A VLSI Hardware Neural Accelerator using Reduced Precision Arithmetic

Zoe F. Butler

Thesis submitted for the degree of

Doctor of Philosophy

University of Edinburgh

October 1990

Acknowledgements

I would like to acknowledge the help and support I have received from many people during the time taken to complete the research for this thesis.

Firstly, in the department, I would like to thank my supervisor Alan Murray for his help, patience and guidance in the last four years.

Thanks are due to Alan Gundlach, Bill Gammie and the Edinburgh Microfabrication Facility for fabricating the wafers required for the research and to Ron Mackie for providing help with the wafer testing and advice on hardware design.

The help of Keith Manning, John Hannah and particularly Henry Bruce was much appreciated. I would like to thank them for their advice on the VMEbus and hardware designs.

Stuart Anderson is to be thanked for giving me a crash course on the ES2 Solo chip design and Sandy Alexander for all his tips in "C" programming. I would also like to thank the Department of Electrical Engineering for providing all the facilities to complete the research.

Secondly, thanks go to the Science and Engineering Research Council for their financial support and to British Aerospace in Filton for CASE sponsorship.

Thirdly, I would like to thank my father for proof reading this thesis and to my parents for their continual support and help throughout my time at university.

Declaration

The work presented in this thesis was carried out entirely by the author, unless indicated otherwise.

Abstract

A synthetic neural network is a massively parallel array of computational units (neurons) that captures some of the functionality and computational strengths of the brain. The functions that it may have are the ability to consider many solutions simultaneously, the ability to work with corrupted or incomplete data without any form of error correction and a natural fault tolerance, which is acquired from the parallelism and the representation of knowledge in a distributed fashion giving rise to graceful degradation as faults appear.

A neuron can be thought of, in engineering terms, as a state machine that signals its "on" *state* by the presence of a voltage on its output and signals its "off" *state* by the absence of a voltage. The level of excitation of the neuron is represented by its quantity of *activity*. The activity is related to the neural state by an *activation function*, which is usually the "sigmoid" or "S-shape" function. This function represents a smooth switching of neural state from off to on as the activity increases through a threshold. Direct stimulation of the neuron from outside the network and contributions from other neurons in the network will change the level of activity. The levels of firing from other neurons to a receiving neuron are weighted by interneural synaptic weights. The weights represent the long term memory storage elements of network. By altering the value of the weights, information is encoded or "learnt" by the network, which adds to its store of knowledge.

There are three broad categories into which neural network research can be divided. These are mathematical description and analysis of the dynamical learning properties of the network, computer simulation of the mathematical models and the VLSI hardware implementation of neural functions or classes of neural networks. It is the final category into which the main thrust of this thesis falls.

The research presented here implements a VLSI digital neural network as a neural accelerator to speed up simulation times. The VLSI design incorporates a parallel array of synapses. The synapses provide the connections between neurons. Each synapse effectively "multiplies" the neural state of the receiving neuron by the synaptic weight between the sending neuron and the receiving neuron. The "multiplication" is achieved

by using *reduced precision arithmetic* that has a "staircase" activation function modelled on the sigmoid activation function and allows the neuron to be in any one of five states. Therefore, with little loss in precision, the reduced precision arithmetic avoids using full multiplication, which is expensive in silicon area. The reduced arithmetic synapse increases the number of synapses that can be implemented on a single die.

The VLSI neural network chips can be easily cascaded together to give a larger array of synapses. Four cascaded chips resulted in 108 synapses in an array. However, this size of array was too small to perform neural network learning simulations. Therefore the synapse array has been configured in a *paging architecture*, that has traded off some of the high speed of the chips (upto 20MHz) against increased network size. The synapse array has been wired with support circuitry on to a board to give a *neural accelerator* that is interfaced to a host Sun computer. The paging architecture of the board allows a network of several hundred neurons to be simulated. The neural accelerator is used with the delta learning rule algorithm and results show its increased acceleration to be up to two orders of magnitude over equivalent software simulations.

Table of Contents

Table of Figures

Table of Tables

Introduction

A neural network can be viewed as large numbers of computational units (neurons) operating in parallel arrays, the functionality of which is based loosely on what is understood to be used in the nervous system. The brain's neurons, which form the basic computing elements, are several orders of magnitude slower than silicon logic gates, but are organised so they are able to perform some computations many times faster than the fastest digital computers now in existence. The brain appears to do this via its massive parallelism. As there are huge numbers of neurons, the weak computing powers of these many slow elements combine to form a powerful resultant computational machine. The scientific desire to understand human behaviour and the brain construction has motivated much of the past and present research into neural networks. Some of the properties a synthetic neural network may aspire to mimic are the ability to consider many solutions simultaneously and the ability to work with corrupted or incomplete data without explicit error correction. Neural networks also have a natural fault tolerance, which arises from the parallelism and distributed knowledge representation giving rise to graceful degradation as faults appear.

In engineering terms, a biological *neuron* is a unit that signals its *state* by the presence ("on") or absence ("off") of a voltage on its output, or *axon*. It decides its state by computing its *activity*, which represents the level of excitation of the neuron. The *state* is related to the activity by an *activation function*. The *activation function* is generally the "sigmoid" or "S-shape" function which represents a smooth switch of neural state from off to on (not firing to firing) as the activity increases through a threshold. The level of activity can be altered by direct stimulation of the neuron from outside the network and by contributions from other neurons in the network. The contributions from other neurons are weighted by interneural synaptic weights, which are in effect, the long term memory storage elements of the network. Information is encoded or "learnt" by the network by altering the value of the weights to add to its store of knowledge.

The present research into neural networks falls into three broad categories. The first is that of mathematical description and analysis of the dynamical and learning properties of the networks. The second category, which is probably the largest, covers research

using computer simulation based on, for example, array processor or other supercomputer architectures to model and extend the mathematical descriptions. The third group of research, into which the thrust of this thesis falls, aims to implement either particular neural functions or classes of neural network in LSI/VLSI hardware.

The LSI/VLSI neural network circuits at present use planar silicon technology, although this technology is almost certainly not the ultimate medium in which neural networks will fully realize their power. Three dimensional materials are more suited to the three dimensional form of a neural network, but there is no solution yet that would enable these materials to be a suitable medium for hardware circuits. However, hardware neural networks can be easily designed and manufactured in silicon VLSI and are able to make use of developments in network design and learning procedures to solve real problems.

The approach in this thesis is to implement a VLSI neural network as a digital neural accelerator to speed up network simulation times. This involves the design in VLSI of a network consisting of a parallel array of synapses. The synapses provide connections between neurons. Each synapse holds locally the synaptic weight between the sending neuron and the receiving neuron and has a means of "multiplying" the weight by the neural state of the receiving neuron. This generally involves the full multiplication of the synaptic weight by the neural state, which can be expensive in silicon area, allowing only a small number of synapses on a die. The *reduced precision arithmetic* approach in this thesis uses a "staircase" activation function modelled on the sigmoid activation function, that allows a neuron to be any one of 5 states. It avoids the use of full multiplication, thus reducing the size of a synapse and greatly increasing the number of synapses that can be integrated on a single die.

Simulation in software of the 5-state activation function obtained from using the reduced precision arithmetic showed that its performance was only degraded a little compared to that of the sigmoid activation function and there was little loss in precision in neural network pattern learning simulations. The simulation results justified the design of a neural network using the reduced precision arithmetic. The nature of this approach lends itself to a VLSI, bit-serial, digital design. A single phase clocking scheme capable of speeds up to 20MHz was, at the time, being developed in the department and was used in the integrated circuit design.

The main attraction of the reduced precision arithmetic is that it provides a means of building a fast, digital hardware neural network, that can be used as a hardware accelerator to reduce the lengthy simulation times of equivalent simulations run totally in software. The problem with software models is that neural networks with only tens of neurons can take many hours to simulate. The main body of this thesis describes how a significant speed-up is achieved by using a VLSI neural network operating in a "neural accelerator board".

The thesis, first of all, provides a brief overview of the function of biological neurons in Chapter 1. The history and background of neural networks is given, from the original ideas about perception and memory up to the present day knowledge, along with the most well known learning algorithms and neural network models that are used today. Chapter 2 explains the motivation behind VLSI implementations and gives an account of the research into hardware neural networks. The research covers digital, analogue, pulse-stream and optical aspects of implementation.

Chapter 3 describes the details of the reduced precision arithmetic and how the 5-state activation function relates to the sigmoid activation function. The simulation procedure that was used to compare the performance of the two activation functions is described. An analysis of the results shows the suitability of the reduced precision arithmetic to be implemented in VLSI.

The VLSI design of the synapse array in Chapter 4, reports two different design procedures. The first design uses a fully custom computer aided design layout tool with 3μm CMOS technology and the second uses the the European Silicon Structures silicon compiler, Solo, for the complete design and layout procedure in 2μm technology. Simulation results of the fabricated devices for each manufacture are presented.

Chapter 5 specifies how the neural network chips can be cascaded together on a neural board to achieve a larger array of synapses. It explains how the synapse array is configured in a *paging architecture*, that trades off some of the fast operating speed of the chips against network size to create an array of several hundred neurons.

The results reported in Chapter 6 compare the performance of the the hardware neural accelerator in a program with a learning procedure, to an equivalent software 5-state activation function network and a software sigmoid activation network. Finally,

Chapter 7 draws conclusions about the design and performance of the hardware accelerator board and suggests improvements than could be made to increase its speed and efficiency.

Chapter 1

Introduction to Neurons and Neural Networks

The human brain is one of most complex structures known. There has been much research over the centuries by anatomists, physiologists and psychologists into its development, structure, the electrical and chemical phenomena that take place in its nerve cells, and into its unique behaviour. Until the beginning of the twentieth century, the brain was believed to be an exception to the basic biological principle, that all tissues are made up of individual cells. Now it is thought to consist of 100 billion (10^{11}) individual neurons arranged in several hundred distinct groups, with 95% in the cerebral cortex [1, 2]. It is this massively parallel computational ability of the brain to perform a wide range of tasks that has urged researchers to build intelligent machines.

This chapter gives a brief introduction to biological neurons and their function. Much of the biological terminology is used in the description of synthetic neural network models and although it is not essential to have a good understanding of the nervous system, some familiarity with the jargon is useful.

An outline of the history of synthetic neural networks in the second part of the chapter, shows how the understanding of the nervous system and brain function developed and inspired early researchers to develop mathematical neural models and later in the 1950's, to build physical models that could perform some sort of learning.

The last section gives an overview of synthetic neural network models and learning procedures used today that are implemented either mathematically, in software or in hardware.

1.1. The Neuron

A typical neuron [2, 3] consists of a cell body containing the nucleus and a number of fibres extending from it as shown in figure 1.1. The neuron transmits information to other cells by sending its activity out through only one fibre, the **axon**. All the other fibrous extensions from the cell body, the **dendrites**, receive information from other neurons. An axon generally divides into a number of small fibres that end in terminals.

Each terminal forms a synapse with a dendrite or the cell body of another neuron and is the point where information is transmitted from one neuron to another. A small space, the **synaptic cleft,** separates the axon terminal from the dendrite or cell body of the other neuron with which it synapses.

1.1.1. The Axon

An **axon** has two essential functions in the neuron. One is to conduct information in the form of the action potential, which is the process axons use to carry information from the neuron's cell body to the synaptic terminals, in order to trigger synaptic transmission. The other function is to transport chemical substances from the cell body to the synaptic terminals and backwards from the synaptic terminals to the cell body.

The resistance of the neuron's cytoplasm is sufficiently high that signals cannot be transmitted along the axon greater than 1 mm before their information is lost. For this reason, the larger axons in the human brain are surrounded by a thin insulating sheath called **myelin**. The myelin increases the speed of conduction of the action potential along the axon by reducing the capacitance between the cytoplasm and the extra-cellular fluid [4]. The sheaths are made up from non-neural cells called **Schwann cells** which are approximately 1mm in length and in general, the larger the the diameter of the axon, the thicker the myelin, up to a possible 100 layers. Gaps of 1μm which occur in the Schwann cells, are **nodes of Ranvier** These nodes act as repeater sites where the signal is periodically restored. A single myelinated fibre can carry signals the length of the longest axons, which may be a metre or greater. Although myelination is the most important distinguishing feature of larger axons, axons of less than 1μm in diameter are unmyelinated.

1.1.2. Dendrites and Synapses

Dendrites constitute all the fibres extending out from the neuron, excluding the axon and serve to extend the neuron's receptive surface. In the cerebral cortex, many of the dendrites have **dendritic spines** which form synapses with axon terminals of other neurons as in figure 1.2. The dendritic spine forms the postsynaptic part and the axon terminal forms the presynaptic part of the synapse. They are separated by the synaptic cleft which is about 20nm wide. The dendritic spine synapses are thought to be

excitatory and synapses that cluster on the cell body are thought to be inhibitory. When a synapse is active and transmits information, vesicles in the axon terminal fuse with the presynaptic membrane and release neurotransmitter into the cleft. The transmitter molecules diffuse across the narrow gap and attach to specific chemical receptor molecules on the postsynaptic membrane, which activates the postsynaptic target cell.

1.1.3. Cell Membrane and Action Potential

The neuron cell membrane has properties that allow it to conduct and transmit information to other neurons. One of these properties is ion channels through the membrane that allow sodium (Na^+), potassium (K^+) and chloride (Cl^-) ions to pass in and out of the cell. The axon has a resting potential of about -70 mV. This is due mainly to a large concentration of K^+ ions inside the cell and a smaller concentration of K^+ ions outside the cell and involves a passive process of ions moving through permanently open ion channels. The distribution of K^+ ions is due in turn to negatively charged proteins in the cell. The distribution of Na^+ and Cl^- ions also contributes to the resting potential, but is less important than that of K^+ ions.

The action potential in a typical neuron begins at the point where the axon leaves the cell body and travels to the axon's terminal. The Na^+ gates open for about 0.5ms and Na^+ ions enter the cell increasing its potential to +50 mV relative to the outside as in figure 1.3. The Na^+ gates then close and the potential goes back towards the resting level. This growth and decay of the action potential is termed the **absolute refactory period.** During this period, the axon cannot be electrically stimulated to generate another action potential. Meanwhile, the K^+ gates have opened, some of the K^+ moves out and the membrane potential becomes even more negative (−75 mV) for a few milliseconds. This is the **after potential** or **relative refactory period.** The axon can be electrically activated in the period of the after potential, but it requires a stronger than normal stimulus. This is the **relative refactory period.**

The activation of a single synapse on a neuron will not cause it to develop an action potential. Enough synapses have to be activated together and exert their influence on the receiving neuron. The activations of all the synapses are summed together. If they are activated repeatedly at a fast enough rate, they will sum over time and generate a

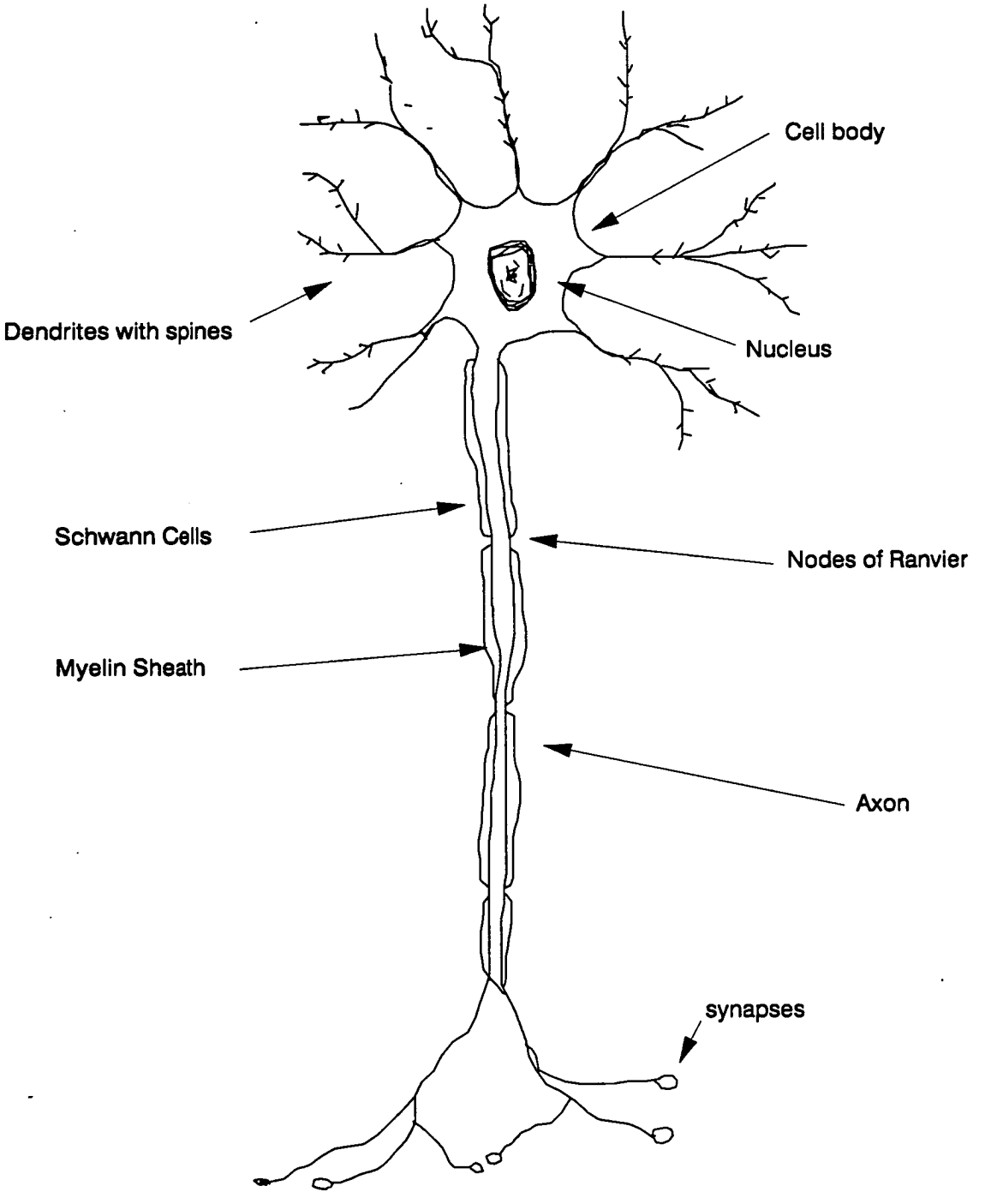


Figure 1.1 Major parts of the neuron

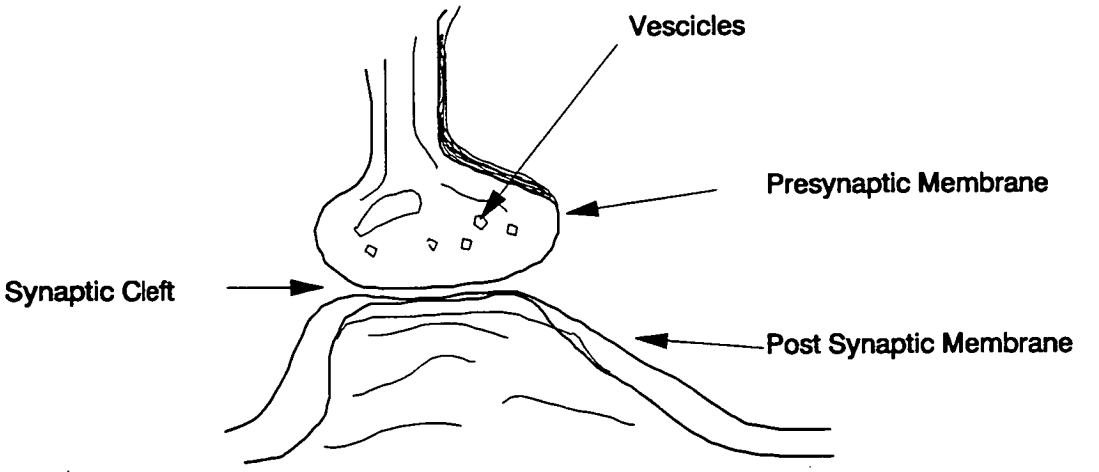


Figure 1.2 Pre-synaptic and post-synaptic membranes

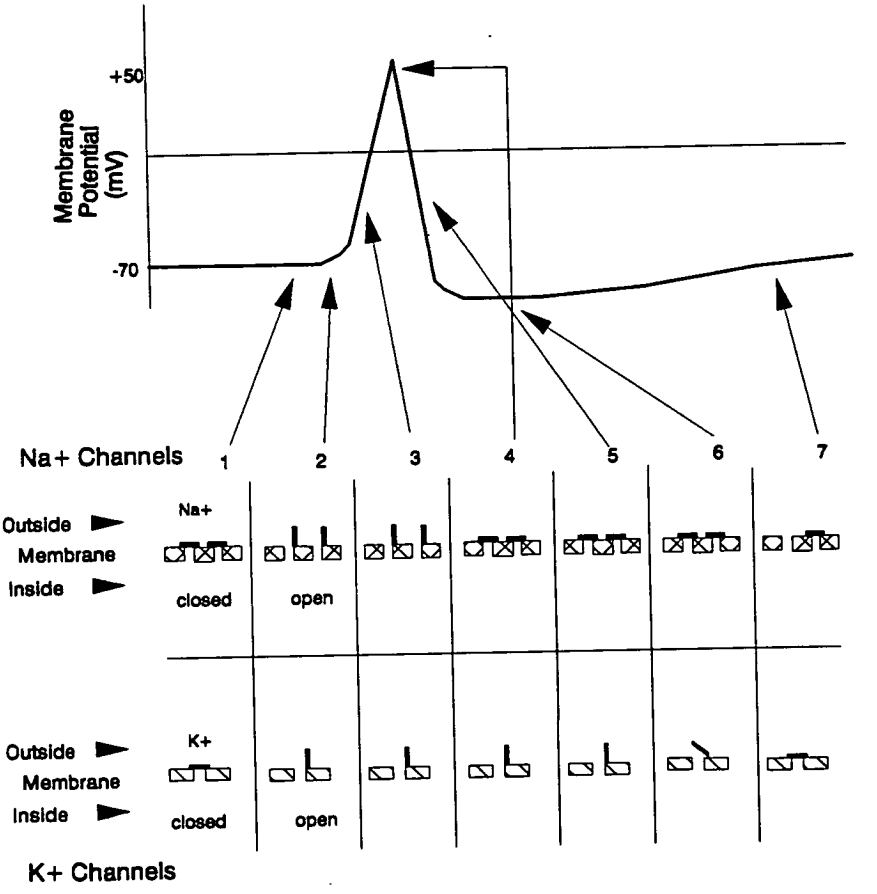


Figure 1.3 Development and firing of the action potential

post synaptic potential large enough to make the cell fire. A normally functioning neuron is continuously summing information over time and space and "deciding" whether or not to fire.

1.2. The History of Neural Research

Research into neural and brain function has a long history starting from the observations of Hippocrates at 500 BC and Plato and Aristotle at around 400 BC, who associated memory with the sensory processes. The attempt to understand the neural structure has captured the interest of philosophers, psychologists, mathematicians, physicians and anatomists, but first major contributions providing an early conceptual framework for the study of nerve net action were undertaken by Pavlov [5, 6], the famous Russian psychologist, with his research on conditioned reflexes and Rashevsky [7], with a mathematical description of biological processes. From then, neural research and understanding has been expanding right up to the present day.

1.2.1. Early Biological Research

The end of the Classic period during the 2nd century was marked by Galen, a Greek physician, who proved that the brain was the seat of intelligence and memory [8]. The increase in knowledge from Hippocrates to Galen was considerable in detail, but there were little changes in attitude. Galen coordinated all that was known in medicine and science, which influenced thinking for the next fourteen centuries. Physiological knowledge of the brain showed few significant advances until Descartes in 1596 first recognised a conditioned reaction, "*when one sees an object that has previously been at the time an emotion has been experienced, it will induce that emotion .... there is a connection between the stimulus and the response being made through a definite path; this connection is the fundamental process of the nervous structures in the body*". This was the basis on which study of the nervous system was established. Over 300 years later, Pavlov started his work on the *conditioned reflex,* the linking up of the action of a new stimulus with an *unconditioned* (or inborn) reflex, using Descartes' idea of the nervous reflex. To show this Pavlov experimented with dogs. He used an *unconditioned* signal of a brief electric shock in a dog's paw to tell it that food was about to appear. This signal was alien to food, but the animal soon learned to salivate on receiving the shock and wanted to eat. Thus he had transformed apparent pain to

overt pleasure.

Research into memory and brain function continued steadily through the 19th century. Some of the main contributions were from James Mill (1773 - 1836) who wrote "*memory is nothing more than the fact of recall through association. It is the appearance of a sensation that can be associated with the time and place it has been presented previously*". Gall, a well known anatomist at the time, asserted that human "faculties" were located in strictly localised areas of the brain [9] and in 1861, Paul Broca, a French anatomist, localised for the first time a complex mental function to a particular part of the brain.

The work of J. H. Jackson [10] in the 1870's put forward the hypothesis that connections in the brain were physical entities that could be changed and that it was likely that a part of the brain's network was prewired to deal with a certain processing task. If that task became irrelevant, then that part of the network could be used for something else. Jackson pointed out this view as a difficulty for strict localisationist views that had become that popular at the time. Some of the earliest roots of the PDP (Parallel Distributed Processing) approach came from Jackson [10] and Luria [11], the Russian psychologist and neurologist. Luria put forward the idea of the *dynamic functional system*. On this view every behavioural or cognitive process resulted from the coordination of a large number of different components, each roughly localised in different regions of the brain, but all working together in dynamic interaction.

In 1913 Henri Poincare [12], a French mathematician, attempted to explain neural action from an atomical point of view and in 1938, Rashevsky [7] gave the first mathematical description of the biological processes. Rashevsky showed how certain logical operations might be carried out by simple nerve arrangements as in figure 1.4. This shows how an exclusive-or function is mechanised by inhibitory and excitatory connections. He also gave an explanation for short term memory by means of recirculating neuron loops, in which an impulse, once initiated, would continue to cycle indefinitely or until terminated by an inhibitory pulse. Another psychologist, Thorndike [13] in his neural research found that "*connections that words have in a person's experience produce modifications in his brain .... the modifications consist of changes at the points where one neuron transmits to another*".

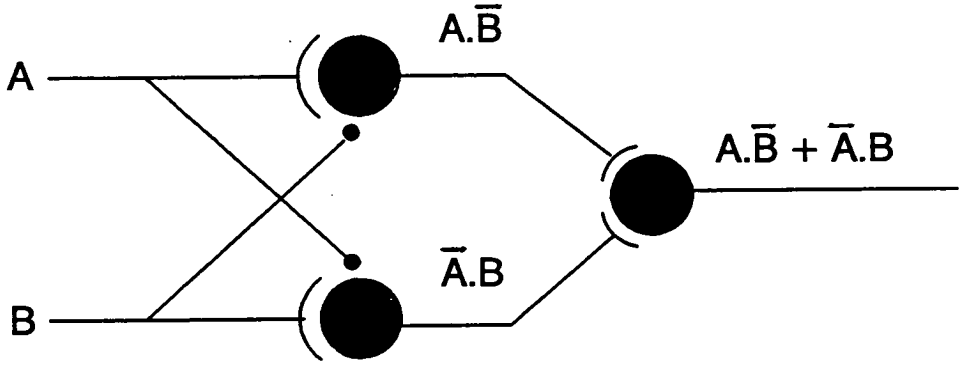


Figure 1.4 Rashevesky's exclusive-OR network using excitatory and inhibitory connections

The work of Lashley [14, 15] may be seen as the beginnings of modern experimental physiological psychology. He moved away from the Pavlovian reflex and worked on the search for the engram [16] and the localisation of function in the rat's brain. He traced the representation of remembered events to the cerebral cortex and proved that the degree of degradation of memory was roughly proportional to the area of cortex, thus showing the distributed representation of memory. He concluded that there was relatively little localization of function in the cerebral cortex. Lashley's paper *In search of the Engram* emphasised the diffuseness of neural mnenomic processes and insisted that no special cells were reserved for special memories. He conceived brain operation as large scale patterns of activation involving a great many active neurons leading to other large patterns of activity.

Two hypotheses, which have become the basis of many nerve net models are the work of Donald Hebb [17-19], who was a student of Lashley in the 1930s. Hebb postulated that the synaptic junction was the site of permanent memory, that consisted of the value of the attenuation (strength) of the junction and that memory of any event was distributed within a network residing in the small changes in strength which occur as the result of the event impinging upon a large number of synapses. He suggested the following rule for the change in strength of a junction as the result of activity: "*When an axon of cell A is near enough to excite a cell B and repeatedly B takes part in firing*

it, some growth process or metabolic change takes place in one or both such that A's efficiency, as one of the cells firing B is increased". Hebb also postulated the formation of what he called "cell assemblies", where there were interconnected, self-reinforcing subsets of neurons that formed the representation of information in the nervous system. Single cells might belong to more than one assembly, depending on the context. Multiple cells could be active at once, corresponding to complex perceptions or thoughts. He said there was a distributed representation at the functional level as well as the anatomical level. Before Hebb's work, it was believed that some physical change must occur in a network to support learning, but it was not clear what this change could be. Hebb's ideas about the nervous system remained untested until it became possible to build some form of simulated network to test learning theories.

1.3. Neural Network Modelling and Learning Procedures

One of the first neural models was introduced by McCulloch and Pitts [20] who, by using Boolean Algebra showed how neural-like networks could compute. They used the "all or none" character of nervous activity, with the activity of any inhibitory synapse preventing the excitation at a given time and allowing only a fixed number of synapses in any given period to excite the receiving neuron.

The neural model of A. E. Roy stored information in binary pulses and on being presented with a section of a message stored previously, it would recall the rest of the message. A discussion of the model can be found in [21-23].

1.3.1. The Perceptron Learning Theorem

The first attempt to build a simulated network was the learning machine of Edmonds and Minsky in 1951, which consisted of hundreds of tubes, motors and automatic electric clutches, with its memory stored on 40 controls knobs. Details of the function can be found in detail in [24]. Rosenblatt, an acquaintance of Minsky, achieved the first neuron-like learning model with the perceptron [25]. He analysed his models mathematically and ran digital simulations of the three-layered perceptron, its environment and memory modification rules in a digital computer program. Rosenblatt's three-layered perceptron is a single transmission network containing 3 types of signal generating unit as in figure 1.5. This shows the basic organisation of

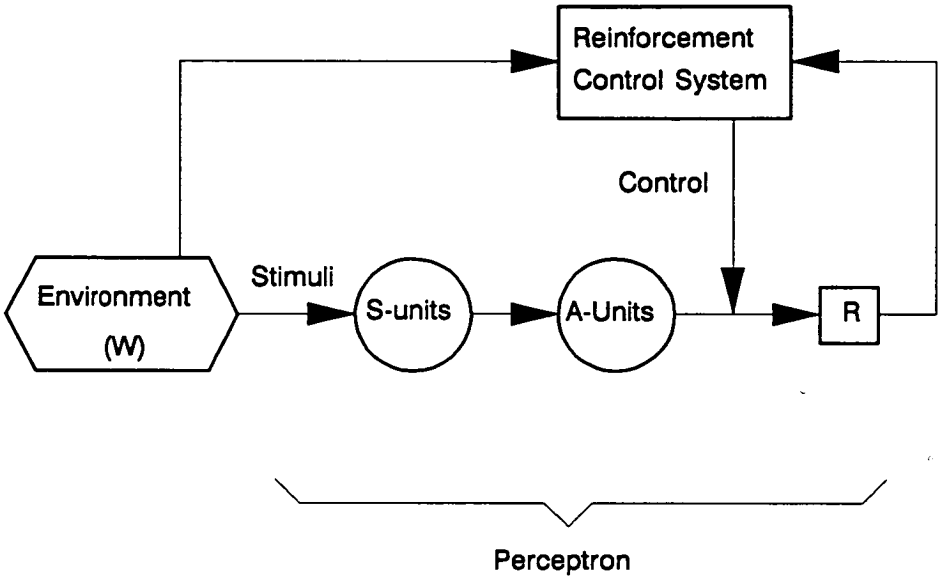


Figure 1.5 Rosenblatt's three-layered perceptron

the experimental system where the output of the perceptron is capable of modifying its stimulus environment. It starts with an S - (sensory) unit (eg. a "retina") which projects to higher levels. The S - unit is a transducer responding to physical energy and forms the first layer. This is connected to a second layer, an association area or A - units, by random, localised connections with fixed synaptic weights. A number of cells in the region of the S - units project onto a single A - unit in the higher layer. The A - unit is a logical decision element, which generates an output signal if the algebraic sum of its inputs is greater than a threshold quantity, $\theta > 0$. The association layer is reciprocally connected to a third layer of R - (response) units. The R - (response) unit emits the output:-

$$r^* = \begin{cases} +1 & \text{if } \Sigma \text{ input signals} > 0, \text{ and} \\ -1 & \text{if } \Sigma \text{ input signals} < 0. \end{cases} \tag{1.1}$$

If the sum of the inputs is zero, the output is zero or indeterminate. The activation of the appropriate R - unit for a given input pattern or class of input patterns is the operation goal of the perceptron. During learning, the values (weights) stored in the r.c.s. (reinforcement control system) are changed when they do not correspond to some

arbitrary *desired* response r_i, for the given input pattern. The perceptron uses an error correcting system in that a correction is made in accordance with the rules of a specified reinforcement system on the network only if an erroneous response is obtained. When it is necessary to correct a response, the strength of the weights connected to that output change simultaneously. This will yield a solution to the input stimulus within a finite time. Rosenblatt commented that the simple three-layered perceptron is capable of learning any type of classification or associating any responses to stimuli. Therefore for a multi-layered perceptron, ie. a perceptron with two or more layers of association units, to offer any functional advantage over the three-layered perceptron there would have to be an increase in efficiency of such responses.

Minsky and Papert undertook a careful mathematical analysis of the one layered perceptron [26]. The machine they examined is in figure 1.6, which shows a set of binary threshold units with fixed connections to a subset of units in the retina.

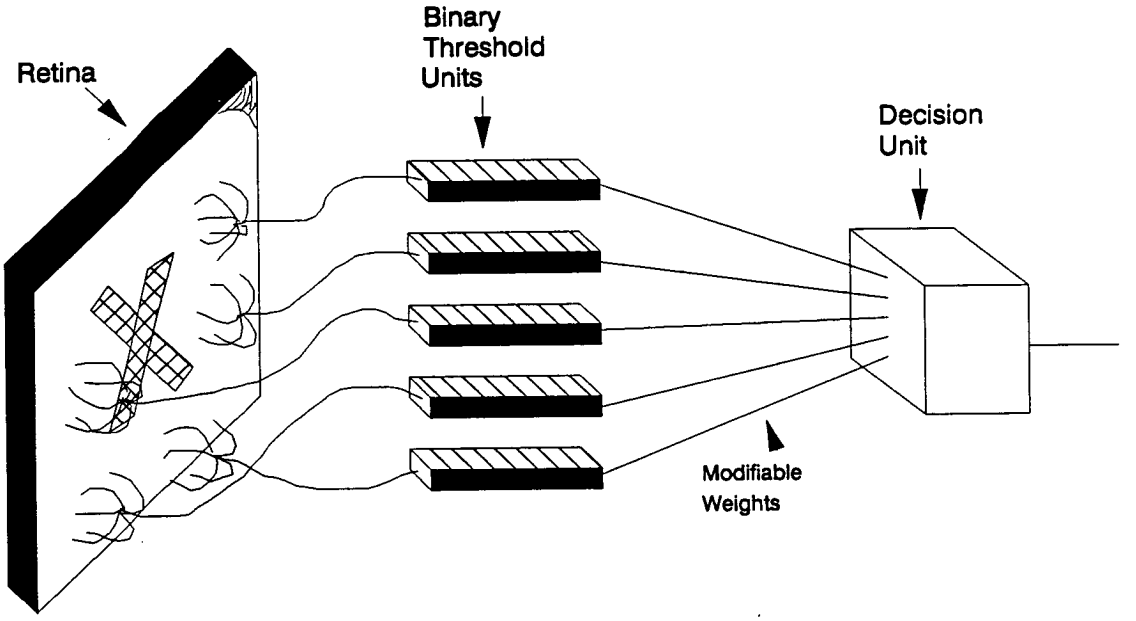


Figure 1.6 Perceptron analysed by Minsky and Papert

From this analysis, Minsky and Papert showed which functions it could and could not compute and demonstrated the importance of a mathematical approach to analysing computational systems. They also argued that there was no indication how a learning procedure could be applied to multi-layered networks. The analysis suggested that perceptron like devices would have no future in artificial intelligence.

Perceptrons and early related work had been in decline for several years before the work of Minsky and Papert, as perceptrons had failed to achieve much beyond their initial success. Practical results had failed to materialize and the Minsky and Papert book "Perceptrons" [26] seemed to prove to the scientific community that there was little future in neural networks. However, perceptron-like models can be successful at modelling a number of aspects of perception and cognition. Multilayered networks [19], which have input, output and hidden units can provide solutions to problems by the internal representation in the hidden units and learning can be achieved by the Generalised Delta Rule using back propagation. This is described in section 1.3.3.

1.3.2. The Delta (Widrow-Hoff) Learning Algorithm.

Neural network models generally are networks of processing units that are connected together in some way. An activation rule combines the inputs applied to a unit with its current state to produce a new level of activation for that unit. A learning rule modifies the existing patterns of connectivity between units through experience. These rules are the bases for parallel distributed processing in that some of the models' units carry out their computations at the same time. Usually the units will be one of three types: *input, output* and *hidden* units. Input units receive inputs from external sources, which may be sensory or otherwise. Output units send signals directly out of the system and hidden units have inputs and outputs from within the system with no external connections. They are connected between the input and output units (sometimes in layers), and are not "visible" to the outside world.

Many of the learning rules for these types of models are variants of Hebb's Learning rule given in section 1.2.1. This can be generalised to: *A connection or synaptic weight,* w_{ij}, *increases or decreases in proportion to a reinforcement signal,* r, such that:-

$$w_{ij}(t+1) = w_{ij}(t) + \eta r_i(t) \qquad (1.2)$$

where η = reinforcement signal to synapse i, at time t and determines the change in connection weight.

The Widrow-Hoff or Standard Delta Rule [27,28] was based on this theory. The Widrow-Hoff system used linear threshold units with random variable connection strengths. Each linear threshold computed a weighted sum of activities of the inputs

times the synaptic weight, plus a bias element. If the sum was greater than zero, the output became +1. If it was equal to or less than zero, the output was −1. It then compares this to a *desired output* or *target* vector. If there is no difference, no learning takes place. Otherwise the weights are changed to reduce the difference. The rule for changing the weights, w_{ji}, between any two units i and j following the presentation of an input/output pair s is given by:-

$$\Delta_s\, w_{ji} = \eta\, (t_{sj} - o_{sj})\, i_{si} = \eta\, \delta_{sj}\, i_{si} \tag{1.3}$$

where t_{sj} is the target input for the jth component of the output pattern for the pattern s, o_{sj} is the jth element of the actual output pattern produced by the presentation of input pattern s, i_{si} is the value of the ith element of the input pattern, and $\Delta_s w_{ji}$ is the change to be made to the weight from the ith to the jth unit following presentation of input pattern s. This learning procedure applies only to models with no hidden units.

1.3.3. The Generalised Delta Rule

The Standard Delta Rule uses two layer associative systems, that have only input and output units and no hidden units, and is useful in applications where similar input patterns can be mapped to similar output patterns. Where the mappings are very different, a network without the internal representation would be unable to perform the necessary computation.

Minsky and Papert [26] in their analysis of conditions under which such systems are capable of carrying out required mappings, showed that in a large number of cases, networks of this kind were unable to solve problems. They also showed that if there is a layer of simple perceptron-like hidden units as in figure 1.7, the input information to the input units is recoded to an internal representation, which generates the appropriate output pattern. The Generalised Delta Rule [19] allows learning to take place in systems with hidden units. It uses a *semi-linear* activation function in which the output of a unit is a non-decreasing and differentiable function of the net total output, as in equation 1.4 below:-

$$o_{sj} = \frac{1}{1 + \exp\left[\, -\left(\sum_i w_{ij} o_{si} + \theta_j\right)\right]} \tag{1.4}$$

The Generalised Delta Rule has the same form as the Standard Delta Rule in equation

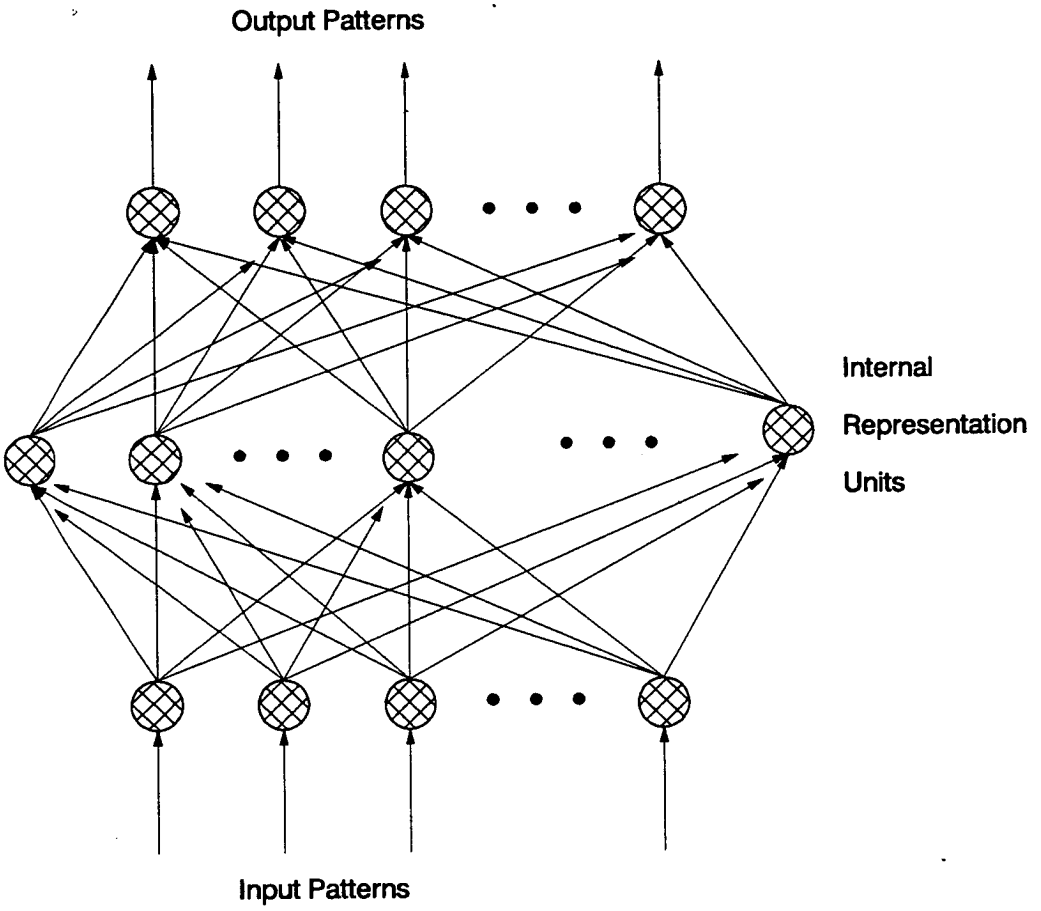


Figure 1.7 A Multilayer network with input, output and hidden units

1.3. The weight on each line should be changed by an amount proportional to the product of an error signal, δ, available to the unit receiving an input along that line and the output of the unit sending activation along that line. The error signal, δ_{sj}, for an output unit is:-

$$\delta_{sj} = (t_{sj} - o_{sj})o_{sj}(1 - o_{sj}) \tag{1.5}$$

and the error for an arbitrary hidden unit, u_j, is given by:-

$$\delta_{sj} = o_{sj}(1 - o_{sj})\sum_{k}\delta_{sk}w_{kj} \tag{1.6}$$

Two stages of computation are involved in the Generalised Delta Rule. The first stage is as follows:

1. Present the input to the network and allow it to propagate through the network to compute the output, o_{sj}, for each unit.

2. Compare the computed output, o_{sj}, to the target output, t_{sj}, and calculate from equation 1.5, the error signal, δ_{sj} , for each output.

The second stage involves:

3. A backward pass through the network where the error signals are passed to the units and the appropriate weight changes are made. The weight changes are first calculated for all connections that feed into the final layer.

4. When this is done, the δ's for all the units in the penultimate layer are computed. This propagates the error back one layer.

5. The same process is repeated for every layer.

The backward pass allows a recursive computation of δ. The learning rule used for change in weights is calculated from:-

$$\delta w_{ji}(n+1) = \eta\ (\delta_{sj} o_{si}) + \alpha\ \delta w_{ji}(n) \tag{1.7}$$

where η is the learning rate and α is a constant which determines the effect of past weight changes on the current direction of movement in weight space. n is the presentation number. This equation is a modified version of the Standard Delta Rule.

1.3.4. Hopfield Model

The Hopfield model can be regarded as a content addressable memory type [29] in that the exact contents of the memory can be retrieved on the basis of sufficient partial or partly erroneous information being presented to it. For example, the system has locally stable points X_a, X_b, (ie. contents in memory). If the system then is presented with $(X = X_a + \Delta)$ it will proceed in time until $X \simeq X_a$, ie. $(X_a + \Delta)$ represents a partial knowledge of X_a and the system then generates the total information X_a.

The processing units in Hopfield's original model are 2-state neurons, the state $V_i = 1$ ("firing at maximum rate") and $V_i = 0$ ("not firing"). The instantaneous state of the system is specified by listing the N values of V_i $(i = 1.....N)$. Each neuron has a fixed threshold U_i such that:-

$$\begin{matrix} V_i \rightarrow 1 \\ V_i \rightarrow 0 \end{matrix} \left\{ \text{if } \sum^{j \neq i} w_{ij} v_j \right\} \begin{matrix} > U_i \\ < U_i \end{matrix} \tag{1.8}$$

Each neuron evaluates randomly and asynchronously, whether it is above or below a

certain threshold, and readjusts it accordingly. The states $V_1 \ldots V_i \ldots V_N$ are the stable states of system.

The model uses an **information storage algorithm** which allows the synaptic weights between neurons to be set for the storage of any particular set of states V^s, s = 1.....n. This is:-

$$w_{ij} = \sum_s (2V_i^s - 1)(2V_j^s - 1) \tag{1.9}$$

The neurons are totally interconnected with $w_{ii} = 0$ and there are no hidden units. From equation 1.9 it can be seen that if two adjacent states are excitatory, the synaptic weight between them is increased. Using equation 1.9, a weights matrix can be formed for states V_j.

There are two limitations to this type of model. The first is that the number of states, s, that a given set of neurons, N, can learn is limited to $s = 0.15N$, otherwise the storage prescription fails. The second limitation is that if a start vector is chosen at random or if it shares many bits in common with another start vector and is allowed to iterate using the weights matrix, sometimes it may fail to "find" one of the stored states. The state that it does finally iterate to is known as a *local minima*.

1.3.5. Wallace-Hopfield Training Algorithm

The problem of the Hopfield storage prescription becoming inexact at small values of s/N has been analysed by Wallace [30]. He has developed a simple iterative algorithm for the Hopfield model which is guaranteed to store exactly any s vectors in a finite number of steps, provided it is known that a solution is possible. Starting with the storage prescription in equation 1.9, the approximate weights for the vectors, V^s, to be stored are calculated. All the vectors are tested to see if they have been stored correctly by iterating equation 1.9 once. This enables an error mask to be calculated for each V_s such that:-

$$\epsilon_i^r = \begin{cases} 1 & \text{if } V_i^s \text{ changes} \\ 0 & \text{if } V_i^s \text{ is stable} \end{cases} \tag{1.10}$$

The storage perscription is then reinforced for those weights wrongly stored, given by:-

$$\Delta w_{ij} = \sum_{s=1}^{N} V_i^s V_j^s \, (\epsilon_i^s + \epsilon_j^s) \tag{1.11}$$

All the vectors, V^s, are tested again with the new w_{ij} and the w_{ij} are modified until convergence has been achieved.

1.3.6. Competitive Learning

Competitive Learning [19] is another learning procedure. Individual units learn to specialise on sets of patterns and thus become *feature detectors* or *pattern classifiers*. The architecture of a competitive learning system uses a set of hierarchical layered units in which each layer connects via excitatory connections with the layer immediately above it. Within a layer, the units are broken into sets of inhibitory clusters, in which all elements inhibit all other elements within the cluster. These elements at one level, compete with one another, to respond to the pattern appearing on the layer below. The more strongly any particular unit responds to an incoming stimulus, the more it shuts down the other members of the cluster.

A number of researchers have developed competitive learning models or variations on models. Examples of these can be found in [31-35]. A general competitive learning model has sets of clusters in a layer, which are of a winner takes all form, such that the unit receiving the largest input achieves its maximum value while all the other units in the cluster are pushed to their minimum value. In general, each unit in a cluster receives inputs from all the units in the layer below and projects outputs to all units in the next higher layer. A unit learns only if it wins the competition with other units in the cluster. Each unit has a fixed amount of weight and learns by shifting weight from the inactive to the active input lines. In von der Malsberg learning rule [31], if a unit wins a competition, each of the input lines gives up some of its weight and the weight is then evenly distributed among the active input lines.

1.3.7. Grossberg's Network

Grossberg [36] has proposed a pair of equations that describe the dynamical behaviour of a set of neurons and their synaptic weights. These equations have a level of generality unmatched by other descriptions of synthetic neural networks and are based on what is known to occur in the brain. The dynamic behaviour of the neurons is

shown by:-

$$\frac{\delta x_i}{\delta t} = -A_i x_i + \sum_{j=1}^{j=n} w_{ij} V_j - \sum_{j=1}^{j=n} \tilde{w}_{ij} V_j + I_i(t) \tag{1.12}$$

where A_i is the passive decay of the activity in the absence of both synaptic and direct external input, w_{ij} ($\tilde{w}_{ij}$) is the excitatory (inhibitory) weight and I_i is a stimuli that can force a state on the network.

The change of synaptic weight over time is given by:

$$\frac{\delta w_{ij}}{\delta t} = -B_{ij} w_{ij} + D_{ij} \tilde{V} u_q(x_i) \tag{1.13}$$

B_{ij} is the passive decay of the synaptic weight. D_{ij} is the learning strength that allows learning to be modulated for each synaptic link, $\tilde{V}$ is a neural "learning signal" and u_q (x_i) is a linear-threshold activation function. The speed of learning is controlled by D_{ij}, but the rate of change of synaptic weights must be much slower than that of the neural states.

Grossberg uses a sigmoid activation function that represents the smooth switching of the neural state V_j from 0 to 1 as the neural activity x_i increases through the threshold value $\tilde{x}$, where T controls the sharpness of the transition, as in equation 1.14:-

$$V_i = \frac{1}{1 + exp[\frac{(\tilde{x}_i - x_i)}{T}]} \tag{1.14}$$

Grossberg has developed a network using his Adaptive Resonance Theory, that forms clusters and is trained without supervision [34,35]. A simplified diagram of the network is in figure 1.8. A binary input is presented to the lower nodes as an exemplar for the first cluster. A second input is then presented and compared to the first cluster exemplar. The dot product of the two exemplars is computed and divided by the number of "1s" in the input. If the ratio is greater than a *vigilance* threshold, the input will be clustered or "classified" with the first exemplar. If the ratio is less than the threshold, the input is considered to be "different" from the first exemplar and is added as a new exemplar. The vigilance threshold can be set between the range 0.0 and 1.0. Inputs are presented sequentially to the network and compared to all stored exemplars and classified in the same way. Each additional new exemplar requires one node and

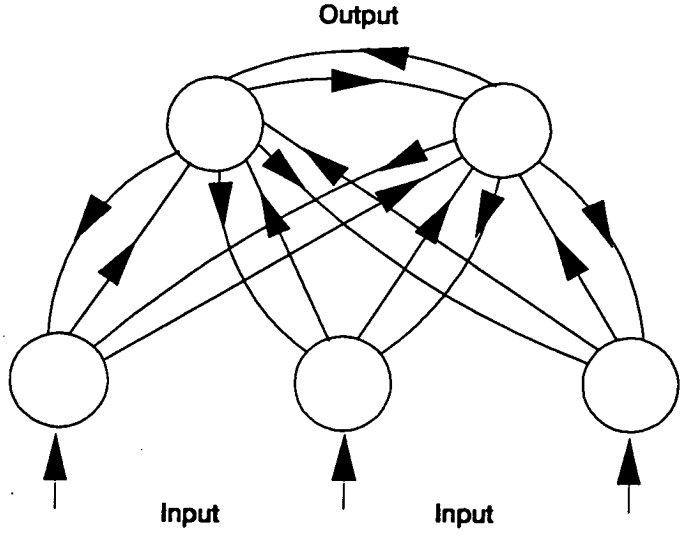


Figure 1.8 Major components of the Grossberg classifier net

2N connections to compute matching scores.

