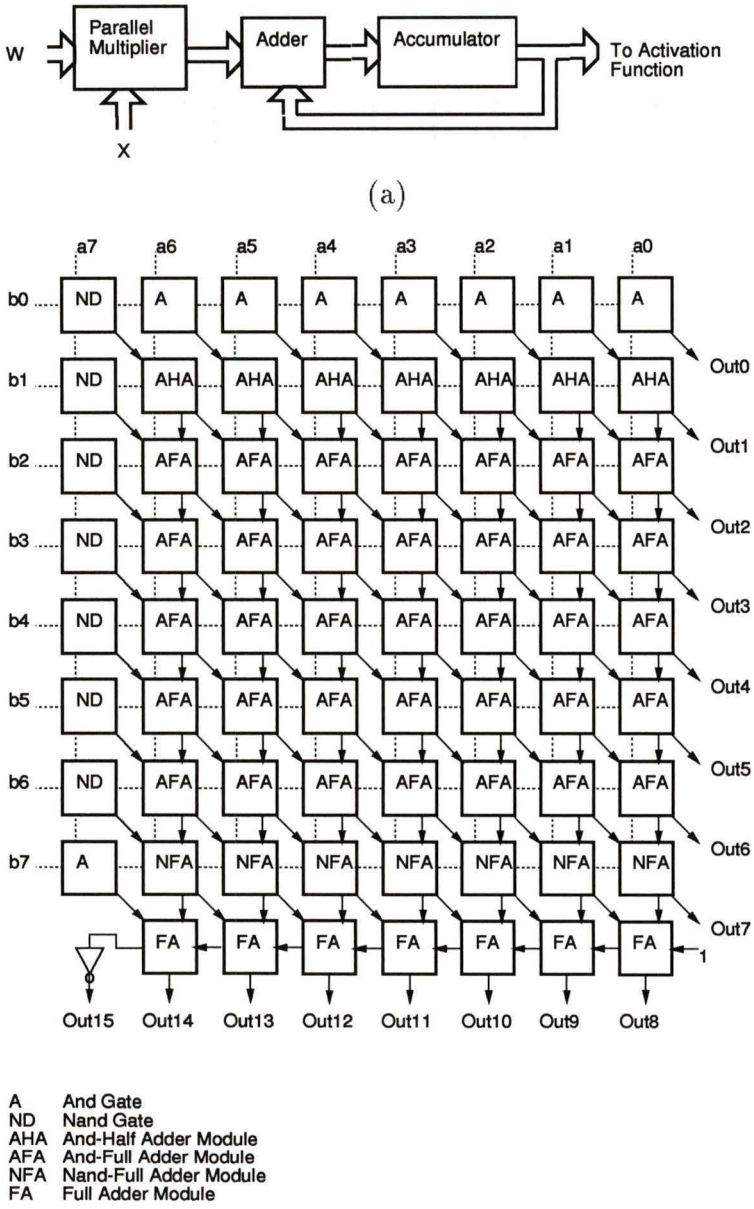


Figure 3.2: Baugh–Wooley and Accumulator Net Function: (a) The use of a parallel multiplier with an accumulator to form a net function, (b) Details of the Baugh–Wooley parallel multiplier

3.3.2 CORDIC

The CORDIC process is an iterative algorithm capable of solving many arithmetic expressions including the multiply and accumulate operation[3]. The CORDIC method is given proper attention in the next chapter; therefore, the inner workings of the method are not covered here. Instead, the simplified diagram of a CORDIC ALU shown in Fig. 3.3 is used to estimate the area and time for such a design.

The 'BS' modules are n–bit shifters to arithmetically shift the input by i bits to the right (typically implemented with a modified barrel shifter, hence the 'BS' designator).

Barrel shifters may be constructed as a square array of transmission gates[61, 26] with the transmission gates routing the input bits to the proper output locations. Only one transmission gate delay is required for each bit. Therefore the area of the barrel shifter is $n^2 A_{\mathrm{TG}}$ and the the delay is T_{TG}, where $A_{\mathrm{TG}}$and $T_{\mathrm{TG}}$are found in Table 3.1.

It is somewhat unfair to describe barrel shifters in standard cell technology, since the majority of implementations in industry use regular array structures. This is a problem with limiting our discussion to the CMOS4S standard cell playing field. In the next section, the reader will encounter a similar problem when modeling ROMs with the standard cell library. However, it must be emphasized that it is not fair to compare designs using different technologies unless the effects of that technology can somehow be removed.

While the details of the CORDIC method are presented later in this thesis, the reader needs to know that the CORDIC method is an iterative process; each iteration through the CORDIC step (i.e. clock pulse) provides an additional bit of accuracy.

From Fig. 3.3 it is seen that the critical delay path for the CORDIC ALU traverses

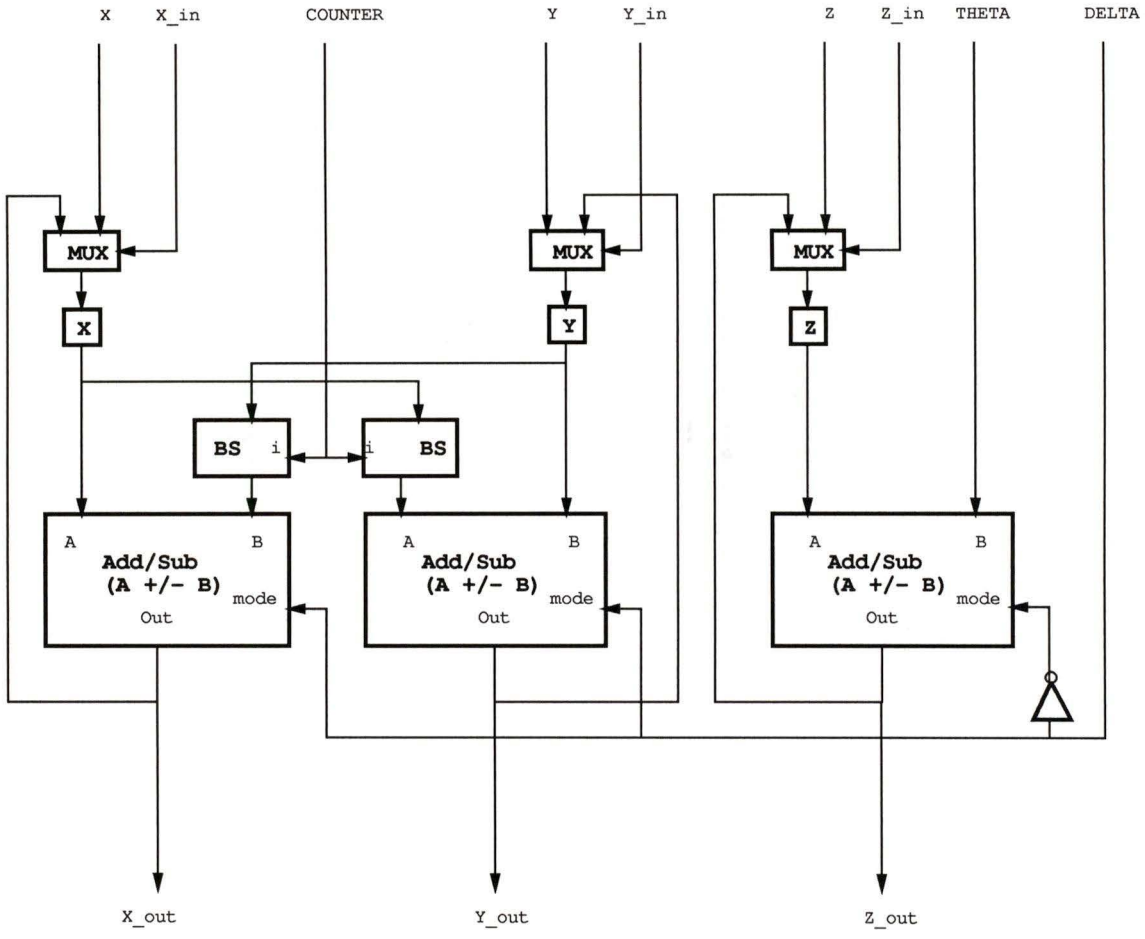


Figure 3.3: Block Diagram of CORDIC Engine

the X or Y register through the shifter module, into the adder/subtracter and finally into the multiplexor. Since the CORDIC ALU requires n cycles to multiply two n–bit numbers, the time is scaled by n to obtain the total computation time.

$$\begin{aligned} T_{\text{CORDIC}}(n,N) &= nN(T_{\text{FF}} + T_{\text{TG}} + nT_{\text{FA}} + T_{\text{Mux}}) \\ &= N(1.59n^2 + 2.15n)\,\text{ns} \end{aligned} \tag{3.11}$$

The area required for the CORDIC ALU is

$$\begin{aligned} A_{\text{CORDIC}}(n,N) &= 3nA_{\text{AFA}} + 3nA_{\text{Mux}} + 3nA_{\text{FF}} + 2n^2A_{\text{TG}} + A_{\text{Inv}} \\ &= 633.6n^2 + 43660.8n + 633.6\,\mu\text{m}^2 \end{aligned} \tag{3.12}$$

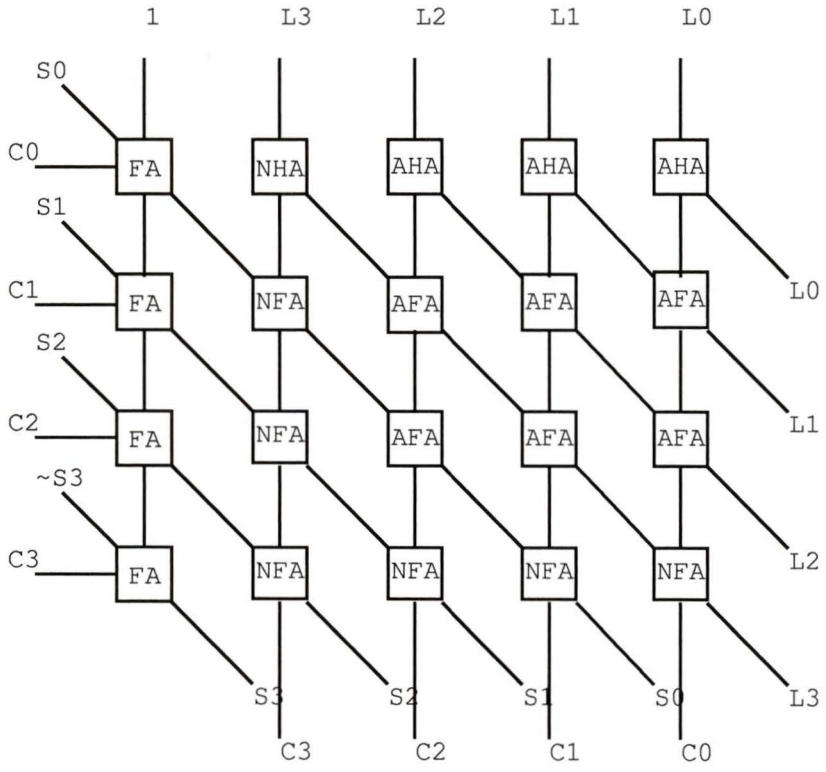
3.3.3 SPIPP

The Serial Parallel Inner Product Processor[22] combines the parallel multiplier with the accumulator/register structure, improving the partitioned Baugh–Wooley design.

Working from the carry–save concept, Fahmi et al altered the Baugh–Wooley parallel multiplier by providing an extra column of full adders on the left edge coupled with some feedback paths[22] (see Fig. 3.4). The column of full adders replaces the ripple adder required in the Baugh–Wooley multiplier for generating the most significant byte from the sums and carries. The final result is formed from the MSB and LSB, where the MSB is $S + C$, and the LSB is L. Therefore the final result is represented as:

$$Q = (S + C)2^n + L \tag{3.13}$$

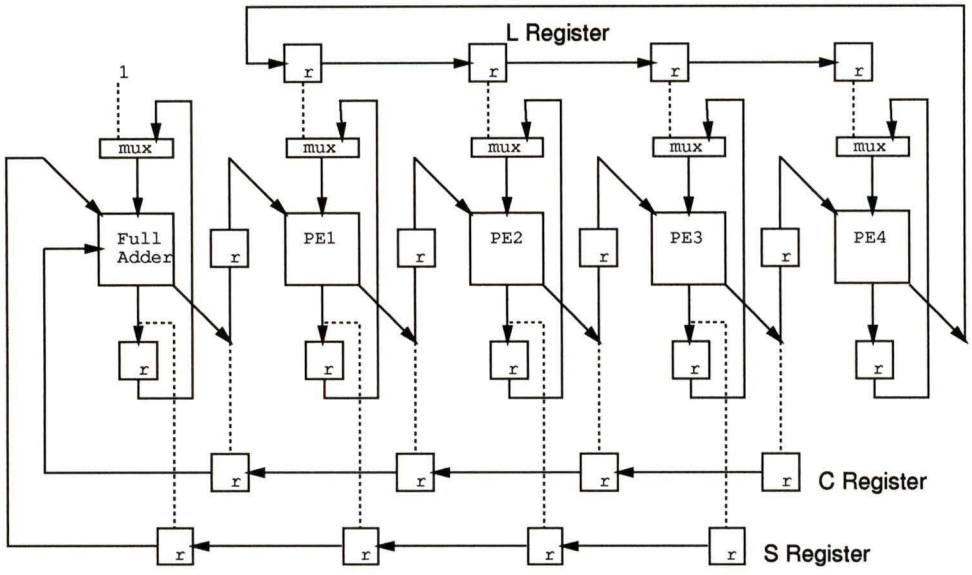
Although faster, this parallel structure requires a lot of area; therefore, Fahmi et al[22] also suggest an alternative serial–parallel design. This reduced–hardware design is obtained by projecting downwards through the multiplier cells, yielding



FA	Full Adder
NHA	Nand Half Adder
NFA	Nand Full Adder
AHA	And Half Adder
AFA	And Full Adder

Note: The Carry signals (C0-C3) on the bottom edge are fed back into the left edge. Similarly the Sum and LSB signals are fed from the bottom edge to the left and top edge respectively. The symbol ~ means logical NOT.

Figure 3.4: A fully parallel inner product processor module based on a Baugh–Wooley multiplier.



Note: The modules (PE1 - PE4) perform the following operations in sequence:

PE1	NHA, NFA, NFA, AFA
PE2	AHA, AFA, AFA, NFA
PE3	AHA, AFA, AFA, NFA
PE4	AHA, AFA, AFA, NFA

Note: Dotted arcs are clocked per word, solid arcs are clocked per bit.

Figure 3.5: A Serial–Parallel Inner Product Processor Module

serial–parallel inner product processor (SPIPP) shown in Fig. 3.5. This structure requires additional control circuitry not shown in the figure to select properly the operations for the processing elements (PE_1 – PE_4).

In the timing analysis below, the final summation from Eq. 3.13 is counted after N multiply and accumulate operations have passed. This is reasonable since it is not necessary to know Q for any but the final values of S and C. Note: T_{PE} is the propagation delay for one processing element, and A_{PE} is the corresponding area.

$$T_{PE} = \max(T_{AHA}, T_{AFA}, T_{NHA}, T_{NFA}) \tag{3.14}$$

$$T_{\text{SPIPP}}(n, N) = Nn(T_{\text{FF}} + T_{\text{Mux}} + T_{\text{PE}} + T_{\text{FF}} + nT_{\text{FA}}) \quad (3.15)$$

$$= N(4.92n) + 1.59n\,\text{ns} \quad (3.16)$$

$$A_{\text{PE}} = A_{\text{AFA}} + A_{\text{Nand}} + A_{\text{And}} \quad (3.17)$$

$$A_{\text{SPIPP}}(n, N) = A_{\text{Mux}} + A_{\text{FF}} + A_{\text{AFA}} + n(5A_{\text{FF}} + A_{\text{Mux}} + A_{\text{PE}}) + nA_{\text{AFA}} \quad (3.18)$$

$$= 30067.2n + 14553.6\,\mu\text{m}^2 \quad (3.19)$$

3.3.4 Net Function Cost Results

The costs for the network function designs are plotted in Fig. 3.6. There are two items of note: (1) the SPIPP method is the best method presented for calculating network values and (2) the CORDIC method is the worst method for calculating network values.

One would expect the SPIPP method to perform well since it is a highly tuned design for calculating inner products (the reader will recall that the net value is an inner product). The value of the CORDIC method, however, is not with calculating inner products. As will be seen soon, the CORDIC method makes up for its higher net function cost by providing a very low cost activation function.

3.4 Activation Functions

In contrast to the net function techniques above, not all the activation function algorithms yield "exact" results. Therefore, the error component of the cost metric is much more important for evaluations of the activation functions.

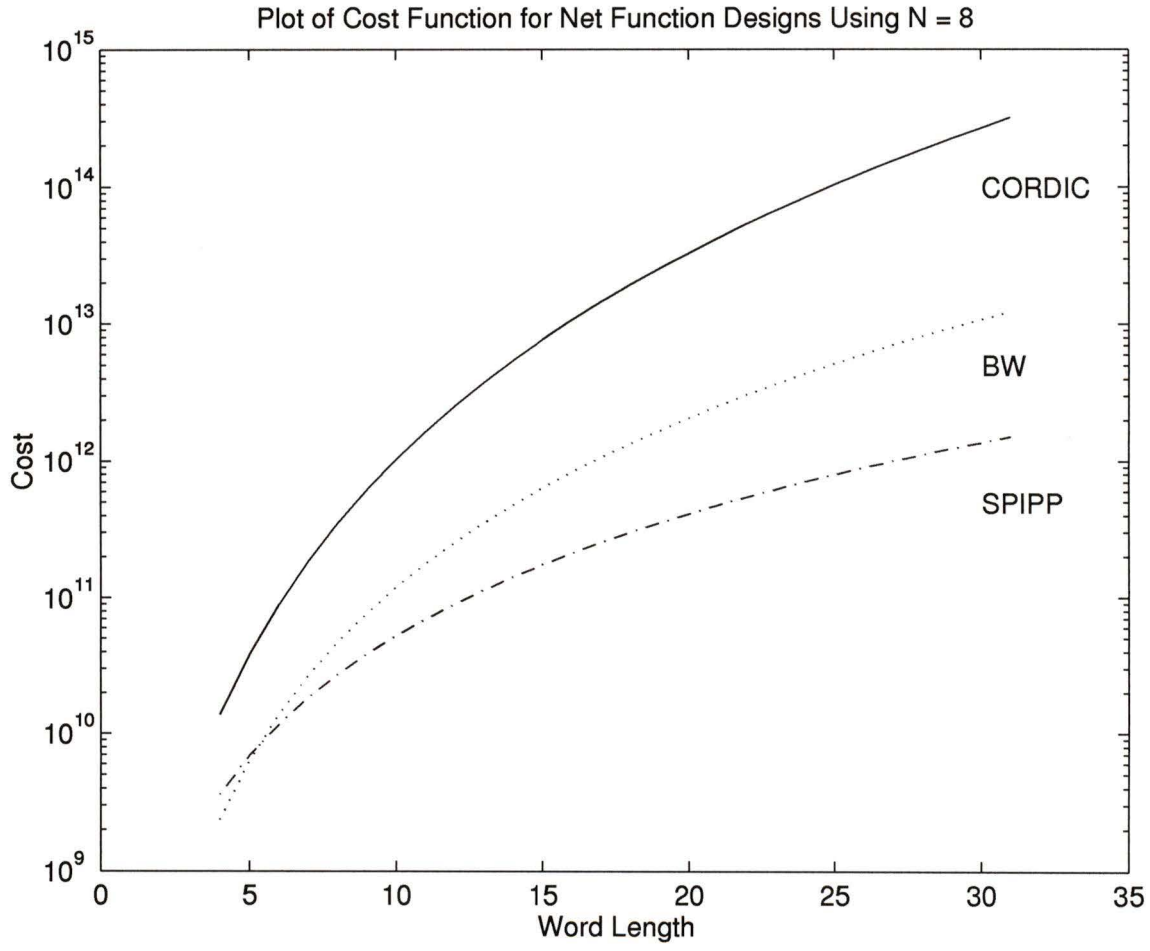


Figure 3.6: Plot of Cost Functions for Net Function Designs

There are five approaches to approximating the hyperbolic tangent activation function under evaluation:

- Hard limiter. The hard limiter is used mostly in Hopfield networks, but occasionally it is used for other network architectures as well.

- Read Only Memory. Large neuro–computers often store the activation function in a ROM for quick access, however, as will be seen shortly, the ROM is not particularly well suited for smaller systems where a major objective is to minimize hardware.

- CORDIC. The CORDIC method performs very well here because of its ability to calculate the hyperbolic tangent function to arbitrary precision.

- Seventh Order Chebyshev Polynomial. Polynomial approximation is often used by mathematicians and engineers, however the large number of multiplications and additions poses major problems for this method.

- Three Segment Linear Approximation. The three segment linear (or saturated ramp) approximation is a natural extension to the hard limiter. The benefits of its simple hardware is negated by its crude approximation to the hyperbolic tangent function.

Of these, only the ROM and CORDIC approaches yield results within one bit of the correct result[5]. The final two approaches are purely approximation techniques.

[5]A distinguishing characteristic between the CORDIC and ROM approaches is that the ROM approach is "exact" within a clock cycle, while the CORDIC technique is iterative in nature, requiring time to converge to the same "exact" result the ROM finds immediately.

3.4.1 Hard Limiter

For feed forward networks most authors and researchers use the soft limiter (sigmoidal) function. The sigmoid function is good for the backprop algorithm (Eqs. 1.8 to 1.11) because the derivative exists at all locations and because the slope of the curve is flat only as $|x| \to \infty$.

The hard limiter function, however, is flat over the entire region (excluding the origin) and offers no chance for the backprop algorithm to alter the weights.

However, even with these bad points the hard limiter function is still used in neural networks. Typically, hard limiters are used for the Hopfield network[31, 32].

This function may be implemented using an n–bit wide mux, with the sign of x as the selector.

$$T_{\text{HARD}}(n) = T_{\text{Mux}} \tag{3.20}$$

$$= 1.11\,\text{ns} \tag{3.21}$$

$$A_{\text{HARD}}(n) = nA_{\text{Mux}} \tag{3.22}$$

$$= 1848.0n\,\mu\text{m}^2 \tag{3.23}$$

$$E^2(n, N) = 0.7725885 \tag{3.24}$$

3.4.2 ROM

A very common method to evaluate the hyperbolic tangent function in neural networks is the ROM based lookup table. A typical mode of operation is to use the n–bit net function result as an address into the ROM, resulting in n bits returned from the ROM. Without any special encoding (such as μLaw), 2^n addressable words of n bits are required — thereby using $n2^n$ bits of storage.

The area and delay for combinational logic is as reported in Table 3.1, and ROM cells are represented as transmission gates since they may be constructed from single transistors.

It must be cautioned that ROMs are not constructed efficiently from standard cells. They are, in general, constructed from regular array structures to ensure high circuit density. Therefore, it is a bit of a stretch to use the area and time measurements for standard cells, Table 3.1, to analyze ROM efficiency. This is the same problem faced when evaluating the barrel shifters within the CORDIC design. The arguments posed there apply equally to this case.

The area of the ROM depends on three things: the area of the bit storage itself, the area of the column decoders and the area of the row decoders.

The column and row decoders are usually designed to distribute the address bits over two dimensions such that the storage bit array becomes square (or at least, nearly square). The row decoder uses m bits, while the column decoder uses k bits. The total number of address bits is, of course, $n = m + k$.

In the evaluations that follow, the row decoder is constructed from NOR gates[61, p. 356], and the column decoder is constructed from a tree of transmission gates[61, p. 361].

$$\begin{aligned} T_{\text{col}}(k) &= kT_{\text{TG}} \\ T_{\text{row}}(m) &= T_{\text{Nor}} \\ T_{\text{ROM}}(n) &= T_{\text{TG}} + \max(T_{\text{col}}, T_{\text{row}}) \\ &= T_{\text{TG}} + \max(kT_{\text{TG}}, T_{\text{Nor}}) \end{aligned}$$

$$\begin{aligned} A_{\text{col}}(k) &= 2(2^k - 1)A_{\text{TG}} \\ A_{\text{row}}(m) &= 2^m A_{\text{Nor}} \end{aligned}$$

$$\begin{aligned} A_{\mathrm{ROM}}(n) &= n2^n A_{\mathrm{TG}} + A_{\mathrm{col}} + A_{\mathrm{row}} \\ &= n2^n A_{\mathrm{TG}} + 2(2^k - 1)A_{\mathrm{TG}} + 2^m A_{\mathrm{Nor}} \end{aligned}$$

For simplicity, it is assumed that $m = k = \frac{1}{2}n$. This actually isn't that far from reality. If the ROM is optimized for the area of the ROM, the best k becomes:

$$k = \frac{1}{2}n + \frac{\log_2(A_{\mathrm{Nor}}/A_{\mathrm{TG}}) - 1}{2}$$

Since the NOR gate uses four transistors to the single transistor in the transmission gate, $k = \frac{1}{2}(n-1)$ which is very close to our approximation for large n. Also, for a ROM of useful size, it can be assumed that the delay through a NOR gate is less than the delay through $\frac{1}{2}n$ transmission gates.

$$\begin{aligned} T_{\mathrm{ROM}}(n) &= T_{\mathrm{TG}} + \frac{1}{2}nT_{\mathrm{TG}} && (3.25) \\ &= 0.075n + 0.15\,\mathrm{ns} && (3.26) \\ A_{\mathrm{ROM}}(n) &= n2^n A_{\mathrm{TG}} + 2(2^{\frac{1}{2}}n - 1)A_{\mathrm{TG}} + 2^{\frac{1}{2}n} A_{\mathrm{Nor}} && (3.27) \\ &= 316.8n2^n + 2063.22^{\frac{1}{2}n} - 316.8\,\mu\mathrm{m}^2 && (3.28) \\ E^2(n,N) &= E_0^2(n) && (3.29) \end{aligned}$$

3.4.3 CORDIC

The CORDIC activation function operates much like the CORDIC net function with slight modifications. When calculating the activation function, the CORDIC technique actually has two distinct stages: the hyperbolic stage, and the division stage. During the hyperbolic stage the values for $\sinh(u)$ and $\cosh(u)$ are calculated in parallel. The division stage calculates the hyperbolic tangent via the identity $\tanh(u) = \sinh(u)/\cosh(u)$.

