VLSI Implementation of a Fuzzy Cognitive System

by

Jay Diamond

A thesis
presented to the University of Manitoba
in partial fulfillment of the requirements
for the degree of
Master of Science

Department of
Electrical and Computer Engineering
University of Manitoba
Winnipeg, Manitoba, Canada 1990

ISBN 0-315-71781-5

VLSI IMPLEMENTATION OF A FUZZY COGNITIVE SYSTEM

BY

JAY DIAMOND

A thesis submitted to the Faculty of Graduate Studies of
the University of Manitoba in partial fulfillment of the requirements
of the degree of

MASTER OF SCIENCE

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Manitoba requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

This thesis deals with the VLSI implementation of a fuzzy cognitive system. Aspects of fuzzy set theory and neural networks are combined to produce a robust system for expressing uncertainty. A novel architecture for digital neural networks is introduced and analyzed. The system concept is implemented in a three-chip set which incorporates fault tolerance and a cellular automata based built-in self test. VLSI design trade-offs are explored and details of the implementation are presented.

Acknowledgements

I would like to thank Dr. Bob McLeod for his guidance, wisdom and support. Never has a graduate student had a finer advisor. This work would simply not have been possible without the patience and inspiration of Dr. Witold Pedrycz, and his *fuzzy* thinking. Dave Blight's willingness to fix any CAD tool problem at the expense of sleep, Roland Schneider's UNIX wizardry, and the diverse talents of other the VLSI gurus have all been invaluable. I would also like to thank my parents for putting up with my nocturnal insanity.

This work was supported through funding from the National Sciences and Engineering Research Council of Canada and through equipment loans from the Canadian Microelectronics Corporation.

Dedication

To all those who boldly agreed
that it was possible to create ANNs for AI
without any RNNs or RI.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Purpose

The purpose of this thesis is to examine the VLSI implementation of a fuzzy cognitive system. This complex title requires a more in-depth explanation of its meaning.

Very Large Scale Integration, or VLSI, refers to the implementation of an algorithm as an application specific integrated circuit, or ASIC. The science surrounding this 'silicon sculpture' is well documented, as well as much of the architecture involved in the digital implementations referred to throughout this thesis.

The *fuzzy set theory* involved in the system refers to the work of L. Zadeh and his school of thought [44]. This theory involves the use of multivalued logics to represent uncertainty. The next chapter of this thesis includes a more indepth introduction to fuzzy set theory, as it pertains to this work.

The word '*cognitive*' has been widely interpreted in most of the papers written on the subject of cognitive science. The term as used in this thesis is not meant to infer rational, *human-like* thought, but rather is used to reflect the system's ability to

perform contextual interpretation. A great problem in analyzing mass quantities of data is discriminating important facts to be extracted. This is the advantage of the system developed in this thesis.

The basic system described here is based on previous work in which software simulations were used for signal classification purposes [4]. The very general system described here is well suited to hardware implementation, and would have the advantage of near-real-time processing performance. This type of system is well suited to all types of pattern recognition/classification problems in which it would be advantageous to produce both quantitative *and* qualitative output.

1.2 The Problem

Recent progress in the development of fuzzy set theory has been centered around areas such as fuzzy controllers, expert systems, digital signal and image processing systems, and robotics [35]. With this progress has come an increase in system complexity, meaning that some software driven systems are very slow.

Concurrently, hardware development has advanced to very large scale integration (VLSI) on a single chip. In addition, a great deal of research has been conducted on massively parallel computing schemes involving neural networks. The learning capabilities of these networks have been rigorously documented in numerous case studies (involving, for example, speech and pattern recognition), [9] [18]. It is also evident that fuzzy sets form a well suited tool for modeling processes of knowledge representation, especially for adjusting the relevant cognitive perspective of a system, [22] [26].

The merger of these technologies would obviously be of great interest, and some research has been done in this field, particularly [13]. Most of this work has been centered around using fuzzy controllers as algorithms for control purposes realized in an uncertain environment.

Introduction of a referential structure to this merger allows a very general, qualitative output to be presented to the end user. This type of output is well suited for expert system applications as well as human interpretation.

By combining many diverse fields of study a very valuable tool for evaluation in general pattern recognition problems may be created.

1.3 Project Scope

This thesis covers introductory theory used in the implementation of a three chip set which forms the fuzzy cognitive system. Since such a large number of diverse fields of study are in incorporated in the development, only a short background and relevant information will be covered. No attempt is made to cover all details of *fuzzy set theory* or *neural networks*.

The main body of this thesis is involved with the development of three integrated circuits. Little attempt was made to connect the chips once fabricated, although the intent was that they were designed to work together. The chips were tested, and the performance and testability were analyzed.

While software simulations have been attempted, these are by no means exhaustive. Furthermore, since some preprocessing must be done off-line, it was assumed that this was relatively straightforward but time-consuming task, outside the scope

of this thesis. The main emphasis of this thesis is on the implementation of novel structures for particular computations.

Chapter 2

Overview

2.1 Fuzzy Set Theory

2.1.1 The Fuzzy Paradigm

Fuzzy set theory is an extrapolation from two-valued logic. Whereas *crisp* definitions are often used in mathematical modeling (especially in practical engineering applications), it is often very useful to use intermediate terms, somewhere between two extremes.

For example, in conventional processing, it is normal to give commands like `move forward at 20 kph`. A robot would then proceed to move at exactly 20 kph (within the tolerance of its machinery).

Often though, our natural language precludes such strict tolerance. A phrase like `move forward at` *about* `20 kph` would be more natural, and while imposing the constraint the the robot travel at *exactly* 20 kph still satisfies the *about 20 kph* constraint, it is an artificial imposition.

Fuzzy logic attempts to eliminate this artificial intrusion with the use of gradual

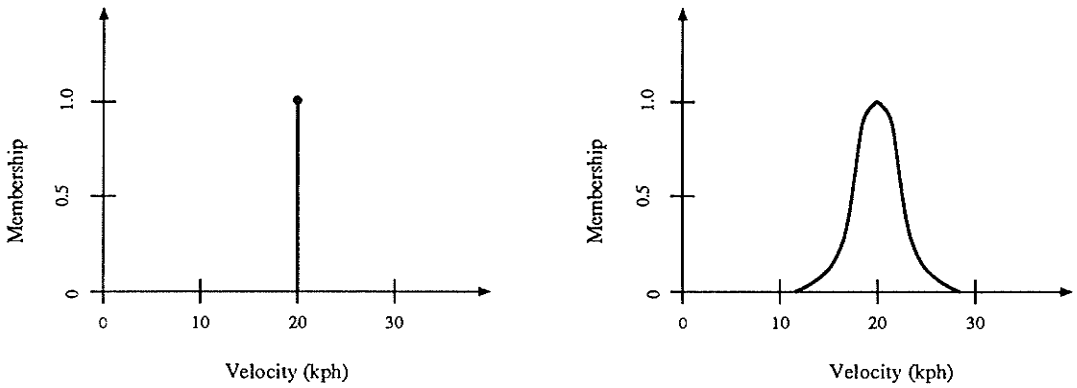

Figure 2.1: **Crisp and Fuzzy Membership in the Robot Velocity Problem**

membership functions in the place of *crisp* numbers. So in the fuzzy paradigm, a two valued *yes-or-no* response is replaced with a gradual membership indicating the degree of belongingness to a concept. The difference in paradigms is displayed in Figure 2.1.

In these figures, a '1' ranking on the membership axis is meant to suggest total belongingness to the concept. A '0' ranking suggests total exclusion. A ranking somewhere in the interval [0,1] suggests partial belongingness. To illustrate this concept, examine the fuzzy set presented as Figure 2.1b. If a robot were to travel at 18 kph, this would under many circumstances be acceptable to the constraint *about* 20 kph. A high degree of membership is therefore assigned. If the robot travels at 10 kph however, this is unacceptable since it is not *about* 20 kph.

This yields two important facts about this kind of approach. The first is that

membership is extremely subjective. A value that one person may class as *about* 20 kph (i.e. 15 kph), another might consider outside of the range of acceptable values. It can be said that the general trend of the function remains the same regardless of the specific interpretation.

The second observation is that the fuzzy set is context dependent. The fuzzy set established by the term *about* 20 kph is related to tolerances determined by the environment.

A mathematics of such fuzzy sets has been developed over the last few years. Zadeh established the concept in 1965 [44], and for many years little work was associated with it. A revival occurred in the mid 1980's and today fuzzy logic is a sort of technological revolution in Japan. The use of this technique has even filtered down from the research laboratory into many consumer products.

Fuzzy set theory has been used mainly to handle uncertainty and imprecision for situations in which conventional techniques perform poorly.

Three kinds of imprecision have been isolated, [2] namely generality (multiplicity of objects), ambiguity (context-dependency) and vagueness (imprecise boundaries). Of these, fuzzy set theory handles generality in some situations and handles vagueness extremely well. Vague terms such as *rich, wise, fast*, etc. may be encoded rather easily into the kinds of membership functions shown in Figure 2.1b.

2.1.2 Mathematical Overview

Now that a familiarity has been established with the notion of a fuzzy set, a discussion of the mathematics involved in the fuzzy cognitive system may proceed. This section

comprises only a tiny segment of fuzzy relational mathematics, and therefore the reader is referred to [14] [24] for more indepth readings.

Basic operators involved in the types of relations involved are the t-norm, represented as $\textcircled{t}$, the s-norm, represented as $\textcircled{s}$, and fuzzy negation, represented as $\textcircled{n}$.

The t-norms and s-norms may be described as any functions satisfying the criteria of Table 2.1, where $a, a^1, b, b^1, c \epsilon [0,1]$.

	t-norms	s-norms
Satisfying Conditions	$[0,1] \times [0,1] = [0,1]$	
i) Boundary Conditions	$a\textcircled{t}0 = 0, a\textcircled{t}1 = a$	$a\textcircled{s}0 = a, a\textcircled{s}1 = 1$
ii) Commutativity	$a\textcircled{t}b = b\textcircled{t}a$	$a\textcircled{s}b = b\textcircled{s}a$
iii) Associativity	$a\textcircled{t}(b\textcircled{t}c) = (a\textcircled{t}b)\textcircled{t}c = a\textcircled{t}b\textcircled{t}c$	$a\textcircled{s}(b\textcircled{s}c) = (a\textcircled{s}b)\textcircled{s}c = a\textcircled{s}b\textcircled{s}c$
iv) Monotonicity	For $a^1 < a, b^1 < b$	
	$a^1\textcircled{t}b^1 \leq a\textcircled{t}b$	$a^1\textcircled{s}b^1 \leq a\textcircled{s}b$

Table 2.1: **Conditions for t-norms and s-norms**

These norms are related by the expression:

$$a\textcircled{s}b = 1 - (1-a)\textcircled{t}(1-b) \tag{2.1}$$

which is simply a form of De Morgan's theorem.

Fuzzy negation may be interpreted as a type of complementation.

$$a^{\textcircled{n}} = 1 - a \tag{2.2}$$

The Fuzzy Matching Operator

An expression comparing two fuzzy sets may be established on the basis of logic and set theory. Consider two fuzzy sets A and B at the same element of the universe of

discourse, $a = A(x), b = B(x)$. The concept that a *is equal to* b may be expressed as $a \equiv b$ (the *matching* operator), and may be represented logically as

$$a \textit{ is contained in } b \textbf{ and } b \textit{ is contained in } a$$

Now representing the **and** conjunction as a t-norm and containment as a pseudo-complementary operation, ψ, the expression for equality index becomes:

$$a \equiv b = (a \psi b) \textcircled{t} (b \psi a) \tag{2.3}$$

A similar expression may be created involving s-norms and another pseudocomplementary operation, β to implement the logical concept

$$a \textit{ is not contained in } b \textbf{ or } b \textit{ is not contained in } a$$

$$(a \beta b) \textcircled{s} (b \beta a) \tag{2.4}$$

Complementing this expression yields another equality index.

$$a \equiv b = 1 - (a \beta b) \textcircled{s} (b \beta a) \tag{2.5}$$

A more robust matching operation may be formed by incorporating both t-norm *and* s-norm equality expressions.

$$a \equiv b = \frac{[(a \psi b) \textcircled{t} (b \psi a)] + 1 - [(a \beta b) \textcircled{s} (b \beta a)]}{2} \tag{2.6}$$

This may then be modified to

$$a \equiv b = \frac{[(a \psi b) \textcircled{t} (b \psi a)] + \{[(1-a) \psi (1-b)] \textcircled{t} [(1-b) \psi (1-a)]\}}{2} \tag{2.7}$$

A *minimum* function, $\wedge$, may be used for t-norms and a *maximum* function, $\vee$, may be used as the s-norms. If a Gödelian implication is used for the psudocomplements,

$$a \rightarrow b = \begin{cases} 1, & \text{if } a \leq b \\ b, & \text{if } a > b \end{cases} \tag{2.8}$$

then the matching expression of equation (2.7) becomes

$$a \equiv b = \frac{[(a \rightarrow b) \wedge (b \rightarrow a)] + \{[(1-a) \rightarrow (1-b)] \wedge [(1-b) \rightarrow (1-a)]\}}{2} \tag{2.9}$$

It can be proven that

$$(a \rightarrow b) \wedge (b \rightarrow a) = \begin{cases} a \wedge b, & \text{if } a \neq b \\ 1, & \text{if } a = b \end{cases} \tag{2.10}$$

$$[(1-a) \rightarrow (1-b)] \wedge [(1-b) \rightarrow (1-a)] = \begin{cases} 1-(a \vee b), & \text{if } a \neq b \\ 1, & \text{if } a = b \end{cases} \tag{2.11}$$

which allows equation (2.9) to become

$$a \equiv b = \begin{cases} \frac{(a \wedge b)+1-(a \vee b)}{2}, & \text{if } a \neq b \\ 1, & \text{if } a = b \end{cases} \tag{2.12}$$

Since we would like to implement a digital system, the [0,1] interval must be remapped onto another discrete space, [0,M], yielding

$$a \equiv b = \begin{cases} \frac{(a \wedge b)+M-(a \vee b)}{2}, & \text{if } a \neq b \\ M, & \text{if } a = b \end{cases} \tag{2.13}$$

Assuming a four-bit resolution, yielding M=15 and only natural numbers used in the discrete space, this leads to the function depicted in Figure 2.2. About half of the resolution is now *wasted*, since the only numbers used in this discrete space are $a \equiv b \leq 7$, or $a \equiv b = 15$. The [8,14] interval is unused. Greater resolution could be *simulated* in the system if this interval were used. It would therefore be advantageous in the digital system to implement the function shown in Figure 2.3, since it fully utilizes the resolution of the [0,15] space.

This function may be easily implemented by multiplying the first condition of equation (2.13) by two, yielding a continuous function.

$$a \equiv b = (a \wedge b) + M - (a \vee b) \tag{2.14}$$

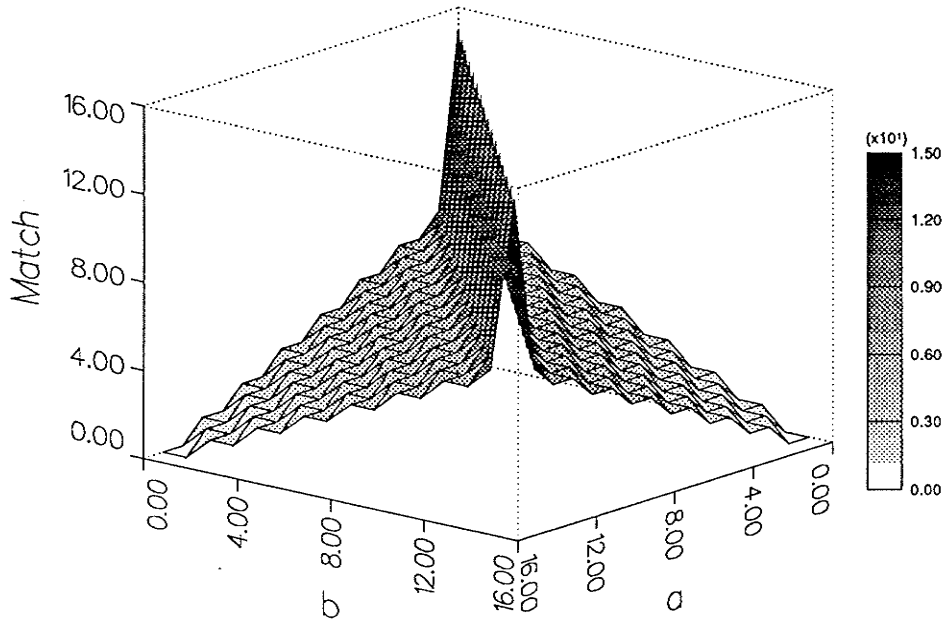

Figure 2.2: **The Fuzzy Matching Function**

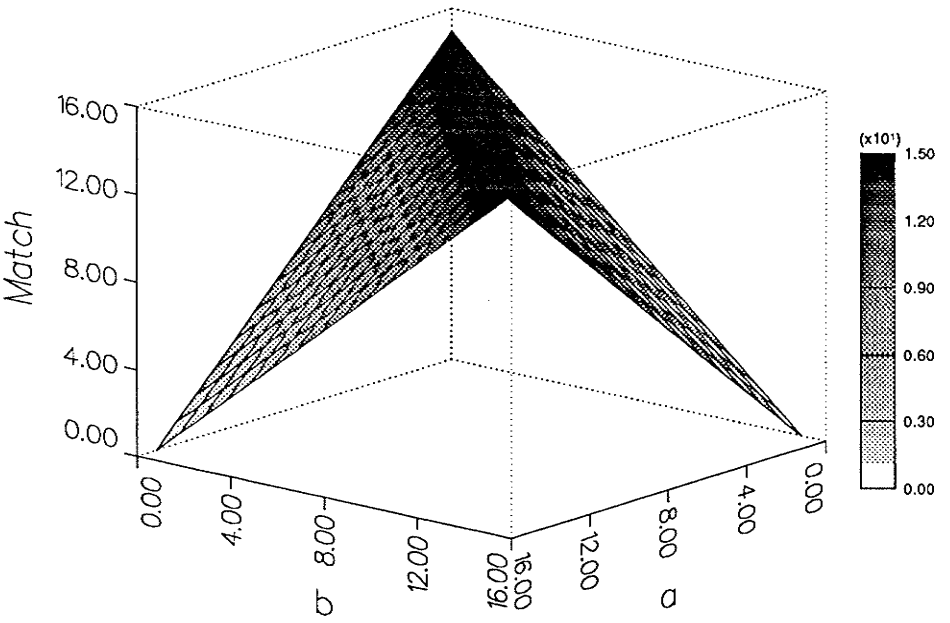

Figure 2.3: **The Fuzzy Matching Function with Data Remapping**

A quick examination of these functions reveals that they experience maxima at the point where $a = b$. The least amount of the matching functions occur when a and b are farthest apart (i.e., a=15, b=0, MATCH=0).

The Inverse Fuzzy Matching Operator

In the situation where the degree of matching between two fuzzy quantities is known as well as one of the quantities, an inverse matching operation is required to generate the second fuzzy quantity.

Since more that one value may be generated as the output of such an inverse function, further analysis must be undertaken to resolve the obvious contradictions that may arise.

For our purpose, we are interested in functions which satisfy the matching criteria, that is, all points which lie under the surface of Figure 2.3. Equation (2.13) may be rewritten (only the $a \neq b$ condition will be analyzed) as

$$Q = a \equiv b \leq \frac{(\min + M - \max)}{2} \tag{2.15}$$

where $\min = a \wedge b$, and $\max = a \vee b$. With some manipulation, this yields,

$$\begin{aligned} 2Q - M &\leq \min - \max \\ M - 2Q &\geq \underbrace{\max - \min}_{\geq 0} \end{aligned} \tag{2.16}$$

We are given either min or max, so now examine each situation individually.

I) max is given, min is unknown.

$$2Q - M + \max \leq \min$$

$$\Rightarrow \min \geq \max + 2Q - M \tag{2.17}$$

II) **min is given, max is unknown.**

$$\begin{aligned} 2Q - M - \min &\leq -\max \\ \Rightarrow \max &\leq \min -(2Q - M) \end{aligned} \tag{2.18}$$

For the general case, where it is unknown in advance whether the given term is the maximum or minimum,

$$a = b \pm (2Q - M) \quad a, b \epsilon [0, 1] \tag{2.19}$$

defines the interval into which the unknown value falls.

For the case of data remapping as in equation (2.14), the Q multiplier drops off to become

$$a = b \pm (Q - M) \quad a, b \epsilon [0, 1] \tag{2.20}$$

The nature of the interval created by the inverse-matching function may be best understood with an example. The scenario is the data compressed matching function of Figure 2.3 with a four-bit [0,15] discrete resolution. Take the case where

$$Q = a \equiv b = 12 \qquad \text{and} \qquad b = 8$$

A vertical plane is constructed, as shown in Figure 2.4.