1.3.8. Other Neural Models

The above sections have given a general overview of the most well known neural models and learning procedures. There are however, many other relevant types of modelling that are discussed here briefly. Among these is the work of Anderson [37, 38] who has worked on distributed representation and neurally inspired models for theories of concept learning and amnesia. Willshaw pursued distributed memory models and analysis of the properties of distributed representative schemes [39]. Kohonen introduced self-organising feature map algorithms [40], which are modelled on the organised mappings of the body surface on to the cortex such that the organisation of neurons at higher levels is created during learning by algorithms which promote self-organisation. Here, the essential mechanism of the scheme is to cause the system to modify itself so that nearby units respond similarly. This is achieved by the units responding randomly to a parameter of interest. When an input signal with some value of the parameter is provided, one unit responds "best" to that input. This unit is located, in order that its neighbours, ie. units in some region around it and the unit itself have their synaptic weights changed, so the units now respond like the best unit did.

Hopfield's contribution [29] of the idea that networks can be seen as seeking minima in energy landscapes played a prominent role in the development of the Boltzmann Machine [41]. The machine is composed of visible and hidden non-linear computational units. Which are connected to each other by *bi-directional* weights. A unit is either *on* or *off* and will adopt either state as a probabilistic function of the two states of its neighbouring units and the states between them. With the right assumptions, units can be made to act so as to minimise global energy. If some of the units are externally forced or "clamped" into particular states to represent a particular input, the system will find the minimum energy configuration that is compatible with that input. A Boltzmann distribution is used to find the global minimum.

The Hamming Net [42] is a maximum likelihood classifier using neural type units. The model calculates the Hamming distance between binary inputs corrupted by noise and the learned state and uses this to classify the input with the correct output.

This chapter has given a brief summary of biological neurons and their function. The history of the development of the understanding of the nervous system has been a major influence in synthetic neural network modelling and learning algorithms. Much of the current research evolves around software modelling of the networks and learning algorithms given in this chapter, but an increasing minority of research work is now in developing hardware implementations. The next chapter discusses how the various types of models have been implemented in hardware.

Chapter 2

Neural Network Implementation in VLSI

This chapter gives an account of the implementation of neural networks in hardware. The majority of the hardware is in the form of VLSI ASICs (Application Specific Integrated Circuits) in either analogue or digital forms or a combination of both, often supported by memory and a host computer. Some VLSI circuits employ learning and recall techniques, but generally they act as hardware accelerators in a "neural system". The vast majority of the work has been carried out in the last 3 - 4 years.

2.1. The Motivation For VLSI Networks

The most general neural model is based on computational units (neurons) that are connected together in a totally interconnected array or in a layered network. The connections are made via synaptic weights. The synapses have the effect of weighting the response of any neuron to its inputs from all other neurons in the network so they may be more or less excitatory to the receiving neuron and the total weighted sum changes the level of activity of that neuron †. Each neuron receives activity from other neurons in the network. The total activity, x_j, of any neuron j [36] is given by:-

$$x_j = \sum_{j=1}^{j=N} T_{ij} V_j \tag{2.1}$$

where T_{ij} is the weight between neuron i and neuron j and V_j is the present state of the neuron. Equation (2.1) is a simplified form of Grossberg's equation (1.12) in section 1.3.7. The activity is thresholded according to an activation function, F,:-

$$V_j = F(x_j) \tag{2.2}$$

The neural activity may be thought of as the level of excitation of the neuron and the activation function as the way it reacts (by altering its state V_j) in response to a change in activation. The activation is not bounded in the same way as V_j. The magnitude of V_j can be changed by interactions from other neurons in the network, by a passive decay of the weight over time and by an external stimulus.

† An excitatory input will tend to turn a neuron *on* and an inhibitory one will tend to turn it *off*.

Figure 2.1 shows a selection of activation functions. These are the *threshold* function (sometimes referred to as the "Hopfield" function), where the state is either 0 or 1, the *linear* threshold function and the *non-linear sigmoidal* function, which represents a smooth switch of state from 0 to 1 as the activity, x_i, increases through a threshold value x_t. A parameter, T, (often termed "temperature") controls the sharpness of the transition.

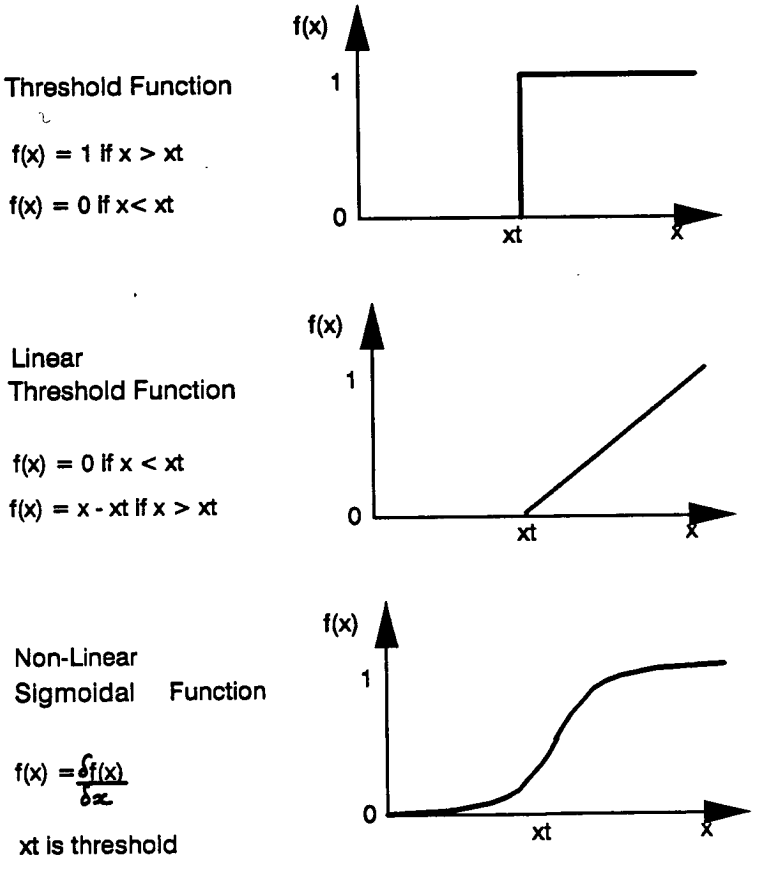


Figure 2.1 Activation functions

The arithmetic operations in equations 2.1 and 2.2 both appear straightforward, but synthetic neural networks consist of a parallel array of units calculating $\Sigma T_{ij} V_j$ synchronously. Therefore, if the number of units is large, the amount of computational power required overwhelms even a supercomputer. Small networks consisting of tens of units take many hours to simulate on computer, thus there is much incentive to

build LSI/VLSI networks which will complete the same computation in milliseconds. A VLSI network also offers the advantage of being cheaper to run after the initial manufacture outlay, than hours of CPU time and it allows several tens or hundeds of neurons to be fabricated on a die.

Although VLSI networks offer many attractions for neural implementation, the majority of current research is aimed at algorithmic development using computer simulation, often based on array processor or other supercomputer architectures to simulate mathematical models and demonstrate their correctness and processing ability. An example of this can be found in the work of Wallace [30, 43]. Here an ICL Distributed Array Processor (DAP) is used with 4096 bit-serial processing elements hard wired in a 64 x 64 square array to develop algorithms to improve the storage performance and content addressability of the Hopfield net for random patterns [29]. Numerical simulations were run to show how the number of perfectly stored vectors, p, depends on the number of nodes N in the network. The storage prescription adopted by Hopfield (equation 1.9) was used with different values of p and N to produce a "signal plus interference" approximation to obtain an approximation for the perfect storage fraction in terms of p and N. The loss of memory capacity as the number of nominal vectors was increased was analysed in terms of phase transitions in statistical mechanics (ie. changes in minimum energy). The results of the simulations led to an extension of the Wallace-Hopfield algorithm based on the Delta Rule in Chapter 1, section 1.3.5.

Another such example is a ten processor, programmable systolic array computer which has been used for back propagation simulations in work done by Pomerleau *et al* [44]. Here, 60 fully interconnected hidden units perform one learning trial in 0.8ms, which is approximately 17 million connections per second. This has proved to be the fastest implementation of back propagation and most cost effective for neural network simulation.

2.2. Hardware Implementation

Hardware synthetic neural networks fall into two broad categories, digital and analogue, with some of a hybrid digital/analogue form. The majority consists of systems with a VLSI circuit specially designed to compute neural functions in some

way and many are based on the Hopfield model. There has also been some implementation using microprocessors and digital signal processing (DSP) integrated circuits [30, 43-45].

An early neural computing machine, the WISARD (Wilkie, Stonham and Aleksander's Recognition Device) [46, 47], is an adaptive pattern recognition machine based on neural principles. The observation that a binary neuron may be viewed as an n-input - single output logic element is related to one column of RAM registers, where each value is set independently and represents the truthtable of a logic device with one output. The WISARD architecture is shown in figure 2.2. The RAM network or "Discriminator" consists of K×N-input RAMs with one output feeding to a summation operator. A binary pattern or training set of K×N bits is input to a Discriminator and a "1" is stored in each RAM. Unknown patterns representing a class are later presented and the Discriminator measures the similarity of an unknown pattern to each of the patterns in the training set. If two patterns are similar, the RAM outputs a "1". The "1s" are then summed to give the response, **r**, of the Discriminator. In a multi-class problem, *M* Discriminators can be used to represent *M* classes as in figure 2.3. An unknown pattern can then be "classified" to a particular Discriminator by the indication of *how close* it is to one of the learned patterns. If an unknown pattern that is completely different from any of the initial "learned" patterns were presented to a Discriminator, no RAM would output a "1" and hence **r**=0 and the patterns would not be classified to the Discriminator. The correct input on a RAM's address line will produce a "1" output. Adjusting the value stored at that location during training will cause the Discriminator's class to emerge, when unknown inputs are presented.

2.2.1. Digital Neural Networks

The neural equations 2.1 and 2.2 can be implemented using digital hardware, resulting in fast and accurate neural network computation. DSP chips used as neural accelerators fall in between the extremely fast computation time of a VLSI circuit and the relative slowness of a computer simulation as has been shown by Penz *et al* [45]. In this work, the TMS 32020 DSP chip is used to accelerate the matrix multiplication in the network. A 256 square component matrix multiplying a 256 component vector performing a single multiply/accumulate instruction showed to be 2.5 times faster than

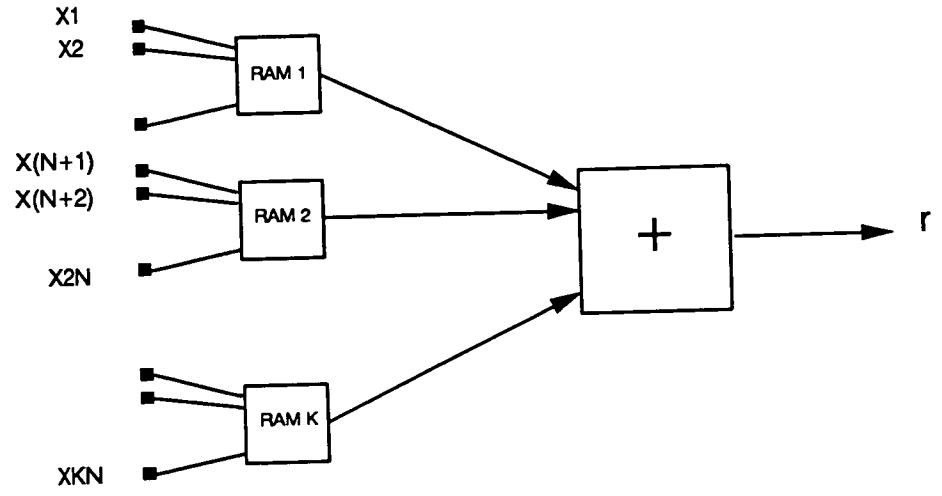


Figure 2.2 WISARD Discriminator

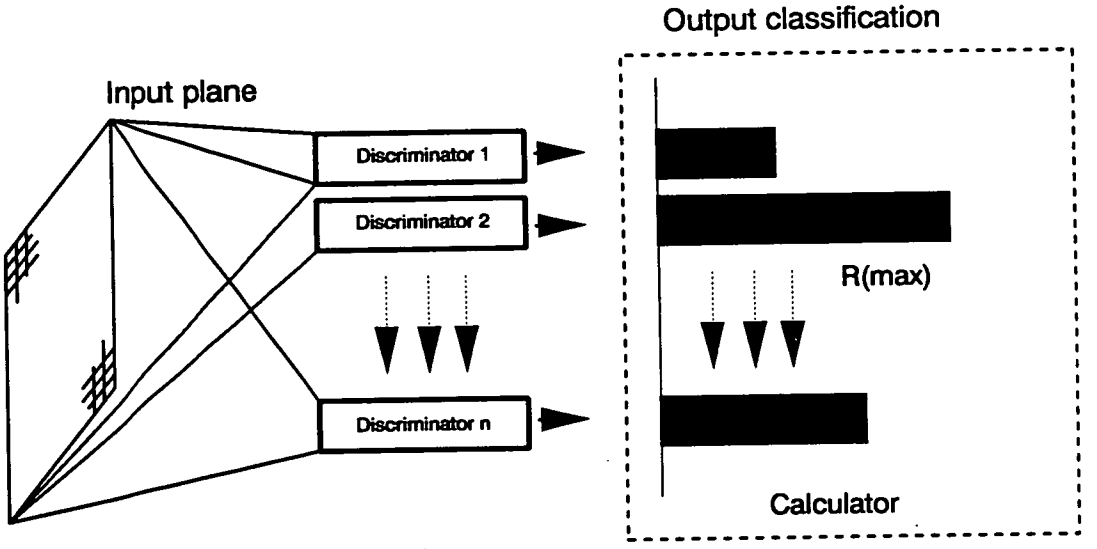


Figure 2.3 WISARD Multi-Discriminator system

the DEC VAX 8600 . The time taken to perform one $\sum_{j=1}^{256} T_{ij} V_j$ is 39ms, but this is still many times slower than the same computation in VLSI which would be about 0.02ms in a device operating at 20MHz. Further acceleration can be achieved using a parallel computer architecture as, for example, the Odyssey Board developed by Texas Instruments, which consists of many DSP modules sharing a common bus to provide the necessary computational power for advanced signal processing. Each module contains a TMS 32020, 16 kbytes of program memory and 128 kbytes of data memory that will store a 256 x 256 array of 16 bit numbers, ie.,,, T_{ij} for a 256 × 256

problem. A board consists of 4 modules and is capable of 20 million arithmetic operations per second. Again in comparison to the DEC VAX, the Odessey board will compute a 1000 x 1000 matrix 40 times faster.

A DSP neural system is well suited to solving small groups of networks, but as the numbers increase the time required for the solution increases accordingly. An alternative solution to this has been developed by Garth [48-50] with the GRIFFIN neural machine. It consists of a distributed array of autonomous neural network simulators called NETSIM as in figure 2.4.

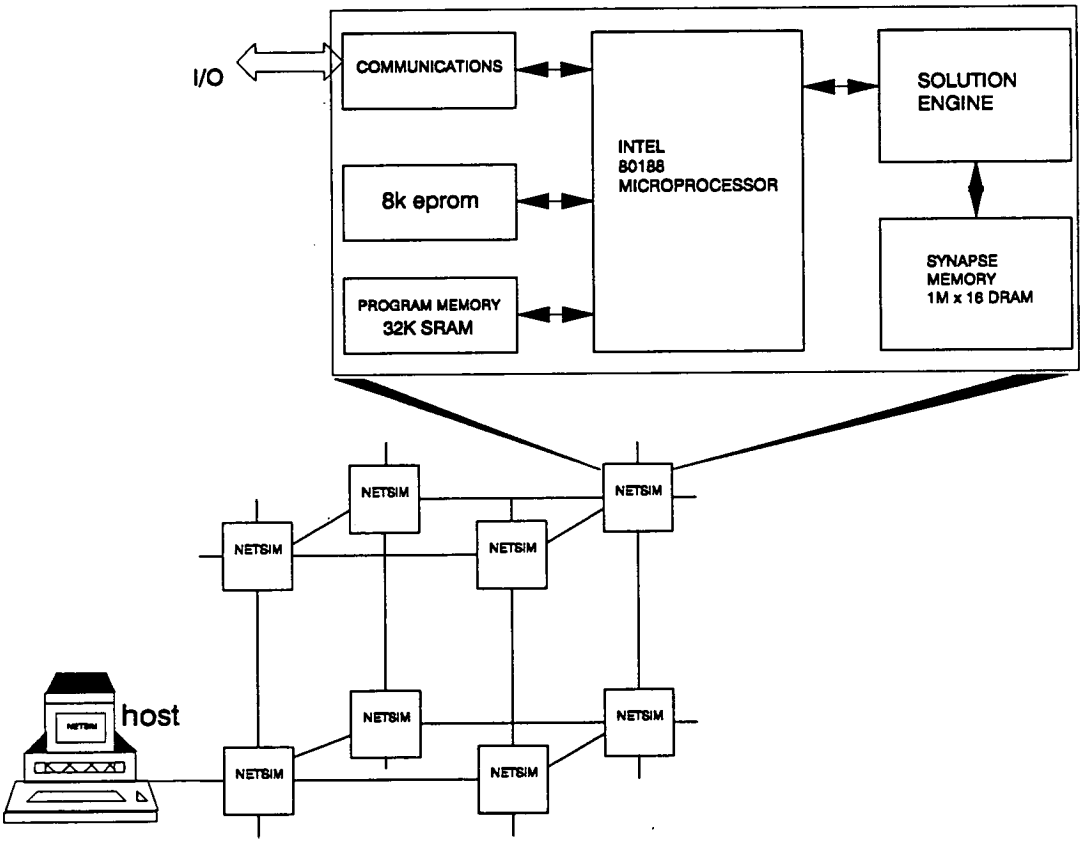


Figure 2.4 NETSIM card within the physical organisation of the GRIFFEN

Each NETSIM card comprises a local microprocessor, a solution integrated circuit (a specialist co-processor to implement the neural network function at high speed, with the weights and states stored in local DRAM) and a communications integrated circuit to allow large numbers of NETSIMs to be connected to form the GRIFFIN. The solution integrated circuit performs the multiply and sum operation required for forward or backward propagation and multiply and update operations required for

synaptic weight update according to the error propagation algorithm in chapter 1, section 1.3.3. The microprocessor computes other elements of the simulation, including the non-linear function and the simulated interconnection. The resultant information is then loaded into the communications integrated circuit for transmitting to the relevant node in the system. For a 256 neuron network a full synaptic update including forward and back propagation is calculated to be 350ms. A fully pipelined 5 x 5 x 5 NETSIM array computes in forward propagation 450 million synapses per second and 90 million in backward propagation. A similar concept using a CMOS special purpose primitive processing element array to build a parallel MIMD neurocomputer [51] is being persued at U. C. L..

An alternative solution for a digital neural network is the implementation of a VLSI array with interconnected, synchronously operating multipliers as in figure 2.5 to compute the neural function in equation 2.1. Each multiplier has a register for weight storage and the activities for neurons are computed in parallel. Here, the multiplication of a synaptic weight by a neural state is achieved by right-shifting the weight. An add/subtract circuit at each multiplier stage allows excitatory and inhibitory inputs to the neuron and computes the accumulating activity of the neuron. The resulting staircase activation function, allows neurons to take intermediate states between off and on. A simple Hopfield net is used and delta rule learning [27] is computed off-chip. This approach forms the main thrust of this thesis and is described in detail in chapters 3 and 4. An idea similar to this using a Hopfield model with multi-state neurons, where the states are {-3, -2, -1, 0, +1, +2, +3} can be found in work by Potu *et al* [52].

A VLSI Hopfield digital network that includes a learning algorithm on-chip is given in [53]. The network has N identical neuron cells, each one with full arithmetic capability for learning and updating and a local memory containing the relevant column of the synaptic matrix. Neural states are stored in a N x 1 bit shift register clocking at 20 MHz and a partial potential update in each neuron is performed at each shift of the register. After 1 cycle, each neuron takes its decision. Other digital VLSI hardware neural accelerator systems are given in [54-56].

Digital techniques offer several useful properties for neural implementations in that weights can be easily stored and programmed, they have greater flexibility, high

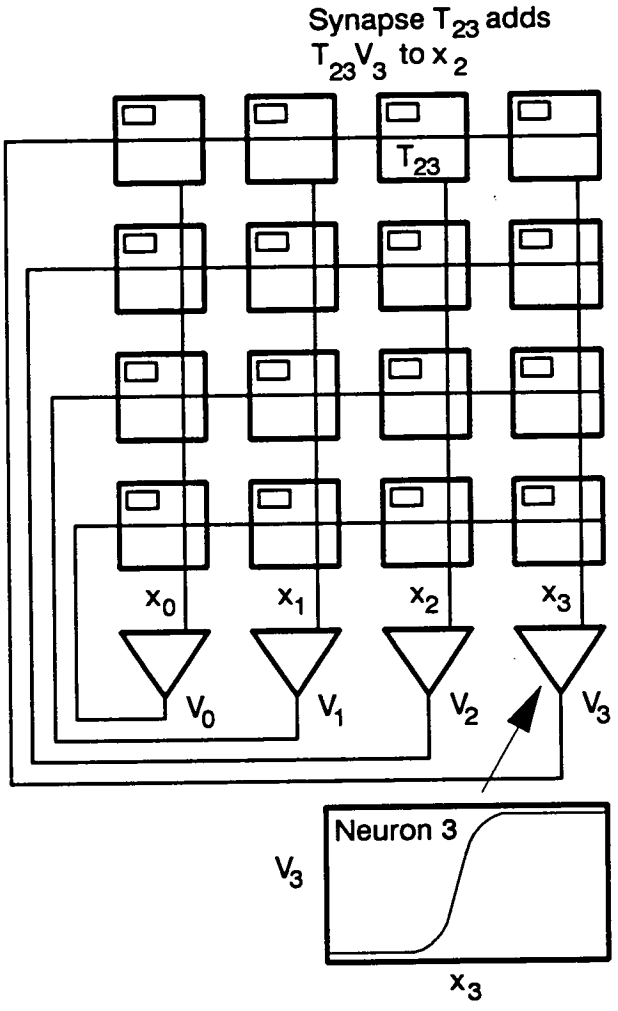


Figure 2.5 Interconnected network of synchronously operating multipliers

precision and clock rates in excess of 20 MHz. However, there is a major drawback in that a large silicon area is required for the multiplication function allowing a maximum of tens of neurons to be fabricated on an integrated circuit giving a small network. Therefore, to obtain a large enough network that will perform useful simulations might require several integrated circuits to be hardwired together to provide a larger system.

2.2.2. Analogue Neural Networks

The implementation of neural networks in analogue VLSI circuitry has taken several forms. These include op-amp resistor networks [57-59], dynamic weight storage [59-62], sub-threshold circuits [4], low-area arithmetic arrays [63-67] and pulse stream networks [68-72]. The major problem within an analogue approach is the storage of

the synaptic weights. Various methods have been developed including resistor arrays [57-59], storing charge on "on-chip" capacitors [59-62, 72], and by using MNOS [73] and α-silicon technologies [74, 75].

The most straightforward form of an analogue neural network can be derived from the digital architecture in figure 2.5, by using operational amplifiers instead of neurons and a resistive input R_{ij} at each synapse location. This approach has been used by Sivilotti *et al* at Caltech [57] and Graf *et al* at Bell Labs [58, 59]. Sivilotti uses resistive elements and achieves negative values for inhibitory connections by using 4 pass transistors operating in their resistive regime. This gives a tri-flop cell allowing the 3 connection strengths of {-1, 0, +1}. The connections are also programmable, but have a large hardware overhead in that a programmable synapse requires 41 transistors instead of 16 required for an unprogrammable one.

The network developed at Bell Labs is given in figure 2.6. It consists of an array of 54 amplifiers with their inputs and outputs fully interconnected through a matrix of resistive coupling elements. The input voltage to each amplifier is determined by summing the contributions from the amplifiers to which it is connected. The outputs are programmed to source or sink current into the input line of every other amplifier. This is controlled by 2 memory cells. Figure 2.7 shows how the resistive elements can be programmed to be excitatory or inhibitory.

The function of associative memory is achieved by simultaneous collective operation of all the amplifiers. Each circuit state is described by a 54 component vector. A desired set of states is made stable by proper choice of the connections in the coupling network. After the circuit is initiated with an input vector, it evolves to the stable state that most closely resembles the input. Data input and output are through a buffer in which one memory cell is connected to each amplifier unit. From this buffer data can be loaded into memory cells or used to initialise the circuit.

Dynamic Weight Storage

The storage of an analogue weight as charge on MOS capacitors or transistor gates allows synapses to have a smaller number of transistors and hence a higher level of integration on a chip. This can be subject to problems of leakage and data corruption and needs refresh circuitry if long hold times are required. Capacitor circuits are used

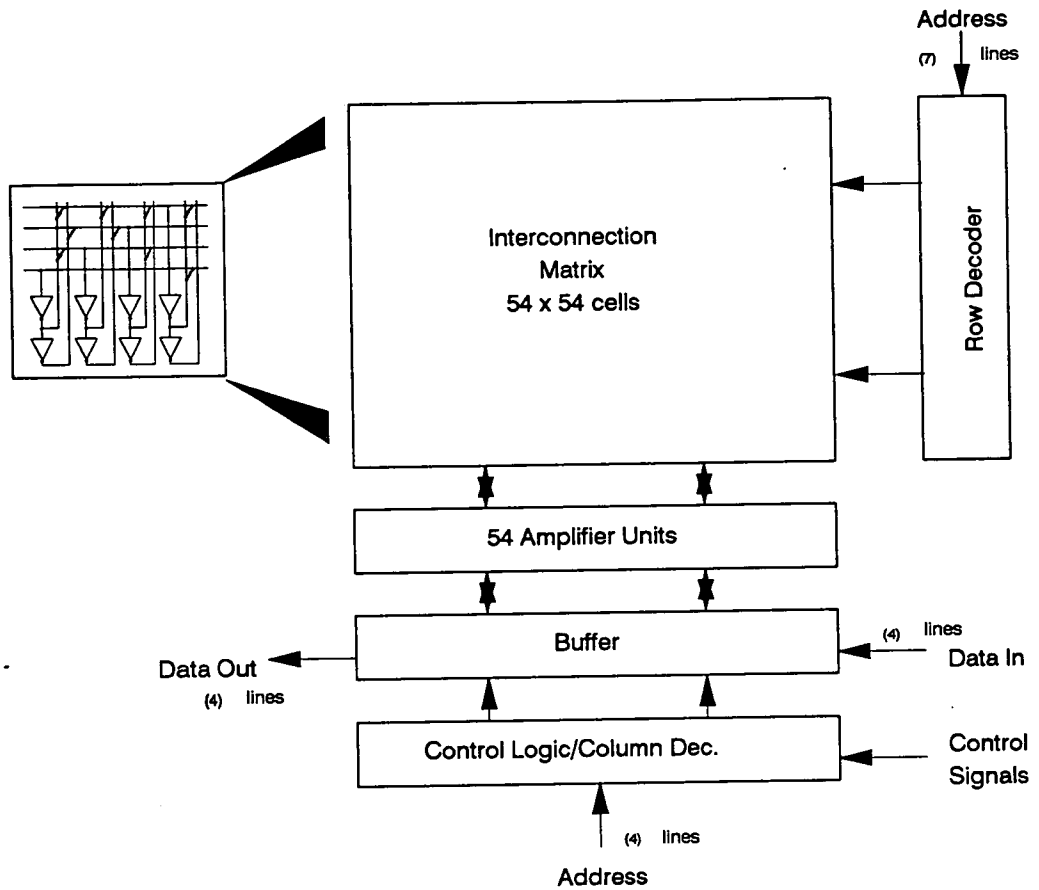


Figure 2.6 Bell Lab chip showing the array of amplifier units and resistors

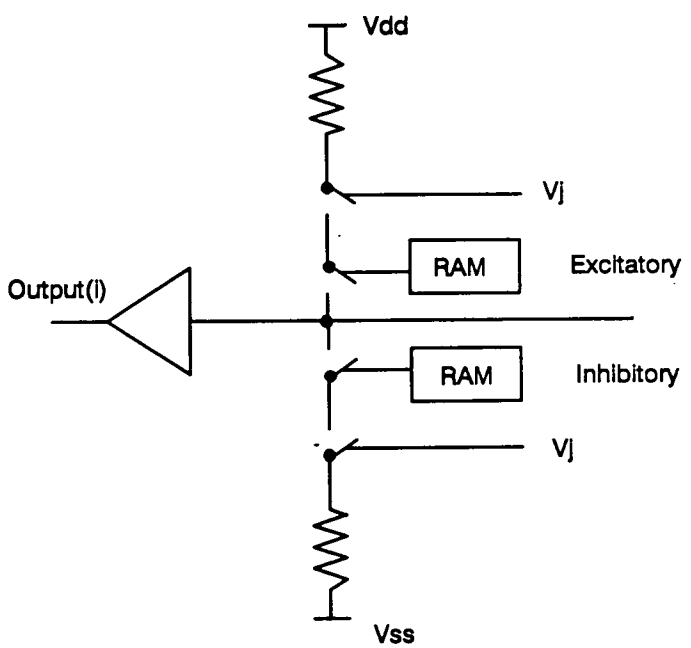


Figure 2.7 Schematic diagram of the programmable resistive connection

by Bell Labs [59-61] and a switched capacitor circuit is described in [62].

The Bell Lab circuit shown in figure 2.8 is of a synaptic connection.

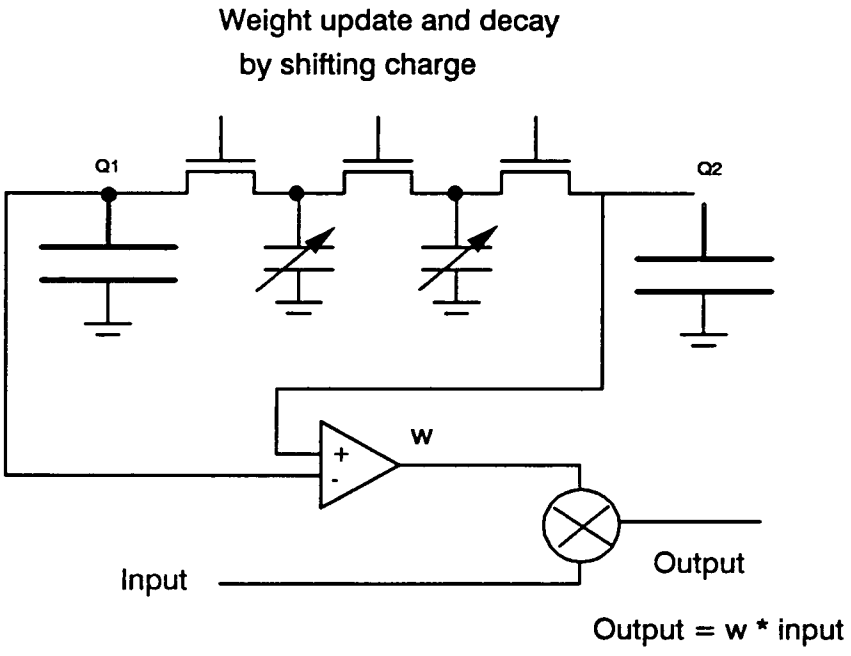


Figure 2.8 Analogue synaptic connection represented by the difference in voltage stored on two capacitors

The connection strength is represented by the difference in voltages stored on 2 MOS capacitors. The capacitors lose about 1% of their charge in 5 minutes at room temperature, but the leakage rate can be reduced by lowering the temperature of the device, e.g. by up to 5 orders of magnitude at −100 degrees C. The output is a current proportional to the product of the input voltage and the connection strength. The output currents are summed on a wire and sent "off-chip" to external amplifiers. Connection strengths can be adjusted for learning by transferring charge between the capacitors through a chain of transistors.

The switched capacitor implementation [61] uses MOS transistors as switches, which are controlled by switch "phase" periodic waveforms. A totally interconnected neuron network is used and charge is transferred from each neuron output to the neurons' inputs by the switches. The total charge input to a neuron is collected and thresholded according to the sigmoid activation function.

A network implementing Kohonen's self-organising feature map algorithm [40] that uses charge stored on the gate of an MOS transistor as a synaptic weight is given in [76]. An analogue input is represented by a voltage on the transistor drain and if the gate voltage exceeds the maximum input voltage by an amount greater than the transistor threshold voltage (so that the device is operating in the ohmic region), then the current through the transistor is proportional to the product of the input and weight voltages. The transistor constitutes the synaptic connections and by connecting synapses to a single wire, current summing is performed to give the neural activity.

Goser [77,78] describes an associative network using a floating-gate transistor technique for weight storage. The device acts as a non-volatile storage cell, where the electrical charge on the floating-gate represents the information and is stored independently from the power supply to the cell. This type of device does not store the analogue value accurately, but the integration of a CCD (charge-coupled device) loop connected to the floating-gate within a synapse cell can overcome the disadvantages of low accuracy and long degradation time, although a large cell area is needed for this. The number of CCD's in the loop yields the accuracy of the connection weight and the information stored in the loops can be read out by opening the loops. In this way, adaptive weights can be written into the loops enhancing the learning procedure.

Technology Dependent Analogue Weights

A CCD/MNOS (metal-nitride-oxide-semiconductor) has been used by Sage *et al* in their analogue neural network [73]. The design uses a totally interconnected array of neurons with charge packets to represent the analogue information transmitted through a synapse and MNOS device structures to store electrically changeable, non-volatile synaptic weight values. A cross section of an MNOS device is in figure 2.9. The structure is similar to an MOS device, except the main gate insulator is silicon nitride with a very thin silicon oxide layer, so at gate voltages of ± 35 volts, electrons and holes move by quantum-mechanical tunnelling between the underlying silicon and long lifetime traps in the nitride layer. A high voltage causes a shift in the charge stored in the traps. If the gate voltage is kept below 10 volts the trapped charge becomes permanent and makes the voltage on the gate appear to shift its switching threshold. The apparent modulation of the gate voltage is used to control the size of the

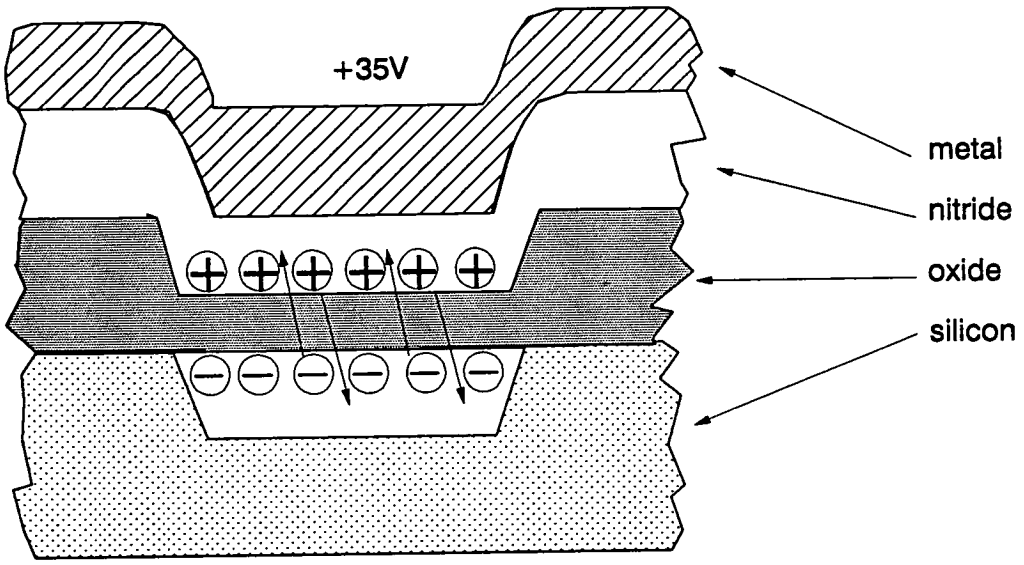


Figure 2.9 Cross section of an MNOS device

synaptically transferred charge packets. The total charge, N_i for a neuron i, is accumulated from the synaptic connections to it, using equation $N_i = \sum_j T_{ij} V_j$, where the state is either 0 or 1. The state $V_j = 1$, allows the charge in the synapse to flow in the gate and be added to the total, N_i. If $V_j = 0$, the charge is blocked. The total charge is compared by a sense circuit to a threshold value to determine the neuron state for the next cycle in the circuit.

Amorphous silicon (α-Si) has been used at Bell Labs [74, 75] in order to achieve 256 neurons on a chip using a resistive network for weight storage and amplifiers with inverting and non-inverting outputs for the neurons to make inhibitory and excitatory connections. Synaptic resistor values are chosen to correspond with the desired memories and the values are derived from an adaptive learning rule [79]. Current summing is used to add together all the contributions to the input of an amplifier. As there is in excess of 100,000 resistors on a chip, their size must be very small with a resistance of a few mega-ohms to keep the power consumption low. High value α-Si is used for this, however this approach does not allow the resistors to be changed once fabrication is finished, hence giving a fixed set of stable states. Electron-beam direct writing is used to pattern the resistors.

Research is taking place at CalTech [80, 81] into how an electrically switchable, resistive component with memory can be incorporated at each synaptic intersection in a

matrix such as that described by Graf *et al* in the Dynamic Weight Storage section above [74,75]. The matrix could become a PROM, with a further possibility of EEPROM, if the memory switch could be made reversible. Hydrogenated α-Si thin film technology is a possible solution. Others include α-Ge/alloys and platinum/aluminium oxide films.

Thin film technology may be a solution for achieving hundreds of neurons integrated on a single chip with a suitable programmable material. Otherwise, large networks are restricted to being non-programmable with limited use for their implementation. Alternatively, CCD techniques have been shown to be programmable, however, their implementation on silicon results in a large area per device and hence a small number of neurons per chip.

Imprecise, Low-area Arithmetic

One method to increase the level of neuron integration on a VLSI chip is to make use of a neural networks natural fault tolerance towards imperfection in synaptic/neuron detail. This is due to the nature of large parallel arrays and learning procedures by using simple transistor circuits to approximate to the neural arithmetic i.e., the multiply and add function [63-67].

The approach used by Akers *et al* at Arizona [63,64] uses a limited interconnect analogue neural cell given in figure 2.10. Weights are stored dynamically on the gates of transistors T1, T2 and T3 and the "multiplication" $T_{ij}V_j$ is performed as the drain terminals of T7 - T9 are charged to voltages equal to the approximate T_{ij} voltage minus the device threshold of T1 - T3. When the clock $\phi 1$ is at a logic "1", the charge accumulations representing these voltages are summed via the analogue adder. V_{out} is then thresholded according to the inverter T16/T17. N-type current sources are used to achieve small synapses.

The circuit shown in figure 2.11 is proposed by Verleysen *et al* [67]. Two values stored in each synapse allow it to take the values {-1, 0 or +1}. Positive currents are sourced on one line and negative currents on the other. The input to the neuron is the sum of all the synaptic currents. The neuron compares the two currents i+ and i- and will switch on if the total positive current is greater than the total negative current, otherwise it will switch off. The use of only N-type transistors avoids the mismatch

between P- and N-type current sources, due to their different mobilities. When the mismatch is multiplied by the number of active synapses, it soon reaches the value of one synaptic current and would therefore limit the number of neurons that could be cascaded together.

Subthreshold Circuits

Neural network modelling using CMOS circuits operating at sub-threshold (weak inversion) has been the work of Mead at CalTech [4]. Digital designs using MOSFETs in saturation (strong inversion) require that $V_{gs} > V_T$, but in sub-threshold operation where $V_{gs} < V_T$, $I_{ds} \propto e^{K.V_{gs}}$,† where K varies inversely with the amount of doping in the CMOS process [4, 82]. The advantage of this type of operation is that the power dissipated in circuits is very low, usually in the region of 10^{-12} to 10^{-6} W. Also, drain currents saturate in a few $\frac{kT}{q}$ †† allowing transistors to operate as current sources over most of the voltage range from ground to $|V_{dd}|$. This property is shared with bipolar transistors, thus allowing bipolar circuits to be adapted for MOS usage. The problems of noise immunity in such circuits, caused partly by the mismatch of transistors due to threshold differences may be lessened by the natural fault tolerance due to the massive parallelism of neural networks.

Mead shows how many biological nerve functions can be translated to equivalent electrical circuits and that the nerve membrane conductance is exponentially dependent on the potential across the membrane, analogous to the I_{ds} - V_{gs} relation above. His work also includes the implementation of some processing functions such as the retina (chapter 15 [4, 83],), the cochlea (chapter 16 [4, 84],) and the problem of motion detection (chapter 14 [4, 85],).

2.2.3. Pulse stream Networks

The inspiration for the pulse stream technique [68-71] is its analogy to the electrical/chemical pulse mechanism of biological neurons and the discovery that some arithmetic operations such as multiplication can be implemented efficiently using pulse

† V_{gs} = gate-source voltage, I_{ds} = drain-source voltage, V_T = threshold voltage.

†† k = boltzman constant, q = electronic charge on an electron.

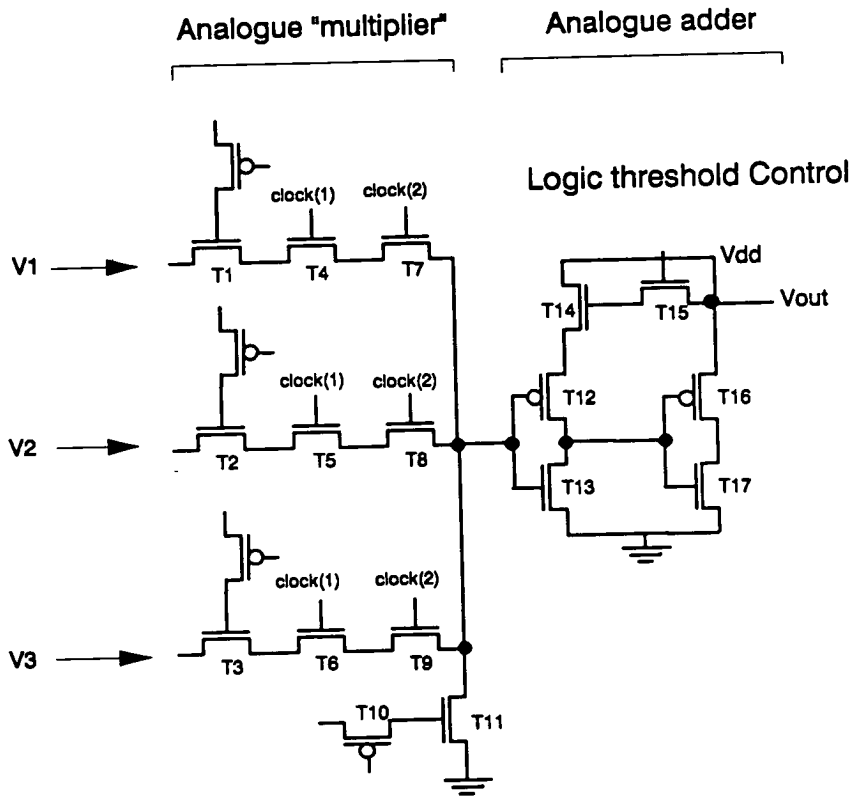


Figure 2.10 Aker's analogue synthetic neural cell

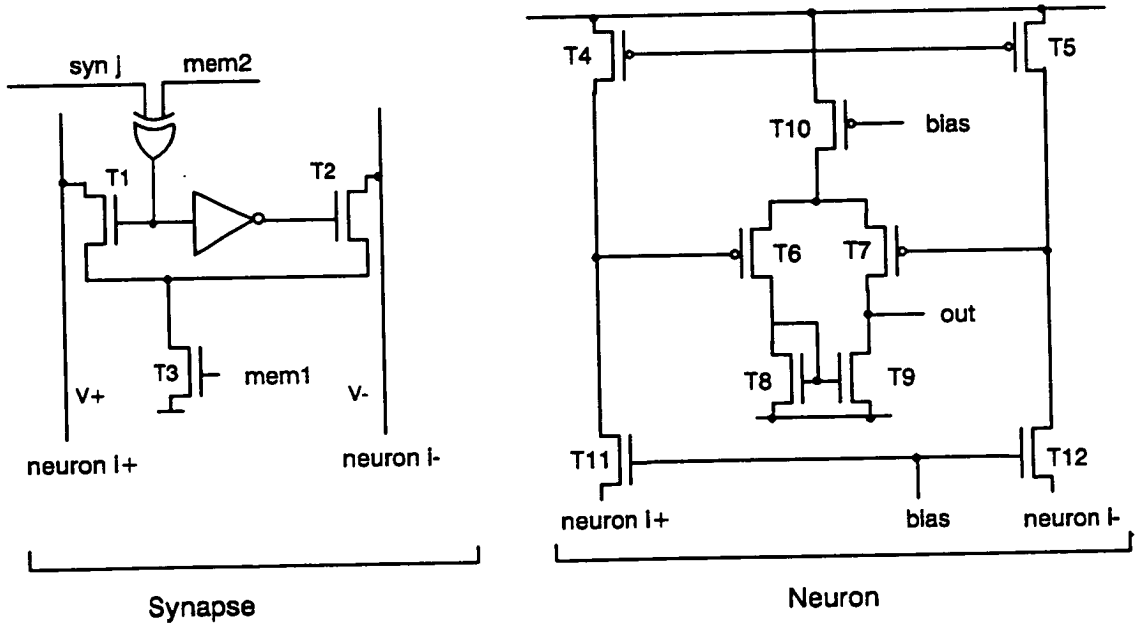


Figure 2.11 Verleysen's synapse and neuron circuits

streams. The name "pulse stream" is derived from the signalling mechanism used in that when a neuron is ON it fires a regular train of voltage spikes (at rate R_j^{max} pulses per second) on its output and when it is OFF it ceases to fire. The neuron circuitry is given in figure 2.12. Excitatory and inhibitory pulses are signalled on separate lines and used to dump or remove charge packets from an activity capacitor. The resultant varying analogue voltage, X_i, is used to control a voltage controlled oscillator (VCO), which outputs short pulses. The voltage based pulse stream synapse in shown in figure 2.13. Synaptic gating is achieved by using synchrounous "chopping clocks" to define time intervals during which pulses may be passed or blocked. The clocks have mark-space ratios of 1:1, 1:2, 1:4, etc. and are used in conjunction with synaptic weights stored in digital RAM, to gate the appropriate portion of pulses to either the excitatory or inhibitory column.

This network proved the viability of the pulse stream technique which has now undergone some refinements involving the removal of the digitally stored weights, the pseudo-clocking scheme and separate signals for the excitatory and inhibitory signals. Accordingly, a fully programmable, totally analogue synapse using dynamic weight storage has been developed [70, 72, 86, 87], which operates on individual pulses to perform arithmetic. The activity capacitor has been distributed amongst the synapses, reducing the neuron to a voltage controlled oscillator

The synapse circuit in figure 2.14 has the synaptic weight, T_{ik}, stored as a voltage on a capacitor. At room temperature, refresh of dynamically stored values is necessary. The viable storage time of the charge is determined by capacitor size, temperature of the chip surface and leakage characteristics of the CMOS process used. Presynaptic input pulses $\{ V_K \}$ at a constant width D_t and frequency determined by the state of neuron k discharge the output of inverter T1/T2 linearly from V_{supply} to 0V as shown, and at the end of a pulse, the capacitor recharges to its original voltage. The second inverter has an output pulse proportional to the synaptic voltage T_{ik}. Multiplication is only linear over the range $1V \leq T_{ik} \leq 3V$, and by choosing suitable values for the aspect ratios of T6/T7, it is possible to achieve excitation ($2V \leq T_{ik} \leq 3V$) and inhibition ($1V \leq T_{ik} \leq 2V$) .