The division stage requires n CORDIC steps to calculate just like the multiplication/addition stage; however, the hyperbolic stage typically requires more than n CORDIC steps. Without going into details (which are covered in the next chapter) the extra steps are modeled by the parameter ρ_i:

$$\rho_i = \text{Count of elements}\{3, 4, 7, 12, 13, 18, 19, 21, \ldots\} < i \tag{3.30}$$

As a simple example; the value for ρ_{16} is the cardinality of the set {3, 4, 7, 12, 13} which is 5. The set of elements is from [27, 3].

$$\begin{aligned}
T_{\text{HYP}} &= (n+\rho_n)T_{\text{FF}} + (n+\rho_n)T_{\text{TG}} + (n+\rho_n)^2 T_{\text{FA}} + (n+\rho_n)T_{\text{Mux}} \\
T_{\text{DIV}} &= nT_{\text{FF}} + nT_{\text{TG}} + n^2 T_{\text{FA}} + nT_{\text{Mux}} \\
T_{\text{CORDIC}}(n) &= T_{\text{HYP}} + T_{\text{DIV}} \\
&= (2n+\rho_n)T_{\text{FF}} + (2n+\rho_n)T_{\text{TG}} + (2n+\rho_n)^2 T_{\text{FA}} + (2n+\rho_n)T_{\text{Mux}} \\
&= 6.36n^2 + 4.3n + 1.59\rho_n^2 + 2.15\rho_n + 3.18n\rho_n \text{ ns} \qquad (3.31) \\
A_{\text{CORDIC}}(n) &= 3nA_{\text{AFA}} + 3nA_{\text{Mux}} + 3nA_{\text{FF}} + 2n^2 A_{\text{TG}} + A_{\text{Inv}} \\
&= 633.6n^2 + 43660.8n + 633.6\,\mu\text{m}^2 \qquad (3.32) \\
E^2(n) &= E_0^2(n) \qquad (3.33)
\end{aligned}$$

3.4.4 Chebyshev

Instead of finding the exact solution for the hyperbolic tangent function it can be approximated with a seventh–order Chebyshev polynomial[36, 46].

The Chebyshev polynomials are defined as follows[46]:

$$\begin{aligned}
T_0(x) &= 1 \\
T_1(x) &= x
\end{aligned}$$

$$T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x) \quad n > 2 \tag{3.34}$$

Using the methods of [46, sec. 5.6], the approximation function for the hyperbolic tangent is:

$$f(x) \approx \left[\sum_{k=0}^{m-1} c_k T_k(x/4)\right] - \frac{1}{2}c_0$$

Where the Chebyshev coefficients c_i are:

$$< 0, 1.238, 0, -0.336, 0, 0.142, 0, -0.0642 >$$

While the approximating polynomial may be obtained from Eq. 3.34, it is suggested that the Clenshaw's recurrence formula is used[46, p. 161]:

$$\begin{aligned} d_{m+1} &= 0 \\ d_m &= 0 \\ d_j &= 2xd_{j+1} - d_{j+2} + c_j \quad j = m-1, m-2, \ldots, 1 \\ f(x) &\approx d_0 = xd_1 - d_2 + \frac{1}{2}c_0 \end{aligned}$$

Please note that since the range of the series is extended from the default [-1, +1] to [-4, +4], x is prescaled by 0.25.

The time and area requirements for the Clenshaw's recurrence solution of the Chebyshev polynomial are:

$$\begin{aligned} T_{\text{mult}} &= 3.18n - 5.25\,\text{ns} \\ A_{\text{mult}} &= 9619.2n^2 - 9124.8n + 2040\,\mu\text{m}^2 \\ T_{\text{Chebyshev}} &= 7T_{\text{mult}} + 13nT_{\text{FA}} \\ &= 22.26n - 36.75\,\text{ns} \qquad (3.35) \\ A_{\text{Chebyshev}} &= A_{\text{mult}} + nA_{\text{AFA}} \\ &\quad 9619.2n^2 + 494.4n + 2040\,\mu\text{m}^2 \qquad (3.36) \\ &\qquad (3.37) \end{aligned}$$

n	$E^2(n)$	n	$E^2(n)$
4	0.0327892	18	0.0269092
5	0.0275411	19	0.0269092
6	0.0270782	20	0.0269092
7	0.0268929	21	0.0269092
8	0.0269145	22	0.0269092
9	0.0268928	23	0.0269092
10	0.0269135	24	0.0269092
11	0.0269103	25	0.0269092
12	0.0269094	26	0.0269092
13	0.0269092	27	0.0269092
14	0.0269093	28	0.0269092
15	0.0269092	29	0.0269092
16	0.0269092	30	0.0269092
17	0.0269092	31	0.0269092

Table 3.2: Table of $E^2(n)$ for the Chebyshev Polynomial Approximation to the Hyperbolic Tangent Function.

The errors for the Chebyshev polynomial approximation are found in Table 3.2.

3.4.5 Three Segment

The three segment approximation to the hyperbolic tangent function is shown in Fig. 3.7. This is really nothing more than adjusting the hard limiter — the vertical jump of the hard limiter is softened to a ramp. For the purposes of area and time analysis please view Fig. 3.8. A simple multiplexor may be used to construct such a design.

$$g(x) = \begin{cases} -1 & x < -1 \\ x & -1 < x < +1 \\ +1 & x > +1 \end{cases} \tag{3.38}$$

$$E^2(n) = \int_{-4}^{+4} \left[\tanh(x) - \beta_n\left(g(x)\right)\right]^2 dx \tag{3.39}$$

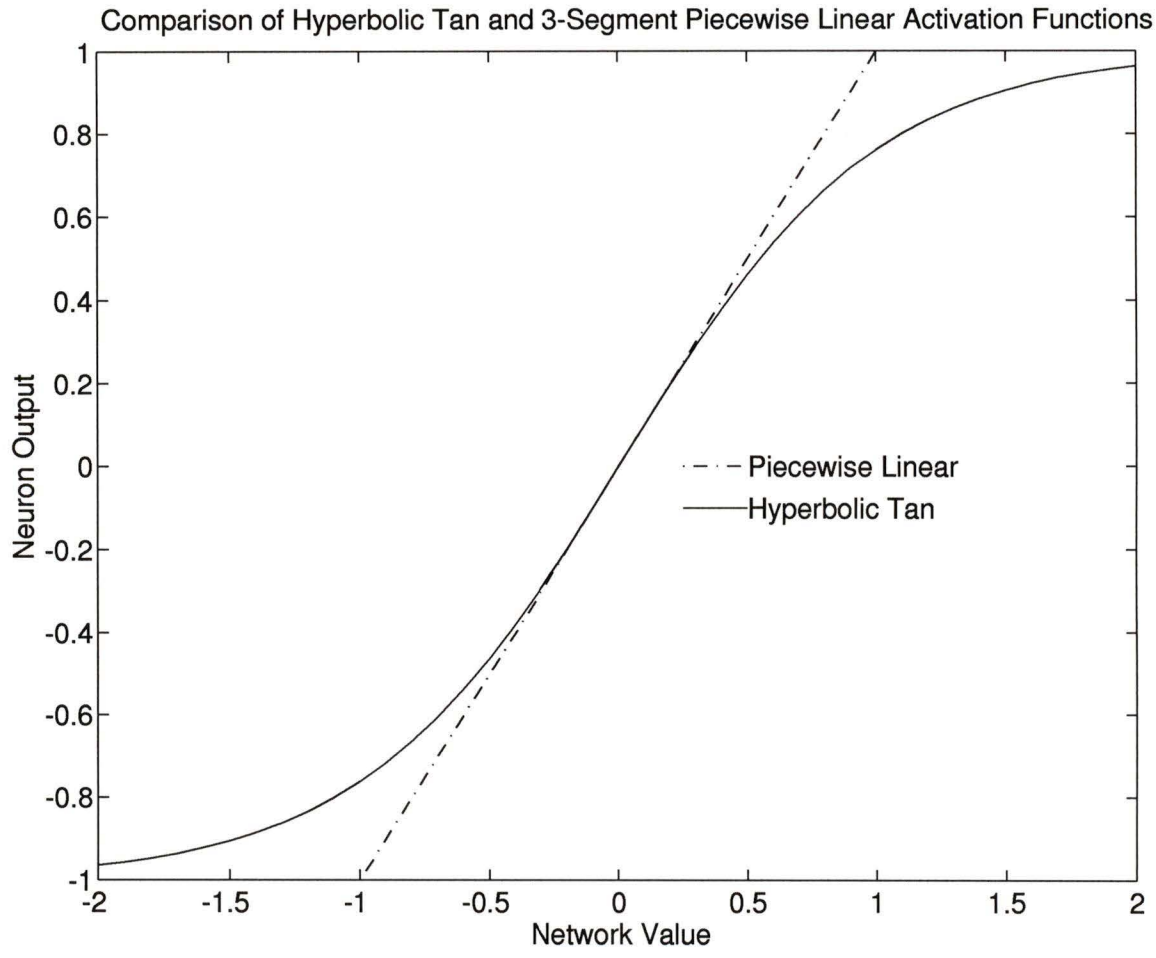


Figure 3.7: Three Segment Piecewise Linear Approximation to the Hyperbolic Tangent Function

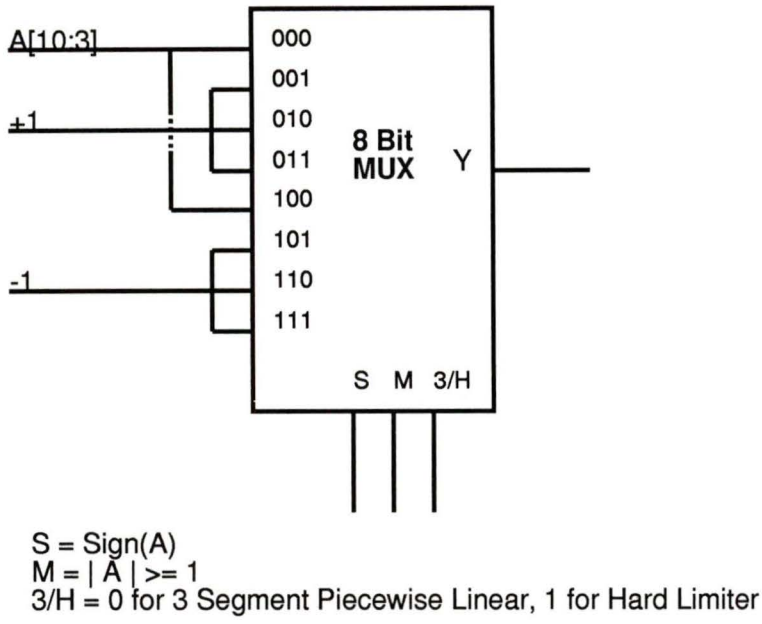


Figure 3.8: Hardware for three Segment Approximation to Hyperbolic Tangent Function

(3.40)

Due to the introduction of $\beta_n(\cdot)$, it is not feasible to generate a symbolic result for $E^2(n)$. I've therefore compiled the results in Table 3.3. When viewing the table please keep in mind that even without any word length errors introduced by $\beta_n(\cdot)$ the error is still 0.04957554; therefore, the effect of these errors are quickly clouded by the inherent error of the approximation itself.

$$T_{3\text{seg}} = 2T_{\text{Mux}} \tag{3.41}$$

$$= 2.22\,\text{ns}$$

$$A_{3\text{seg}} = 2nA_{\text{Mux}} \tag{3.42}$$

$$= 3696n\,\mu\text{m}^2 \tag{3.43}$$

$$E_{3\text{seg}} = \text{Table 3.3} \tag{3.44}$$

n	$E^2(n)$	n	$E^2(n)$
4	0.05038086	18	0.04957554
5	0.04977688	19	0.04957554
6	0.04962528	20	0.04957554
7	0.04958827	21	0.04957554
8	0.04957887	22	0.04957554
9	0.04957645	23	0.04957554
10	0.04957580	24	0.04957554
11	0.04957559	25	0.04957554
12	0.04957552	26	0.04957554
13	0.04957554	27	0.04957554
14	0.04957554	28	0.04957554
15	0.04957554	29	0.04957554
16	0.04957554	30	0.04957554
17	0.04957554	31	0.04957554

Table 3.3: Table of $E^2(n)$ for the Three Segment Piecewise Linear Approximation to the Hyperbolic Tangent Function

(3.45)

3.5 Complete Systems

The complete system is made up of a net function and activation function. In general, the net function operates independently from the activation function. For example, the area consumed by the Baugh–Wooley net function does not overlap with the ROM activation function, neither do they overlap temporally during calculation.

Therefore, for designs where the net function is independent from the activation function in both area and time, the total cost function is:

$$\mathcal{C}(n,N) = (A_{\text{net}}(n,N) + A_{\text{act}}(n,N))(T_{\text{net}}(n,N) + T_{\text{act}}(n,N))^2 \frac{E^2_{\text{act}}(n,N)}{E^2_0(n,N)}$$

Only the CORDIC–CORDIC combination breaks this rule. This is because the CORDIC net function is the same hardware as the CORDIC activation function.

Since these two modules operate sequentially, the the same physical module can be used for both functions. Therefore the area of one CORDIC module is counted only once.

All possible combinations of activation function and network function are considered, yielding 15 composite cost functions. Since it is very difficult to interpret the composite cost functions directly, all 15 combinations are plotted in semilog format in Fig. 3.9. It may interest the reader that Hollis, Harper and Paulos have found that a word length of 13 is optimal for multi–layer perceptrons[30]. This lies just after the intersection of the ROM activation family of cost curves, and the three–segment activation curves.

Note that the CORDIC activation methods appear in the bottom grouping. Furthermore, the CORDIC–CORDIC design has the lowest cost of all the designs examined for the entire range plotted. Therefore the CORDIC–CORDIC method is superior to the other tested designs.

It is worth mentioning that the definition of the error function has a dramatic effect on the composite curves. The three components of the cost function (area, delay, and error) can be weighted against each other. This makes it difficult to compare designs. However, since all the designs are evaluated on an equal basis, the relative strengths for each design ought to be apparent in the cost plots.

3.6 Summary

This chapter examined arithmetic designs prior to the implementation phase. The standard VLSI complexity metric is not useful for the designs considered; therefore, an extended merit function was introduced to account for the errors caused by approximating the target function.

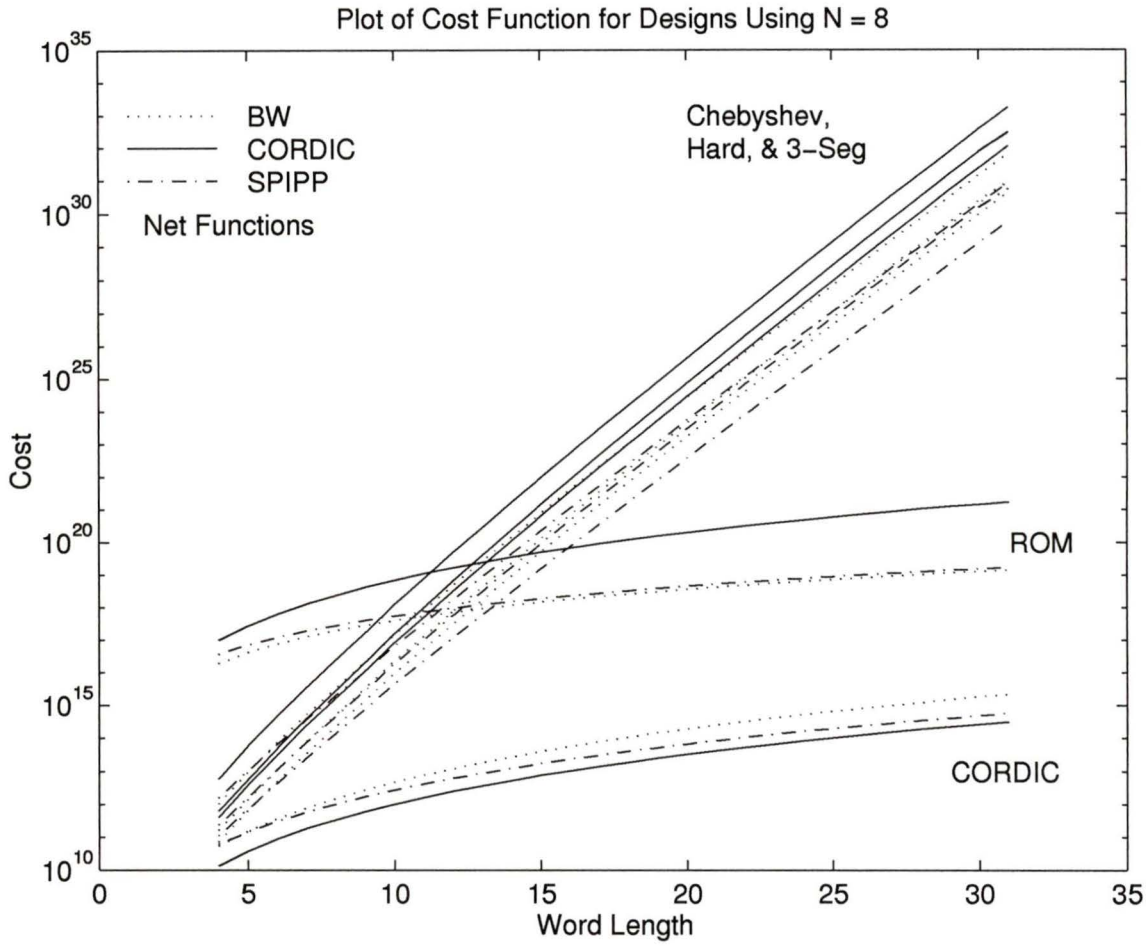


Figure 3.9: Plot of Cost Functions. As mentioned in the text, the case of CORDIC–CORDIC is represented as a single instance of the hardware for both net function and activation function.

With the new merit function the chapter evaluated three network functions (Baugh–Wooley, CORDIC, and SPIPP), and five activation functions (hard limiter, ROM, CORDIC, Chebyshev, and three segment). The plots of the total cost curves are presented in Fig. 3.9.

From the area and time expressions for both the net function and the activation function it is observed that the CORDIC function is not the best net function, but makes up for any deficiency through its superior activation function. It is the ability to have the net function and activation function implemented in the same hardware that gives the CORDIC–CORDIC design the edge.

Chapter 4

The CORDIC Artificial Neuron

The coordinate rotation digital computer (CORDIC), created by J. E. Volder[59] in 1959, was originally intended to solve air navigation problems by using properties of rotation matrices. Since then, researchers have used the CORDIC method for many other applications, such as: DSP[4, 18], hand held calculators[60], and computer graphics[57]. Part of this chapter shows that the CORDIC method is also well suited for neural networks.

4.1 Derivation of the CORDIC Step

As is well known, rotations may be performed by multiplying a vector by a rotation matrix. Furthermore, a single rotation can be divided into an arbitrary number of small rotations, as long as the accumulated angles of the small rotations is the same as the single large rotation. Volder's CORDIC method employs a series of rotations; large at first, progressing towards smaller angles over time.

In 1971, Walther[60] extended the CORDIC method by showing that, with the

introduction of a coordinate system parameter, the circular, hyperbolic, and linear coordinate planes have a similar rotation matrix. This unified approach forms the foundation material for much of this chapter. The derivation, along the lines of Walther, is given below.

The starting point is the rotation matrices for the circular and hyperbolic coordinate systems:

$$\mathbf{R}_{\text{cir}} = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i) \end{bmatrix} \tag{4.1}$$

$$\mathbf{R}_{\text{hyp}} = \begin{bmatrix} \cosh(\theta_i) & \sinh(\theta_i) \\ \sinh(\theta_i) & \cosh(\theta_i) \end{bmatrix} \tag{4.2}$$

The coordinate domain parameter, m, is defined as follows: $m = +1$ for the circular coordinate system, and $m = -1$ for the hyperbolic coordinate system[1]. This definition allows the unified definition of the norm and angle for both coordinate systems:

$$N_m = \sqrt{x_0^2 + m y_0^2} \tag{4.3}$$

$$\theta_m = (1/\sqrt{m}) \tan^{-1}(\sqrt{m} y_0 / x_0) \tag{4.4}$$

The coordinate domain parameter applied to the two rotation matrices gives us a single rotation matrix. The form of the rotation matrix is indexed by i to emphasize that the CORDIC method rotates through a series of angles.

$$\begin{aligned} \mathbf{R}_{m,i} &= \begin{bmatrix} \cos(\sqrt{m}\theta_i) & -\sqrt{m}\sin(\sqrt{m}\theta_i) \\ m\sqrt{m}\sin(\sqrt{m}\theta_i) & \cos(\sqrt{m}\theta_i) \end{bmatrix} \\ &= \cos(\sqrt{m}\theta_i) \begin{bmatrix} 1 & -\sqrt{m}\tan(\sqrt{m}\theta_i) \\ m\sqrt{m}\tan(\sqrt{m}\theta_i) & 1 \end{bmatrix} \end{aligned} \tag{4.5}$$

[1]The reader will see later that $m = 0$ represents the linear coordinate system.

The rotation matrix in Eq. 4.5 rotates vectors by a known rotation angle; in fact, the rotation is *defined* in terms of the rotation angle. However, the reverse is also possible; define the rotation matrix first, and derive the rotation angle from it. For lack of a better term, the cross diagonal terms of the modified rotation matrix are called the rotation magnitude. The following rotation matrix shows this transformation:

$$\mathbf{R}_i = k_i \begin{bmatrix} 1 & -m\mu_i\delta_i \\ \mu_i\delta_i & 1 \end{bmatrix} \tag{4.6}$$

where

$$\begin{aligned} k_i &= \cos(\sqrt{m}\theta_i) = \text{Scale Factor} \\ \mu_i &= \text{Direction of Rotation} \in \{+1, -1\} \\ \delta_i &= \text{Magnitude of Rotation} \end{aligned}$$

Comparing Eq. 4.5 with Eq. 4.6 reveals the rotation angle for the corresponding magnitude of rotation:

$$\theta_i = \frac{1}{\sqrt{m}} \tan^{-1}(\sqrt{m}\mu_i\delta_i) \tag{4.7}$$

A vector may be rotated by applying the rotation matrix as in:

$$\mathbf{X}_{i+1} = \mathbf{R}_{m,i}\mathbf{X}_i \tag{4.8}$$

The rotation matrix, Eq. 4.6, may be segmented into the scalar part and the matrix part. The scalar portion is denoted k_i as above, and further denote the matrix part as $\mathbf{R}^*$. With these changes in place Eq. 4.8 may be represented by the following:

$$\begin{aligned} \mathbf{X}_{i+1} &= \mathbf{R}_{m,i}\mathbf{X}_i \\ &= k_i\mathbf{R}^*\mathbf{X}_{m,i} \end{aligned} \tag{4.9}$$

Since these operations are linear, the successive scaling factors, k_i, may be applied after the series of rotations have been performed:

$$\mathbf{X}^*_{i+1} = \mathbf{R}^*_{m,i}\mathbf{X}^*_i \tag{4.10}$$

$$X_n = K\mathbf{X}^*_n \tag{4.11}$$

where

$$K = \prod_{i=1}^{n} k_i$$

$$X^*_1 = X_1$$

Equations 4.10 and 4.11 break the successive rotation operations into the rotation phase and the scaling phase respectively[2].

The so called CORDIC step is Eq. 4.10 written out in full. In order to trace the accumulated angle represented by the series of steps, a new variable, z_i, is introduced. Another enhancement, is to assign the magnitude of rotation to a value easily represented with binary computers; $\delta_i = 2^{-\mathcal{S}}$. The generalized function $\mathcal{S}$ represents the CORDIC schedule. In brief, the CORDIC schedule determines the order of the CORDIC steps. This is covered with more detail in Section 4.3. The CORDIC step is now represented as follows:

$$x_{i+1} = x_i - m\mu_i 2^{-\mathcal{S}} y_i \tag{4.12}$$

$$y_{i+1} = y_i + \mu_i 2^{-\mathcal{S}} x_i \tag{4.13}$$

$$z_{i+1} = z_i - \mu_i \theta_{i,m} \tag{4.14}$$

As considered earlier, there are two distinct ways the CORDIC step may be employed: (1) rotation mode, (2) vector mode. A mode variable, r, is used to differ-

[2]Please note that the result of the CORDIC rotations is *multiplied* by the scale factor; the literature often divides the result by the reciprocal scale factor. The form shown here follows directly from the rotation matrix derivation.

entiate between these modes: $r = 1$ is rotation mode, while $r = 0$ represents vector mode.

Rotation mode selects a known (x_0,y_0) starting position and rotates by a preset angle. This is accomplished by setting z_0, and rotating until $z_n = 0$. The final position (x_n,y_n) is returned. From examining Eqs. 4.12–4.14, the direction of rotation, μ_i, is the sign of z_i.

Vector mode sets a known (x_0,y_0) starting position and rotates until $y_n = 0$ (i.e. the vector rests on the y axis). The total angle of rotation is accumulated in z_n. To reduce y_i to zero the quadrant the vector is in must be examined For vectors in quadrants (I) and (III) the rotation direction is clockwise. Quadrants (II) and (IV) should rotate anti–clockwise to meet the y axis. The quadrants can be found by testing the polarity of the x and y registers.

The variable y or z is called an indicator variable when it is intended to converge to zero as given above. The term 'indicator variable' implies that it tracks the remaining angle (for z) or y–value (for y), and may also indicate when the iterations have progressed far enough.

The rotation direction is presented easier in equation form. Therefore the following equation symbolizes the ideas posed in the previous few paragraphs:

$$\mu_i = \begin{cases} \text{sign}(z_i) & r = 1 \quad \text{(Rotating)} \\ -\text{sign}(x_i)\,\text{sign}(y_i) & r = 0 \quad \text{(Vectoring)} \end{cases} \tag{4.15}$$

For each of the coordinate systems, Eq.4.7 can be shown to be as follows[60]:

$$\theta_{i,m} = \begin{cases} \tan^{-1}(2^{-s}) & m = +1 \quad \text{(circular)} \\ 2^{-s} & m = 0 \quad \text{(linear)} \\ \tanh^{-1}(2^{-s}) & m = -1 \quad \text{(hyperbolic)} \end{cases} \tag{4.16}$$

Function CORDIC(x, y, z, m, r)
x,y,z $\leftarrow$ inital states
m $\leftarrow$ Coordinate System
r $\leftarrow$ Rotation/Vectoring

for i = 1 to n, step size 1
{
$\mu \leftarrow (r \stackrel{?}{=} 1)\ ?\ \text{sign}(z)\ :\ -\text{sign}(x)\,\text{sign}(y)$

$x' \leftarrow x$
$y' \leftarrow y$

$x \leftarrow x - m\mu 2^{-\mathcal{S}} y'$
$y \leftarrow y + \mu 2^{-\mathcal{S}} x'$
$z \leftarrow z - \mu\theta_{i,m}$
}

Figure 4.1: Pseudocode Illustrating the CORDIC Step. The CORDIC schedule is represented by $\mathcal{S}$.

Please note that Eq. 4.16 exists for $m = 0$. This is interpreted as the linear coordinate system, where rotating and vectoring operations result in multiplication and division mathematical operations respectively.

Figure 4.1 gives the CORDIC algorithm in a pseudocode format. The algorithm, as shown in Fig. 4.1, exits when $i = n$; it could, however, exit early if the indicator variable is below some threshold. Both the CORDIC schedule and the early termination concepts are examined, and the results are given later on.