It is visually obvious that an interval ranging from 5 to 11 is created, and this may be mathematically derived from equation (2.20)

$$\begin{aligned} a = b \pm (Q - M) &= 8 \pm (12 - 15) \\ &= 8 \pm (-3) \\ &\Rightarrow [5, 11] \end{aligned}$$

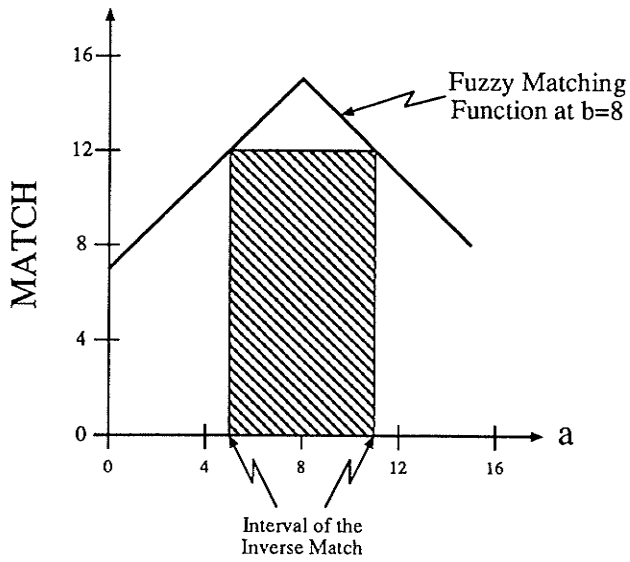

Figure 2.4: **Example of Inverse Matching**

It will be seen in later sections that the nature of this interval provides a very robust qualitative *and* quantitative output to the fuzzy cognitive system.

2.2 Neural Networks

2.2.1 The Purpose of Neural Networks

Conventional computers simply cannot achieve many things that we desire of them, and so an obvious comparison is often made to the human brain. While conventional Von Neumann and Harvard computer architectures have proved themselves extremely useful for both scientific research and easing the burdens of the average life in general, they have some evident shortcomings. It has been estimated that it would take over 1000 super computers to do the kind of real-time visual processing accomplished by the Darwinianly optimized brain.

On the other hand, the human brain has perhaps $\frac{1}{10}^{\text{th}}$ the power to do purely mathematical calculations when compared to a simple pocket calculator. The brain is simply not built to perform these kinds of calculations with high speed and precision.

Another startling comparison may be made with respect to memory. Brains forget or distort many facts that are labeled 'IMPORTANT'. Computers seldom if ever forget vital facts. Brains, on the other hand, excel at completing incomplete facts, even when some of the information given is erroneous. Conventional computers handle this kind of problem very poorly.

Obviously the two machines are built for different purposes, and perform poorly on problems outside of their scope of expertise. We would, however, like computers to handle many of the tasks that humans take for granted such as image processing, recognition, and associative memorization. It is for this reason that the quest for a more *brain-like* computer began.

Background

Artificial neural network (ANN) models have a long and spurious history. More than 4 decades ago, [21], a number of theories began to evolve regarding representations of biological neural connection systems in mathematics. A great deal of interest accumulated into the 1950's and 60's, until the *new wave* of artificial intelligence (AI) began to dominate the 1970's. The less than anticipated results generated from these physical symbol systems [22] brought a resurgence of connectionism in the 1980's

The connectionist attempt has been concentrated in two areas. The first is the understanding of how biological neurons interrelate. Figure 2.5 gives a very simple

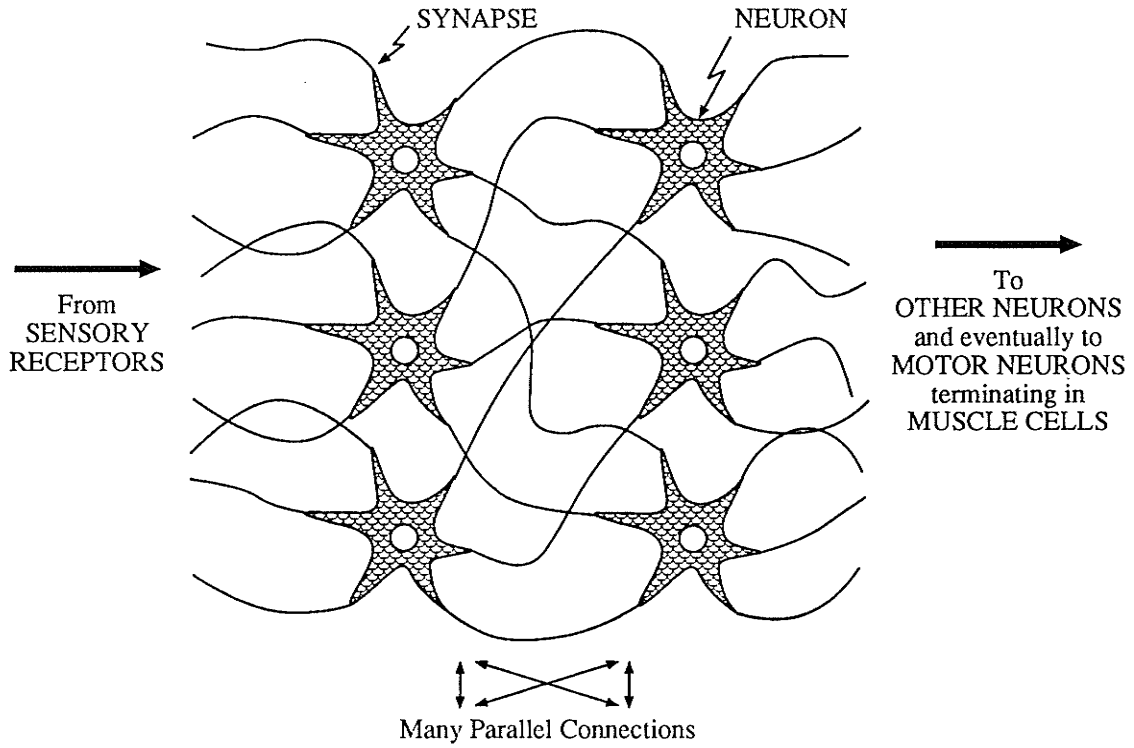

Figure 2.5: **A Simple Depiction of Biological Neurons**

illustration of the neural connections. Biological neural networks utilize massive parallelism to accomplish high speed computation. This massive parallelism includes not only connections to subsequent layers, but also connections within a given layer and feedback to previous layers. For simplicity, this thesis will examine only feed-forward networks (no inter-layer connections or feedback).

The neuron is often modeled as shown in Figure 2.6. A weighted summation is used to model the biological neurons chemical activity. A nonlinearity is used at the output to constrain the network values. Some thresholding functions classically used are shown in Figure 2.7.

Several methods may be used to implement the functions of an ANN, including

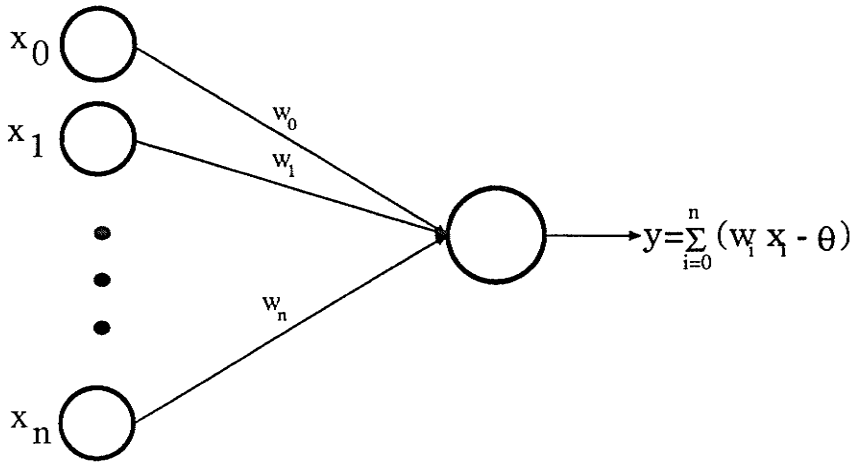

Figure 2.6: **Artificial Neural Model**

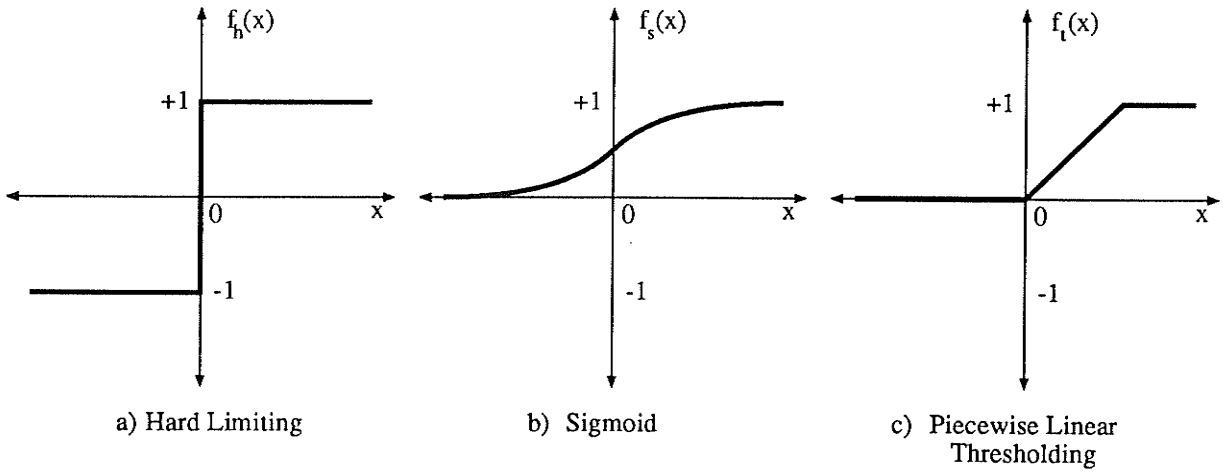

Figure 2.7: **Thresholding Functions**

fully digital, fully analog or hybrid (some combination of digital and analog) methodologies. While an analog design would be fast and consume very little space, design time is prohibitive. Digital design is very area intensive and much slower than analog systems, however design time is minimal due to the great body of knowledge that has been accumulated throughout the computer revolution. Furthermore, since existing computer hardware utilizes almost exclusively digital technology, an analog neural network must use some sort of digital interface to communicate. Hybrid designs are very attractive, since they may apply the advantageous features of both analog and digital design. Unfortunately design time is again prohibitive, since the interfacing of digital and analog parts is not a trivial task.

The hard limiter shown in Figure 2.7a is used in a simple device called a perceptron. When Minsky and Papert wrote their famous book *Perceptrons*, [21], they pointed out some limitations of single layer perceptron networks (such as the XOR problem). For the single layer network, linear separability is a crucial issue. This problem was later 'solved' with the back-propagation model of learning, also known as the multilayered perceptron network. In this type of network, several layers of neurons may be used to capture high order statistics.

The back-propagation algorithm is so named since learning is accomplished through the back-propagation of the error derivative from the output to the input. Because derivatives are required, continuous functions such as the sigmoid of Figure 2.7b are used, but often approximated by the piecewise-linear thresholding function of Figure 2.7c. In effect, back-propagation combines a nonlinear perceptron-like network with gradient descent optimization to achieve a minimum global error, given enough neurons. It is difficult to foresee how many neurons are sufficient for a given problem.

A lack of convergence to the desired output space is often attributed to searching a large plateau for minima. Many thousands of learning epochs may pass without any noticeable change in convergence if this occurs. Appendix A illustrates an example of the back-propagation algorithm and discusses some methods of improvement.

Because back-propagation *works* it has become the most popular learning method, and many software models exist to simulate performance. Since it is generally quite slow to learn, other methods have been investigated.

One of the most interesting alternate methods is Hebbian learning. With this technique, correlations between neurons strengthen the connection between them, while anticorrelations weaken connection strength.

Regardless of the learning method used, neural networks are computational structures that are not programmed in a conventional sense. They *learn* the required statistics, and can thereafter perform real-time computations in a parallel distributed fashion.

2.2.2 Limitations and Possibilities

Although there is a tremendous research effort in neural networks and many claims regarding their capabilities, application areas are still experimental. The following discussion contrasts neural networks with more traditional AI techniques. Similar comparisons can be made in other fields where neural networks have the potential to outperform more conventional computations (i.e., pattern recognition).

Neural networks are not nearly as well understood as the sequential symbol manipulators of the 1970's AI endeavors. Classical AI attempts were limited in their

heuristic approach to the problem at hand. That is, if a program was specifically written to play chess, it was extremely difficult to modify it to handle vision problems, for example. On the other hand, neural networks are of such a general nature that a given network may be used to learn a multitude of different problems.

While the heuristics involved in a symbol-manipulating AI attempt are extremely complex to code and perfect (sometimes requiring years), the heuristics involved in a neural network are self-generated. The user of a neural network does not (necessarily) need to utilize expertise on the subject at hand, as the 'program' (connections) used is generated by the hardware from an examination of the problem. This lack of a formal symbolic language is one of connectionism's greatest advantages and downfalls.

Ideally a conventional AI solution involves enough heuristics to approximate an algorithmic solution. In this way the optimal (or near optimal) solution is consistently found. Problems unfortunately occur in the combinatorial explosions formed by extremely large searches (ie: chess cannot be played algorithmically - there are too many combinations to search). Heuristics used for this type of system are inherently plagued by unexpected situation that the 'rules of thumb' supplied by the programmer do not cover. (If extremely good heuristics are intensively programmed by the skilled operator, the program will operate within its built-in limiting constraints.) On the other hand, the heuristics formed by the neural network will consistently generate an optimized solution (although perhaps not globally minimized), provided enough neurons are available for the problem's complexity. Furthermore, the generalization abilities of neural networks allow formulation of new 'rules of the game' as they become necessary. So not only might neural networks solve the problem of knowledge access in an environment of combinatorial explosion with content-addressable mem-

ories, but also extrapolate from those memories in unpredicted situations.

Perhaps the greatest advantage of neural networks is this ability to learn, generalize and extract increasingly high levels of classification. These abilities form important distinctions with classical AI (Searle's Chinese room thought experiment is quite confounded by the absence of a central administrator), and have provided extremely encouraging preliminary results.

An additional advantage of the neural network approach is its use of parallelism. A classic problem in distributed processing is getting it to be well *distributed.* Not only do neural networks solve this dilemma extremely well, but performance also degrades gracefully as neurons become faulty.

A major downfall of neural networks (and particularly back-propagation) is its slow speed at learning new tasks and generating meaningful statistics. A conventional programming approach relies on already-trained human networks to provide these. Alternately, other networks (particularly Hopfield networks) are very slow to produce correct output (despite their speed at learning).

It has been argued that physical symbol manipulation attempts have reached a 'brick wall' because they are looking at the wrong level of knowledge required to do some fundamental problems. While this may also be true for connectionist systems, this science has not yet reached as high a level of sophistication, and it is therefore much more difficult to judge.

With the very complementary abilities of these two approaches, it would be extremely attractive to combine the two. One possible approach might be to share silicon. The conventional digital von Neumann architecture would oversee the work-

ings of an analog neural network. The neural network would then become another tool in the computer scientist's bag-of-tricks. At this time it is difficult to predict the role that neural networks will play in computation. It is, however, clear that they are a powerful computational paradigm that will become more prevalent in the future.

2.3 VLSI

VLSI, or Very Large Scale Integration, is an acronym that was spawned from the computer revolution. It suggests a silicon die on which there are approaching 100,000 gates. None of the devices designed in this thesis are close to this level of complexity, but clearly illustrate the potential of VLSI for artificial neural network implementations. The term VLSI has evolved to suggest large, complex Application Specific Integrated Circuits (ASICs).

VLSI methodology is a science in itself. An attempt must be made to minimize area and power dissipation while maximizing performance and flexibility. At all times an attempt is made to minimize routing length since long lines add capacitance to the system thereby reducing speed. When long lines or large loads are required, *fan-out* and *drive* must be considered. Additional drivers added to the circuit at precise points can be used to increase speed.

The ability to optimize the many constraints involved in ASIC design is a skill requiring many years to achieve proficiency. While the author in no way claims proficiency, an attempt was made to adhere to good design techniques.

Outside of the requirements already mentioned, an attempt was made to incorporate a Built-In Self Test (BIST) and fault tolerance in all of the designs. BIST

allows each chip to verify its own operation before being inserted into a circuit board. Error-free operation is ensured, therefore reducing the cost of troubleshooting the system once it has left the production facility. Fault tolerance is used to increase product life by including some sort of redundancy to ensure that the device may be damaged yet still perform its function. These considerations are considered in more detail in the next chapter of this thesis.

The advantages of ASICs over off-the-shelf technologies are indispensable to those who require them. The most important advantage for the purpose of this research is speed. Where real-time computations are necessary, only an ASIC can provide maximal performance. A second advantage is size. A customized IC reduces the space and therefore the complexity of Printed Circuit Boards (PCBs). A third advantage is that of the proprietary information contained on the chip. An ASIC is much more difficult to copy than a PCB covered with commercially available parts, and this is often an important consideration for industry. Two disadvantages of ASICs are cost and design time. It is simply less expensive and less time consuming to assemble prefabricated devices into a PCB.

Of the plethora of commercial technologies available for microelectronics, one of the most common is the Complementary Metal Oxide Semiconductor (CMOS). In this process, both PMOS and NMOS structures are constructed, usually as duals. While the redundancy requires approximately double the area of either PMOS or NMOS, the power dissipation of the *complementary* technology is remarkably low (power is only dissipated at switching).

Several techniques are available for decreasing the size of CMOS such as pseudo-

NMOS, dynamic CMOS, clocked CMOS, CMOS domino and cascode voltage switch logic. Since the University of Manitoba standard cell library was readily available, fully static complementary CMOS logic was used for all designs with the exception of the occasional transmission gate used for multiplexing. While most of these cells were created by others [5] several have been modified for various reasons. Still other cells have been created from scratch, such as a static random access memory cell used to estimate memory capacity in the given technology.

The fully digital route was taken due to its speed of design, low power dissipation and ease of interfacing with more common digital devices such as personal computers. This also infers that some of the calculations that were not essential on-chip could be accomplished in software.

Fabrication of all devices was through the Canadian Microelectronics Corporation's 3μm double-level-metal CMOS technology. This was the most complex technology available at the onset of this research.

Through the course of this research, many software aids have been used for design and simulation. Electric, a hierarchical layout tool [30], APLSIM (APL SIMulator), an interactive analog simulator [31], and BSIM, a switch level simulator [20] were extensively used tools. Two of the three chips developed in this research were totally hand placed and routed. This technique was found to be extremely time consuming and tedious, although at the time there was no alternative.

Initially, a Very high speed application specific integrated circuit Hardware Description Language (VHDL) [11] [1] with Queen's UnIversity Silicon Compiler (QUISC) [15] were used to speed up the design of regular structures by eliminating some of the

tedious manual routing. While a structural definition was used to design these blocks, it was found to be awkward and unsatisfactory for larger sections of the design. It is anticipated that a behavioral description would greatly simplify the design process, and that VHDL will become the design methodology of choice in the near future.

The design of the neural network would not have been possible without a high level schematic capture system like Cadence [32]. The speed of this tool allowed experimentation with different implementation styles that are simply too time consuming with hand-placed and routed systems. Since one problem with automatic place and route is possible poor performance due to long lines, performance was enhanced with a hierarchical place and route of the larger blocks in the neural network. This technique ensured maximal routing length of only one neuron (as opposed to the entire die size).

2.4 System Overview

Now that the main components of theory surrounding this thesis have been introduced, a more indepth explanation of the system structure and operation will be presented.

The fuzzy cognitive system may be viewed as the multiple stage structure presented as Figure 2.8.

The system consists of:

I) Matching (in the input space, followed by aggregation)

II) Transformation (of the matched input space to the output space)

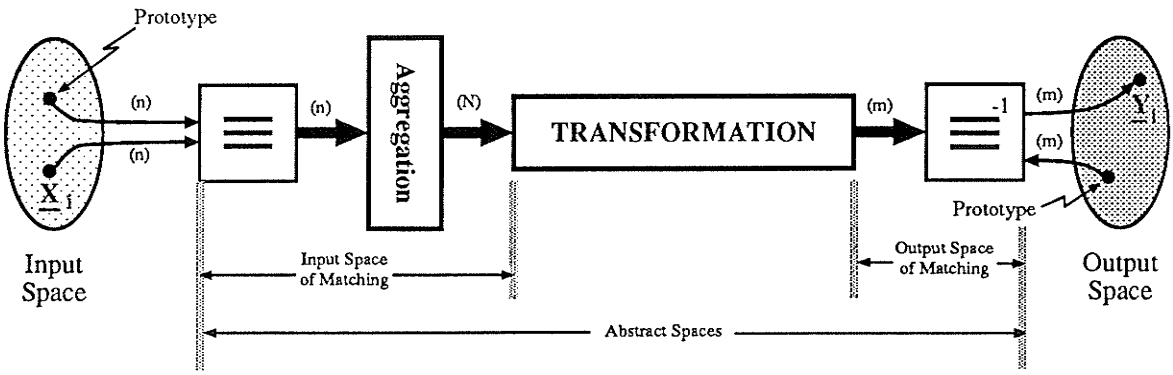

Figure 2.8: **Structure of the Fuzzy Cognitive System**

III) Inverse matching (in the output space)

The structure possesses a referential character, since it does not work with directly input information. The input data (called the objective) is compared to a prototype (reference) description in the input space. This results in a vector of degrees of equality having the same dimension as the object in the input space.