This arrangement allows synapses to be cascadable as in figure 2.6, with the activity capacitor on the drain connections of T6 and T7 aggregating the total activity x_i, for

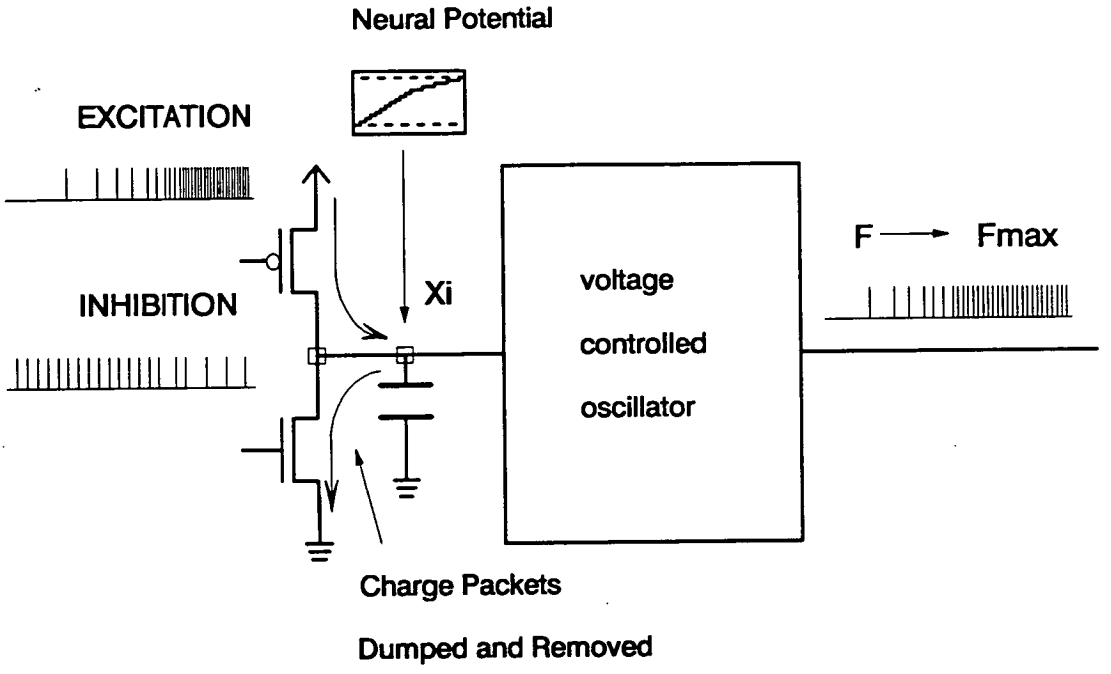


Figure 2.12 Pulse stream neuron

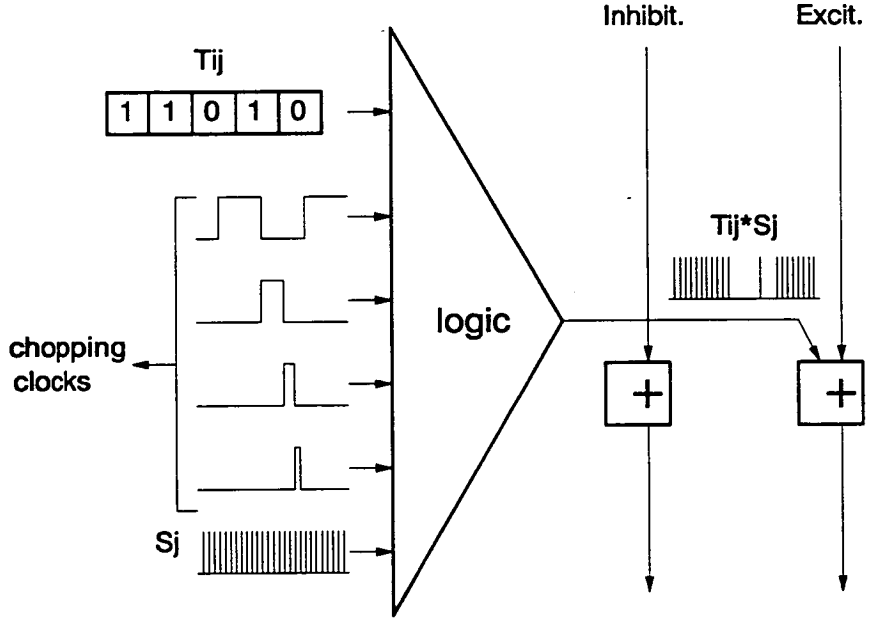


Figure 2.13 Pulse stream synapse using chopping clocks and digital weight storage

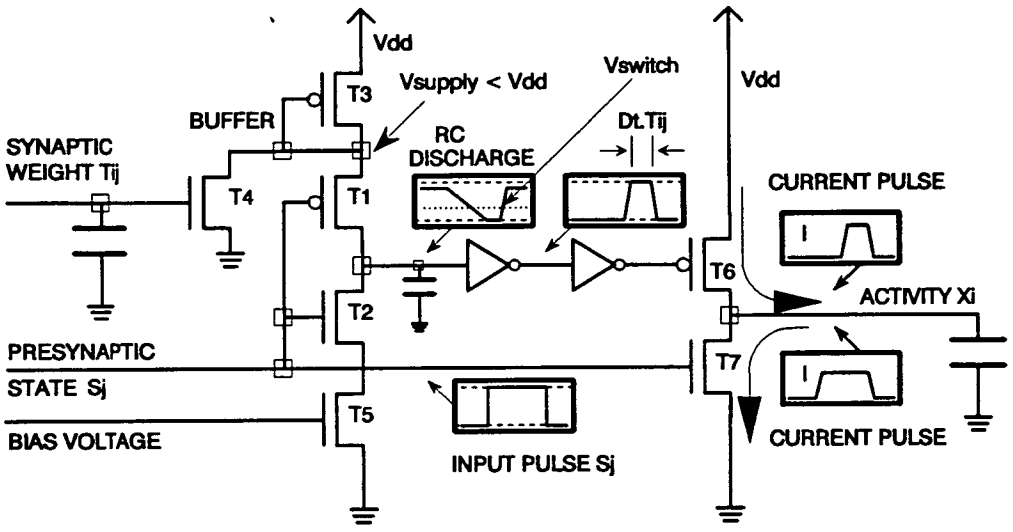


Figure 2.14 An excitatory and inhibitory analogue synapse

any neuron *i* from all the other neurons connected to it. This circuit has been implemented in 2μm CMOS array of 100 synapses and functions correctly.

Further work is being done on analogue synapse circuits using only 3 N-type devices in addition to the storage capacitor. Details of this will be published at a later date.

2.2.4. Optical Neural Networks

The vast majority of neural network implementations use VLSI technology, but optical neural computing with its parallelism and speed offers an alternative to VLSI, however there has been little in the way of neural computing optical devices. The first analogue optoelectronic hardware implementation of neural networks, introduced in 1985, received attention for several reasons. The main one is that the optoelectronic approach combines the massive interconnectivity and parallelism of optics and the flexibility, high gain, and decision making capability offered by electronics. The construction of large scale optoelectronic neurocomputers can solve optimisation problems at potentially very high speeds by learning to perform mappings and associations.

An example of one of the earliest optoelectronic neurocomputers consists of a totally interconnected network and is shown in figure 2.15 [88]. To avoid interference

effects, an incoherent light source is used, which also relaxes the stringent alignment required in coherent light systems. An optical crossbar interconnect carries out the vector-matrix multiplication $\{T_{ij}V_j\}$ required. The state vector is represented by a linear light emitting array (LEA), the connectivity matrix $\{T_{ij}\}$ is implemented in a photographic transparency mask and the activation potential x_i, is measured with a photodiode array (PDA). Light from the LEA is smeared vertically onto the $\{T_{ij}\}$ mask. Light passing through the rows of weights is focussed on the PDA. The neuron threshold θ_i and external input stimuli are injected optically with the aid of a pair of LEAs, whose light is focussed on the PDA. A third PDA is used for the injection of noise. This architecture has been successfully employed as a 32 neuron network with associative memory.

To implement learning, the network needs to be partitioned into input, output and hidden layers of neurons. An efficient way to do this is in figure 2.16 where the layers are partially interconnected and the weight matrix is divided into an input group, V_1, an output group, V_2 and hidden units, H. V_1 and V_2 are only connected via H. The connection weights are programmably computer controlled by a spatial light modulator (SLM). The architecture uses supervised learning and the weights are updated according to a prescribed formula until all the training vectors evoke the correct desired output. This network has also been used to demonstrate supervised stochastic learning by simulated annealing. For this, the computer controller controls the annealing profile, monitors the convergent state vectors and computes and executes the weight modifications.

The use of neurocomputers in practical applications involving fast learning or the solution of optimisation problems requires large networks that still have programmability and flexibility as in the network described above. One method being developed at the University of Pennsylvania [88] uses a "clusterable photonic neural chip" concept. Here the architecture in figures 2.15 and 2.16 is modified to include internal optical feedback and "non-linear" reflection (optical detection, amplification and thresholding) on both sides of the connectivity matrix. Another approach has been to use a 2-D arrangement of neurons to increase packing density [89].

A Hopfield neural network using optical techniques has been developed at British Aerospace [90]. Computer generated holograms are used to form fixed weighted

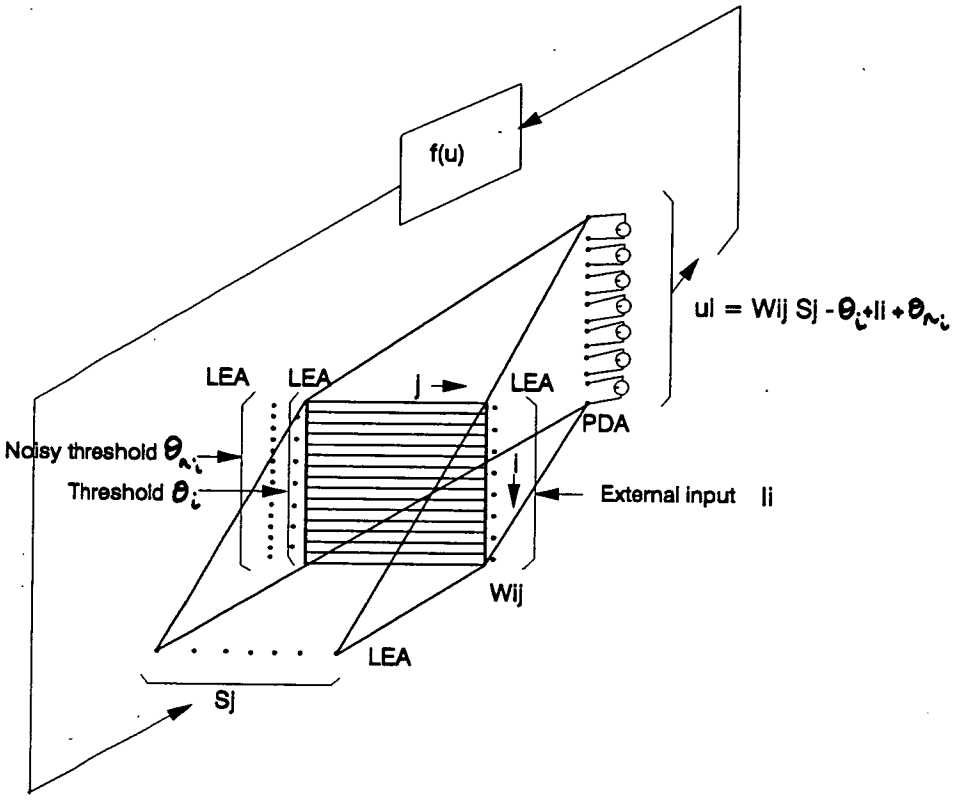


Figure 2.15 Optoelectronic analogue circuit of a fully interconnected neural network

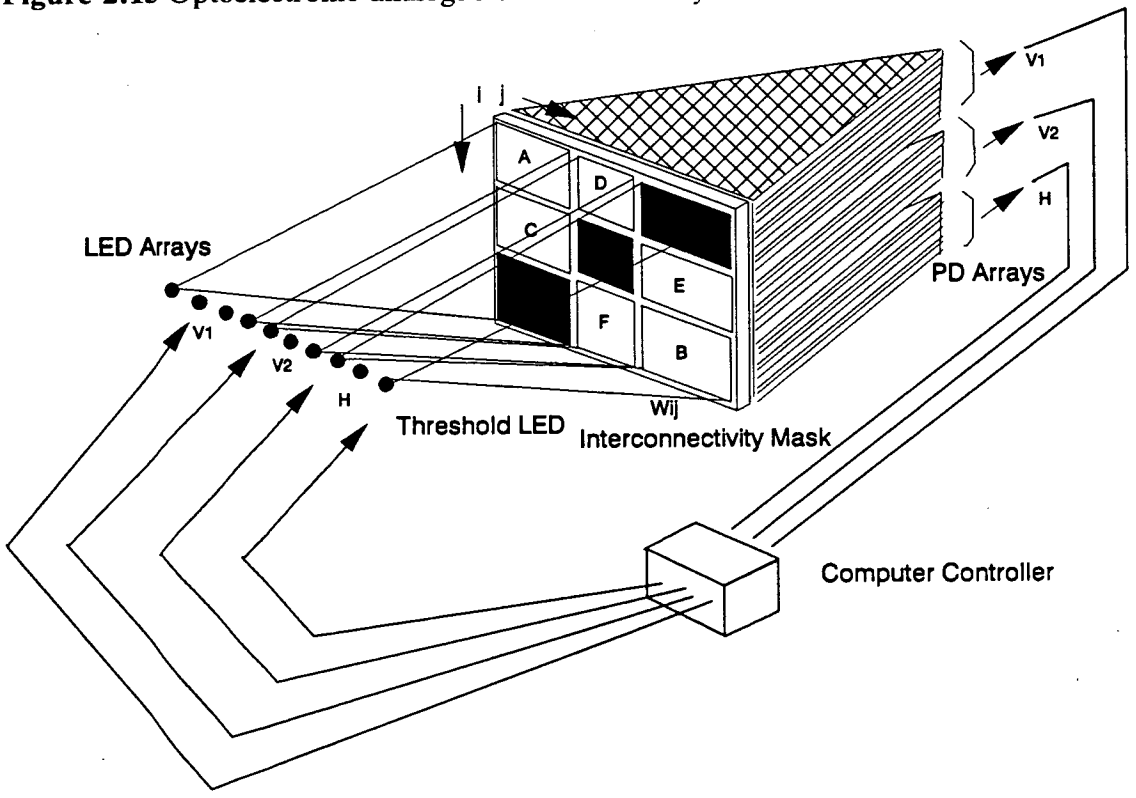


Figure 2.16 Partitioned fully interconnected network to implement learning

interconnections and a spatial light modulator enters the input image. The holographic interconnections perform the vector-matrix multiplication and the resultant product is thresholded and fed back into the matrix multiplier. There is a drawback however in this type of network, in that material limitations severely limit the size of such a machine and the weight connections are fixed. Despite this, a factor in favour of this optical system is the ease in producing a complex hologram compared to an extensively wired electronic system. The largest machine that could seriously be constructed using this method is a 25 x 25 neuron array. Nevertheless, this still represents a powerful processing capability which can be applied to less extensive networks such as edge detection algorithms.

Optoelectronics offer advantages for the design and construction of a new generation of analogue neurocomputers capable of performimg computational tasks at high speed. The architectures of the present optical prototypes aim to demonstrate the best attributes of optics and electronics and can be combined with programmable non-volatile spatial light modulators and displays to form neural networks that include associative storage and recall, self-organisation and adaptive learning.

Chapter 3

A Digital, Reduced Arithmetic Neural Network

This chapter describes a digital bit-serial neural network, that uses a "reduced arithmetic" multiplication function to implement the $\{T_{ij}V_j\}$ product. Software simulation results using this computation style are given, showing comparisons of the 2-state, reduced arithmetic and sigmoid activation functions.

3.1. Digital verses Analogue Network

There are fundamentally two approaches to implementing any function in silicon - digital and analogue. Each approach has its advantages and disadvantages. These are listed below along with the merits and demerits of bit-serial architectures in digital (synchronous) systems.

The primary advantage of digital design for a synapse array is that digital memory is well understood and can be incorporated easily for programmable synaptic weights. Learning networks are therefore possible without recourse to unusual techniques or technologies. Other strengths of a digital approach are that the design techniques are more advanced, automated and easily amenable in VLSI implementation than their analogue counterparts and noise immunity and computational speed can be high. Unattractive features are that digital circuits of this complexity need to be synchronous and all states and activities are quantised, while real neural networks are asynchronous and unquantised. Furthermore, digital multipliers occupy a large silicon area and an n neuron network requires n parallel multipliers, resulting in a low synapse count on a single chip.

The advantages of analogue circuitry are that synchronous behaviour and smooth neural activation are inherent. Circuits elements can be small with faster settling than digital ones, but noise immunity is relatively low and arbitrarily high precision is not possible. However, a drawback of analogue networks until the last year, has been that no reliable analogue non-volatile memory technology was readily available, but as discussed in chapter 2, section 2.2.2, there are now several implementations using analogue weight storage. For this reason, the first learning networks lent themselves

more naturally to digital design and implementation.

3.2. Bit-Serial verses Bit-Parallel Network

Bit-serial arithmetic and communication can be efficient for computational processes. It allows good communication within and between VLSI chips, with signals leaving and entering the chips on single pins, an essential requirement for achieving the largest possible number of synapses on a single device. Structures can be pipelined for maximum efficiency, eg., each synapse in an interconnected array computes its partial activity and passes it immediately to the next synapse in the column so that the accumulating activity of the neurons is being calculated every clock cycle. A bit-serial strategy is ideal for neural networks as it minimises the interconnect requirement by eliminating multi-wire busses. Although a bit-parallel design would have a lower computational latency (delay between input and output), a bit-serial synaptic array lends itself to pipelining and thus can make optimal use of the high bit rates possible in serial systems and allows efficient use of silicon area.

3.3. Reduced Arithmetic

In a digital network each synaptic weight, T_{ij}, is represented by a binary word. The division of a binary number by 2, simply requires the *right-shift* of the number by 1 bit. For example, when 10110_2 (= 22_{10}) is right-shifted, the word becomes 01011_2 (= 11_{10}). Since the synaptic function is $\{T_{ij} \times V_j\}$, the right-shift of the weight by 1 bit is equivalent to the multiplication of the weight by the state, V_j =0.5. Similarly the right-shifting of the weight by 2 bits, would be equivalent to multiplying the weight by V_j = 0.25. Therefore a full multiplication and add function can be reduced to a "right-shift" and add. The state $V_j = 1$ only requires the synapse to add the weight to the total activity, x_i, of the receiving neuron and the state $V_j = 0$ requires no weight to be added to the total. By allowing $V_j < 0$ and replacing the adder with a switchable added/subtractor, gives the further states of -1, -0.5, -0.25, etc., which need only a few extra transistors. The reduced arithmetic approach gives a staircase activation function shown in figure 3.1. Five, seven, nine, etc. neural states are therefore feasible with circuitry that is slightly more complex than a serial adder. For example, for a 5-state activation function, the synapse function between neurons i and j now becomes:-

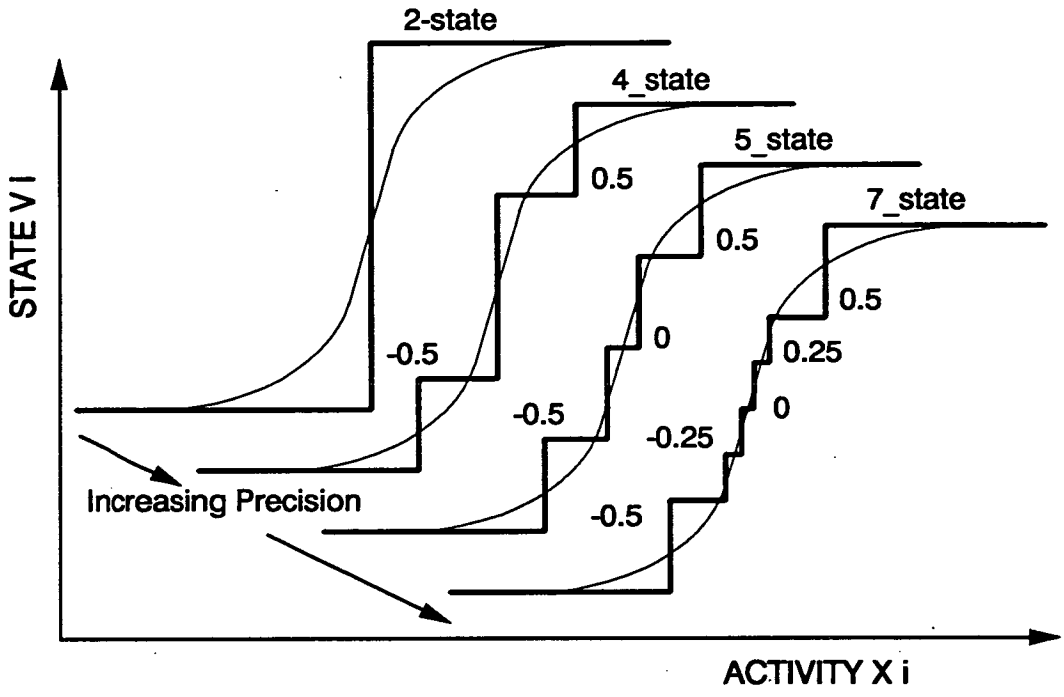


Figure 3.1 Reduced arithmetic allows four, five, seven, etc., state activation functions

$V_j = 1$, add T_{ij} to x_i

$V_j = 0.5$, right-shift and add T_{ij} to x_i

$V_j = 0$, add 0 to x_i

$V_j = -0.5$, right-shift and subtract T_{ij} from x_i

$V_j = -1$, subtract T_{ij} from x_i

where $x_i = \sum_{j=0}^{n} T_{ij} V_j$ for neuron i

The activation function showing quantisation is given in figure 3.2. The sharpness of the transition of the staircase is represented by the gradient of the sigmoid activation function.

The use of a 5-state activation function instead of a sigmoid activation function allows the size of a VLSI synapse to be greatly reduced. However software simulation comparing the learning and recall capabilities of a network using the 5-state activation function with those of the sigmoid is required to verify that the 5-state activation function performs adequately to justify its use in a VLSI implementation.

3.4. Verification of the Reduced Arithmetic

Firstly, a relationship between the staircase function and a smooth sigmoid had to be defined before a network of neurons using reduced arithmetic could be simulated. A neuron with a sigmoid activation function has a neural state that can take on a continuous value between 1 and 0. The state is described by the equation:-

$$V_j = \frac{1}{1 + \exp\left(\frac{x_t - x}{T}\right)} \tag{3.1}$$

where x_t is the neuron threshold and x is the neuron activity as given in figure 2.1, chapter 2. T is the parameter "Temperature" that determines the slope of the function. The form of equation 3.1 is derived from the *Fermi - Dirac* statistics of electrons in conductors, where the *Fermi - Dirac distribution function, F(ϵ,t)* gives the probability that an available energy state, ϵ, will be occupied by an electron at absolute temperature, t. The *Fermi - level* is the energy state that has a probability of ½ of being occupied by an electron which is analogous to the threshold, x_t, for a neuron. At a "Temperature" $T = 0$, the function becomes the rectangular, 2-state threshold function as used by Hopfield and there is no probability that an electron will occupy an energy state above the Fermi level. As the temperature increases the gradient of the sigmoid becomes lower and the probability that states above the Fermi level will be filled increases.

Threshold limits were calculated by experiment for the thresholds x_1, x_2, x_3 and x_4 in the 5-state approximation given in figure 3.2, for any value of T. The limits are derived from equation 3.1 by obtaining the threshold, x_t, in terms of V_j, x and T. These are:-

$V_j = 1$ when $x_4 < x$, $\quad x_4 = x_t + (T \cdot \log(8.00))$

$V_j = 0.5$ when $x_3 < x \leq x_4$, $\quad x_3 = x_t + (T \cdot \log(1.75))$

$V_j = 0$ when $x_2 < x \leq x_3$, $\quad x_2 = x_t - (T \cdot \log(1.75))$

$V_j = -0.5$ when $x_1 < x \leq x_2$, $\quad x_1 = x_t - (T \cdot \log(8.00))$

$V_j = -1$ when $x \leq x_1$.

As the "Temperature" increases, the threshold values become further apart on the x-axis as the gradient of the sigmoid decreases.

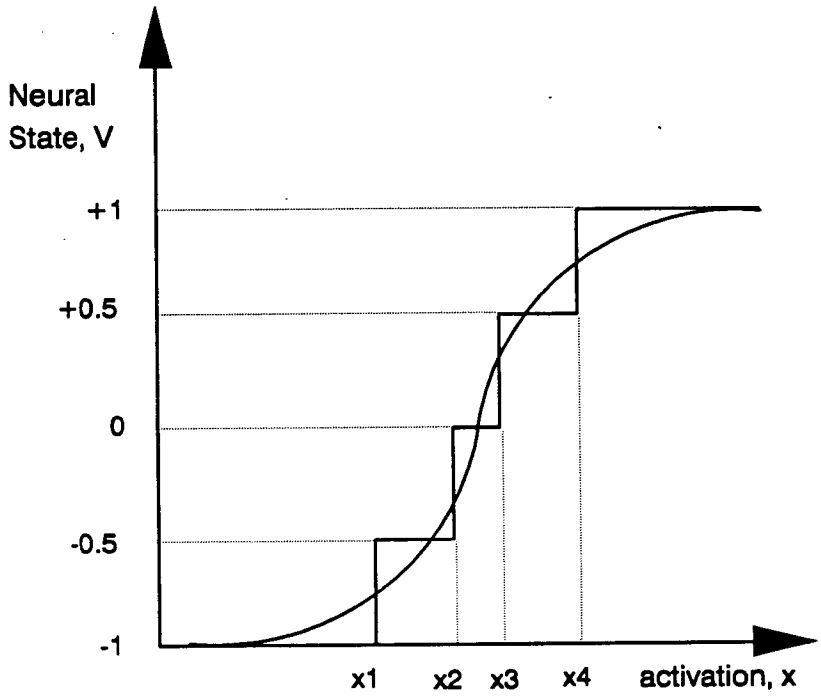


Figure 3.2 The 5-state activation function

3.4.1. Simulated Performance of the Reduced Arithmetic

The simulated performance of the reduced arithmetic was work carried out by A. V. W. Smith and is reported in [71]. A totally interconnected neural network with 64 neurons was chosen for the simulation. The network was restricted to this size due to the lengthy computation time of larger networks. The Hopfield - Wallace Learning Algorithm ([30] and Chapter 1, section 1.3.5) was used to store 32 random patterns using:

1. **2-state** activation function, with the neural states -1 and $+1$.
2. **5 - state** activation function, with the neural states -1, -0.5, 0, $+0.5$, $+1$.
3. **Sigmoid** activation function, with the neural states in the continuous range from -1 to $+1$.

In each case the network was iterated until each pattern being learned matched the initial set-up pattern. The weights, T_{ij}, were floating point, integer numbers and were limited to the ranges $-20 \leq T_{ij} \leq +20$, $-30 \leq T_{ij} \leq +30$, $-40 \leq T_{ij} \leq +40$, $-50 \leq T_{ij} \leq +50$, $-60 \leq T_{ij} \leq +60$, $-70 \leq T_{ij} \leq +70$ in the simulations so the optimum

range of weight might be found. As it was not possible to encode all patterns correctly using the above restricted weights set, a maximum of 150 iterations was allowed.

Recall was then attempted with 12.5% noise introduced to the initial random patterns. This was achieved by selecting 8 nodes at random and changing them from +1 to −1 or from −1 to +1. Each set of patterns was learned using each activation function and then recalled by all three activation functions separately, giving nine possible combinations of the activation functions for the learning and recall. The simulation of each combination was repeated several times and the average number of correct recalled patterns for each was noted.

3.4.2. The Simulation Procedure

The simulation followed the succeeding steps.

Pattern Learning

1. **Random pattern array**: 32 random patterns were produced. Each was stored in a 64 (8 x 8) node array. The nodes were either +1 or −1.
2. **Network weight and state initialisation**: Weights were initialised to a small (almost zero) value and the states were initialised to the first random pattern.
3. **Network iteration**: The network was iterated according to the equation:-

$$x_i = \sum_{j=1}^{N} T_{ij} V_j \tag{3.2}$$

where T_{ij} is the weight between nodes i and j, V_j is the state of node j and x_i is the new activity of node i.

4. **New neural state calculation**: The node activities were thresholded according to the activation function with which the network was learning, to give the new neural states and hence the new output pattern.
5. **Error array calculation**: The new output pattern was compared to the initial random pattern to produce an error array such that:-

$$e_i^{(r)} = \begin{cases} 1 & \text{if } V_i^{(r)} \neq V_i^{(r)} \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

where r is the pattern number, V_i is the new neural state, V_{i-1} is the previous

neural state of the same neuron i and $e_i^{(r)}$ is the error of node i in pattern r.

The above 5 steps were repeated for each random pattern.

6. **Weight update**: When an error mask for each pattern had been calculated, the weights were updated according to:

$$\delta T_{ij} = \sum_{j=1}^{N} V_i^{(r)} V_j^{(r)} \left(e_i^{(r)} + e_j^{(r)}\right) \tag{3.4}$$

where V_i and V_j are the present states of nodes i and j in pattern r.

7. **Iterate to update weights**: The procedure in steps 1 - 6 was repeated. The network was reset to each random pattern, the new neural states were calculated and the weights updated accordingly. This was iterated 150 times maximum or fewer if a pattern was stored correctly in under 150 iterations.

Steps 1 - 7 were repeated for 3 different "temperatures".

Pattern Recall

The weight set was used to recall the random patterns corrupted with 12.5% noise.

1. **Initialise network**: The network was initialised to the first noise corrupted pattern and equation 3.2 was iterated until the new neural states calculated were stable. The states were compared to the initial uncorrupted pattern. If there was no difference, the patterns had been recalled correctly.
2. Step 1 was repeated for each pattern in the array.
3. Steps 1 and 2 were repeated for each activation function.
4. Steps 1 - 3 were repeated for 3 different "temperatures".

Learning with Fixed Weights

Updating the weights in the learning procedure can lead to weight saturation when a fixed weight set is used. This happens when the weight grows above the maximum limit allowed. Three methods to reduce the saturated weights to within the weight limit were simulated, while still achieving the maximum learning capability of the network.

1. **Renormalisation**: The complete weight set was renormalised so the largest weight was reduced to fit within the weight limit. Eg. if the maximum weight limit is ±

30 and a weight is updated to +36, then each weight is renormalised to $\frac{30}{36}$ of it original value. The majority of weights however, are small integer values. Therefore a weight of +2 would become renormalised to +1. This introduces a large error into the learning procedure. A typical example of a weight value distribution is given in figure 3.3.

2. **Forgetting**: The inclusion of a decay or "forgetting" term can be introduced in the learning cycle [36]. At each weight update a "forgetting" term subtracts a proportion of each weight to keep them within the weight limit. This can cause the information that is learned at each iteration to be destroyed.

3. **Clipping**: Any weight that becomes saturated is set to the maximum allowed weight value. Weights within the limits remain untouched. In the learning procedure unclipped weights readjust for the clipped ones.

There have been other experiments, which have indicated that a Hopfield network can "forget" in a different way, under learning control, giving preference to recently acquired memories [91].

3.4.3. Simulation Results

The simulations showed how the properties of the different activation functions effect the learning and recall capabilities of the network. Different weight limits and temperatures also determine the number of iterations required to learn and recall patterns.

Learning with Different Activation Functions

The number of iterations required to learn the 32 random patterns using the 3 different activation functions with varying "Temperatures" and weights limits is presented in figure 3.4. Integer weights with a dynamic range greater than ± 30 were necessary to preserve storage capability. The following conclusions can be drawn from the graphs:-

1. For all values of "Temperature" and weight limit, the 5-state activation function required more iterations than the 2-state function to learn the random patterns.

2. A rise in "Temperature" from 10 to 20 showed a subsequent increase in the number of iterations needed to learn using the 5-state function.

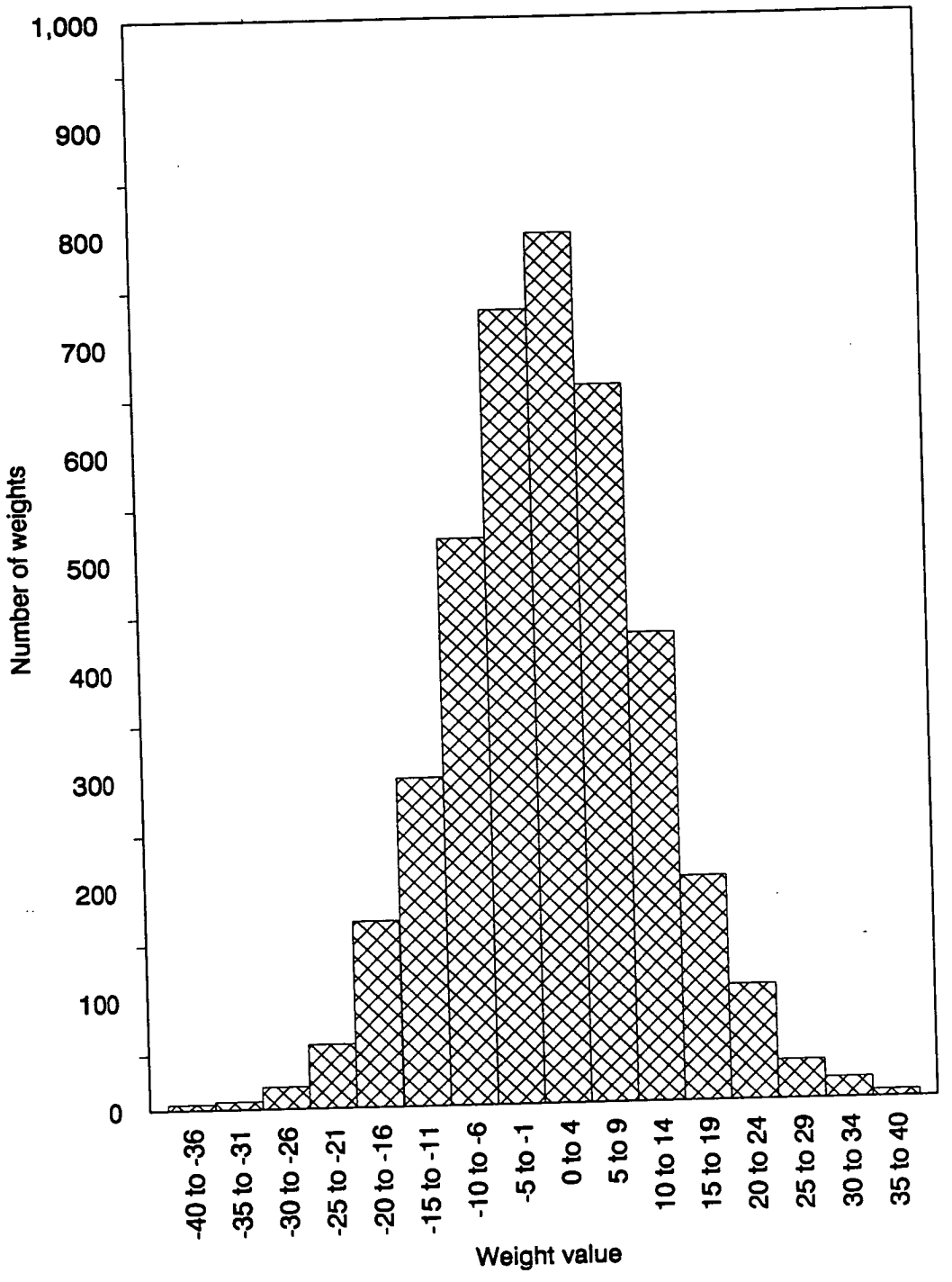


Figure 3.3 Example of the weight value distribution in a network

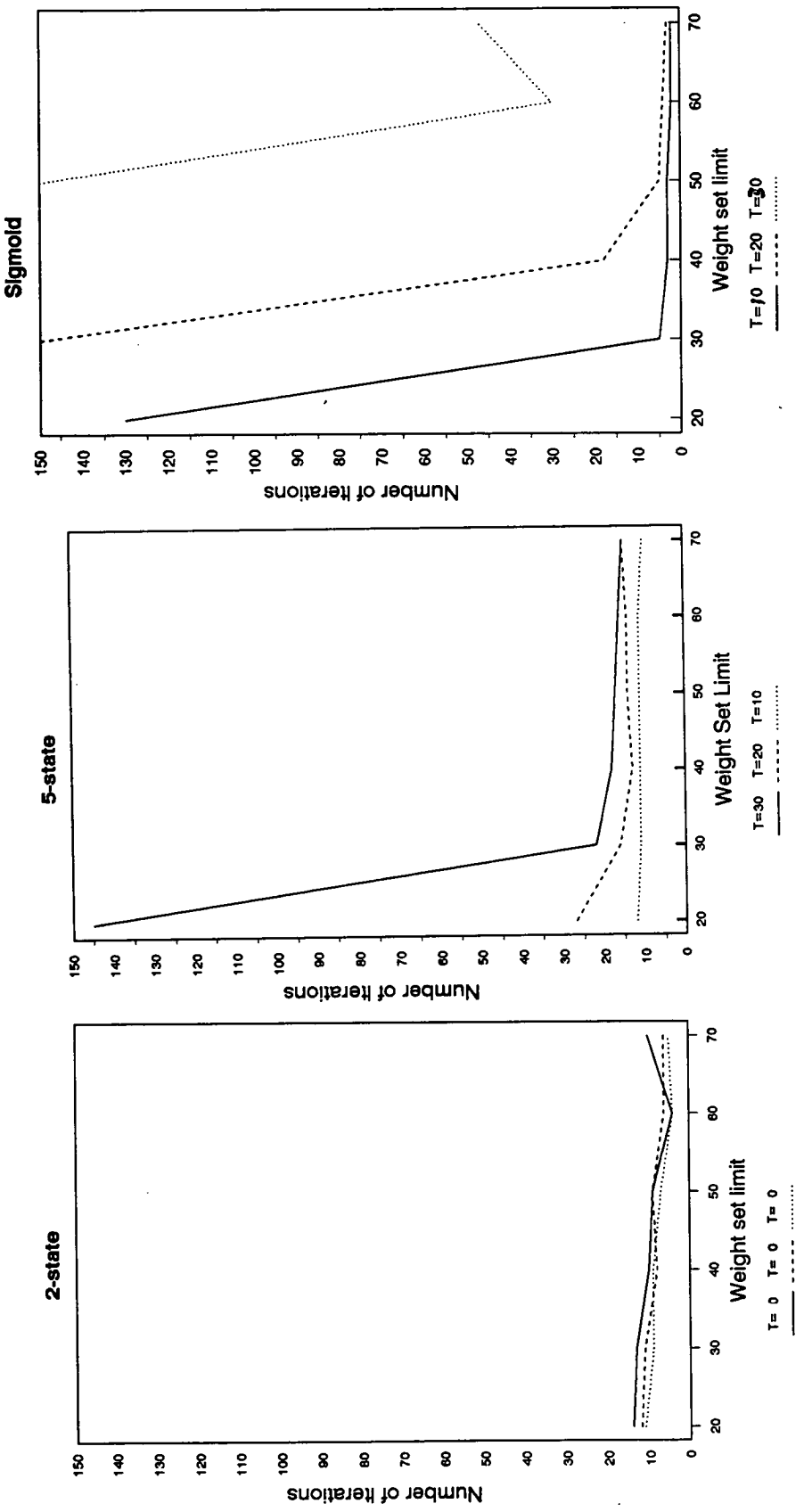


Figure 3.4 The graphs show the number iterations required to learn 32 random patterns using the 3 activation functions with varying "Temperatures" and weight size limits.

3. The sigmoid activation function exhibited the most efficient learning when the weight limit was $\geq \pm 50$ with a "temperature" of 10 and 20.

Effects of Weight Limits and Temperature on Learning

1. **Renormalisation**: Renormalisation of the weights was unsuccessful, suggesting that information distributed throughout the numerically small weights was being destroyed. This led to no solution being found.

2. **Forgetting**: In the time available for experiments, a rate of decay could not be "tuned" sufficiently well to confirm that including a "forgetting" term in the learning cycle can produce the desired weight limiting property. As a result, errors could not be reduced in the learning cycle and no patterns were stored perfectly.

3. **Clipping:** Clipping proved to be a successful method as the learning algorithm adjusted the weights over which it still had control to compensate for the saturation effect in the upper weights. As the sigmoid function has more intermediate states than the 5-state function, it takes longer to readjust the non-saturated weights. The results show that for high temperatures and small weight limits (< 50), clipping occurs and the 5-state function learns faster than the sigmoid. Clipping also occurred for weights with the values $T_{ij}^{max} = 50 \rightarrow 70$, but network performance was not seriously degraded over that with an unrestricted weight set.

Recall with Different Activation Functions

1. **Patterns learned with the Sigmoid Function**: Figure 3.6 shows the results of patterns recalled using the 5-state and sigmoid activation functions, having first been learned with the sigmoid function. The results are for patterns learned and recalled at the same "Temperature". For $T_{ij}^{max} > 50$, the recall ability of the functions is approximately the same and up to 70% of the patterns were recalled. However, at T_{ij}^{max} and "temperature" = 20 and 30, the 5-state function recalled a small number of patterns and the sigmoid function function recalled none. These graphs suggest that the higher the weight limit and temperature, the greater the number of recalled patterns.

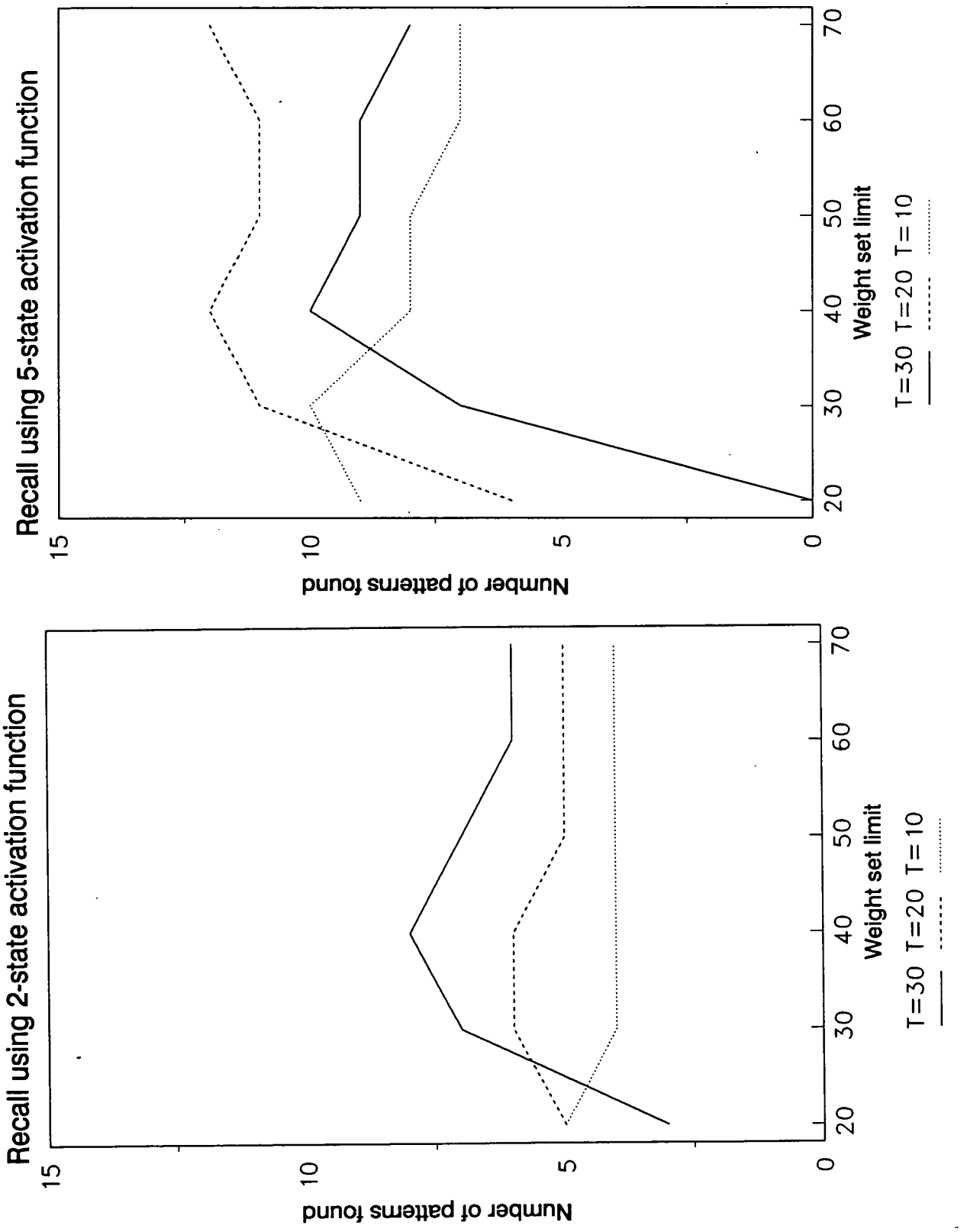


Figure 3.5 The graphs show the number of patterns recalled with the 5-state and 2-state activation functions. Learning was with the 5-state activation function.

2. **Patterns learned with the 5-state Function**: Figure 3.5 shows the number of patterns recalled with the 5-state and 2-state activation functions, which were first learned the 5-state activation function. At each "Temperature", the 5-state function recalled more patterns than the 2-state function. The 5-state activation function with a middle "Temperature" (T = 20) gives the best recall (38% of patterns) with the weight limit, $T_{ij}^{max} \geq 40$.

3. **Patterns learned with the 2-state Function**: A low number of patterns were recalled using the 2-state function to obtain a weight set. Hence the results do not merit discussion.

3.5. Conclusions

The 5-state activation function required the most iterations to learn the 32 patterns and the sigmoid required the least provided the "Temperature" was low. Both the 5-state and the sigmoid function had faster learning as the "Temperature" was decreased. Over the whole temperature range, the 2-state activation function exhibited the best learning ability. The reason for this is that as the 2-state network learns, any neuron has 50% chance of being in the wrong state, therefore on the next iteration the neuron will be in its correct state. Neurons learning with the 5-state activation function have 80% chance of being in the wrong state at the start of learning and the network will take longer to iterate through the 5 states until each neuron arrives at the correct state.

The 5-state function had better recall ability than the 2-state function for patterns learned using the 5-state function. For patterns learned with the sigmoid function, the recall abilities of the 5-state and the sigmoid were very similar for $T_{ij}^{max} > 30$. However, many more patterns were recalled with the 5-state function (70%), that were first learned with the sigmoid, than were learned and then recalled with the 5-state function (38%).

The simulations proved that learning and recall using the 5-state function were significantly better than that using the 2-state. Full sigmoidal activation was better than the 5-state, but the enhancement was not so great as that incurred by moving from the 2-state to the 5-state. This suggests that the law of diminishing returns applies to the addition of the levels to the neural states. This issue has been studied mathematically [92], with results that agree qualitatively with those given above.

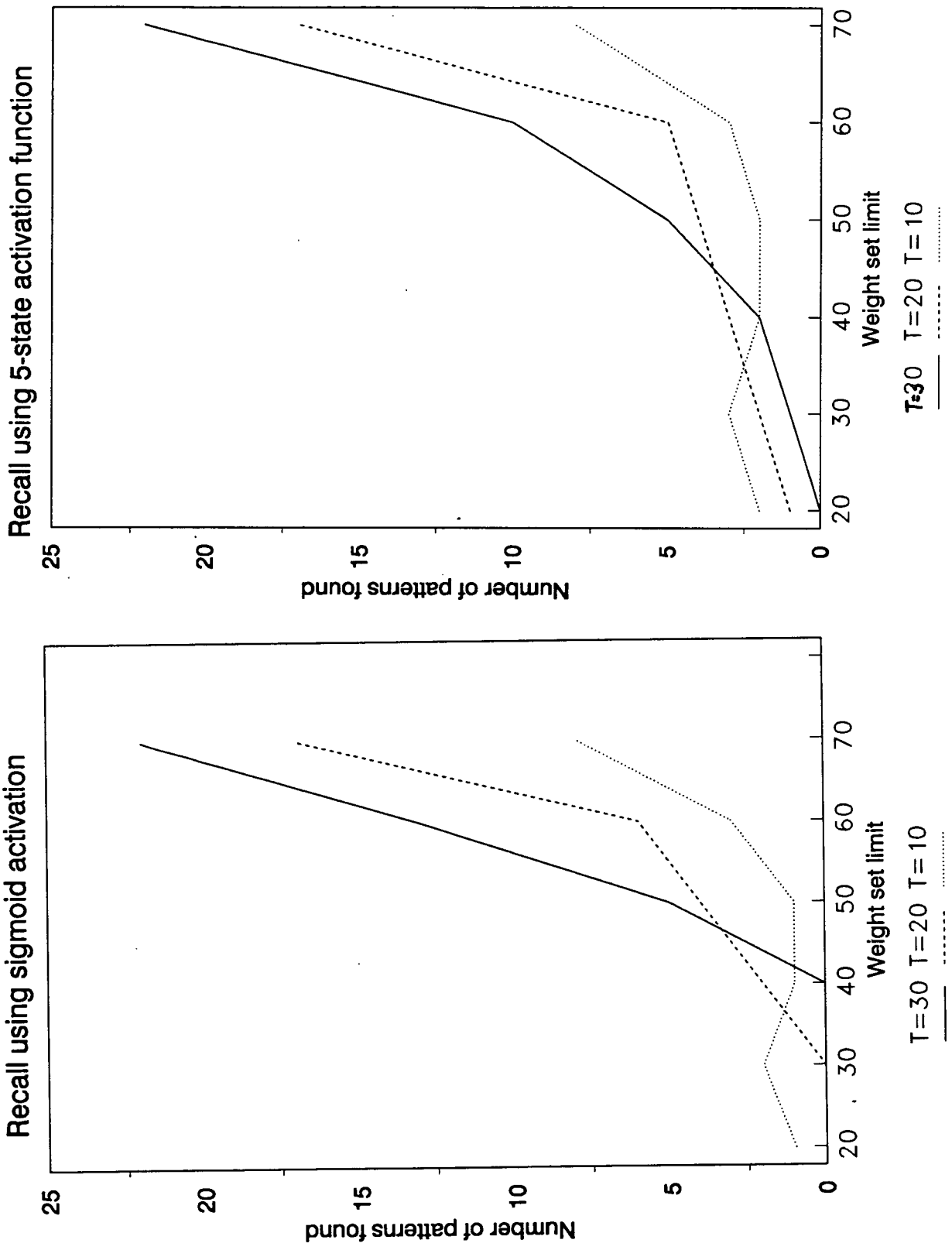


Figure 3.6 The graphs show the number of patterns recalled with the 5-state and sigmoid activation functions. Learning was with the sigmoid activation function.

The results of the 5-state activation function simulations proved that the reduced arithmetic approach was well worth pursuing for the synaptic function instead of full multiplication, without degradation of the synapse performance. The succeeding chapter gives details of how a reduced arithmetic synapse has been implemented in a VLSI circuit.

Chapter 4

VLSI Design of the Synapse Array

The previous chapter has shown how a "reduced arithmetic" approach can be used as a compromise for the full multiplication required for the synaptic function $\{T_{ij} \times V_j\}$. This chapter gives the design details and simulations of two bit-serial, digital integrated synaptic array circuits using reduced arithmetic. The first integrated circuit was designed and fabricated using a fully custom 3μm design, however, owing to a layout error the integrated circuit did not function correctly. A second fabrication of the design using the same process had 0% yield owing to problems with the metal layers. Therefore a second integrated circuit was designed with the ES2 (European Silicon Structures) Solo 1400 silicon compiler as this process guaranteed working devices and a fabrication time of 8 weeks.

4.1. Synapse Requirements

The synapse array was designed to operate at maximum speed and efficiency with minimum latency (bit delay between input and output) and full pipelining. The size of the array was restricted by the area of silicon and the number of pins on the packages available.

Each synapse required a programmable weight storage capability for T_{ij}, a state multiplexor to allow any of the 5-states representing the value V_j to be selected and a full adder/subtractor. The adding/subtracting of the activity at each synapse had also to accommodate word growth for the total activity calculated in each column.

4.1.1. Weight Storage

The storage of a digital weight is straightforward as an n-bit shift register will hold an n-bit weight. The simulation results of the learning and recall capabilities of the 5-state activation function given in the previous chapter showed that the best performance is achieved when the weights have a large dynamic range. The range used in simulation was integer values up to ± 70 and therefore an 8-bit weight (ie. 2^7) was used for the hardware as this gives integer values over the range -127 to $+128$. A 6-bit weight (ie.

2^6) would, for example, only give the range −63 to +64 and hence is not large enough. The dynamic range −127 to +128 was suitable for this type of network and the learning procedure used in the software simulation. As the same network is to be used in hardware, an 8-bit weight was chosen.

A single phase clocking technique was implemented in preference to a two-phase clocking technique. Single phase has advantages in that it allows a higher speed as it does not suffer from overlapping clocks and race hazards that occur in two phase systems.

An 8-bit shift register per synapse required one input pin to load the synaptic weight. In order to integrate the maximum possible number of synapses on a die it was imperative that the weight input pin count be kept to a minimum. This was achieved by connecting each 8-bit shift register to form one long shift register through the array as shown in figure 4.1.