The list of available functions is given in Table 4.1. The table is formed as a matrix of the coordinate system, m, and the rotation/vectoring mode, r. Functions are referred by using the coordinates in the table. For example, the multiply/accumulate function is referred to as $(r = 1, m = 0)$.

m	r = 1 (Rotating)	r = 0 (Vectoring)
-1 (Hyp)	$x_n = K_{-1}(x_0\cosh(z_0) + y_0\sinh(z_0))$ $y_n = K_{-1}(x_0\sinh(z_0) + y_0\cosh(z_0))$	$x_n = K_{-1}\sqrt{x_0{}^2 - y_0{}^2}$ $z_n = z_0 + \tanh^{-1}(y_0/x_0)$
0 (Lin)	$x_n = x_0$ $y_n = x_0 z_0 + y_0$	$x_n = x_0$ $z_n = z_0 + y_0/x_0$
+1 (Cir)	$x_n = K_{+1}(x_0\cos(z_0) - y_0 sin(z_0))$ $y_n = K_{+1}(x_0\sin(z_0) + y_0\cos(z_0))$	$x_n = K_{+1}\sqrt{x_0{}^2 + y_0{}^2}$ $x_n = z_0 + \tan^{-1}(y_0/x_0)$

Table 4.1: Functions Generated by CORDIC Method

For nomenclature purposes the functions in Table 4.1 are known as CORDIC operations. When considering performance, one may speak of COPS (CORDIC Operation Per Second).

4.1.1 Summary of CORDIC method

There are four components to the CORDIC approach as seen in the derivation of the CORDIC method:

1. The CORDIC step equations (Eqs. 4.12–4.14) represent the heart of the CORDIC method. The rest of the method is meant to control these CORDIC steps.

2. The direction of rotation (Eq. 4.15) induces a rotation or vector operation depending on r. By defining μ_i in terms of the signs of x, y, and z the so–called "zero seeking" property of CORDIC processors is achieved.

3. The angle of rotation (Eq. 4.16) tracks the accumulated rotated angle. Generally this is implemented in hardware as a relatively short ROM for the circular and hyperbolic CORDIC operations.

4. The CORDIC schedule determines the process by which the CORDIC method converges to the final result.

4.2 Convergence Requirements

The choice of $\delta_i = 2^{-S}$ in Eqs. 4.12–4.14 combined with the CORDIC schedule determines a sequence of rotation angles, $\theta_{i,m}$, executed by the CORDIC step. The nature of the rotation sequence determines if the CORDIC method converges to the correct value.

There are two convergence requirements: (1) the domain of the CORDIC function must be large enough to accept any anticipated input (the external convergence requirement), and (2) given that the external convergence requirement is met, the sequence of CORDIC steps must converge to the correct value (the internal convergence requitement). The case where $z_0 \rightarrow 0$ is considered here. The alternative case when $y_0 \rightarrow 0$ follows a similar line of logic.

The external convergence requirement is:

$$\max(|z_0|) \leq \Theta_m + \theta_{n,m} \tag{4.17}$$

where the maximum accumulated rotation is given as:

$$\Theta_m = \sum_{i=0}^{n} \theta_{i,m} \tag{4.18}$$

Eq. 4.17 gives the maximum range of inputs the CORDIC method can handle. The input domain can be increased by a number of methods, some of which are: (1) use identities for the circular functions to reduce the required range of the inputs[60], (2) use floating point normalization procedure[57], or (3) multiple initial CORDIC cycles may be used to increase the input domain (called the precycle approach)[3, pp. 87–94].

The internal convergence requirement ensures that a particular sequence of CORDIC steps reduces z to within $\theta_{n,m}$ of zero. Given that z is within in the convergence domain, i.e. Eq. 4.17 is satisfied, then the magnitude of any successive value for z must be limited by the step size of the convergence, $\theta_{k,m}$. The index, k, is an arbitrary index into the sequence of CORDIC steps. In order to guarantee convergence, every possible k must be considered.

From the above, it is seen that the following relation must hold in order to guarantee convergence during the sequence of CORDIC steps:

$$\theta_{n,m} \geq \theta_{k,m} - \sum_{i=k+1}^{n} \theta_{i,m} \quad k = 0, 1, 2, \ldots, n-1 \tag{4.19}$$

4.3 CORDIC Schedules

The discussion of CORDIC schedules begins with an example, the simplest CORDIC schedule is:

$$\mathcal{S}_i = \begin{cases} i & m = 1 \\ i+1 & \text{else} \end{cases} \tag{4.20}$$

The literature[59, 3, 60] indicates that this baseline CORDIC schedule converges internally for the circular and linear coordinate systems, but not for the hyperbolic coordinate system. A common solution[60, 3, 27, 57] repeats certain CORDIC steps to compensate for insufficient step range. The issue to the circuit designer is when should these so called double cycles be performed.

The CORDIC schedules proposed in the literature[59, 3, 27, 57, 60] are static; that is, the schedule remains the same regardless of the state of the registers. An alternative way to look at it is to say that for static schedules, $\mathcal{S}$ depends only on i. Table 4.2 shows three examples of static schedules found in the literature. Recall that the baseline schedule, Eq. 4.20, is also a static schedule.

m	Schedule	max($\lvert z_0 \rvert$)	Remark
-1	1,2,3,4,4,5,...	1.12	Volder's Schedule[59]
0	1,2,3,4,5,6,...	1.00	For m = -1, repeat the integers
+1	0,1,2,3,4,5,...	1.74	4,13,40,...,k,...,$3k+1$
-1	1,2,3,3,4,4,...	1.25	Haviland's Schedule[27]
0	1,2,3,4,5,6,...	1.00	For m = -1, repeat the integers
+1	0,1,2,3,4,5,...	1.74	3,4,7,12,13,18,19,21
	1,2,2,2,2,3,4,5,	1.84 (m=-1)	Timmermann's Schedule[57]
	6,6,7,8,9,10,11,	1.76 (m= 0)	This schedule is for a 24–bit pipelined
	12,13,13,14,15,	1.71 (m=+1)	implementation. Therefore the CORDIC
	16,17,18,19,20,		schedule is an amalgamation of all
	21,22,23,24		three CORDIC schedules.

Table 4.2: Examples of Static CORDIC Schedules Found in the Literature

As stated earlier, the input domain (Eq. 4.17) can be expanded for static schedules by employing precycles. Consider:

$$|z_0'| \leq \Theta_m + \theta_{i,m}$$

and

$$|z_0'| = |z_0| - k\theta_{0,m}$$

where $|z_0'|$ represents the initial value for the regular CORDIC schedule that immediately follows the precycle stage. Assuming $|z_0| \geq k\theta_{0,m}$ the number of precycles k is:

$$k \geq \left\lceil \frac{|z_0| - (\Theta_m + \theta_{i,m})}{\theta_{0,m}} \right\rceil \tag{4.21}$$

The zero seeking nature of the CORDIC step ensures that the indicator variable remains within $\theta_{0,m}$ of zero. Therefore, convergence is assured even if k is larger than necessary. Since increasing k lowers the effective throughput of the CORDIC neuron, it is to the designer's best interest to keep k as low as possible.

Ahmed[3] presents CORDIC schedules that produce easily reducible scale factors,

and simultaneously enlarge the input domain via precycling. For example[3, p. 93]:

$$\mathcal{S}_{-1} = < 1,1,1,1,2,2,2,3,3,4,4,6,7,8,9,10 >^{3} \tag{4.22}$$

yields $K_{-1} = 1.9987$ and $z_0 \leq 3.37$. The scale factor is close enough to a power of two that it can be easily handled via a bit–shift. In addition, the input domain has grown to 3.37 from the 1.13 for Volder's schedule.

The author proposes an alternative approach; the dynamic CORDIC schedule. Dynamic CORDIC schedules execute CORDIC steps according to the contents of x, y, and z. For example, the following dynamic CORDIC schedule ensures the convergence requirements are met:

$$\mathcal{S}_0 = \begin{cases} 0 & m = 1 \\ 1 & \text{otherwise} \end{cases} \tag{4.23}$$

$$\mathcal{S}_{i+1} = \mathcal{S}_i + \begin{cases} 1 & \text{sign}(z) = \text{sign}(z'), r = 1 \\ 1 & \text{sign}(y) = \text{sign}(y'), r = 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.24}$$

where z', and y' represent the saved value of z, and y for each iteration through the while loop as represented in the CORDIC algorithm (see Fig. 4.1).

This dynamic schedule repeats CORDIC steps until a zero crossing is observed for the indicator variable. This guarantees that the indicator variable is less than $\theta_{i,m}$; hence, the internal *and* external convergence requirements are met simultaneously.

All is not well, however, as this dynamic CORDIC method can not predict the accumulated scale factor, K_m, that is applied to the result of the CORDIC rotations. This is because k_i depends on $\mathcal{S}$ (recall Eq. 4.6). Since $\mathcal{S}$ is not predictable, then it is not possible to know K_m without multiplying k_i throughout the CORDIC rotations (hence n multiplications, not to mention the inherent difficulty in finding k_i).

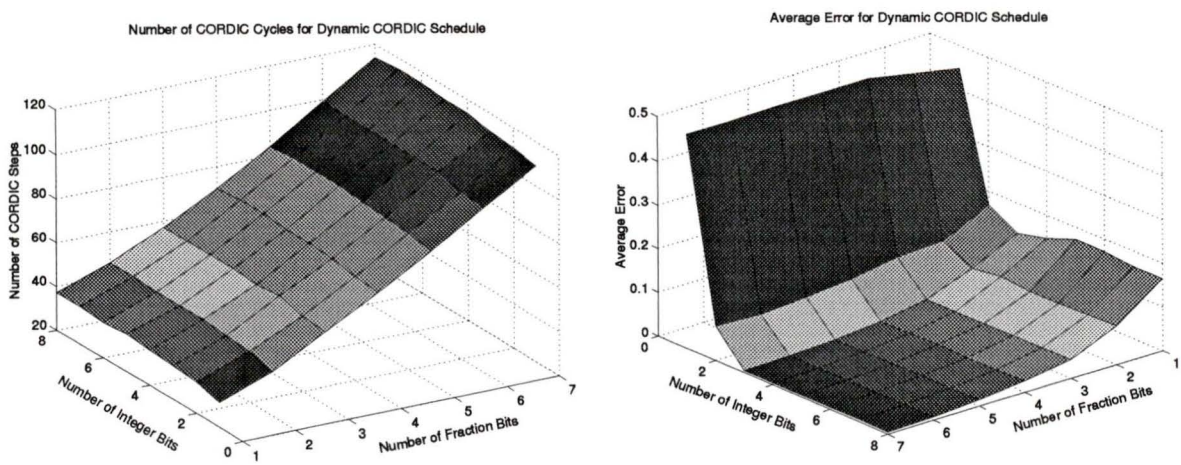
[3]The angle brackets delimit an ordered list of elements, indexed by $i \in \{0, 1, 2, \ldots, 15\}$.

More work is required to consider the dynamic CORDIC schedule for general purpose CORDIC operations. For example, it may be possible to find a dynamic CORDIC schedule, perhaps using the extended CORDIC rotations of Haviland[27], that forces K to some known value that may be dealt with in a similar manner to Ahmed's example. However, the author did not pursue this path because, as will be seen later, the scale factor cancels out for the CORDIC neuron. For general purpose CORDIC processors, however, the scale factor problem with the dynamic CORDIC schedule will need to be solved for this technique to have much value.

A CORDIC neuron, with eight inputs, was implemented in C using the dynamic CORDIC schedule. This neuron was evaluated to determine the average error and the average number of cycles required to convergence. The results are plotted in Fig. 4.2. The variables x, y, and z are represented as fixed point numbers with $\mathcal{I}$ bits for the integer part, and $\mathcal{F}$ bits in the fraction part. Under overflow and underflow conditions, the program saturates variables rather than allowing wrap around. The plots demonstrate the effects of altering $\mathcal{I}$ and $\mathcal{F}$ for simulation runs (1000 samples per simulation run) using values randomly distributed between +4 and -4 for the weights, and +1 and -1 for the neuron inputs.

Figure 4.2(a) shows a fairly linear growth for $\mathcal{F}$, but a much smaller effect for $\mathcal{I}$. This is as expected since the number of CORDIC cycles ought to depend on the expected resolution of the result, and not on the range of the stored values. The general trend for $\mathcal{I} \geq 5$ reveals that each bit of the fraction constitutes approximately 14.162 CORDIC steps for all eight inputs (normalized to about 1.77 steps per bit per input).

Figure 4.2(b) is a bit more interesting as the limits of the CORDIC method are displayed better. The graph exhibits a dramatic drop in error along the $\mathcal{I}$ axis as the number of integer bits grows larger than two. This is due to the inability of the



(a) (b)

Figure 4.2: Simulation Results for Dynamic CORDIC schedule: (a) plot of Average Number of CORDIC steps for Dynamic Schedule, varying the number of integer bits and fraction bits, and (b) plot of Average Error for Dynamic Schedule, varying the number of integer bits and fraction bits.

CORDIC method to function when the magnitude of the registers is less than about 4. However, the CORDIC method benefits little by increasing the number of integer bits beyond three. This observation holds for an uncorrelated input stream, it is quite possible to construct an example in which the registers must be large enough to contain values up to $N2^{\mathcal{I}}$ (where $N = 8$ in our example) in order to perform the calculations exactly. However, it is still noted that such extreme demands on the internal registers are not necessary as long as the input domain is reasonably arranged.

Along the $\mathcal{F}$ axis, the error drops exponentially as expected — each bit of fraction should reduce the error by a factor of about two. In particular, for $\mathcal{I} = 5$ the error is reduced by a factor of 1.88 per bit.

Even with the dynamic CORDIC schedule's advantages in efficiency over the static CORDIC schedule, there are some reasons why the static CORDIC schedule may be

more appropriate:

1. The static CORDIC schedule is predictable. One model of a single layer perceptron is a pool of CORDIC neurons operating in parallel. A static CORDIC schedule allows the designer to construct a single controller to operate the pool of CORDIC neurons simultaneously; effectively, an SIMD computer model. The controller of the dynamic CORDIC schedule can not be globalized, therefore it is not possible to construct such a system. Instead, an MIMD model is used.

2. The evaluation of the static CORDIC schedule is simpler since it is possible to predict throughput and average error without extensive empirical simulation.

   The dynamic CORDIC schedule requires *extensive* empirical testing, preferably on test data for the application at hand, to determine the average number of CORDIC cycles to converge and the average error.

3. The static CORDIC schedule fixes the values for K_m. As stated previously, this is more interesting for generalized CORDIC applications since the scale factors cancel out in our CORDIC neuron.

For these reasons, the implementation of the CORDIC neuron follows the static CORDIC schedule.

4.4 Implementation Strategy

As seen previously, the neuron can be divided into two stages: (1) net function, and the (2) activation function. Because the CORDIC method is used for implementation, the activation function is further divided into two CORDIC operations: (1) a hyperbolic rotation, and (2) a division operation.

4.4.1 Net function

The multiply/accumulate CORDIC operation, $y_n = x_0 z_0 + y_0$, ($r = 1$, $m = 0$) is the backbone for the net function. The neuron bias may be established by simply pre–setting the initial condition of the updated variable, y_i (i.e. $y_0 = \theta$).

The internal convergence requirement is known to hold for the multiply/accumulate CORDIC operation from Section 4.2, however the external convergence must be examined to ensure that the input domain does not exceed the capability of the CORDIC method. From Eq. 4.17 the external convergence requirement is written as:

$$\begin{aligned} |z_o| &\leq \Theta_0 + \theta_{n,0} \\ &\leq \sum_{i=0}^{n} 2^{-(i+1)} + 2^{-(n+1)} \\ &\leq 1 - 2^{-(n+1)} + 2^{-(n+1)} \\ &\leq 1 \end{aligned}$$

The reader will recall that the input to the neuron is always less than one in magnitude. Therefore z_0 can be assigned to the neuron's input, and x_0 to the synaptic weight to deal with the external convergence requirement.

4.4.2 Activation Function

The activation function is calculated by finding the hyperbolic sine and cosine of the net function and then dividing them. They are named the hyperbolic rotation CORDIC operation, and the division CORDIC operation respectively.

Recall that the hyperbolic rotation CORDIC operation ($m = -1$, and $r = 1$) calculates

$$y_n = K_{-1} \left(x_0 \sinh(z_0) + y_0 \cosh(z_0) \right)$$

$$x_n = K_{-1}\left(x_0 \cosh(z_0) + y_0 \sinh(z_0)\right)$$

When $x_0 = 1$, and $y_0 = 0$ they reduce to

$$\begin{aligned} y_n &= K_{-1} \sinh(z_0) \\ x_n &= K_{-1} \cosh(z_0) \end{aligned}$$

which can be followed directly by the division stage. The choice for $x_0 = 1$, and $y_0 = 0$ enables a simple division of y/x to yield the answer; any other choice would have complicated the division process.

Once again, the issue of external convergence must be considered. Table 4.2 displays many limits for z_0, all of them are significantly less than two (as low as 1.12 for Volder's schedule).

Recall that Fig. 4.2 shows that the error drops significantly if the x, y, and z registers are given more than two integer bits. Therefore, the limits imposed by the static schedules in Table 4.2 are simply too low – the input domain of the hyperbolic CORDIC operation must be extended.

From Eq. 4.21 and Table 4.2 and using $|z_0| = 4$, the number of precycles for the static schedules is given as follows:

$$\begin{aligned} k &\geq \left\lceil \frac{|z_0| - \max(z_0)}{\theta_{0,m}} \right\rceil && (4.25) \\ &\geq 6 \text{ [Volder]} && (4.26) \\ &\geq 5 \text{ [Haviland]} && (4.27) \\ &\geq 4 \text{ [Timmermann]} && (4.28) \end{aligned}$$

Once the hyperbolic rotation is complete, the hyperbolic sine is divided by the hyperbolic cosine to yield the hyperbolic tangent function. From Table 4.1, the division

CORDIC operation ($m = 0$, and $r = 0$) calculates:

$$\begin{aligned} x_n &= x_0 \\ z_n &= z_0 + y_0/x_0 \end{aligned}$$

To remove confusion, please note that the index is reset at the beginning of the division CORDIC operation; therefore $x_n^{\text{hyp}} \rightarrow x_0^{\text{div}}$, and $y_n^{\text{hyp}} \rightarrow y_0^{\text{div}}$. For the sake of brevity, the superscripts are dropped with the understanding that the operation under discussion is the division CORDIC operation.

Setting $z_0 = 0$ gives the final result:

$$\begin{aligned} z_n &= [K_{-1}\sinh(z_0^{\text{hyp}})]/[K_{-1}\cosh(z_0^{\text{hyp}})] \\ &= \tanh(z_0^{\text{hyp}}) \\ &= \tanh(u) \end{aligned}$$

where z_0^{hyp} is the input to the hyperbolic rotation operation, or, in neural network parlance, the net function u.

The scale factors, K_{-1}, from the hyperbolic operation cancel out during division; thus greatly simplifying the CORDIC implementation[4]. Much work has been presented in the literature regarding innovative techniques to reduce the time required to compensate for the scale factors[3, 27]; it is reassuring that a simple cancellation improves both accuracy and speed, without complicating the design.

The external convergence requirement for division is adapted from Eq. 4.17:

$$y_0 \leq \sum_{i=0}^{n-1} 2^{-s} x_i + y_n \tag{4.29}$$

[4]The author cautions that the word length must be long enough to encompass both the sinh/cosh and the scale factor. It is noted, however, that the scale factor is typically only 1.17[3], therefore few guard bits are required to handle the scale factor.

$$\leq \sum_{i=1}^{n} 2^{-(i+1)} x_0 + y_n \tag{4.30}$$

$$\leq x_0(1 - 2^{-n} + y_n) \tag{4.31}$$

$$\lim_{n\to\infty} y_0 \leq x_0 \tag{4.32}$$

Therefore, for large n it is required only that $y_0 \leq x_0$, which is guaranteed by the hyperbolic rotation operation. While it is required in a strict sense to introduce a precycle to ensure convergence for moderate n, the author's experience shows that the benefits of the additional CORDIC cycle are minimal[5].

4.4.3 Interface between Net and Activation

The net function outputs to y while the activation function takes its input from z. Unfortunately it is not possible to interface these two CORDIC operations by simply adjusting parameters of the CORDIC operations[6]. Therefore the contents of y must be explicitly moved to z. This can be accomplished through the CORDIC operations themselves (for example, with suitable selection of extraneous variables, the division operation can move y into z). However, that requires a costly full CORDIC operation.

Instead, the data path diagram is enhanced to provide register transfer from from y to z. This requires an extra cycle in the implementation. Therefore, it is prudent to initialize the variables x and y in parallel with the transfer $y \rightarrow z$ to minimize the overhead of initializing the CORDIC unit. The reader may be interested to know that the analysis of the dynamic CORDIC schedule, Fig. 4.2, included the extra CORDIC cycle between the net function calculation and the activation function calculation.

[5]For example, in the worst case $y_n = -2^{-n}$. This gives a range of $y_0 \leq x_0(1 - n^{-n+1})$. For $n = 8$, the maximum range is $y_0 = 0.992x_0$. The effect of x_0 going over this range by up to 0.0078125, is not significant.

[6]Recall the simplification made in the activation function because the outputs of the hyperbolic rotations flowed directly to the inputs of the division operation. This does not occur for the interface between the net function and the activation function.

4.5 CORDIC Engine Architecture

The CORDIC neuron is split into two parts: (1) the CORDIC engine, and the (2) CORDIC engine controller. In terms of previous material, the reader may view (1) as the CORDIC step equations, and (2) as the CORDIC schedule. This section is primarily concerned with the CORDIC engine part.

Regarding the engine, the CORDIC step equations (Eqs. 4.12–4.14, and the support Eqs. 4.15 and 4.16) can be implemented in various configurations, for example: (1) parallel engine unit, (2) serial engine unit, or (3) pipelined engine unit[7].

The parallel CORDIC engine configuration uses an adder plus registers for x, y, and z. This is the most straightforward implementation. The parallel CORDIC was introduced in chapter three; Fig. 3.3 shows a top level block diagram. Walther[60] describes a parallel CORDIC implementation that executes most cordic operations in 70μs, or 14.3 Kilo–COPS (thousand CORDIC Operations Per Second).

The serial CORDIC approach serializes the three adder/subtracters into a single unit, operated serially for the x, y, and z variables. The CORDIC Arithmetic Chip of Haviland and Tuszynski[27], with an effective gate delay of 40μs, operates with the serial CORDIC mode. This gives a performance rating of 25 Kilo–COPS.

Pipelining the parallel CORDIC implementation frees up CORDIC stages to work on successive inputs. The speedup for pipelining without pipeline stalls (i.e. no pipeline hazards)[8] is given as:

$$\mathrm{SU} = \frac{T}{T+\tau} * d \tag{4.33}$$

[7]While it is true that you may apply pipelining techniques to either serial or parallel CORDICs, the driving force is to pipeline the parallel form only.

[8]Occasionally dependencies exist within the pipeline such that a pipe stage must wait until another finishes before continuing. When this happens, all the stages prior to the "stalled" stage must also wait; this is a pipeline stall. The conditions that cause these stalls are called pipeline hazards.

where T is the cycle time for the non–pipelined version, and τ is the overhead due to the extra registers to store state information between successive pipe stages. There is a one to one correspondence between the pipeline and the CORDIC schedule. Therefore, since the pipeline itself is hardwired, the CORDIC schedule must be static. Thus, the pipeline depth d is the number of cycles in the static CORDIC schedule.

The CMOS Floating–Point Vector–Arithmetic Unit of Timmermann, Rix, Hahm, and Hosticka[57] is a 44 stage pipelined CORDIC producing 10 Mega–COPS (million CORDIC function results per second). The pipeline consists of 29 stages for CORDIC functions, and 8 stages for scaling and normalizing.

As nice as pipelined CORDIC processors look from a speedup point of view there are a number of disturbing problems[9]:

1. The CORDIC neuron is actually a string of many CORDIC operations, some of which have different number of required pipe stages. This leads to pipeline stalls when the pipeline depth is smaller than the total static schedule, thus limiting the obtainable speedup to the latency of the slowest CORDIC unit.

2. The latency involved in a pipelined CORDIC neuron is quite large; particularly given that there could be in excess of 64 pipe stages.

3. The speedup of a pipelined CORDIC neuron is only feasible if there is a readily available stream of information to feed the neuron. If data appears sporadically, the speedup gains from the pipeline unit shrink, while the penalties in terms of latency remain the same.

4. Understandably, pipelined CORDIC processors are very large. As will be discussed in more detail later, the author has a single Xilinx 4003APC84-6 Field

[9]It really ought to be mentioned that these problems are not unique to CORDIC. In fact, these issues come up in most discussions of pipeline units.

Programmable Gate Array device at his disposal. Each chip is roughly equivalent to 3000 gates of traditional logic. The pipelined CORDIC is simply not feasible given these limited hardware resources.

4.5.1 CORDIC Engine Datapath

Because of the problems with pipelined CORDIC, the author chose the parallel CORDIC implementation; Fig. 4.3 (the clock line is not shown to reduce visual distractions). The "mu" generator controls the direction of rotation based on the sign bit of each register (the signal paths are also not shown for brevity).

The control signals and data bus names are tabulated in Table 4.3. Roughly speaking, the data lines **theta[]**, **r**, and **i[]** are used for the CORDIC operations, while the remaining data lines perform the housekeeping operations so that the CORDIC operation may begin.

The **Xen** line serves a dualpurpose: it performs housekeeping purposes and also prevents the X register from changing during linear CORDIC operations[10].

4.6 Controller Architecture

The Register Transfer Language (RTL) definition for six CORDIC μ–instructions are shown in Table 4.6, and the corresponding control signal definitions are found in Table 4.5.

The CORDIC neuron is implemented using the CORDIC μ–instructions. For example, Table 4.6 shows the μ–instructions for an implementation following Ahmed's

[10]When $m = 0$, the X register is unchanged as seen in Table 4.1.