The matching (as introduced previously) in a broad sense of the word, is a process of comparison of successive coordinates of two objects (patterns). Denote by $A = [a_1 a_2 \cdots a_n]$, and $X = [x_1 x_2 \cdots x_n]$ the objects matched where lower case letters represent corresponding coordinates to be matched, $a_i, x_i \epsilon [0,1]$. The matching procedure returns a vector of results of comparison realized coordinatewise, $a_i \equiv x_i$, where $\equiv$ represents the matching operator. The matching block converts data from the physical space, in which the objects are characterized, into an abstract space representing degrees of matching achieved at the input space.

These values are then compressed, arranged into a vector form and sent to the inputs of the transformation block. Aggregation may be desirable to reduce the

dimensionality of the problem ($N << n$). It is not, however, necessary to implement the structure, rather it is used to compress the incoming data.

At the next stage of processing, the vector of the degrees of equality in the input space of matching is transformed to the relevant vector in the output space of matching. Transformation could be accomplished with special purpose hardware, but this is highly dependent on the specific problem under investigation. For the sake of generality and flexibility, the transformation block is implemented as a neural network with 'N' inputs and 'm' outputs. For the sake of simplicity, the basic network being implemented is a single-layer with a threshold element. This single layer may be used in conjunction with subsequent chips to produce additional layers or may be used to cycle upon itself, creating iterative layers.

After transforming the input equality vector, the relevant vector in the output space is calculated with the inverse matching operation (as described previously). This procedure results in a range of values given by an upper and lower limit. The larger the range, the less exact one can be in finding the exact result within the interval. If the range is very small it is much simpler to estimate the exact value that generated the matching value. If there is no interval (i.e., the upper and lower limits are identical), the crisp case is generated. Because the interval gap is a measure of imprecision in the system it is referred to as an *interval of confidence.*

Since a very general structure is used to perform the transformation, a supervised learning phase must be associated with the system. This phase is depicted in Figure 2.9.

Since both values of the objective and prototype are known in the output space,

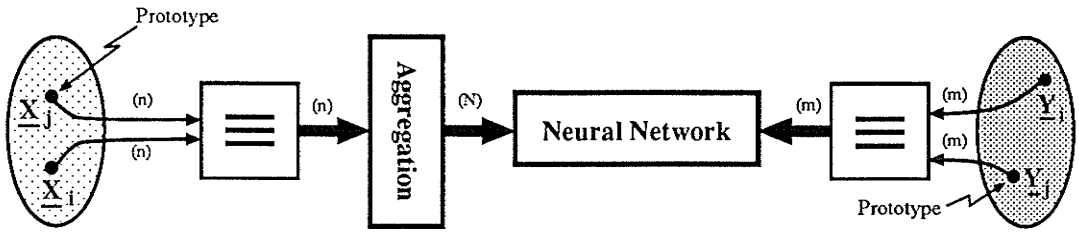

Figure 2.9: **Learning Phase**

the inverse matching block of Figure 2.8 is now replaced by a matching block. The values are generated from both ends and appear to the transformation block as data at the input and output. The purpose of this learning phase is to obtain the unique mapping from input vector (N) to output vector(m). Once this mapping has been learned, it may be stored in the form of system weights, and recalled to perform the calculations for that specific problem as needed. The learning may be performed in a number of ways including the back-propagation algorithm.

Several constraints must be placed on the fuzzy cognitive system to implement it in hardware. Figure 2.10 shows the structure of the system implementation.

The first major constraint is resolution. Since the system is digital, it was necessary to assign some number of bits to the input, output and intermediate blocks. A uniform resolution of four bits was decided upon. This decision was primarily based on implementation concerns such as area and preliminary system simulations with fixed-bit arithmetic. From a VLSI standpoint, it would be possible to prove the validity of design methodologies, while still having the ability to implement functional devices.

The second constraint was silicon area, and it was decided to break the system into three separate chips. This would ensure that if an alteration was required on

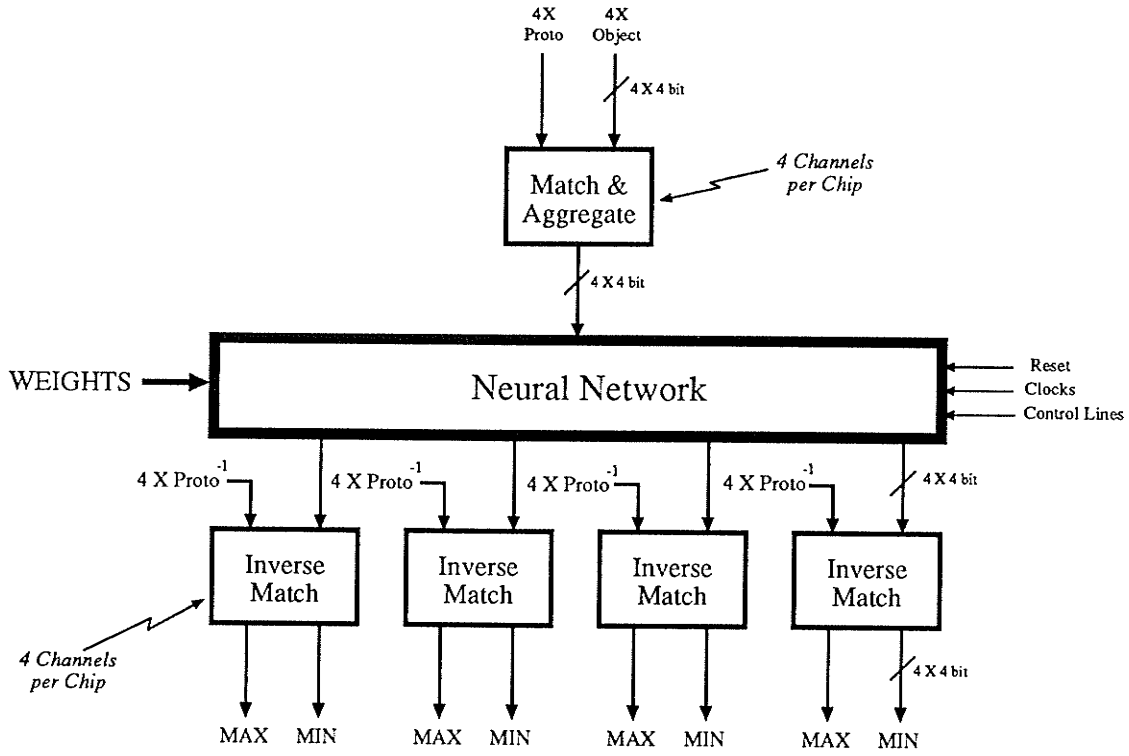

Figure 2.10: **Hardware Overview**

one of the stages of operation, only that chip need be redesigned. VLSI limitations also enforced that the neural network would be a massive undertaking on its own, requiring a very large die size to fabricate even a simple network. The three chip system would consist of a matching chip with optional aggregation, a reconfigurable neural network chip and an inverse matching chip.

It was also decided that 4 sets of objectives/prototypes would be allowed as inputs and that the output would have sixteen discrete points of resolution (i.e., a vector of 16 sets of upper/lower bounds). To investigate the workings of the system, a simulation program was written. The code from this program and an example are presented in Appendix B.

Chapter 3

System Implementation

3.1 Enhancements to the Implementation

3.1.1 The Fault Tolerant Methodology

Fault tolerance is a technique used to ensure reliability of performance in a system [29][12]. Normally some form of redundancy is used to maintain system integrity in the event of some failure, be it transient, intermittent or permanent. The failure could take the form of a flaw in material processing when the device is fabricated, or some unforeseen circumstances while in operation such as an alpha strike or voltage surge.

In the constraints of the fuzzy cognitive system, three different techniques of fault tolerance were attempted (one for each chip designed).

On the first one, the matching chip, triple modular redundancy (TMR) with voting was used. In this scenario, three processors each independently performs identical calculations. The outputs of these sections are then compared and a *two-out-of-three* approach taken. If one of the processing elements is faulty, the other two units

would overrule it at the voting stage. In this way the system may suffer a fault in one unit without consequence to the resulting calculations. In this error-masking scheme, it may be advantageous to produce information when one of the processors is consistently found to be at fault. The information may then be used to judge how long the system will continue before total failure.

No attempt was made in the matching chip to indicate time-to-failure. Additionally, each redundant block would ideally be a unique hardware implementation, meaning that the same calculation is performed several times in different ways. Coded checking (such as residues) is one commonly used scheme [3]. All of the redundant blocks are identical on this chip since time constraints on the design did not allow experimentation of this kind.

The second chip designed, used for inverse matching, utilizes a different scheme. In this case, duplex redundancy (DR) with error detection was used. Two identical blocks were used to compute the inverse matching output, and these results were compared. If they **do not** agree, one output line changes indicating that an error has been detected. At this stage, the user may choose to replace the part completely, or ignore the error flag and take the output as valid (a very dangerous decision). Regardless of the action, steps may be taken to ensure acceptable performance. The main reason for choosing this less acceptable form of fault tolerance is complexity considerations. The inverse matching operation is much more computationally demanding (and therefore more area intensive) than the matching operation. The DR scheme utilizes only $\frac{2}{3}$ of the area of the previous scheme, and furthermore the comparison equipment at the output is less complex in this technique.

The final scheme used for fault tolerance comes from the nature of parallel computations in a neural network. Fault tolerance of neural networks is a relatively new field of study, although a few attempts have been made [7].

It has long been known that a biological systems degrade gracefully as neurons become faulty (i.e., brain damage). Since the computation is both highly parallel and distributed, a few bad neurons do not totally disable the complete system. In our system in which 16 neurons will feed 16 neurons, if one neuron produced an erroneous result it is then passed on to the next layer. At this point, a weighted summation is taken of all neurons on the previous layer and one of the 16 values in this summation is in err. In a worst case scenario, this means that the total is correct 15 parts out of 16. While this is tolerable, it reduces for larger networks. The built-in self tolerance is a very attractive feature of neural networks.

The training process may also reduce the effect of faults by recovering from them intelligently. The working neuron's weights may be compensated to lessen the impact of the faulty areas. While it would also be advantageous to make fault tolerant neurons, the demands were found to be too area intensive for this application.

An interesting scenario for fault tolerance will be examined after the neural network system structure has been presented.

3.1.2 The Built-In Self Test Methodology

Built-in self test (BIST) is a technique used to reduce the difficulty of testing [39]. Random test vectors are generated on chip and presented to the circuit where the output is then compressed and compared to known correct code. A signal is presented

to the user indicating the success or failure of the test.

BIST eliminates the need for bulky, expensive test equipment and is potentially much faster, since computations are accomplished on-chip.

The attractive nature of using cellular automata (CA) for BIST has been well documented [10] [28]. This structure is simply a special case of a one dimensional neural network, in which only next-neighbor connections are made. Under constrained conditions, these networks will specific generate all 2^n possible combinations of an n-bit system in a pseudorandom order. CA's are a very good structure for VLSI since they require only localized connections, are simple to implement and offer good fault coverage.

Rule 90/150 hybrid CA's were used in all three chips to perform BIST. The derivation tables for rules 90 and 150 are shown in Table 3.1, and these functions can be easily implemented with one (rule 90) or two (rule 150) exclusive OR gates. In this table, i_{t+1} represents the temporal evolution of a single automaton, i, connected to nearest neighbors (i-1) and (i+1).

The CA's in all chips were two-bits larger than actually required for the test, with the two outer bits ignored (and null boundary conditions). This overestimation approach was taken to increase fault coverage (the vectors used in the test are *more random*) at the expense of nominal silicon area.

The first two chips designed (matching and inverse matching) take advantage of the redundant channels. The technique used involved creating pseudorandom numbers with the CA, propagating the same random number through each channel in parallel, and comparing the outputs. A discrepancy in any of the outputs sends a

$(i-1)_t$	i_t	$(i+1)_t$	Rule 90 i_{t+1}	Rule 150 i_{t+1}
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	1
1	0	1	0	0
1	1	0	1	0
1	1	1	0	1

Table 3.1: **Cellular Automata Rules 90 and 150**

signal to the user, inviting further testing on individual channels.

The neural network chip also uses the pseudorandom numbers to cycle through a chain of neurons, comparing all neurons at the output (they should have identical responses). Discrepancies indicate further testing is required to isolate the damaged neuron.

3.2 External Processing from the Neural Network

3.2.1 Preprocessing on the Neural Network

The matching concept of equations (2.13) and (2.14) may be implemented as shown in Figure 3.1. For division-by-two, the least significant bit (LSB) is simply discarded. An option was included to use either of the two equations. The block was created with ripple-carry adders, inverters and multiplexors.

Another option was created to include aggregation. Aggregation was implemented with the barrel-shift adder configuration shown in Figure 3.2. The technique simply keeps a running total of inputs, ignoring the least significant bits. For a resolution of

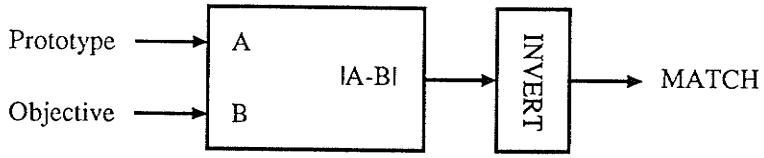

Figure 3.1: **Fuzzy Matching Implementation**

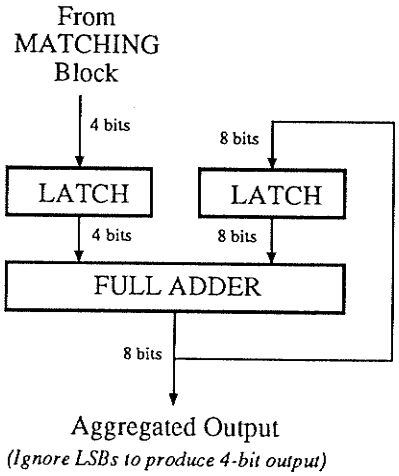

Figure 3.2: **Implementation of the Aggregation Option**

16 points in the input space, the aggregating function would sum 16 numbers coming from the matching block, and divide the result by 16 (ignore lower 4 LSBs), essentially averaging the vector.

The completed implementation is presented as Figure 3.3. A graphical description of the blocks in the design is presented as Figure 3.4. Some specifications of this chip are presented as Table 3.2.

Four identical independent channels are implemented on the chip, and these are used as comparisons against each other in the BIST.

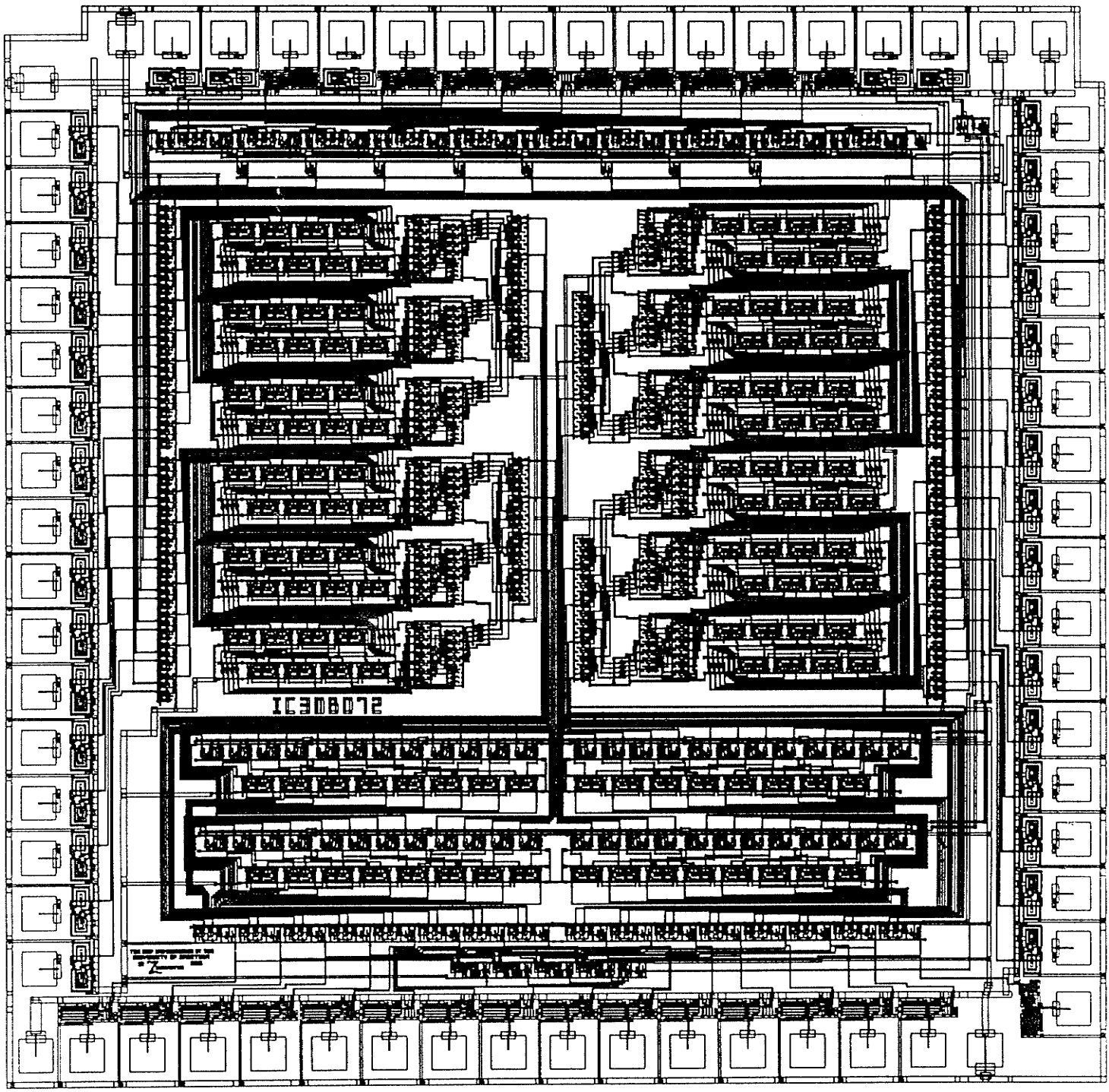

Figure 3.3: Matching Chip with Aggregation

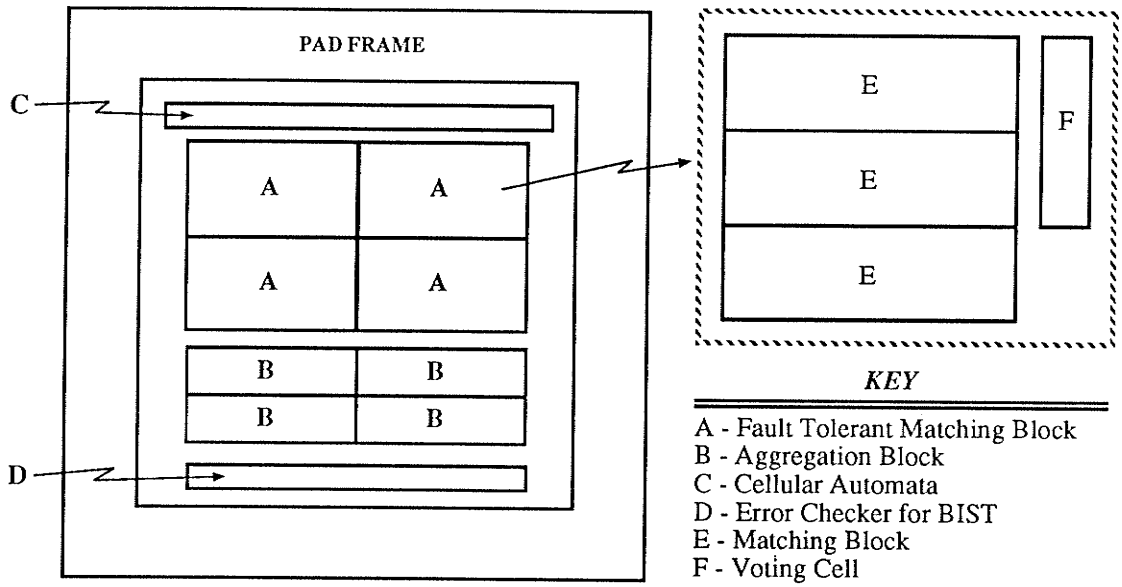

Figure 3.4: **Breakdown of the Matching Chip**

Total Chip Size†	4503 × 4503
Number of Pins	68
Input Pins	32
Output Pins	16
Devoted Test Pins	5

Table 3.2: **Matching Chip Specifications**

† in 3 μm CMOS

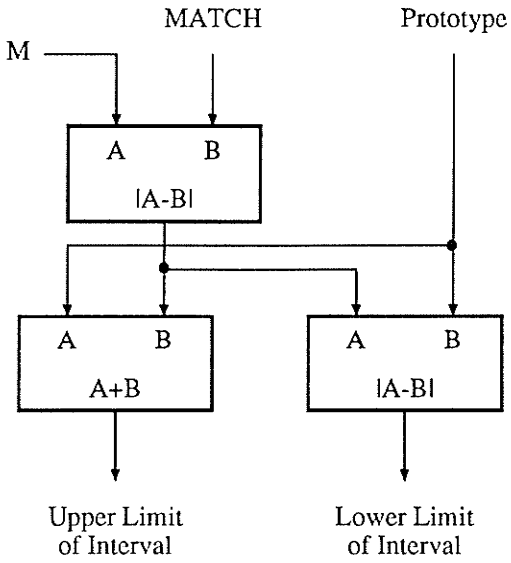

Figure 3.5: **Inverse Matching Implementation**

3.2.2 Postprocessing on the Neural Network

The inverse matching function of equations (2.19) and (2.20) may be implemented as shown in Figure 3.5. An option for multiplication-by-two is accomplished with simple shifting.