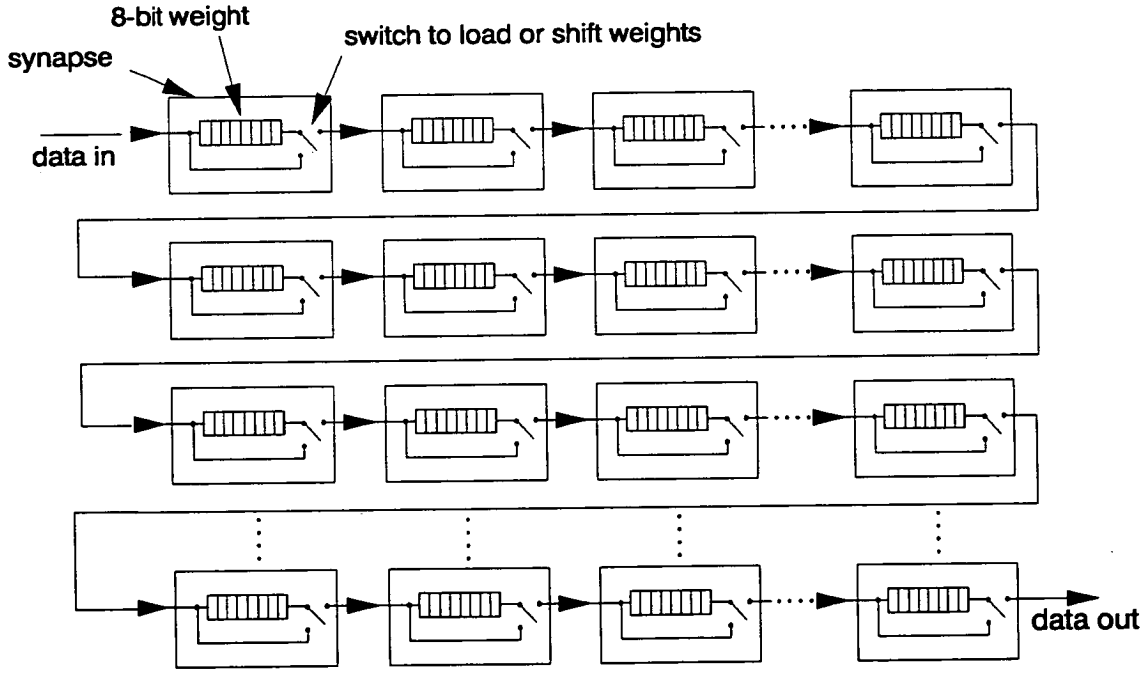


Figure 4.1 The connection between synaptic weights in the array for loading purposes.

Synaptic weights are loaded sequentially through one pin until each weight has reached the appropriate synapse. For example, a 10 synapse array would require 80 clock cycles

for the LSB (least significant bit) of the synaptic weight furthest from the input pin to reach its correct location. A "load/shift" multiplexor controls the synaptic weight operation. The multiplexor allows each weight to be loaded and subsequently isolated and shifted through the register so that it may be "multiplied" by the neural state. During the $\{T_{ij} \times V_j\}$ computation, the weight is shifted around the the register. A further complete shift cycle of the weight permits an 8-bit word growth allowing a 16-bit activity word to be summed down the synapse column. The MSB (most significant bit) of each weight is sign-extended for 8-bits during the second shift phase by a "sign-extend" control signal. These control signals were easily incorporated within the logic required to select the neural state.

4.1.2. Synapse Logic

The **neural state multiplexor** in the synapse controls whether the weight is right-shifted ($\times \pm 0.5$), killed ($\times 0$), or its full value ($\times \pm 1$). A positive state adds the 16-bit $\{T_{ij} \times V_j\}$ to the running total in the synapse column and a negative state subtracts it. Each neural state is signalled on a 3-bit bus that runs horizontally across the array, as shown in figure 4.2. The signals are *rs* (right shift), *kill* and *pm* (plus/minus). The sign extension of the neural state is controlled by two further signals, sel and se2. sel sign extends the weight 8 bits when $V_j = \pm 1$ and se2 sign extends the weight 9 bits when $V_j = \pm 0.5$ as the weight has already been right-shifted by 1 bit.

The **adder/subtractor** has two parts: The first is *summing logic* that adds the accumulating activity from the previous synaptic computation in the neuron column to the present $\{T_{ij} \times V_j\}$ along with any carry from the summation of the previous bits of the synapses' weights. The second part is *carry logic* that signals any carry that occurs in the summation part.

Two's complement arithmetic is used. In this representation, positive numbers are in normal signed binary. The difference lies in the representation of negative numbers. The *one's complement* or *inverse* of the negative number is first computed and then *one* is added, as shown in the following example, which gives the *two's complement* representation of -5.

$$+5_{10} = 00000101$$

The *one's complement* of $+5_{10} = 11111010$

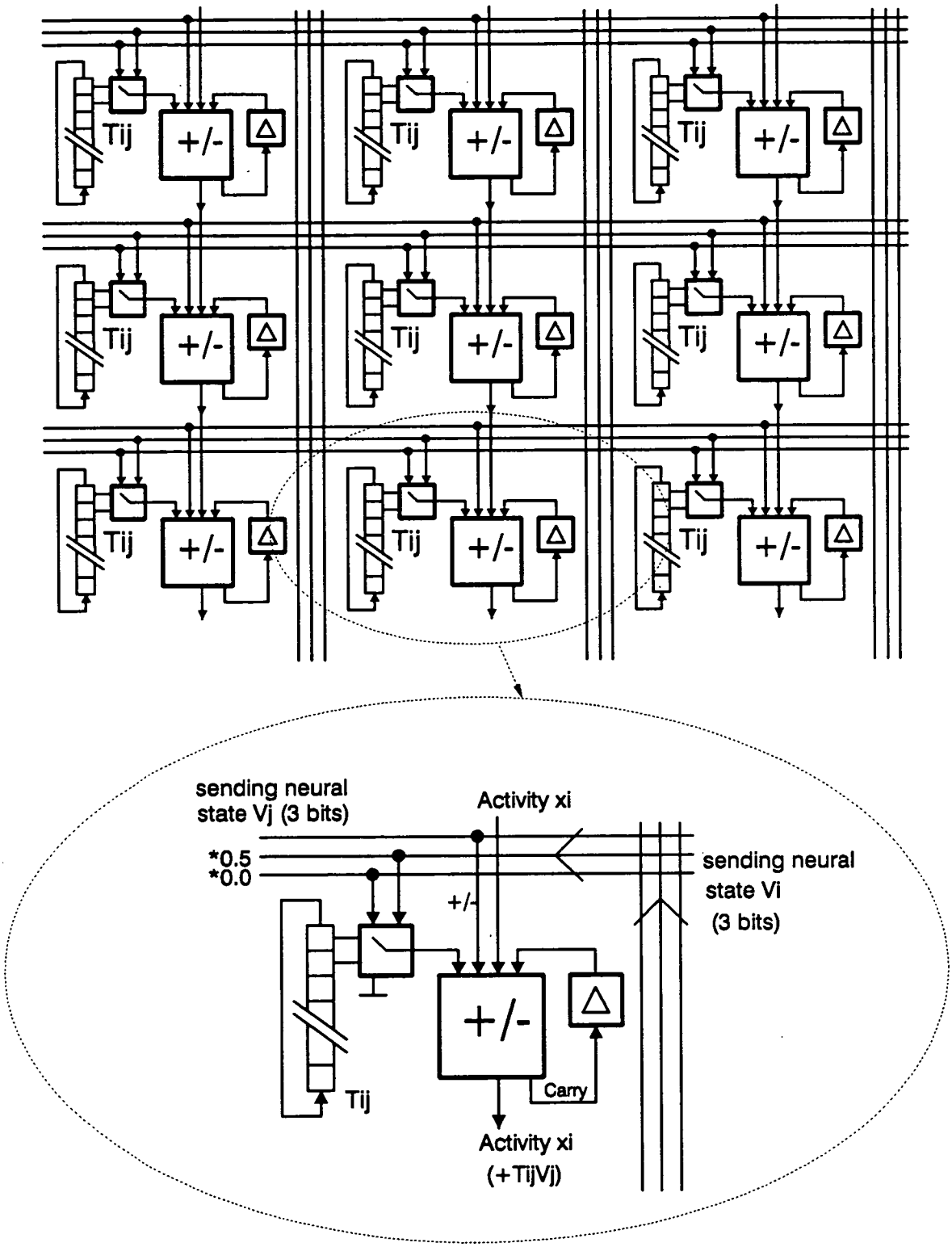


Figure 4.2 Synapse array structure, showing the 8-bit weight, 3-bit state multiplexor and adder/subtractor.

(adding 1)	+1
Therefore, the *two's complement* is	11111011
Hence, (+3)	00000011
(−5)	+11111011
	11111110

The result can be identified by computing the *two's complement*

The *one's complement* of 11111110 is	00000001
(adding 1)	+1
Hence, the result ($=+2_{10}$)	00000010

Therefore, the result represented by *11111110* is correct.

It is possible to add or subtract signed numbers, with the significant advantage that a carry generated by adding the *MSBs* is ignored in the value of the result. If this were not the case, the result would have to be corrected for the sign, each time.

A table giving the sum/carry logic output based on the equation 4.1 of a 16-bit word being added/subtracted is given in table 4.1.

$$\text{When } pm = \begin{cases} 0, \text{ then } \text{Sum} = \sum_{j=1}^{j=N} T_{ij-1}V_{j-1} - T_{ij}V_j - \text{carry} \\ 1, \text{ then } \text{Sum} = \sum_{j=1}^{j=N} T_{ij-1}V_{j-1} + T_{ij}V_j + \text{carry} \end{cases} \tag{4.1}$$

A *LSB* signal is synchronised to the LSB of the accumulating activity word and clears any carry that is generated at the addition/subtraction of the MSB of the previous activity word to avoid it being added into the LSB of the present activity word.

4.1.3. Synapse Array Design

The synapse array architecture is also shown in figure 4.2. Each neural state is input to the array on 3 pins and carried on a 3-bit bus. The first synapse in each column has an external input that allows accumulating activities to be added in when the integrated circuits are cascaded together. The last synapse in each column outputs the total

pm	$\sum_{j=1}^{j=N} T_{ij-1}V_{j-1}$	$T_{ij}V_j$	Carryin	Sum	Carryout
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	1	1
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	0	0
0	1	1	0	0	0
0	1	1	1	1	1
1	0	0	0	0	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	0	1
1	1	0	0	1	0
1	1	0	1	0	1
1	1	1	0	0	1
1	1	1	1	1	1

Table 4.1 Sum and carry output values calculated from equation 4.1.

activity. The three control signals are shifted in registers adjacent the array in order that they coincide with the activity computation.

Once the functional requirements of the synapse were realised, the transistor level design of the synapses could be begun.

4.2. Fully Custom Integrated Circuit Design

The main advantage of a fully custom design over a silicon compiled design, is that designing circuits at transistor level enables the minimum number of transistors to compute any particular logic function to be used, achieving optimum use of the silicon area. The technology used was 3μm 2-metal P-well CMOS. The CAESAR CAD

(computer aided design) tool was used to layout the complete circuit.

4.2.1. Weight Storage Shift Register

The single phase technique [93] used for the custom design was based on static logic trees charging and discharging dynamic latches (or shift registers). A logic tree is a circuit of transistors that computes a required logic function. It effectively uses two phases of the clock, the *logic 1* (5 volts), referred to as the π-*phase* and the *logic 0* (0 volts), referred to as the μ-*phase*. Figure 4.3 shows the shift register, with pre-charge circuitry. On the π-phase, the input is passed by n-transistor T1 to node 2 and inverted at node 3. Any input at *logic 1* will not be passed as a "good" logic *1* by transistor T1, therefore when the clock goes low, p-transistors T4 and T5 pull node 2 up to a good logic *1*. The value at node 3 is now passed by transistor T6 to node 7. The two "pull-down" transistors T8 and T9 ensure that any logic 0 passed by transistor T6, will be pulled down to a good logic 0 at node 7. When the clock returns low, the output appears 1 clock cycle later at node 10. Eight shift registers in each synapse hold the weight bit-serially.

The shift register had previously been fabricated on a 3μm CMOS integrated circuit [71] and the test results showed it functioned correctly up to 20 MHz. Tests were not attempted above this frequency owing to the limitations of the test equipment.

4.2.2. Synapse Logic Tree Design

The logic trees in the synapse required the pre-charge circuitry and a shift register to evaluate the correct output. Logic trees with the minimum number of transistors were obtained from boolean type functions or logic level descriptions of the state multiplexor, sum logic and carry logic by the *MOSYN* CAD Circuit Synthesis program [94, 95]. These descriptions of the circuit function that give the minimum circuits generated by *MOSYN* are given in Appendix A.

The minimum transistor logic trees are shown in figure 4.4. Each tree is joined to the pre-charge circuitry given in figure 4.3. On the first (μ) phase of the clock cycle (logic 0), node 11 is "pulled up" to a logic 1, regardless of the inputs to the transistors in the tree. During the π-phase of the clock, the node 11 evaluates the correct output. The inverter I12 allows node 11 to be "pulled down" to a good logic 0, if the transistor

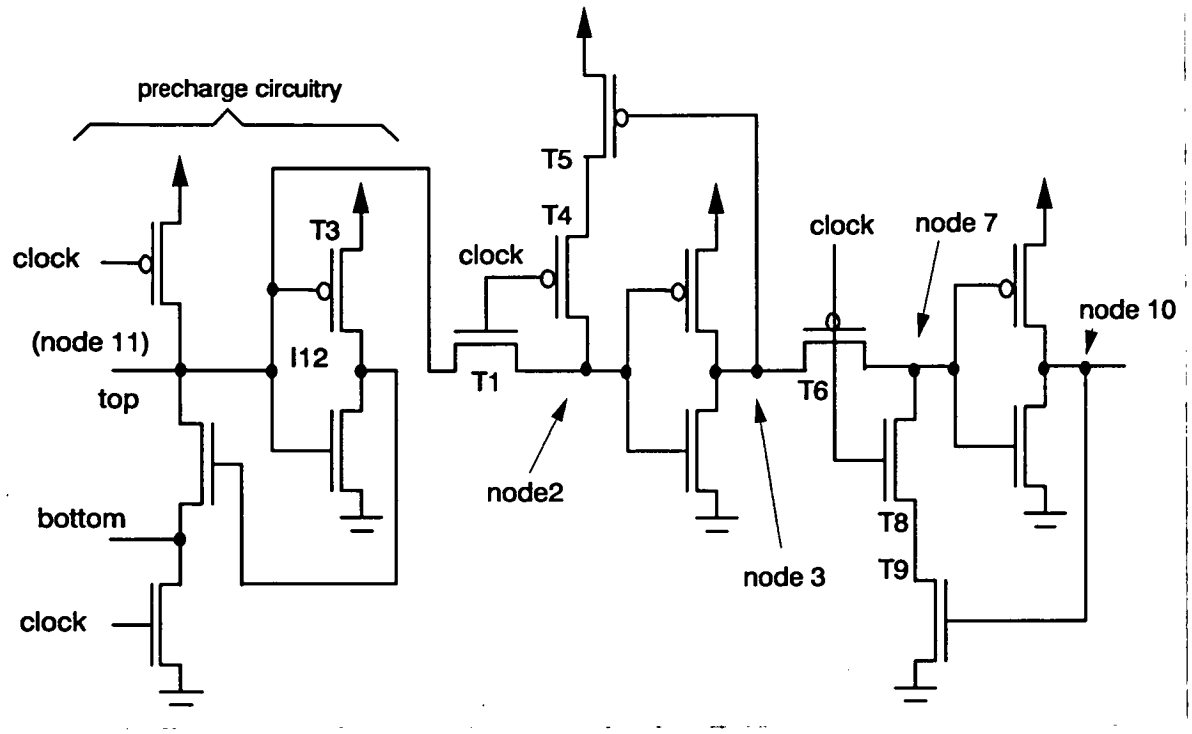


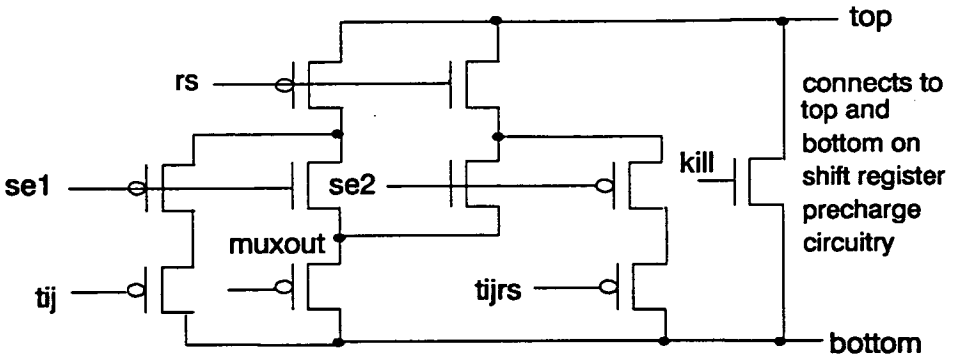
Figure 4.3 Single phase shift register.

inputs are such that there is a path to ground. The shift register outputs the correct value one clock cycle later. The correct logic function of each tree was verified by the *RNL* [96] timing logic simulator for digital MOS circuits. RNL is an event driven simulator that uses a simple RC (resistance capacitance) model of the circuit to estimate node transition times and to estimate the effects of charge sharing. Two intermediate programs *NETLIST* and *PRESIM* are required to be run first using a transistor netlist derived from the *MOSYN* netlist, to generate the correct binary netlist input *file.bin* for *RNL*. The netlist, *sum.net* the input *RNL* logic data file, *sum.l* and the output, *sum.out* generated by *RNL* for the sum logic tree are given in Appendix B.

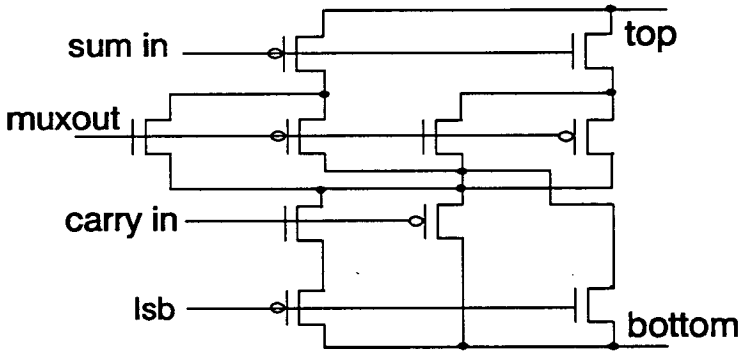
A complete synapse required 170 transistors. Once this had been verified by *RNL* as functioning correctly, the custom layout could proceed.

4.2.3. Synapse Array Layout and Simulation

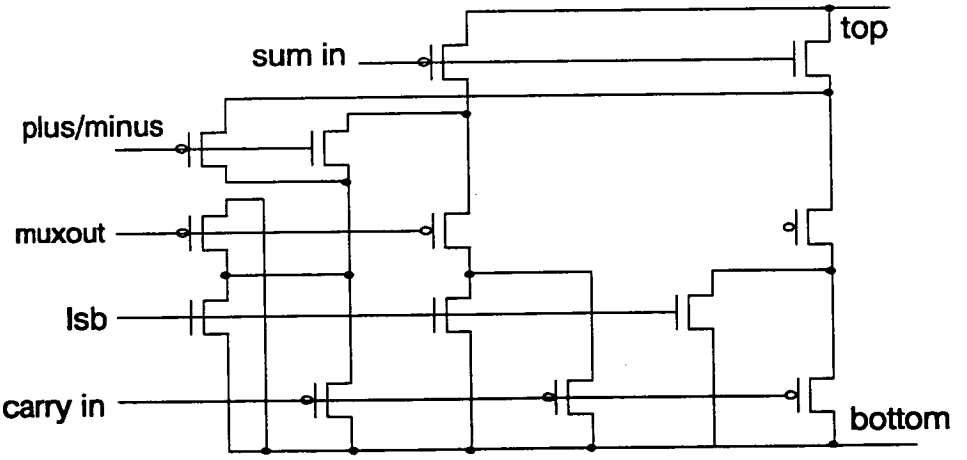
The layout of a fully custom integrated circuit involves each transistor being constructed from the nine layers available in the 2-metal CMOS process. First, the synapse was designed to be as compact as possible. Each transistor had the minimum



State logic tree



Sum logic tree



Carry logic tree

Figure 4.4 Transistor logic trees for the synapse.

geometrical dimensions, ie. 4μm long and 3μm wide. As the 10 transistor shift register had been previously designed [71] and shown to function correctly, the layout was used in the synapse. The metal 2 layer was used to carry all signals from the pads to the synapse array and for the *Vdd* (+5 Volts) and *GND* (O Volts) power lines. A layout plot of a synapse is shown in figure 4.5. The area of silicon required for 1 synapse was 670μm by 240μm and approximately one third of this was taken up by the 8-bit weight storage shift register. The layout also required substrate contacts to be placed every 70μm in the design. The CAESAR design rule checker *LYRA*, was employed at each new stage in the design to ensure that the layout dimensions did not violate the minimum geometries permitted for each layer.

Once the synapse design was completed a *CIF* (Caltech Intermediate Format) file was created by *CAESAR*. *CIF* is a low level graphics language used for describing integrated circuit layouts. The *CIF* file is then read by *MEXTRA*, a circuit extraction tool for VLSI simulation, to create a circuit description file for use by *PRESIM* as described in section 4.2.2. *RNL* was then run to verify that the layout was functioning correctly. The flow diagram given in figure 4.6 shows the design procedure using the CAD tools.

The second stage of the layout involved placing the maximum number of synapses in the allowed silicon area to form an array. The array size was 3 by 9 synapses, as is shown in a chip photograph in figure 4.7. The photograph in figure 4.8 shows a synapse in the array. This can be compared to the layout plot of a synapse in figure 4.5. The area of the array including the control circuitry shift registers was 5.70 mm^2 (2.49 × 2.29 mm) and accommodated 4958 transistors. The area of the chip including pads was 16.61 mm^2 (4.27 × 3.89 mm). *MEXTRA*, *PRESIM* and *RNL* verified that the layout was correct. The array was simulated as fully as possible by the CAD tools before fabrication by MCE (Micro Circuit Engineering).

A design of this size pushes the design capabilities of *CAESAR*, *RNL* and the Vax to their limits. Complete screen and paper layout plots took 2 to 3 hours and one column of 9 synapses required approximately 9 man hours and several hundred CPU hours of simulation. This makes the correction of errors a slow and tedious task.

The 3μm technology allowed 27 synapses to be fabricated in an area of 5.70mm^2. This may appear to be a disappointingly low number, but it is favourable when compared to

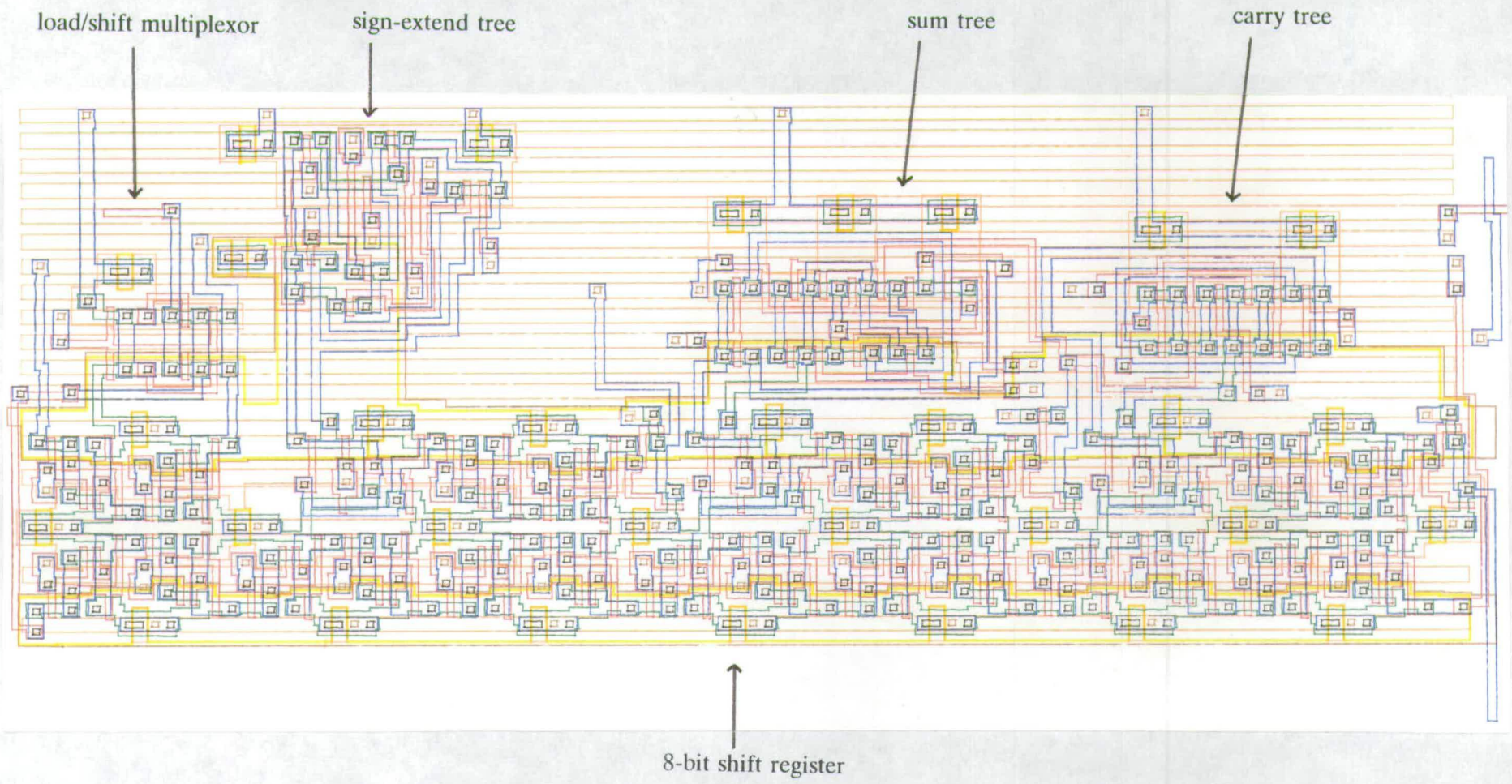


Figure 4.5 3μm CMOS technology penplot of one synapse

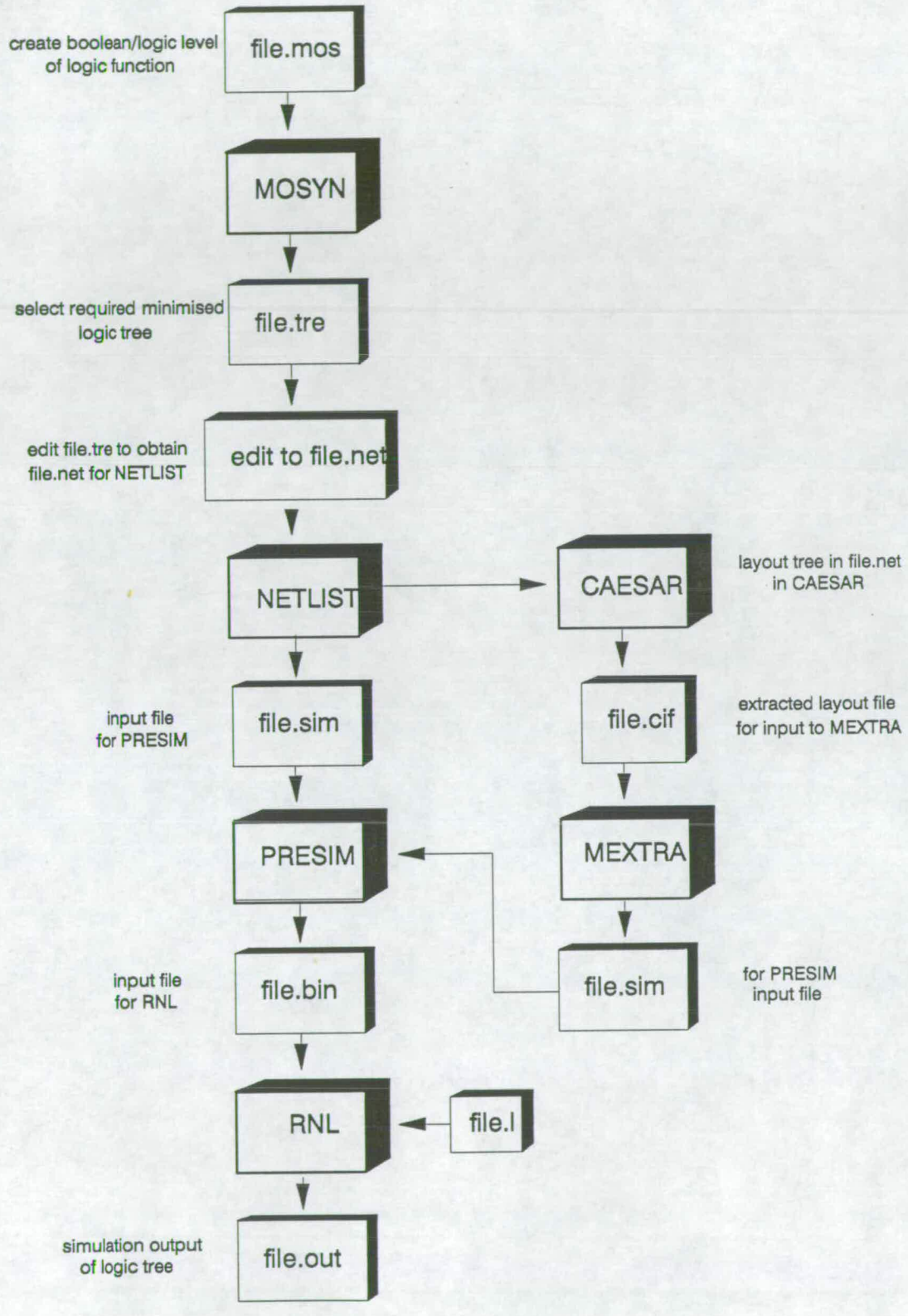


Figure 4.6 Flow diagram showing the stages in the chip design.

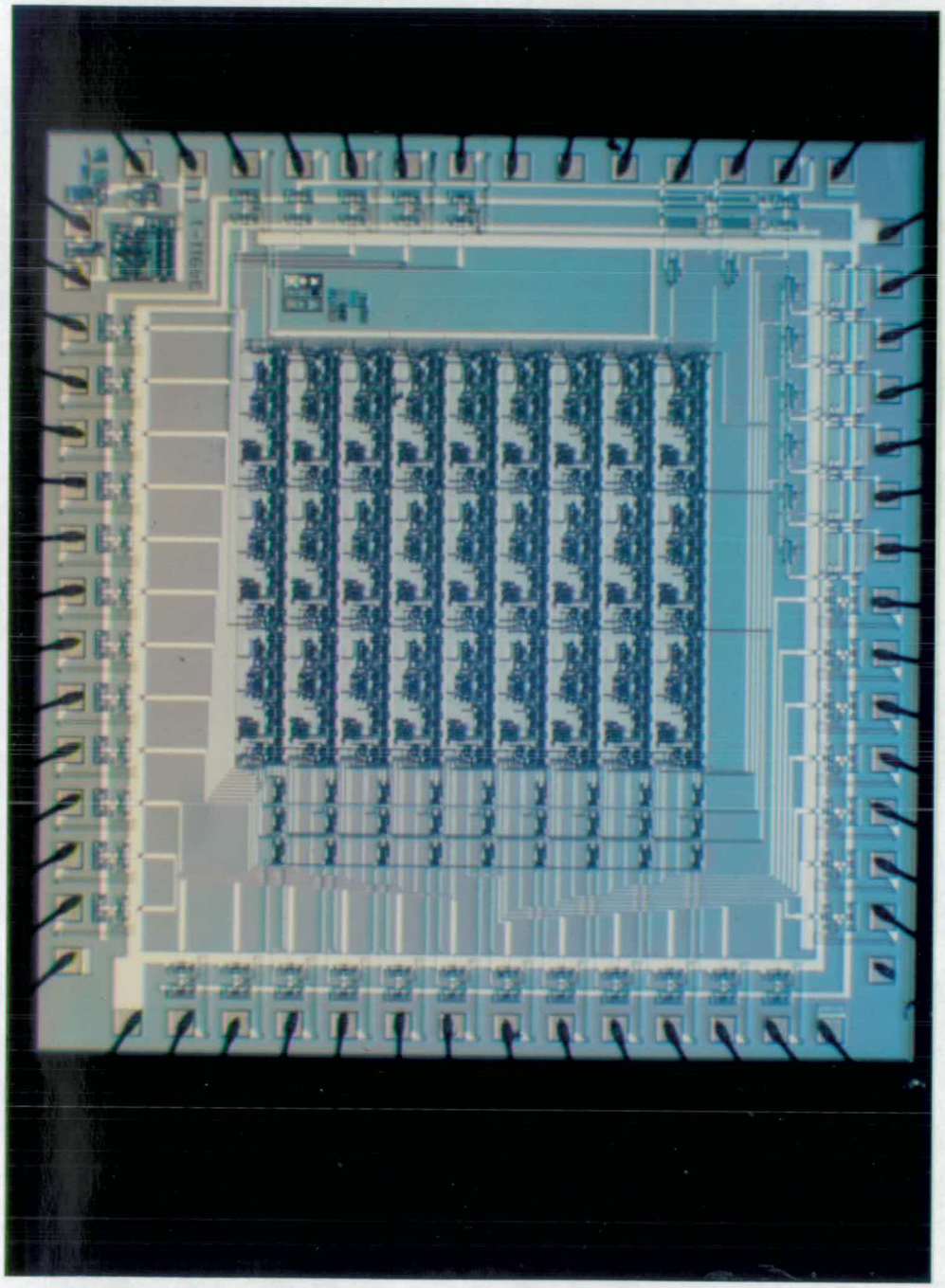


Figure 4.7 Silicon layout of the synaptic array.

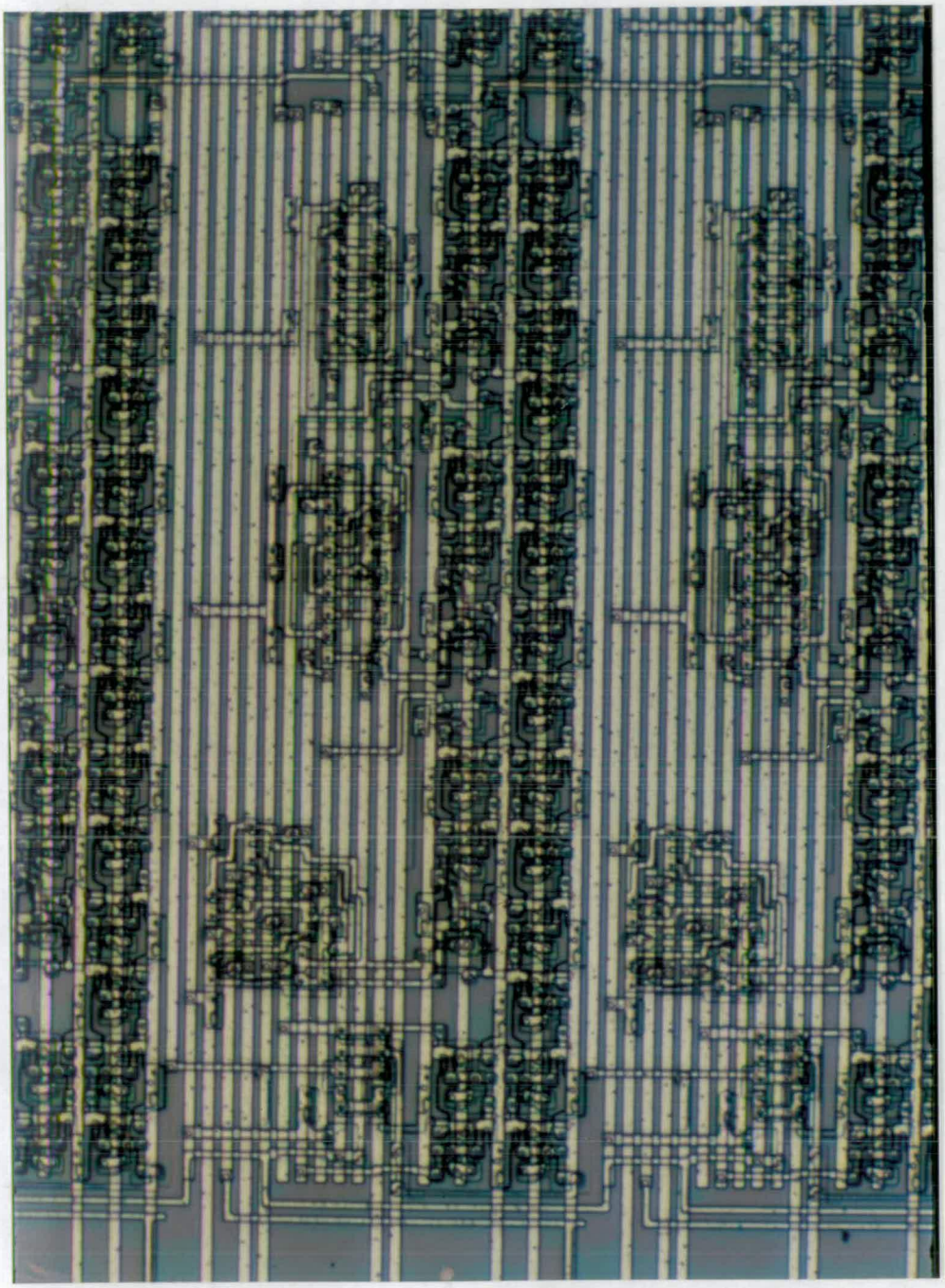


Figure 4.8 Silicon layout of a synapse in the array.

a full multiplier which would require approximately 1mm × 1mm area in 3μm technology. However if a projection is made to a 1μm minimum feature design on a 10mm × 10mm die, based on the area required for 27 synapses, it would be possible to achieve approximately a 58 neuron array (58^2 synapses).

For example,

If 27 synapses require 5.70mm^2 in 3μm technology, then 27 × 3^2 (=243) synapses require 5.70mm^2 in 1μm technology.

If the available area of silicon on a 10mm × 10mm die is 81mm^2 (excluding pad area), then the number of synapses = $\frac{81}{5.70} \times 243 \simeq 3400 \simeq 58$ neurons (58^2 synapses).

The size of this array is more suitable for learning and recall simulations.

4.2.4. Test procedure for the integrated circuits.

The manufactured integrated circuits were tested using a DAS (Digital Analysis System). It allows patterns of logic 1s and 0s that are generated by the user to be input to the integrated circuit under test and reads back logic 1s and 0s output from the device. The DAS generates a clock pulse up to a maximum of 5MHz.

The first test ensured the correct functioning of the 216-bit weight shift register, which is loaded bit serially and the 3 control shift registers for the signals sel, se2 and lsb described in section 4.1.2. The DAS timing diagram in figure 4.9 shows the registers shifting the data correctly. For example, the data on line "DOUT" which is the output from the weight shift register appears 216 clock cycles after the input on line "DATA" and the data is then shifted around the 8-bit shift register once the "load" line has returned to a logic 0.

The next test was to apply a neural state, V_j, to each row of synapses and observe the 3 outputs "sa out", "sb out" and "sc out", which should give the total activation in each column, ie.:

$$x_a = \sum_{j=1}^{9} T_{aj} V_j$$

where x_a is the total activity in column a and T_{aj} is the synaptic weight between neurons a and j. However, each device had the "sum out" output pins held at a logic

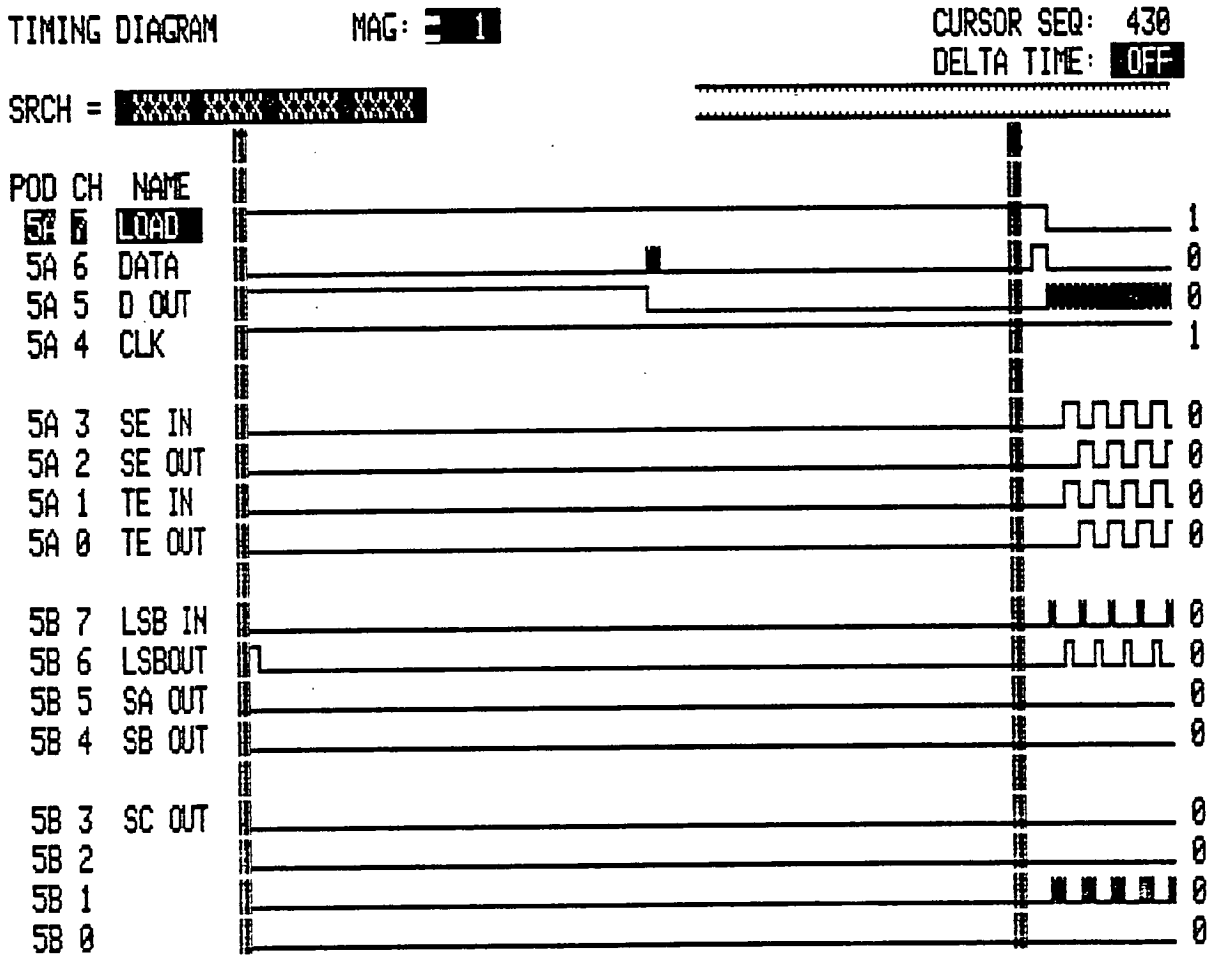


Figure 4.9 DAS timing diagram for the shift registers in the array.

0, implying that a design or manufacturing error had been made, that was holding the outputs to the GND line on the device. Careful examination of the layout plot of a synapse showed that a "contact" had been omitted from the precharge circuitry of the logic tree shift register. This had the effect of shorting the output of each logic tree to GND. The circuit extractor *MEXTRA* does not extract substrate contacts into the *file.sim* to be simulated by *PRESIM*. Hence the layout error was not detected. †

Once the substrate contact had been redesigned, the chip was manufactured by the EMF (Edinburgh Microfabrication Facility), University of Edinburgh, as the process was no longer available through MCE. Unfortunately, the resulting wafers had faulty metal layers and had 0% yield. At this stage, as the time to complete the research was running short, the only alternative route to achieve a fully working chip, was to completely redesign the synapse array was using the ES2 Solo 1400 Compiler for fabrication at ES2 (European Silicon Structures), as this process guaranteed working devices and a fabrication time of 8 weeks.

4.3. ES2 Solo 1400 Design

The Solo 1400 silicon compiler is a software tool for designing custom integrated circuits to manufactured in 2μm 2-metal N-well CMOS. The fundamental building blocks of the solo design are basic library parts. The parts are NAND, NOR, XOR, buffer, flipflops etc., which are stored in the Base Library and are recognised by the design subsystem. The Solo software provides facilities, which will guarantee a "working" integrated circuit if they are used in the correct order. The Base Library provided enough parts to design the neural accelerator chip. The following procedure was used for the design.

1. Design Entry: This allowed circuits to be input to Solo as a gate level schematic or in a text form using a Hardware Description Language (HDL) or in a combination of both.
2. Compilation: The Hardware Description Language (HDL) was compiled to produce an Intermediate Design Language (IDL) file, which was used by the simulation and physical design phases. At this stage the FANOUT of the design was checked to ensure that it met the fanout restrictions on components.

† The contact was between active area and metal1 in the inverter of the precharge circuitry, therefore the inverter had no *GND* connection. As the contact was in the active area of a substrate contact, *MEXTRA* assumed that a connection to *GND* existed. This meant the logic tree output was always pulled down to *GND*.

3. Simulation: This was done at gate level and had to be carried out before the physical design stage. The IDL file provided the input to the EXERT simulator. After the physical design a "load" file was produced that contained data on circuit capacitances. This allowed a second EXERT simulation file to check the maximum and minimum gate delays and to check that the final integrated circuit would work correctly for all operational conditions.
4. Physical Design: Circuit layout was performed by the PLACE, GATE, and ROUTE programs. Transistor sizing was already preselected in the Base Library and could not be changed. The only influence a user had in the PLACE program was in the placing of the gates in the layout. This was not necessary in the neural accelerator chip.
5. Validation and Production: Solo allowed the required package for the design to be chosen and checked that all the programs required for a "working" integrated circuit had been run in the predefined order. In this way, a correct functioning manufactured design was virtually guaranteed.

4.3.1. Synapse Gate Level Design

The synapse array function and size were kept the same as that in the MCE 3μm design. The reason for this was that the neural board with the hardware support for the accelerator chips had already been designed specifically for a 3 × 9 synapse array.

The boolean and logic descriptions for the state multiplexor, sum logic and carry logic that were described in section 4.2.2 were used to represent the functions at gate level. Figure 4.10 and 4.11 shows the Solo gate level design for this. The dynamic shift register used for weight storage and logic tree evaluation in the previous design was replaced by a static D-type flipflop. Each D-type consists of 10 gates (30 stages). Solo defines a stage as "a single, equal size p and n type transistor". The complete synapse circuit required 331 stages (662 transistors). This is approximately four times as many as those needed for the CAESAR synapse. The reason for this is that Solo has predefined parts consisting of logic gates that perform specific functions. The synapse had to use the available parts wired together to perform the required logic function, whereas the synapse designed with CAESAR, used the minimum possible number of transistors to perform the logic function. At this point the synapse was simulated by

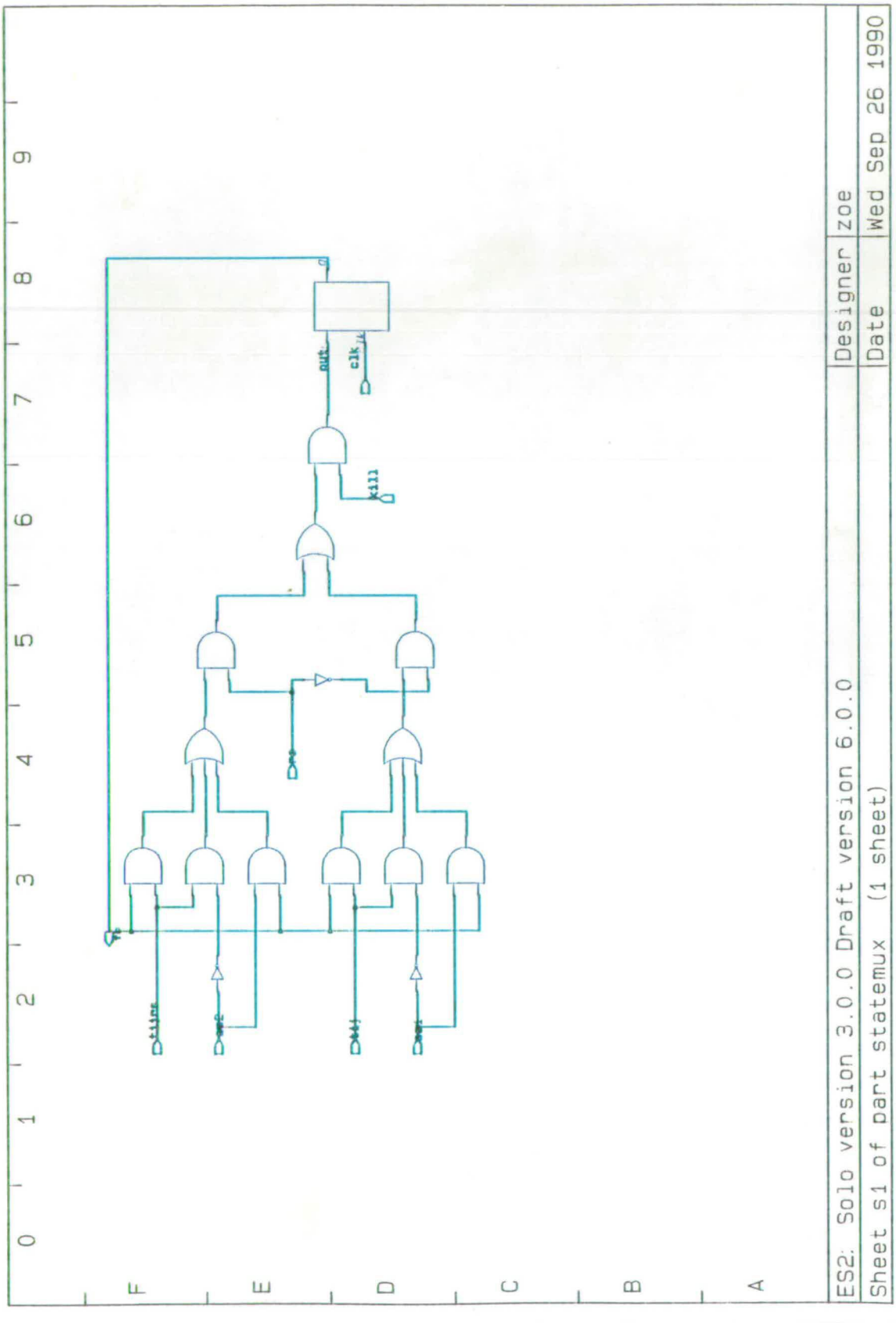


Figure 4.10 Gate level design of the sign-extend logic.