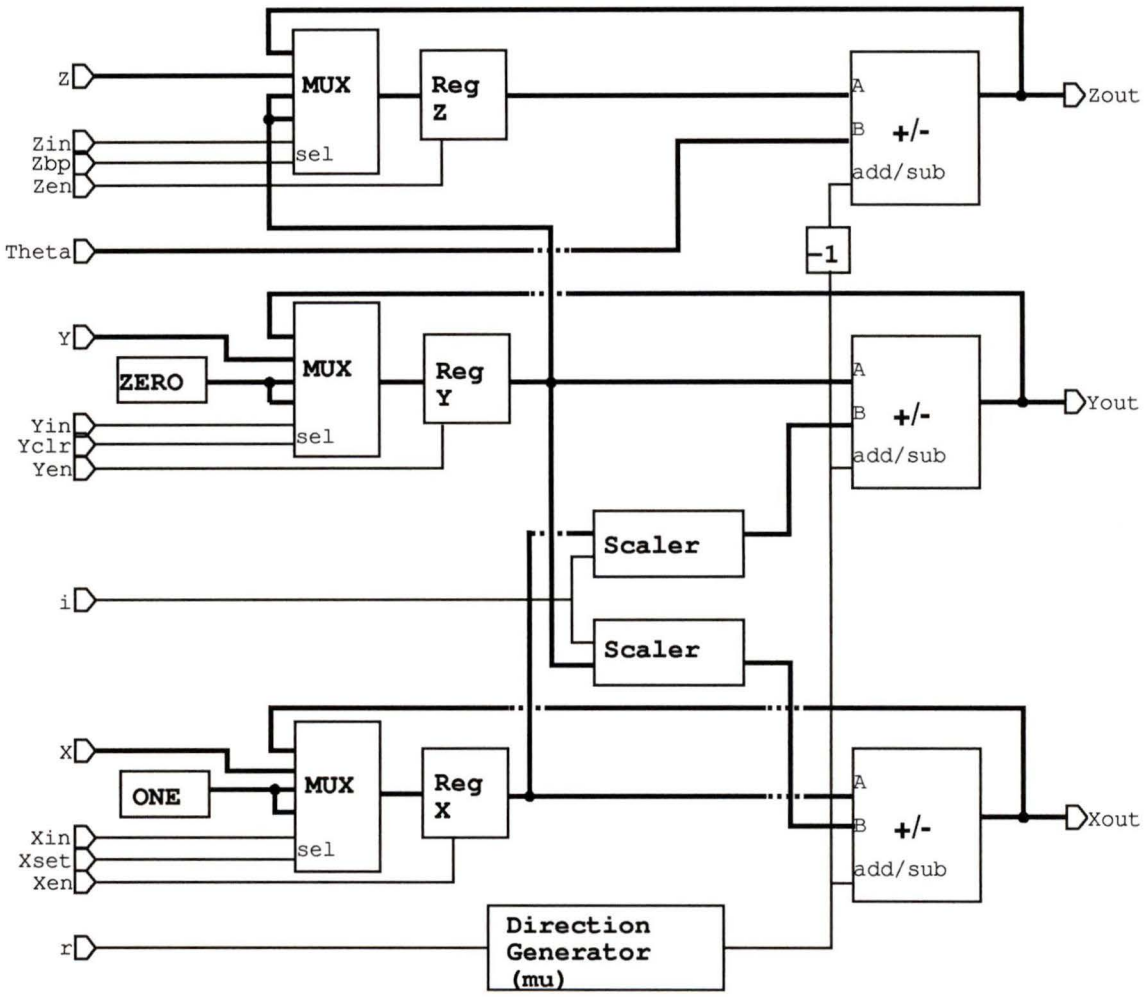


Figure 4.3: Data Flow Diagram for CORDIC Engine

Control Line or Data Bus	Description and/or Comment
X[],Y[], Z[]	Input buses representing x_0, y_0, and z_0.
Xout[],Yout[], Zout[]	Output buses representing x_{i+1}, y_{i+1}, and z_{i+1}.
Xin,Yin, Zin,	When high, registers are loaded the input bus. When low, registers are loaded from feedback path.
Xen,Yen,Zen	Clock enable for registers. Note: Xen can be held low for the duration of a linear CORDIC operation to simulate $x_{i+1} = x_i$.
Zbp	When high, Z is loaded with contents of Y. This control signal overrides Zin.
Yclr	When high, Y is set to 0.0. This control signal overrides Yin.
Xset	When high, X is set to 1.0. This control signal overrides Xin.
theta[]	Data bus for incremental angles.
r	Rotation or Vector Mode.
i[]	CORDIC sequence index.
clk	Clock.

Table 4.3: Control Lines and Data Buses for Dataflow Diagram of the CORDIC Engine

μ–instruction	Register Transfer Language	Comments
Start	X $\leftarrow w$, Y $\leftarrow \theta$, Z $\leftarrow x$	Initialization
Ms	X $\leftarrow w$, Z $\leftarrow x$	Multiplier Setup
Mc	Y $\leftarrow$ Y + sign(Z)2^{-i}X, Z $\leftarrow$ Z - sign(Z)theta[]	Multiplier Cycle
Hs	X $\leftarrow$ 1.0, Y $\leftarrow$ 0.0, Z $\leftarrow$ Y	Hyperbolic Setup
Hc	X $\leftarrow$ X - sign(X)sign(Y)2^{-i}Y, Y $\leftarrow$ Y - sign(X)sign(Y)2^{-i}X, Z $\leftarrow$ Z + sign(X)sign(Y)theta[]	Hyperbolic Cycle
Dc	Y $\leftarrow$ Y - sign(X)sign(Y)2^{-i}X, Z $\leftarrow$ Z + sign(X)sign(Y)theta[]	Division Cycle

Table 4.4: Register Transfer Language Definition for CORDIC μ–Instructions

μ–instruction	Control Signals									
	Xset	Yclr	Zbp	Xen	Yen	Zen	Xin	Yin	Zin	r
Start	0	0	0	1	1	1	1	1	1	-
Ms	0	0	0	1	0	1	1	-	1	-
Mc	0	0	0	0	1	1	-	0	0	1
Hs	1	1	1	1	1	1	0	0	0	-
Hc	0	0	0	1	1	1	0	0	0	1
Dc	0	0	0	0	1	1	-	0	0	0

Table 4.5: Control Signals for CORDIC μ–Instructions Based on the Datapath Diagram of Fig. 4.3, Where '-' is the Don't Care Symbol

example schedule, Eq.4.22. The figure combines the μ–instruction sequence and the CORDIC schedule for readability, but the reader should be warned that the μ–instructions do not include the CORDIC sequence. A separate CORDIC sequence generator is provided to generate this function.

The top level block diagram for the CORDIC engine controller may be found in Fig. 4.4. The reader will notice that there are three potential ROMs: (1) the μProgramme ROM, (2) the $\theta_{i,m}$ ROM, and (3) the programme ROM (not shown) where the instruction register gets its values from.

The sequence of μ–instructions can be fetched and decoded with the regular fetch–decode–execute from RAM paradigm, or, as is the case for Table 4.6, the sequence of μ–instructions can be stored in a programme ROM. If a static CORDIC schedule is also used, then the control circuit can be simplified by combining the instruction register into the μprogramme ROM. See Fig. 4.6 for the reduced controller. The μProgramme ROM can be reduced by using nanoprogramming techniques[29].

If the μ–instruction sequence is fixed, then only a fixed number of inputs can be accumulated in the neuron. This makes the *neuron* static. On the plus side, static neurons do not require synchronizing; the time for a static neuron to calculate its output is always the same, therefore the outputs for an array of static neurons

μ-instruction Sequence	Comments
Start, Mc/i=1, Mc/i=2, Mc/i=3, Mc/i=4, Mc/i=5	x_1, w_1, θ
Ms, Mc/i=1, Mc/i=2, Mc/i=3, Mc/i=4, Mc/i=5	x_2, w_2
Ms, Mc/i=1, Mc/i=2, Mc/i=3, Mc/i=4, Mc/i=5	x_3, w_3
Ms, Mc/i=1, Mc/i=2, Mc/i=3, Mc/i=4, Mc/i=5	x_4, w_4
Ms, Mc/i=1, Mc/i=2, Mc/i=3, Mc/i=4, Mc/i=5	x_5, w_5
Ms, Mc/i=1, Mc/i=2, Mc/i=3, Mc/i=4, Mc/i=5	x_6, w_6
Ms, Mc/i=1, Mc/i=2, Mc/i=3, Mc/i=4, Mc/i=5	x_7, w_7
Ms, Mc/i=1, Mc/i=2, Mc/i=3, Mc/i=4, Mc/i=5	x_8, w_8
Hs, Hc/i=1, Hc/i=1, Hc/i=1, Hc/i=1, Hc/i=2 Hc/i=2, Hs/i=2, Hc/i=3, Hc/i=3, Hc/i=4, Hc/i=4	Hyperbolic
Ds/i=1, Ds/i=2, Ds/i=3, Ds/i=4, Ds/i=5	Division, final result in Z

Table 4.6: Sequence of static CORDIC μ–instructions based on Ahmed's example schedule for the hyperbolic rotation, Eq.4.22, to implement an eight input neuron with five bits of precision.

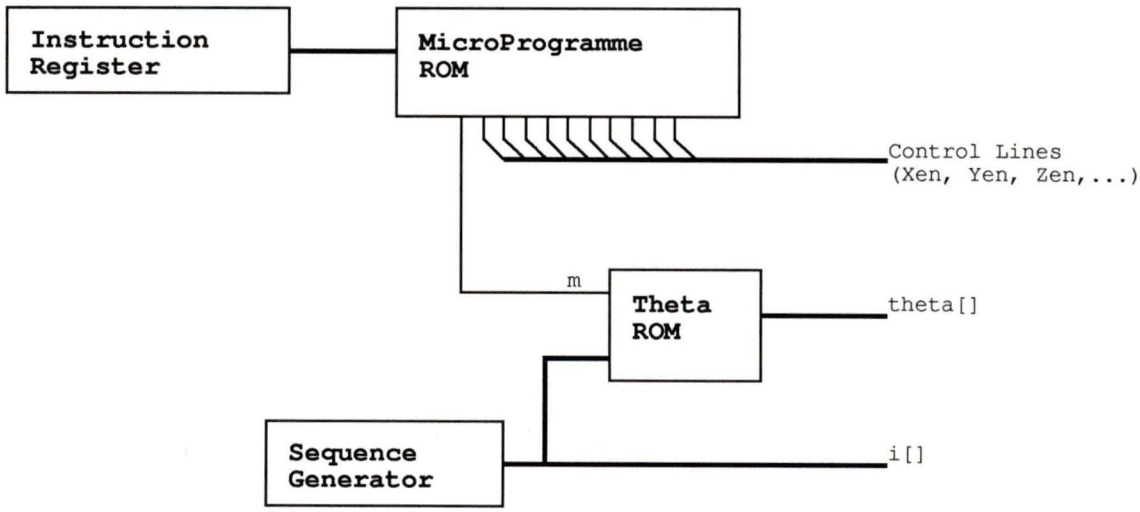


Figure 4.4: The top level block diagram for the CORDIC engine controller circuit. Notice that the sequence generator (CORDIC schedule) and the μProgramme ROM (μ–instructions) are independent.

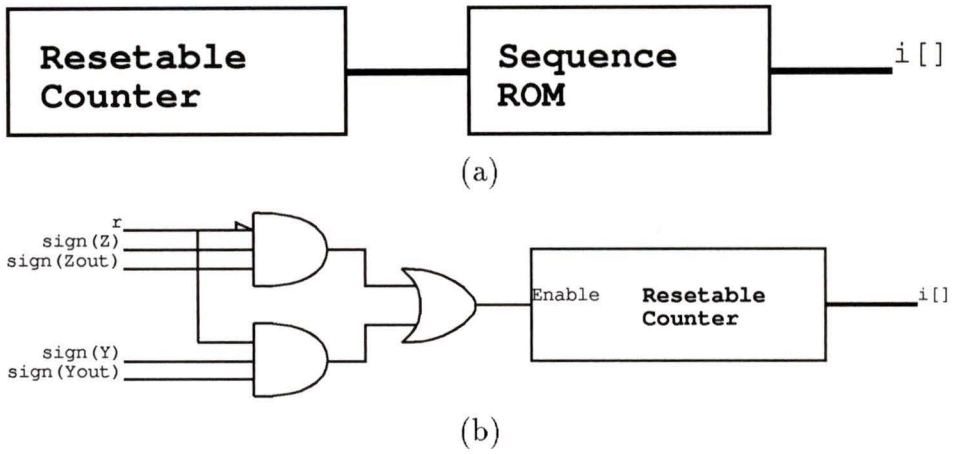


Figure 4.5: Block Diagrams for the CORDIC Sequence Generator: (a) is for static CORDIC schedules, and (b) is for dynamic CORDIC schedules.

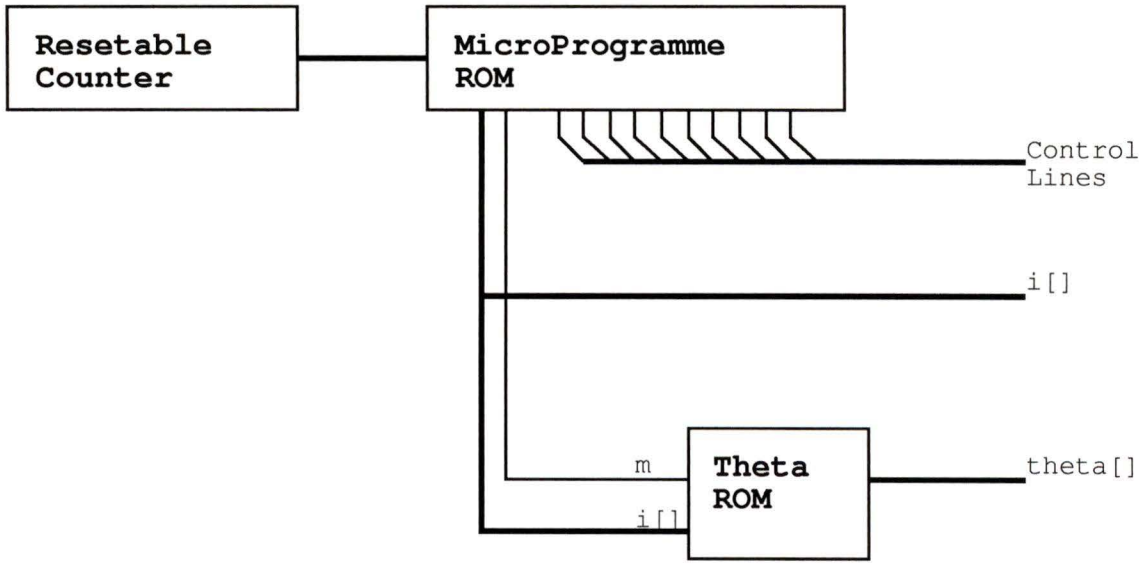


Figure 4.6: Simplified CORDIC Controller with a Single μProgramme ROM for use with Static μ–Instruction Sequences and Static CORDIC Schedules

synchronize their outputs automatically. The negative side is that the size of the weight matrix for a single layer of these static neurons is also fixed. Weight matrices will need to be filled with zeros to reduce the rank of the matrix for smaller neural applications. The ease of synchronizing the outputs may be offset by the efficiency drains by padding the weight matrices.

4.7 CORDIC Neuron Emulator

The optimised static schedule module of Fig. 4.6 is the starting point for the CORDIC implementation; Table 4.7 shows the CORDIC schedule as implemented, based on Table 4.6 with the exception of the the hyperbolic operation. The change to the hyperbolic operation brings the total number of μ–instructions down to 64 from the 65 μ–instructions found in Table 4.6. ROMs must be made in multiples of 32 words for the Xilinx hardware; therefore, the space savings outweighed the efficiency loss due to the non–optimal CORDIC hyperbolic sequence. Notice that there are eight inputs and five bits of precision.

Before the CORDIC neuron was built for the Xilinx hardware, it was implemented in a software emulation environment. The decision to construct a CORDIC neuron emulator was made early in the design process because:

1. At the early stage of a project, a quick and flexible development environment is crucial. Hardware designs are much more difficult to change midstream than software prototypes.

2. It is much easer to debug and test software designs than hardware designs. This is due, in part, to the relative maturity of software debuggers. It is noted that

μ-instruction Sequence	Comments
Start, Mc/i=1, Mc/i=2, Mc/i=3, Mc/i=4, Mc/i=5	x_1, w_1, θ
Ms, Mc/i=1, Mc/i=2, Mc/i=3, Mc/i=4, Mc/i=5	x_2, w_2
Ms, Mc/i=1, Mc/i=2, Mc/i=3, Mc/i=4, Mc/i=5	x_3, w_3
Ms, Mc/i=1, Mc/i=2, Mc/i=3, Mc/i=4, Mc/i=5	x_4, w_4
Ms, Mc/i=1, Mc/i=2, Mc/i=3, Mc/i=4, Mc/i=5	x_5, w_5
Ms, Mc/i=1, Mc/i=2, Mc/i=3, Mc/i=4, Mc/i=5	x_6, w_6
Ms, Mc/i=1, Mc/i=2, Mc/i=3, Mc/i=4, Mc/i=5	x_7, w_7
Ms, Mc/i=1, Mc/i=2, Mc/i=3, Mc/i=4, Mc/i=5	x_8, w_8
Hs, Hc/i=1, Hc/i=1, Hc/i=1, Hc/i=1, Hc/i=2 Hc/i=3, Hs/i=4, Hc/i=5, Hc/i=5, Hc/i=5	Hyperbolic
Ds/i=1, Ds/i=2, Ds/i=3, Ds/i=4, Ds/i=5	Division, final result in Z

Table 4.7: Sequence of Static CORDIC μ–Instructions Similar to Table 4.6 with Modifications to Fit Within 64 Cycles. This change allows the μProgramme ROM to fit within the multiple of 32 address that the Mentor Graphics Tool imposes without undue wastage.

most hardware simulators are much more difficult to use than the standard software debuggers supplied with the compiler.

3. The emulation environment gives designers the capability to examine neural networks that are much larger than the hardware at hand can support.

The CORDIC emulator is written in the C language, with the Xilinx XC4003 FPGA hardware design model in mind. For this reason the Xilinx implementation is little more than a port of the emulator implementation to the Xilinx implementation. The reader will find the C language implementation of the emulator in Appendix A.1

4.7.1 Emulator Results

Figure 4.7 shows the execution profile for the static CORDIC emulator; the output trace file is found in Appendix A.2. The profile is divided into three regions for clarity:

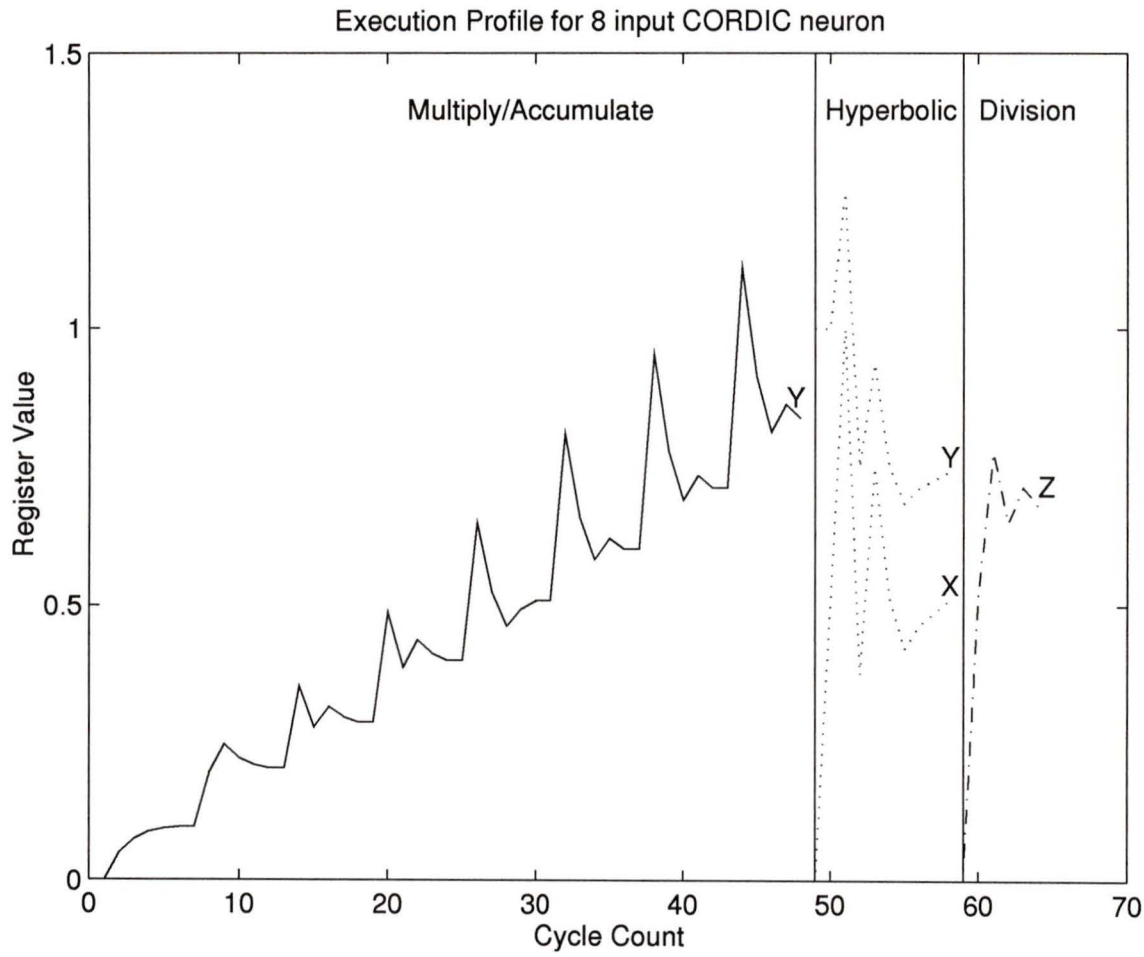


Figure 4.7: Scatter Plot of CORDIC Execution Profile; register contents vs cycle count

(1) the net function, (2) the hyperbolic rotation, and (3) the division operation. The reader can clearly see the iterative nature of the algorithm.

In addition, the emulated neuron was subjected to 1000 simulated runs with pseudo–random inputs. The plot of error vs expected output is shown in Fig. 4.8. The majority of the accumulated errors are within one bit of error of the Xilinx implementation.

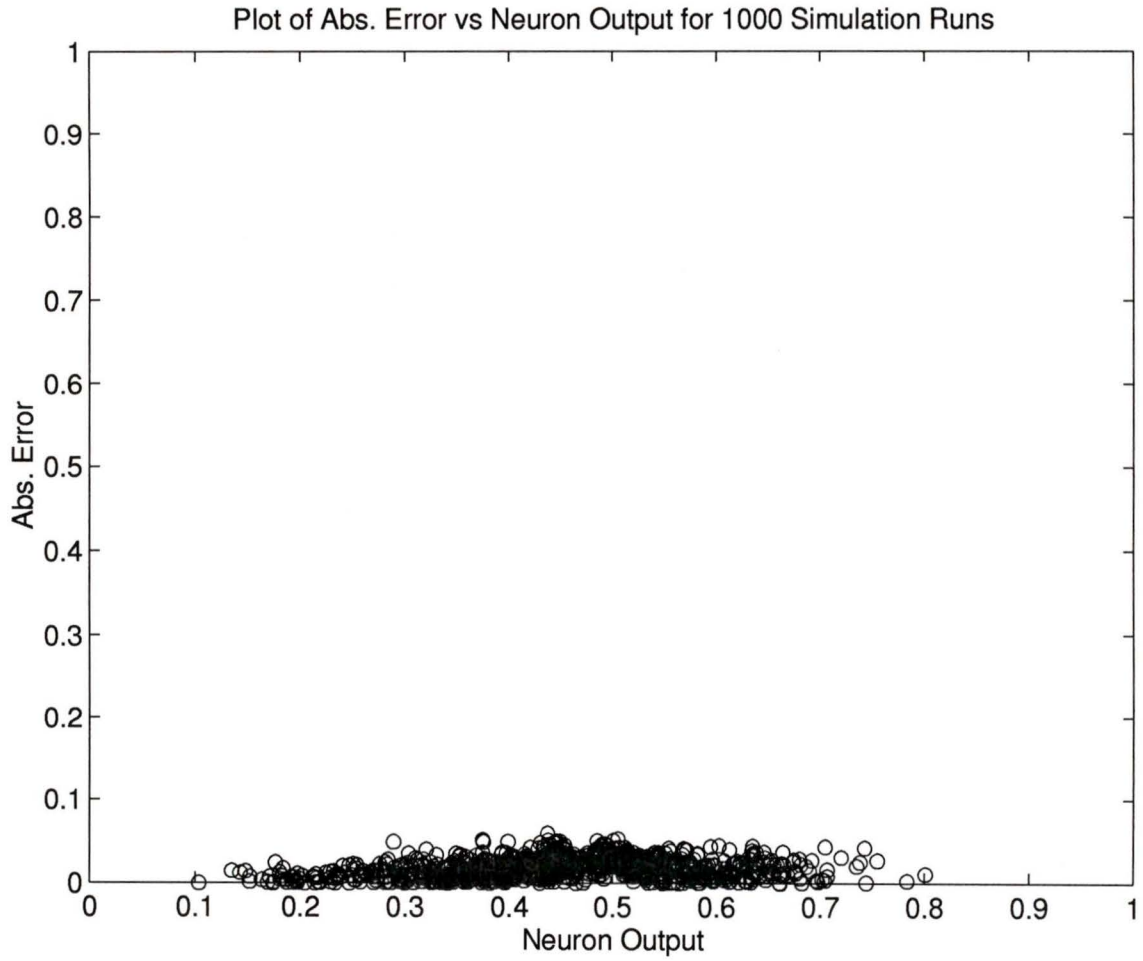


Figure 4.8: Plot of Absolute Error. Each circle represents a simulated run of a neuron with 8 random inputs; the height of the circle represents the absolute error (difference between CORDIC neuron output and the mathematical model), and the distance of the circle from the left edge represents the neuron's expected output.

4.8 Xilinx Hardware Implementation

The basic component of the Xilinx Field Programmable Gate Array (FPGA) architecture is the Configurable Logic Block. Each of the 100 CLB's contains: (1) two logic generators, F and G, each capable of creating arbitrary Boolean functions of four variables; (2) a third logic generator H' implements any Boolean function with the function generators F and G, and the external connection H1 as input variables, (3) two edge triggered flip flops, and (4) various dedicated arithmetic logic circuits for fast carry and borrow circuits[64, pp.2:9–12,2:19]. The CLB cells are arranged in a 10×10 matrix. In addition to the CLBs, the interconnecting wires can also be configured via a network of switch matrices. The result is a very powerful and flexible foundation to perform digital circuit design.

Similar to most digital design processes (e.g. VLSI, and PCB with discrete components) the design process is segregated into a series of procedures to follow. The steps to design a Xilinx chip using the Mentor Graphics CAD tool are (note: steps 2a, 5a, 5b are optional):

1	lca_da	Create design schematics.
2	lca_dve	Create xnf viewpoint (run once per design).
2a	quicksim	Design simulation with unit delay timing model.
3	enwrite	Create edif file.
4	edif2xnf	Convert edif file to xnf file.
5	ppr	Partition, place and route the xnf file.
5a	xdelay	Timing delay analysis.
5b	quicksim	Design simulation with full timing model.
6	xde	Create bit file, and download to Xilinx chip.