This block is obviously more complex than the matching operation, and it is justifiably more area intensive. Part of this complexity is derived from the function's dual output nature. Simple ripple-carry adders, inverters and multiplexors were again used for the implementation. The completed implementation is presented as Figure 3.6. A graphical description of the blocks in the design is presented as Figure 3.7. Some specifications of this chip are presented as Table 3.3.

Four identical independent channels are again implemented on the chip, and these are used as comparisons against each other in the BIST.

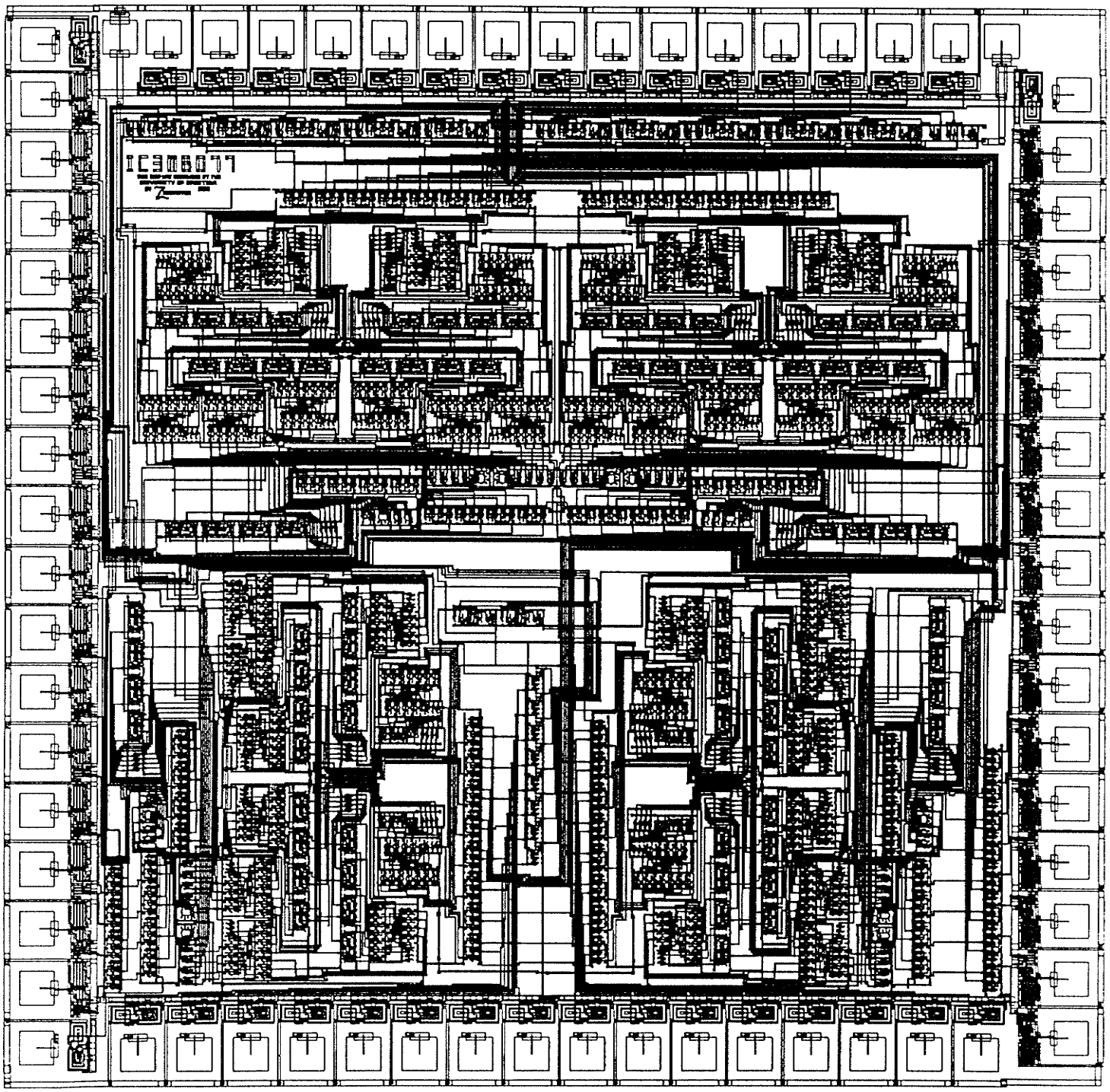

Figure 3.6: **Inverse Matching Chip**

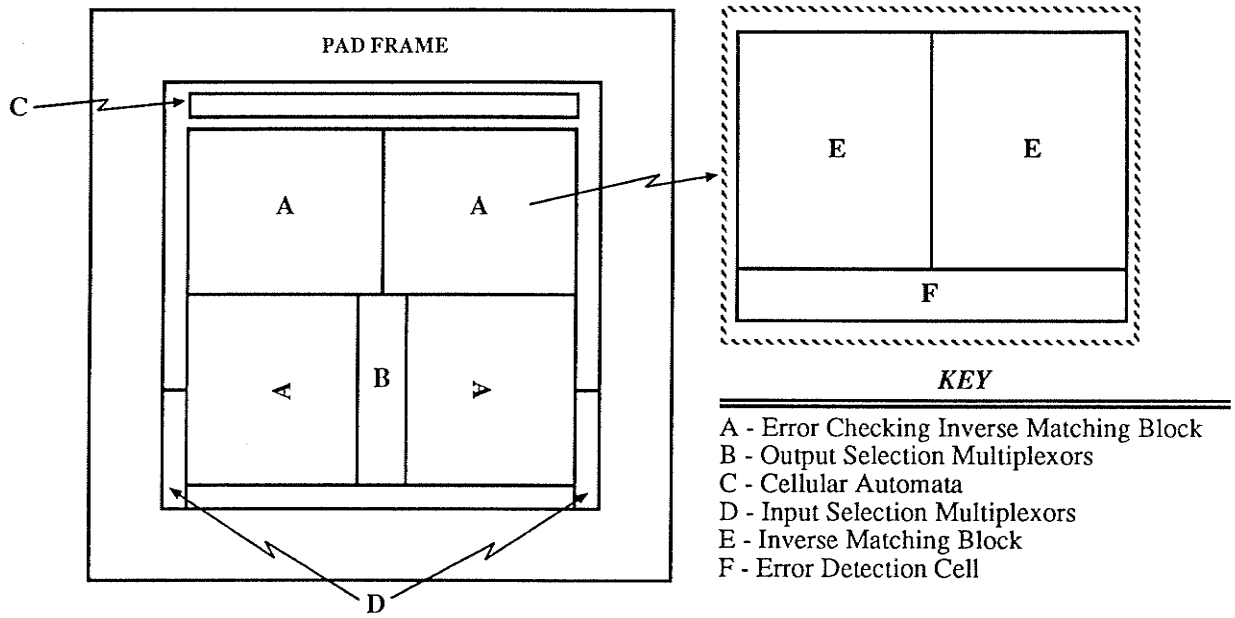

Figure 3.7: **Breakdown of the Inverse Matching Chip**

Total Chip Size†	4503 × 4503
Number of Pins	68
Input Pins	32
Output Pins	32
Devoted Test Pins	5

Table 3.3: **Inverse Matching Chip Specifications**

† in 3 μm CMOS

3.3 Neural Processing

The most complex part of the fuzzy cognitive system is the neural network. Many identical neurons transmit their information to many other identical neurons, and the immense parallelism is very difficult to implement in VLSI. The more neurons on a single chip, the less complex the neurons must be, The more serialized the computation, the smaller the neuron, and therefore more may be place on a single chip. Each serialized neuron is now slower than a more parallel implementation.

In an attempt to find an optimized solution, a compromise was arrived at: parallel computation would be performed with a distributed serial system. While each neuron performs a pipelined serial operation, many neurons perform in parallel, thereby maximizing the computing potential and distribution.

3.3.1 Neural Architecture

Since the number of available neurons on the chip was uncertain at the onset of this research, it was decided that the synapses and neuron would be designed as a single unit. Based on this fact, the term *neuron* may be used to infer both the neural and synaptic structures.

It was decided at an early stage to use a three-bit resolution for the weights since this was a complementary resolution to the four-bit activations. Other schemes, including a one-bit weighting algorithm were attempted but discarded. The single-bit weighting scheme has a very narrow field of use and would require many, many neurons to accomplish any task of practical use.

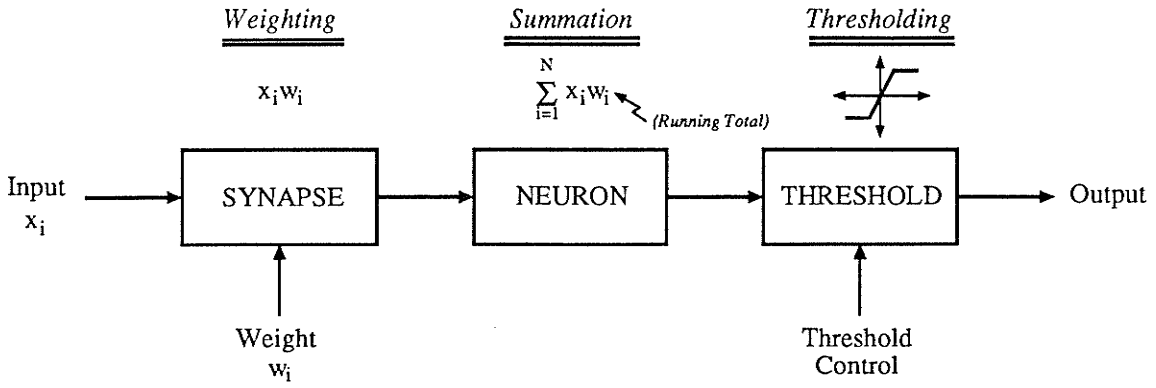

Figure 3.8: **General Architecture for Neural Processing**

The basic conceptual architecture of Figure 3.8 was used to implement the neuron. A form of bounded multiplication is performed at the synapse, and a running bounded accumulation is used at the neuron itself. At the thresholding stage, the neural output is forced through a piecewise linear thresholding function with variable slope. A threshold was used instead of a bias term since it gives more control of the neural output.

If a 4×4 full multiplier were used, few neurons could be placed on the chip. Since multiplication is extremely area intensive in VLSI, a simplified multiplication scheme was used, as shown in Table 3.4.

The simplification of multiplication/division by factors of 2 yields a tremendous simplification in terms of hardware complexity since it may be implemented by base-2 shifting. For example, the number $(0110)_2 = (6)_{10}$ may be multiplied by a factor 2 by shifting the bits once to the left, as $(1100)_2 = (12)_{10}$. The original number may be halved by shifting once to the right, as $(0011)_2 = (3)_{10}$.

At the thresholding stage, the neural output is forced through a piecewise lin-

a_2	a_1	a_0	Multiplier
1	1	1	-2
1	1	0	-1
1	0	1	$-\frac{1}{2}$
1	0	0	-0
0	0	0	$+0$
0	0	1	$+\frac{1}{2}$
0	1	0	$+1$
0	1	1	$+2$

Table 3.4: **Simplified Synaptic Scheme**

a_2	a_1	a_0	Results
0	0	0	ZERO
0	0	1	/2
0	1	0	$\times 2$
0	1	1	$-\frac{1}{2}$MAX
1	0	0	$+\frac{1}{2}$MAX
1	0	1	/4
1	1	0	$\times 4$
1	1	1	MAX

Table 3.5: **Thresholding Scheme**

ear thresholding function with variable slope. A three-bit resolution was used for threshold control, and the scheme implemented is presented as Table 3.5. Options are included to either totally disconnect the neuron (output ZERO) or turn the output always 'on' (output MAX). The $\pm\frac{1}{2}$MAX option could be used to induce a more extreme, *hard limiting* output.

Another source of area saving may be found in the adder-tree associated with a fully parallel implementation, shown in Figure 3.9. Figures 3.10-3.12 show some attempts at pipelining the operation.

The results of this investigation are summarized in Table 3.6. While speed is

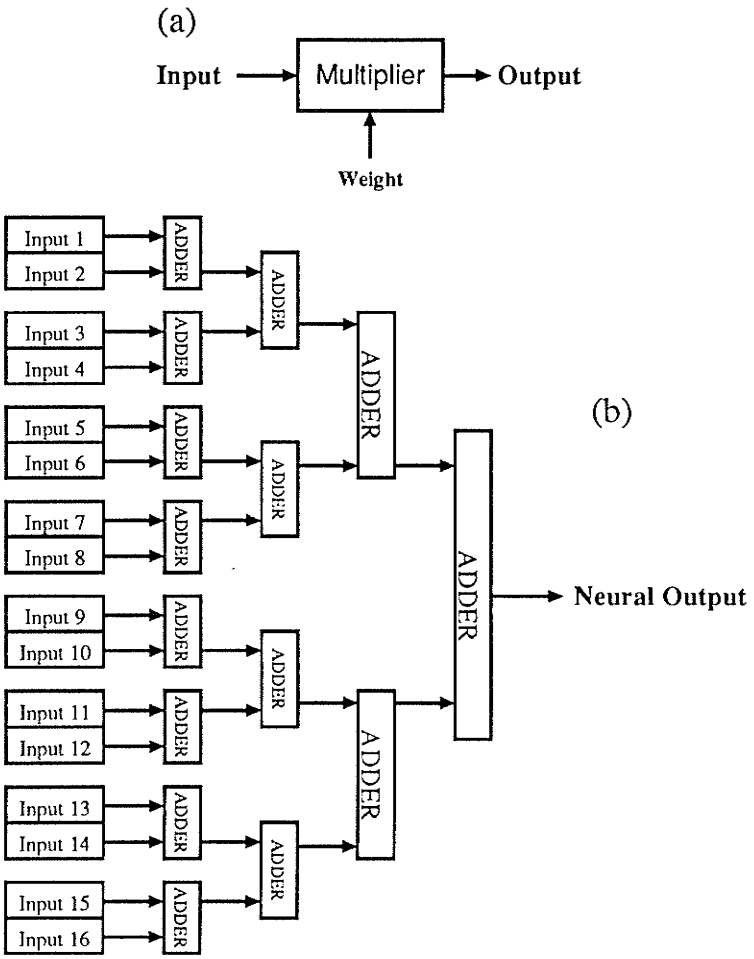

Figure 3.9: Parallel Neural Architecture (Unpipelined)

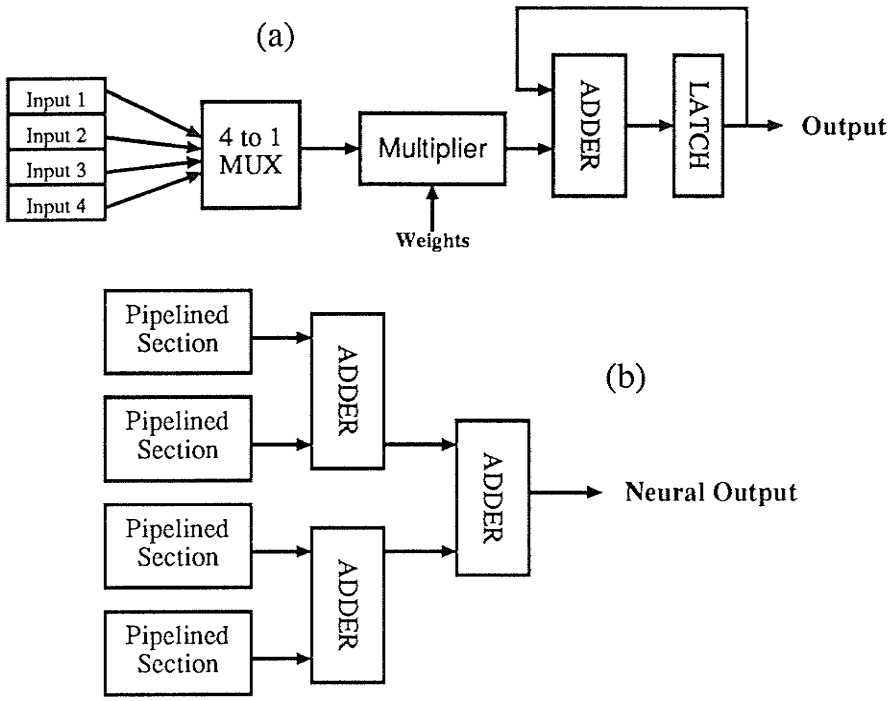

Figure 3.10: **Level 2 Pipelined Neural Architecture**

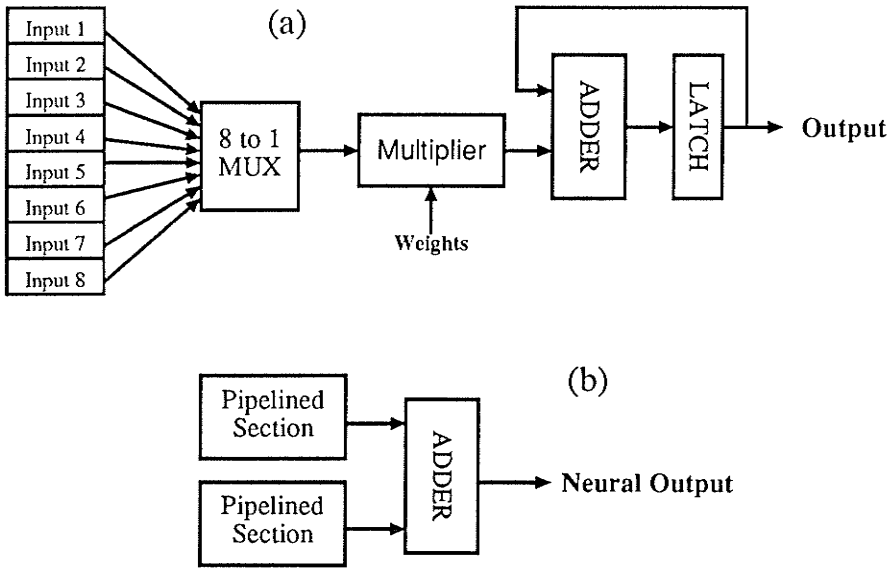

Figure 3.11: **Level 3 Pipelined Neural Architecture**

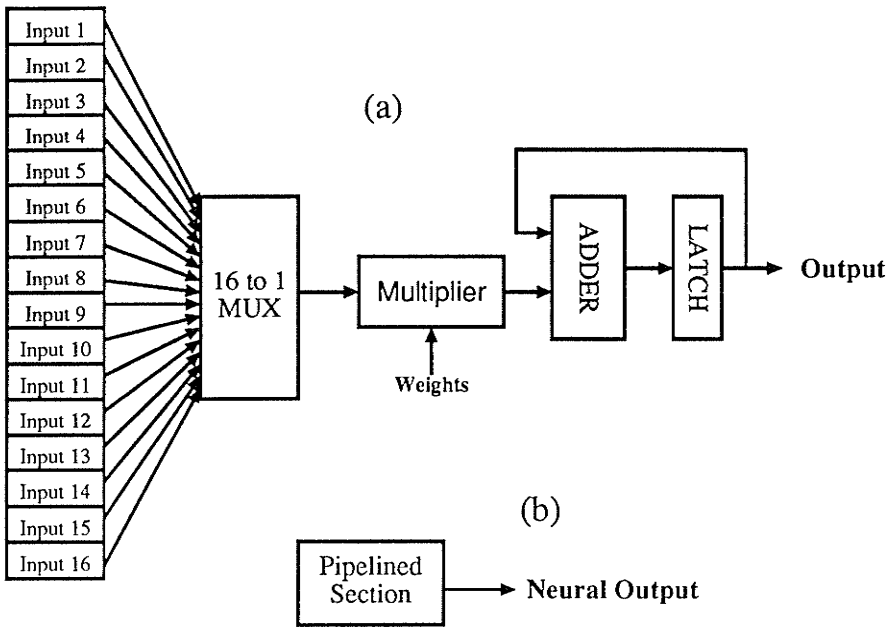

Figure 3.12: **Level 4 Pipelined Neural Architecture**

reduced, size is **greatly** reduced.

After several months of careful study, it was decided that a serial processing neuron would be ideal for VLSI implementation. Since it was desired that 16 neurons would be implemented on a chip, the most serialized network would be used (level 4). It should be noted that while the level 3 estimate in Table 3.6 indicates that 18 of these

Levels Pipelined	Area of Neuron ($\mu m^2 \times 10^6$)	Clock Cycles per Neural Cycle	Neurons per Chip†
0	21.29	1	4
2	8.44	4	11
3	5.36	8	18
4	4.76	16	26

Table 3.6: **Trade-offs in Neural Pipelining**

† Assuming $96.48 \times 10^6 \mu m^2$ maximum chip size.

neurons could be placed on a chip, this does not account for interconnectivity, the pad frame area or any equipment used for testability.