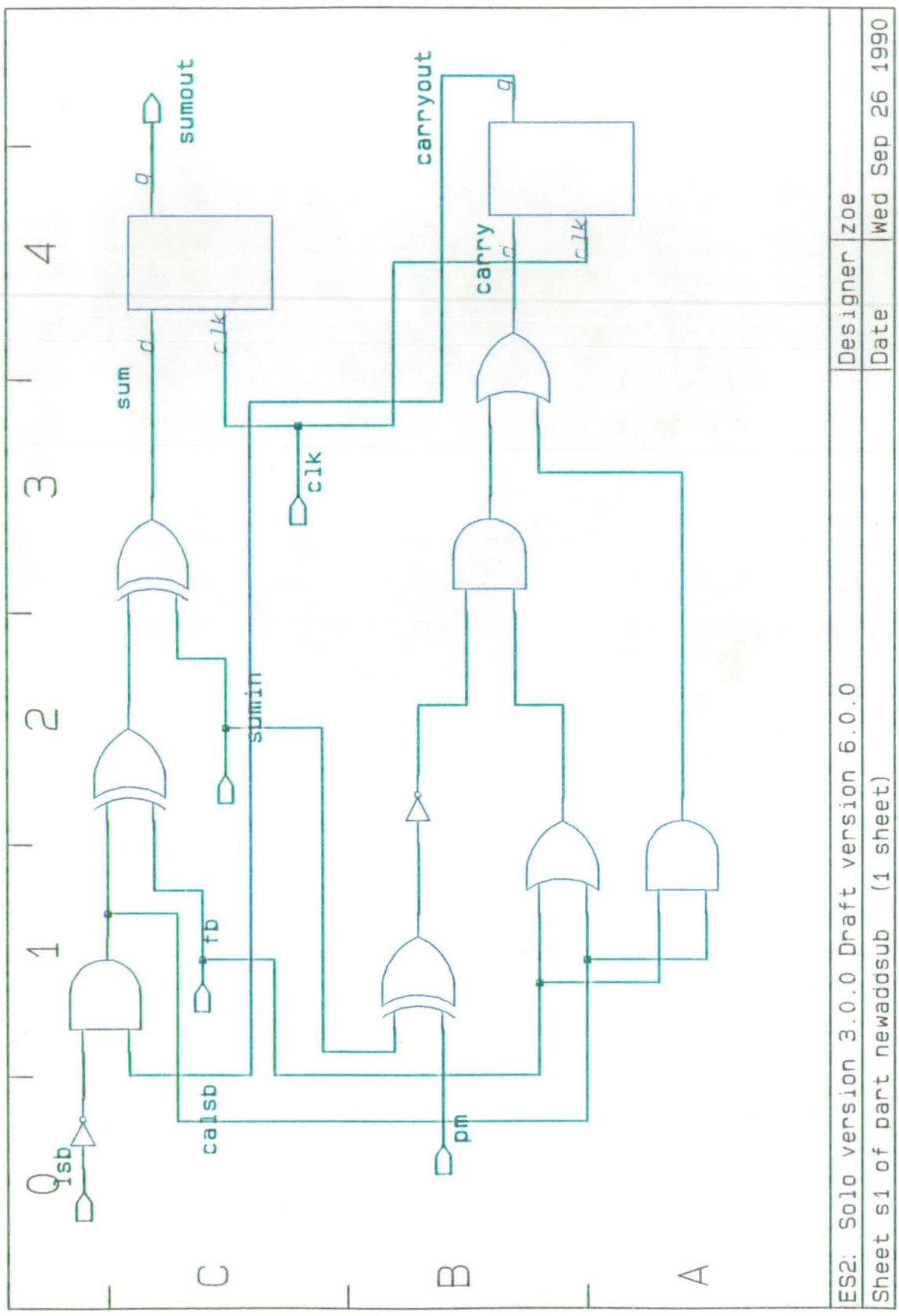


Figure 4.11 Gate level design of the sum/carry logic.

the EXERT simulator to verify that it was adding, subtracting and shifting without error.

4.3.2. Synapse Array Layout and Simulation

A 3 × 9 synapse array was constructed again at the gate level design and D-type flipflops were used for the control circuitry shift registers. Figure 4.12 shows the Solo interpretation of the synapse array. Each "part" represents the 3 synapses in each row sharing the same neural state. The three "sum out" lines are adjoined to the adjacent "part" to the right. The final "sums out" appear at the rightmost "part", the ninth synapse in each column. The 8-bit and 9-bit shift registers required for the lsb, sel and se2 signals are below the synapse array. Once the EXERT simulator had verified the function of the array, the design work for the accelerator chip was complete.

The layout procedure was performed entirely by the compiler using the PLACE, GATE and ROUTE programs. EXERT was used again as a final check that the gate delays and FANOUT were still within the set limits. As with the CAESAR, the simulation on Solo, proved to be the slowest part of the design, taking approximately the same man and CPU hours to achieve a full simulation of the array. For this, at least 95% of the transistor nodes had to toggled as part of the design validation process.

Another design validation constraint required that all metal track lengths must be less than 10000 μm. As neural network designs have a large number of interconnects between parts, several long track lengths occurred in the routing around transistors. Therefore buffers had to be added on all long track lengths. Figure 4.12 shows the *clock* and *load* signals with buffers dividing up the lengths of track. The area of silicon used for the array was 19.28 mm^2 (4.79 × 4.03 mm) made up from 9360 stages (18720 transistors) and the area of the array including pads was 29.13 mm^2 (5.78 × 5.04).

4.3.3. Test Procedure for the integrated circuit

The manufactured integrated circuit was tested using the Digital Analysis System with the same data input programs generated for the MCE design test. The first test showed that the 216 weight shift register and the 3 control shift registers had the correct function. This is shown in figure 4.13. The second test applied a neural state to

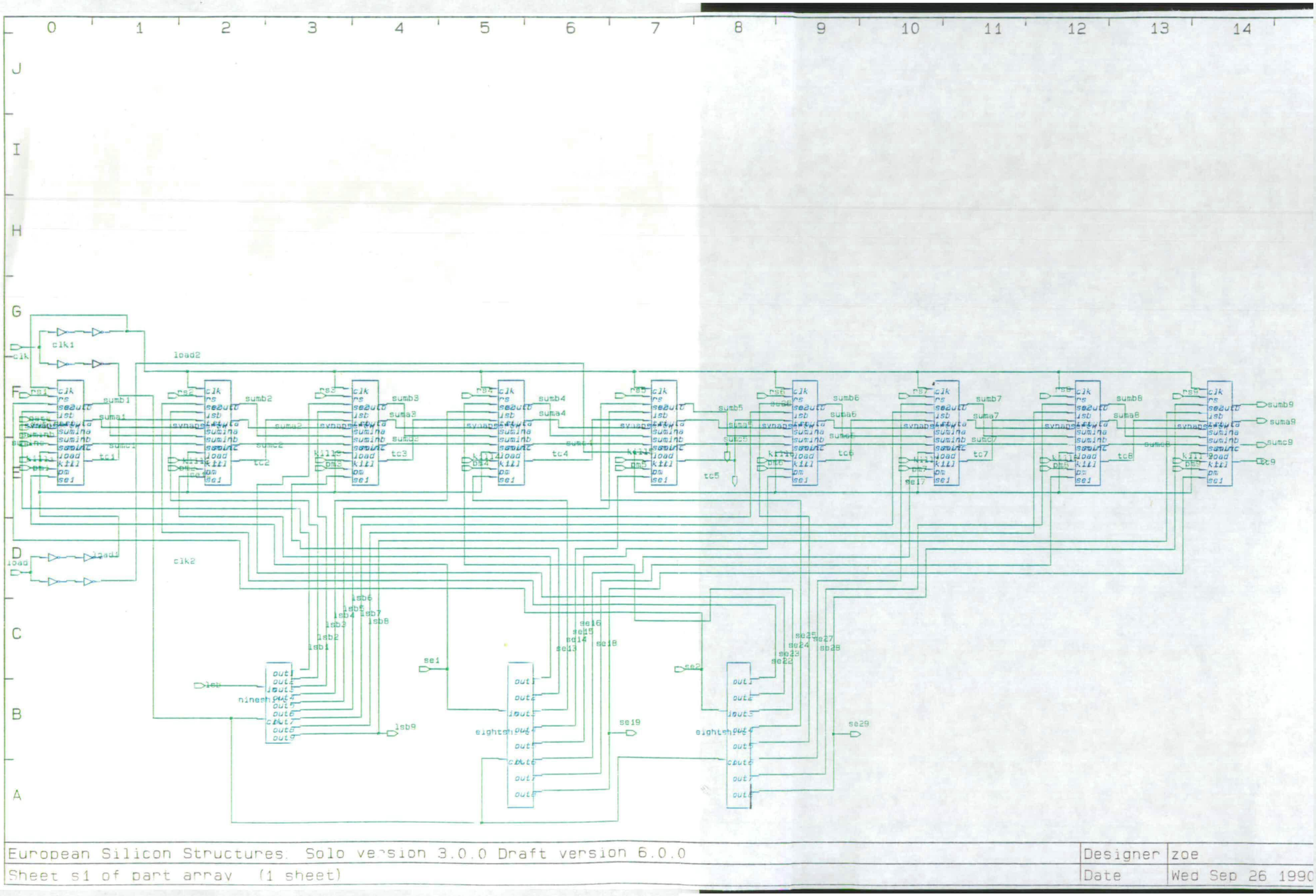


Figure 4.12 Solo interpretation of the synapse array

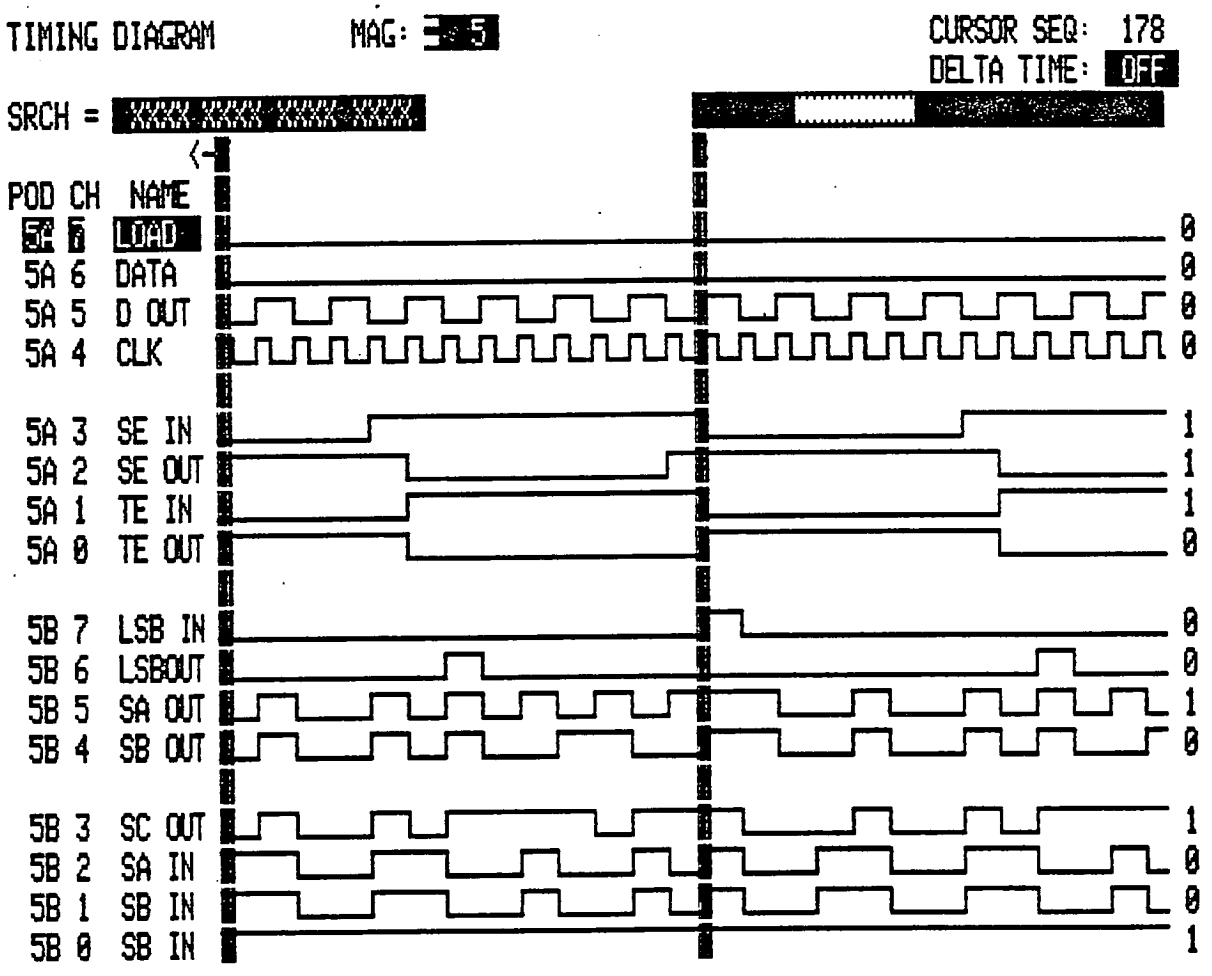


Figure 4.13 DAS timing diagram for the Solo synapse array.

each synapse row and allowed the chip to compute the total activity in each column according to equation 4.1.

The following tests were done.

1. All T_{ij} = 0 and all V_j = 1, "sa in", "sb in" and "sc in", the sums at the top of each synapse column were given 16-bit random words.

The integrated circuits gave the correct results of "sa in" = "sa out", "sb in" = "sb out" and "sc in" = "sc out".

2. Positive and negative random values of T_{ij} were input to the array and each synapse row was given a known state. x_a, x_b and x_c were calculated by hand and then compared to the values calculated by the integrated circuit. The answers were the same indicating that the array was computing in the correct way. These results were obtained at 5MHz, the maximum operating frequency of the DAS.

Once the integrated circuit function had been verified, four of the chips were mounted with hardware support on a board. The hardware support consisted of memory to hold the neural weights and states and control circuitry to supervise the calculations in and out of the neural accelerator chips. The board was interfaced to a host Sun 3/110 Work station and its function is described fully in Chapter 5.

4.4. Conclusions

The designs of the two synapse array integrated circuits using CAESAR and Solo 1400 CAD tools gave interesting comparisons between the tools. Although the two designs had exactly the same function, major differences occurred in the design time, the number of transistors used and the area of silicon used as shown in table 4.2.

Design	Synapse transistor count	Array transistor count	Array area in mm^2	Chip area in mm^2	No. of weeks to design	No. of weeks to manufacture
CAESAR	170	4958	5.70	16.61	36	24
ES2 Solo	662	18720	19.28	29.13	8	6

Table 4.2 Comparisons between the MCE and Solo designs.

The advantage of the layout facility of CAESAR was that it allowed a compact design using the minimum number of transistors possible for the function. This was proved by the large difference in the number of transistors per synapse. The ES2 design had four times as many transistors and used five times as much silicon area, allowing for the minimum geometry differences in the process. The disadvantage of CAESAR was that the layout process was long and tedious and prone to errors.

The major advantage of Solo was the turnaround time from the initial design to receiving working chips. This was 14 weeks for the neural accelerator chips compared to 60 weeks for the CAESAR/MCE design. Solo was easy to learn and by its design facilities guaranteed that the chip would function in hardware.

The software simulation times on both designs were approximately the same, although the Solo simulation had rigorous design constraints that had to be adhered to, hence the guarantee of working silicon.

Chapter 5

Neural Network Accelerator Board

The neural network integrated circuit described in chapter 4 is a 3 × 9 synapse array using a reduced arithmetic technique to compute the neural function $x_i = \sum_{j=0}^{n} T_{ij} V_j$, where x_i is the activity of neuron i, T_{ij} is the synaptic weight between neurons i and j and V_j is the state of neuron j. The integrated circuit is to be used as a hardware accelerator for simulations as, for example, described in chapter 3 for the learning and recall of patterns.

The size of the synapse array on one integrated circuit alone was too small to be used with a learning algorithm to perform learning and recall simulations. SPICE simulations of the single-phase clocking technique showed that the neural accelerator would operate upto 20MHz allowing each synaptic column to compute the activity in a minimum time of 1.3μs and if the synaptic weight set were stored in supporting RAM (random access memory) with an access time of 45ns, the weights for 27 synapses could be loaded into an integrated circuit in 9.72μs. Therefore, the projected minimum total computation time for the three activities is 11.02μs. This performance of a hardware accelerator is much faster than speeds attainable in a natural neural network. Hence, a *paging* architecture described in the following section has been developed to "trade off" some of this excessive speed for increased network size.

5.1. "Moving Patch" Paging Architecture

To increase the number of simultaneous synaptic calculations, 4 of the chips have been cascaded to give a 12 × 9 array. The *paging* architecture uses this new array size to give a virtual array size of 288 potentially totally interconnected neurons and acts as a neural accelerator to a host Sun computer. The *paging* architecture can be visualised as a "moving patch" where the small "patch" (the 12 × 9 array) simulates a small number of synapses sliding across a much larger array. On each new simulation, the "patch" moves down the synaptic column to the adjacent patch and the new synaptic weights are loaded into it. A general architecture showing this is in figure 5.1.

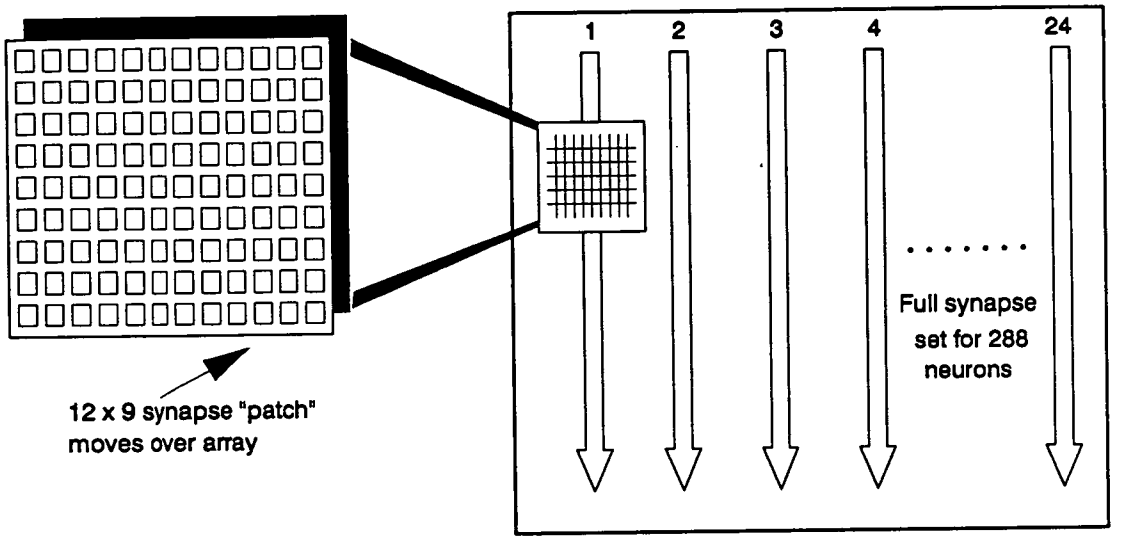


Figure 5.1 *Paging* architecture of passing a small synaptic "patch" over a larger synaptic array.

Each "patch" computes the partial activity for each column of synapses. These activities are subsequently held in memory until the next "patch" is ready to compute. The partial activities are then fed into the top of the new "patch" to be added to the new activities being computed. This ensures that each synaptic column receives a contribution of activity from each synapse in it. A virtual array size of 288 neurons and a "patch" size of 12 × 9 synapses gives 32 "patches" in a column and 24 columns of "patches". After 32 iterations the "patch" reaches the bottom of the first column, the total activities for the first 12 neurons have been computed and are then stored in local RAM. The "patch" then proceeds to iterate down the next column of "patches", until all 24 columns have been iterated. When all the activities have been computed, they are downloaded to the host, which then thresholds them according to the 5-state activation function to form the new set of neural states for the array. The host computes the learning and the weight updating for the network. Details of the neural accelerator functioning as a pattern associator using the *delta* learning rule are given in Chapter 6.

5.1.2. Hardware Support for the Paging Architecture

The neural network accelerator board requires hardware support circuitry to perform the *paging* operation described in the section above. The board runs as a SLAVE

module, which responds to the data bus transfer (DBT) operations generated by the MASTER, the host Sun processor board. The neural accelerator board is interfaced to the host Sun by the Sun's VMEbus [97]. The VMEbus allows communication between the MASTER and the accelerator board and in this case, it enables the weights and states calculated by the host to be transferred to the on-SLAVE RAM. The accelerator board runs independently from the host while it computes the neural activities. Once this has been done, the board signals to the host to read and threshold the activities.

One complete computation cycle requires the synaptic weights and neural states for every patch to be unloaded from on-board RAM, the partial activities between adjacent patch computations to be stored in shift registers and the total activities to be stored in memory, until the "patch" has iterated across the whole board. Therefore, 3 separate RAMs are required to store the synaptic weights, present neuron states and new neuron activities. Twelve 16-bit shift registers synchronise the partial activities of the previous "patch" to be added correctly to the present "patch" computation and an 18-bit counter controls the "patch" iteration across the array. Figure 5.2 shows the main structure of the board.

These parts constitute the major components of the neural accelerator board, that would be required regardless of the host and interface environment for the activity thresholding and weight updating. However, the operation of the VMEbus interface system influenced some of the details of the board design in order to tailor it to the VMEbus specification requirements for the transferring of data between the SLAVE and MASTER. Hence the neural accelerator board is divided into 2 parts. The first is buffering and control circuitry to allow the VMEbus to write to and read from the SLAVE RAMs. The second part is buffering, control circuitry and a SLAVE address system for the neural activity computation when the board is running independently. Section 5.2 gives a brief description of the VMEbus and its implementation with respect to the neural accelerator board requirement and section 5.3 gives the subsequent design of the board, showing details of how the "patch" computation and the "patch" iteration is achieved.

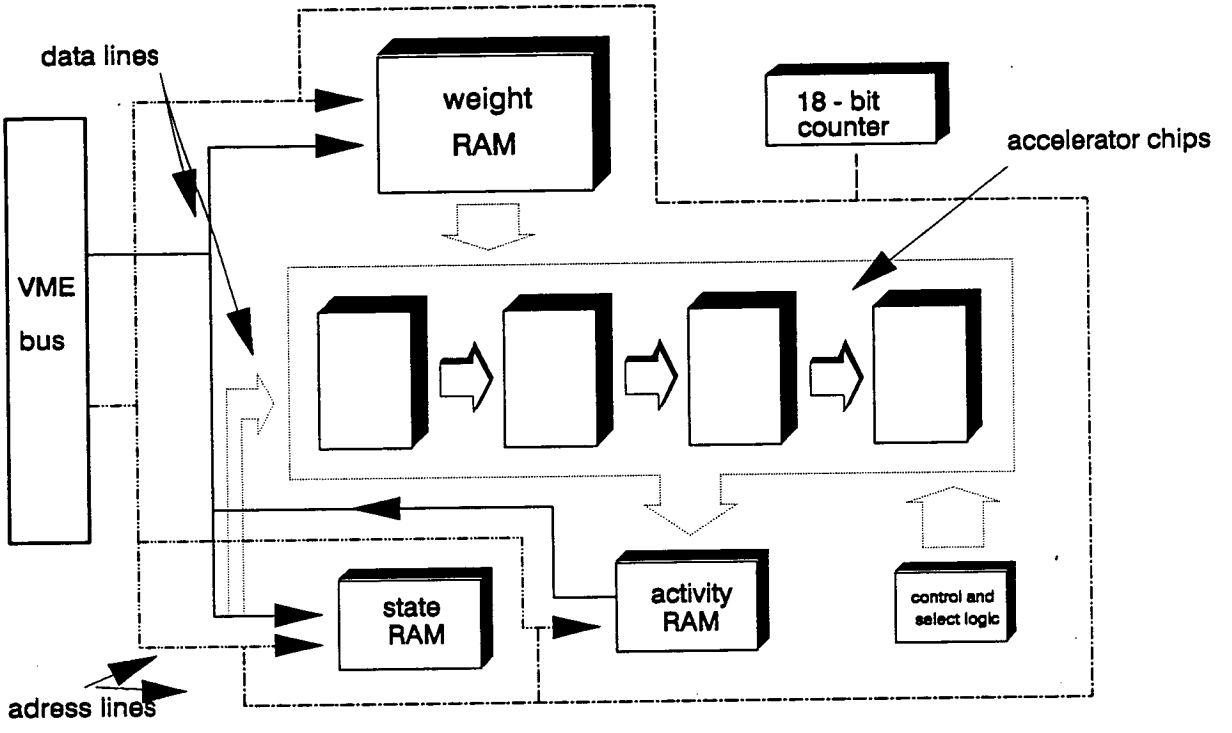


Figure 5.2 Main structure of the neural accelerator board

5.2. VMEbus Interface

The VMEbus is an interfacing system for use in interconnecting data processing, data storage and peripheral data control devices in a closely coupled configuration. The mechanical structure of the VMEbus is a backplane, which is a printed circuit (PC) board with a pair of 96 pin connectors that provide the bus signal paths to the SLAVE module. The VMEbus interface system consists of backplane interface logic that takes into account the characteristics of the backplane (its signal impedance, propagation time, terminal values, etc.), four groups of signal lines called "buses" and a collection of "functional modules" that can be configured as required to interface devices to the buses. The functional modules communicate with one another by means of bus signal lines provided by the backplane.

The interface functions of the VMEbus are divided into 4 areas. Each functional area consists of a bus and associated functional modules which work together to perform specific duties within the interface system. The only area used by the neural accelerator and therefore discussed here is that of "Data Transfer". Details of the other

three areas - Data Transfer Bus Arbitration, Priority Interrupt and Utilities can be found in [97-99].

The Data Transfer bus (DTB) contains the data and address pathways and associated control signals. In this area, functional modules called "DTB Masters" and "DTB Slaves" use the DTB to transfer data between each other. The VMEbus MASTER allows 32-bit (long word), 16-bit or 8-bit (short word) data transfers and 32-bit, 24-bit or 16-bit addresses. The neural accelerator board requires a 32-bit data bus as the neural states for one neural accelerator chip are signalled on a 27-bit and a 24-bit address bus. The SUN processor board allocates specific areas of its memory for the different sizes of VME data and address buses [99, 100]. The area of memory in the 24-bit VMEbus Address Space Allocation reserved for "small user devices" has sufficient memory space for the data which is to be transferred to the neural accelerator board. The physical (hexadecimal) address range of this area is 0xD00000 - 0xDF0000 representing the addresses A_{23} - A_{20}, A_{19} - A_{16}, A_{15} - A_{12}, A_{11} - A_8, A_7 - A_4 and A_3 - A_0. Address lines A_{23} - A_{19} are permanently at address 0xD (1011_2).

5.2.1. Slave Interface to the VMEbus

The 3 neural accelerator board RAMs are addressed by the VMEbus. The weight RAM requires 12 data lines (as 12 RAM chips are required to store the weights for 288 neurons) and the neural state RAMs require 27 data lines to load the states (the maximum number of data lines is 32), therefore the physical address range in the VMEbus address space allocation is split up so that each of the 3 RAMs is addressed individually, as is shown in table 5.1. The sectioning of the Address Space Allocation allows the address line A_{19}–A_{16} (ie. the lines signalling address 3,4,5 or 6 to determine the accessing of the RAMs) to be gated to form the "chip select" and "read/write" control signals to each RAM. Figure 5.3 shows the SLAVE interfacing to the VMEbus and how the address lines have been used for the various control signals.

When the neural accelerator board runs independently of the MASTER while it is computing the neural activities, all the VMEbus data and address lines must be buffered to allow them to be switched off. The address lines use uni-directional buffers and the data lines use bi-directional buffers as data is written to and read from the SLAVE. Table 5.1 shows that data is written only over the VMEbus address range

Address	RAM
D6xxxx	"RUN" signal to SLAVE to compute activities
D5FFFF to D50000	Address and read from "activity" RAM
D4FFFF to D40000	Address and write to "neural state" RAM
D3FFFF to D00000	Address and write to "synaptic weight" RAM

Table 5.1 VMEbus address space allocation to the SLAVE RAMs

0xD00000 - 0xD4FFFF. Over the range 0xD50000 to 0xD5FFFF the activities are read back from the SLAVE. In figure 5.3, the signal "read" is active low on address 0xD5FFFF, which sets the data buffer direction so that data may be read from the board only at VMEbus address 0xD5XXXX. The VMEbus has its own control lines for the Data Transfer Bus which are used in the SLAVE interface design. These are given in Appendix C.

Once the MASTER has written the weights and states to the neural accelerator board, the VMEbus address is set to 0xD60000, which sets the signal "run" shown in figure 5.3 to "active low", to enable the accelerator board to compute independently. The "run" signal is gated to the interface buffers so that they become disabled at this address. When all the activities are computed, the VMEbus switches to address 0xD50000 to read back the new values.

5.3. Neural Accelerator Board Architecture

The VMEbus address 0xD60000 allows the SLAVE operation to be controlled completely by an 18-bit counter. The frequency of the counter is determined by a quartz oscillator. The counter controls the paging architecture of the "patch" giving the 288 neuron array. It also provides the address lines, chip select and read/write lines to the RAMs and the control signals necessary for the neural accelerator chips. Table 5.2

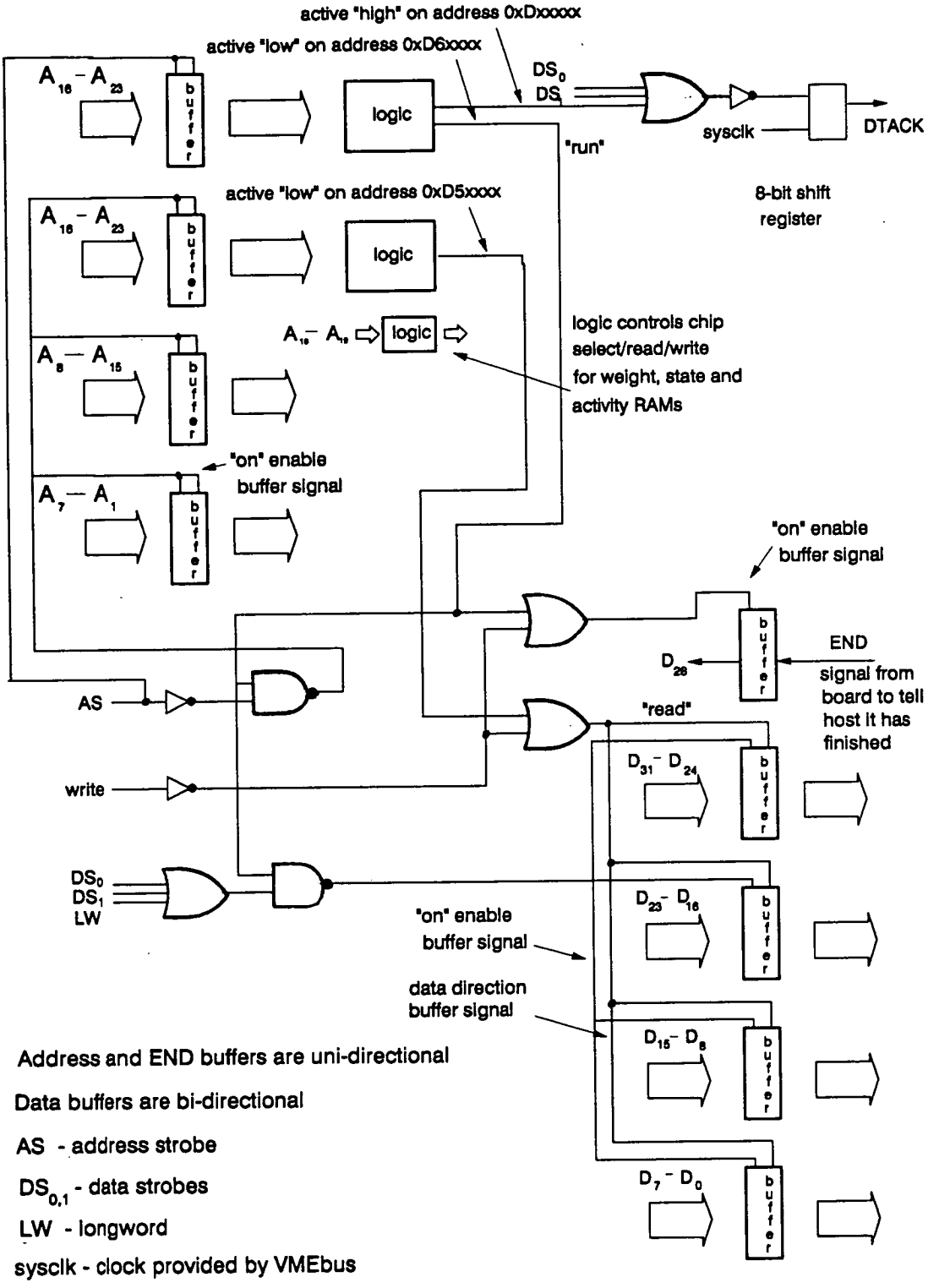


Figure 5.3 SLAVE interface to the VMEbus

shows how the counter controls the "patch" iteration.

Count or Address Line	$A_{17}\,A_{16}\,A_{15}\,A_{14}\,A_{13}$	$A_{12}\,A_{11}\,A_{10}\,A_9\,A_8$	$A_7\,A_6\,A_5\,A_4\,A_3\,A_2\,A_1\,A_0$
Function	5 MSBs count the number of columns of 32 "patches" (ie. 0 - 23) to give 288 neurons	Next 5 bits count the number of "patches" in a column (ie. 0 - 31) to give 288 synapses per column	8 LSBs count 1 complete "patch" of 12 × 9 synapses computation (0 - 255 clks)
Example:	$1\ 0\ 0\ 1\ 1_2$	$1\ 0\ 1\ 1\ 0_2$	$x\ x\ x\ x\ x\ x\ x\ x_2$

Table 5.2 The function of the SLAVE 18-bit counter

The example in table 5.2 gives addresses A_{12} - A_8 as $10110_2 = 22_{10}$ and addresses A_{17} - A_{13} as $10011_2 = 19_{10}$. Therefore, this address implies that the 23rd patch in the 20th column is computing the 23rd partial sum of the 20th neuron. Address A_7 - A_0 counts through the patch computation. The paging architecture of the "patch" is given in greater detail in section 5.3.2.

5.3.1. "Patch" Computation

A "patch" uses the 8 LSBs of the board counter (ie. clocks 0 - 255) to compute the partial activity, with the remaining 11 bits of the counter being used to count the iteration of the "patch" over the whole array, as is described in the section above, to achieve 288 neurons. Figure 5.4 gives a flow diagram explaining the "patch" operation. The first 216 clock cycles require that the 8-bit weights for the 27 synapses are loaded bit-serially into each accelerator chip. The neural states are applied to the accelerator chips on clock cycles 216 - 241 while the activity is being calculated. The shift registers are active on clock cycles 217 - 232 to allow the previous partial activity of the adjacent "patch" above to be added to the present "patch" partial activity. The LSB of the present partial activity exits the accelerator chip on clock cycle 226 and the partial activity is loaded to the shift register over the clock cycles 226 - 241, where it is held

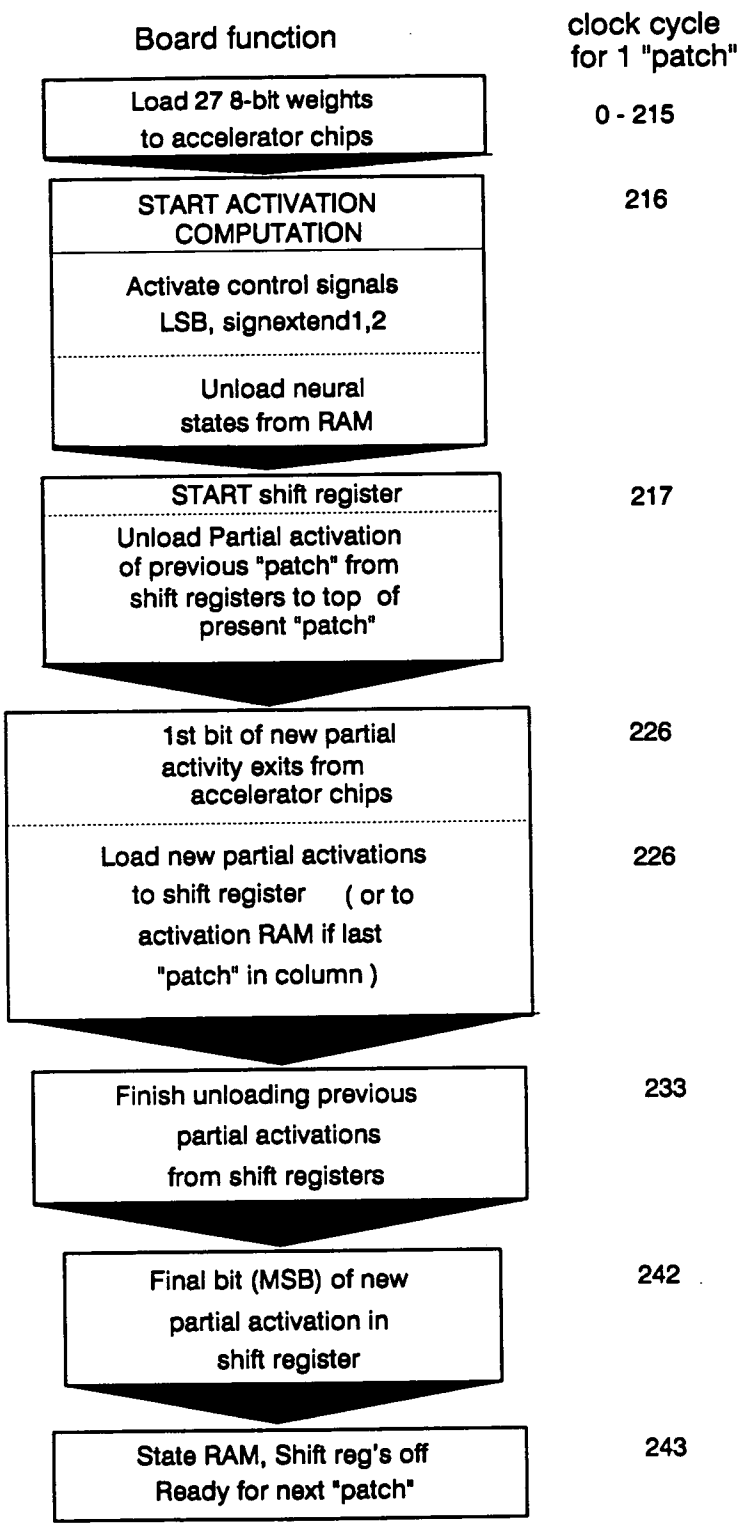


Figure 5.4 Flow diagram explaining the "patch" operation

until it is required for the next adjacent "patch" in the column. There is, therefore, an overlap of 7 clock cycles (226 - 232) while the shift registers are unloading the previous partial activities and simultaneously loading the new partial activities. If the "patch" is the last one in a column, the activities computed will be the total activities for that column and hence will be loaded directly to the activity RAM, instead of the shift register, to be read back later by the VMEbus. The shift registers are then cleared for the start of the next column as the first "patch" in each column does not have any previous partial activity to be added in.

A detailed schematic of the neural board is shown in figure 5.5. The weight RAM requires 12 - 64K x 1 data bit RAMs, the state RAM requires 4 - 2K x 8 data bit RAMs for 27 state lines (3 lines per state) and the activity RAM requires 2 - 2K x 8 data bit RAMs for 12 activity lines per "patch". The weight and state RAMs are written to and addressed by the VMEbus, and read on being addressed by the board counter. The activity RAM is written to on being addressed by the board counter and read from, when addressed by the VMEbus. Buffering is therefore required to separate the board data and address lines from the VME data and address lines to avoid contention. The maximum operating speed of all the RAMs is 45ns. The partial activities are stored in 12 16-bit shift registers (2 each × 8-bit shift registers). The control circuitry is made up of separate integrated circuits which include 2, 3 or 4 input NOR, NAND, OR, AND and inverter gates. This provides the signals "load", "lsb", "sel", "se2" for the accelerator chips as described in Chapter 4, along with the shift register clock signal and the "chip select","read/write" and "output enable" signals for the weight, state and activity RAMs. The "load" signal allows the synaptic weights to be loaded into the accelerator chips and is active "high" for the clock cycles 0 - 215. When the "load" signal is "low" on clock cycles 216 - 255, it allows the weights to be shifted around its 8-bit shift register during the activity computation. A waveform diagram showing the board control signals is given in figure 5.6.

5.3.2. Array Computation

As described earlier in the chapter, the 288 neuron array is achieved by iterating the "patch" 32 times down a column across 24 columns. The "patches" in a column are counted by the board addresses A_{12} - A_8 and the number of columns is counted by the addresses A_{17} - A_{13} as is shown in figure 5.7. The 12 x 9 synapse array of the "patch"

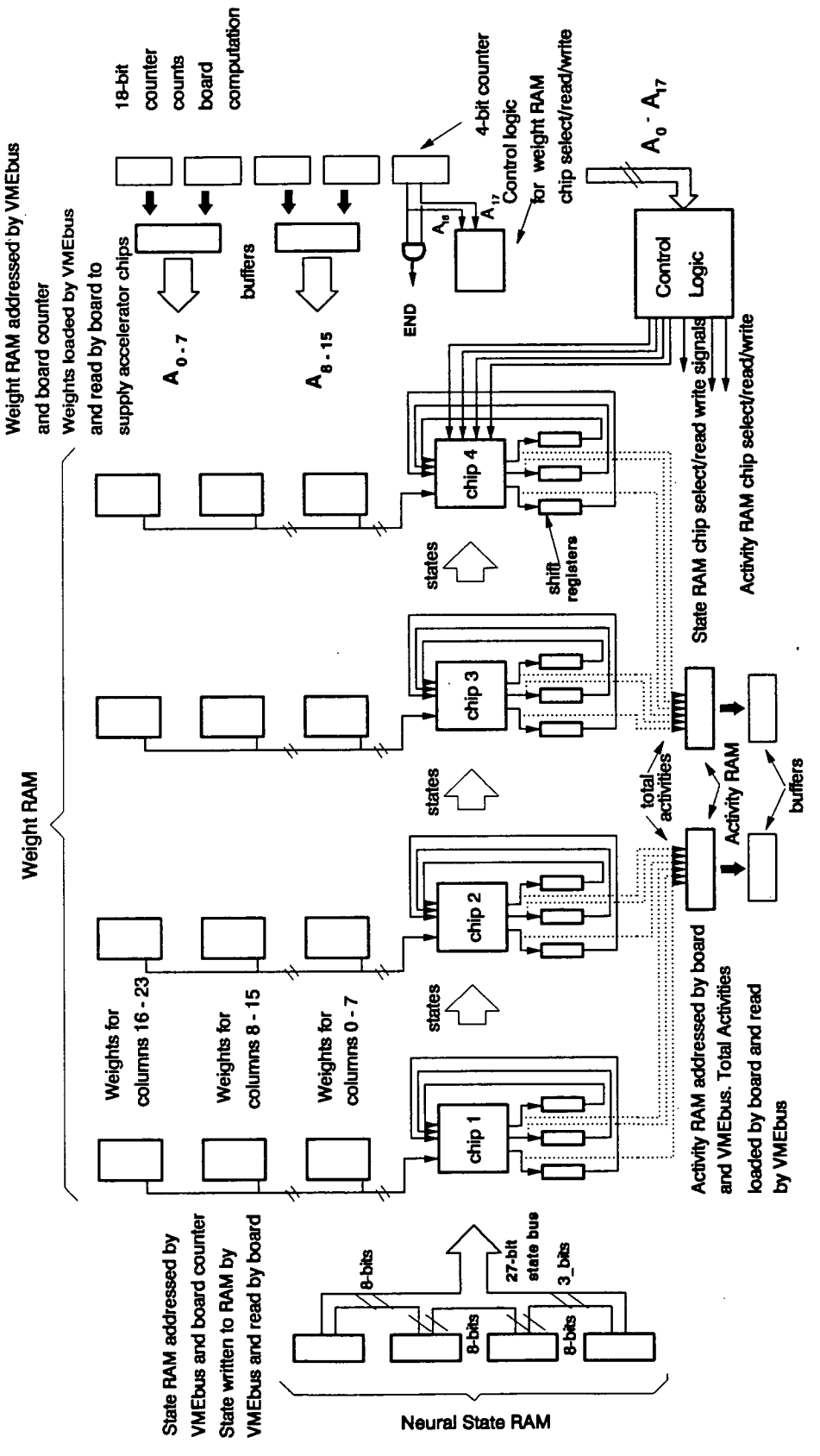


Figure 5.5 Detailed schematic of the neural board

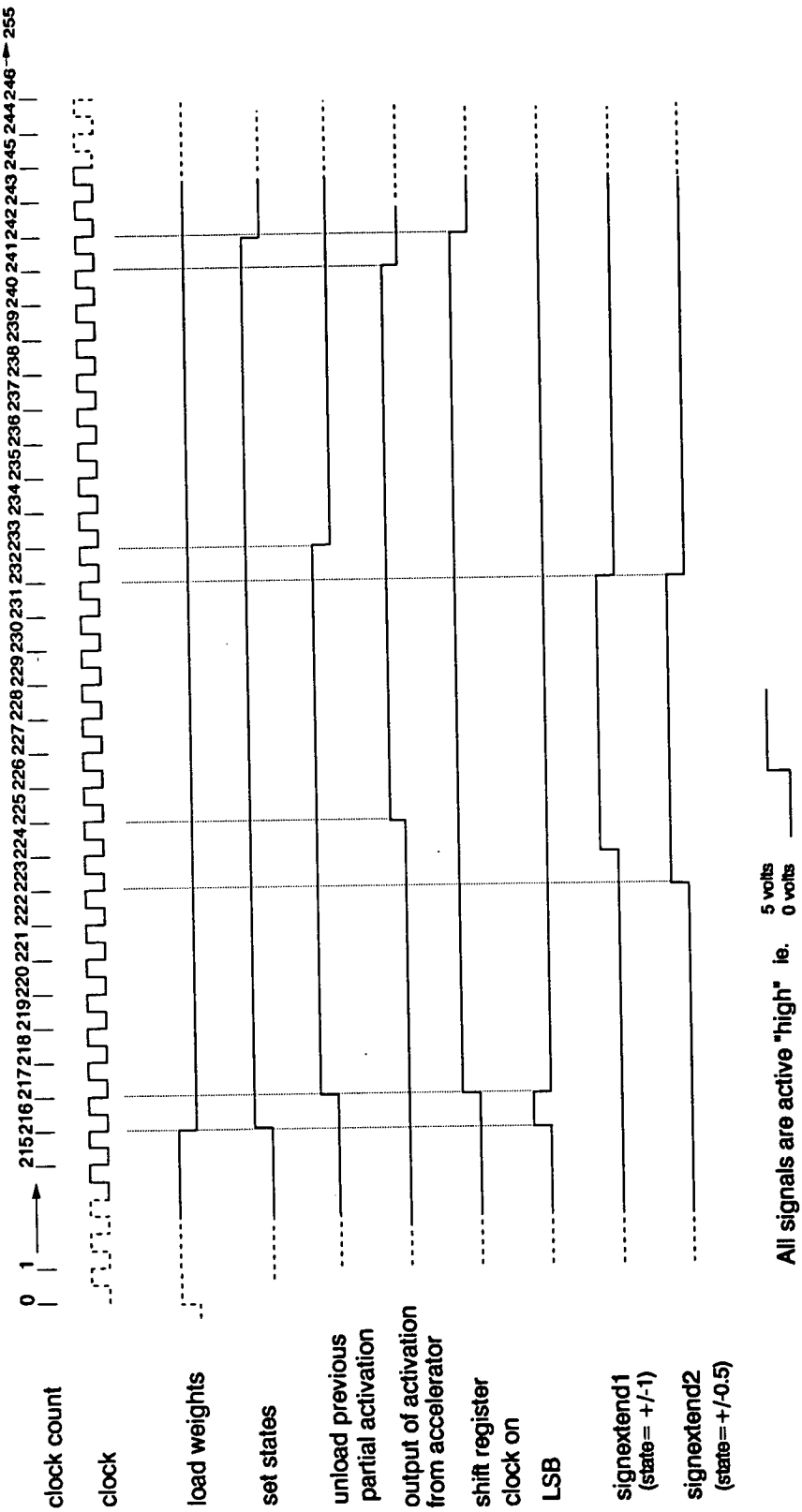


Figure 5.6 Waveform diagram of the board control signals

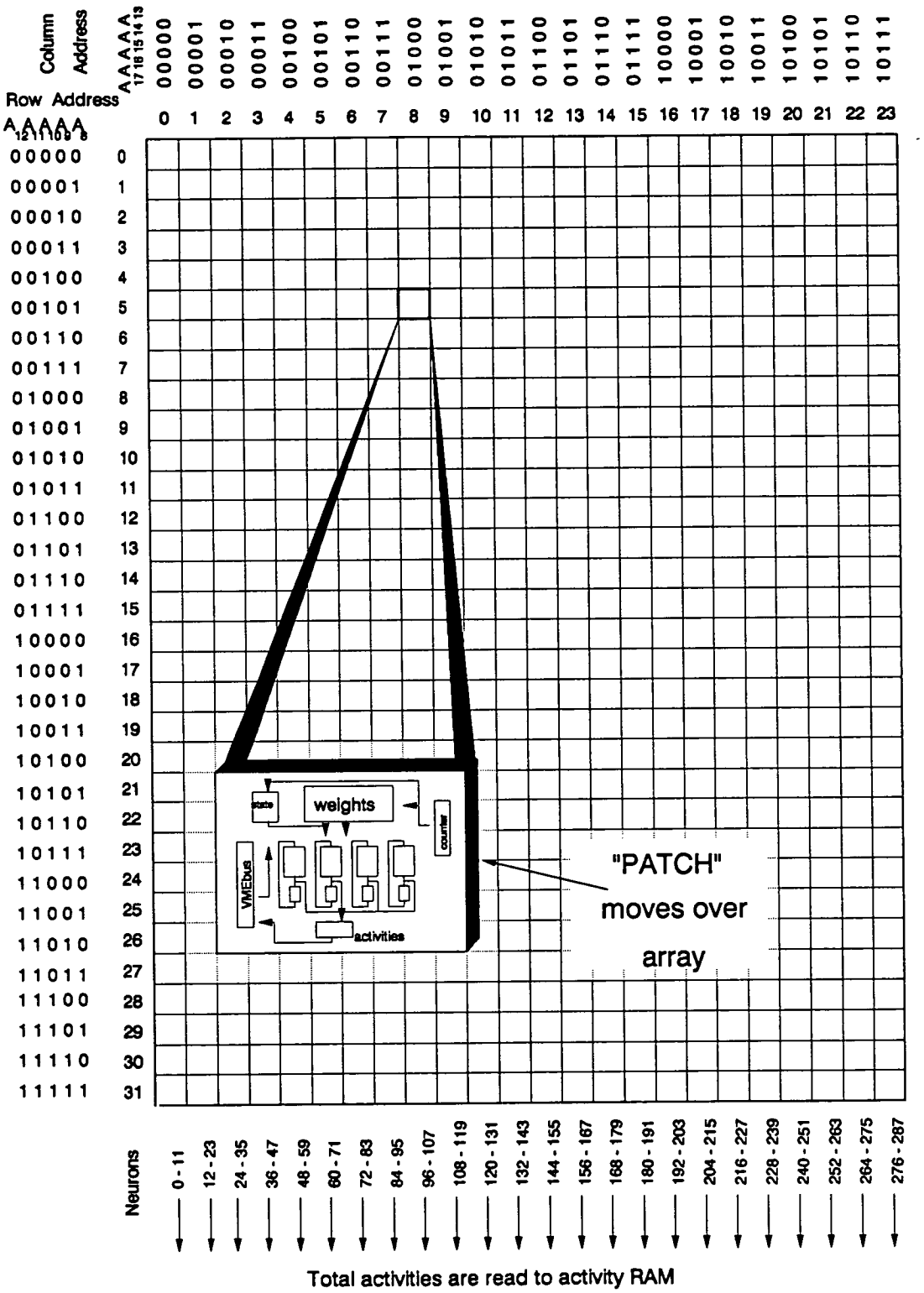


Figure 5.7 Array computation

fitted conveniently into a 288 neuron array and could be counted easily with an 18-bit counter.

With each "patch" iteration down a column, the counter address given by A_{12} - A_8 increments by 1, until it reaches the address $11111_2 = 31_{10}$, at the bottom of a column. This address then signals to the activity RAM to read in the total activity values and to clear the partial activity shift registers ready for the 1st "patch" in the next column.

Each column computes the activity for 12 neurons and is iterated 24 times counted by the address A_{17} - A_{13}. Hence, $10111_2 = 23_{10}$ is the last column in the array. Address A_{17} - A_{13} and A_{12} - A_8 = 1011111111_2 signals the final 12 activities of the array are ready to be loaded to the activity RAM. The next address 1 1 0 0 0 0 0 0 0 0 has A_{17} and A_{16} both equal to 1, which is used to signal to the VMEbus that the board has completed its calculations and is ready to unload its activities to the host Sun. This is the signal "END" shown in figures 5.3 and 5.5.

The maximum RAM speed of 45ns would allow the board to run at a maximum speed of 22.2 MHz. However, with the control circuitry and buffers incorporated in the board level design to support the accelerator chip computation, delays were introduced which allowed the circuit to operate at only 8 MHz. Although this speed is slower than was originally anticipated, it did not greatly affect the array computation time with respect to the speed of operation of the software. This is because the computing and loading of the new neural states takes, by far, the majority of time in a complete board run cycle of loading weights and states via the VMEbus, computing new activities, thresholding the activities to the new states of the network and subsequently calculating the new weight set for board updating.