Step 1 is the main design phase, created with the schematic capture tool. Design testing with the unit delay timing model can be performed within the Mentor Graphics tool set at this time, however a full Xilinx layout is required to test with full timing simulation turned on. The remainder of the steps, 2 through 6, convert the schematics into a Xilinx MakeBits format compatible with the XC4003AP84-6 chip.

The author designed three versions of the CORDIC neuron (schematics are located in Appendix B.1):

SLP The SLP design embodies an 8 input neuron with five bits of precision, suitable for implementing a Single Layer Perceptron by operating the neuron serially. This neuron is based on the static CORDIC schedule of Table 4.7. External pins are supplied for the neural inputs, biases, weights, and outputs (x_i, θ_i, w_i, and y_i respectively). Due to the vast number of external pins, this version was tested within the Mentor Graphics hardware simulation environment, QuickSim. The Xilinx layout is found in Fig. 4.9.

SLP_TEST This is a special version of the SLP design, where the inputs, $x_{i,j}$ are set to 0.1, and the weights, $w_{i,j}$, are set to 1.0. This corresponds to the neuron operation of Fig. 4.7. This is a test version for downloading to the XC4003AP84-6 Demo Board.

ENGINE This design evaluates the CORDIC engine without the control unit. The I/O ports are wired to common buses to reduce the number of I/O pads. The purpose of this design is to provide some indication of the resources allocated to the CORDIC engine alone.

The report generated from step 5 provides information about the resulting Xilinx layout (e.g. size of the chip, number of I/O pads, and traditional gate array estimates); the full report from the ppr program for the SLP design is provided in Appendix B.2. Step 5a gives the longest delay paths through the design, thereby calculating the maximum clock speed. These two reports are summarized for each design in Table 4.8. A few observations may be made regarding the summary of implementation results:

1. According to the table, the engine requires 62% of the available CLBs to implement the CORDIC engine on the XC4003AP84-6 chip. As seen in the Appendix,

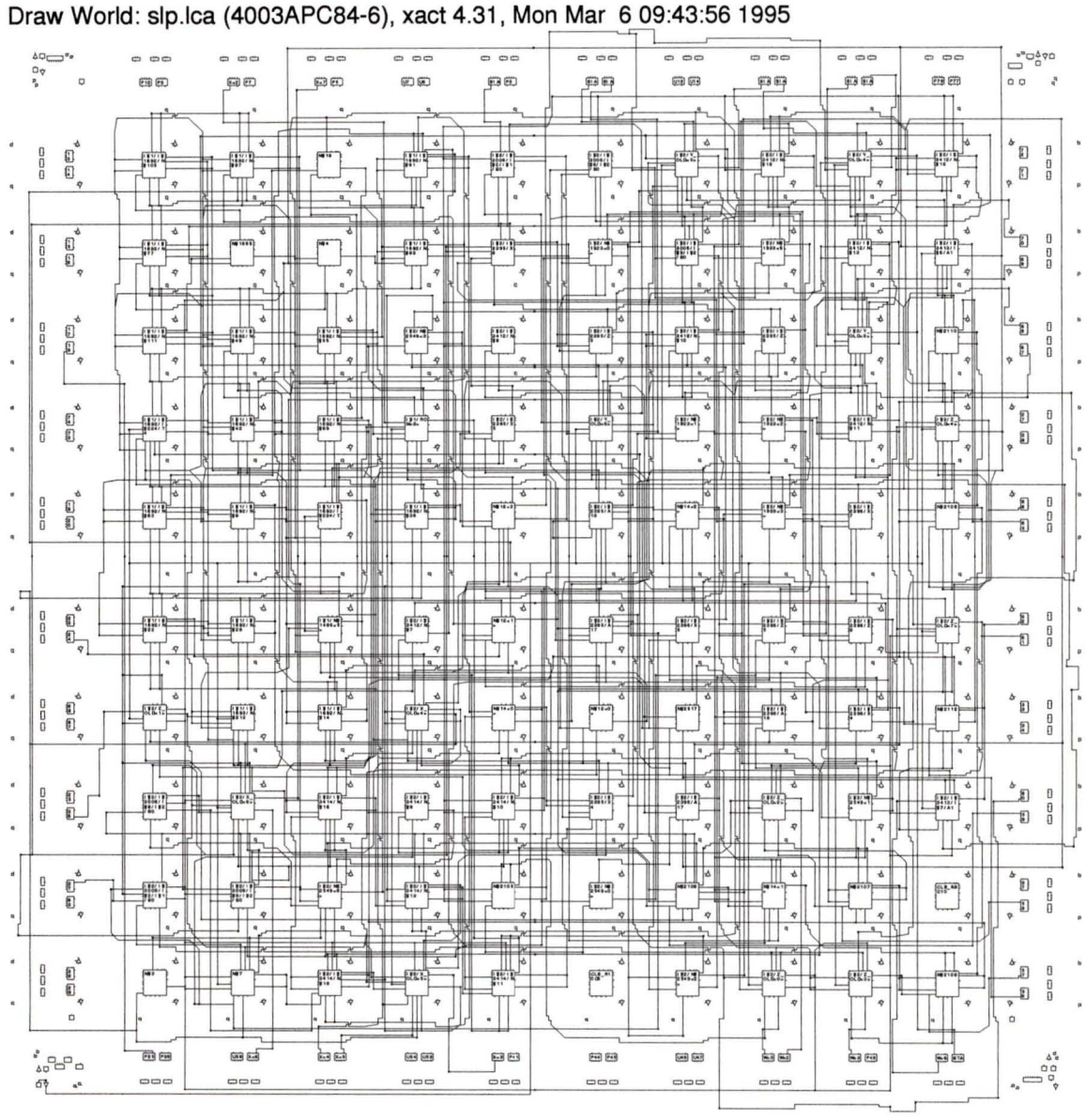


Figure 4.9: Xilinx PPR Layout for the Eight Neuron SLP Design

the schematic for the barrel shifter is quite large, in fact the Xilinx data book[64] reports that 13 CLBs are required for each barrel shifter. The two barrel shifters represent roughly 42% of all the CLBs occupied in the packed Engine design. In CMOS, however, barrel shifters can be constructed using a densely packed array of transmission gates[61, 26].

While Xilinx has many advantages for fast prototyping, and circuit verification, the author feels that CMOS is better suited for the final implementation of the CORDIC neuron because of its higher circuit density, and lower production costs.

2. The propagation delay of the control unit dominates the SLP design; with minor modifications the author feels that the clock speed can be increased beyond 10MHz. Carry look–ahead adders would further increase the clock speed.

3. The design is circuit bound, not I/O bound. For VLSI environments, the area required by the circuit is determined by the rectangle bounding the I/O pads, called the pad frame. In circuit dominated designs, the pad frame barely encloses the circuit core, whereas in I/O dominated circuits there is a large space between the core and the pad frame. Since this design is circuit dominated (illustrated by 98% occupied CLBs, but only 42% occupied I/O pads), it stands to reason that it would represent an efficient VLSI design[11].

[11]The Canadian Microelectronics Corporation (CMC), requires chip designs to contain less than 25% wasted space. Therefore it is important to consider if a design required excessive I/O pads prior to layout.

Category	Max	SLP	SLP_TEST	ENGINE
Logic Symbols		737	737	650
Flip Flops	360	30	30	24
IO Pads	84	35	11	30
Nets		869	822	741
"Gate Array" Gates	3000	2644	2596	1884
Logic		2164	2116	1884
ROM		480	480	0
Occupied CLBs	100	98	94	66
Packed CLBs	100	90	85	62
FG Function Generators	200	180	170	124
H Function Generators	100	49	41	33
Clock Speed		7.9Mhz	7.9Mhz	9.3Mhz

Table 4.8: Summary of Xilinx Implementation Results for SLP, SLP_TEST, and ENGINE Designs.

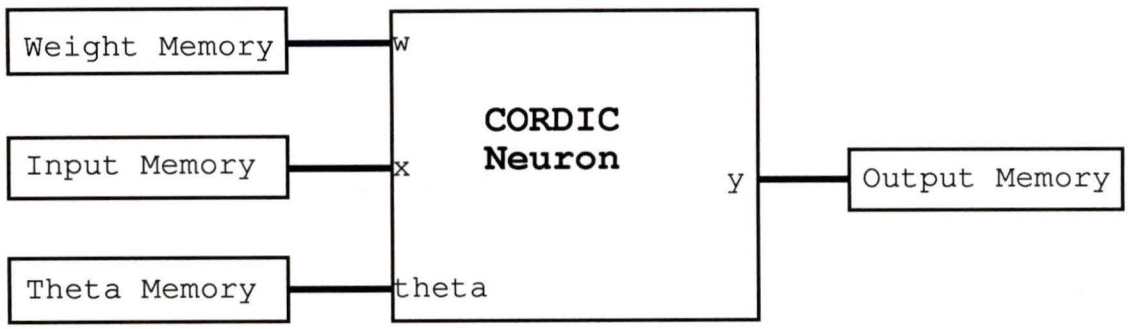
4.9 SLPs from CORDIC Neurons

While the emphasis of this work is the CORDIC neuron, some design for system level neural networks must be presented to demonstrate that neural networks constructed from CORDIC neurons are feasible.

To this end, two approaches to neural network system design are provided: (1) the serial SLP approach, and (2) the semi–parallel SLP approach.

4.9.1 Serial SLP

The serial SLP approach treats the physical CORDIC neuron circuitry as an arithmetic engine for computing the entire single layer of a multi–layer perceptron. As seen in Fig. 4.10, x, w, and θ for each neuron are introduced to the CORDIC neuron unit and the result stored in an output buffer. The buffer is made available to the next layer in the perceptron. Naturally, the layers may also sequentially operate the same




```
Weight Memory = W11, W12, W13, ..., W1n;
                W21, W22, W22, ..., W2n;
                .
                :
                Wn1, Wn2, Wn3, ..., Wnn

Input Memory  = X1, X2, X3, ..., Xn

Output Memory = Y1, Y2, Y3, ..., Yn

Theta Memory  = Theta_w1, Theta_2, Theta_3, ..., Theta_n
```

Figure 4.10: Serial Single Layer Perceptron Module. This implementation of an SLP ought to fit within a single XC4003 Chip.

physical CORDIC neuron, forming a nested sequential loop. The SLP design from Section 4.8 follows this design methodology using off–chip control for the sequencing of individual neurons within the layer.

Some advantages and disadvantages of this method of implementing perceptrons are:

- Slow: this implementation of the multi–layer perceptron is slower than any method with parallelism involved. Given that neural networks inherently involve parallelism of neurons, it is unnecessary to give up that parallelism.

+ Flexibility: the serial intra– and inter–layer design offers the most flexibility at the neuron level. The CORDIC schedule can be constructed to ensure the best possible result.

+ Dynamic CORDIC schedule: with the increased flexibility, the CORDIC neuron is free to employ the dynamic CORDIC schedule.

+ Requires little hardware: the serial intra– and inter–layer design has a very good chance of fitting within a single XC4003 chip. Given that each CORDIC neuron engine requires 62% of the XC4003, it is not possible to construct an entire parallel neural network with one chip.

4.9.2 Semi–Parallel SLP

To improve the performance of the serial SLP, several physical CORDIC neurons may operate in parallel to compute the SLP output vector. Figure 4.11 is an example of such a design. This particular example shows four physical neurons calculating a 64 neuron SLP output vector.

+ Fast (with parallelism)

– Clock skew: as with all SIMD models, the clock will tend to skew. Furthermore, the large capacitance generated from the multiple chip connections for the clock (and input lines) causes an even larger clock skew problem.

– Flexibility: due to the modular nature of the architecture, CORDIC modules can be added in as desired.

– Static CORDIC schedule only: the global control unit enforces a static CORDIC schedule. In order to support a dynamic CORDIC schedule, the control unit

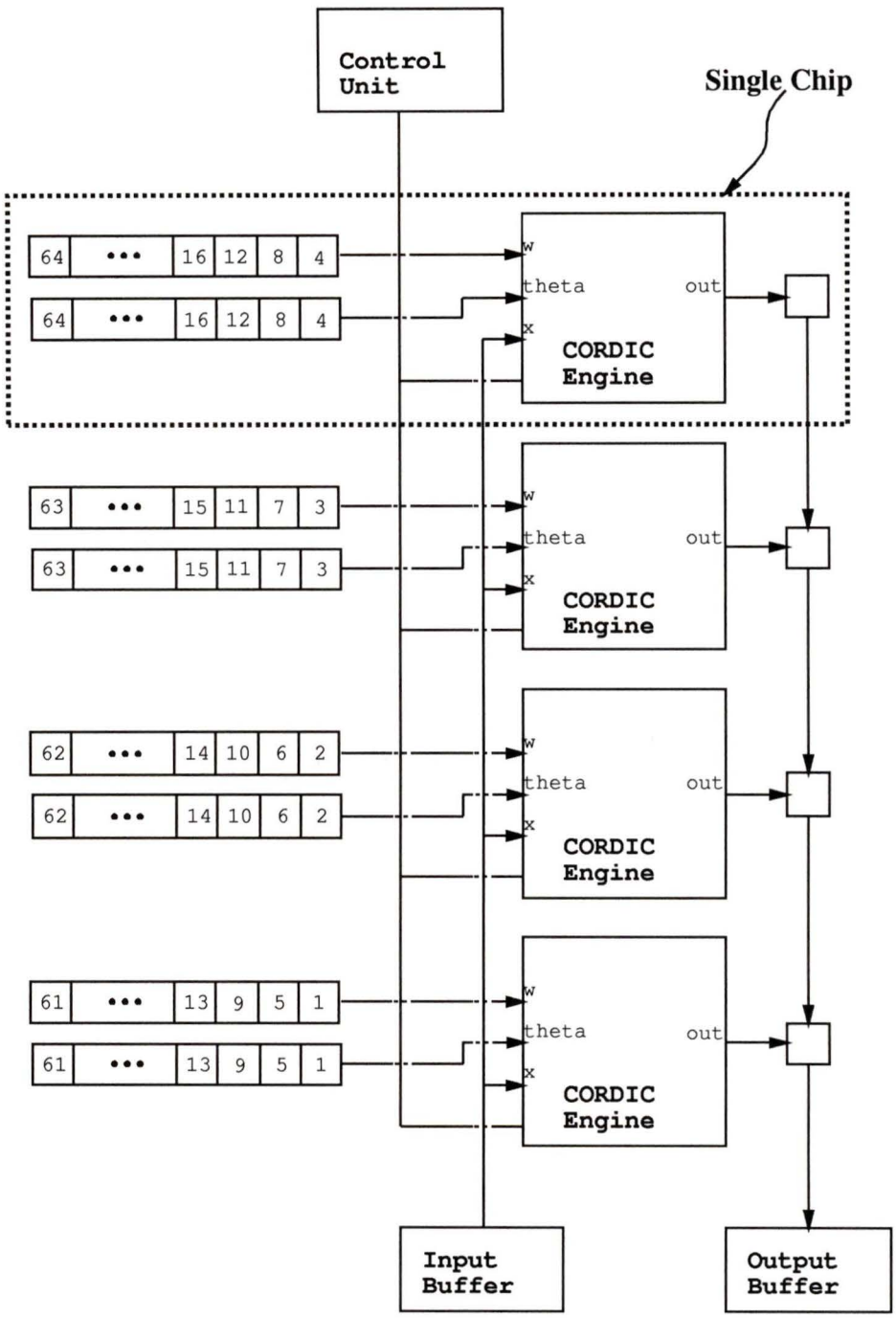


Figure 4.11: Semi–Parallel Single Layer Perceptron. Each CORDIC Engine and its associated weight and theta memories ought to fit in one CMOS VLSI chip. Each block of the weight memory, numbered by the virtual neuron it belongs to, represents a 64 element vector. Each block, again numbered by the virtual neuron to which it belongs, of the θ memory represents a single neuron bias.

must be replicated amongst all the CORDIC engines. Even then, a signaling scheme must be constructed to indicate when all the CORDIC engines have completed.

- Requires lots of hardware: for example, Fig. 4.11 requires perhaps five XC4003 chips for just four CORDIC engines. Much more hardware is required for a 64 neuron SLP.

- More complicated to design and test.

4.10 Evaluation of Neural Networks Built with CORDIC Neurons

As a demonstration of the CORDIC neuron's capability to solve problems, two designs are presented below: (1) the first design is a fairly simple character recognition problem involving three characters, 'A', 'L', and 'C', and (2) a more complicated example where 204 seismic data samples are classified by the neural net into one of five classes of seismic activity.

It really ought to be stressed that the author does not consider these examples to be anything but simple demonstrations of the CORDIC neural network. The merit in these exercises is not the neural network solutions, but rather the demonstration that the CORDIC neuron, as has been described, can function as a neural network.

4.10.1 'A', 'C', and 'L' Character Recognition

This simple example demonstrates that even the eight neuron SLP described in Section 4.8 is capable of performing simple character recognition tasks. However, the

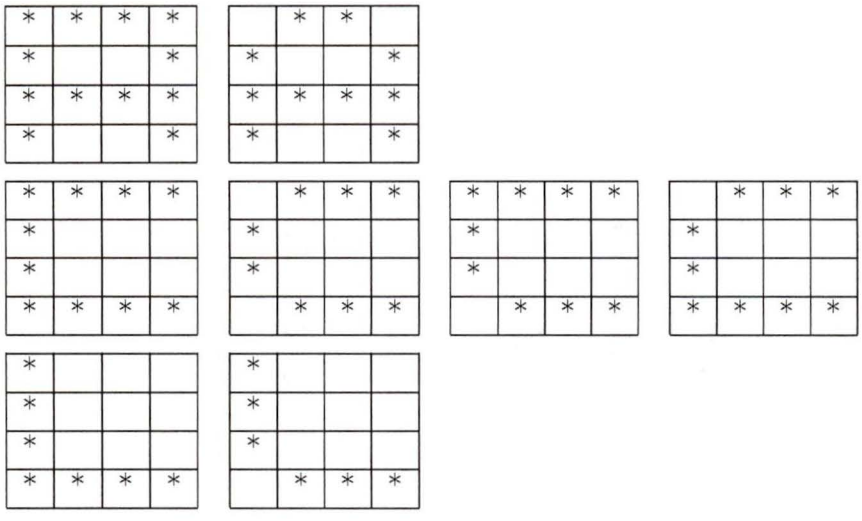


Table 4.9: Training set for the (8 Input, 8 Hidden, 3 Output) Neural Network Classifier

analysis was performed using the eight neuron static CORDIC neuron emulator because it is much easier to gather results using the emulator rather than the hardware simulator or the hardware itself. The training set, shown in Table 4.9, consists of eight exemplars describing the three classes.

The feature extraction phase converts these exemplars into an eight element vector. A simple algorithm used during the 80s for character recognition finds the average "bit density" for each row and column, and write them down starting from the top right working clockwise to the bottom left. For example, the first exemplar of 'A' is:

$$< 1.0, 0.5, 1.0, 0.5, 1.0, 0.5, 0.5, 1.0 >$$

The two–layer neural network to solve this problem is shown in Fig. 4.12. This network is trained via the SNNS simulator. The resulting weight matrices and neuron bias vectors are shown below; the weight matrix and neuron bias vector for the hidden layer are W_h and θ_h, and W_o and θ_o reflect the output layer. Please note that the W_o matrix contains many zeros because the hardware implementation described in

Section 4.8 expects eight inputs for each SLP[12].

$$W_h = \begin{bmatrix} -1.180 & 0.325 & -0.833 & -0.094 & -1.146 & 0.032 & 0.354 & -0.047 \\ -0.612 & -0.313 & -0.910 & -0.529 & -0.619 & -0.845 & -0.217 & 0.405 \\ 0.094 & 0.399 & -0.066 & -1.229 & -0.155 & -0.263 & -1.216 & 0.971 \\ 1.453 & -0.230 & -2.524 & 0.905 & -0.471 & -0.076 & 1.439 & -1.003 \\ 0.116 & -0.014 & -1.818 & 0.049 & -0.101 & -0.224 & -0.175 & -0.123 \\ -0.133 & -0.997 & 0.630 & 0.638 & 0.334 & -0.715 & -0.038 & 0.252 \\ 0.942 & -0.861 & 0.739 & 0.947 & 0.267 & 0.746 & -0.722 & 0.065 \\ 1.118 & 0.813 & 0.813 & 0.373 & 0.584 & 1.196 & 1.200 & -0.485 \end{bmatrix}$$

$$\theta_h = \begin{bmatrix} 0.048 & -0.053 & 0.809 & -0.226 & 1.426 & 0.576 & 0.685 & -1.645 \end{bmatrix}$$

$$W_o = \begin{bmatrix} 0.214 & -0.057 & 0.062 & -0.841 & -1.184 & -0.141 & 0.466 & 0.830 \\ -0.602 & -1.178 & -0.489 & 1.413 & 0.442 & 0.074 & -0.425 & 0.758 \\ 0.303 & -0.005 & 0.489 & -0.415 & 0.572 & 0.100 & 0.839 & -1.907 \\ 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \end{bmatrix}$$

$$\theta_h = \begin{bmatrix} 0.971 & -0.325 & 0.446 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \end{bmatrix}$$

The following test is performed to evaluate the neural network:

1. The feature vector for an exemplar (in this case 'L') is taken from the training

[12]The static μ–instruction sequence causes this effect. There is nothing that prevents the construction of a dynamic μ–instruction CORDIC neuron. This latter design would allow for weight matrices of varying sizes.

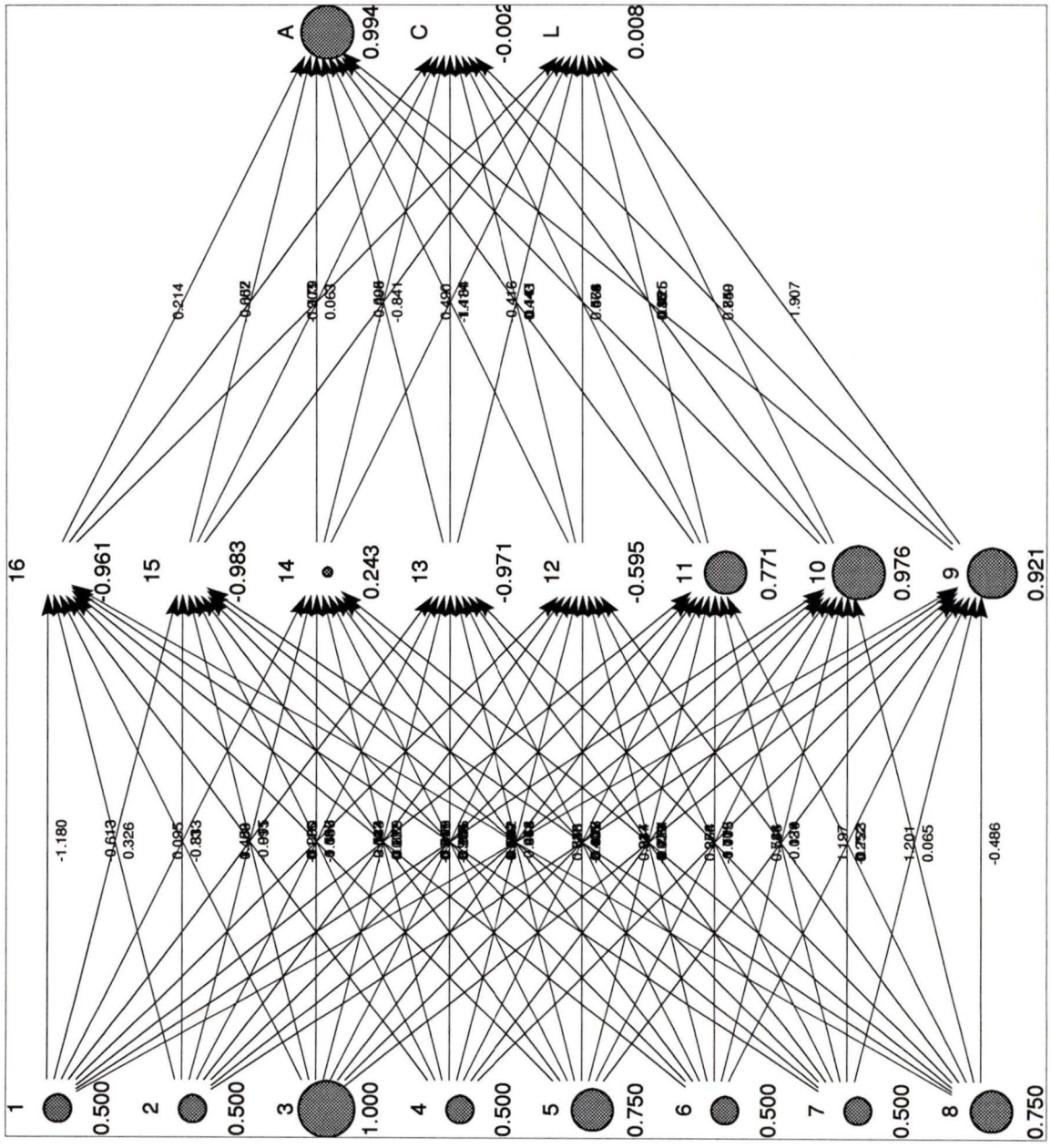
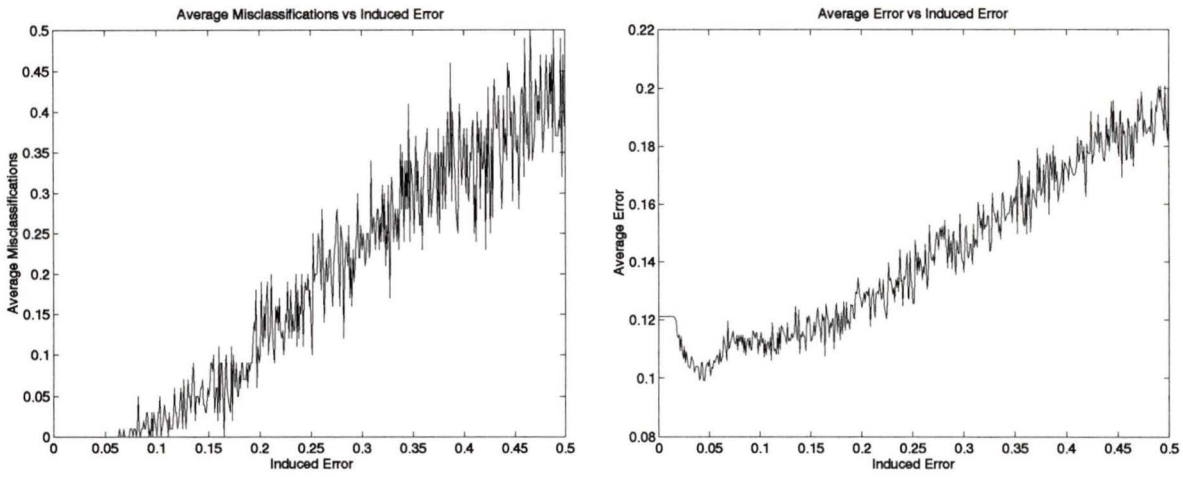


Figure 4.12: The 'A', 'C', 'L' Character Recognition Neural Network.