Some estimations were also done to see how much memory (SRAM) would fit on our maximum allowable die size. A reasonable approximation would be 7-8 kilobits of data. For one layer, we require:

		#bits		#neurons		#per neuron		Total #bits
Weights	→	3	×	16	×	16	=	768
Threshold	→	3	×	16	×	1	=	48
Activation	→	4	×	16	×	1	=	64
								880

Since a reasonable sized neural network has 3 layers, this becomes $3 \times 880 = 2640$ bits of information. This would require approximately $\frac{1}{3}^{\text{rd}}$ of the full die size. With 16 neurons on the chip **and** some testability, it was deemed unwise to hold the weights and threshold controls on-chip. Given this fact, it was totally unrealizable to perform learning computations on-chip. It would be necessary to develop a system architecture which would allow fast access to both weights and threshold values.

3.3.2 System Architecture

At the system level several trade-offs were required. After several months of careful study, it was decided that a serial processing neuron would be ideal for VLSI implementation. While several engineers have attempted to implement digital neural networks in VLSI, most of the attempts have been very regimented, that is, a specific network was envisioned, and that precise network was implemented.

A technique was discovered for greatly increasing flexibility and processing speed while greatly decreasing power consumption and size. It seems so trivial and beneficial

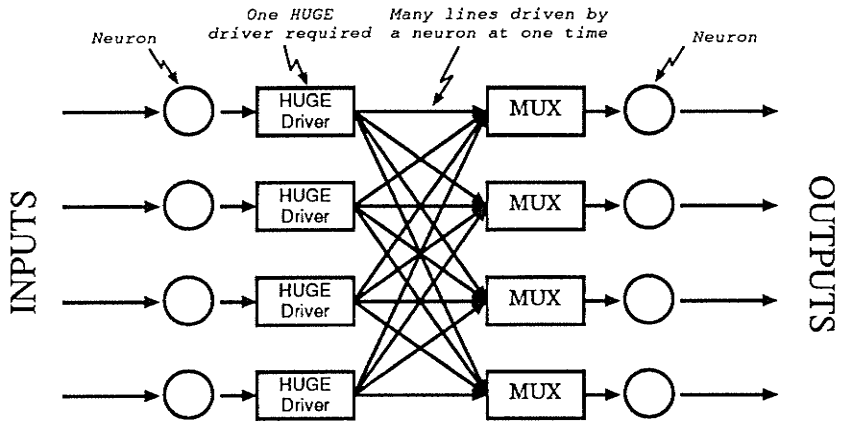

Figure 3.13: **Configuration of a Feed-Forward Network**

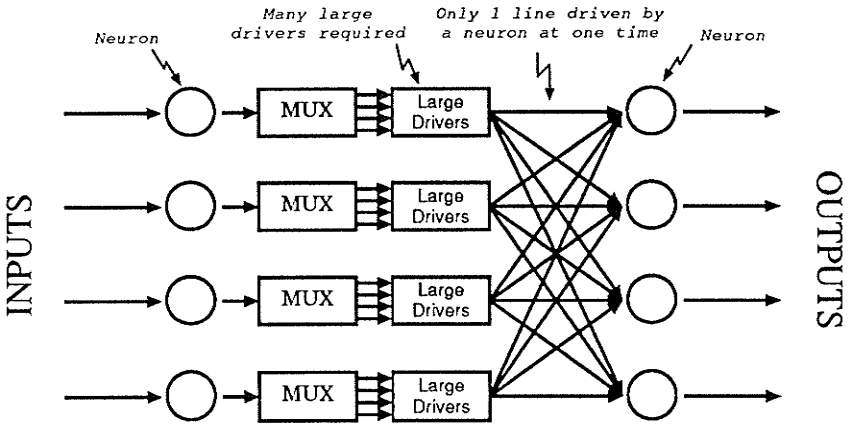

Figure 3.14: **Another Configuration of a Feed-Forward Network**

that it is not directly obvious why this solution has not previously been used, yet it has not.

Figures 3.13 and 3.14 show standard configurations of forward propagating neural networks. This technique is perhaps the most often used for implementations [43] since it is straightforward.

Most implementation attempts have utilized multiplexors at either the input or output of the neuron to allow serial processing in the neurons at each layer. This

requires that a physical connection from each neuron on each layer to each neuron on the next layer be present. Therefore, for a layer of 16 neurons feeding a layer of 16 other neurons, 256 physical connections must be present. This high degree of interconnection greatly reduces neuron complexity, since only a specific area of silicon is allowed for an implementation. Large drivers must also be used for each of these lines, as they may be propagating signals as far as the diagonal of the chip. If these drivers are not used, speed will be greatly decreased, and so an area/speed trade-off is encountered. The greater the distance a signal must travel, the larger the driver should be. In parallel systems, this is usually difficult to foresee without meticulous planning and accuracy. The solution is to use fairly large drivers, in the hope that signals that are only required to travel a short distance are overly driven, while only the very longest lines are slightly underdriven. The result is mediocre performance, and medium size.

Furthermore, if the network is to be reconfigured, the multiplexor block must be reconfigured and replaced in each neuron. For example, if 17 neurons (instead of 16 on the previous layer) are required on a given layer, all multiplexors for the new layer must be changed to a 17-to-1 (rather than 16-to-1). While this increases size (the new multiplexors are inherently larger) and slows computation (more logic implies slower response), the major problem is that of reconfigurability for the designer. The new multiplexors must be design from scratch, and replaced in the specific neurons that require it. Furthermore, once fabricated, there is **no** reconfigurability whatsoever. The structure of the network simply *cannot* be changed at this stage (i.e., no more neurons may be added on the layer).

Another scheme that has been used [7][42] is the bus architecture shown in Figure

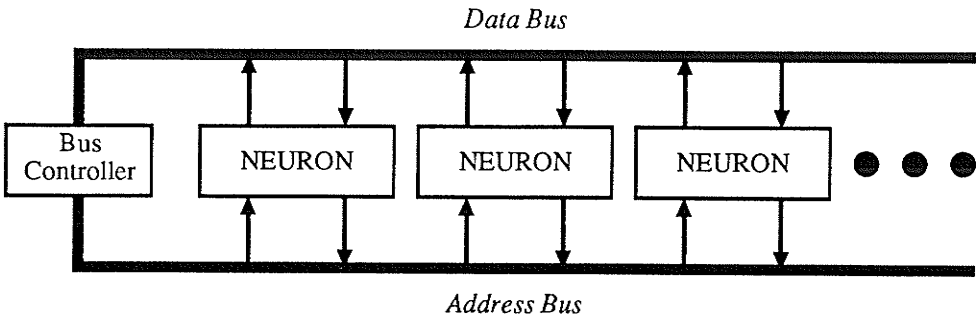

Figure 3.15: **Bus Architecture for a Digital Neural Network**

3.15. This architectural system is taken from the often-used von Neumann single processor concept of using one bus to access many peripherals. It has been adapted for use in parallel systems with interesting consequences. It allows any number of computational elements to use the bus, and even to access devices off-chip that are given access to the bus. The long lines require very large drivers for realistic speed. The single communication channel also becomes a bottleneck when communications become intensive. While a multibus structure alleviates some of this problem, a multitude of bus arbitrators and controllers becomes necessary. This in turn makes it difficult to design efficient structures in VLSI.

In contrast to the parallel and bus systems, consider the architecture of Figure 3.16. In this configuration, each neuron is fed one value of the input vector. The neuron processes the datum and passes it to the next neuron in the chain (of the same layer), which in turn processes the piece of information. Each neuron maintains a running total for its cumulative weighted summation. This process continues until all neurons on the layer have processed each piece of information from the previous layer. The cyclic chain ensures that each neuron on the layer receives every piece of information. To add one neuron to a particular layer, the chain for that layer must

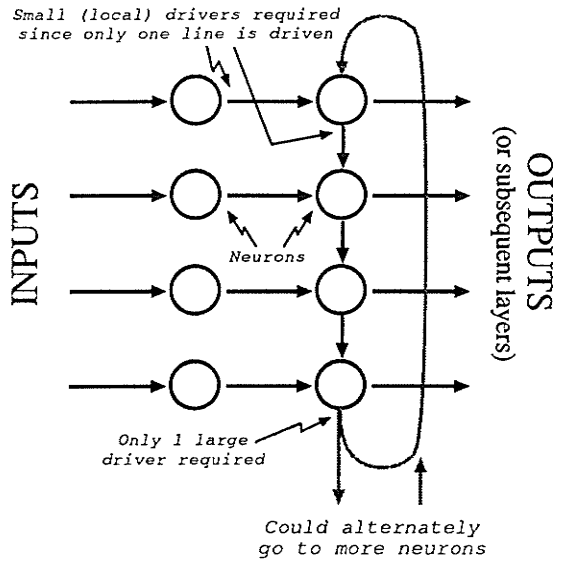

Figure 3.16: **A Neural-Slice Feed-Forward Network**

be expanded by one 'link'. Note that neither the neuron itself nor any of it's parts need be redesigned. The addition of a link increases processing time by one *neural cycle*, that is, the number of clock cycles required to process one piece of information at one neuron.

Each neuron feeds only the 'next neighbor', thereby using local connections. Only small drivers are needed for this, since the distance between two neurons may be specified at the *placement* stage of design. Speed is also improved because only a predictably small driver is required for the very short line. Only one line from each neuron to its nearest neighbor is required, thereby eliminating $256 - 16 = 240$ lines, several of which may run the diagonal length of the chip.

Although this technique greatly reduces area, its greatest advantage is reconfigurability. Several chips may be interfaced together as required to create a network with any number of neurons on a layer.

Iterative techniques may be used to cycle the chip upon itself. By collecting the data at each neuron until all data for the layer has been processed, the neuron is then ready to transmit its information to the subsequent layer of neurons. The information may then be transmitted to another chip, or returned to its own inputs to process further layers. An infinite number of layers on a single chip may be created in this way. Alternately, several chips may be used to pipeline layers.

Obviously a very flexible system has been developed, in which pipelined neurons feed pipelined layers and the network consists of any number of neurons on any number of layers.

Built-In Self Test for the Neural Network

With this architecture, BIST works as follows. First, pseudorandom test vectors are produced by the CA. Next, a single vector is propagated through the shift register's chain system to the activation/weight/threshold inputs for each neuron. Another vector is produced and cycled through the system. After an acceptable number of test vectors have been sent through the system, the outputs of all neuron are compared for discrepancies.

Again the inherent system redundancy is used to better the self test. No data compression is required since it is assumed that the number of correctly functioning neurons outnumbers the faulty ones.

A concept not actually implemented on the test chip involves BIST with fault tolerance. As the network is processing its application, it could use every $1{,}000^{\text{th}}$ or $10{,}000^{\text{th}}$ calculation to run a short BIST. If a given neuron is found to be consistently

in err, it may be **removed from the chain**. This effectively trims the network of faulty processors as they become unusable. The system is then quite robust and resistant to faults.

3.3.3 Implementation Issues

A chip was designed using the aforementioned design principles, and a metal layer representation is presented as Figure 3.17. A graphical description of the blocks in the design is presented as Figure 3.18. The upper-level schematic that was used to generate the chip is presented as Figure 3.19. Some specifications of this chip are presented as Table 3.7. Padframe requirements forced serial loading and unloading of the shift register chain, although parallel off-chip communication is trivial task given more compact pads.

At the time of implementation it was felt that there might be some advantage keeping the shift register as an autonomous entity. It seems more practical, upon reflection, to embed the shift register elements within each neuron. This would become more desirable if the number of neurons on a chip were increased.

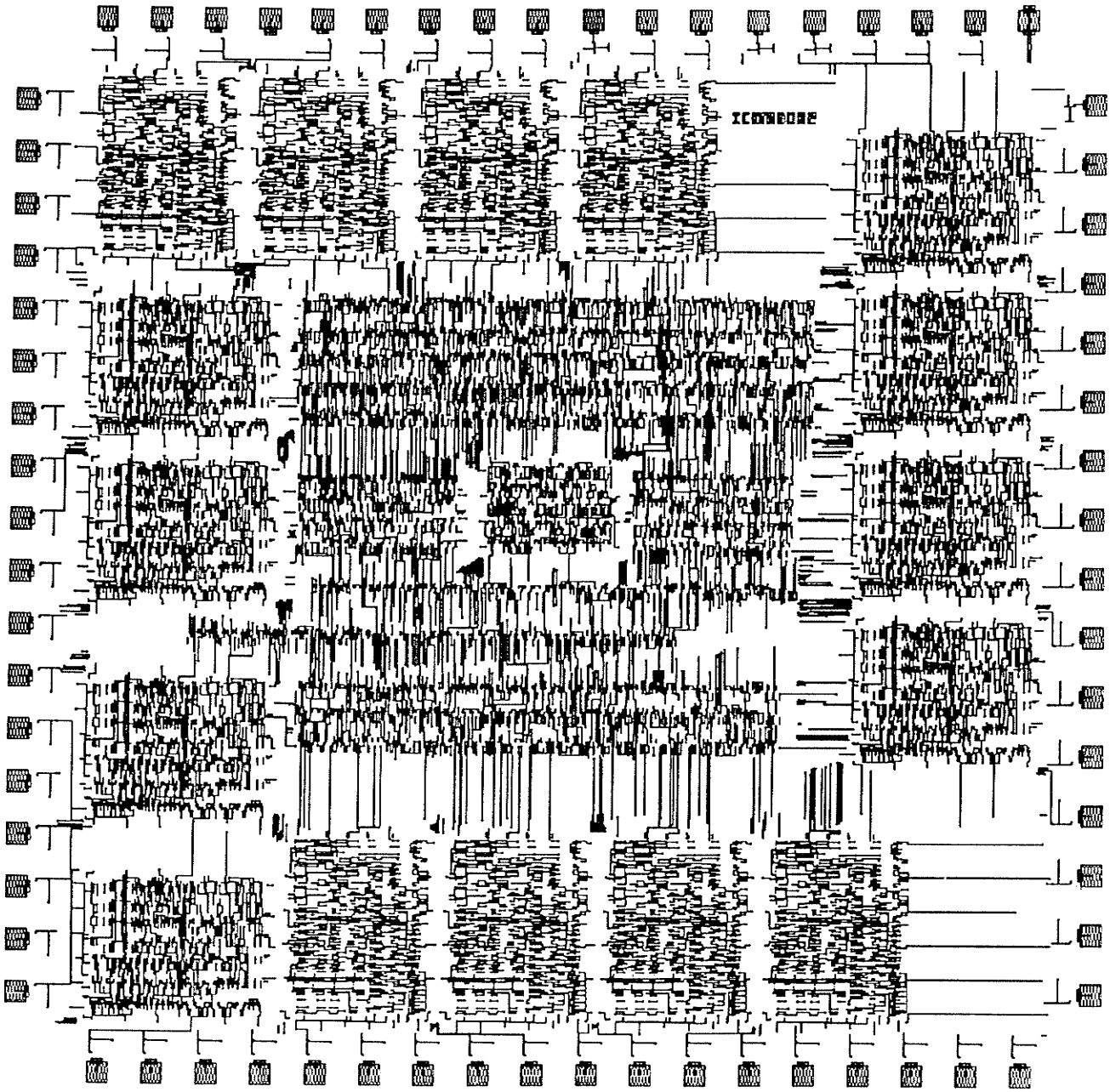

Figure 3.17: Neural Network Chip

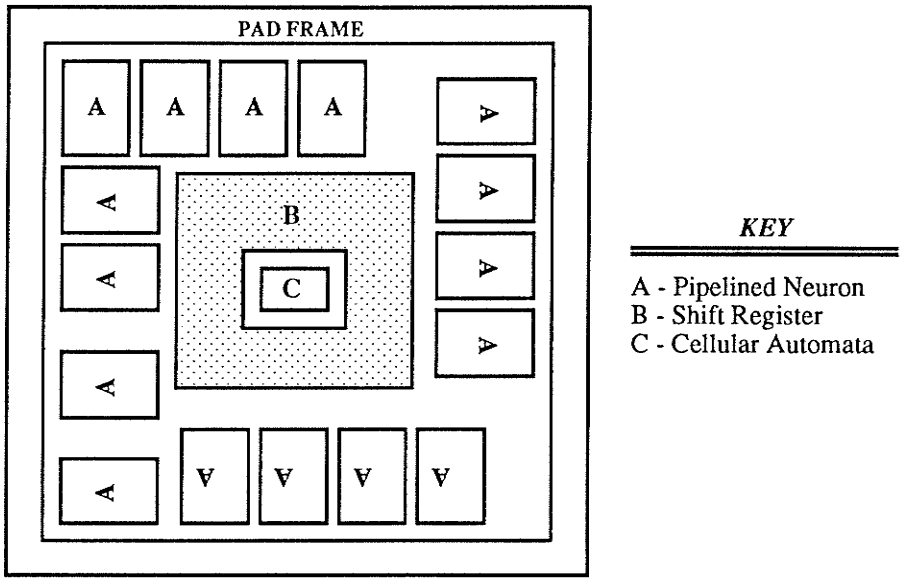

Figure 3.18: **Breakdown of the Neural Network Chip**

One disadvantage of this architecture is that the weights and threshold control lines must be available *as required* by the neuron. Since it is extremely area intensive to implement memory devices on-chip, pins must be reserved to load these values from off-chip as required. Fortunately, only one weight per neuron is required at any given time. Furthermore, the weights and threshold controls are not required at the same time.

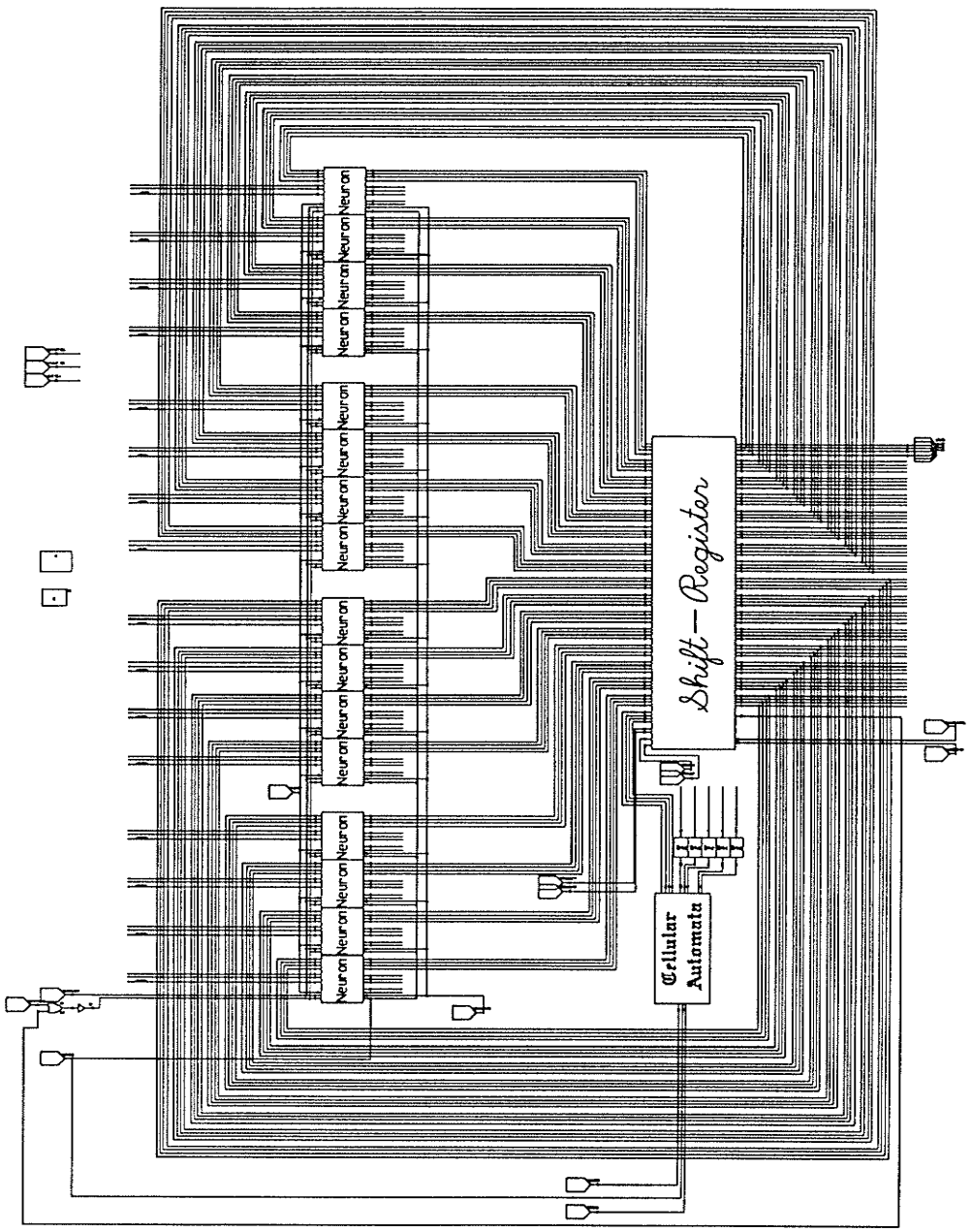

Figure 3.19: **Upper-Level Schematic for the Neural Network Chip**

Neuron Size†	1070 × 825
Shift Register Size†	2650 × 2370
CA Size†	805 × 640
Total Chip Size†	7200 × 7450
Number of Pins	70
Input Pins	4 (Serial)
Output Pins	4 (Serial)
Weight Pins	48††
Threshold Pins	32††
Devoted Test Pins	3

Table 3.7: **Neural Network Chip Specifications**

† in 3 μm CMOS
†† Multiplexed

Chapter 4

Conclusions and Recommendations

4.1 Summary and Conclusions

VLSI, neural networks and fuzzy set theory were combined to produce a system for real-time application. Fault tolerance was attempted and redundancies of the layouts were advantageously used for built-in self test. The result of this eclectic merger is a very robust chip set. A brief synopsis of the three chips designed is presented in the following table.