Figure 5.8 shows a photo of the neural board and figure 5.9 shows the neural board interfaced to the host Sun workstation.

5.4. Software to Control the VMEbus

Software controls the function of the VMEbus and transfer of data between the host Sun and the neural board. The program in Appendix D shows how this is done. The declaration "addr= VME24d32(VME_BASE,VME_SIZE,&fd)" maps the 32 bit data, 24 bit address VMEbus memory area in the host directly to the addressing on the

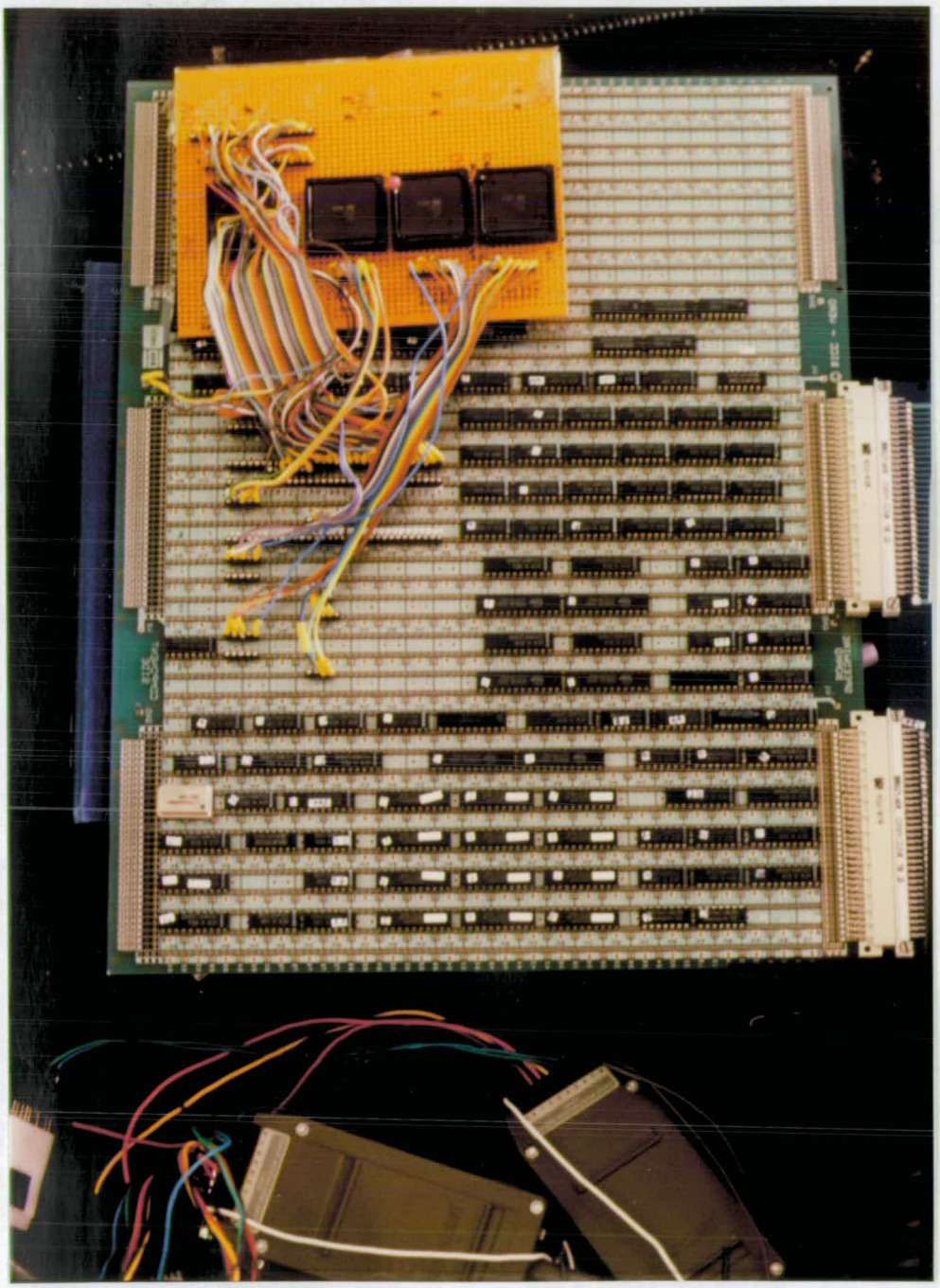


Figure 5.8 Photo of the Neural board

Figure 5.9 Neural board interfaced to the host Sun workstation

board.

The program reads weights from a file "weight_in" and writes them in parallel to the appropriate address locations in the 12 integrated circuits of the weight RAM on the board. It then reads the states from a file "state_in" and writes them to the state RAM. Line 64, "addr[RUN]=0", signals to the board to start running and line 66, "check=addr[RUN]", reads back from the board to check if A_{17} and A_{16} are both at a logic "1". When this condition is satisfied, the neural activities are read back to the host.

At this point the board was verified as operating correctly, with all weights in the input file identical † and the states for each "patch" identical (states were changed between adjacent "patches"). The file "weight_in" was in the form of 3-bit hexadecimal numbers (ie. D_{11} - D_0), the state file was 7-bit hexadecimal numbers (ie. D_{26} - D_0) and the activities were read back as 3-bit hexadecimal numbers. This is the form in which the VMEbus handles data. This level of format of the input and output data is suitable for a low level test of the board. Ideally, the weight file has each weight as an 8-bit two's complement number in a 288 x 288 byte array with indices corresponding to the 288 x 288 synapse array and the state file has each state as +1 or -1 in a 288 bit array corresponding to the 288 neural states. In this form, the data is much easier to handle for neural learning simulations. The activities also need reorganising from 24 blocks of 12 16-bit binary numbers to a decimal number that can be thresholded to one of the 5 neural states. Chapter 6 gives details of this along with simulation results of the board operating as a pattern associator.

† Although all the weights in the input file were identical, the weights in each row of synapses in a "patch" become shifted. During computation the weights are shifted around the shift register. Therefore, row 1 has no shifts, row 2 has 1 shift, row 3 has 2 shifts....., row 8 has 7 shifts and row 9 has 8 shifts (ie. no change) before the weight is involved in the "multiplication". Avoidance of this shifting of weights should have been taken into account during the integrated circuit design.

Chapter 6

Simulations using the Neural Accelerator Board

The testing of the neural board described in Chapter 5 verified that it computed the neural function $\sum_{i,j=0}^{287} T_{ij}V_j$ successfully and loaded the new activities back to the host Sun for thresholding to new neural states. At this stage, identical weights and states were used for each "patch" to make the testing easier. However, this level of testing proved only that the board was loading and unloading data to and from the RAMs and shifting and calculating data on the correct clock cycles. Further software was required, that incorporated a learning algorithm with the board computing $\sum T_{ij}V_j$ to verify that the 5-state activation function hardware implementation could learn and recall patterns. A second software program was required for the same size network as the hardware and the same learning algorithm, but which computed $\sum T_{ij}V_j$ in software, in order that the hardware and software versions of the 5-state activation could be compared.

6.1. Software Requirement for the Neural Board

There are four main requirements for the software to enable simulations using the neural accelerator board.

1. The nine neural states for each "patch" are passed by the VMEbus in a 7-bit hexadecimal word. To make the handling of the simulations more user friendly, the neural states should be written as $+1_{10}$ or -1_{10} for each state. Therefore, software was needed to convert the nine states for each "patch", written as $+1_{10}$ of -1_{10}, into a 7-bit hexadecimal number. For example, if each neural state for a "patch" = -1, the hexadecimal number would be 0x2492492, ie., the least significant number, 2_{16} represents 0010_2, which represents in turn the values for rs2, pm1, kill1, and rs1. Therefore, if the neural state is -1, then rs1=0, kill1=1, pm1=0 and rs2=0. The second least significant number, $9_{16} = 1001_2$, gives the values for kill3, rs3, pm2 and kill2 respectively etc..

2. The weights are calculated in software using floating point numbers. The accelerator chips, however, require that the weights are two's complement numbers. Therefore, each weight must be converted from a floating point number to a two's complement number. Furthermore, the precision of the 8-bit weight allows the weight to be in the range -127 to $+128$ so weights outside this range must be clipped [101] to the permitted maximum and minimum values.

   The weights must also be loaded in the correct order into the accelerator chips. Each accelerator chip is a 3 x 9 array, so for example, the 1st chip in "patch" 1 uses the weights $T_{0,0}-T_{0,8}$, $T_{1,0}-T_{1,8}$ and $T_{2,0}-T_{2,8}$, the 2nd chip in "patch" 1 uses weights $T_{3,0}-T_{3,8}$, $T_{4,0}-T_{4,8}$ and $T_{5,0}-T_{5,8}$ and the 1st chip in "patch" 2 uses weights $T_{0,9}-T_{0,17}$, $T_{1,9}-T_{1,17}$ and $T_{2,9}-T_{2,17}$ etc.. If the new weights are computed in a 288 x 288 array, ie., 1st column is $T_{0,0}-T_{0,287}$, 2nd column is $T_{1,0}-T_{1,287}$ etc., then the appropriate weights must be taken from the array in the correct order for each "patch" and loaded bit-serially into the accelerator chips.

3. The total activities are also 16-bit two's complement numbers, which are loaded to the host Sun in that form. The software is required to convert these to floating point numbers, which can then be thresholded according to the 5-state activation function to give the new states of the network.

4. The final part of the software is a learning algorithm that uses the new states of the network to update the weights. Details of the learning algorithm are given in the next section.

6.2. Neural Accelerator Board as a Pattern Associator

The Pattern Associator Model

The simulations run to verify the 5-state hardware learning capabilities used the network configured as a *pattern associator*. This is where a set of input to the units will cause a certain pattern on a set of outputs from the units, whenever the input is applied. Pattern associators can be implemented as a set of units causing a pattern of activation over another set of units without any intervening units. They have been widely used in distributed memory modelling with the *Hebb rule* and the *delta rule*. A pattern associator has a set of input units connected to a set of output units by a single

layer of modifiable connections (weights) that are suitable for training with the Hebb and the delta rule. It would, for example, be capable of associating a pattern of activation of one set of units corresponding to the appearance of an object with a pattern on another set corresponding to the aroma of the object, so that when an object is presented visually, causing its visual pattern to become active, the model produces the pattern corresponding to its aroma.

Single layer pattern associators, have several properties that make them attractive as models of learning and memory. They can learn to act as content addressable memories and can generalise the responses they make to novel inputs. Hence, if a new pattern on the input units to the network is similar to one of the old ones, it will tend to have similar effects and as learning of the interconnections occurs in small increments, similar patterns reinforce the strengths of the links they share in common with other patterns. Therefore, if the same pair of patterns is presented again and again, but each time a small percentage of random noise is added to each pair, the system will automatically learn to associate the similarities of the two patterns and will learn to ignore the noise. Effectively, an average of the two patterns will be stored with the slight variations removed. Conversely, if the network is presented with completely uncorrelated patterns, they do not interact with each other. Another property of pattern associators is their pattern retrieval performance degrades gracefully with damage and noise, in that they do not require a perfect copy of the input to produce the correct output, although its strength will be weaker in this case.

The Delta Rule

The learning algorithm implemented in the simulations was the *delta rule*. The delta rule involves the presentation of a set of input and target output patterns. The network uses the input pattern to produce its own output pattern and then compares this to the *desired output*, or *target*. It is the difference between the *target* pattern and the obtained pattern that drives learning. If there is no difference, no learning takes place, otherwise the weights are changed to reduce the difference. The rule for changing weights following the presentation of input/output pair n is given by:-

$$\Delta_n w_{ij} = \eta(t_{ni} - o_{ni})v_{nj} \tag{6.1}$$

where η is the learning rate, t_{ni} is the target the the ith component of the output

pattern for pattern n, o_{ni} is the ith element of the actual output pattern produced by the presentation of input pattern n, v_{nj} is the jth element of the input pattern and $\Delta_n w_{ij}$ is the change to be made to the weight from the jth to the ith unit following presentation of pattern n.

The delta rule also requires that the *pattern sum of squares*, *pss*, is measured [102]. The pattern sum of squares is the sum of the squared error over all output units, where the error for each output unit is the difference between the target and the obtained state of the neuron, ie., for an N neuron network:-

$$pss = \sum_{n,i=0}^{N} (t_{ni} - o_{ni})^2 \tag{6.2}$$

This quantity is calculated for each pattern processed to give the *total sum of squares*, *tss*, where:-

$$tss = \sum_{n=0}^{N} pss_n \tag{6.3}$$

The total measure of all patterns, *tss*, gives the error between the target and the actual output states. Therefore, on the first iteration, *tss* will be large ($tss >> 0$). As the connections between the input and the output are learned, $tss \rightarrow 0$. When $tss = 0$, the actual output equals the target output and the pattern associator can be deemed to have learned the mapping between the input and the output.

The software for implementing the delta rule is straightforward in that it only requires input and target pattern pairs to be read from files, calculation of the actual output from the network activation and calculation of $\Delta_n w_{ij}$ from equation 6.1. Optimum values for the learning rate, η, and "Temperature", T, for the network were found by trial and error using a 36 neuron software model with the 5-state activation function. The combination of T and η that gave the least number of iterations to learn sets of patterns was used. The learning rate is a constant of proportionality that dictates how fast the network will learn, and the "Temperature" controls the sharpness of the transition of the 5-state activation function between the states -1 to $+1$.

6.2.1. Performance of the Accelerator Board

Three programs were used to obtain results to show the performance of the 5-state activation function in hardware. The first, given in appendix E, incorporates running

the neural accelerator board in a pattern associator network. The main four sections described in section 6.1 can be seen. The function "boardrun", declared on line 61 and called on line 140, allows the board to compute. The length of computation time taken by the board to compute can be measured by using the *prof* command in unix. *prof* (display - profile data) produces an execution profile of the program, which gives the number of milliseconds spent on a call to a function.

The second program in Appendix F has the same number of synapses, weight size and activation function as the first, but executes the neural activation computation in software with the weights and activations as floating point numbers. The function "Actsum", declared on line 19 and called on line 65, computes in software the equivalent task of the neural board. Therefore, by using the display - profile data command, the run time for the function "Actsum" could be found. The values given for "boardrun" and "Actsum" showed the comparative speeds of the hardware and the software for the computation $\sum_{i,j=0}^{n-1} T_{ij} V_j$ for an n-neuron network.

The third program is identical to the second, but used a sigmoidal activation function instead of the 5-state activation function. The three programs allowed a comparison of performance between the 5-state hardware, the 5-state software and the sigmoid software activation functions. Comparisons were done on the time to compute $\sum_{i,j=0}^{n-1} T_{ij} V_j$ and the number of iterations each took to learn the requisite sets of input and target patterns. The expected results from these comparisons should show the sigmoid activation function to learn patterns using the least number of iterations of the three programms. The 5-state activation function in hardware and software should learn with the same number of iterations.

The size of the network chosen to run the simulations was 36 x 36 synapses (36 neurons). Although the board computes the activation for a 288 neuron network, simulation to judge the performance of the three networks alone can be run with equivalent results on a smaller network. The main reasons for using a smaller network were:

1. Software simulations of the 288 neuron network were intolerably lengthy.
2. Formatting the input files input for a 288 neuron network was also a long and

tedious task.

3. The ordering and loading of the weights of the 288 neuron network was slow on the host Sun 3/110. The Sun's own memory was not large enough to hold the array sizes required for the weights and therefore the Sun had to access the file server for extra memory. The process of swapping data to and from the file server slows down the iteration time of the hardware simulations. This slowness could be alleviated by the Sun having more memory of its own.

Simulation Procedure

The simulation procedure for each of the 5-state hardware, 5-state software and sigmoid software activation function networks was as follows:-

1. Six sets of 20 random input and target pair patterns were generated.
2. Each of the networks learned each set of patterns in turn.
3. The number of iterations taken by each network to learn each set of patterns was noted.
4. The time taken for each network to compute $\sum_{i,j=0}^{35} T_{ij} V_j$ was taken for each set of patterns (to find the average time over 6 sets).
5. For the hardware network only, noise was added to the input patterns and the noise on the corresponding degraded output patterns was measured and compared to the target.

For each network simulation:

Each input and target pattern pair had 36 elements set either to $+1$ or -1,

"Temperature", $T = 50$,

Learning rate, $\eta = 5$.

6.2.2. Results

The results comparing the performance of the 5-state hardware, 5-state software and the sigmoid software activation functions are given in table 6.1. The results give the times and iterations for each set of 20 patterns and the average values over the six sets. A discussion of the results falls into 3 categories.

	5-state Hardware		5-state Software		Sigmoid Software	
number of pattern set	time for $\sum_{i,j=0}^{287} T_{ij}V_j$ in ms	number of iterations	time for $\sum_{i,j=0}^{35} T_{ij}V_j$ in ms	number of iterations	time for $\sum_{i,j=0}^{35} T_{ij}V_j$ in ms	number of of iterations
1	25.11	7	33.1	5	34.5	4
2	24.00	11	34.0	7	34.43	7
3	25.26	10	34.67	6	33.2	5
4	25.00	7	33.75	4	33.75	4
5	25.00	8	33.98	6	33.6	5
6	24.17	4	33.75	4	33.67	3
average over 6 sets	24.76	7.8	33.89	5.3	33.86	4.7

Table 6.1 Results comparing the performance of the 5-state hardware, 5-state software and sigmoid software activation functions

1. Activity Computation Times

The results give the average time for the sigmoid and 5-state software activation functions to compute $\sum_{i,j=0}^{35} T_{ij}V_j$ to be 33.86ms and 33.89ms. The results for the hardware give the time to compute $\sum_{i,j=0}^{287} T_{ij}V_j$, since the design of the board is such that it will run for only that size of network. A 36 neuron network is obtained by using a 36 × 36 synapse array and setting the unused weights and states to 0. From the board run times of the 288 neuron network, the times for a 36 neuron network to run can be calculated, which can then be compared to the sigmoid and 5-state software results. Table 6.2 gives the measured time to compute the activity for 288 neurons at different clock frequencies. The results agree with the theoretical ones calculated for the board run time.

The calculation is as follows:

Time to compute the activity for 288 neurons at clock frequency, f.

$$= \text{clock cycles per patch} \times \text{no. of columns} \times \text{no. of rows} \times \frac{1}{f}$$

$$= 256 \times 32 \times 24 \times \frac{1}{f}$$

Neural board run times		
frequency in MHz	actual time in ms	theoretical time in ms
6.0	33.11	32.77
7.5	26.82	26.21
8.0	24.86	24.58
20.0	-	9.83

Table 6.2 Time to compute the activity for 288 neurons at varying clock frequencies

$$= \frac{196608}{f}$$

The time taken to compute the activity for 36 neurons

$$= \text{clock cycles per patch} \times \text{no. of columns} \times \text{no. of rows} \times \frac{1}{f}$$

$$= 256 \times 3 \times 4 \times \frac{1}{f}$$

$$= \frac{3072}{f}$$

Hence, if frequency = 8MHz, time taken $= \frac{3072}{8 \times 10^6}$

$= 0.38\text{ms}$

Therefore, theoretical time to compute a 36 neuron network $= 0.38\text{ms}$

From table 6.1, actual time for a 36 neuron network $= \frac{3072}{196608} \times 24.76$

$= \frac{1}{64} \times 24.76$

$= 0.39\text{ms}$

This shows an 87 times acceleration using the hardware. Although the board-run times should be the same, the prof command gives varying times for the execution of the "boardrun" function. This is due to other processes running in the Sun's central processing unit, that will vary the run times of the function.

These results are accurate enough to show that the board run time at 20MHz would be within 2.5% of the theoretical time. This is the maximum difference

between the actual and theoretical times in table 6.2. A frequency of 20MHz would have given a 220 times acceleration in hardware. A frequency of 20MHz was not obtainable owing to timing delays occurring in the board level design. These slowed down the maximum clock frequency to 8MHz.

2. Comparison of Iterations

The sigmoid activation function required the smallest number of iterations to learn the sets of patterns, followed closely by the 5-state software activation function. The hardware 5-state activation function takes on average 2.5 more iterations than the equivalent software. There are two reasons for this.

The first reason is due to floating point weights being truncated to integer numbers in the conversion to two's complement numbers. For example if a floating point weight = 7.75, this becomes truncated to 7 (integer), giving a 10.7% loss in accuracy. The larger the modulus of the weight the smaller the loss in accuracy.

The second reason is a small change in the value of odd number weights if they are right-shifted to be multiplied by 0.5. For example,

	00000111_2	$= 7_{10}$	
right-shift by 1-bit	00000011_2	$= 3_{10}$	(instead of 3.5)
		= 14.3% change	
	01100111_2	$= 103_{10}$	
right-shift by 1-bit	00110011_2	$= 51_{10}$	(instead of 51.5)
		= 1.0% change	

This change effects small weights more than the large ones, as shown in the example, a weight of 3.5 is rounded to 3.

This implies that the overall effect of a truncated, right-shifted weight will have a considerable loss in accuracy. From the examples above, a weight of 7.75 is truncated and right-shifted to 3, where its true value should be 3.875, this represents a 29% error in the final weight value.

3. Degradation of Input

To observe the robustness of the neural board in a pattern associator network under degraded input patterns, the number of degraded elements per output pattern was measured, along with the maximum deviation of any element in the output from the equivalent element in the target. The results for one set of patterns are given in table 6.3. The region where the majority of the numbers of degraded elements per pattern fall is also given.

Number of bits changed per input pattern	Percent noise %	Max. no. of bits degraded in output per pattern	Majority of degraded bits per pattern	Maximum deviation per pattern
0	0	0	0	
1	2.8	0-11	0-4	0.5
2	5.6	0-16	2-8	1.0
3	8.3	2-18	4-12	1.5
4	11.1	2-19	5-14	2.0
5	13.8	2-20	6-15	2.0

Table 6.3 Table showing degradation in the output patterns with degradation in the input patterns

The results show that for 1 element change per pattern in the input, the majority of the changes in the output patterns are between 0 and 4 elements and the maximum deviation of any output element from the target was 0.5. "Change" means either $+1$ becomes -1 or -1 becomes $+1$. As the noise increases in the input patterns, the number of elements degraded in the output patterns and the maximum deviation of output elements from the target increases correspondingly. Therefore, the strength of the output pattern becomes weaker with the rise in noise in the input. The results vary slightly if different elements in the patterns are changed or if different sets of patterns are used.

6.3. Conclusions

The neural board operated successfully as a pattern associator network and accelerated the speed of the activity calculation by 87 times at a derated frequency

of 8MHz over an equivalent software calculation. The hardware took, on average, 2.5 more iterations (ie. 47% more iterations) to learn the weight set for a 36 neuron network than the software. This was owing to the loss of resolution in the weights when they were converted from a floating point number to a two's complement number. Finally, the pattern retrieval performance degraded gradually with increasing noise added to the input patterns.

Chapter 7

Conclusions and Discussion

This final chapter draws together the results from the previous sections regarding the design, construction and demonstration of the chip set and accelerator board. In particular, the successes and the shortcomings of the work are highlighted and recommendations are made for future improvements.

7.1. Conclusions about the Accelerator Board

The 5-state activation function has been shown to operate correctly as implemented by the accelerator chips. The neural board that incorporates the accelerator chips has also proved that it can be used as a hardware accelerator in a simple pattern associator network for the learning and recall of random patterns. The neural board accelerated the calculation $\sum_{i,j=0}^{n-1} T_{ij} V_j$ by amost two orders of magnitude over that possible in equivalent software simulations. The board also recalled patterns with the level of degradation in the output following the level of degradation in the input. A minor drawback resulted from the inaccuracy of changing floating point weights to two's complement weights, which is caused by the computer's software rather than the board implementation. The inaccuracy was incurred by the "multiplication" of a binary number by 2 (right-shifting the number by one bit), as in the examples in Chapter 6, section 6.2.2, is a peculiarity of the 5-state activation function. Integer arithmetic would have avoided the truncation from floating point to integer numbers, decreasing the total error, but an inaccuracy would still occur in right-shifting an odd two's complement number. This, for example, would be 14% for the number 111_2 (7_{10}) and 1% for 110011_2 (51_{10}).

Even though the board accelerated the calculation of the activation, the design of both the accelerator chip and the hardware support for the chip is now seen to be non-optimal, so that the advantage of the increased speed was lost by the time taken to run the software to support the accelerator board. The following successes

and shortcomings summarize the operation of the neural board as a hardware accelerator.

Successes

1. The neural board accelerated the calculation time of the activity to 87 times that of the software.
2. There was no marked loss in performance using the hardware 5-state activation function compared to equivalent software models.
3. The neural board can be used with any learning algorithm.
4. Any size of neural network can be used up to a maximum of 288 neurons.

Shortcomings

1. The inaccuracy occurring in small weights due to the conversion of floating point numbers to two's complement numbers and in the right-shifting of odd two's complement numbers.
2. The time taken for the software to run the neural board, which is increased by the the calculation and reordering of the weights.
3. The run time of the neural board is always that for a 288 neuron network regardless of the the actual network size.
4. The maximum operating frequency of the board is 8 MHz, although the accelerator chips should operate upto 20MHz.

The major disadvantage is the speed of the software. This is incurred by the reordering of the weights, so that they are input in the correct order to the neural board weight RAM. Some redesign to the synapse array at the chip level would help minimise the software required. Extra circuitry in the integrated circuit would also reduce the quantity of support hardware required at the board level. This would increase the overall speed of the neural board. Details of this are given in the next two sections.

7.2. Improvements to the Accelerator Chip Design

There are three areas in the design of the accelerator chip where improvements could be made to increase the speed and improve the overall efficiency of the

neural accelerator board. These are: changes to the array structure; alterations to the weight shift register in the synapse to reduce the software required for weight ordering and the addition of extra circuitry on the integrated circuit (separate from the synapse array to reduce the complexity and quantity of hardware required for the board level design).

1. Array Structure

The simplest design of the neural board and the software support required to run it, would be achieved if an accelerator chip contained only one column of synapses. For example, if 30 synapses formed part of a column of synapses for one neuron, and if 5 chips were cascaded, a 30×5 synapse array would form a "patch", allowing the ordering and loading of weights to be much simpler. However, this structure loses the parallelism made possible by a VLSI design.

There are not likely to be optimal array dimensions that will enable a more efficient operation of the "patch" in the paging architecture. The best size of the array will to some extent depend on the number of chips that are to be cascaded to form a "patch". The "patch" size, in turn, will depend on the number of neurons to be implemented in the network. It was fortunate with the present 9×3 accelerator chip, that one "patch" could be computed in 256 (2^7) clock cycles, with only the last 12 clock cycles unused. These 12 clock cycles per "patch" represent an overall redundancy of 4.5% of the total computation time. The design of the integrated circuit would be more efficient if all the clock cycles were used during the board computation.

A useful addition to the synaptic weight as it stands, would be to provide a synaptic weight input pin per column of synapses. This would require 2 extra pins, which does not create a significant problem, as only 53 of the 68 pins in the package are used in the design. This alteration would trade off some of the advantages of a bit-serial approach, in order that the loading of the neural weights from the RAMs would be faster and would also necessitate less software for the ordering of weights.

2. Alterations to the synaptic weight storage

As described in the footnote in chapter 5, section 5.4, the synaptic weights are shifted around the weight shift registers while computation proceeds down the synaptic columns. This shifting was not taken into account during the integrated circuit design and therefore software was used to ensure the bits in each weight were in the correct order, ie., while the weights were shifted during computation, the LSB of the weight about to be multiplied by the neural state was correctly in the LSB of the shift register. This could be overcome by simply tapping off the LSB of the weight from the bit in the shift register where it has been shifted to, as drawn in figure 7.1, instead of tapping off each weight at the LSB of the shift register.

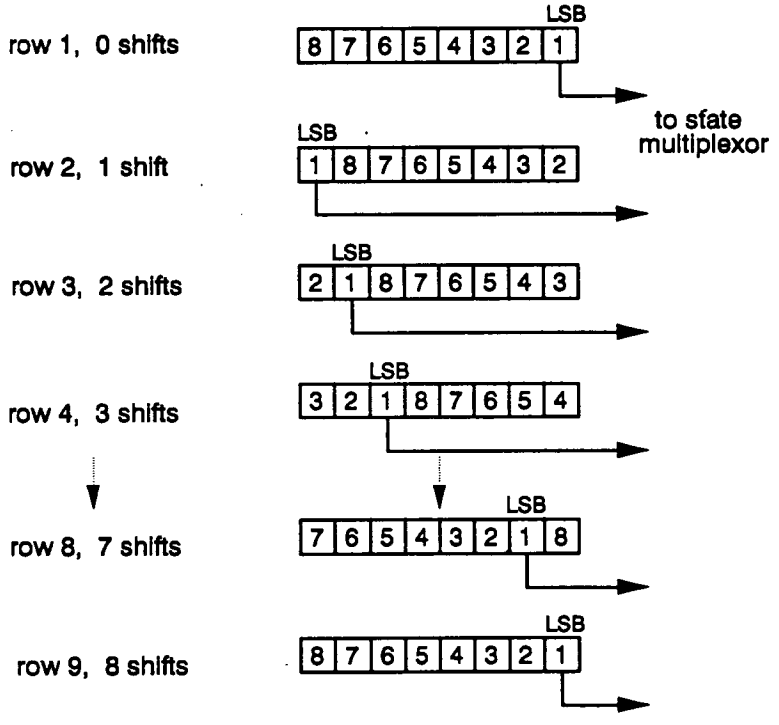


Figure 7.1 Alterations to the synaptic weight shift register

3. Extra Circuitry

The extra circuitry that could be implemented in the integrated circuit design, but separate from the synapse array, would reduce the number of standard small scale integrated (SSI) circuits required for the support hardware for the accelerator chips. The SSI circuits form, effectively, "glue" logic that enables the functioning of the accelerator chips in a neural network system. The circuitry falls into two

sections. The first is the inclusion of a 16 bit shift register for each synaptic column to hold the partial activation of the neuron before it is required for the adjacent "patch" computation in the column. This would increase the area of silicon required, but it would reduce by 24, the number of SSI circuits in the support hardware. One extra pin on the accelerator chip would be used to control the shifting of the registers.

The second section of extra circuitry would be the addition of the control circuitry for the signals load, lsb, sel and se2. At present, the signals are derived by connecting NAND, NOR, AND and OR gates to give the desired signal. However, each gate incurs a delay and with several gates wired together in series, delays became large enough to miss the change time allowed to ensure steady data for the single phase clocking scheme, ie., the data must be held steady on the low to high transition of the clock. This introduces timing problems in the board design and provides one of the reasons for the slower than anticipated operating speed for the neural board. The inclusion of the control circuitry would require 8 extra input pins for the address lines A_0–A_7, but the present input pins for the signals load, lsb, sel and se2 would no longer be required (however, it is recommended that the signal are connected to test pins so the correct timing of the signals can be checked).

It is estimated, based on the fully custom 3μm CMOS integrated circuit design, that the partial activation shift registers, the control circuitry and the 3×9 synapse array would take up approximately double the silicon area that is presently used, which is 5.7mm^2, for the 3×9 synapse array alone. A 1μm design would, therefore, reduce this area by a factor of 9. Nine extra pins would be needed, but there are already fifteen pins unused on the package. On the current board design, 43 of the 110 SSI circuits used would no longer be required, reducing the total by 39%.

7.3. Improvements to the Neural Board Design

The neural board design and size could be considerably reduced by including the shift registers and control circuitry on the accelerator chip, thus leaving only RAMs and buffers in the support hardware. This is the ideal solution, but the

control circuitry would add complexity to the integrated circuit design. An alternative to this would be to use a PLA (programmable logic array) for the control, which could then also include the chip select and read/write control logic for the weight, state and activity RAMs.

Larger RAMs than the present weight RAMs (64k each) are now more readily available than when the board was designed. Each accelerator chip requires 0.2 megabits of RAM, therefore a RAM of this size to supply each accelerator chip could be used instead of the the present system of multiplexing 3 RAMs for each chip. Also, reorganisation of the VMEbus data lines to the weight RAMs would make the design more efficient and simplify the software.

7.4 Concluding Remarks

The design stages of the neural accelerator board were done separately in that the accelerator chip design took place without anticipating a paging architecture board using a VMEbus. The board design took place without full knowledge of how to implement a learning algorithm program using the neural board as a neural hardware accelerator. With hindsight, a substantially revised design for the integrated circuit, to remove the stress from the board and software development can be proposed. This would be to keep the present (or approximately the same) array size (or a slightly larger array if a smaller design feature size were available) with alterations to improve the neural board efficiency and speed. These alterations are summarised in the following points:-

1. Inclusion of partial activation shift registers in the integrated circuit.
2. Inclusion of the control logic in the integrated circuit.
3. One weight input pin per synapse column in the integrated circuit.
4. Alterations to the synapse weight shift register to avoid weight ordering in software.
5. Larger weight RAMs at the board level design to avoid multiplexing.
6. Better organisation of the VMEbus data lines to the weight RAMs
7. A large reduction in software after implementation of the above 6 points.

A conclusion of this nature is inevitable when the length of time for the VLSI design, fabrication, testing and board building, testing and implementing in a learning algorithm is so long. Effectively, major decisions about the chip design had to be made three years ago and their consequences endured through the later board design stages.

References

1. Young J. Z., *A Model of the Brain,* Oxford Clarendon Press, 1964.
2. Thompson R. F., *The Brain - An Introduction to Neuroscience,* W. H. freemann & Co, New York, 1985.
3. Shepherd G. M., *Synaptic Organisation of the Brain,* Oxford University Press, New York, 1974.
4. Mead C., *Analog VLSI and Neural Systems,* Addison-Wesley, 1989.
5. Pavlov I. P., *Conditioned Reflexes: An Investigation of the Pyschological Activity of the Cerebral Cortex,* Oxford University Press, London, 1927.
6. Blakemore C., *Mechanics of the Mind,* Cambridge University Press, Cambridge, New York, 1977.
7. Rashevsky N., *Mathematical Biophysics: Physicomathematical Foundations of Biology,* University of Chicago Press, 1938.
8. Pillsbury W. B., *A History of Psychology,* G. Allen & Unwin, London, 1929.
9. Luria A. R., *The Working Brain,* Allen Lane, London, 1973.
10. Jackson J. H., *On Localistaion. In Selected Writings,* 2, Basic Books, New York, 1869.
11. Luria A. R., *Higher Cortical Functions in Man,* Tavistock Publications, London , 1966.
12. Poincare H., *Foundations of Science: Science and Hypothesis,* Science Press, New York, 1913.
13. Thorndike E. L., *Selected Writings from a Connectionists Psychology,* Greenwood Press, New York, 1949.
14. Lashley K. S., *Brain Mechanisms and Intelligence,* University of Chicago Press, Chicago, Ill, 1929.
15. J. Orbach, *The Neuropsychology after Lashley,* Erlbaum, New Jersey, 1982.
16. Lashley K. S., "In Search of the Engram," in *Society of Experimental Biology Symposium No 4: Psychological Mechanisms in Animal Behaviour*, pp. 478 - 505, Cambridge University Press, London, 1950.

17. Hebb D. O., *The Organisation of Behaviour,* Wiley, New York, 1949.

18. Hawkins J. K., "Self-Organising Systems - A Review and Commentary," *Proceedings of the IRE*, pp. 31 - 48, January, 1961.

19. Rumelhart D. E. and J. L. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition,* 1, MIT Press, Cambridge, Massachusetts, 1986.

20. McCulloch W. S. and W. Pitts, "A Logical Calculus of the Ideas Imminent in Nervous Activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 115 - 133, 1943.

21. Roy A. E., "On a Method of Storing Information," *Bulletin of Mathematical Biophysics*, vol. 22, pp. 139 - 168, University of Chicago Press, Chicago, 1960.

22. Roy A. E., "On a Method of Storing Information II. A Further Study of Model Properties.," *Bulletin of Mathematical Biophysics*, vol. 24, 1962.

23. Shimbel A. and Rapoport A., "Approach to the Theory of the Central Nervous System," *Bulletin of Mathematical Biophysics*, vol. 10, pp. 41 - 55, 1948.

24. Minsky M., *Neural Nets and the Brain Model Problem,* Unpublished Doctoral Dissertation, Princeton University, 1954.

25. Rosenblatt F., *Principles of Neurodynamics - Perceptrons and the Theory of Brain Mechanisms,* Sparten, New York, 1962.

26. Minsky M. and S. Papert, *Perceptrons,* MIT Press, Cambridge, Masachusettes, 1969.

27. Sutton R. S. and Barto A. G., "Toward a Modern Theory of Adaptive Networks: Expectation and Prediction," *Psychological Review*, vol. 88, pp. 135 - 170, 1981.

28. Widrow G. and Hoff M. E., "Adaptive Switching Circuits," *IRE, Western Electronic Show and Convention, Convevtion Record*, vol. Part 4, pp. 96 - 104, 1960.

29. Hopfield J. J., "Neural Networks and Physical Systems with Emergent

Collective Computational Abilities," *Proc of Nat Acad Sci*, vol. 81, pp. 3088 - 3092, USA, 1982.

30. Wallace D. J., "Memory and Learning in a Class of Neural Network Models," *Proc Workshop Lattice Gauge Theory: A Challenge in Large Scale Computing*, pp. 313 - 331, 1985.

31. Von der Malsberg C., "Self - Organizing of Orientation sensitive cells in the Striate Cortex ," *Cybernetics*, vol. 14, pp. 85 - 100, 1973.

32. Fukushima K., "Cognitron: A Self-Organizing Multilayered Neural Network," *Biological Cybernetics*, vol. 20, pp. 121 - 136, 1975.

33. Grossberg S., "Adaptive Pattern Classification and Universal Recoding: Part 1. Parallel development and Coding of Neural Feature Detectors," *Biological Cybernetics*, vol. 23, pp. 121 - 134, 1976.

34. Lippmann R. P., "An Introduction to Computing with Neural Nets," *IEEE ASSP Magazine*, pp. 4 - 22, April, 1987.

35. Carpenter G. A. and S. Grossberg, "Neural Dynamics of Category Learning and Recognition: Attention Memory Consolidation and Amnesia," *AAAS Symposium Series*, 1986.

36. Grossberg S., "Some Physiological and Biochemical Consequences of Psychological Postulates," in *Proc. Natl. Acad. Sci. USA*, vol. 60, pp. 758 - 765, 1968.

37. Anderson J. A., "A Theory for the Recognition of Items from Short Memorised Lists," *Psychological Review*, vol. 80, pp. 417 - 438, 1973.

38. Anderson J. A., "Neural Models with Cognitive Implications," in *Basic Processes in Reading Perception and Comprehension*, ed. D. LaBerge and S. J. Samuals, pp. 27 - 90, Erlbaum, 1977.

39. Willshaw D. J., "Holography, Associative memory and Inductive Generalisation," in *Parallel Models of Associative Memory*, pp. 83 - 104, Erlbaum, New Jersey, 1981.

40. Kohonen T., *Self - Organization and Associative Memory*, Springer - Verlag, Berlin, 1984.

41. Hinton G. E., T. J. Sejnowski, and D. H. Ackley, *Boltzmann Machines: Constraint Satisfaction Networks that Learn*, Tech. Rep. No. CMU-CS_84_119, Carnegie - Mellon University, Department of Computer Science, Pittsberg, PA, 1984.

42. Lippmann R. P., B. Gold, and M. L. Malpass , *A Comparison of Hamming and Hopfield Neural Nets for Pattern Classification*, MIT Lincoln Lab Technical Report TR-769 to be published.

43. Wallace D. J., "Spin Glass Models of Neural Memory; Size Dependence of Model Properties," in *Proc Conf on Advances in Lattice Theory, Tallahassee*, April 1985.

44. Pomerleau D. A., Gusciora G. L., and Touretzky D. S. et al, "Neural Network Simulation at Warp Speed: How we got 17 Million Connections per second," in *IEEE International Conf on Neural Networks, San Diego*, pp. 143 - 150, July, 1988.

45. Penz P. A. and Wiggins R., "Digital Signal Processor Accelerators for Neural Network Simulations," in *AIP Conf Proc, Neural Networks for Computing, Snowbird*, ed. J. Denker, vol. 151, pp. 345 - 355, New York, 1986.

46. Aleksander I. and Morton H., *Introduction to Neurocomputing*, MIT Pres, North Oxford, USA, UK, 1989.

47. Aleksander I., "Exploding the Engineering Bottleneck in Neural Computing," in *2nd European Seminar on Neural Computing: the Commercial Prospects*, London, Feb., 1989.

48. Garth S., *Associative Network Solution Chipset*, Internal Report, Engineering Department, Cambridge University.

49. Garth S., "A Chipset for High Speed Simulation of Neural Network Systems," in *Proc IEEE 1st Int Conf on Neural Networks*, vol. 3, pp. 443 - 452, San Diego, July, 1987.

50. Garth S., "An Integrated System for Neural Network Simulations," in *Proc Conf on VLSI for AI, Oxford*, ed. W. R. Moore, Kluwer Academic, USA, 1988.

51. Pacheco M., Bavan S., and Lee M et al, "A Simple VLSI Architecture for Neurocomputing," in *International Neural Network Society 1st Annual Meeting*, ed. T. Kohonen, vol. 1, p. 398, Pergamon Press, Sept., 1988.

52. Potu B. and Ramamoorthy P. A., "A Fully Digital Architecture for Multi-state Neural Networks," in *International Neural Network Society 1st Annual Meeting*, ed. T. Kohonen, vol. 1, p. 400, Pergamon Press, Sept., 1988.

53. Weinfeld M., "A Fully Digital Integrated CMOS Hopfield Network including the Learning Algorithm," in *Proc Conf on VLSI for AI, Oxford*, ed. W. R. Moore, pp. 169 - 178, Kluwer Academic, USA, 1988.

54. Faure B. and Mazare G., "A VLSI Implementation of Multilayered Neural Networks," in *VLSI for AI, Oxford*, ed. W. R. Moore, pp. 159 - 168, Kluwer Academic, USA, 1988.

55. Blayo F. and Hurat P., "A VLSI Systolic Array dedicated to Hopfield Neural Network," in *VLSI for AI, Oxford*, ed. W. R. Moore, pp. 255 - 264, Kluwer Academic, USA, 1988.

56. Ae T. and Aibara R., "A Neural Network for 3 - D VLSI Accelerator," in *VLSI for AI, Oxford*, ed. W. R. Moore, pp. 179 - 188, Kluwer Academic, USA, 1988.

57. Sivilotti M. A., Emerling M. R., and Mead C. A., "VLSI Architectures for Implementation of Neural Networks," in *AIP Conf Proc, Neural Networks for Computing, Snowbird*, vol. 151, pp. 408 - 413, New York, 1986.

58. Graf H. P. and de Vegvar P., "A CMOS Associative Memory Chip Based on Neural Networks," in *IEEE Int Conf Solid-States Circs*, pp. 304 - 305, February 1987.

59. Schwartz D. B., Howard R. E., and Hubbard W. E., "A Programmable Neural Network Chip," *IEEE Solid State Circs*, vol. 24, no. 2, pp. 313 - 319, April, 1989.

60. Mackie S., Graf H. P., and Schwartz D. B., "Microelectronic Implementations of Connectionist Neural Networks," in *Neural Infomation Processing Systems*, ed. D. Z. Anderson, pp. 515 - 523, AIP, New York, 1987.

61. Schwartz D. B., Howard R. E., and Hubbard W. E., "Adaptive Neural Networks using MOS Charge Storage," in *Advances in Neural Information processing 1*, ed. D. S. Touretzky, pp. 761 - 767, Morgan Kaufmann, San Mateo, CA, 1988.

62. Tsividis Y. P. and Anastassiou D., "Switched - Capacitor Neural Networks," *Electronic Letters*, vol. 23, no. 18, pp. 958 - 959, August, 1987.

63. Akers L. A., Walker M. R., and Ferry D. et al, "A Limited - Interconnect, Highly Layered Synthetic Neural Architecture," in *VLSI for AI, Oxford*, ed. W. R. Moore, pp. 218 - 226, Kluwer Academic, USA, 1988.

64. Akers L. A. and Walker M. R., "A Limited - Interconnect Synthetic Neural IC," in *IEEE International Conf on Neural Networks, San Diego*, pp. 151 - 158, July, 1988.

65. Card H. C., Tarassenko L., Moore W., and Murray A. F., "VLSI Approximations to Hebbian Learning," *Electronic Letters*, 1989 (to be published).

66. Murray A. F., Brownlow M., and Hamilton A. et al, *Pulse Firing Neural Chips Implementing Hundreds of Neurons*, 1989 (to be published).

67. Verleysen M., Sirletti B., and Jespers P., "A New CMOS Architecture for Neural Networks," in *VLSI for AI, Oxford*, ed. W. R. Moore, pp. 227 - 235, Kluwer Academic, USA, 1988.

68. Murray A. F. and Smith A. V. W., "Asynchronous Arithmetic for VLSI Neural Systems," *Electronic Letters*, vol. 23, no. 12, p. 642, June 1987.

69. Murray A. F. and Smith A. V. W., "A Novel Computational and Signalling Method for VLSI Neural Networks," in *Proc Conf European Solid State Circuits*, 1987.

70. Murray A. F., Smith . V. W., and Tarassenko L., "Fully Programmable Analogue VLSI Devices for the Implementation of Neural Networks," in *VLSI for AI, Oxford*, ed. W. R. Moore, pp. 227 - 235, Kluwer Academic, USA, 1988.

71. Smith A. V. W., "The Implementation of Neural Networks as CMOS Integrated Circuits," *PhD. Thesis (University of Edinburgh)*, 1988.

72. Murray A. F., Hamilton A., Reekie H. M., and Tarassenko L., "Pulse-stream Arithmetic in Programmable Neural Networks ," in *Int Symposium on Circuits and Systems, Portland, Oregon*, 1989.

73. Sage J. P., Thompson K., and Withers R. S., "An Artificial Neural Network Integrated Circuit based on MNOS/CCD Principles.," in *AIP Conf Proc, Neural Networks for Computing, Snowbird*, vol. 151, pp. 381 - 385, New York, 1986.

74. Hubbard W., Schwartz D., Denker J., and Graf H. P., "Electronic Neural Networks," in *AIP Conf Proc, Neural Networks for Computing, Snowbird*, vol. 151, pp. 227 - 234, New York, 1986.

75. Graf H. P., Jackel L. D., Howard R. E., and Straughn B., "VLSI Implementation os a Neural Network with Several Hundred Neurons," in *AIP Conf Proc, Neural Networks for Computing, Snowbird*, vol. 151, pp. 182 - 187, New York, 1986.

76. Mann J. R. and Gilbert S., "An Analog Self - Organising Neural Network Chip," in *Advances in Neural Information processing 1*, ed. D. S. Touretzky, pp. 739 - 747, Morgan Kaufmann, San Mateo, CA, 1988.

77. Goser K. and Ruckert U., "VLSI Design of Associative Networks," in *VLSI for AI, Oxford*, ed. W. R. Moore, pp. 227 - 235, Kluwer Academic, USA, 1988.

78. Goser K., "Technological Challenge of Artificial Neural Networks," in *19th European Solid State Device Research Conf, Berlin*, ed. P Lange, pp. 321 - 328, Springer - Verlag, 1989.

79. Denker J. S., in *Neural Network Models of Learning and Adaption*, Internal Publication, 1987.

80. Thakoor A. P., Lamb J. L., and Moopenn A. et al, "Binary Synaptic Connections based on Memory Switching in Hydrogenated Amorphous-Si," in *AIP Conf Proc, Neural Networks for Computing, Snowbird*, vol. 151, pp. 426 - 431, New York, 1986.

81. Moopenn A., Langanbacher H, and Thakoor A. P. et al, "Programmable Synaptic Chip For Electronic Neural Networks," in *Neural Infomation*

Processing Systems, ed. D. Z. Anderson, pp. 565 - 573, AIP, New York, 1987.

82. Vittoz E., "Analog VLSI Implementation of Neural Networks," in *Proc. of Journees D'Electronique - Artificial Neural networks, Lausanne*, pp. 224 - 250, October 1989.

83. Allen T., Mead C., and Faggin F. et al, "An Orientation-Selective VLSI Retina," *Visual Communication ans Image Processing (Proc. 1988 SPIE Conf.)*, p. 1040, 1988.

84. Lyon R. F. and Mead C., "An Analogue Electronic Cochlea," *IEEE Trans. Acoustic Speach and Signal Processing*, vol. 36 , pp. 1119 - 1134, 1988.

85. Hutchison J., Koch C., Luo J., and Mead C., "Computing Motion using Analog and Binary Resistive Networks," *IEEE Computer Magazine*, pp. 53 - 63, March 1988.

86. Murray A. F., Tarassenko L., and Hamilton A., "Programmable Analogue Pulse-Firing Neural Networks," in *Conf on Neural Information Processing*, Dec., 1988.

87. Murray A. F., "Silicon Implementation of Neural Networks," *IEE International Conf on Artificial Neural Networks*, October, 1989.

88. Farhat N. H., "Optoelectronic Neural Networks and Learning Machines," *IEEE Circuits and Devices Magazine*, pp. 32 - 41, September, 1989.

89. Farhat N., Psaltis D., and Prata A., "Optical Implementation of the Hopfield Model," *Applied Optics*, vol. 24, p. 1469, 1985.

90. H. J. White and W. A. Wright, "Holographic Implementation of a Hopfield model with Discrete Weightings," *Applied Optics*, vol. 27, no. 2, pp. 331 - 338, January, 1988.

91. Parisi G., "A Memory which Forgets," *Journal Phys. A: Math. Gen.*, vol. 19, pp. 617 - 620, 1986.

92. Fleisher M. and Levin E., "The Hopfield Model with Multilevel Neuron Models," *IEEE Conf on Neural Infomation Processing Systems, Denver*, 1987.

93. McGregor M. S., Denyer P. B., and Murray A. F., "Single - Phase Clocking

Phase for CMOS VLSI," *Advanced Research in VLSI: Proceedings of the Stanford 1987 Stanford Conference*, 1987.

94. Asada K. and Mavor J., "Mosyn: An MOS Circuit Synthesis Programm employing 3-way Decomposition and Reduction based on 7-valued Logic," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 1986.

95. Asada K., "MOS Circuit Synthesiser: MOSYN/2.0 User Mannual," *Department of Electrical Engineering, University of Edinburgh*, 1986.

96. UW/NW VLSI Consortium, "VLSI Design Tools," *Reference Manual, Department of Computer Science, University of Washington, Seattle.*, 1985.

97. *VMEbus Specification Mannual*, p. Printex Publishing, Inc., 1985.

98. *User's Guide to the Sun-3/110 VMEbus*, 1987.

99. *Configuration Guide for the Sun-3 Product Family, Hardware Configuration-Chapter5*, 1986.

100. *Writing Device Drivers for the Sun Workstation, Hardware Context-Chapter 2*, 1986.

101. Murray A. F., Smith A. V. W., and Butler Z. F., "Bit - Serial Neural Networks," in *Proc. AIP Conf. on Neural Information Processing Systems - Natural and Synthetic*, pp. 573 - 583, 1987.

102. McClelland J. L. and Rumelhart D. E., in *Explorations in Parallel Distributed Processing, A Handbook of Programs and Exercises*, pp. 84 - 99, MIT Press, London, England, 1988.