(a) (b)

Figure 4.13: Results from the Simple Three Letter Character Recognition Example: (a) percentage of misclassification vs induced noise, and (b) average difference in output between theory and CORDIC implementation vs induced noise. Data points are averaged over 100 simulation runs for each level of induced error.

set. Each element of the vector is perturbed by an induced error. The induced error is a random number scaled within a known range.

2. For each known range of error, 100 simulation runs are performed.

3. After each simulation run, a count of misclassifications and the distance of the output from the desired output are maintained.

4. At the end of the 100 simulation runs, the average misclassifications and absolute error are stored in a file indexed by the error induced.

5. The induced error traverses the range [0, 0.5], stepped by 0.001.

From Fig. 4.13(b), notice the drop of about 0.02 in output error for small induced errors (for inputs of about 0.02 to about 0.16). The author believes that this is caused by a signal dithering phenomena[55] resulting from averaging over multiple runs, and the bit truncation used in the emulator. Note, however, that the plot of

misclassifications, Fig. 4.13(a), does not exhibit the dithering effect. Instead, the average misclassifications rise fairly linearly with induced error.

4.10.2 Seismic Data Classifier

In this example, a 1–of–5 decoder categorizes seismic events. In particular, the training set is made up of 204 events divided into five categories: regional events, local events, teleseismic events, mine blasts, and non–seismic events.

The task of the neural network is limited to correctly classifying the training set. The related task of classifying unknown seismic events is more difficult, and will not be attempted for this demonstration.

Since this example is more complicated than the 'A', 'L', and 'C' classifier above, the steps to build a CORDIC neural network capable of classifying these 204 seismic events are:

1. Feature extraction. This stage brings the vast number of data points representing each seismic event into a reasonable number of inputs for the feed forward neural network to deal with.

2. Build and Train Neural Network. As discussed in Chapter 1, this work is carried out with a neural network simulator to quickly develop a network both large enough to perform the required tasks, but also small enough to be practical.

3. Build a CORDIC Neural Network to match the SNNS network.

4. Evaluate the CORDIC Neural Network.

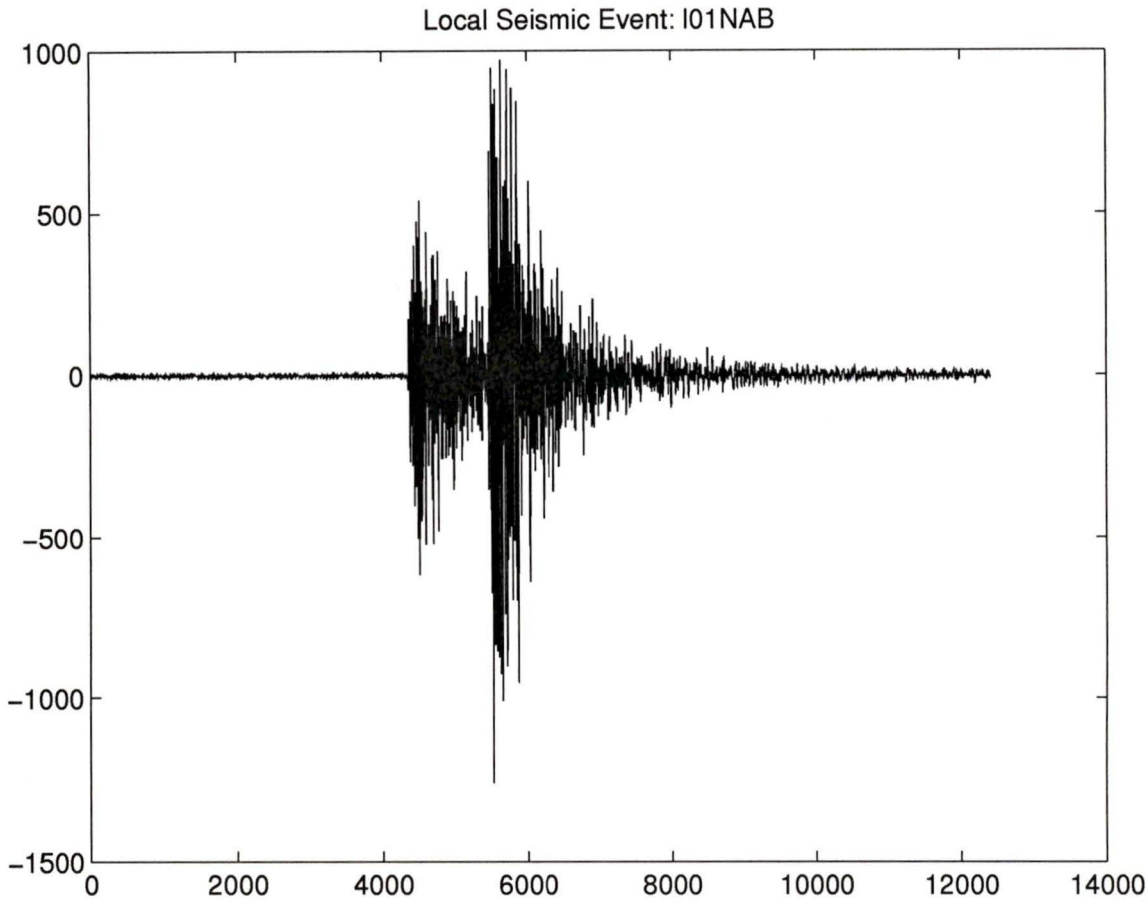


Figure 4.14: Time–Series Plot of a Seismic Event

Feature Extraction

Each seismic event is a set of data samples taken at a regular, but unknown, interval, often exceeding 10,000 samples in a single seismic event. Figure 4.14 shows a time–series plot for one of these seismic events. Given the large size of the initial feature vector, it is not reasonable to provide the neural network access to the raw data.

To reduce the rank of the raw input, the author calculated 16 linear prediction coefficients (LPC)[56]. The author had prior knowledge that LPC features are a good

indicator for seismic data[13]. Without that prior knowledge, the author would need to construct a suitable feature extraction process. The feature extraction phase offers an incidental bonus, the feature vector is now 16 elements long. This fits better into the fixed input domain paradigm of the multi–layer neural network approach.

The author would also like to note that unlike the prior example where the inputs were effectively quantised (1.0, 0.75, 0.25, or 0.0), the inputs in this example are quantised by the word length. Therefore small errors in the inputs of the neural network have larger effects on the output.

Building and Training Neural Network

The first task is to determine the neural network topology. The number of inputs and outputs are given by the problem statement; the boundaries between the neural network and the outside systems are distinguished in the block diagram, Fig. 4.15. The number of inputs is 16, one for each linear predictive coefficient, and the number of outputs is five, one for each class.

The neural network designer decides on the number of hidden layers and how many neurons are in each layer. Designing neural networks by hand is an art, not a science: there are no hard and fast rules for building neural networks. Trial and error, experience, and luck are the tools neural network designers rely on most. For machine generated networks, the network growth algorithms (cascade correlation for example[38, pp. 183–184]) offer some promise for the future, but require vast amounts of CPU time to repeatedly construct and evaluate different neural network topologies.

The author used the Stuttgart Neural Network Simulator to build and train the neural network shown in Fig. 4.16. The network was built with the "trial and error,

[13]Linear prediction coefficients are used extensively to model time–series data[56]. It is a straightforward exercise to apply LPC feature extraction to seismic data.

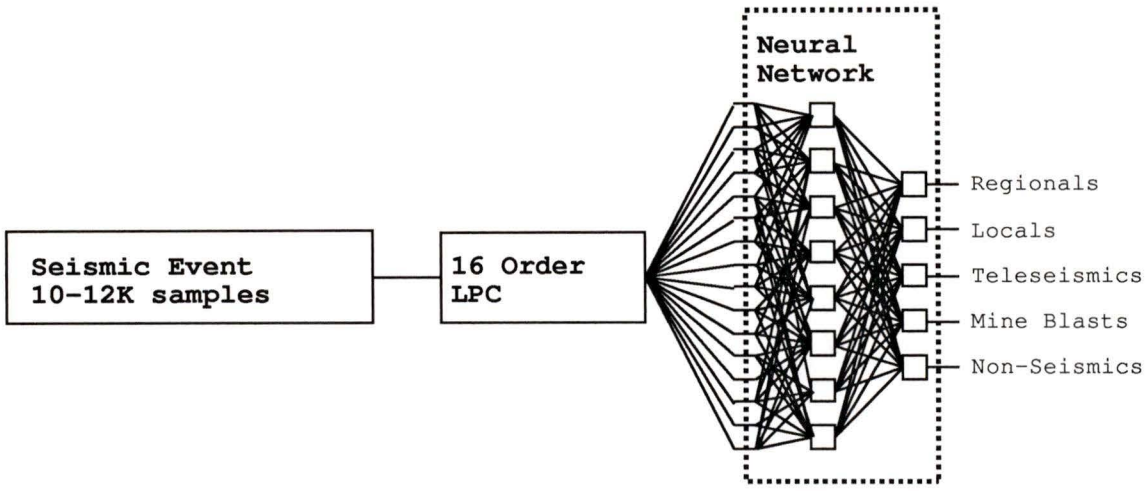


Figure 4.15: Block Diagram of a Seismic Event Classifier Based on a Neural Network One–of–Five Classifier

Percent Correct	95.6%
Percent Wrong	1.5%
Percent Unknown	2.9%
No. of Training Steps	≈15000

Table 4.10: Results of Training the Seismic Event Classifier with SNNS. These performance figures relate to the training data set only; the performance for arbitrary seismic events would naturally be worse than reported here.

experience, and luck" approach to designing networks. Some effort was made to limit the size of the layers, 16 neurons per layer, so that the network would fit the hardware environment better.

After the network topology is determined, the network must learn the training set. The author required that the neural network learn the training set well enough to correctly classify 95% of the 204 seismic events. Table 4.10 shows the results of the training stage.

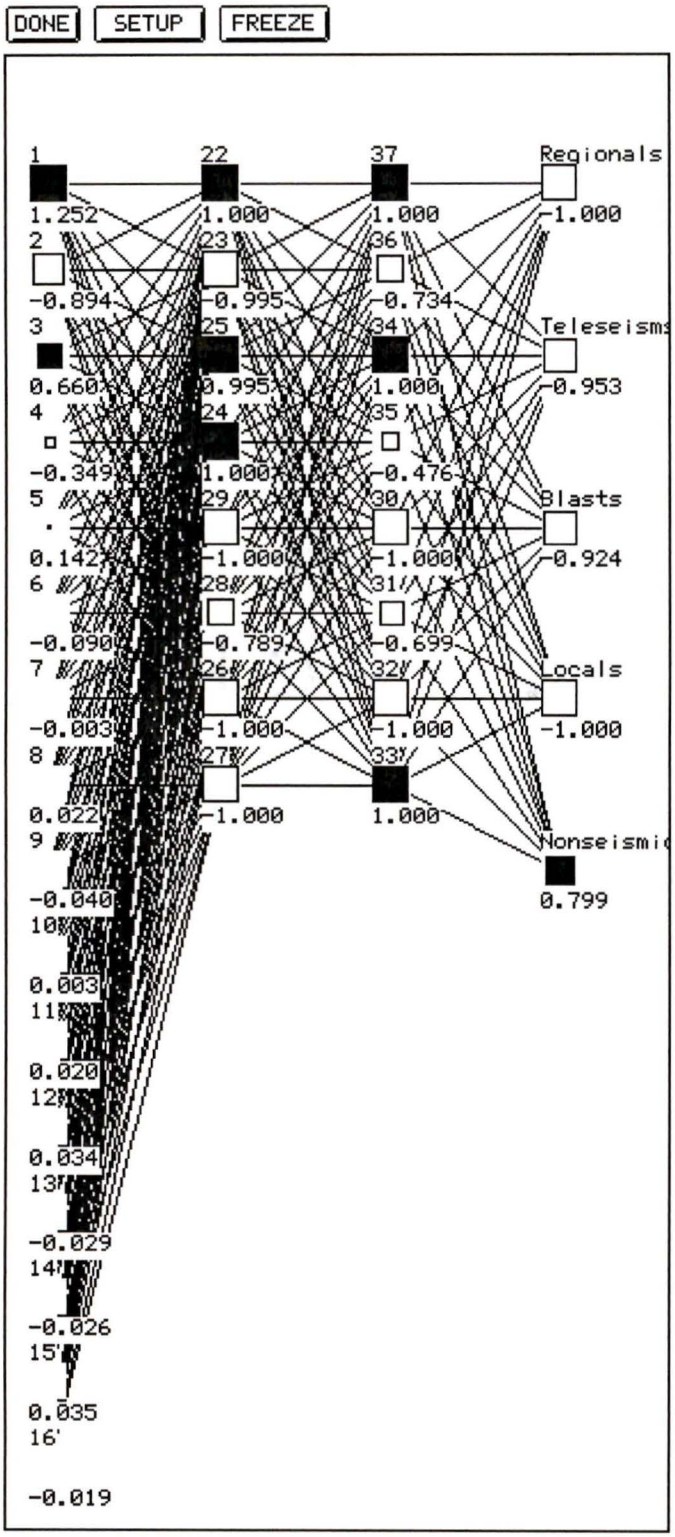


Figure 4.16: The Seismic Event Classifier Neural Network.

Area	162 CLBs
No. CORDIC cycles/SLP	112
Throughput	8.3036×10^4 Classifications/s
Latency	48.17μs

Table 4.11: Performance Estimates for Seismic Event Classifier. The underlying assumptions are: (1) 16 neuron serial SLPs similar to the SLP design in Section 4.8, (2) three separate SLP chips connected together in a pipeline, (3) performance figures from Table 4.11 are used as a basis, and (4) inter chip communication delays are ignored.

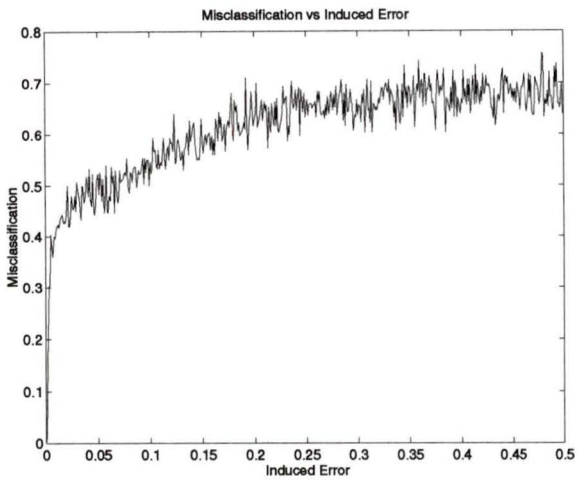
Build CORDIC Neural Network

As in the simple three character recognition example, the results for this example were gathered via the CORDIC neuron emulator. Three 16 neuron SLPs are cascaded to form the neural network. Since the second layer and output layer have fewer than 16 neurons, the weight matrices for these layers are padded with zeros. The performance of the neural network, ignoring the controller, is estimated from the ENGINE design results of Section 4.8. The performance results are shown in Table 4.11.

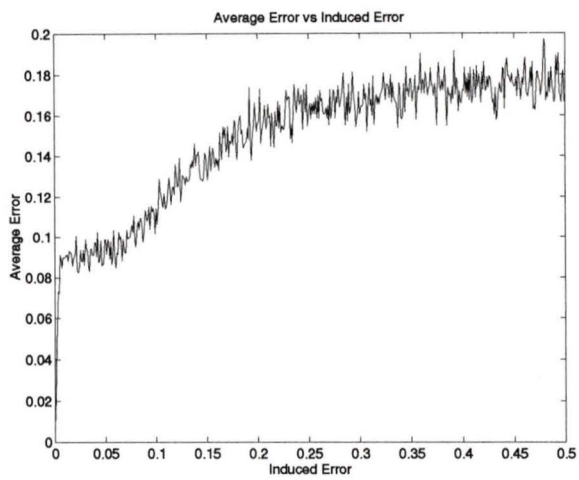
Evaluation of the CORDIC Neural Network

As in the previous example, two figures of merit are considered: (1) the tendency to misclassify a known exemplar under varying degrees of induced error, and (2) the difference between the desired output and the actual output of the neural network under varying degrees of induced error. These two merits are plotted in Fig. 4.17.

As expected, Fig. 4.17 indicates that the seismic classifier is much more sensitive to input errors than the three letter classifier. In fact, Fig. 4.17(a) shows that an induced error as small as 0.01 retards the classification rate to approximately 40% correct classification. The curve appears to level off at about 40% correct classification. These performance figures, however, are not specific to the CORDIC neuron. The faults in



(a)



(b)

Figure 4.17: Results from the Seismic Event Classifier Example: (a) misclassification vs induced noise, and (b) average difference between desired output and CORDIC implementation output vs induced noise. Data points are averaged over 25 simulation runs for each level of induced error.

this neural network design actually lie with the choice of feature extraction techniques, the network topology, and the training methodology. Some specific reasons for this poor network results are: (1) the neural network may have been overtrained, (2) there may not be enough neurons in the hidden layers to generalize well, and (3) the feature extraction procedure may not spread the feature clusters throughout the feature space effectively.

Regardless of the classification performance, it is clearly indicated that the CORDIC neuron is suitable for constructing real neural networks.

4.11 Summary

The CORDIC step was presented early in the chapter, immediately followed by the convergence requirements. The convergence requirements are used to evaluate

CORDIC schedules: (1) the external convergence requirement gives the input domain for a given CORDIC schedule, and (2) the internal convergence requirement evaluates the CORDIC schedule to ensure that it converges when within the input domain.

The CORDIC schedule concept describes the sequence of CORDIC steps required to perform a particular CORDIC operation. Two schedules are examined: (1) the dynamic CORDIC schedule, and (2) the static CORDIC schedule.

The implementation strategy section relates the steps required to build a neuron with the set of operations available with the CORDIC method. The CORDIC engine description with the implementation strategy provides the tools to build a CORDIC neuron. This work supplies two implementations: (1) an emulator, and (2) a Xilinx test implementation.

Since the CORDIC neuron is a building block element, two examples are also provided to show that the CORDIC neuron can construct real neural networks: (1) a three letter character recognition classifier, and (2) a seismic event classifier.

Chapter 5

Summary and Future Work

5.1 Summary

From the title of the thesis, the reader should expect a derivation and a justification for CORDIC neurons. For the first part of the title: the derivation of the artificial neuron is found in Chapter 1, while the derivation of the CORDIC method (and neuron) is found in Chapter 4. The second part of the title reflects the work in Chapter 3; the a priori evaluation of digital neurons.

The following is a summary for each chapter:

1. This chapter examines the current neural network theory. The theory helps the reader understand how the neural network's weight matrices are calculated in the examples of Chapter 4.

2. Various forms of electronic implementations of neural networks are considered. Of particular interest are the digital designs: CNAPS, SYNAPSE–1, MIND–1024, and BNN. The author determines that there are few dedicated digital

neural network designs in the literature. Of the four digital designs considered, only the BNN might be considered a dedicated design. Even so, the limitations of the BNN are so severe that other solutions are sought.

3. The focus of this chapter is to evaluate dedicated digital neural network designs. In order to do so, an extended merit function is derived that accounts for the inherent errors produced in convergent algorithms. With this tool, the author examines three net function designs (Baugh–Wooley, CORDIC, and SPIPP), and five activation function designs (hard limiter, ROM, CORDIC, Chebyshev series approximation, and the three–segment).

   The merit functions for all 15 combinations of activation function with net function are computed and graphed. The result of this analysis is that the CORDIC–CORDIC design has the least cost.

4. The chapter starts with a derivation of the CORDIC method, following Walther's example, from basic principles of rotation matrices. The connection between guaranteeing convergence and the CORDIC schedule is also considered. Furthermore, two classes of CORDIC schedule are given: (1) static CORDIC schedules, and (2) the dynamic CORDIC schedule.

   The implementation strategy showcases the modular nature of the CORDIC method; the results of one CORDIC operations flow into the next CORDIC operation. The connection between the CORDIC schedule and the controller architecture is given some consideration; RTL, and μ–instruction signal tables are constructed to show one method of constructing a CORDIC engine controller.

   Two implementations are presented: (1) a static CORDIC emulator, and (2) a static CORDIC Xilinx chip. The speed and area performance results are

obtained from hardware simulations of the Xilinx chip, while the algorithmic evaluations are supplied by the emulator.

For demonstration purposes, two examples are given: (1) a simple three letter character recognition problem, and (2) a seismic event classifier.

5.2 Future Work

The author also considers a list of areas to expand on this work:

- Create a neuro–computer based on the CORDIC neuron.
- The CORDIC neurons can be pipelined to improve performance. It is not obvious, at this point, what difficulty exist for pipelining non static μ–instruction sequences. One would naturally expect the static CORDIC schedule, and static μ–instruction case to be very easy to pipeline, however a neuro–computer should be flexible to be usable.
- DSP applications in a multiprocessor model. There is a similarity between the algorithms used for DSP applications and neural networks. Some research is needed to determine the maximal overlap between the hardware requirements for these two disciplines so that the CORDIC multiprocessor computer can have the largest group of users.
- Examine dynamic CORDIC schedules and scale factor incompatibility. The dynamic schedule, as given in the text, suffers the tragic flaw that it does not handle the scale factors well. This prevents generalized CORDIC implementations from using it.

- Consider alternative activation functions. The sigmoid function $(1 + e^{-u})^{-1}$ is an alternative to the hyperbolic tangent function. The main difference is that the sigmoid function squashes to between 0 and 1, while the hyperbolic tangent function squashes to between -1 and +1.

- Back propagation algorithm in hardware. The back propagation algorithm, as given in Chapter 1, can be programmed in hardware for extra speed.

- Consider alternative derivations of the extended VLSI merit function. The VLSI merit function in Chapter 3 is tightly coupled to the definition of error. Some research is needed to examine the propagation of errors from module to module so that the aggregate VLSI merit function of a block can be found by knowing the VLSI merit functions of the sub–blocks.

Bibliography

[1] Adaptive Solutions, Inc., Beaverton, OR. *CNAPS - 1064 Digital Neural Processor*, 1993.

[2] A. Agranat and A. Yariv. Semiparallel microelectronic implemenation of neural network models using CCD technology. *Electronics Letters*, 23(11):580–581, May 1987.

[3] Hassan M. Ahmed. *Signal Processing Algorithms and Architectures.* PhD thesis, Stanford University, June 1982.

[4] Hassan M. Ahmed. Alternative arithmetic unit architectures for VLSI digital signal processors. In S. Y. Kung, H. J. Whitehouse, and T. Kailath, editors, *VLSI and Modern Signal Processing*, chapter 16. Prentice–Hall, Englewood Cliffs, New Jersey, 1985.

[5] Michael A. Arbib. *Brains, Machines, and Mathematics.* Springer–Verlag, New York, 1964.

[6] Michael A. Arbib. *The Metaphorical Brain 2: Neural Networks and Beyond.* Wiley, New York, 1989.

[7] S. Bhide, S. Gazula, V. Guerrow, G. Shebert, and M. Kabuka. An ASIC implementation of a user–configurable boolean neural network chip. In *World Congress on Neural Networks*, pages IV–239–244, Hillsdale, NJ, July 11–15 1993. International Neural Network Society, Lawrence Erlbaum Associates, Inc.

[8] H. D. Block. The perceptron: A model for brain functioning: I. *Reviews of Modern Physics*, 34:113–135, 1962.

[9] T. H. Borgstrom, M. Ismail, and S. B. Bibyk. Programmable current mode network for implementation in analogue VLSI. *IEE Proceedings*, 137(2):75–84, April 1990.

[10] Canadian Microelectronics Coporation, Kingston, Ontario. *The CMOS4S Standard Cell Library*, March 1990. Report ICI-021R0.

[11] R. M. M Chen, Y. M. Lai, and Y. S. Lee. A CAD technique for tollerance design and yield maximization of feedback loops in switching regulators. In *International Symposium on Circuits and Systems*, pages 746–749. IEEE, May 1992.

[12] A. Cichocki, J. Ramirez-Angulo, and R. Unbehauen. Architectures for analog VLSI implementations of neural networks for solving linear equations with inequality constraints. In *International Symposium on Circuits and Systems*, pages 1529–1532. IEEE, May 1992.

[13] A. Cichocki and R. Unbehauen. *Neural Networks for Optimization and Signal Processing.* Wiley, Chichester, England, 1993.

[14] Dean R. Collins and P. Andrew Penz. Considerations for neural network hardware implementations. In *International Symposium on Circuits and Systems*, volume 2, pages 834–836, Portland, OR., May 1989. IEEE.

[15] W. Robert Daasch and Martine Wedlake. Rapid layout of a continuous–time transconductance–C filter. In *International Symposium on Circuits and Systems*, pages 2256–2259. IEEE, May 1992.

[16] W. Robert Daasch, Martine Wedlake, and Rolf Schaumann. Automatic generation of CMOS continuous-time elliptic filters. *Electronic Letters*, 28:2215–2216, November 1992.

[17] W. Robert Daasch, Martine Wedlake, Rolf Schaumann, and Pan Wu. Automation of the IC layout of continuous–time transconductance–C filters. *International Journal of Circuit Theory and Applications*, 20:267–282, 1992.

[18] Alvin M. Despain. Fourier transform computers using CORDIC iterations. *IEEE Transactions on Computers*, C–23(10):993–1001, October 1974.

[19] R. Dominguez-Castro, A. Rodriguesz-Vazquez, J. L. Huertas, and E. Sanchez-Sinecio. Architectures and building blocks for CMOS VLSI analog "neural" programmable optimizers. In *International Symposium on Circuits and Systems*, pages 1525–1528. IEEE, May 1992.

[20] John G. Elias and Samer M. Meshreki. Implementing dendritic tree resistances with switched capacitors. In *World Congress on Neural Networks*, pages IV–523–526, Hillsdale, NJ, July 11–15 1993. International Neural Network Society, Lawrence Erlbaum Associates, Inc.

[21] S. E. Fahlman and C. Lebiere. The cascade–correlation learning architecture. Technical Report CMU-CS-91-100, School of Computer Science, CMU, August 1991.

[22] M. Fahmi, F. El Guibaly, S. Sunder, and D. Shpak. Design of novel serial–parallel inner-product processors. Pre–published copy, 1994.

[23] R. Fletcher. *Practical Methods of Optimization.* Wiley, Toronto, 2nd edition, 1987.

[24] B. Furman and A. A. Abidi. An analog CMOS backward error–propagation LSI. In *Proceedings of the 22nd ASILOMAR Annual Conference on Signals, Systems, and Computers*, pages 645–648, 1988.