	Matching	Neural Network	Inverse Matching
Chip Size in 3 μm Tech.	4503×4503	7200×7446	4503×4503
Number of Pins	68	70	68
Number of Input Pins	32	4†	32
Number of Output Pins	16	4†	32
Number of Pins for Weights	N/A	48††	N/A
Number of Pins for Thresholding	N/A	32††	N/A
Number of Dedicated Test Pins	5	3	5††

† Serial

†† Multiplexed

In conclusion, the objectives outlined at the onset of this thesis have been achieved.

4.2 Recommendations

Several interesting aspects of the system were only superficially explored, yet have great potential impact on the design.

The on-chip pseudorandom number generation has some interesting possibilities. Since a shift register is used in the implementation (to load/unload the CA), this structure could be directly embedded into the neural network structure. This would lessen the 2-3% of chip area already consumed by the CA.

A common problem with the backpropagation algorithm occurs when the search point is driven far out on a plateau, making gradient decent very difficult. If the system does not converge, it would be advantageous to begin the learning procedure again with new random weights, and the CA could provide these.

Of course on-chip learning would greatly increase processing speed of the system. A coprocessing chip for learning could be created with the same architectural concept as the feed-forward chip. While the coprocessing idea would not be as fast as learning on-chip, it would be much faster than software learning, and the system architecture lends itself nicely to this kind of interfacing. Weights could be cycled in a shift register in the same way that activation levels are transported in the already-made chip.

Given the chain-like structure developed for neural computations, percolation theory has some obvious applications [34]. This is a relatively new science which involves clustering of randomly occupied sites in a lattice. The application is specifically applicable to transport phenomena. A large series of neurons could be fabricated in a

lattice configuration with an emerging technology that causes many sites to be faulty. An attempt would then be made to produce a maximal length chain of working sites within the lattice, yielding a relatively large number of neurons on the layer. While many working neurons on the chip might not be used, the increase in yield for the new technology could make the technique quite effective.

An area that has not been examined in this thesis is the system software shell, shown in Figure 4.1. To make the system more effective, a software shell could be created to aid in fuzzification (entering the membership functions), defuzzification, general system processing, graphical and numerical data examination, neural network learning and weight manipulation.

Finally, it would be advisable to implement another generation of the chips discussed here, now that unforeseen problems and potential have been documented. The new chips could then be incorporated into a circuit board and used for practical applications.

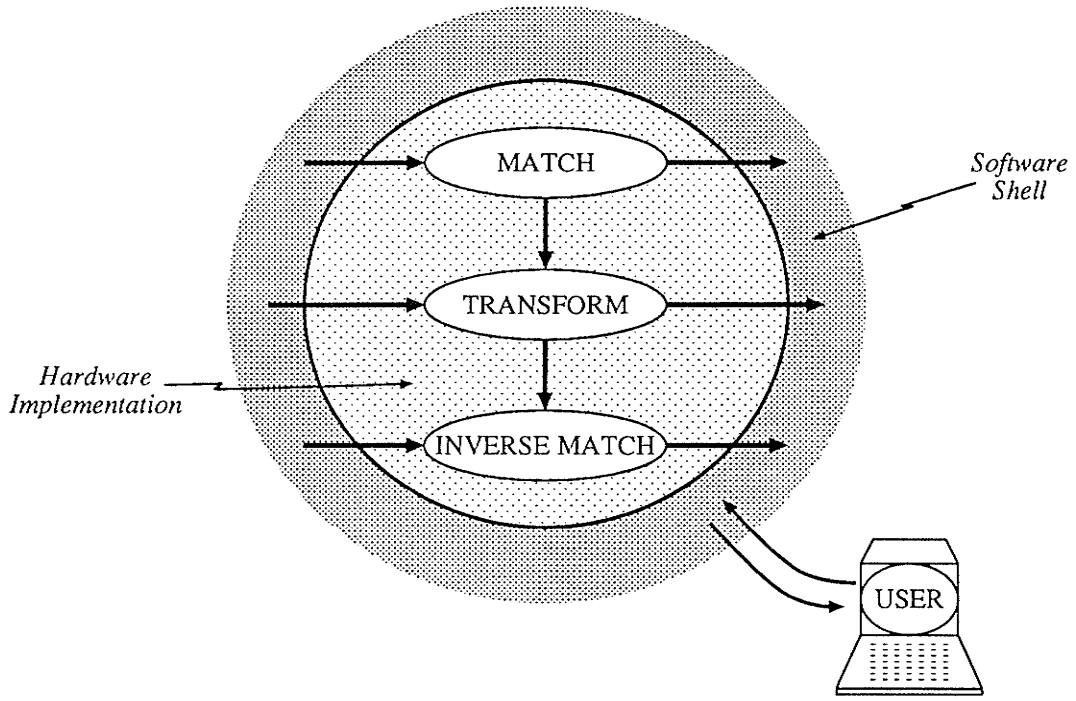

Figure 4.1: **Software Shell for the Fuzzy Cognitive System**

Appendix A

The Backpropagation Algorithm

An Example

To illustrate the mechanism of the backpropagation algorithm, an example is now presented. In the network of Figure A.1, the error derivatives of all weights will be calculated using the nonlinear transfer function:

$$y_j = \frac{1}{1 + e^{-x_j}}$$

The technique shown in this example is used in backpropagation to propagate error from the output back to the input, thereby indicating the optimal direction of movement for gradient decent.

y_a :

$$\frac{\partial E}{\partial y_{a1}} = y_{a1} - d_{a1} = 0.4924 - 0.75 = -0.2576$$

$$\frac{\partial E}{\partial y_{a2}} = 0.3560 - 1 = -0.6440$$

$$\frac{\partial E}{\partial y_{a3}} = 0.4962 - 0.5 = -0.0038$$

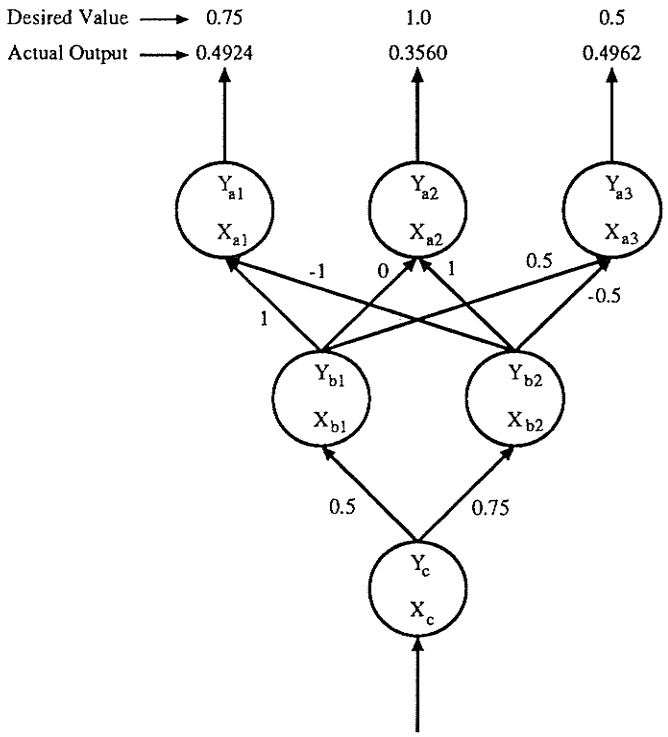

Figure A.1: Network for the Backpropagation Example

$\mathbf{x_a}$:

$$\frac{\partial E}{\partial x_{a1}} = \frac{\partial E}{\partial y_{a1}} \frac{\partial y_{a1}}{\partial x_{a1}} = \frac{\partial E}{\partial y_{a1}} [y_{a1}(1 - y_{a1})]$$

$$= -0.2576[0.4924(1 - 0.4924)] = -0.06439$$

$$\frac{\partial E}{\partial x_{a2}} = (-0.6440)[0.3560(1 - 0.3560)] = -0.1476$$

$$\frac{\partial E}{\partial x_{a3}} = (-0.0038)[0.4926(1 - 0.4926)] = -9.498 \times 10^{-4}$$

Now we must calculate y_{b1} and y_{b2}:

$$x_{a1} = -\ln(\frac{1}{y_{a1}} - 1) = -0.03040$$

$$x_{a2} = -0.5928$$

$$x_{a3} = -0.01520$$

$$x_{a1} = -0.03040 = (1)y_{b1} + (-1)y_{b2}$$

$$x_{a2} = -0.5928 = (0)y_{b1} + (-1)y_{b2}$$

$$\Rightarrow y_{b2} = 0.5928$$

$$\Rightarrow y_{b1} = -0.0304 + 0.5928 = 0.5624$$

Check:

$$x_{a3} = (0.5)y_{b1} + (-0.5)y_{b2}$$

$$-0.01520 = (0.5)(0.5624) + (-0.5)(0.5928) \quad \surd$$

$\mathbf{w_{(a)(b)}}$:

$$\frac{\partial E}{\partial w_{(a1)(b1)}} = \frac{\partial E}{\partial x_{a1}} \frac{\partial x_{a1}}{\partial w_{(a1)(b1)}} = \frac{\partial E}{\partial x_{a1}} y_{b1} = (-0.06439)(0.5624) = -0.03621$$

$$\frac{\partial E}{\partial w_{(a1)(b2)}} = (-0.06439)(0.5928) = -0.03817$$

$$\frac{\partial E}{\partial w_{(a2)(b1)}} = (-0.1476)(0.5624) = -0.08301$$

$$\frac{\partial E}{\partial w_{(a2)(b2)}} = (-0.1476)(0.5928) = -0.08750$$

$$\frac{\partial E}{\partial w_{(a3)(b1)}} = (-9.498 \times 10^{-4})(0.5624) = -5.342 \times 10^{-4}$$

$$\frac{\partial E}{\partial w_{(a3)(b2)}} = (-9.498 \times 10^{-4})(0.5928) = -5.630 \times 10^{-4}$$

Next Layer:

$\mathbf{y_b}$:

$$\frac{\partial E}{\partial y_{b1}} = \sum_a \left(\frac{\partial E}{\partial x_a} \frac{\partial x_a}{\partial y_{b1}}\right) = \sum_a \left(\frac{\partial E}{\partial x_a} w_{(a)(b1)}\right)$$

$$= (-0.06439)(1) + (-0.1476)(0) + (-9.498 \times 10^{-4})(0.5) = -0.06486$$

$$\frac{\partial E}{\partial y_{b2}} = (-0.06439)(-1) + (-0.1476)(-1) + (-9.498 \times 10^{-4})(-0.5) = 0.2125$$

$\mathbf{x_b}$:

$$\frac{\partial E}{\partial x_{b1}} = \frac{\partial E}{\partial y_{b1}} \frac{\partial y_{b1}}{\partial x_{b1}} = \frac{\partial E}{\partial y_{b1}} [y_{b1}(1 - y_{b1})]$$

$$= -0.06486[0.5624(1 - 0.5624)] = -0.01596$$

$$\frac{\partial E}{\partial x_{b2}} = (0.2125)[0.5928(1 - 0.5928)] = 0.05129$$

Now we must calculate y_c:

$$0.5y_c = x_{b1} = -\ln(\frac{1}{y_{b1}} - 1)$$

$$\Rightarrow y_c = 0.5$$

Check:

$$(0.75)(0.5) = -\ln(\frac{1}{0.5928} - 1) \quad \surd$$

$\mathbf{w_{(b)(c)}}$:

$$\frac{\partial E}{\partial w_{(b1)(c)}} = \frac{\partial E}{\partial x_{b1}} \frac{\partial x_{b1}}{\partial w_{(b1)(c)}} = \frac{\partial E}{\partial x_{b1}} y_c = (-0.01596)(0.5) = -0.00798$$

$$\frac{\partial E}{\partial w_{(b2)(c)}} = (0.05129)(0.5) = 0.02565$$

Obviously the error derivatives on the weights attached to neuron a3 are the smallest. Considering the actual output error on this neuron is the smallest this is sensible. By the same logic, the error derivative of the weights on neuron a2 are largest.

The weight derivatives from the input layer to the hidden layer are much more difficult to interpret. At this stage the direction the the weights should take is much more distributed, and therefore approximately the same magnitude. There is no preference given to *sides* as there is in the hidden-to-output weight derivatives.

Improving Back-Propagation

Back-propagation uses a gradient descent method (a generalization of the least mean error method) for learning. Although learning the weights is NP complete in the

worst case with the learning time growing as $o(N)^3$ with the number of inputs, we may use a number of techniques for reducing learning time. Several other techniques may be used for increasing the conceptual power of the networks.

Perhaps the most obvious way to improve performance is to hardwire preprocessing and postprocessing units to the network. This in-effect reduces the amount of work required by the network to obtain valid results, thus speeding up the learning. The method of 'bins' may be used to reduce a continuous problem to discrete. This may greatly reduce the complexity of some problems at the expense of precision.

Preprocessing may also be used to reduce the correlation of input data. This technique allows the gradient to point directly at the minimum, allowing a linear system to calculate the solution.

By examining the change of weights, other simplifications can be made. While the gradient may change only slightly for a small change in some weights, the gradient may change greatly for small movements in other weights. This implies that if we can detect the changes in weights, we can affect a localized adaptive learning rate, thus speeding movement toward the minimum.

Alternately, we may calculate the error gradient for the case when a given weight is equal to zero. If the gradient changes only slightly, we may choose to sever the connection completely to speed up the remainder of the network (fewer weights to learn).

We may speed up learning, particularly in an asynchronous hardware scheme, by performing a constant weight reduction, or forgetting procedure. This technique ensures that the weights are kept to a reasonable limit, thus yielding less complex

multiplications.

A 'momentum' method may also be used to speed up learning in directions with small but constant gradients. In this method, momentum builds up *along* ravines, but cancels out *across* the ravines (when they are traversed).

A radial basis function may be used to draw-in solutions with landmarks. This technique causes a best-fit approximation to a high dimensional space by allowing radial basis function centers (in effect, the weights from the input-to-hidden layer) to capture the data points (training set). With this complete, the remaining weights are determined with a linear least-squares optimization. This procedure is quite fast and efficient.

A number of other methods may be used to modify the network itself. Since minimizing the squared error is equivalent to to an estimation of maximum likelihood, some problems may benefit from a reinterpretation of the problem.

By utilizing symmetric weights and mapping the output back onto the inputs, a self-supervised network may be used as a feature detector. The outcome of the hidden layer is unknown, but neurons may be reduced until the input may be adequately reconstructed at the output. The features classified in the hidden layer may then be considered the minimum necessary for recognition.

Each of the techniques presented here has it's own advantages and disadvantages, and adaptation to a given problem is required. There is, however, no shortage of ways to improve back-propagation.

Appendix B

Software Simulations

In this Appendix, a presentation will be made of some software created to assist the understanding of various aspects of the fuzzy cognitive system. All software is written in 'C' language and although it was written on a SUN workstation, should be quite portable.

The Matching Operator

This program produces all possible combinations for a four bit resolution [0,15] matching operator.

```
/*****************************************************************/
/* This program generates all matching combinations, 4-bit res. */
/*****************************************************************/


main()
{
int x,y,z;
x = 0;
y = 0;
z = 0;

do{
    do{
        z = match(x,y);
        printf("\n%d %d %d",y,x,z);
        ++x;
    } while (x<=15);

    ++y;
    x = 0;

} while (y<=15);
}

match(object,proto)
int object, proto;
{
    int max, min, answer;
    if (object<proto){
        min = object;
        max = proto;
    }
    else{
        min = proto;
        max = object;
    }

    if (max==min){
        answer = 15;
    }else{
        answer = (min + 15 - max)/2;
/**** Eliminate the '/2' to make the data compressed matching operator ****
    }
    return answer;
}
```

The Inverse Matching Operator

This program produces all possible combinations for a four bit resolution [0,15] inverse matching operator.

```
/*************************************************************************/
/* This program generates all inverse matching combinations, 4-bit res. */
/*************************************************************************/


main()
{
int match, proto, answer1, answer2, mb, zb;

for(match = 0; match <16; ++match) {

          for (proto = 0; proto < 16; ++proto) {

                    if (match > 7) {

                              answer1 = proto;
                              answer2 = proto;
                              mb = 1;
                              zb = 0;

                    } else {

                              answer1 = proto + (2 * match) - 15;
                              answer2 = proto - (2 * match) + 15;
/**** Eliminate the '2 *' to make the data compressed inverse matching operator ****
                              zb = 0;
                              mb = 0;

                              if (answer1 < 0)
                                        answer1 = 0;

                              if (answer2 > 15)
                                        answer2 = 15;

                              if (answer1 == 0) {
                                        if(answer2 == 15)
                                                  zb = 1;
                              }

                    }
                    printf("match =
                    printf("\n");
          } printf("\n");
}}
```

System Simulation and Example

The following code is used to simulate the complete system. In the example used, the data is fabricated but based on an imaginary example discussed following the presentation of the code.

```
/*****************************************************************/
/* This program simulates the 4-bit fuzzy cognitive system.      */
/*****************************************************************/


int object[4][16] ={{5,5,5,5,7,9,11,12,11,9,6,6,5,5,5,5},
                    {8,8,8,8,8,8,8,8,9,9,10,9,7,8,8,8},
                    {0,0,1,3,6,9,13,12,10,9,9,11,12,14,14,13},
                    {11,5,8,9,10,10,11,13,13,12,10,11,11,10,10,10}};

int proto[4][16] = {{6,6,6,6,6,6,15,15,15,2,6,6,6,6,6,6},
                    {11,11,11,11,11,11,11,11,9,9,11,11,7,11,11,11},
                    {3,4,5,7,8,10,12,11,10,10,10,11,12,13,13,12},
                    {3,4,5,7,9,11,13,15,15,14,13,10,11,10,10,9}};

int invproto[16] = {8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8};

int w[4][16] =     {{15,14,13,14,15,14,15,15,13,12,11,0,5,6,9,10},
                    {0,0,0,1,2,3,5,9,11,14,10,8,6,8,10,11},
                    {12,12,12,12,12,12,12,12,12,12,12,12,12,12,12,12},
                    {13,12,11,10,9,10,11,12,13,14,15,15,14,10,13,12}};

int d[16] =        {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

int s[4][16] =     {{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
                    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
                    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
                    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}};
```

```
/*****************************************************************/
/* This is the main simulation for the fuzzy cognitive system.  */
/*****************************************************************/


main()
{
int n, m;
int zen, sum, time, neurin[4], big, small;
int low, high, bits;
n = 0;
time = 0;
sum = 0;

do{

   printf("\nn=%d",n+1);
   printf(" INPUTS\n");
   printf("=== ========================\n");

   do{
      zen = match(object[n][time],proto[n][time]);
      sum = zen + sum;
      printf("time =%d, proto=%d, object=%d, match=%d, sum=%d\n",
             time,proto[n][time],object[n][time],zen,sum);
      ++time;
} while (time<=15);

   neurin[n] = sum / 16;
   time=0;
   printf("Neural Net Input for objective [%d] = %d\n",n+1,neurin[n]);
   sum = 0;
   ++n;

} while (n<=3);
n=0;
printf("\nneural net inputs are :  ");
do{
   printf("%d ",neurin[n]);
   ++n;
} while (n!=4);

printf("\n\n");
```

```
/*******************************/
/* Neural Network Simulation.  */
/*******************************/


n=0;
m=0;

do{

   do{
      s[m][n] = w[m][n] * neurin[m] / 16;
      if ((w[m][n] ==15) && (neurin[m] == 15))
                     s[m][n] = 15;
      ++n;           } while (n!=16);
   n = 0;
   ++m;
} while (m!=4);

n=0;
m=0;

do{
   do{
      d[m] = d[m] + s[n][m];
      ++n;
   }while (n!=4);

   d[m] = d[m] / 4;
   n = 0;
   ++m;
} while (m!=16);

printf("\n");

n=0;
m=0;

printf("Neural Net Weights are:\n");

do{
   do{
      if (w[m][n] > 9)
                     printf("%d ",w[m][n]);
      else
                     printf(" %d ",w[m][n]);
      ++n;
```

```
   } while (n!=16);

   ++m;
   n=0;
   printf("\n");
} while (m!=4);

n=0;
m=0;

printf("\n\nNeural Net Matrix is:\n");

do{
   do{
      if (s[m][n] > 9)
                     printf("%d ",s[m][n]);
      else
                     printf(" %d ",s[m][n]);

      ++n;
   } while (n!=16);

   ++m;
   n=0;
   printf("\n");
} while (m!=4);

printf("\n\nOutput of Neural Net is:\");

m = 0;

do{
   if (d[m] > 9)
      printf("%d ",d[m]);
   else
      printf(" %d ",d[m]);
   ++m;
} while (m!=16);
printf("\n\n\n\n\n");
```

```
/*******************************/
/* Inverse Matching Stage.     */
/*******************************/



time = 0;

printf(" Input Output\n");
printf(" ------------ ========================================\n");

do{
   low = answer1(d[time],invproto[time]);
   high = answer2(d[time],invproto[time]);
   bits = status(low,high);
   printf("d[%d] = %d invproto = %d low = %d high = %d Status = %d\n", time, d[time],invproto[
   ++time;
} while (time<=15);

printf("\n\nStatus Key:\n 0 ===> some interval of length > 1 but <15\n");
printf(" 1 ===> interval of maximal length\n 2 ===> interval of minimal length\n");

}



/*****************************************/
/* This subroutine performs matching.    */
/*****************************************/


match(object,proto)
int object, proto;
{
                    int max, min, zenmatch;
                    if (object<proto){
                              min = object;
                              max = proto;
                    }
                    else{
                              min = proto;
                              max = object;
                    }

                    zenmatch = min + 15 - max;
```

```
                return zenmatch;
}



/*************************************************************************/
/* This subroutine calculates the low boundary of inverse matching.     */
/*************************************************************************/


answer1(nnmatch,invproto)
int nnmatch, invproto;

{

int lowanswer;

            if (nnmatch >7) {
                        lowanswer = invproto;
            } else {
                        lowanswer = invproto + (2 * nnmatch) - 15;
                        if (lowanswer <0)
                                    lowanswer = 0;
            }
return lowanswer;

}



/*************************************************************************/
/* This subroutine calculates the high boundary of inverse matching.     */
/*************************************************************************/


answer2(nnmatch,invproto)
int nnmatch, invproto;

{

int highanswer;

             if (nnmatch >7) {
                         highanswer = invproto;
             } else {
```

```
                        highanswer = invproto - (2 * nnmatch) + 15;
                        if (highanswer >15)
                                highanswer = 15;
            }
return highanswer;

}



/*************************************************************************/
/* This subroutine calculates the status bits after inverse matching.    */
/*************************************************************************/


status(low,high)
int low, high;

{

int mb, zb, stats ;

                if (low ==high) {
                        mb = 1;
                        zb = 0;
                } else {
                        mb = 0 ;
                        zb = 0 ;
                        if (low ==0) {
                                if (high ==15)
                                        zb = 1;
                        }
                }
stats = mb * 2 + zb * 1;
return stats;
}
```