Appendix A

Carry tree input

```
; Input for Carry Tree
*function carryout
CH: major(not(xor(sigjm1,pm)),b,and(c1,lsb))
else C0
```

Sum tree input

```
; Input for Sum Tree
*function sigout
CH: not(xor(xor(not(xor(sigjm1,pm)),b,and(c1,not(lsb))),pm))
else C0
```

Sign-extend tree input

```
; Input for Sign Extend
;Sign extension with RS
*function SIGNEXT
*input FB SE1 SE2 TL TLP1 RS
C0: 000000 000001 000010 000101 001000 001001 001010 001011 001101 001111
    010000 010001 010010 010100 010101 010110 011000 011001 011010 011011
    011100 011101 011110 011111 100000 100001 100010 100101 101000 101010
    110001 110101
CH: 100011 100100 100110 100111 101001 101011 101100 101101 101110 101111
    110000 110010 110011 110100 110110 110111 111000 111001 111010 111011
    111100 111101 111110 111111 001100 001110 010011 010111 000011 000100
    000110 000111
```

Appendix B

RNL Netlist file, sum.net

```
(load "latchlib.net")
(node top bot intnode out sigjm1 pm b c1 lsb clk N2 N4 N6 N7 N8)
(ptrans sigjm1 top N4 4 3)
(etrans sigjm1 top N2 4 3)
(ptrans b N4 N8 4 3)
(etrans b N4 N6 4 3)
(ptrans c1 N8 bot 4 3)
(etrans lsb N8 bot 4 3)
(etrans c1 N6 N7 4 3)
(ptrans lsb N7 bot 4 3)
(ptrans b N2 N6 4 3)
(etrans b N2 N8 4 3)
(mu intnode top bot clk)
(pisolo out intnode clk)
)
```

RNL logic data input file, sum.l

```
(load "uwstd.l")
(load "icstd.l")
(read-network "sum_mp.bin")
(setq inc 100)
(log-file "sum_mp.out")
V clk    lhlhlhlhlhlhlhlhlhlhlhlhlhlhlh
V sigjm1 llhhllhhllhhllhhllhhllhhllhhllhh
V b      llllhhhhllllhhhhllllhhhhllllhhhh
V c1     lllllllhhhhhhhhllllllllhhhhhhhh
V lsb    llllllllllllllllhhhhhhhhhhhhhhhh
w clk  sigjm1  c1 lsb  clk top bot intnode out N2 N4 N6 N8 N9
sim-init
R
```

RNL output file, sum.out

RNL simulation results : SWITCH LEVEL

TIME (ns)	clk	sigjm1	c1	lsb	clk	top	bot	intnode	out	N2	N4	N6	N8
0	0	0	0	0	0	0	0	0	0	0	0	0	0
100	0	0	0	0	0	0	0	0	0	0	0	0	0
200	1	0	0	0	1	0	0	0	0	0	0	0	0
300	0	1	0	0	0	1	X	0	0	1	X	1	X
400	1	1	0	0	1	1	0	1	0	1	0	1	0
500	0	0	0	0	0	1	X	1	1	X	1	1	X
600	1	0	0	0	1	1	0	1	1	0	1	1	0
700	0	1	0	0	0	1	1	1	1	1	X	X	1
800	1	1	0	0	1	0	0	0	1	0	X	X	0
900	0	0	1	0	0	1	X	0	0	X	1	X	1
1000	1	0	1	0	1	1	0	1	0	0	1	0	1
1100	0	1	1	0	0	1	1	1	1	1	X	1	X
1200	1	1	1	0	1	0	0	0	1	0	X	0	X
1300	0	0	1	0	0	1	1	0	0	X	1	1	X
1400	1	0	1	0	1	0	0	0	0	X	0	0	X
1500	0	1	1	0	0	1	X	0	0	1	X	X	1
1600	1	1	1	0	1	1	0	1	0	1	0	0	1
1700	0	0	0	1	0	1	1	1	1	X	1	X	1
1800	1	0	0	1	1	0	0	0	1	X	0	X	0
1900	0	1	0	1	0	1	X	0	0	1	X	1	X
2000	1	1	0	1	1	1	0	1	0	1	0	1	0
2100	0	0	0	1	0	1	X	1	1	X	1	1	X
2200	1	0	0	1	1	1	0	1	1	0	1	1	0
2300	0	1	0	1	0	1	1	1	1	1	X	X	1
2400	1	1	0	1	1	0	0	0	1	0	X	X	0
2500	0	0	1	1	0	1	1	0	0	X	1	X	1
2600	1	0	1	1	1	0	0	0	0	X	0	X	0
2700	0	1	1	1	0	1	X	0	0	1	X	1	X
2800	1	1	1	1	1	1	0	1	0	1	0	1	0
2900	0	0	1	1	0	1	X	1	1	X	1	1	X
3000	1	0	1	1	1	1	0	1	1	0	1	1	0
3100	0	1	1	1	0	1	1	1	1	1	X	X	1
3200	1	1	1	1	1	0	0	0	1	0	X	X	0

q

Appendix C

1. Address strobe (AS^*): On its falling edge, AS^* informs the SLAVE that the address is stable and can be captured.

2. Data strobes ($DS0^*$, $DS1^*$): Both $DS0^*$ and $DS1^*$ are required for a long word (32-bit) data transfer. The first data strobe falling edge indicates when the MASTER has placed valid data on the data bus.

3. Long word (LW^*): LW^* is active when a long word data transfer is in operation.

4. Write*: $Write^*$ is used by the MASTER to indicate the direction of data transfer operations. When $write^*$ is low, the data transfer is from the MASTER to the SLAVE. When $write^*$ is high, the data transfer direction is from the SLAVE to the MASTER.

5. Data acknowledge ($DTACK^*$): The SLAVE drives $DTACK^*$ low to indicate that it has successfully received the data on a write cycle. On a read cycle, the SLAVE drives $DTACK^*$ low to indicate that it has placed data on the data bus.

Appendix D

```
#include "def.h"
#define VME_BASE       0xD00000
#define VME_SIZE       0x80000
#define Tij_syn        0x0001C/4
#define Tij_patchend   0x00360/4
#define Tij_colend     0x07C00/4
#define Tij_8colend    0x38000/4
#define STATE_start    0x40060/4
#define STATE_end      0x400C4/4
#define STATE_colend   0x01F00/4
#define SUM_startcol   0x50000/4
#define SUM_endcol     0x50B80/4
#define SUM_startpatch 0x50008/4
#define SUM_endpatch   0x50048/4
#define RUN            0x60000/4

extern longword *VME24d32();
main()
begin
  int p,q,r,t,u,v,w,x,a;
  int fd,weight,state,check;
  longword *addr,l,y,activity;
  FILE *fp,*fp1;

  addr= VME24d32(VME_BASE,VME_SIZE,&fd);

    /* This section calculates weight address for each synapse and loads weight */

  fp=fopen("weight_in","r");
  p=0;
  for (w=1; w<=8; ++w) begin              /* No of columns/RAM1 chip */
    q=0;                                  /* reset patch count at col top */
    for (v=1; v<=32; ++v) begin           /* no of patches/column */
      r=0x340/4;                          /* reset synapse count at patch start */
      for (u=1; u<=27; ++u) begin         /* synapse count per patch */
        t=0;                              /* resets weight bit count */
        for (x=0; x<8; ++x) begin         /* counts bits per weight */
          fscanf(fp,"%X",&weight);
          y=p+q+r+t;                      /* calulates address */
          addr[y]=weight;
          t=t+0x1;
        end
        r=r-0x20/4;                       /* counts synapses per patch */
      end
      q=q+0x400/4;                        /* counts patches per column */
    end
    p=p+0x8000/4;                         /* counts columns */
  end

    /* This section calculates the state address and loads it to state RAM */

```

```
52  fp=fopen("state_in","r");
53  for (v=0; v<=STATE_colend; v=v+0x100/4) begin
54    fscanf(fp,"%X",&state);
55    for (a=STATE_start; a<=STATE_end; ++a) begin  /* Load States to RAM 2 */
56      addr[a+v]=state;
57    end
58  end
59  fclose(fp);
60  fclose(fp1);
61
62    /* Board is now loaded and is set to RUN */
63
64  addr[RUN]=0;                          /* Set to D6 for Board Run */
65  do begin
66    check=addr[RUN];
67  end while ((l&0x10000000)!=0x10000000); /* Ends board run on A17 = A16 = 1 */
68
69    /* Board has ended and activies are read back to the Sun */
70
71  fp=fopen("activity.out","w");
72  for (a=SUM_startcol; a<=SUM_endcol; a=a+0x20) begin
73    for (b=0x0008/4; b<=0x0044/4; ++b) begin
74      activity=addr[a+b];              /* reads activities from board */
75    end
76  end
77  fclose(fp);
78  munmap(addr);
79  close(fd);
80 end
81
```

Appendix E

```
1 #include "def.h"
2 #include <math.h>
3 #define VME_BASE       0xD00000
4 #define VME_SIZE       0x80000
5 #define WEIGHTS_end    0x3FFFF/4
6 #define STATE_start    0x40060/4
7 #define STATE_end      0x400C4/4
8 #define STATE_colend   0x00300/4
9 #define SUM_startcol   0x50000/4
10 #define SUM_endcol     0x50100/4
11 #define SUM_startpatch 0x50008/4
12 #define SUM_endpatch   0x50048/4
13 #define RUN            0x60000/4
14
15 extern longword *VME24d32();
16 int sum[12],totalsum;
17 int getweight[36][36][8],order[216];
18 int statenum[]={2,3,0,7,6};
19 float status[]={-1.0,-0.5,0.0,0.5,1.0};   /* status is assigned to any of the 5-states */
20 float temperature;
21 double x1,x2,x3,x4,xmid=0;
22 float targetstate[720];
23 float newstate[720];          /* thresholded activities */
24 float stateval[720];          /* state input value */
25 float delta[720];
26 float deltaweight[36][36];
27 float newweight[36][36];
28 int   trunweight[36][36];
29 int   orderweight[36][36][8];
30
31 void Threshold(xmid,temperature)  /* computes the threshold values */
32 float temperature;                /* to threshold activities      */
33 double xmid;
34 begin
35    x1 = xmid - (temperature* log(8.0));
36    x2 = xmid - (temperature* log(1.75));
37    x3 = xmid + (temperature* log(1.75));
38    x4 = xmid + (temperature* log(8.0));
39 end
40
41 int Get_state(data)                  /* works out which state has been */
42 float data;                          /* read from "state_in" and returns */
43 begin                                /* t=1 -> 4 for state -1 -> +1    */
44    register int t;
45    t=0;
46    while (data != status[t])
47       t++;
48    return(t);
49 end
50
51 int Power(base, sup)
```

```
52 int base, sup;
53 begin
54    int i,j;
55    j=1;
56    for (i=1; i<=sup; ++i)
57       j=j*base;
58    return(j);
59 end
60  /* FUNCTION ALLOWS BOARD TO RUN */
61 void boardrun(addr)
62 longword *addr;
63 begin
64    longword l;
65    addr[RUN]=0;                          /* Set to D6 for Board Run */
66    do begin
67       l=addr[RUN];
68    end while ((l&0x10000000)!=0x10000000);
69 end
70
71 main()
72 begin
73    int v,a,b,c,d,f,g,h,i,j,k,n,readweight,counter,index1,index2;
74    int a_col,a_patch,a_syn,a_bit,patchcount,pattcount,count,icount,jcount,syncount;
75    int result,pattno,pattmax,pp,MSB,ord,ordcount,nop;
76    float lrate,state_result,stateinfo,x,m,ptss,tss;
77    longword *addr,l,e,y;
78    int fd;
79    void Threshold();
80    FILE *fp,*fp1,*fp2,*fp3,*fp4,*fp5,*fp6,*fp7;
81
82    addr= VME24d32(VME_BASE,VME_SIZE,&fd);
83    fp=fopen("new_weight","w");
84    fp1=fopen("state_in","r");
85    fp2=fopen("target_in","r");
86    fp3=fopen("temp","r");
87    fp4=fopen("learn","r");
88    fp5=fopen("pattern_no","r");
89    fscanf(fp3,"%f",&temperature);
90    fscanf(fp4,"%f",&lrate);
91    fscanf(fp5,"%d",&pattmax);
92    Threshold(xmid,temperature);
93    a_col=0;                          /* lines 93 - 102 initialise weights */
94    for (g=0; g<=7; g++) begin                    /* No of columns/RAM1 chip */
95       a_patch=0;                                 /* reset patch count at col top */
96       for (n=0; n<=31; n++) begin                 /* no of patches/column */
97          for (a_syn=0x340/4; a_syn>=0; a_syn=a_syn-0x20/4) begin /* 27 synapses */
98             for ( a_bit=0; a_bit<=7; a_bit++) begin
99                y=a_col+a_patch+a_syn+a_bit;
100               addr[y]=0;
101            end
102         end
103         a_patch=a_patch+0x400/4;                 /* counts patches per column */
104         patchcount=(g*4)+n+1;
```

```
105     end                                    /* end n */
106     a_col=a_col+0x8000/4;                  /* counts columns */
107   end
108   pattcount=1;
109   count=1;
110   do begin
111     tss=0;
112     for (pattno=0; pattno<=pattmax; pattno++) begin
113       ptss=0;
114       pp=pattno*36;
115       if (pattno==0)
116         { fseek(fp1,0,0);
117           fseek(fp2,0,0);
118         }
119       /* SECTION TO READ INPUT STATES */
120             /* Statenum gets the statenumber computed by Get_state then */
121             /* * (8 to power i (for i=0 -> 8)) to get the 9 states to   */
122             /* input to VMEbus                                          */
123       for (v=0; v<=STATE_colend; v=v+0x100/4) begin /* count 4 patches in a column
124         result=0;
125         for (i=0; i<=8; i++) begin          /* want 9 states of patt pattno */
126           fscanf(fp1,"%f",&stateinfo);      /* reads in a state */
127           result=result+(statenum[Get_state(stateinfo)]*Power(8,i));
128           counter= pp+(c*9)+i;              /* counts total no of states */
129           stateval[counter]=stateinfo;
130         end
131         for (a=STATE_start; a<=STATE_end; ++a) begin  /* Load States to RAM 2 *
132           addr[a+v]=result;                 /* 26 clks for state */
133         end
134       end
135       for (v=0x0400/4; v<=0x1F00/4; v=v+0x100/4) begin
136         for (a=STATE_start; a<=STATE_end; ++a) begin  /* Load States to RAM 2 *
137           addr[a+v]=0;
138         end
139       end
140       boardrun(addr);                       /* accelerator board runs */
        /* SECTION READS STATES BACK FROM BOARD */
141       for (a=SUM_startcol; a<=SUM_endcol; a=a+0x20) begin
142         for (i=0; i<=11; i++) begin
143           sum[i]=0;
144         end
145         for (b=0x0008/4; b<=0x0044/4; b++) begin
146           e=addr[a+b];          /* Unload 16bit activities from RAM3 */
147           g=e&0x00000FFF;                   /* makes 3 LSB bits only valid */
        /* SECTION COVERTS HEX TO FLOATING POINT ACTIVITIES */
148           for (i=0; i<=11; i++) begin
149             sum[i]=(!!(g&(Power(2, 11-i)))*Power(2,c))+sum[i];
150           end
151           for (i=0; i<=11; i++) begin
152             MSB=!!(sum[i]&Power(2,15));
153             if ( MSB>0 )
154               {
155               sum[i]=(~sum[i]+1)&0x0000FFFF;
```

```
156             sum[i]=-1*sum[i];
157             }
158           end
159         end
          /* SECTION THRESHOLDS ACTIVITIES */
160         for (i=0; i<=11; i++) begin
161           x=sum[11-i];
162           if (x < x1) state_result = -1.0;
163           if ((x < x2) and (x >= x1)) state_result = -0.5;
164           if ((x < x3) and (x >= x2)) state_result = 0.0;
165           if ((x < x4) and (x >= x3)) state_result = 0.5;
166           if (x >= x4) state_result = 1.0;
167           newstate[pp+((d-1)*12+i)]=state_result;
168         end
169       end                                    /* end a=sum_col */
          /* SECTION IMPLEMENTS DELTA RULE */
170       for (i=0; i<=35; ++i) begin
171         fscanf(fp2,"%f",&targetstate[pp+i]);
172         delta[pp+i]=targetstate[pp+i]-newstate[pp+i];
173         ptss=ptss+delta[pp+i]*delta[pp+i];  /* partial sum sq's for pattno */
174         for (j=0; j<=35; j++) begin
175           deltaweight[i][j]=lrate*delta[pp+i]*stateval[pp+j];
                                  /* computes weight change */
176           newweight[i][j]=newweight[i][j]+deltaweight[i][j]; /* updates weight */
177           if (newweight[i][j] > 127) newweight[i][j]=127;   /* weight max */
178           if (newweight[i][j] <-127) newweight[i][j]=-127;  /* weight min */
179           trunweight[i][j]=newweight[i][j];                 /* set weight to int */
180           for (h=0; h<=7; h++) begin                        /* set weight to binary */
181             getweight[i][j][h]=!!(trunweight[i][j]&Power(2,h)); /* 2's comp */
182           end
183         end                                 /* end j */
          /* SECTION ORDERS AND LOADS WEIGHTS BACK TO BOARD */
184         ord=0;
185         for (nop=0; nop<=3; nop++) begin
186           ordcount=0;
187           for (j=ord; j<=ord+8; j++) begin
188             for (h=0; h<=7; h++) begin
189               if (h-ordcount>=0)
190                 orderweight[i][j][h]=getweight[i][j][h-ordcount];
191               if (h-ordcount<0)
192                 orderweight[i][j][h]=getweight[i][j][8+h-ordcount];
193             end
194             ordcount=ordcount+1;
195           end
196           ord=ord+9;
197         end
198       end                                 /* end i*/
199       tss=tss+ptss;                       /* total sum sq's for all patterns */
200       a_col=0;
201       a=0;
202       patchcount=0;
203       for (g=0; g<=2; g++) begin          /* No of columns/RAM1 chip */
204         a_patch=0;                        /* reset patch count at col top */
```

```
205         b=0;
206         for (n=0; n<=3; n++) begin            /* no of patches/column */
207           for (i=0; i<=215; i++) begin       /* no of patches/column */
208             order[i]=0;
209           end
210           for (k=0; k<=3; k++) begin
211             jcount=0;
212             for (j=b; j<=b+8; j++) begin     /* synapse count per patch */
213               icount=0;
214               for (i=a; i<=a+2; i++) begin
215                 for (h=0; h<=7; h++) begin   /* counts bits per weight */
216                   order[(24*jcount)+(8*icount)+h]=
217                   orderweight[i][j][h]*Power(2,3*k)+
218                   order[(24*jcount)+(8*icount)+h];
219                 end                          /* reads  weights from weight */
220                 icount=icount+1;               /* matrix */
221               end
222               jcount=jcount+1;
223             end                              /* end j */
224             a=a+3;
225           end                                /* end k */
226           a=a-12;
227           b=b+9;
228           syncount=0;
229           for (a_syn=0x340/4; a_syn>=0; a_syn=a_syn-0x20/4) begin
230             for ( a_bit=0; a_bit<=7; a_bit++) begin
231               y=a_col+a_patch+a_syn+a_bit;
232               addr[y]=order[(8*syncount)+a_bit];
233             end
234             syncount=syncount+1;
235           end
236           a_patch=a_patch+0x400/4;              /* counts patches per column */
237           patchcount=(g*4)+n+1;
238         end                                  /* end n */
239         a=a+12;
240         a_col=a_col+0x8000/4;                  /* counts columns */
241       end                                    /* end g */
242     end                                      /* end pattno */
243     printf("%d %f0,count,tss);
244     count=count+1;
245   end while (tss>0);
246   for (i=0; i<=35; i++) begin
247     for (j=0; j<=35; j++) begin
248       for (h=0; h<=7; h++) begin
249         fprintf(fp,"%d ",orderweight[i][j][h]);
250       end
251       fprintf(fp,"0);
252     end
253   end
254   fclose(fp);
255   fclose(fp1);
256   fclose(fp2);
257   fclose(fp3);
```

```
258   fclose(fp4);
259   munmap(addr);
260   close(fd);
261end
```

Appendix F

```
1 #include "def.h"
2 #include <math.h>
3 float weight[36][36],deltaweight[36][36],newweight[36][36];
4 float stateval[720],target[720],recstateval[720],newstate[720],sum[36];
5 float state_result,tss,ptss;
6 float x1,x2,x3,x4;
7 float lrate,xmid=0,temperature;
8 int   pattno;
9
10 void Threshold(xmid,temperature)
11 float temperature,xmid;
12 begin
13   x1=xmid-(temperature*log(8.0));
14   x2=xmid-(temperature*log(1.75));
15   x3=xmid+(temperature*log(1.75));
16   x4=xmid+(temperature*log(8.0));
17 end
18
19 void Actsum(weight,stateval,sum,pattno)   /* Computes neural activity */
20 float stateval[72],weight[36][36],sum[36];
21 begin
22   int  i,j;
23   for (i=0; i<=35; i++) begin
24     sum[i]=0;
25     for (j=0; j<=35; j++) begin
26       sum[i]=sum[i]+stateval[pattno+j]*weight[i][j];
27     end
28   end
29 end
30
31 main()
32 begin
33
34   int i,j,k,l,count;
35   float x,v;
36   void Threshold();
37   void Actsum();
38   FILE *fp0,*fp5,*fp,*fp1,*fp2,*fp3,*fp4,*fp6,*fp8;
39   fp0=fopen("learn","r");
40   fp5=fopen("temp","r");
41   fp=fopen("new_weight","w");
42   fp1=fopen("state_in","r");
43   fp2=fopen("target_in","r");
44   fscanf(fp0,"%f",&lrate);
45   fscanf(fp5,"%f",&temperature);
46   Threshold(xmid,temperature);
47
48   for (i=0; i<=35; i++) begin
49     for (j=0; j<=35; j++) begin
50       weight[i][j]=0;     /* sets weights to 0 */
51     end
```

```
52  end
53  for (i=0; i<=19; i++) begin
54    for (j=0; j<=35; j++) begin
55      fscanf(fp1,"%f",&stateval[36*i+j]);   /* reads input patterns */
56      fscanf(fp2,"%f",&target[36*i+j]);     /* reads target patterns */
57    end
58  end
59  count=1;
60  do begin
61    tss=0;
62    for (k=0; k<=19; k++) begin
63      pattno=36*k;
64      ptss=0;
65      Actsum(weight,stateval,sum,pattno);
66      for (i=0; i<=35; i++) begin      /* thresholds activities */
67        x=sum[i];
68        if (x < x1) state_result = -1.0;
69        if ((x < x2) and (x >= x1)) state_result = -0.5;
70        if ((x < x3) and (x >= x2)) state_result = 0.0;
71        if ((x < x4) and (x >= x3)) state_result = 0.5;
72        if (x >= x4) state_result = 1.0;
73        newstate[pattno+i]=state_result;
74        v=target[pattno+i]-newstate[pattno+i];
75        for (j=0; j<=35; j++) begin     /* for loop calculates new weights */
76          deltaweight[i][j]=lrate*v*stateval[pattno+j];
77          weight[i][j]=deltaweight[i][j]+weight[i][j];
78          if (weight[i][j] > 127) weight[i][j]=127;
79          if (weight[i][j] < -127) weight[i][j]=-127;
80        end
81        ptss=ptss+(v*v);     /* calculates error in output patern */
82      end
83      tss=tss+ptss;
84    end
85    printf("%d %f0,count,tss);
86    count=count+1;
87  end while (tss>0);
88  for (i=0; i<=35; i++) begin
89    for (j=0; j<=35; j++) begin
90      fprintf(fp,"%f0,weight[i][j]);
91    end
92  end
93  fclose(fp);
94  fclose(fp1);
95  fclose(fp2);
96  fclose(fp3);
97  fclose(fp5);
98  fclose(fp0);
99 end
```

Appendix G

Published Papers

Murray A. F., Smith A. V. W., and Butler Z. F., "Bit - Serial Neural Networks," in *Proc. AIP Conf. on Neural Infomation Processing Systems - Natural and Synthetic*, pp.573 - 583, 1987.

Murray A. F., Butler Z. F., and Smith A. V. W., "VLSI Neural Networks," *IEE Colloquium on Parallel Processing*, February, 1988.

Butler Z. F., Murray A. F., and Smith A. V. W., "VLSI Bit - Serial Neural Networks," in *VLSI for AI*, Kluwer Academic Press, UK, pp. 201 - 208, 1988.

BIT - SERIAL NEURAL NETWORKS

Alan F. Murray, Anthony V. W. Smith and Zoe F. Butler.
Department of Electrical Engineering, University of Edinburgh,
The King's Buildings, Mayfield Road, Edinburgh,
Scotland, EH9 3JL.

ABSTRACT

A bit - serial VLSI neural network is described from an initial architecture for a synapse array through to silicon layout and board design. The issues surrounding bit - serial computation, and analog/digital arithmetic are discussed and the parallel development of a hybrid analog/digital neural network is outlined. Learning and recall capabilities are reported for the bit - serial network along with a projected specification for a 64 - neuron, bit - serial board operating at 20 MHz. This technique is extended to a 256 (256^2 synapses) network with an update time of 3ms, using a "paging" technique to time - multiplex calculations through the synapse array.

1. INTRODUCTION

The functions a synthetic neural network may aspire to mimic are the ability to consider many solutions simultaneously, an ability to work with corrupted data and a natural fault tolerance. This arises from the parallelism and distributed knowledge representation which gives rise to gentle degradation as faults appear. These functions are attractive to implementation in VLSI and WSI. For example, the natural fault - tolerance could be useful in silicon wafers with imperfect yield, where the network degradation is approximately proportional to the non-functioning silicon area.

To cast neural networks in engineering language, a neuron is a state machine that is either "on" or "off", which in general assumes intermediate states as it switches smoothly between these extrema. The synapses *weighting* the signals from a transmitting neuron such that it is more or less excitatory or inhibitory to the receiving neuron. The set of synaptic weights determines the stable states and represents the learned information in a system.

The neural state, V_i, is related to the total neural activity stimulated by inputs to the neuron through an *activation function*, F. Neural activity is the level of excitation of the neuron and the activation is the way it reacts in a response to a change in activation. The neural output state at time t, V_i^t, is related to x_i^t by

$$V_i^t = F(x_i^t) \tag{1}$$

The activation function is a "squashing" function ensuring that (say) V_i is 1 when x_i is large and -1 when x_i is small. The neural update function is therefore straightforward:

$$x_i^{t+1} = x_i^t \ \ldots\ldots + \delta \sum_{j=0}^{j=n-1} T_{ij}\ V_j^t \tag{2}$$

where δ represents the rate of change of neural activity, T_{ij} is the synaptic weight and n is the number of terms giving an n - neuron array [1].

Although the *neural* function is simple enough, in a totally interconnected n - neuron network there are n^2 synapses requiring n^2 multiplications and summations and

a large number of interconnects. The challenge in VLSI is therefore to design a simple, compact synapse that can be repeated to build a VLSI neural network with manageable interconnect. In a network with fixed functionality, this is relatively straightforward. If the network is to be able to learn, however, the synaptic weights must be programmable, and therefore more complicated.

2. DESIGNING A NEURAL NETWORK IN VLSI

There are fundamentally two approaches to implementing any function in silicon - digital and analog. Each technique has its advantages and disadvantages, and these are listed below, along with the merits and demerits of bit - serial architectures in digital (synchronous) systems.

Digital vs. analog: The primary advantage of digital design for a synapse array is that digital memory is well understood, and can be incorporated easily. Learning networks are therefore possible without recourse to unusual techniques or technologies. Other strengths of a digital approach are that design techniques are advanced, automated and well understood and noise immunity and computational speed can be high. Unattractive features are that digital circuits of this complexity need to be synchronous and all states and activities are quantised, while real neural networks are asynchronous and unquantised. Furthermore, digital multipliers occupy a large silicon area, giving a low synapse count on a single chip.

The advantages of analog circuitry are that asynchronous behaviour and smooth neural activation are automatic. Circuit elements can be small, but noise immunity is relatively low and arbitrarily high precision is not possible. Most importantly, no reliable analog, non - volatile memory technology is as yet readily available. For this reason, learning networks lend themselves more naturally to digital design and implementation.

Several groups are developing neural chips and boards, and the following listing does not pretend to be exhaustive. It is included, rather, to indicate the spread of activity in this field. Analog techniques have been used to build resistor / operational amplifier networks [2,3] similar to those proposed by Hopfield and Tank [4]. A large group at Caltech is developing networks implementing early vision and auditory processing functions using the intrinsic nonlinearities of MOS transistors in the subthreshold regime [5,6]. The problem of implementing analog networks with electrically programmable synapses has been addressed using CCD/MNOS technology [7]. Finally, Garth [8] is developing a digital neural accelerator board ("Netsim") that is effectively a fast SIMD processor with supporting memory and communications chips.

Bit - serial vs. bit - parallel: Bit - serial arithmetic and communication is efficient for computational processes, allowing good communication within and between VLSI chips and tightly pipelined arithmetic structures. It is ideal for neural networks as it minimises the interconnect requirement by eliminating multi - wire busses. Although a bit - parallel design would be free from computational latency (delay between input and output), pipelining makes optimal use of the high bit - rates possible in serial systems, and makes for efficient circuit usage.

2.1 An asynchronous pulse stream VLSI neural network:

In addition to the digital system that forms the substance of this paper, we are developing a hybrid analog/digital network family. This work is outlined here, and has been reported in greater detail elsewhere [9,10,11]. The generic (logical and layout) architecture of a single network of n totally *interconnected* neurons is shown

schematically in figure 1. Neurons are represented by circles, which signal their states, V_i upward into a matrix of synaptic operators. The state signals are connected to a n - bit horizontal bus running through the synaptic array, with a connection to each synaptic operator in every column. All columns have n operators (denoted by squares) and each operator adds its synaptic contribution, $T_{ij}V_j$, to the running total of activity for the neuron i at the foot of the column. The synaptic function is therefore to *multiply* the signalling neuron state, V_j, by the synaptic weight, T_{ij}, and to *add* this product to the running total. This architecture is common to both the bit - serial and pulse - stream networks.

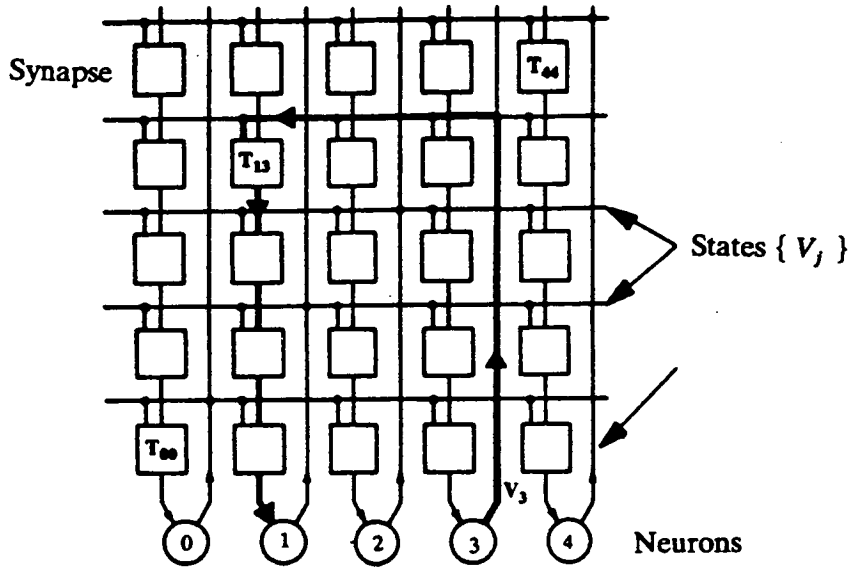


Figure 1. *Generic architecture for a network of n totally interconnected neurons.*

This type of architecture has many attractions for implementation in 2 - dimensional silicon as the summation $\sum_{j=0}^{j=n-1} T_{ij}V_j$ is distributed in space. The interconnect requirement (n inputs to each neuron) is therefore distributed through a column, reducing the need for long - range wiring. The architecture is modular, regular and can be easily expanded.

In the hybrid analog/digital system, the circuitry uses a "pulse stream" signalling method similar to that in a natural neural system. Neurons indicate their state by the presence or absence of pulses on their outputs, and synaptic weighting is achieved by time - chopping the presynaptic pulse stream prior to adding it to the postsynaptic activity summation. It is therefore asynchronous and imposes no fundamental limitations on the activation or neural state. Figure 2 shows the pulse stream mechanism in more detail. The synaptic weight is stored in digital memory local to the operator. Each synaptic operator has an excitatory and inhibitory pulse stream input and output. The resultant product of a synaptic operation, $T_{ij}V_j$, is added to the running total propagating down either the excitatory or inhibitory channel. One binary bit (the MSBit) of the stored T_{ij} determines whether the contribution is excitatory or inhibitory.

The incoming excitatory and inhibitory pulse stream inputs to a neuron are integrated to give a neural activation potential that varies smoothly from 0 to 5 V. This potential controls a feedback loop with an odd number of logic inversions and

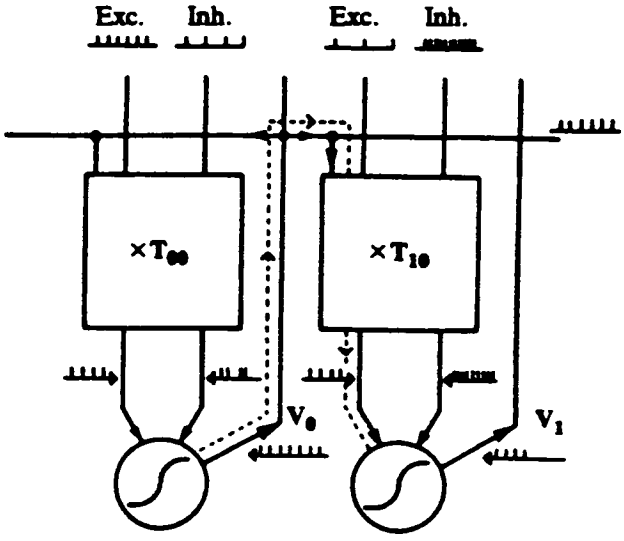


Figure 2. *Pulse stream arithmetic. Neurons are denoted by* ○ *and synaptic operators by* □.

thus forms a switched "ring - oscillator". If the inhibitory input dominates, the feedback loop is broken. If excitatory spikes subsequently dominate at the input, the neural activity rises to 5V and the feedback loop oscillates with a period determined by a delay around the loop. The resultant periodic waveform is then converted to a series of voltage spikes, whose pulse rate represents the neural state, V_i. Interestingly, a not dissimilar technique is reported elsewhere in this volume, although the synapse function is executed differently [12].

3. A 5 - STATE BIT - SERIAL NEURAL NETWORK

The overall architecture of the 5 - state bit - serial neural network is identical to that of the pulse stream network. It is an array of n^2 interconnected synchronous synaptic operators, and whereas the pulse stream method allowed V_j to assume all values between "off" and "on", the 5 - state network V_j is constrained to 0, ± 0.5 or ± 1. The resultant activation function is shown in Figure 3. Full digital multiplication is costly in silicon area, but multiplication of T_{ij} by $V_j = 0.5$ merely requires the synaptic weight to be right - shifted by 1 bit. Similarly, multiplication by 0.25 involves a further right - shift of T_{ij}, and multiplication by 0.0 is trivially easy. $V_j < 0$ is not problematic, as a switchable adder/subtractor is not much more complex than an adder. Five neural states are therefore feasible with circuitry that is only slightly more complex than a simple serial adder. The neural state expands from a 1 bit to a 3 bit (5 - state) representation, where the bits represent "add/subtract?", "shift?" and "multiply by 0?".

Figure 4 shows part of the synaptic array. Each synaptic operator includes an 8 bit shift register memory block holding the synaptic weight, T_{ij}. A 3 bit bus for the 5 neural states runs horizontally above each synaptic row. Single phase dynamic CMOS has been used with a clock frequency in excess of 20 MHz [13]. Details of a synaptic operator are shown in figure 5. The synaptic weight T_{ij} cycles around the shift register and the neural state V_j is present on the state bus. During the first clock cycle, the synaptic weight is multiplied by the neural state and during the second, the most significant bit (MSBit) of the resultant $T_{ij}V_j$ is sign - extended for

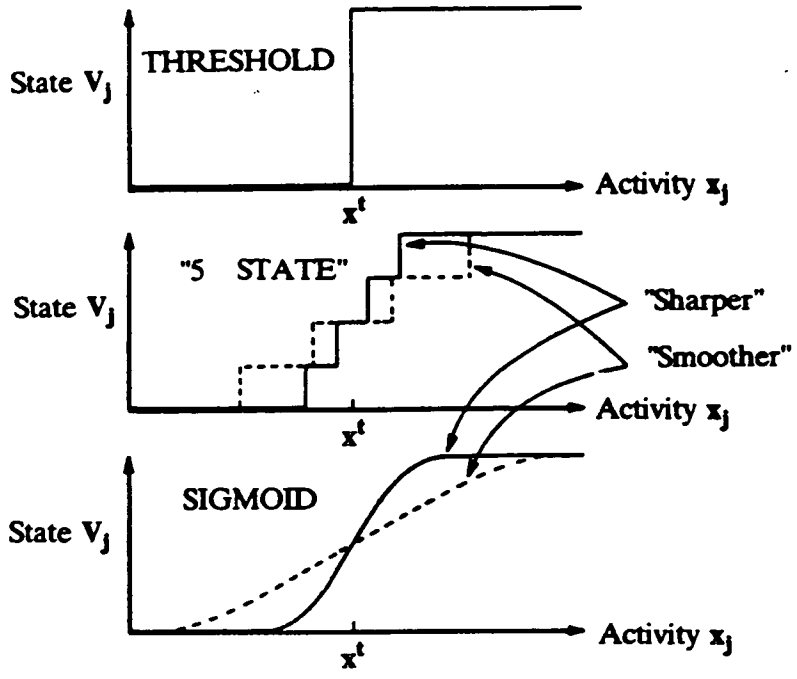


Figure 3. *"Hard - threshold", 5 - state and sigmoid activation functions.*

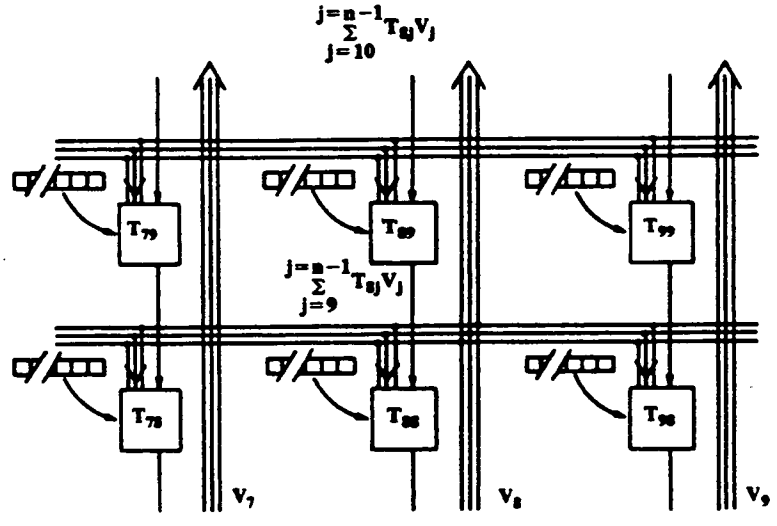


Figure 4. *Section of the synaptic array of the 5 - state activation function neural network.*

8 bits to allow for word growth in the running summation. A least significant bit (LSBit) signal running down the synaptic columns indicates the arrival of the LSBit of the x_i running total. If the neural state is ± 0.5 the synaptic weight is right shifted by 1 bit and then added to or subtracted from the running total. A multiplication of ± 1 adds or subtracts the weight from the total and multiplication by 0

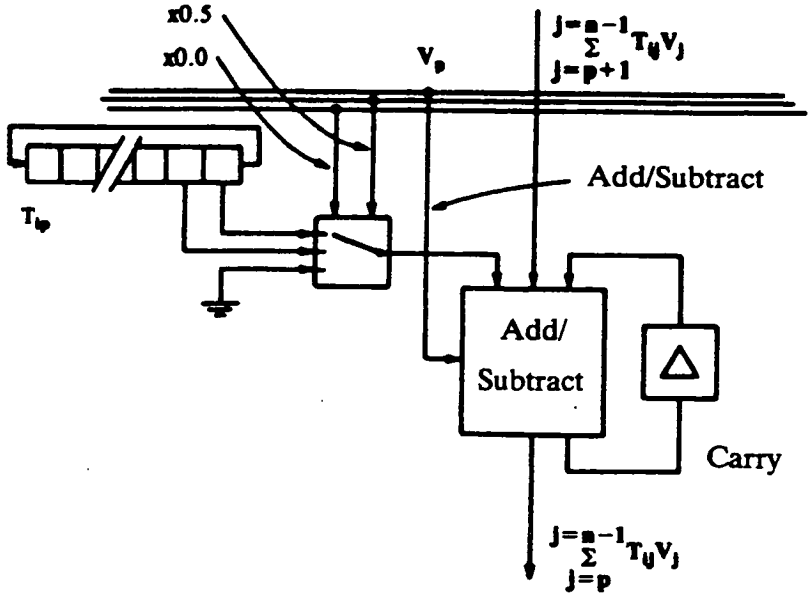


Figure 5. *The synaptic operator with a 5 - state activation function.*

does not alter the running summation.

The final summation at the foot of the column is thresholded externally according to the 5 - state activation function in figure 3. As the neuron activity x_j, increases through a threshold value x_t, ideal sigmoidal activation represents a smooth switch of neural state from -1 to 1. The 5 - state "staircase" function gives a superficially much better approximation to the sigmoid form than a (much simpler to implement) threshold function. The sharpness of the transition can be controlled to "tune" the neural dynamics for learning and computation. The control parameter is referred to as temperature by analogy with statistical functions with this sigmoidal form. High "temperature" gives a smoother staircase and sigmoid, while a temperature of 0 reduces both to the "Hopfield" - like threshold function. The effects of temperature on both learning and recall for the threshold and 5 - state activation options are discussed in section 4.

4. LEARNING AND RECALL WITH VLSI CONSTRAINTS

Before implementing the reduced - arithmetic network in VLSI, simulation experiments were conducted to verify that the 5 - state model represented a worthwhile enhancement over simple threshold activation. The "benchmark" problem was chosen for its ubiquitousness, rather than for its intrinsic value. The implications for learning and recall of the 5 - state model, the threshold (2 - state) model and smooth sigmoidal activation (∞ - state) were compared at varying temperatures with a restricted dynamic range for the weights T_{ij}. In each simulation a totally interconnected 64 node network attempted to learn 32 random patterns using the delta rule learning algorithm (see for example [14]). Each pattern was then corrupted with 25% noise and recall attempted to probe the content addressable memory properties under the three different activation options.

During learning, individual weights can become large (positive or negative). When weights are "driven" beyond the maximum value in a hardware implementation,

which is determined by the size of the synaptic weight blocks, some limiting mechanism must be introduced. For example, with eight bit weight registers, the limitation is $-128 \leq T_{ij} \leq 127$. With integer weights, this can be seen to be a problem of *dynamic range*, where it is the relationship between the smallest possible weight (± 1) and the largest ($+127/-128$) that is the issue.

Results: Fig. 6 shows examples of the results obtained, studying *learning* using 5 - state activation at different temperatures, and recall using both 5 - state and threshold activation. At temperature T=0, the 5 - state and threshold models are degenerate, and the results identical. Increasing smoothness of activation (temperature) during learning improves the *quality* of learning regardless of the activation function used in recall, as more patterns are recognised successfully. Using 5 - state activation in recall is more effective than simple threshold activation. The effect of dynamic range restrictions can be assessed from the horizontal axis, where T_{ij}^{max} is shown. The results from these and many other experiments may be summarised as follows:-

5 - State activation vs. threshold:

1) Learning with 5 - state activation was protracted over the threshold activation, as *binary* patterns were being learnt, and the inclusion of intermediate values added extra degrees of freedom.
2) Weight sets learnt using the 5 - state activation function were "better" than those learnt via threshold activation, as the recall properties of both 5 - state and threshold networks using such a weight set were more robust against noise.
3) Full sigmoidal activation was better than 5 - state, but the enhancement was less significant than that incurred by moving from threshold → 5 - state. This suggests that the law of diminishing returns applies to addition of levels to the neural state V_j. This issue has been studied mathematically [15], with results that agree qualitatively with ours.

Weight Saturation:

Three methods were tried to deal with weight saturation. Firstly, inclusion of a decay, or "forgetting" term was included in the learning cycle [1]. It is our view that this technique *can* produce the desired weight limiting property, but in the time available for experiments, we were unable to "tune" the rate of decay sufficiently well to confirm it. Renormalisation of the weights (division to bring large weights back into the dynamic range) was very unsuccessful, suggesting that information distributed throughout the numerically small weights was being destroyed. Finally, the weights were allowed to "clip" (ie any weight outside the dynamic range was set to the maximum allowed value). This method proved very successful, as the learning algorithm adjusted the weights over which it still had control to compensate for the saturation effect. It is interesting to note that other experiments have indicated that Hopfield nets can "forget" in a different way, under different learning control, giving preference to recently acquired memories [16]. The results from the saturation experiments were:-

1) For the 32 pattern/64 node problem, integer weights with a dynamic range greater than ± 30 were necessary to give enough storage capability.
2) For weights with maximum values $T_{ij}^{max} = 50 \rightarrow 70$, "clipping" occurs, but network performance is not seriously degraded over that with an unrestricted weight set.

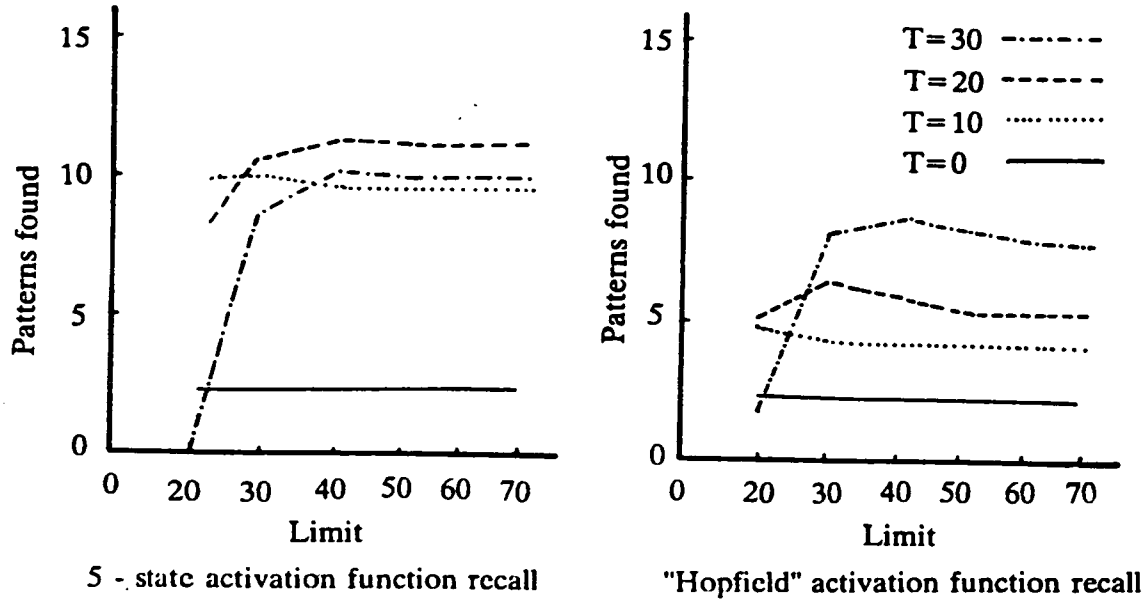


Figure 6. *Recall of patterns learned with the 5 - state activation function and subsequently restored using the 5-state and the hard - threshold activation functions.*
T is the "temperature", or smoothness of the activation function, and "limit" the value of T_{ij}^{max}.

These results showed that the 5 - state model was worthy of implementation as a VLSI neural board, and suggested that 8 - bit weights were sufficient.

5. PROJECTED SPECIFICATION OF A HARDWARE NEURAL BOARD

The specification of a 64 neuron board is given here, using a 5 - state bit - serial 64 x 64 synapse array with a derated clock speed of 20 MHz. The synaptic weights are 8 bit words and the word length of the running summation x_i is 16 bits to allow for growth. A 64 synapse column has a computational latency of 80 clock cycles or bits, giving an update time of 4μs for the network. The time to load the weights into the array is limited to 60μs by the supporting RAM, with an access time of 120ns. These load and update times mean that the network is executing 1 x 10^9 operations/second, where one operation is $\pm\ T_{ij}V_j$. This is much faster than a natural neural network, and much faster than is necessary in a hardware accelerator. We have therefore developed a "paging" architecture, that effectively "trades - off" some of this excessive speed against increased network size.

A "moving - patch" neural board: An array of the 5 - state synapses is currently being fabricated as a VLSI integrated circuit. The shift registers and the adder/subtractor for each synapse occupy a disappointingly large silicon area, allowing only a 3 x 9 synaptic array. To achieve a suitable size neural network from this array, several chips need to be included on a board with memory and control circuitry. The "moving patch" concept is shown in figure 7, where a small array of synapses is passed over a much larger n x n synaptic array.

Each time the array is "moved" to represent another set of synapses, new weights must be loaded into it. For example, the first set of weights will be T_{11} ... T_{ij} ... T_{21} ... T_{2j} to T_{jj}, the second set $T_{j+1,1}$ to T_{xx} etc.. The final weight to be loaded will be

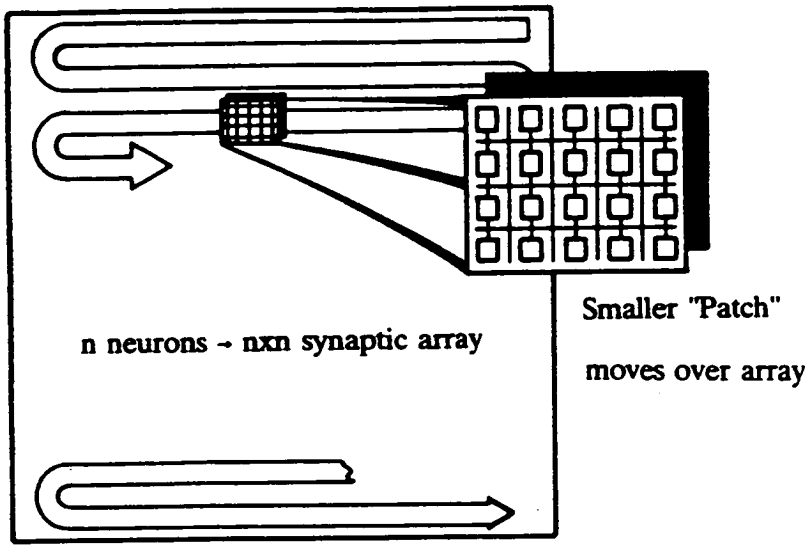


Figure 7. *The "moving patch" concept, passing a small synaptic "patch" over a larger nxn synapse array.*

T_{nn}. Static, off - the - shelf RAM is used to store the weights and the whole operation is pipelined for maximum efficiency. Figure 8 shows the board level design for the network.