[25] C. Gamrat, P Peretto, A. Mougin, and O. Ulrich. MIND–1024: A 1024 neurons fully connected real-time neurocomputer. In *World Congress on Neural Networks*, pages IV–783–786, Hillsdale, NJ, July 11–15 1993. International Neural Network Society, Lawrence Erlbaum Associates, Inc.

[26] Lance A. Glasser and Daniel W. Dobberpuhl. *The Design and Analysis of VLSI Circuits.* Addison–Wesley, Reading, Ma., 1985.

[27] G. L. Haviland and A. A. Tuszynski. A CORDIC arithmetic processor chip. *IEEE Transactions on Computers*, C–29(2):68–78, February 1980.

[28] Donald O. Hebb. *The Organization of Behavior.* Wiley, Ney York, 1949.

[29] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitaytive Approach.* Morgan Kaufmann Publishers, San Mateo, California, 1990.

[30] Paul W. Hollis, John S. Harper, and John J. Paulos. The effects of precision constraints in a backpropagation learning network. *Neural Computation*, 2:363–373, 1990.

[31] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79:2554–2558, 1982.

[32] J. J. Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the National Acadamey of Sciences*, 81:3088–3092, 1984.

[33] V. Hu, A. Kramer, and P. K. Ko. EEPROMs as analog storage device for neural nets. *Neural Networks*, 1:3185, 1988.

[34] International Neural Network Society. *World Congress on Neural Networks*, Hillsdale, NJ, July 11-15 1993. Lawrence Eribaum Associates, Inc.

[35] William James. *Psychology (Briefer Course)*. Holt, Ney York, 1890.

[36] David Kincaid and Ward Cheney. *Numerical Analysis:Mathematics of Scientific Computing*. Brooks/Cole, Pacific Grove, Ca., 1991.

[37] M. E. Kole, Z. Q. Ning, A. J. Mouthaan, and H. Wallinga. Comparator design automation in SEAS. In *International Symposium on Circuits and Systems*, pages 1949–1952. IEEE, May 1992.

[38] S. Y. Kung. *Digital Neural Networks*. PTR Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[39] B. Linares-Barranco, E. Sanchez-Sinecio, A. Rodriguez-Vazquez, and J. L. Huertas. Modular analog continuous-time VLSI neural networks with on chip hebbian learning and analog storage. *International Symposium on Circuits and Systems*, 3:1533–1536, May 1992.

[40] Richard P. Lippmann. An introduction to computing with neural nets. *ASSP Magazine*, pages 4–22, April 1987.

[41] Richard Lyon. VLSI and machines that hear. In Robert Suaya and Graham Birtwistle, editors, *VLSI and Parallel Computation*, chapter 6, pages 416–441. Morgan Kaufmann, San Mateo, California, 1990.

[42] T. H. Madraswala, B. J. Mohd, M. Ali, R. Premi, and Dr. M. A. Bayoumi. A reconfigurable 'ANN' architecture. *International Symposium on Circuits and Systems*, 3:1569–1572, May 1992.

[43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[44] Marvin L. Minsky and Seymour A. Papert. *Perceptrons*. MIT Press, Cambridge, Massachusetts, expanded edition, 1988.

[45] Aria Nosratinia, M. Ahmadi, M. Shridhar, and G. A. Jullien. A hybrid architecture for feed-forward multi-layer neural networks. In *International Symposium on Circuits and Systems*, volume 3, pages 1541–1544, San Diego, May 1992. IEEE.

[46] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipies in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge, 1990.

[47] U. Ramacher, W. Raab, J. Anlauf, U. Hachmann, J. Beichter, N. Bruls, M. Weßeling, and E. Sicheneder. Multiprocessor and memory architecture of the neurocomputer SYNAPSE-1. In *World Congress on Neural Networks*, pages IV–775–778, Hillsdale, NJ, July 11–15 1993. International Neural Network Society, Lawrence Erlbaum Associates, Inc.

[48] Russell Reed. Pruning algorithms – a survey. *IEEE Transactions on Neural Networks*, 5(5):740–747, September 1993.

[49] Sameh E. Rehan and Mohamed I. Elmasry. VLSI implementation of a prototype MLP using novel programmable switched-resistor CMOS ANN chip. In *World Congress on Neural Networks*, pages IV–95–98, Hillsdale, NJ, July 11–15 1993. International Neural Network Society, Lawrence Erlbaum Associates, Inc.

[50] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958.

[51] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, volume 1, pages 318–362. MIT Press, Cambridge, MA, 1986.

[52] Jay P. Sage, Richard S. Withers, and Karl E. Thompson. MNOS/CCD circuits for neural network implementations. In *International Symposium on Circuits and Systems*, volume 2, pages 1207–1209, Portland, OR., May 1989. IEEE.

[53] Charles L. Seitz. Concurrent architectures. In Robert Suaya and Graham Birtwistle, editors, *VLSI and Parallel Computation*, chapter 1, pages 1–84. Morgan Kaufmann, San Mateo, California, 1990.

[54] R. L. Shimabukuro, P. A. Shoemaker, and M. E. Stewart. Circuitry for artificial neural networks with non–volatile analog memories. In *International Symposium on Circuits and Systems*, volume 2, pages 1217–1220, Portland, OR., May 1989. IEEE.

[55] Bernard Sklar. *Digital Communications: Fundamentals and Applications.* PTR Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[56] Charles W. Therrien. *Decision Estimation and Classification: An Introduction to Pattern Recognition and Related Topics.* Wiley, New York, 1989.

[57] D. Timmermann, B. Rix, H. Hahn, and B. J. Hosticka. A CMOS floating–point vector–arithmetic unit. *IEEE Journal of Solid–State Circuits*, 29(5):634–639, May 1994.

[58] Jeffrey D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, Rockville, Maryland, 1984.

[59] J. E. Volder. The CORDIC trigonometric computing technique. *IRE Transactions on Electronic Computers*, EC–8(3):330–334, September 1959.

[60] J. S. Walther. A unified algorithm for elementary functions. In *Spring Joint Computer Conference*, pages 379–385, USA, May 18–20 1971. American Federation of Information Processing Societies, IEEE.

[61] Neil Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective.* Addison–Wesley, Reading, Ma., 1988.

[62] Marvin H. White and Chun-Yu Chen. Electrically modifiable nonvolatile synapses for neural networks. In *International Symposium on Circuits and Systems*, volume 2, pages 1213–1216, Portland, OR., May 1989. IEEE.

[63] Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits. *IRE WESCON Convention Record*, pages 96–104, 1960.

[64] Xilinx, Inc. *The Programmable Logic Data Book*, 1994.

[65] Andreas Zell, Niels Mache, Ralf Hubner, Gunter Maimer, Michael Vogt, Kai-Uwe Herrmann, Michael Schmalzl, Tillman Somer, Artemis Hatzigeorgiu, Sven Doring, and Dietmar Posselt. *SNNS: Stuttgart Neural Network Simulator User Manual.* University of Stuttgart, Institute for Parallel and Distributed High Performance Systems, version 3.0 edition, March 1993. SNNS is freely available via anonymous ftp from ftp.informatik.uni-stuttgart.de (129.69.211.2).

[66] Yi-Shen Zhu and Wai-Kai Chen. A CAD method for the design of a doubly–matched broadband network having complex transmission zeros. In *International Symposium on Circuits and Systems*, pages 1965–1968. IEEE, May 1992.

Appendix A

Static CORDIC Emulator

A.1 C Language Implementation

```
/*
 * Author  : Martine Wedlake
 */

#include <math.h>
#ifdef solaris
#include <sunmath.h>
#endif /* solaris */

#define NUM_FRAC 5      /* number of fraction bits */
#define NUM_INT 2       /* number of integer portion bits */

#define TRUNCATE_BITS  /* Truncate bits? */


float T(x)
float x;
{
    float data;
#ifdef TRUNCATE_BITS
    data = (rint(exp2((float)NUM_FRAC)*x)/exp2((float)NUM_FRAC));
```

```
    if (data >  exp2((float)NUM_INT)) data =  exp2((float)NUM_INT);
    if (data < -exp2((float)NUM_INT)) data = -exp2((float)NUM_INT);
#else
    data = x;
#endif /* TRUNCATE_BITS */
    return data;
}



/*
 * This is implemented in hardware by returning the
 * uppermost bit (sign bit) of a value.
 */

float sign(x)
float x;
{
    return x < 0 ? -1.0 : 1.0;
}



/*
 * For longer bit lengths than 8 we need to represent this
 * as a ROM, but for less than that we can recognize that
 * tanh() is fairly linear about 0.  Thus tanh() and exp2()
 * are very close for most of the values of i.  (In fact,
 * for 5 bits of accuracy, only one bit is different when
 * i = 0.5).  In the hardware implementation, we can use
 * this to reduce hardware requirements by using a simple
 * decoder to translate i into exp2() and then depending
 * on the value of m, and i we can add in the extra bit.
 */

float theta(m, i)
int m;
int i;
{
    float exp_two;
```

```
    exp_two = exp2(-1.0*i);

    switch (m)
    {
    case 1:
        return T(atanh(exp_two));
    case  0:
        return T(exp_two);
    default:
        printf("theta(m,i) given incorrect m (=%d)", m);
        exit(1);
    }
}



/*
 * The Barrel Shifter is provided by the XACT library, however
 * I had to make modifications to the cell in order to ensure
 * that the shifter is performing an arithmetic shift right
 * operation (not a rotate right).
 */

float BS(in, i)
float in;
int i;
{
    float exp_two;

    exp_two = exp2(-1.0*i);

    return T(in * exp_two);
}



/*
 * This is the CORDIC Neuron Unit.  The CORDIC method has
 * been modified in a few places to allow for my implementation
```

```
 * here.  Most notably is the addition of a Zbp signal that
 * implements a register forwarding structure between the Y
 * register and the Z register (Zbp stands for Z by-pass).
 *
 * The basic structure of this emulator is to divide the
 * design into two parts: control, and engine.
 *
 * The control part is represented as an array of values.  This
 * emulates a control ROM in actual hardware implementaton.  the
 * control vector is passed through some logic  to interface it
 * to the CORDIC engine.
 *
 * The CORDIC engine is nothing more than a few adder/subtractors,
 * registers, etc. all controled from the controller.
 *
 * Debugging statements are included to verify operation of the
 * engine.
 *
 */

neuron(w_in, x_in, bias, y_out)
float w_in[8];
float x_in[8];
float bias;
float *y_out;
{
    int ctr, i, m, r, mult, hyp, Ystart;/* output control lines */

    float old_x, old_y, old_z;          /* CORDIC registers  & IO*/
    float x, y, z;
    int x_ctr, w_ctr;

    int Xin, Yin, Zin;                  /* CORDIC control lines */
    int Xen, Yen, Zen;
    int Xset, Yclr, Zbp;
    int delta;

                                    /*  i m r mult hyp Ystart */
    static char controlROM[48+11+5][6] = {
                                        {0,0,1,1   ,0   ,1}, /* M 1 */
```

```
{1,0,1,0   ,0  ,0},
{2,0,1,0   ,0  ,0},
{3,0,1,0   ,0  ,0},
{4,0,1,0   ,0  ,0},
{5,0,1,0   ,0  ,0},

{1,0,1,1   ,0  ,0}, /* M 2 */
{1,0,1,0   ,0  ,0},
{2,0,1,0   ,0  ,0},
{3,0,1,0   ,0  ,0},
{4,0,1,0   ,0  ,0},
{5,0,1,0   ,0  ,0},

{1,0,1,1   ,0  ,0}, /* M 3 */
{1,0,1,0   ,0  ,0},
{2,0,1,0   ,0  ,0},
{3,0,1,0   ,0  ,0},
{4,0,1,0   ,0  ,0},
{5,0,1,0   ,0  ,0},

{1,0,1,1   ,0  ,0}, /* M 4 */
{1,0,1,0   ,0  ,0},
{2,0,1,0   ,0  ,0},
{3,0,1,0   ,0  ,0},
{4,0,1,0   ,0  ,0},
{5,0,1,0   ,0  ,0},

{1,0,1,1   ,0  ,0}, /* M 5 */
{1,0,1,0   ,0  ,0},
{2,0,1,0   ,0  ,0},
{3,0,1,0   ,0  ,0},
{4,0,1,0   ,0  ,0},
{5,0,1,0   ,0  ,0},

{1,0,1,1   ,0  ,0}, /* M 6 */
{1,0,1,0   ,0  ,0},
{2,0,1,0   ,0  ,0},
{3,0,1,0   ,0  ,0},
{4,0,1,0   ,0  ,0},
{5,0,1,0   ,0  ,0},
```

```
                {1,0,1,1    ,0   ,0}, /* M 7 */
                {1,0,1,0    ,0   ,0},
                {2,0,1,0    ,0   ,0},
                {3,0,1,0    ,0   ,0},
                {4,0,1,0    ,0   ,0},
                {5,0,1,0    ,0   ,0},

                {1,0,1,1    ,0   ,0}, /* M 8 */
                {1,0,1,0    ,0   ,0},
                {2,0,1,0    ,0   ,0},
                {3,0,1,0    ,0   ,0},
                {4,0,1,0    ,0   ,0},
                {5,0,1,0    ,0   ,0},

                {1,1,1,0    ,1   ,0}, /* HYP */
                {1,1,1,0    ,0   ,0},
                {1,1,1,0    ,0   ,0},
                {1,1,1,0    ,0   ,0},
                {1,1,1,0    ,0   ,0},
                {2,1,1,0    ,0   ,0},
                {3,1,1,0    ,0   ,0},
                {4,1,1,0    ,0   ,0},
                {5,1,1,0    ,0   ,0},
                {5,1,1,0    ,0   ,0},
                {5,1,1,0    ,0   ,0},

                {1,0,0,0    ,0   ,0}, /* DIV */
                {2,0,0,0    ,0   ,0},
                {3,0,0,0    ,0   ,0},
                {4,0,0,0    ,0   ,0},
                {5,0,0,0    ,0   ,0},
                    };

/* Init */

x = y = z = -9.9999;
i = x_ctr = w_ctr = 0;
```

```
/*
 * We loop through the entire control ROM.
 */

for (ctr = 0; ctr < 48+11+5; ctr++)
{

    /* ------------------- */
    /* Control part begins */
    /* ------------------- */

    /*
     * Retrieve the control signals from the control ROM
     */

    i      = controlROM[ctr][0];
    m      = controlROM[ctr][1];
    r      = controlROM[ctr][2];
    mult   = controlROM[ctr][3];
    hyp    = controlROM[ctr][4];
    Ystart= controlROM[ctr][5];

    /*
     * Boolean assignments (This interfaces from the
     * control ROM to the CORDIC engine)
     */

    Xset = hyp;
    Yclr = hyp;
    Zbp  = hyp;

    Xen = m || mult;
    Yen = m || Yclr || (!mult);
    Zen = 1;

    Zin = mult;
    Yin = Ystart;
    Xin = mult;

    /* ------------------------------------ */
```

```
        /* Control part ends, CORDIC engine begins */
        /* ------------------------------------- */


#ifdef PRINT
        /*
         * Debugging statements
         */

        printf("%2d:X=%8.5f Y=%8.5f Z=%8.5f ", ctr, x, y, z);
        printf("Xset=%1d Yclr=%1d Zbp=%1d ", Xset, Yclr, Zbp);
        printf("Xen=%1d Yen=%1d ", Xen, Yen);
        printf("Xin=%1d Zin=%1d |", Xin, Zin);
        printf("Ystart=%1d mult=%1d hyp=%1d ", Ystart, mult, hyp);
        printf("r=%1d m=%1d i=%1d\n", r, m, i);
#endif /* PRINT */

        /*
         * Need these because in the hardware implementation
         * we don't worry about side effects since everything
         * in the CORDIC step is run in parallel, synchronized
         * by the clock.
         */

        old_x = (x);
        old_y = (y);
        old_z = (z);

        /*
         * booleans local to CORDIC engine
         */

        delta = (r ? sign(z) : -sign(x)*sign(y));

        /*
         * The CORDIC step
         */

        x = T(old_x + delta * BS(old_y,i));
        y = T(old_y + delta * BS(old_x,i));
```

```
        z = T(old_z - delta * theta(m,i));

        /*
         * Code to simulate control lines
         *
         * These implement the operations controled by the
         * control signals.
         */

        if (!Zen) z = old_z;
        if (!Yen) y = old_y;
        if (!Xen) x = old_x;
        if (Xset) x = 1.0;
        if (Yclr) y = 0.0;
        if (Zbp)  z = old_y;
        if (Yclr) y = 0.0;
        if (Xin)  z = T(x_in[x_ctr++]); /* because |x_in| < 1 */
        if (Yin)  y = bias;
        if (Zin)  x = T(w_in[w_ctr++]);
        }

#ifdef PRINT
        /*
         * Debugging statements
         */

        printf("   X=%8.5f Y=%8.5f Z=%8.5f\n", x, y, z);
#endif /* PRINT */

    *y_out = z;
}
```

A.2 Emulator Output Trace

#	X	Y	Z	Xset	Yclr	Zbp	Xen	Yen	Xin	Zin	r	i
0:	*******	*******	*******	0	0	0	1	0	1	1	1	0
1:	0.10000	0.00000	1.00000	0	0	0	0	1	0	0	1	1
2:	0.10000	0.05000	0.50000	0	0	0	0	1	0	0	1	2
3:	0.10000	0.07500	0.25000	0	0	0	0	1	0	0	1	3
4:	0.10000	0.08750	0.12500	0	0	0	0	1	0	0	1	4
5:	0.10000	0.09375	0.06250	0	0	0	0	1	0	0	1	5
6:	0.10000	0.09688	0.03125	0	0	0	1	0	1	1	1	1
7:	0.10000	0.09688	1.00000	0	0	0	0	1	0	0	1	1
8:	0.10000	0.14688	0.50000	0	0	0	0	1	0	0	1	2
9:	0.10000	0.17188	0.25000	0	0	0	0	1	0	0	1	3
10:	0.10000	0.18438	0.12500	0	0	0	0	1	0	0	1	4
11:	0.10000	0.19063	0.06250	0	0	0	0	1	0	0	1	5
12:	0.10000	0.19375	0.03125	0	0	0	1	0	1	1	1	1
13:	0.10000	0.19375	1.00000	0	0	0	0	1	0	0	1	1
14:	0.10000	0.24375	0.50000	0	0	0	0	1	0	0	1	2
15:	0.10000	0.26875	0.25000	0	0	0	0	1	0	0	1	3
16:	0.10000	0.28125	0.12500	0	0	0	0	1	0	0	1	4
17:	0.10000	0.28750	0.06250	0	0	0	0	1	0	0	1	5
18:	0.10000	0.29063	0.03125	0	0	0	1	0	1	1	1	1
19:	0.10000	0.29063	1.00000	0	0	0	0	1	0	0	1	1
20:	0.10000	0.34063	0.50000	0	0	0	0	1	0	0	1	2
21:	0.10000	0.36563	0.25000	0	0	0	0	1	0	0	1	3
22:	0.10000	0.37813	0.12500	0	0	0	0	1	0	0	1	4
23:	0.10000	0.38438	0.06250	0	0	0	0	1	0	0	1	5
24:	0.10000	0.38750	0.03125	0	0	0	1	0	1	1	1	1
25:	0.10000	0.38750	1.00000	0	0	0	0	1	0	0	1	1
26:	0.10000	0.43750	0.50000	0	0	0	0	1	0	0	1	2
27:	0.10000	0.46250	0.25000	0	0	0	0	1	0	0	1	3
28:	0.10000	0.47500	0.12500	0	0	0	0	1	0	0	1	4
29:	0.10000	0.48125	0.06250	0	0	0	0	1	0	0	1	5
30:	0.10000	0.48438	0.03125	0	0	0	1	0	1	1	1	1
31:	0.10000	0.48438	1.00000	0	0	0	0	1	0	0	1	1
32:	0.10000	0.53438	0.50000	0	0	0	0	1	0	0	1	2
33:	0.10000	0.55937	0.25000	0	0	0	0	1	0	0	1	3
34:	0.10000	0.57187	0.12500	0	0	0	0	1	0	0	1	4
35:	0.10000	0.57812	0.06250	0	0	0	0	1	0	0	1	5
36:	0.10000	0.58125	0.03125	0	0	0	1	0	1	1	1	1
37:	0.10000	0.58125	1.00000	0	0	0	0	1	0	0	1	1

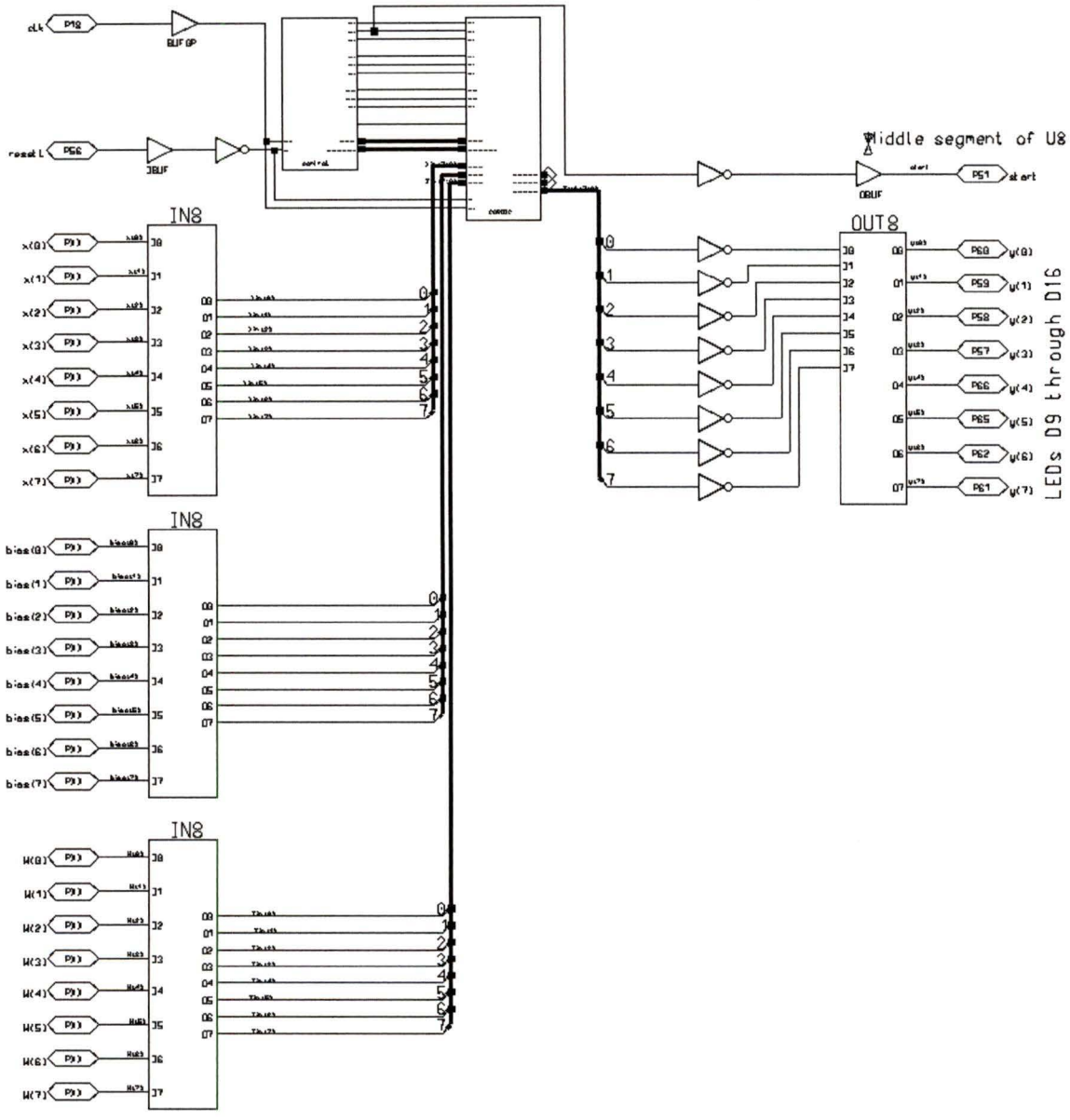
```
38: 0.10000  0.63125  0.50000    0   0   0   0   1   0   0 1 2
39: 0.10000  0.65625  0.25000    0   0   0   0   1   0   0 1 3
40: 0.10000  0.66875  0.12500    0   0   0   0   1   0   0 1 4
41: 0.10000  0.67500  0.06250    0   0   0   0   1   0   0 1 5
42: 0.10000  0.67813  0.03125    0   0   0   1   0   1   1 1 1
43: 0.10000  0.67813  1.00000    0   0   0   0   1   0   0 1 1
44: 0.10000  0.72813  0.50000    0   0   0   0   1   0   0 1 2
45: 0.10000  0.75313  0.25000    0   0   0   0   1   0   0 1 3
46: 0.10000  0.76562  0.12500    0   0   0   0   1   0   0 1 4
47: 0.10000  0.77188  0.06250    0   0   0   0   1   0   0 1 5
48: 0.10000  0.77500  0.03125    1   1   1   1   1   0   0 1 1
49: 1.00000  0.00000  0.77500    0   0   0   1   1   0   0 1 1
50: 1.00000  0.50000  0.22569    0   0   0   1   1   0   0 1 1
51: 1.25000  1.00000 -0.32361    0   0   0   1   1   0   0 1 1
52: 0.75000  0.37500  0.22569    0   0   0   1   1   0   0 1 1
53: 0.93750  0.75000 -0.32361    0   0   0   1   1   0   0 1 2
54: 0.75000  0.51562 -0.06820    0   0   0   1   1   0   0 1 3
55: 0.68555  0.42188  0.05746    0   0   0   1   1   0   0 1 4
56: 0.71191  0.46472 -0.00512    0   0   0   1   1   0   0 1 5
57: 0.69739  0.44247  0.02614    0   0   0   1   1   0   0 1 5
58: 0.71122  0.46427 -0.00512    0   0   0   1   1   0   0 1 5
59: 0.69671  0.44204  0.02614    0   0   0   0   1   0   0 0 1
60: 0.69671  0.09369  0.52614    0   0   0   0   1   0   0 0 2
61: 0.69671 -0.08049  0.77614    0   0   0   0   1   0   0 0 3
62: 0.69671  0.00660  0.65114    0   0   0   0   1   0   0 0 4
63: 0.69671 -0.03695  0.71364    0   0   0   0   1   0   0 0 5
    0.69671 -0.01517  0.68239
Result = 0.682386
```