The output of this simulation is the following:

```
n=1                    INPUTS
===              ==========================
time =0,         proto=6,         object=5,       match=14,       sum=14
time =1,         proto=6,         object=5,       match=14,       sum=28
time =2,         proto=6,         object=5,       match=14,       sum=42
time =3,         proto=6,         object=5,       match=14,       sum=56
time =4,         proto=6,         object=7,       match=14,       sum=70
time =5,         proto=6,         object=9,       match=12,       sum=82
time =6,         proto=15,        object=11,      match=11,       sum=93
time =7,         proto=15,        object=12,      match=12,       sum=105
time =8,         proto=15,        object=11,      match=11,       sum=116
time =9,         proto=2,         object=9,       match=8,        sum=124
time =10,        proto=6,         object=6,       match=15,       sum=139
time =11,        proto=6,         object=6,       match=15,       sum=154
time =12,        proto=6,         object=5,       match=14,       sum=168
time =13,        proto=6,         object=5,       match=14,       sum=182
time =14,        proto=6,         object=5,       match=14,       sum=196
time =15,        proto=6,         object=5,       match=14,       sum=210
Neural Net Input for objective [1] = 13

n=2                    INPUTS
===              ==========================
time =0,         proto=11,        object=8,       match=12,       sum=12
time =1,         proto=11,        object=8,       match=12,       sum=24
time =2,         proto=11,        object=8,       match=12,       sum=36
time =3,         proto=11,        object=8,       match=12,       sum=48
time =4,         proto=11,        object=8,       match=12,       sum=60
time =5,         proto=11,        object=8,       match=12,       sum=72
time =6,         proto=11,        object=8,       match=12,       sum=84
time =7,         proto=11,        object=8,       match=12,       sum=96
time =8,         proto=9,         object=9,       match=15,       sum=111
time =9,         proto=9,         object=9,       match=15,       sum=126
time =10,        proto=11,        object=10,      match=14,       sum=140
time =11,        proto=11,        object=9,       match=13,       sum=153
time =12,        proto=7,         object=7,       match=15,       sum=168
time =13,        proto=11,        object=8,       match=12,       sum=180
time =14,        proto=11,        object=8,       match=12,       sum=192
time =15,        proto=11,        object=8,       match=12,       sum=204
Neural Net Input for objective [2] = 12
```

```
n=3                    INPUTS
===              ==========================
time =0,         proto=3,         object=0,       match=12,       sum=12
time =1,         proto=4,         object=0,       match=11,       sum=23
time =2,         proto=5,         object=1,       match=11,       sum=34
time =3,         proto=7,         object=3,       match=11,       sum=45
time =4,         proto=8,         object=6,       match=13,       sum=58
time =5,         proto=10,        object=9,       match=14,       sum=72
time =6,         proto=12,        object=13,      match=14,       sum=86
time =7,         proto=11,        object=12,      match=14,       sum=100
time =8,         proto=10,        object=10,      match=15,       sum=115
time =9,         proto=10,        object=9,       match=14,       sum=129
time =10,        proto=10,        object=9,       match=14,       sum=143
time =11,        proto=11,        object=11,      match=15,       sum=158
time =12,        proto=12,        object=12,      match=15,       sum=173
time =13,        proto=13,        object=14,      match=14,       sum=187
time =14,        proto=13,        object=14,      match=14,       sum=201
time =15,        proto=12,        object=13,      match=14,       sum=215
Neural Net Input for objective [3] = 13

n=4                    INPUTS
===              ==========================
time =0,         proto=3,         object=11,      match=7,        sum=7
time =1,         proto=4,         object=5,       match=14,       sum=21
time =2,         proto=5,         object=8,       match=12,       sum=33
time =3,         proto=7,         object=9,       match=13,       sum=46
time =4,         proto=9,         object=10,      match=14,       sum=60
time =5,         proto=11,        object=10,      match=14,       sum=74
time =6,         proto=13,        object=11,      match=13,       sum=87
time =7,         proto=15,        object=13,      match=13,       sum=100
time =8,         proto=15,        object=13,      match=13,       sum=113
time =9,         proto=14,        object=12,      match=13,       sum=126
time =10,        proto=13,        object=10,      match=12,       sum=138
time =11,        proto=10,        object=11,      match=14,       sum=152
time =12,        proto=11,        object=11,      match=15,       sum=167
time =13,        proto=10,        object=10,      match=15,       sum=182
time =14,        proto=10,        object=10,      match=15,       sum=197
time =15,        proto=9,         object=10,      match=14,       sum=211
Neural Net Input for objective [4] = 13

neural net inputs are : 13 12 13 13
```

```
Neural Net Weights are:
15 14 13 14 15 14 15 15 13 12 11  0  5  6  9 10
 0  0  0  1  2  3  5  9 11 14 10  8  6  8 10 11
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
13 12 11 10  9 10 11 12 13 14 15 15 14 10 13 12


Neural Net Matrix is:
12 11 10 11 12 11 12 12 10  9  8  0  4  4  7  8
 0  0  0  0  1  2  3  6  8 10  7  6  4  6  7  8
 9  9  9  9  9  9  9  9  9  9  9  9  9  9  9  9
10  9  8  8  7  8  8  9 10 11 12 12 11  8 10  9


Output of Neural Net is:
 7  7  6  7  7  7  8  9  9  9  9  6  7  6  8  8
```

	Input	Output		
	invproto = 8	low = 7	high = 9	Status = 0
d[0] = 7				
d[1] = 7	invproto = 8	low = 7	high = 9	Status = 0
d[2] = 6	invproto = 8	low = 5	high = 11	Status = 0
d[3] = 7	invproto = 8	low = 7	high = 9	Status = 0
d[4] = 7	invproto = 8	low = 7	high = 9	Status = 0
d[5] = 7	invproto = 8	low = 7	high = 9	Status = 0
d[6] = 8	invproto = 8	low = 8	high = 8	Status = 2
d[7] = 9	invproto = 8	low = 8	high = 8	Status = 2
d[8] = 9	invproto = 8	low = 8	high = 8	Status = 2
d[9] = 9	invproto = 8	low = 8	high = 8	Status = 2
d[10] = 9	invproto = 8	low = 8	high = 8	Status = 2
d[11] = 6	invproto = 8	low = 5	high = 11	Status = 0
d[12] = 7	invproto = 8	low = 7	high = 9	Status = 0
d[13] = 6	invproto = 8	low = 5	high = 11	Status = 0
d[14] = 8	invproto = 8	low = 8	high = 8	Status = 2
d[15] = 8	invproto = 8	low = 8	high = 8	Status = 2

```
Status Key:
                0 ===> some interval of length > 1 but <15
                1 ===> interval of maximal length
                2 ===> interval of minimal length
```

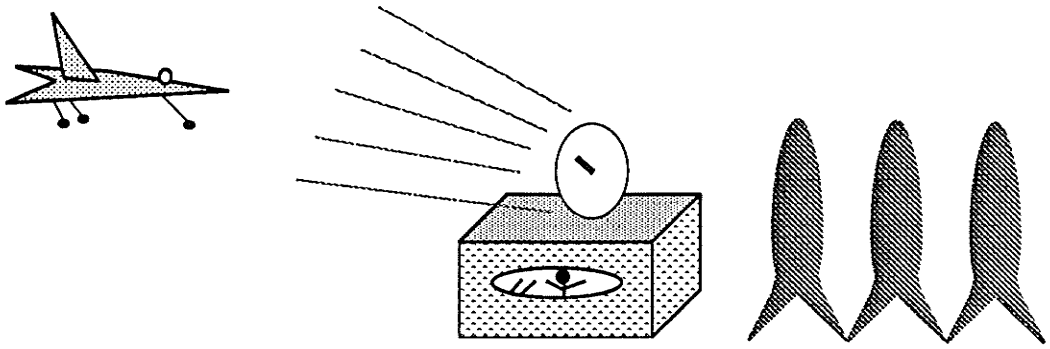

Figure B.1: **Prototype Jet for the Simulation**

Consider the following example. Imagine that a fighter jet is approaching the protected missile site of Figure B.1. This jet is friendly and therefore would be used to train the neural network (it is used as the *prototype*). The plane is flown towards the missile site under several conditions and angles. Four methods are used to sense incoming craft:

- Visual Inspection
- Sonic Characteristics
- Radar Profile
- Communications/Beacons

Now the system is taken out of its learning mode, and another craft is sensed approaching, as shown in Figure B.2. While this is also a jet and very similar in most ways, it is not exactly the same. This new subject is termed the *objective.*

The data of the previous simulation may be interpreted as the comparison of these two aircraft. The input data is presented graphically as Figures B.3-B.6. The output of the simulation is presented as Figure B.7.

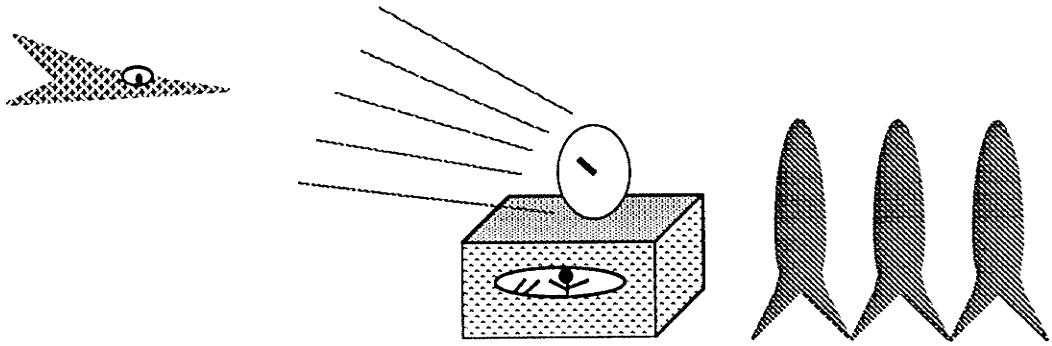

Figure B.2: Objective Jet for the Simulation

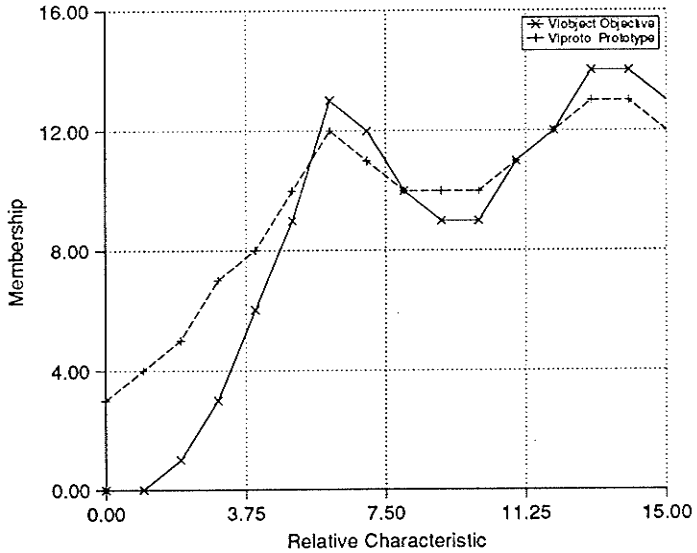

Figure B.3: Visual Inspection for Simulation 1

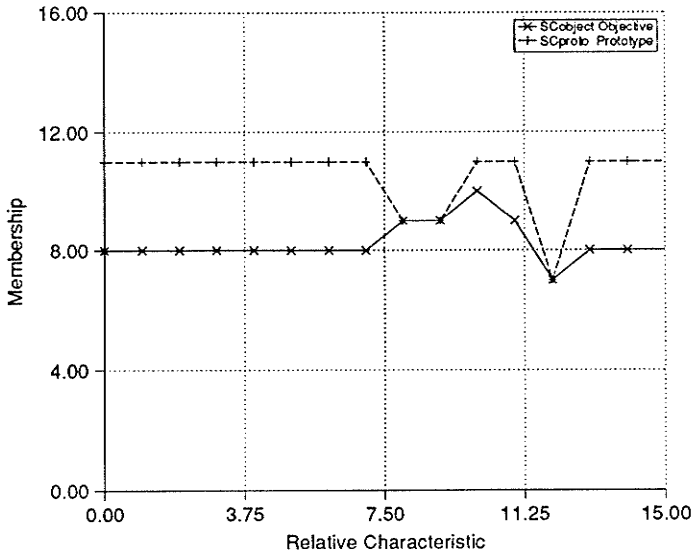

Figure B.4: **Sonic Characteristics for Simulation 1**

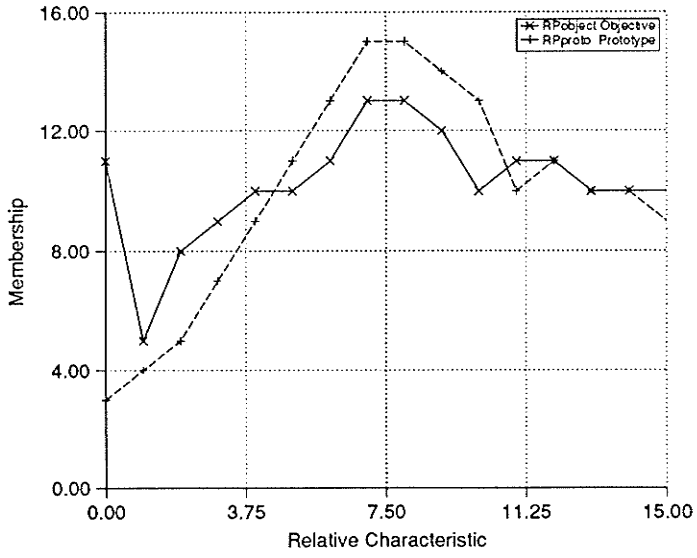

Figure B.5: **Radar Profile for Simulation 1**

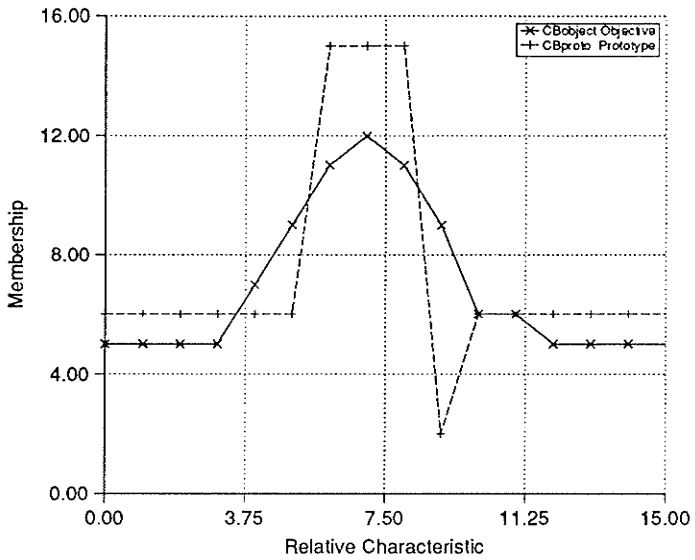

Figure B.6: Communications/Beacons for Simulation 1

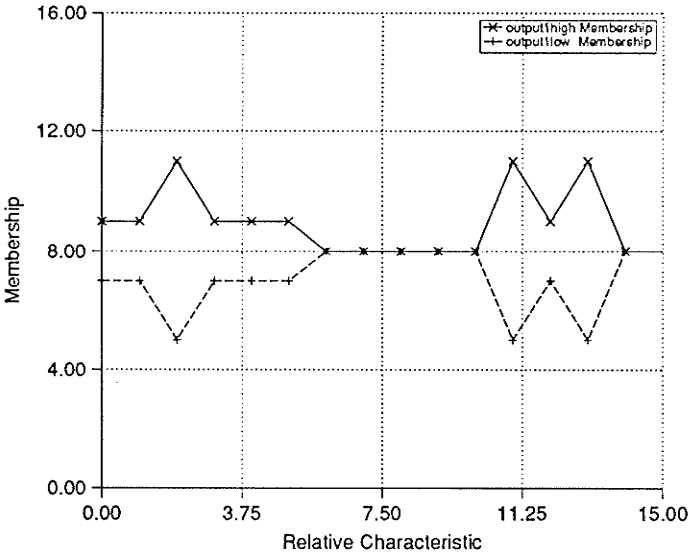

Figure B.7: Output of Simulation 1

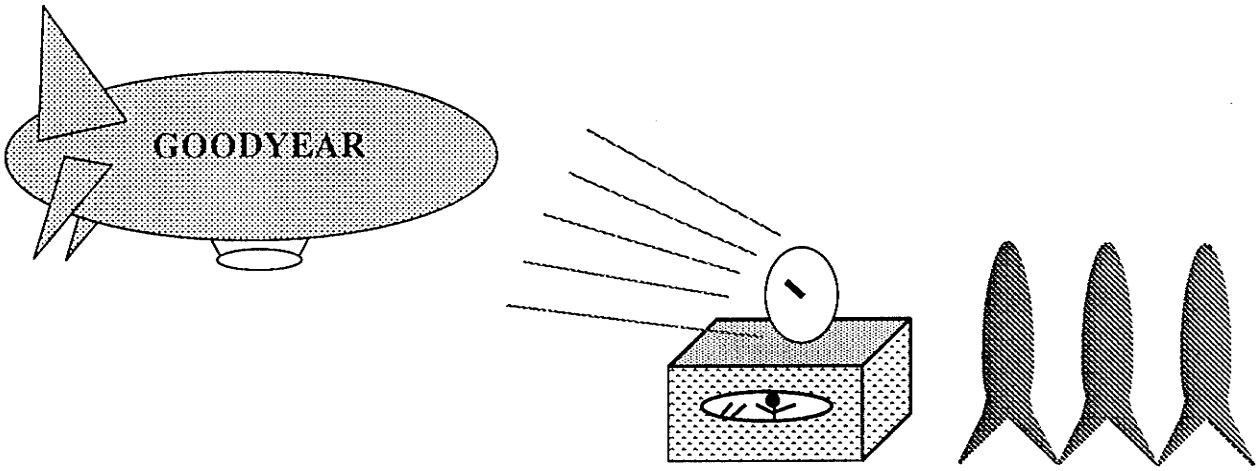

Figure B.8: **New Objective Jet for the Simulation**

The narrow confidence interval of the output indicates that the object under scrutiny is similar to the training data but not exact. A human or expert system could then be used to take appropriate action.