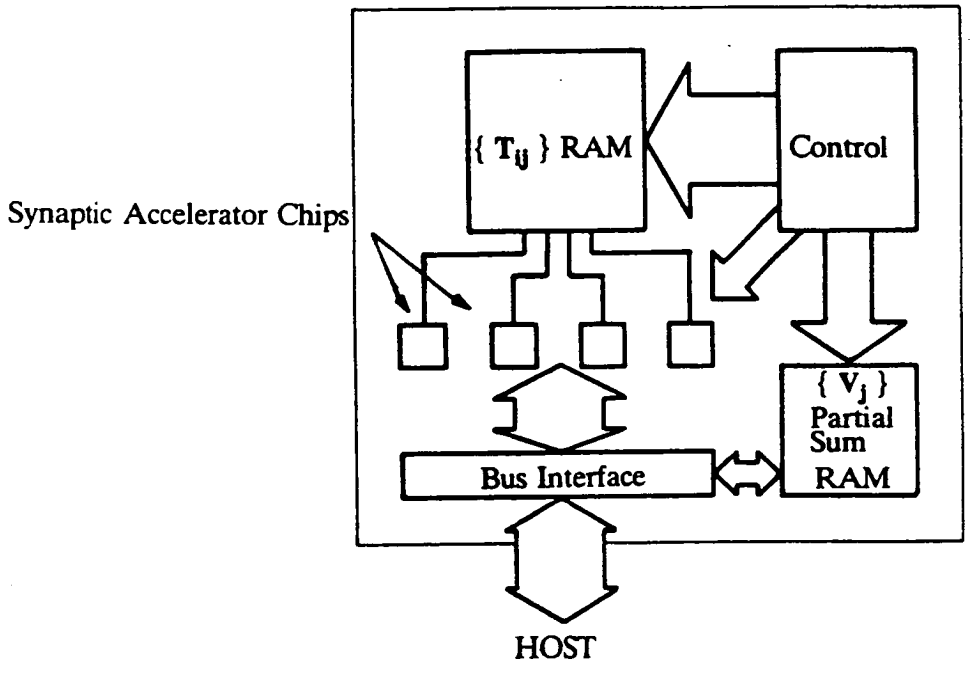


Figure 8. *A "moving patch" neural network board.*

The small "patch" that moves around the array to give n neurons comprises 4 VLSI synaptic accelerator chips to give a 6 x 18 synaptic array. The number of neurons to be simulated is 256 and the weights for these are stored in 0.5 Mb of RAM with a load time of 8ms. For each "patch" movement, the partial running summation, $\bar{x}_j$,

calculated for each column, is stored in a separate RAM until it is required to be added into the next appropriate summation. The update time for the board is 3ms giving 2 x 10^7 operations/second. This is slower than the 64 neuron specification, but the network is 16 times larger, as the arithmetic elements are being used more efficiently. To achieve a network of greater than 256 neurons, more RAM is required to store the weights. The network is then slower unless a larger number of accelerator chips is used to give a larger moving "patch".

6. CONCLUSIONS

A strategy and design method has been given for the construction of bit - serial VLSI neural network chips and circuit boards. Bit - serial arithmetic, coupled to a reduced arithmetic style, enhances the level of integration possible beyond more conventional digital, bit - parallel schemes. The restrictions imposed on both synaptic weight size and arithmetic precision by VLSI constraints have been examined and shown to be tolerable, using the associative memory problem as a test.

While we believe our digital approach to represent a good compromise between arithmetic accuracy and circuit complexity, we acknowledge that the level of integration is disappointingly low. It is our belief that, while digital approaches may be interesting and useful in the medium term, essentially as hardware accelerators for neural simulations, analog techniques represent the best ultimate option in 2 - dimensional silicon. To this end, we are currently pursuing techniques for analog pseudo - static memory, using standard CMOS technology. In any event, the full development of a nonvolatile analog memory technology, such as the MNOS technique [7], is key to the long - term future of VLSI neural nets that can learn.

7. ACKNOWLEDGEMENTS

The authors acknowledge the support of the Science and Engineering Research Council (UK) in the execution of this work.

References

1. S. Grossberg, "Some Physiological and Biochemical Consequences of Psychological Postulates," *Proc. Natl. Acad. Sci. USA*, vol. 60, pp. 758 - 765, 1968.
2. H. P. Graf, L. D. Jackel, R. E. Howard, B. Straughn, J. S. Denker, W. Hubbard, D. M. Tennant, and D. Schwartz, "VLSI Implementation of a Neural Network Memory with Several Hundreds of Neurons," *Proc. AIP Conference on Neural Networks for Computing, Snowbird*, pp. 182 - 187, 1986.
3. W. S. Mackie, H. P. Graf, and J. S. Denker, "Microelectronic Implementation of Connectionist Neural Network Models," *IEEE Conference on Neural Information Processing Systems, Denver*, 1987.
4. J. J. Hopfield and D. W. Tank, "Neural" Computation of Decisions in Optimisation Problems," *Biol. Cybern.*, vol. 52, pp. 141 - 152, 1985.
5. M. A. Sivilotti, M. A. Mahowald, and C. A. Mead, *Real - Time Visual Computations Using Analog CMOS Processing Arrays*, 1987. To be published
6. C. A. Mead, "Networks for Real - Time Sensory Processing," *IEEE Conference on Neural Information Processing Systems, Denver*, 1987.

7. J. P. Sage, K. Thompson, and R. S. Withers, "An Artificial Neural Network Integrated Circuit Based on MNOS/CCD Principles," *Proc. AIP Conference on Neural Networks for Computing, Snowbird*, pp. 381 - 385, 1986.

8. S. C. J. Garth, "A Chipset for High Speed Simulation of Neural Network Systems," *IEEE Conference on Neural Networks, San Diego*, 1987.

9. A. F. Murray and A. V. W. Smith, "A Novel Computational and Signalling Method for VLSI Neural Networks," *European Solid State Circuits Conference* , 1987.

10. A. F. Murray and A. J. W. Smith, "Asynchronous Arithmetic for VLSI Neural Systems," *Electronics Letters*, vol. 23, no. 12, p. 642, June, 1987.

11. A. F. Murray and A. V. W. Smith, "Asynchronous VLSI Neural Networks using Pulse Stream Arithmetic," *IEEE Journal of Solid-State Circuits and Systems*, 1988. To be published

12. M. E. Gaspar, "Pulsed Neural Networks : Hardware, Software and the Hopfield A/D Converter Example," *IEEE Conference on Neural Information Processing Systems, Denver*, 1987.

13. M. S. McGregor, P. B. Denyer, and A. F. Murray, "A Single - Phase Clocking Scheme for CMOS VLSI," *Advanced Research in VLSI : Proceedings of the 1987 Stanford Conference*, 1987.

14. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation," *Parallel Distributed Processing : Explorations in the Microstructure of Cognition*, vol. 1, pp. 318 - 362, 1986.

15. M. Fleisher and E. Levin, "The Hopfiled Model with Multilevel Neurons Models," *IEEE Conference on Neural Information Processing Systems, Denver*, 1987.

16. G. Parisi, "A Memory that Forgets," *J. Phys. A : Math. Gen.*, vol. 19, pp. L617 - L620, 1986.

NEURAL NETWORKS

Murray, Z. F. Butler and A. V. W. Smith

TRODUCTION

ietic neurons are simple computational units operating in massively parallel arrays, that capture of the functionality and computational strengths of the brain. In engineering terms, a biological on (for example, member i of a network of n neurons) is a unit that signals its state V_i, by the nce ("on") or absence ("off") of voltage pulses on its output, or **axon.** Neuron i decides its state mputing its **activity** x_i, which can be altered by direct stimulation of the neuron from outside the ork and by contributions from other neurons in the network. The neuron state, V_j, is related to an **activation function,** f. Neural activity is the level of excitation of the neuron and the activa- function describes its response to a change in activation. The contributions from other neurons veighted by interneural **synaptic weights** $\{T_{ij}\}$. The state of neuron i in a n - neuron array [1] is given by:-

$$V_i = f(x_i) = f\left(\sum_{j=0}^{j=n-1} T_{ij}V_j + I_i\right) \qquad (1)$$

activation function $f(x_i)$ defines the range and resolution of V_i, and the smoothness with which uron moves between the "off" and "on" states and ensures that (say) V_i is 1 when x_i is large and nen x_i is small. I_i is a direct input that may be arbitrarily strong to force a value on V_i. Synaptic hts $\{T_{ij}\}$ may be positive (excitatory) or negative (inhibitory) and any neuron may tend to turn ner neuron "on" or "off" respectively. Information is encoded in or "learnt" by the network by ng the long term memory storage elements $\{T_{ij}\}$. Recall or computation is performed as the net- moves around the n - dimensional space defined by the $\{V_j\}$ with the $\{T_{ij}\}$ constant. This is valent to a recursive and asynchronous evaluation of eqn. (1) until equilibrium is reached. The al function is straightforward, but in a totally interconnected n - neuron array, eqn. (1) requires ultiplications and a large number of interconnections for each network update cycle. Therefore, challenge in VLSI is to design a simple, compact synapse with minimal inter-synapse connections can be easily implemented in silicon. This is relatively simple for a network with fixed functional- However if the network is to be able to learn, it becomes more complicated as the synaptic hts must be programmable.

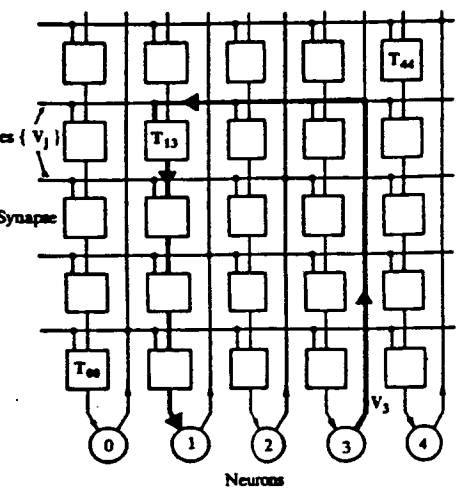
EURAL NETWORK ARCHITECTURE

e are fundamentally two approaches to implementing any function in silicon - digital and analo- The two neural systems designed here use a hybrid analogue/digital method and a bit-serial digi- ethod. The general architecture (logical and layout), used by both designs is shown schemati- in figure 1. This is a single network of n totally interconnected neurons. Neurons are esented by circles, that signal their states, V_i upward into a matrix of synaptic operators. The state ls are connected to a n bit horizontal bus running across the synaptic array, with a connection to naptic operator in every column. Each column has n operators (denoted by squares) that add synaptic contribution $T_{ij}V_j$, to the running total of activity for the neuron i at the end of the mn. The synaptic function is therefore to *multiply* the signalling neuron state, V_j, by the synaptic ht, T_{ij} and to *add* this product to the running total.

type of architecture has many attractions for implementation in 2 - dimensional silicon as the nation is distributed in space. The interconnect requirement is distributed through a column, cing the need for long-range wiring. The architecture is modular, regular and easily expanded.

hybrid analogue/digital system: This uses a "pulse stream" method similar to that in a natural m. Neurons indicate their state by the presence or absence of pulses on their outputs and synaptic hting is achieved by time-chopping the presynaptic pulse stream prior to adding it to the post ptic activity summation. It is therefore asynchronous and imposes no fundamental limitations on ctivation or neural state. Figure 2 shows the pulse stream mechanism in more detail. The synap- veight is stored in digital memory local to the synapse. Each synaptic operator has an excitatory

hibitory pulse stream output. The resultant product of the operation, $T_{ij}V_j$, is added to the run-total propagating down either the excitatory or the inhibitory channel. One binary bit (the t) of the stored T_{ij} determines whether the contribution is excitatory or inhibitory. The incom-citatory and inhibitory pulse stream inputs to a neuron are integrated to give a neural activation ial that varies smoothly from 0 to 5 V. This potential controls a feedback loop with an odd er of logic inversions and thus forms a switched "ring-oscillator". If the inhibitory input dom-, the feedback loop is broken. If excitatory spikes subsequently dominate at the input, the neural y rises to 5 V and the feedback loop oscillates with a period determined by a delay around the The resultant periodic waveform is then converted to a series of voltage spikes, whose pulse rate ents the neural state, V_i. A 64 synapse array using this method has been fabricated in 3μ S technology. The work outlined here has been reported in greater detail elsewhere [2, 3, 4].



e 1. *Generic architecture for a network otally interconnected neurons.*

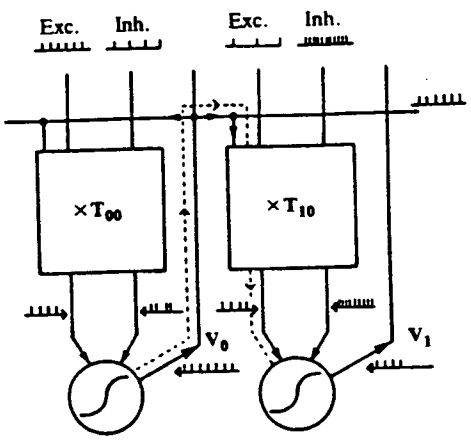


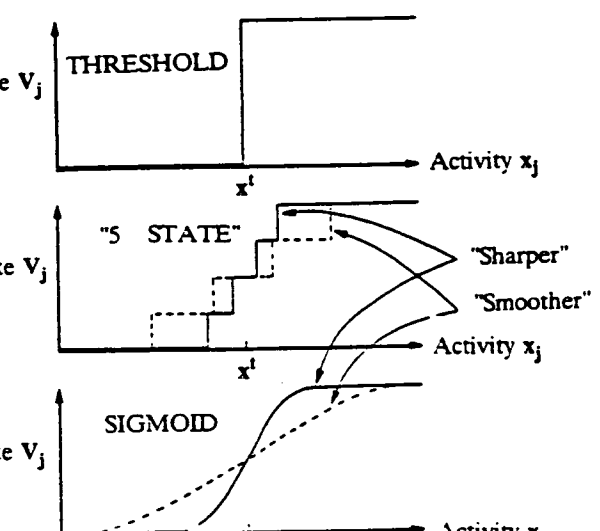
Figure 2. *Pulse stream arithmetic. Neurons are denoted by ○ and synaptic operators by □.*

bit-serial digital system: This system again comprises an array of n^2 interconnected synchronous tic operators. The major difference between the two, is that the pulse stream method allows V_j sume all values between "off" and "on", whereas the bit-serial network is constrained to 5-states n are $V_j = 0, \pm 0.5$ or ± 1. The resultant activation functions for the pulse stream and 5-state orks are shown in figure 3. Multiplication of T_{ij} by $\{V_j = 0.5\}$ simply requires that T_{ij} be right-d by 1 bit and multiplication by 0 requires the product to be set to 0. $V_j < 0$ is implemented in tchable adder/subtractor. Figure 4 shows details of synaptic operators in the array. Each operator n 8-bit shift register memory block holding the synaptic weight, which is "multiplied" by the l state, V_j, signalled on a 3-bit bus. The running summation $T_{ij}V_j$ is 16 bits to allow for word h down the column. A least significant bit (LSBit) signal running down the synaptic columns ates the arrival of the LSBit of the x_i running total.

inal value of the activity arriving at the neuron in each column is thresholded externally accord-the 5-state activation function in figure 3. As the neuron activity increases through a threshold x_t, the ideal activation represents a smooth switch of neural state from -1 to +1. The 5-state case" function gives a superficially much better approximation to the form than the (simpler to ment) threshold function. The sharpness of the transition affects the neural ability for learning computation. The control parameter is referred to as "temperature" by analogy to statistical func-with this form. High temperature gives a smoother staircase and sigmoid and zero temperature es the sigmoid to the threshold function.

EARNING AND RECALL CAPABILITIES WITH VLSI CONSTRAINTS

ning and recall capabilities of the 5-state function were simulated in software against those of the e threshold model and the sigmoidal activation, at varying temperatures with a restricted mic range for the weights, T_{ij}. In each simulation a totally interconnected 64 node network pted to learn 32 patterns using the delta rule algorithm [5]. Each pattern was then corrupted 25 % noise. The results showed that weight sets learnt using the 5-state activation function were r" than those learnt via the threshold activation. Recall of the patterns was also more effective the 5-state model. Full sigmoid activation was superior to the 5-state, but the enhancement was



re 3. *"Hard - threshold", 5 - state and noid activation functions.*

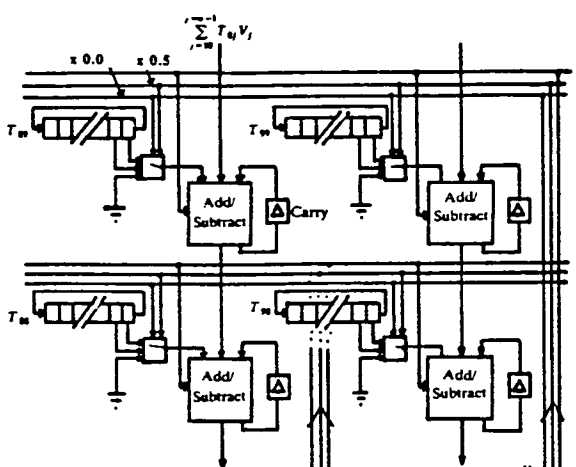


Figure 4. *Section of the synaptic array of the 5 - state activation function neural network.*

ignificant than that incurred by moving from threshold to 5-state. The best method to deal with nt saturation during learning was to permit any weight outside the dynamic range to be set to its mum allowed value. These results showed that the 5-state model was worthy of fabrication at a l level and implementation on a neural board. A full discussion of the results can be found in [6].

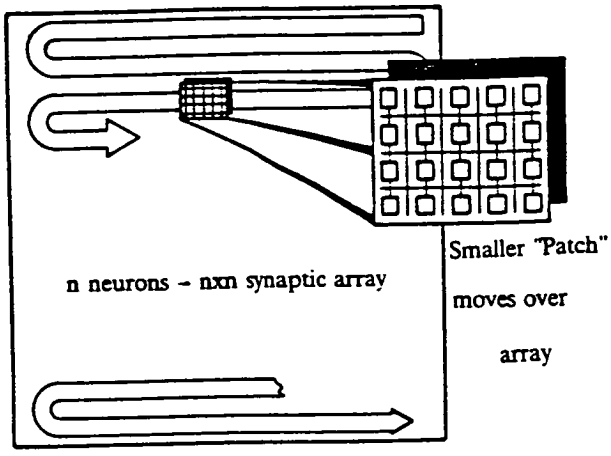
HARDWARE NEURAL BOARD

ecification has been calculated for a 64 neuron board using a 5-state bit-serial 64 x 64 synapse . The weight set is stored in supporting RAM with an access time of 120 ns. This limits the ht loading time to the RAM to 60 μs. These load and access times enable the network to operate x 10^9 operations/second, where one operation is $\pm T_{ij}V_j$. This is much faster than a natural al network and faster than is necessary in a hardware accelerator. A "paging" architecture has fore been developed to "trade-off" some of this excessive speed for increased network size.

oving-patch" neural board: An array of the 5 - state synapses is currently being fabricated as a l integrated circuit using singe phase 3μ CMOS technology. [7]. The full custom layout for each ose occupies a disappointingly large silicon area, allowing only a 3 x 9 synaptic array. To achieve table size neural network from this array, several chips need to be included on a board with ory and control circuitry. The "moving patch" concept is shown in figure 5, where a small array napses is passed over a much larger n x n synaptic array. Each time the array is "moved" to sent another set of synapses, new weights must be loaded into it. For example, the first set of hts will be $T_{11} \ldots T_{ij} \ldots T_{21} \ldots T_{2j}$ to T_{jj}, the second set $T_{j+1,1}$ to T_{zz} etc.. The final weight to be ed will be T_{nn}. Static, off-the-shelf RAM is used to store the weights and the whole operation is ined for maximum efficiency. Figure 6 shows the board level design for the network. The small h" that moves around the array comprises four VLSI synaptic accelerator chips to give a 6 x 18 otic array. The number of neurons to be simulated is 256 and the weights for these are stored in Mb of RAM with a load time of 8ms. For each "patch" movement, the partial running summa- $\bar{x}_j$, calculated for each column, is stored in a separate RAM until it is required to be added into next appropriate summation. The update time for the board is 3ms giving 2 x 10^7 ations/second. This is slower than the 64 neuron specification, but the network is 16 times larger, e arithmetic elements are being used more efficiently. To achieve a network of greater than 256 ons, more RAM is required to store the weights. The network is then slower unless a larger ber of accelerator chips is used to give a larger moving "patch".

ONCLUSIONS

egies and design methods have been given for the construction of a hybrid analogue/digital VLSI al network chip and a bit-serial VLSI network and board. Bit-serial and "reduced-style" arith- c enhances the level of integration beyond more conventional digital, bit-parallel schemes. The ctions imposed on both synaptic weight size and arithmetic precision by VLSI constraints have



re 5. *The "moving patch" concept, passing a synaptic "patch" over a larger nxn synapse array.*

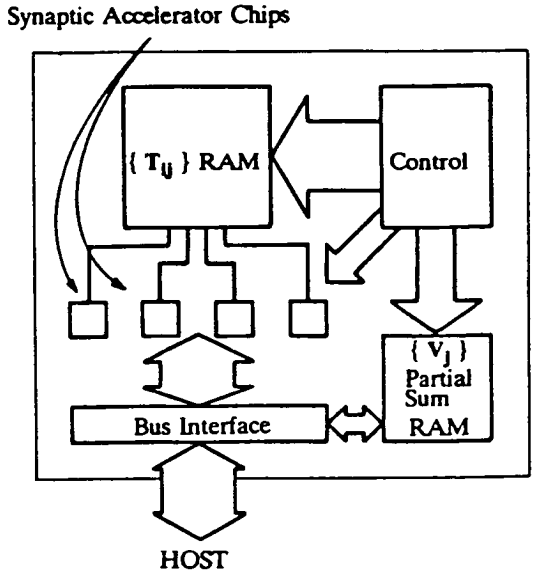


Figure 6. *A "moving patch" neural network board.*

examined and shown to be tolerable, using the associative memory problem as a test.

e we believe our digital approach to represent a good compromise between arithmetic accuracy circuit complexity, we acknowledge that the level of integration is disappointingly low. It is our f that, while digital approaches may be interesting and useful in the medium term, essentially as ware accelerators for neural simulations, analogue techniques represent the best ultimate option - dimensional silicon. To this end, we are currently pursuing techniques for analogue pseudo - memory, using standard CMOS technology. In any event, the full development of a nonvolatile ogue memory technology, such as the MNOS technique [8], is key to the long - term future of I neural nets that can learn.

authors acknowledge the support of the Science and Engineering Research Council (UK) in the ution of this work.

rences

S. Grossberg, "Some Physiological and Biochemical Consequences of Psychological Postulates," *Proc. Natl. Acad. Sci. USA*, vol. 60, pp. 758 - 765, 1968.

A. F. Murray and A. V. W. Smith, "A Novel Computational and Signalling Method for VLSI Neural Networks," *European Solid State Circuits Conference* , 1987.

A. F. Murray and A. J. W. Smith, "Asynchronous Arithmetic for VLSI Neural Systems," *Electronics Letters*, vol. 23, no. 12, p. 642, June, 1987.

A. F. Murray and A. V. W. Smith, "Asynchronous VLSI Neural Networks using Pulse Stream Arithmetic," *IEEE Journal of Solid-State Circuits and Systems*, 1988. To be published

D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Internal Representations by Error Propagation," *Parallel Distributed Processing : Explorations in the Microstructure of Cognition*, vol. 1, pp. 318 - 362, 1986.

A. F. Murray, A. V. W. Smith, and Z. F. Butler, "Bit - Serial Neural Networks," *IEEE Conference on Neural Information Processing Systems - Natural and Synthetic, Denver*, 1987. To be published.

M. S. McGregor, P. B. Denyer, and A. F. Murray, "A Single - Phase Clocking Scheme for CMOS VLSI," *Advanced Research in VLSI : Proceedings of the 1987 Stanford Conference*, 1987.

J. P. Sage, K. Thompson, and R. S. Withers, "An Artificial Neural Network Integrated Circuit Based on MNOS/CCD Principles," *Proc. AIP Conference on Neural Networks for Computing, Snowbird*, pp. 381 - 385, 1986.

VLSI BIT - SERIAL NEURAL NETWORKS

Zoe F. Butler, Alan F. Murray and Anthony V.W. Smith

INTRODUCTION

A synthetic neural network can be viewed as a large parallel array of n^2 synaptic operators, (for n neurons) that is able to model some of the brain's characteristics. The VLSI neural network described, functions with bit-serial, two's complement arithmetic and uses a single phase clocking technique operating at a minimum of 20 MHz (McGregor et al 1987).

A synthetic neuron is a state machine that is either "on" or "off", assuming intermediate states as it switches smoothly between these extrema. A synapse weights the signal from a transmitting neuron such that it is more or less excitatory or inhibitory to the receiving neuron. The total level of activation of a neuron is represented by its *activity*, x_i. This is related to the state of the receiving neuron by an activation function, f, that describes its response to a change in activation. Biologically, this function is sigmoidal, but in our synthetic network it is simplified so that $V_i = 1$ when x_i is large and -1 when x_i is small, with 3 states in between. The interneural *synaptic weights*, T_{ij}, are the contributions from other neurons, that are weighted by the receiving neuron. Therefore, the state of neuron i in an n - neuron array is given by:-

$$V_i = f(x_i) = f\left(\sum_{j=0}^{j=n-1} T_{ij} V_j + I_i\right) \qquad (1)$$

Synaptic weights may be positive (excitatory) or negative (inhibitory) and any neuron may tend to turn any other neuron "on" or "off" respectively. I_i is a direct input that may be arbitrarily strong to force some value on V_i. The synaptic weights, determine the stable states and represent the information learned by the network. *Learning* is therefore, a controlled modification of the $\{T_{ij}\}$ to adjust the stable states. Recall or computation is performed as the network moves around the n - dimensional space defined by the neural states V_j, with the $\{T_{ij}\}$ constant.

The neural architecture is based on eqn. (1). It involves n^2 digital multiplications and summations in an array of n totally interconnected neurons. This is relatively straight forward in a network with fixed functionality. However, if the network is to be able to learn patterns, the synaptic weights must be programmable, thus making it more complicated.

NETWORK COMPUTATION AND DESIGN

An advantage of bit-serial arithmetic in a neural network is it minimises the interconnect requirement by eliminating multi-wire busses. Pipelining makes optimal use of the high bit-rates possible in serial systems allowing good communication within and between VLSI chips. The primary advantage of using digital CMOS circuitry is that on-chip digital memory design is more easy to implement than any analogue counterpart and can be easily incorporated for the programming and storage of the synaptic weights. Design techniques are advanced, automated and well understood, and noise immunity and computational speed can be high.

Architecture

The general neural architecture in figure 1 shows a single network of n totally interconnected neurons. A neuron is represented by a circle, with its column of n synapses (shown by squares) communicating with all other neurons in the array. Each synaptic operator *adds* the weighted contributions from other neurons down the column. When the total summation reaches the foot of the column, the neuron thresholds it according to the 5-state activation function shown in figure 2. The new state of the neuron is then signalled back to the array. The state signals are connected to a n bit bus running across the synaptic array, with a connection to a synaptic operator in every column. Therefore, the two functions of a synaptic operator are to multiply the signalling neuron state V_j, by the synaptic weight, T_{ij}, and to add the product to the running total of activity. For example, in figure 1, neuron 3 signals its state V_3, to neuron 1 along the dark path shown, and the product $T_{1,3}V_3$ is added to the running total in column 1.

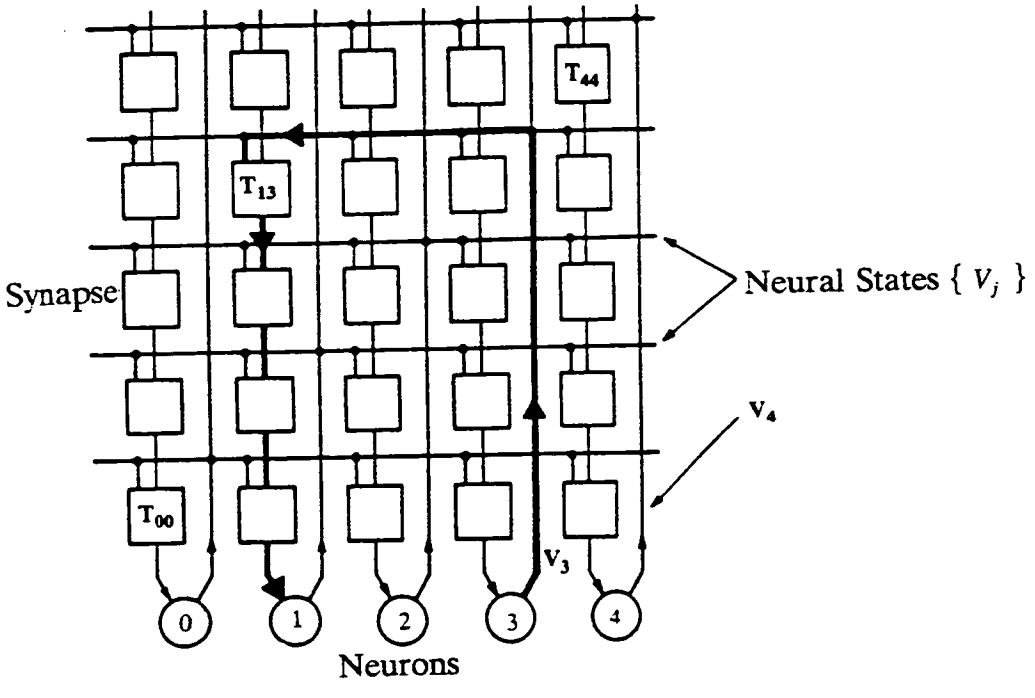


Figure 1 Generic Architecture for a totally interconnected n - neuron network.

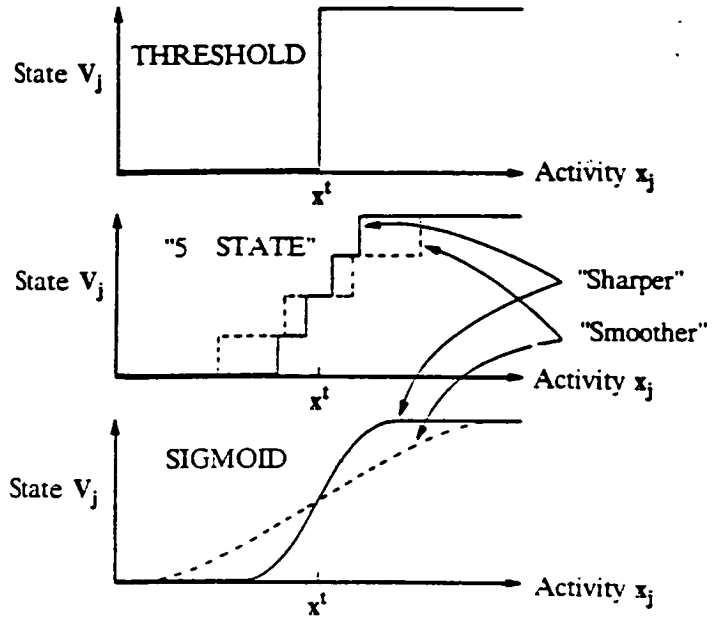


Figure 2 The 5-state, sigmoid and 2-state activation functions.

Reduced Arithmetic

Full digital multiplication can be expensive in silicon area, but the 5-state activation function allows reduced arithmetic to be used. Hence, multiplication of a synaptic weight by $V_j = 0.5$ simply requires the synaptic weight to be right-shifted by 1 bit. Likewise, multiplication by 0.25 involves two right-shifts of $\{ T_{ij} \}$, and multiplication by 0.0 is easy. A negative (inhibitory) neuron state is not problematic, as a switchable adder/subtractor is only slightly more complicated than than an adder. Hence, 5 neural states can be easily obtained from circuitry a little more complex than the simple adder required for 2 states (Hopfield, 1982). The neural state bus expands from a 1 bit to a 3 bit representation, where the 3 control bits are add/subtract?, shift? and multiply by zero?

Details of a synaptic operator are given in figure 3. Each operator has an 8 bit shift register memory holding its synaptic weight. During computation, the synaptic weight cycles round the register while the neural state is signalled on the 3 bit bus running horizontally above each synaptic row. A complete synapse computation requires two complete shift register cycles (16 clock cycles). During, the first cycle the synaptic weight is multiplied by the neural state and during the second, the MSBit of the resultant $T_{ij}V_j$ is sign-extended for the remainder of the shift register cycle. This allows a maximum 8 bit word growth in the running summation. The LSBit of each neuron's running summation is indicated by an LSBit signal running down the synaptic column.

The final 16 bit summation at the foot of the column is thresholded according to its activation function. As the neuron activity x_j, increases through threshold value x_t (figure 2), the ideal activation represents a smooth switch of neural state from -1 to +1. The 5-state "staircase" function gives a better approximation to this than the 2-state threshold function. Control of the sharpness of this transition can "tune" the neural dynamics for learning and computation. The control parameter is

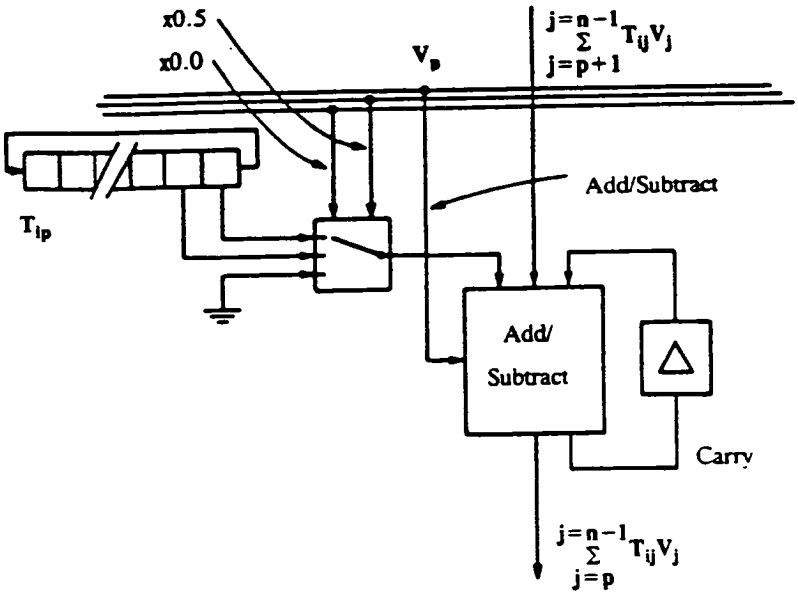


Figure 3 Synaptic Operator with a 5-state activation function.

referred to as temperature by analogy to statistical functions with this form. Higher temperatures give the staircase and sigmoid a lower gradient.

LEARNING AND RECALL OF THE ACTIVATION FUNCTIONS

Software simulations of learning and recall capabilities of the 5-state model were compared with those of the 2-state and sigmoid activation functions at varying temperatures with a restricted dynamic range for the synaptic weights. A 64 node network in each simulation attempted to learn 32 patterns using the delta rule algorithm (Rumelhart 1986). Results showed that the 5-state activation function learned the weight sets "better" than the 2-state activation function. The sigmoid activation was still superior to the 5-state, but the discrepancy was noticeably less than between the 5-state and the 2-state activations. The best method to deal with weight saturation during learning was to permit any weight outside the dynamic range to be set to its maximum value. A full discussion of these results can be found in Murray et al, 1987.

HARDWARE NEURAL BOARD

A 5-state synaptic operator array is being fabricated in 3μm CMOS technology. Full custom layout allowed a 12 x 9 synaptic array in a 64 pin package and figure 4 shows part of the design. Several chips, therefore, need to be wired together with memory ICs and control circuitry to achieve a suitable size network for simulations.

Neural Paging Architecture

A neural board has been designed with 4 synaptic chips wired together giving a 12 x

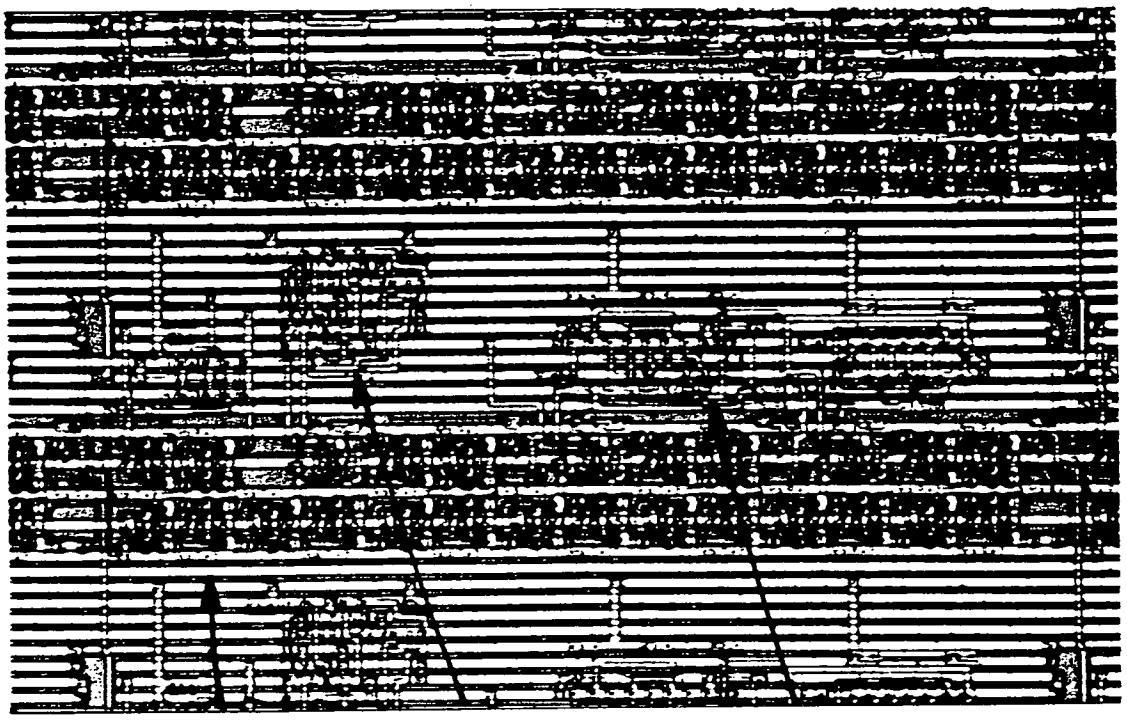


Figure 4 Silicon Layout of a Synapse in the Array.

9 synaptic array. The small array will be used in a *paging* architecture to give a network of 256 neurons that will act as a *neural accelerator* to a host computer. The *paging* architecture can be thought of as a "moving patch", where the small array or patch will simulate a small number of synapses in a large array, and then pass onto the adjacent patch to repeat the computation until all 256 synapses have been simulated. This idea is shown in figure 5. Each time the array is moved to represent another set of synapses, the weights for that patch must be loaded into it. For example, the first set of weights to be loaded will be $T_{1,1}...T_{1,12}...T_{2,1}...T_{2,12}...T_{9,1}$ to $T_{9,12}$, the second set to be loaded will be $T_{10,1}...T_{10,12}.....T_{18,1}$ to $T_{18,12}$. The final weight to be loaded is $T_{256,256}$ etc.. The memory required for 256 neurons is 0.5 Mbits of static RAM. A RAM speed of 70ns will allow the weights to be loaded in 9ms. A larger number of neurons can be simulated by simply loading the extra synaptic weights into more memory.

The "patch" will move down the 1st set of 12 columns to compute the complete running activities. It will then compute the 2nd set, 3rd set etc., until each set has been computed. For each "patch" simulation in the array, the emerging partial running summations of the 12 *partial* column blocks, are synchronised to coincide with the top of the running summation of the new patch. This ensures that each column has a contribution (excitatory or inhibitory) from each synapse. As the total summations occur for each block, they are stored in an on - board static RAM as indicated in the board design in figure 6.

When the total summation has been completed in each column, the neurons' activities are thresholded off - board according to the 5 - state activation function. The new neural states are signalled back to the synaptic accelerator chips for the next

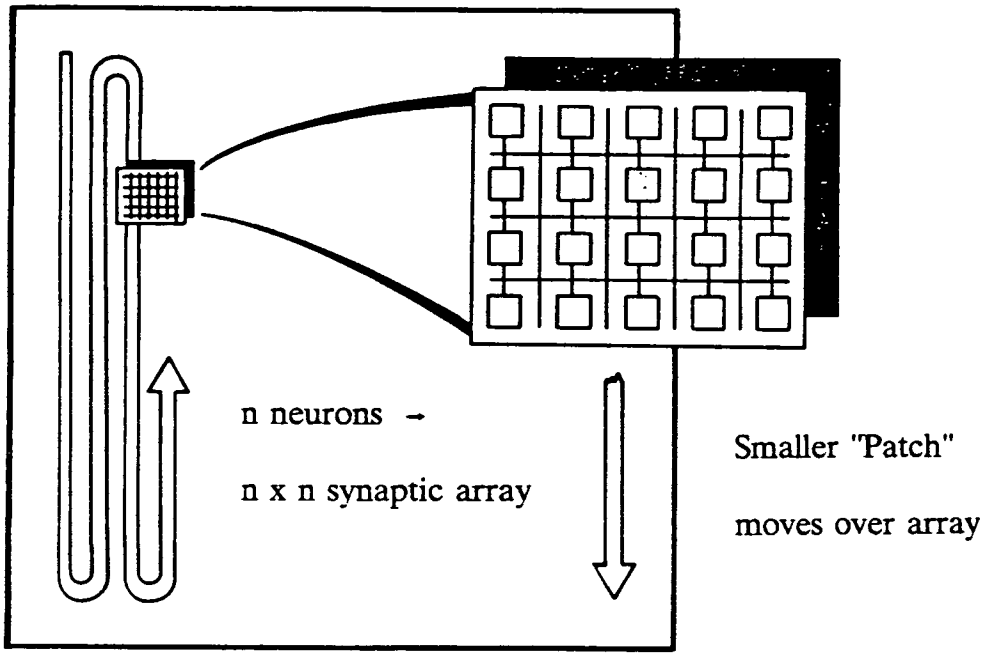


Figure 5 "Paging Architecture" of passing a small synaptic "patch" over a larger n x n synaptic array.

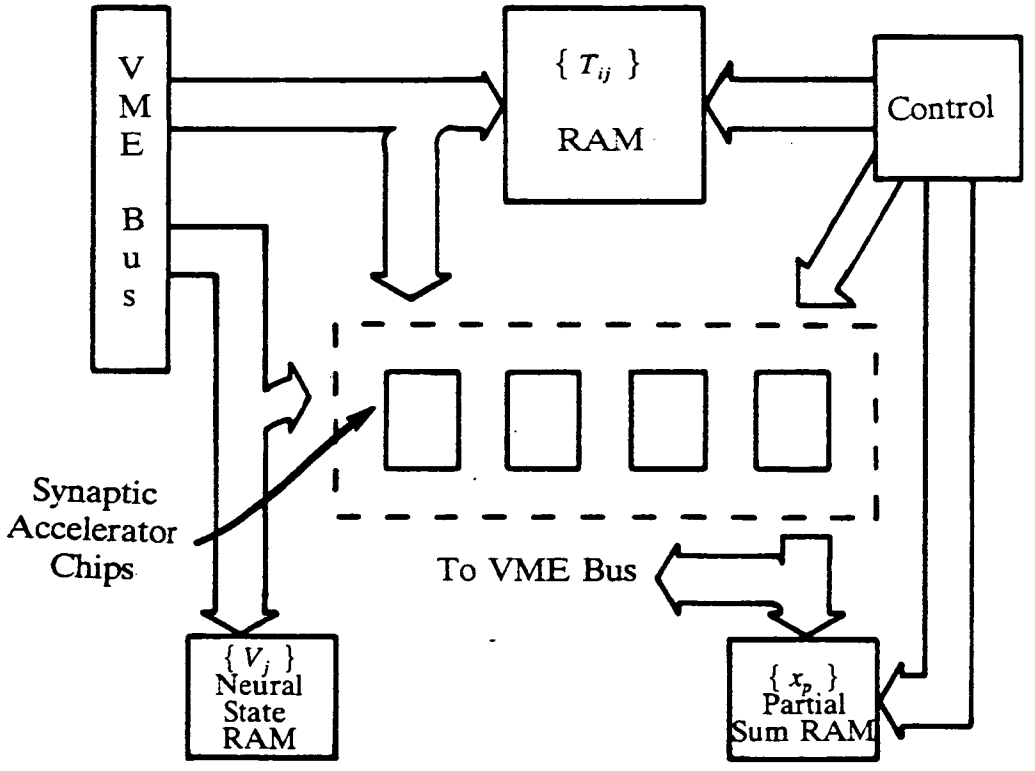


Figure 6 "Paging Architecture" for a Neural Network Board.

array computation. Once the states become stable, the synaptic weights are adjusted accordingly until learning is complete.

Control Circuitry

Microcode control circuitry operates all RAM loading and accessing and control signals to the synaptic accelerators. The flow diagram in figure 7 shows the small control overhead required, along with the timing of all operations for a complete update of 256 neurons. The calculated update time for the board is 1ms giving 6 x 10^7 operations/second. The number of synaptic accelerators determines the operating speed. A faster speed or more neurons and the same speed would require more accelerators. Hence, the design is versatile in that any specification for network size and speed can be met easily.

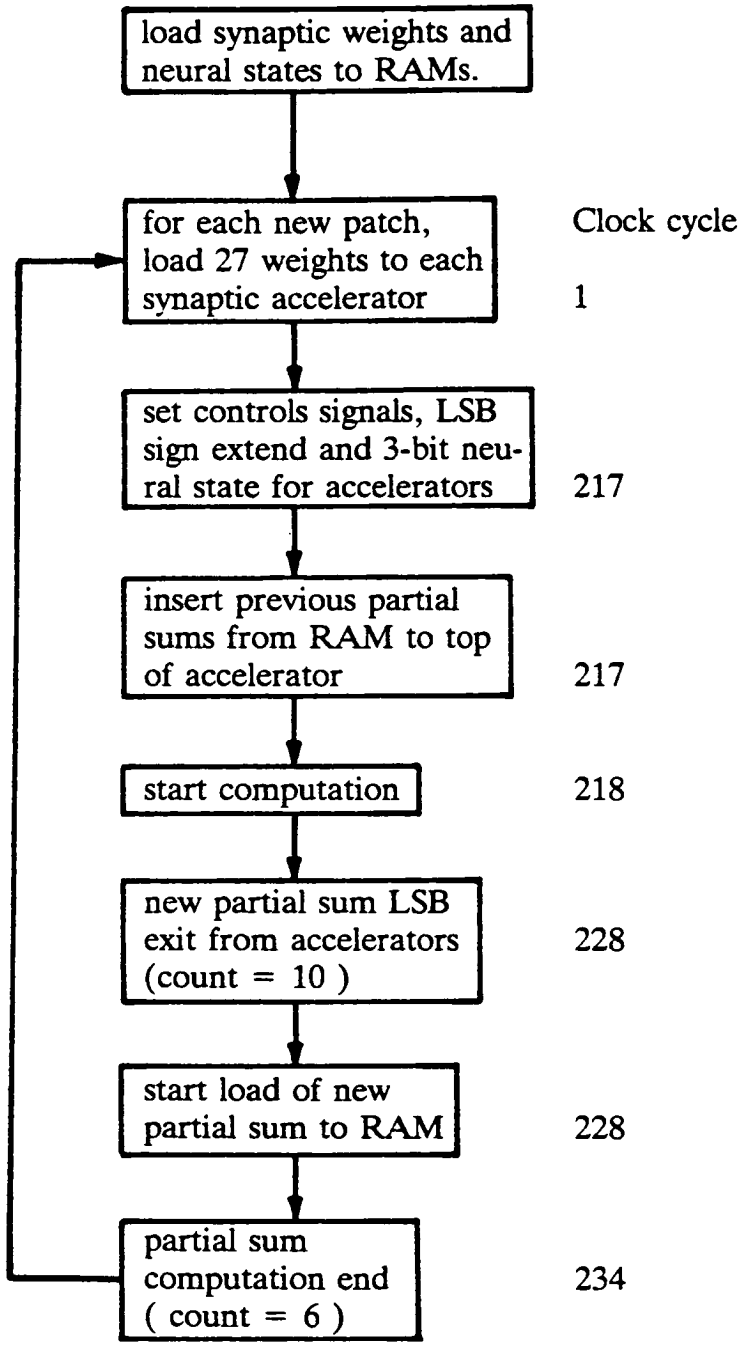


Figure 7 Flow Diagram of the Control Operation.

CONCLUSIONS

The design method has been given for the construction of a VLSI neural hardware accelerator and its implementation in a neural board. Bit-serial, reduced arithmetic improved the level of integration compared to more conventional digital, bit-parallel schemes. The restrictions on synaptic weight size and arithmetic precision by VLSI constraints have been examined and proved to be tolerable, using the associative memory problem as a test.

The digital design gives a good compromise between arithmetic accuracy and circuit complexity, but the level of integration is disappointingly low. This has been somewhat overcome by the paging architecture of the neural board to enable the simulation of a large number of neurons. It is our belief that, while digital approaches are useful in the medium term, especially as hardware accelerators, analogue techniques represent the best ultimate option in 2 - dimensional silicon.

The authors acknowledge the support of the Science and Engineering Research Council (UK) in the execution of this work.

References

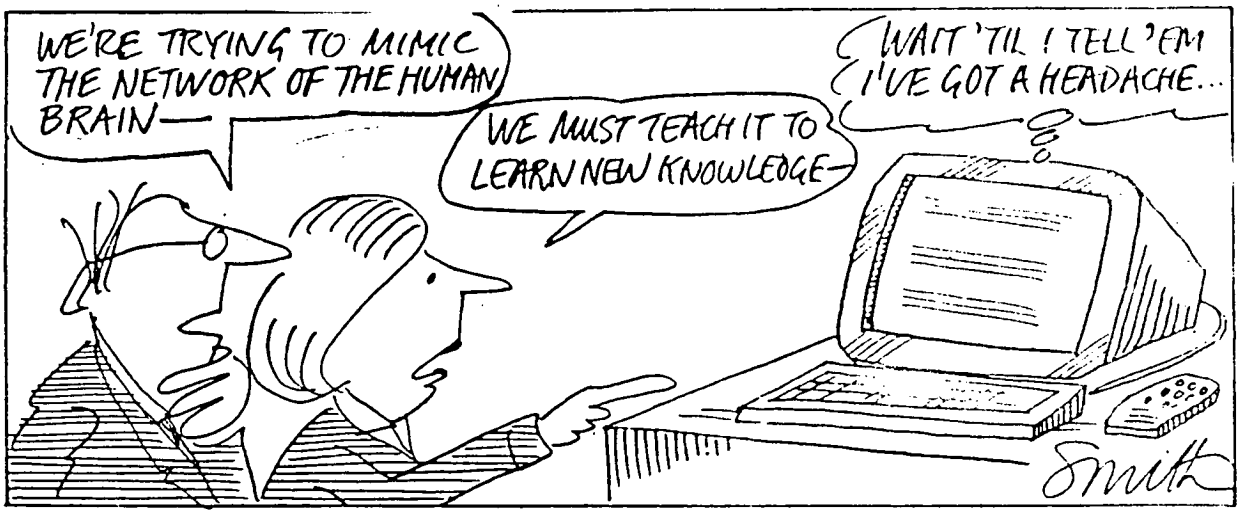
Hopfield, J. J., "Neural Networks with Emergent Collective Computational Abilities", *Proceedings of the National Academy of Science, USA*, vol. 79, pp. 2554-2558, 1982.

McGregor, M.S., Denyer, P.B. and Murray, A.F., "A Single - Phase Clocking Scheme for CMOS VLSI," *Advanced Research in VLSI : Proceedings of the 1987 Stanford Conference*, 1987.

Murray, A.F., Smith, A.V.W. and Butler, Z. F., "Bit-serial Neural Networks," *IEEE Conf. on Neural Infomation Processing Systems - Natural and Synthetic, Denver*, 1987.

Rumelhart, D.E., Hinton, G.E. and Williams, R.J., "Learning Internal Representations by Error Propagations", *Parallel Distributed Processing : Explorations in the Microstructure of Cognition*, vol. 1, pp. 318-362, 1986.


IT APPEARS TO BE BRAIN-DEAD — DO WE CALL IN A TECHNICIAN OR A NEUROLOGIST?
NEURO NETWORKS MANUAL
Smith


WE'RE TRYING TO MIMIC THE NETWORK OF THE HUMAN BRAIN—
WE MUST TEACH IT TO LEARN NEW KNOWLEDGE—
WAIT 'TIL I TELL 'EM I'VE GOT A HEADACHE...
Smith