Appendix B

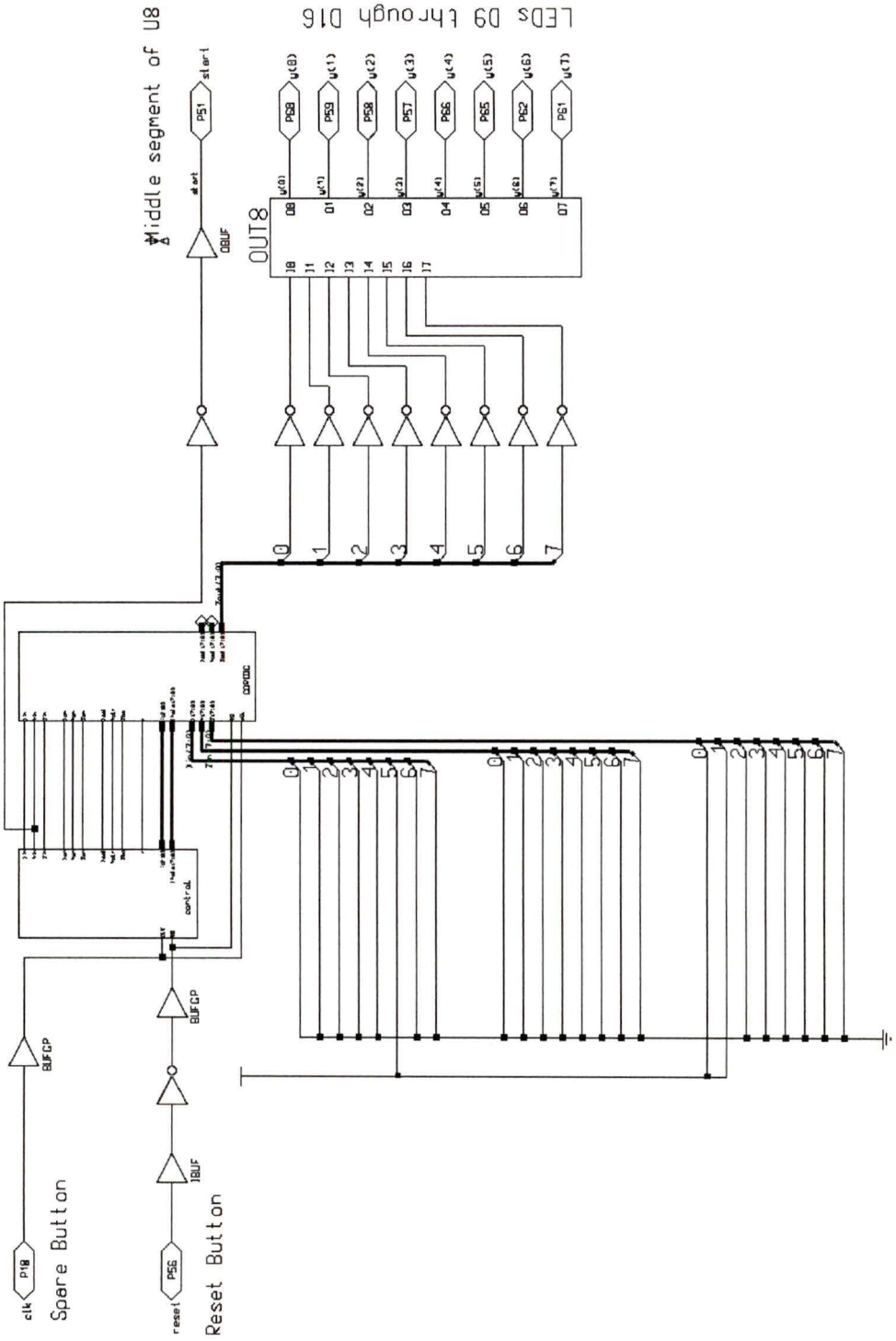
Xilinx Implementation

B.1 Schematics for Xilinx Implementation

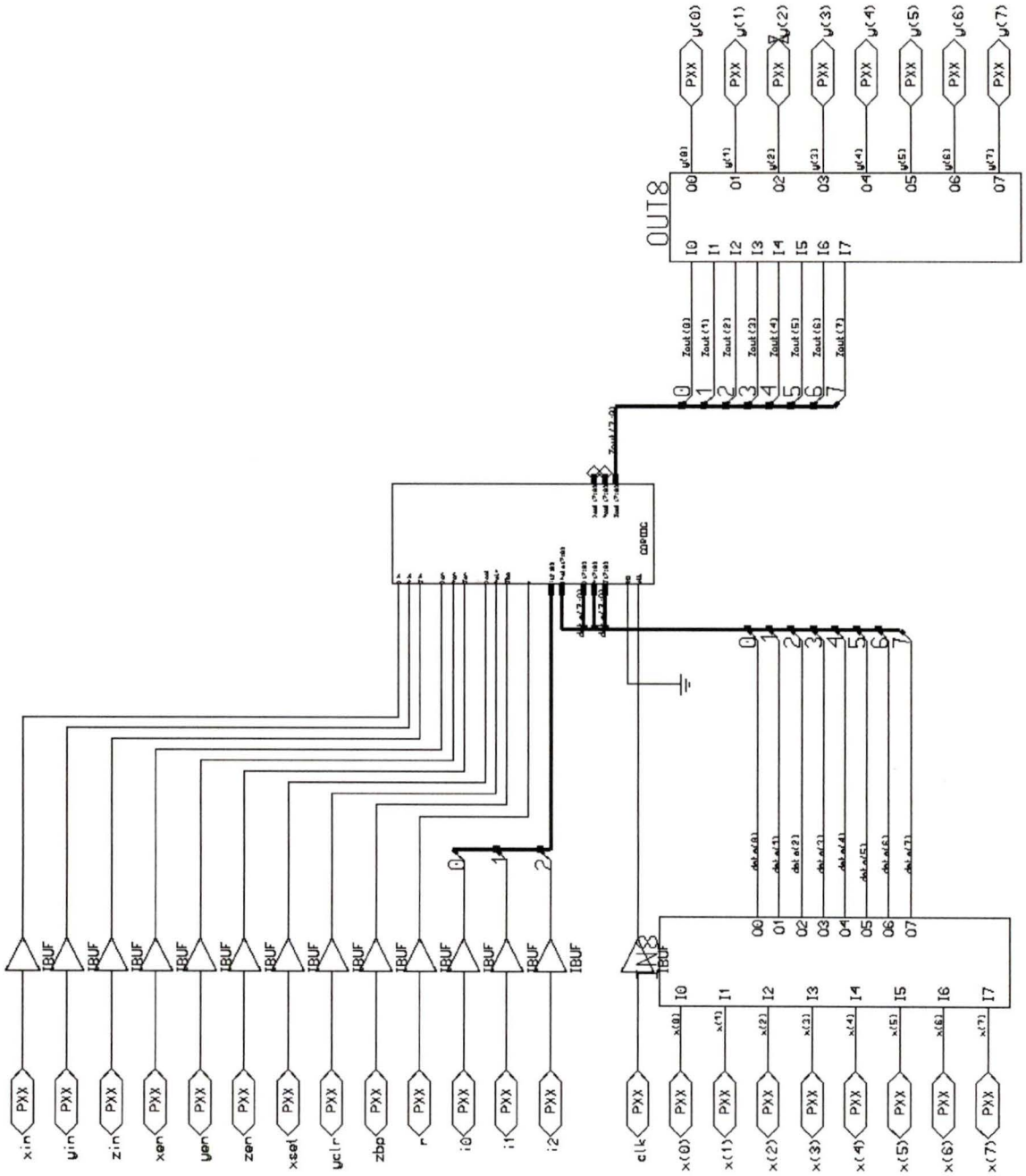
B.1.1 SLP



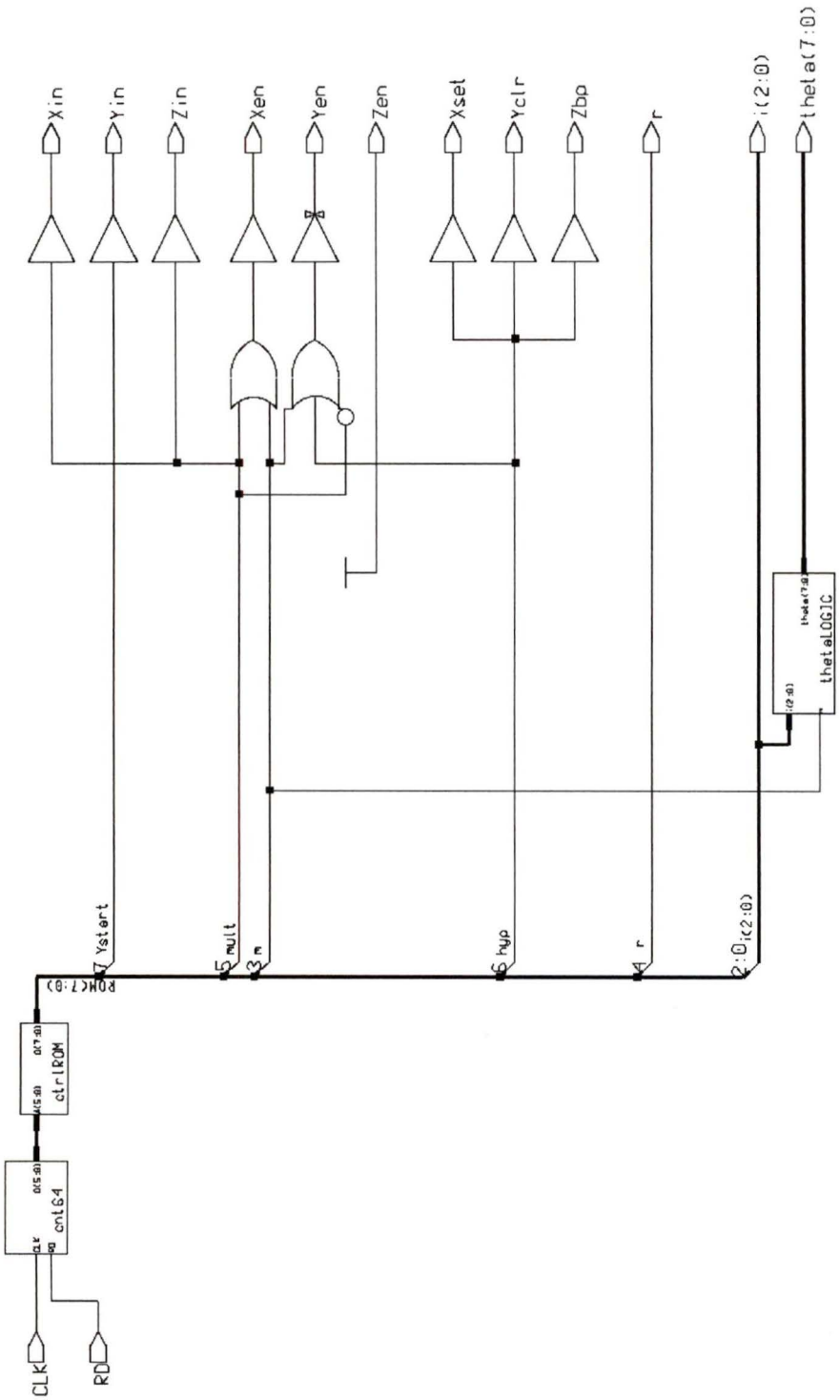
B.1.2 SLP_TEST


Middle segment of U8
LEDs D9 through D16
OUT8
OBUF
BUFGP
IBUF
clk
Spare Button
reset
Reset Button

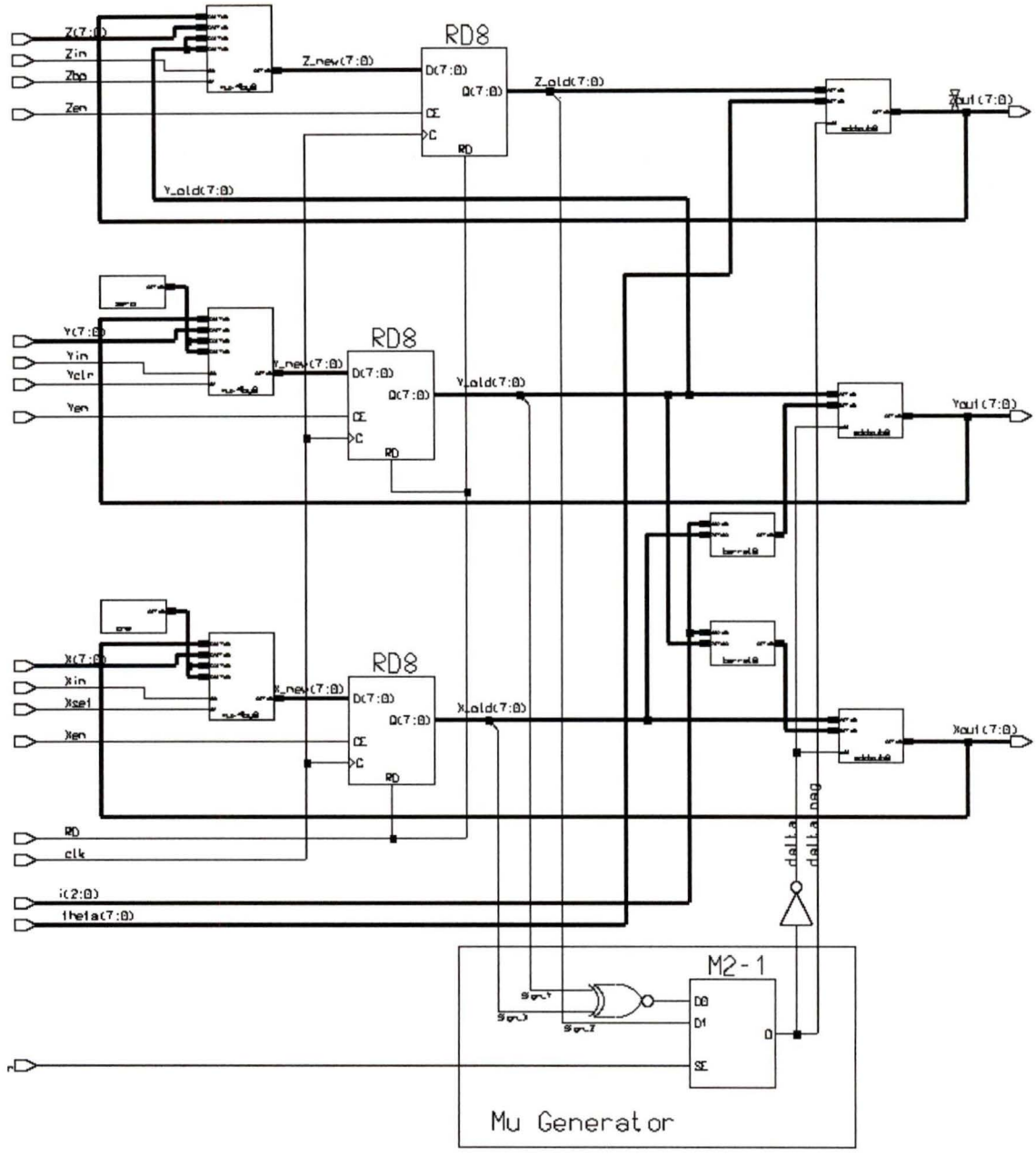
B.1.3 ENGINE


OUT8
IBUF
IN8
PXX
xin
yin
zin
xen
yen
zen
xsel
yclr
zbp
r
i0
i1
i2
clk
x(0)
x(1)
x(2)
x(3)
x(4)
x(5)
x(6)
x(7)
y(0)
y(1)
y(2)
y(3)
y(4)
y(5)
y(6)
y(7)

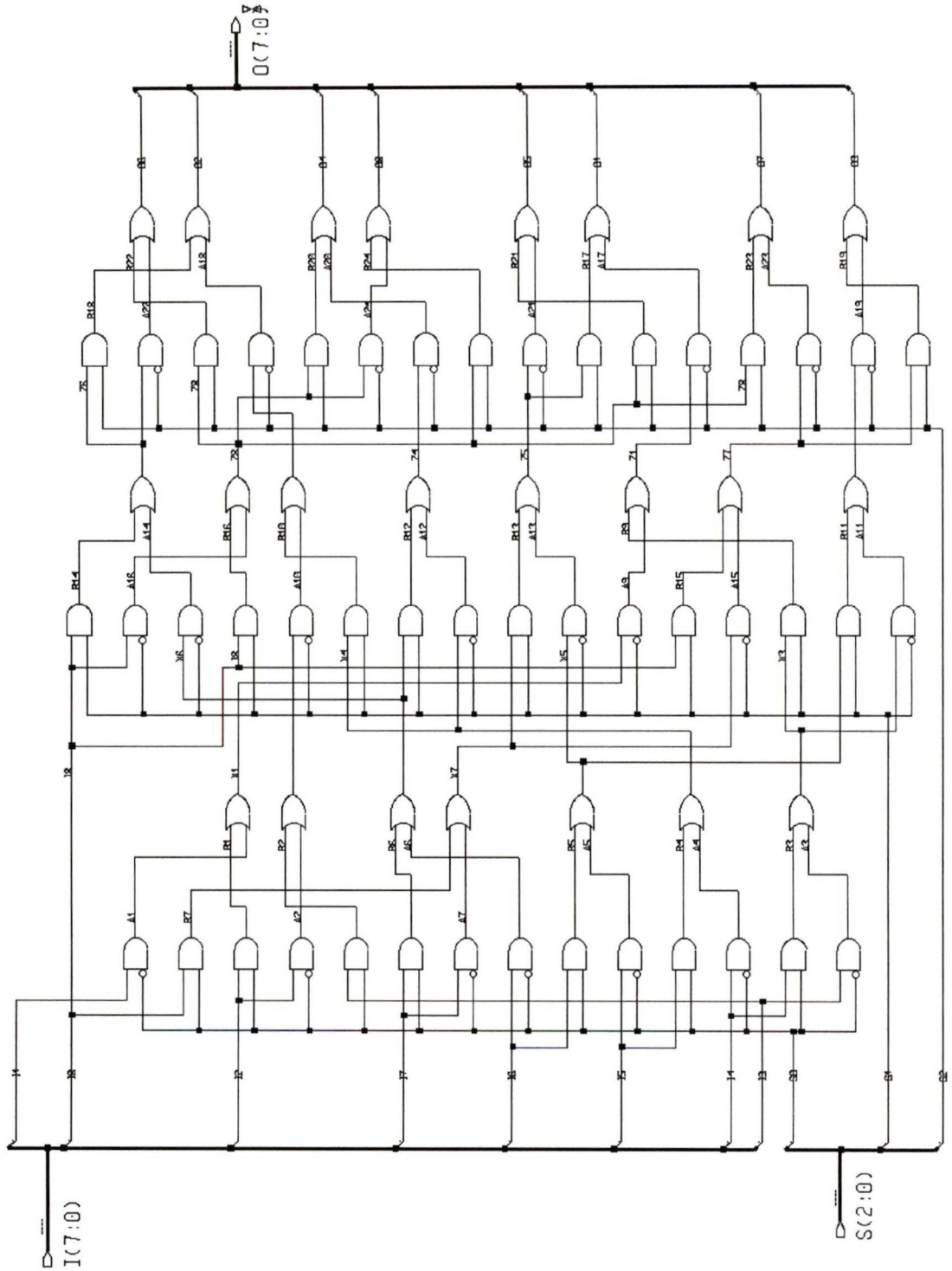
B.1.4 Design of Static CORDIC Controller



B.1.5 Design of CORDIC Engine



B.1.6 Eight Bit Barrel Shifter



B.2 Xilinx PPR Report

The following is a slightly edited form of the PPR Report generated during the ppr design process. The table of contents, and vertical spacing are removed.

```
                    PPR RESULTS FOR DESIGN SLP
                      From PPR Version 1.31
                   Run on 1995/03/03 at 16:01:23

                            Xilinx, Inc.

                (c) Copyright 1995.  All Rights Reserved.


Partition, Place and Route Summary
----------------------------------

Input XNF Design Statistics (Note 1)

  Number of Logic Symbols:        737
  Number of Flip Flops:           30
  Number of 3-State Buffers:      0
  Number of IO Pads:              35
  Number of Hard Macros:          0
  Number of Nets:                 869
  Number of Pins:                 2633
  Equivalent "Gate Array" Gates:  2644  (Note 2)
  - From Logic:                   2164
  - From Random Access Memories:  0
  - From Read Only Memories:      480


Partitioned Design Utilization Using Part 4003APC84-6 (Note 3)

  Occupied CLBs (Note 4):         Used=98   Max=100  Util=98%
  Packed CLBs   (Note 4):         Used=90   Max=100  Util=90%
  Package Pins  (Note 5):         Used=35   Max=80   Util=43%
  FG Function Generators:         Used=180  Max=200  Util=90%
  H  Function Generators:         Used=49   Max=100  Util=49%
```

```
  Flip Flops     (Note 5):          Used=30   Max=360  Util=8%
  Memory Write Controls:            Used=0    Max=100  Util=0%
  3-State Buffers:                  Used=0    Max=280  Util=0%
  3-State Buffer Output Lines:      Used=0    Max=40   Util=0%
  Address Decoders:                 Used=0    Max=120  Util=0%
  Address Decoder Output Lines:     Used=0    Max=32   Util=0%


Routing Summary

  Number of unrouted connections: 0
  Number of pips used:            1630
  Number of local lines used:     451
  Number of double lines used:    302
  Number of long lines used:      131
  Number of global lines used:    9
  Number of decoder lines used:   0


PPR Parameters

  Design          = slp
  Parttype        = 4003APC84-6
  Improvecount    = 3
  Justflatten     = FALSE
  Seed            = 794246270
  Estimate        = FALSE
  Outfile         = <design name>


PPR Summary Explanatory Notes
-----------------------------
This section of the report contains the explanatory notes for the
entries and values which we presented on the previous page.


Note 1
------
The first section gives statistics about your design which served
as input to PPR.  The data file used to compute these statistics
```

```
is PPR's internal net list file which is derived from (and
closely related to) your input XNF file.  There is one case where
the internal file may be very different from your input XNF file:
if your XNF file contains references to other XNF files, PPR will
automatically merge the lower level net list into the top level
net list.  The internal net list file therefore represents a FLAT
version of your design.


Note 2
------
The value listed for the number of "gate array" gates is an
estimate which we provide for your reference.  It is based on the
symbols (AND, OR, FDRD, etc) in the internal net list file.  The
value is computed by adding the equivalent gate counts of each
symbol.  The symbol gate counts are derived from gate array data
books.

Since the number of "gate array" gates is based on the internal
net list file, it includes the contents of any lower level sub-
designs which are referenced in your top level XNF file.  It also
includes the contents of any hard macros which you have used in
your design.

We also show a break down of the total number of gates into
three categories: those which are due to combinatorial and
sequential logic, those which are due to random access memories,
and those which are due to read only memories. If you are using
MEMGEN to generate a memory structure, you can also refer to its
output files to determine how the gate count for the structure
was computed.


Note 3
------
The second section gives statistics about your design after it
has been partitioned for the selected part.  Most often, you want
to know the percent utilization of the LCA by your design.  There
are many ways to compute the utilization for a design -- it
depends on what LCA resource you are interested in.  In this
```

section, we present several statistics which we have found to be of interest.

Note 4

The most general utilization statistic is the CLB utilization. We give two values here. The "Occupied CLB" utilization is computed by counting the CLBs that are not empty. This utilization value includes feedthrus which have been inserted into the implementation of your design by the router. The "Packed CLB" utilization is computed by counting the number of function generators and flip flops in your design, dividing each by 2, and taking the maximum of the two results. The packed CLB value represents the LCA utilization if we were to pack your design into as small an area as possible. In order to improve PPR execution time and LCA routability, PPR does not pack CLBs as tightly as possible. Instead, it spreads your design over all available CLBs with the result that the elements (function generators, flip flops, 3-state buffers) of any one CLB may not fully utilized.

The packed CLB utilization is a measure of how much more logic you may be able to put in your design and still have it fit on the LCA.

Note 5

You may sometimes notice differences between the number of package pins and flip flops given in this section, and the number of IO pads and flip flops given in the previous section. These differences, if they appear, are due to logic trimming which occurs before the partioning process. Pads are removed if they connect to a loadless or sourceless net. Flip flops are removed if one of the following conditions is true:

1. The output pin (Q) has no load.
2. The data pin (D) has no source.
3. The data pin is attached to GND and the init attribute value is "rd" (the output in such a case is always low, so

```
     the symbol is replaced by a GND net).
  4. The data pin is attached to VCC and the init attribute
     value is "sd" (the output in such a case is always high,
     so the symbol is replaced by a VCC net).
  5. The clk pin has no source, or is attached to GND or VCC.
     (the output in such a case depends on the value of the init
     attribute, so the symbol is replaced by a GND or VCC net).
  6. The reset pin (RD) is attached to VCC (the output in such
     a case is always low, so the symbol is replaced by a GND
     net).
  7. The set pin (SD) is attached to VCC (the output is this
     case is always high, so the symbol is replaced by a VCC
     net).
Flip flops may also be deleted if they are part of a sequential
logic loop which feeds itself and no other symbols.


Chip Pinout Description
-----------------------

Package Pin Location   Pin Name
--------------------   ----------------
P4                   : BIAS<0>
P81                  : BIAS<1>
P80                  : BIAS<2>
P69                  : BIAS<3>
P79                  : BIAS<4>
P82                  : BIAS<5>
P83                  : BIAS<6>
P84                  : BIAS<7>
P18                  : CLK
P56                  : RESETL
P51                  : START
P46                  : W<0>
P26                  : W<1>
P47                  : W<2>
P48                  : W<3>
P67                  : W<4>
P70                  : W<5>
P50                  : W<6>
```

```
P68                 : W<7>
P27                 : X<0>
P24                 : X<1>
P8                  : X<2>
P40                 : X<3>
P38                 : X<4>
P39                 : X<5>
P37                 : X<6>
P6                  : X<7>
P60                 : Y<0>
P59                 : Y<1>
P58                 : Y<2>
P57                 : Y<3>
P66                 : Y<4>
P65                 : Y<5>
P62                 : Y<6>
P61                 : Y<7>
```

VITA

Surname: Wedlake — Given Names: Martine Bruce

Place of Birth: Guelph, Ontario — Date of Birth: December 13, 1968

Educational Institutions Attended:

University of Victoria	1992 to 1995
Portland State University	1987 to 1992

Degrees Awarded:

BaSC (High Honours)	Portland State University	1992

Honours and Awards:

Eta Kappa Nu Honour Society	1991

Publications:

W. Robert Daasch and Martine Wedlake. Rapid layout of a continuous–time transconductance–C filter. In *International Symposium on Circuits and Systems*, pages 2256–2259. IEEE, May 1992.

W. Robert Daasch, Martine Wedlake, Rolf Schaumann, and Pan Wu. Automation of the IC layout of continuous–time transconductance–C filters. *International Journal of Circuit Theory and Applications*, 20:267–282, 1992.

W. Robert Daasch, Martine Wedlake, and Rolf Schaumann. Automatic generation of CMOS continuous-time elliptic filters. *Electronic Leters*, 28:2215–2216, November 1992.

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

The Derivation and Justification of a CORDIC Implemented Artificial Neuron.

Autho[redacted]

Martine Bruce Wedlake

5/17/95

Date