Now imagine another incoming craft, shown in Figure B.8. The data for this new encounter is presented in the following simulation.

```
n=1                    INPUTS
===              ==========================
time =0,         proto=6,         object=15,      match=6,         sum=6
time =1,         proto=6,         object=15,      match=6,         sum=12
time =2,         proto=6,         object=15,      match=6,         sum=18
time =3,         proto=6,         object=15,      match=6,         sum=24
time =4,         proto=6,         object=15,      match=6,         sum=30
time =5,         proto=6,         object=14,      match=7,         sum=37
time =6,         proto=15,        object=14,      match=14,        sum=51
time =7,         proto=15,        object=13,      match=13,        sum=64
time =8,         proto=15,        object=12,      match=12,        sum=76
time =9,         proto=2,         object=11,      match=6,         sum=82
time =10,        proto=6,         object=10,      match=11,        sum=93
time =11,        proto=6,         object=8,       match=13,        sum=106
time =12,        proto=6,         object=6,       match=15,        sum=121
time =13,        proto=6,         object=4,       match=13,        sum=134
time =14,        proto=6,         object=2,       match=11,        sum=145
time =15,        proto=6,         object=0,       match=9,         sum=154
Neural Net Input for objective [1] = 9

n=2                    INPUTS
===              ==========================
time =0,         proto=11,        object=0,       match=4,         sum=4
time =1,         proto=11,        object=1,       match=5,         sum=9
time =2,         proto=11,        object=2,       match=6,         sum=15
time =3,         proto=11,        object=4,       match=8,         sum=23
time =4,         proto=11,        object=5,       match=9,         sum=32
time =5,         proto=11,        object=7,       match=11,        sum=43
time =6,         proto=11,        object=13,      match=13,        sum=56
time =7,         proto=11,        object=12,      match=14,        sum=70
time =8,         proto=9,         object=11,      match=13,        sum=83
time =9,         proto=9,         object=9,       match=15,        sum=98
time =10,        proto=11,        object=13,      match=13,        sum=111
time =11,        proto=11,        object=12,      match=14,        sum=125
time =12,        proto=7,         object=8,       match=14,        sum=139
time =13,        proto=11,        object=6,       match=10,        sum=149
time =14,        proto=11,        object=3,       match=7,         sum=156
time =15,        proto=11,        object=2,       match=6,         sum=162
Neural Net Input for objective [2] = 10
```

```
n=3                    INPUTS
===                ==========================
time =0,           proto=3,          object=11,      match=7,          sum=7
time =1,           proto=4,          object=11,      match=8,          sum=15
time =2,           proto=5,          object=11,      match=9,          sum=24
time =3,           proto=7,          object=9,       match=13,         sum=37
time =4,           proto=8,          object=5,       match=12,         sum=49
time =5,           proto=10,         object=5,       match=10,         sum=59
time =6,           proto=12,         object=5,       match=8,          sum=67
time =7,           proto=11,         object=5,       match=9,          sum=76
time =8,           proto=10,         object=5,       match=10,         sum=86
time =9,           proto=10,         object=5,       match=10,         sum=96
time =10,          proto=10,         object=5,       match=10,         sum=106
time =11,          proto=11,         object=5,       match=9,          sum=115
time =12,          proto=12,         object=5,       match=8,          sum=123
time =13,          proto=13,         object=5,       match=7,          sum=130
time =14,          proto=13,         object=5,       match=7,          sum=137
time =15,          proto=12,         object=0,       match=3,          sum=140
Neural Net Input for objective [3] = 8

n=4                    INPUTS
===                ==========================
time =0,           proto=3,          object=5,       match=13,         sum=13
time =1,           proto=4,          object=7,       match=12,         sum=25
time =2,           proto=5,          object=9,       match=11,         sum=36
time =3,           proto=7,          object=13,      match=9,          sum=45
time =4,           proto=9,          object=14,      match=10,         sum=55
time =5,           proto=11,         object=11,      match=15,         sum=70
time =6,           proto=13,         object=9,       match=11,         sum=81
time =7,           proto=15,         object=8,       match=8,          sum=89
time =8,           proto=15,         object=7,       match=7,          sum=96
time =9,           proto=14,         object=6,       match=7,          sum=103
time =10,          proto=13,         object=5,       match=7,          sum=110
time =11,          proto=10,         object=4,       match=9,          sum=119
time =12,          proto=11,         object=5,       match=9,          sum=128
time =13,          proto=10,         object=6,       match=11,         sum=139
time =14,          proto=10,         object=9,       match=14,         sum=153
time =15,          proto=9,          object=9,       match=15,         sum=168
Neural Net Input for objective [4] = 10

neural net inputs are : 9 10 8 10
```

```
Neural Net Weights are:
15 14 13 14 15 14 15 15 13 12 11  0  5  6  9 10
 0  0  0  1  2  3  5  9 11 14 10  8  6  8 10 11
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
13 12 11 10  9 10 11 12 13 14 15 15 14 10 13 12


Neural Net Matrix is:
 8  7  7  7  8  7  8  8  7  6  6  0  2  3  5  5
 0  0  0  0  1  1  3  5  6  8  6  5  3  5  6  6
 6  6  6  6  6  6  6  6  6  6  6  6  6  6  6  6
 8  7  6  6  5  6  6  7  8  8  9  9  8  6  8  7


Output of Neural Net is:
 5  5  4  4  5  5  5  6  6  7  6  5  4  5  6  6
```

	Input	Output		
	----------	==========	==========	==========
d[0] = 5	invproto = 8	low = 3	high = 13	Status = 0
d[1] = 5	invproto = 8	low = 3	high = 13	Status = 0
d[2] = 4	invproto = 8	low = 1	high = 15	Status = 0
d[3] = 4	invproto = 8	low = 1	high = 15	Status = 0
d[4] = 5	invproto = 8	low = 3	high = 13	Status = 0
d[5] = 5	invproto = 8	low = 3	high = 13	Status = 0
d[6] = 5	invproto = 8	low = 3	high = 13	Status = 0
d[7] = 6	invproto = 8	low = 5	high = 11	Status = 0
d[8] = 6	invproto = 8	low = 5	high = 11	Status = 0
d[9] = 7	invproto = 8	low = 7	high = 9	Status = 0
d[10] = 6	invproto = 8	low = 5	high = 11	Status = 0
d[11] = 5	invproto = 8	low = 3	high = 13	Status = 0
d[12] = 4	invproto = 8	low = 1	high = 15	Status = 0
d[13] = 5	invproto = 8	low = 3	high = 13	Status = 0
d[14] = 6	invproto = 8	low = 5	high = 11	Status = 0
d[15] = 6	invproto = 8	low = 5	high = 11	Status = 0

```
Status Key:
                0 ===> some interval of length > 1 but <15
                1 ===> interval of maximal length
                2 ===> interval of minimal length
```

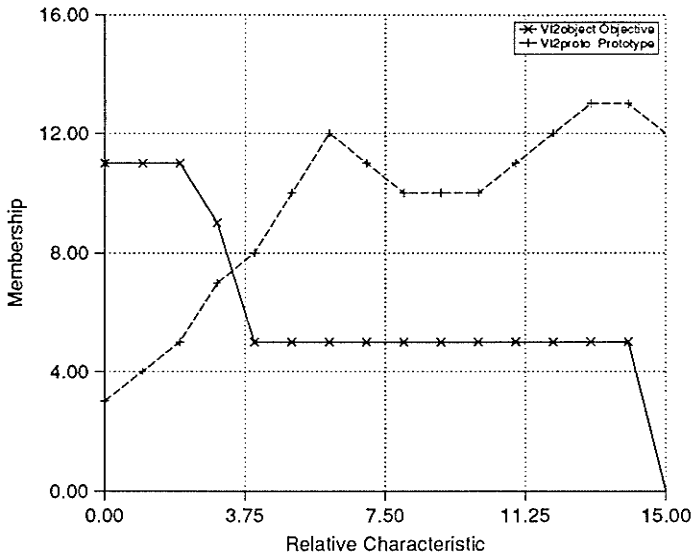

Figure B.9: **Visual Inspection for Simulation 2**

This data is represented graphically in Figures B.9-B.12. The output of the second simulation is presented as Figure B.13.

Obviously this new craft is very unlike the training data of Figure B.1. The sensory data reflects this discrepancy. The large confidence interval indicates that there is a great deal of uncertainty as to the nature of the new objective.

Again, an expert system or human operator could make a final judgment as to the action to be taken. In this case, that could be to launch a more detailed investigation or simply repeat the process later.

This example illustrates some important characteristics of the fuzzy cognitive system. First, we are dealing with real-time computations and so speed is vital. Second, the system produces quantitative *and* qualitative information. This is not simply pattern recognition since an expression of uncertainty is produced.

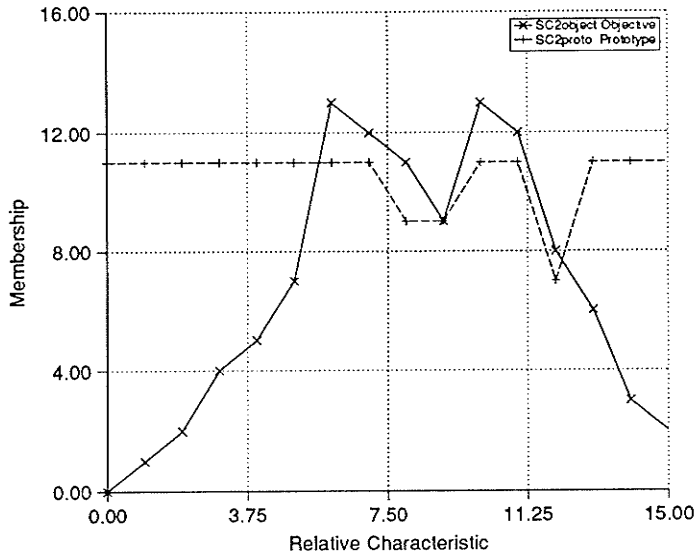

Figure B.10: **Sonic Characteristics for Simulation 2**

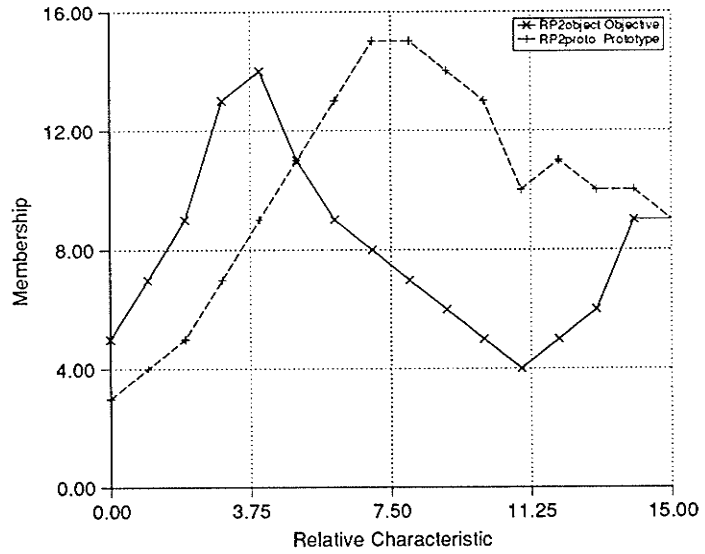

Figure B.11: **Radar Profile for Simulation 2**

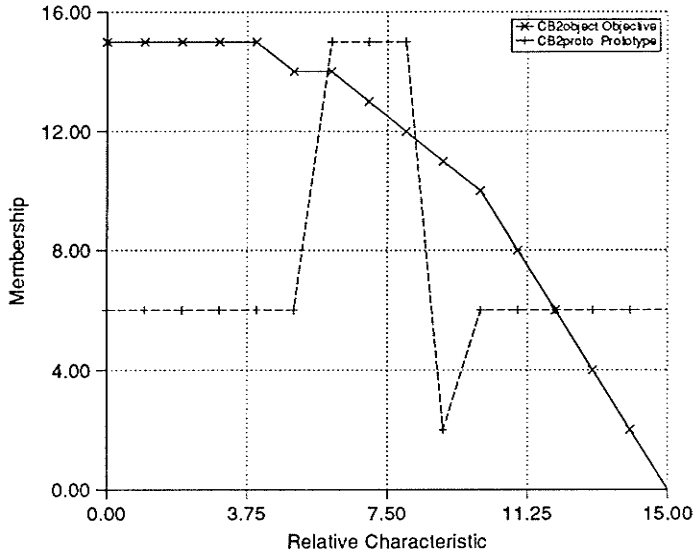

Figure B.12: **Communications/Beacons for Simulation 2**

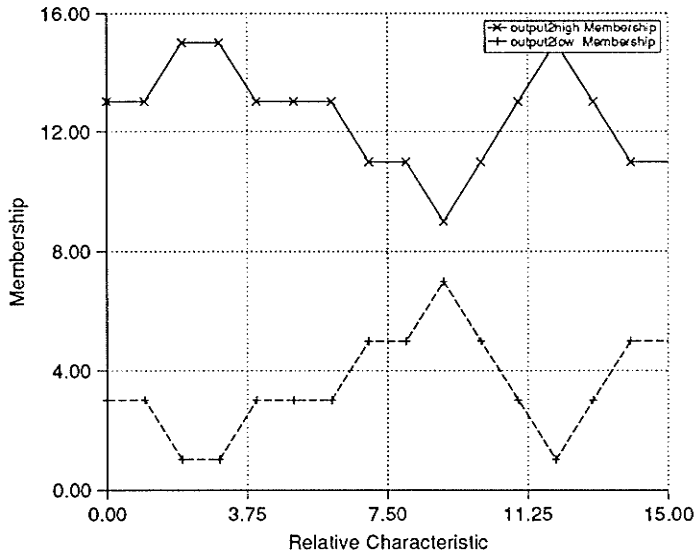

Figure B.13: **Output of Simulation 2**

Bibliography

[1] Armstrong J. A. *Chip-Level Modeling with VHDL*, Prentice Hall, 1989.

[2] Black, M. *Vagueness - An excercise in Logical Analysis*, Philos.Sci., Vol.4, 1937, pp.427-455.

[3] Blight, D.C. *Concurrent Error Detection Techniques*, internal document, University of Manitoba, 1988.

[4] Bortolan, G. and R. Degani, K. Hirota, W. Pedrycz. *Classification of Electrocardiographic Signals with the aid of Fuzzy Pattern Matching*, Proc. Symp. Applications of Fuzzy Sets, Iizuka88, Japan.

[5] Canadian Microelectronics Corporation. *CMOS3 DLM Cell Library*, Kingston, Ontario, Canada, 1989.

[6] Canadian Microelectronics Corporation. *Guide to the Integrated Circuit Implementaiton Services of the Canadian Microelectronics Corporation*, Version 4.0, Queen's University, Canada, March 1989.

[7] Chu, L.C. and B.W. Wah. *Fault Tolerant Neural Networks with Hybrid Redundancy*, International Joint Conference on Neural Networks, San Diego, Califonia, USA, 1990, pp.II639-II649.

[8] Diamond, J. and R. McLeod, W. Pedrycz. *A Fuzzy Cognitive System: Foundations and VLSI Implementation*, 3rd IFSA Congress, Seattle, Washington, 1989, pp.396-399.

[9] Grossberg, S. *Neural Networks and Natural Intelligence*, MIT Press, Cambridge, Mass. 1988.

[10] Hortensius, P.D. and R.D. McLeod, W. Pries, D.M. Miller, H.C.Card. *Cellular Automata-Based Pseudorandom Number Generators for Built-In Self Test*, Computer Aided Design of ICs and Systems, Vol.8, No. 8, Aug. 1989, pp.842-859.

[11] IEEE Standard VHDL. *Language Reference Manual*, IEEE Std 1076-1987.

[12] Johnson, B.W. Design and Analysis of Fault-Tolerant Digital Systems, Addison Wesley, USA, 1989.

[13] Johnson, C. *NASA Unites Neural Nets, Fuzzy Logic*, Electronic Engineering Times, May 23 1988, pp.40-41.

[14] Kaufmann, A. and M.M. Gupta. *Introduction to Fuzzy Arithmetic : Theory and Applications*, Van Nostrand Reinhold Co., New York, N.Y., 1985.

[15] Kostiuk, A. R. *QUISC: An Interactive Silicon Compiler*, M.Sc. Thesis, Department of Electrical Engineering, Queen's University, Kingston, Ontario, June 1987.

[16] Lim, M. and Y. Takefuji. *Implementing Fuzzy Rule-Based Systems on Silicon Chips*, IEEE Expert. Vol. 5, No. 1 February 1990, pp.31-45.

[17] Mano, M. M. *Digital Logic and Computer Design*, Prentice Hall, New Jersey, 1979.

[18] McClelland, J.L. and D.E. Rumelhart. *Explorations in Parallel Distributed Processing: A handbook of Models, Programs and Excercises*, Cambridge, MA, USA, MIT Press, 1988.

[19] McCulloch, W.S. and W. Pitts. *A Logical Calculus of the Ideas Imminent in Nervous Activity*, Bulletin of Mathematical Biophysics, Number 5, 1943, pp.115-133.

[20] Miller, D.M. *A Simple Switch Level Simulator and Its Applications to Stuck-Open Faults in CMOS*, Second Technical Workshop on New Directions for Integrated Circuit Testing, Winnipeg, April 1987. pp. 195-205.

[21] Minsky, M. and S. Papert. *Perceptrons*, Cambridge, MA, USA, MIT Press, 1969.

[22] Newell, A. and H. Simon. *Computer Science as Empirical Inquiry: Symbols and Search*, Mind Design, USA, MIT Press, 1981, pp.35-66.

[23] Pedrycz, W. *A Fuzzy Cognitive Structure for Pattern Recognition*, Pattern Recognition Letters, to appear.

[24] Pedrycz, W. *Course Notes for Recent Advances in Computer Engineering*, University of Manitoba, 1989.

[25] Pedrycz, W. *Direct and Inverse Problems in Comparison of Fuzzy Data*, Fuzzy Sets and Systems, vol.34 1990, pp.233-236.

[26] Pedrycz, W. *Fuzzy Control and Fuzzy Systems*, Research Studies Press, J. Wiley, New York, 1989.

[27] Pedrycz, W. *Selected Issues of Frame of Knowledge Representation Realized by Means of Linguistic Labels*, submitted to the Journal of Intelligent Systems.

[28] Podaima, B.W. and R.D. McLeod. *Weighted Test Pattern Generation for Built-In Self Test using Cellular Automata*, Third Technical Workshop on New Directions in Testing, Halifax, October, 1988.

[29] Rennels, D.A. *Fault-Tolerant Computing - Concepts and Examples*, IEEE Transactions on Computers, Vol. C-33, Num. 12, Dec. 1984, pp. 1116-1129.

[30] Rubin, S. R. *An Integrated Aid for Top-Down Electrical Design*, Fairchild Laboratory for Artificial Intelligence Research, Palo Alto California, 1983.

[31] Schneider, R. *An Interactive MOS Digital Timing Simulator with an APL User Interface*, M.Sc. Thesis, Department of Electrical Engineering, University of Manitoba, 1985.

[32] SDA Systems. *SDA System Manual*, USA, 1988.

[33] Simucad, Inc. *Silos II Logic and Fault Simulator User's Manual*, USA, 1988.

[34] Stauffer, D. *Introduction to Percolation Theory*, Taylor & Francis Ltd., London, England, 1985.

[35] Tanenbaum, Andrew S. *Structured Computer Organization*, Prentice Hall, Englewood Cliffs, N.J, 1984. pp. 124-126.

[36] Togai, H. and H. Watanabe. *Expert System on a Chip: An Engine for Real-Time Approximate Reasoning*, IEEE Expert, 1, 1986, pp. 55-62.

[37] Watanabe, H. and W.D. Dettloff, K.E. Yount. *VLSI Chip for Fuzzy Logic Inference*, Proc. 3rd Intern. Fuzzy System Assoc. Congress, August 6-11, 1989, pp.292-295.

[38] Weste, N. and K. Eshraghian. *Principles of CMOS VLSI Design: a systems perspective*, Addison-Wesley, Reading, Mass., 1985.

[39] Williams, T.W. *VLSI Testing*, IEEE Computer, 1984, pp.126-136.

[40] Xilinx, Inc. *The Programmable Gate Array Design Handbook*, Xilinx Inc., USA, 1986.

[41] Yamakawa, T. *A Simple Fuzzy Computer Hardware System Employing Min and Max Operations - A Challenge to 6th Generation Computer*, Proc. 2nd IFSA Congress, Tokyo, 1987, pp.827-830.

[42] Yasunaga, M. et al. *Design, Fabrication and Evaluation of a 5-Inch Wafer Scale Neural Network LSI composed of 576 Digital Neurons*, International Joint Conference on Neural Networks, San Diego, Califonia, USA, 1990, pp.II527-II535.

[43] Yestrebsky, J., and P. Basehore, J. Reed. *Neural Bit-Slice Computing Element*, Micro Devices, Lake Mary, Florida, USA.

[44] Zadeh, L.A. *Fuzzy Sets*, Information and Control, Vol.8, 1965, pp.338-353.