AN INSTRUCTION SYSTOLIC ARRAY ARCHITECTURE FOR MULTIPLE NEURAL NETWORK TYPES

by Andrew Kane

September 1998

A doctoral thesis submitted in partial fulfilment of the requirements

for the Degree of Doctor of Philosophy in

Department of Computer Studies,

Loughborough University

Loughborough, Leicestershire, LE11 3TU, UK

To my darling wife

Hilary

for her love, support and encouragement

ABSTRACT

Modern electronic systems, especially sensor and imaging systems, are beginning to incorporate their own neural network subsystems. In order for these neural systems to learn in real-time they must be implemented using VLSI technology, with as much of the learning processes incorporated on-chip as is possible. The majority of current VLSI implementations literally implement a series of neural processing cells, which can be connected together in an arbitrary fashion. Many do not perform the entire neural learning process on-chip, instead relying on other external systems to carry out part of the computation requirements of the algorithm.

The work presented here utilises two dimensional instruction systolic arrays in an attempt to define a general neural architecture which is closer to the biological basis of neural networks - it is the synapses themselves, rather than the neurons, that have dedicated processing units. A unified architecture is described which can be programmed at the microcode level in order to facilitate the processing of multiple neural network types.

An essential part of neural network processing is the neuron activation function, which can range from a sequential algorithm to a discrete mathematical expression. The architecture presented can easily carry out the sequential functions, and introduces a fast method of mathematical approximation for the more complex functions. This can be evaluated on-chip, thus implementing the entire neural process within a single system.

VHDL circuit descriptions for the chip have been generated, and the systolic processing algorithms and associated microcode instruction set for three different neural paradigms have been designed. A software simulator of the architecture has been written, giving results for several common applications in the field.

Keywords: Systolic Array; Instruction Systolic Array; Systolic Algorithm Design; Parallel Processing; Computer Architecture; Neural Networks; Backpropagation; Kohonen; Counter-Propagation

ACKNOWLEDGEMENTS

I would like to acknowledge the supervision and guidance given to me over the last few years by Professor D.J .Evans of the Department of Computer Studies at Loughborough University. His advice over the subjects that I choose to pursue during my studies was both stimulating and sincere, showing time and time again that he wished for me to get the very best out of my research, as well as managing to keep me motivated and interested in my subjects in spite of any problems that arose over my time at Loughborough. He also managed to keep me writing-up at a reasonable rate once I had left University to enter the world of work; when your industrial work is so unrelated to your studies it is often difficult to keep going and it is a credit to Professor Evans that I finally managed to complete my studies.

I would like to thank Professor S. Jones of the Department of Electrical Engineering at Loughborough University who, through the help of his staff, allowed me access to the resources so that I was able to carry out the microelectronic circuit designs that were vital to this work. It is only a shame that the funding was never available to actually fabricate the device that I designed.

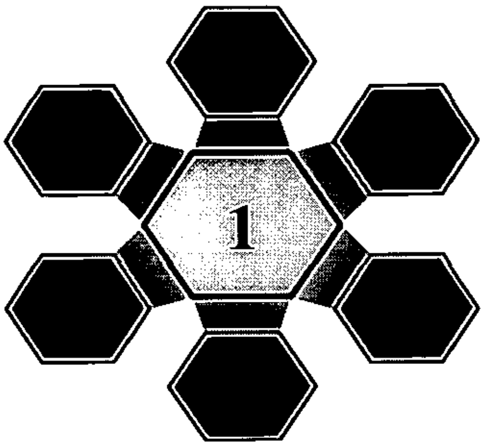
I would like to thank those at Jasmin Simtec Ltd in Bulwell, Nottingham, for taking me on board even though I had not completed my studies. Many people there have given me encouragement in completing, as well as lending me the odd resource (especially time) to help me complete my research. It has been almost 18 months since I started at Jasmin and the studies are at an end - I hope it was worth the wait for them!

Finally, I would like to thank my wife, Hilary, for putting up with this for so long. Spending the first year of married life with me always saying *"I'll just do another page or two"* was not exactly the start in life together that we both anticipated. Her support in this work was invaluable and I doubt that I could have done it without her. Now that it is complete I hope we can go on now and begin to enjoy our lives together.

CONTENTS

LIST OF FIGURES

LIST OF TABLES



INTRODUCTION

1.1 Introduction

Modern electronic systems, especially sensor and imaging systems, are beginning to incorporate their own neural network sub-systems. One of the major advantages of neural networks is that they have the capacity to extract the essential characteristics of input data, which may well contain much irrelevant data [Wass89]. This, coupled with their ability to generalize from previous examples to new ones, makes their use in real-time image-processing and sensor applications very attractive. However, the response times of such applications necessitate that the neural systems be implemented in hardware rather than software, and in order to maximize their performance application-specific integrated circuits (ASIC) would have to be designed rather than use standard general-purpose components [MeCo80].

In many existing hardware neural network systems [Nayl94] [KuHw89] [Hamm91] the processing elements (PE) within an ASIC perform the majority of the neural processing, with each PE being mapped on to a physical neuron in the network. With a large network there can be a tremendous number of communication channels, due to the number of connection combinations, and managing such channels can take up a large proportion of the available processing time. This can make the hardware implementation of neural networks either rather slow or limited in scope, with the former being especially true if the ASIC is to be 'general purpose' and allow conceivably any inter-neuron connection strategy.

There is little work [Lehm93] which focuses on using the synapses within a neuron as the main PE's of a hardware system; such systems compute partial results on each of the synapses and combine them all on a parallel pipeline. This is a perfect example as to the benefits of using systolic array structures: they can utilize the inherent parallelism present in neural networks for very few overheads and, thus, solve the communication problem through the extensive use of concurrency [KungHT82].

Very large scale integration (VLSI) allows for many millions of transistors per IC. Although such large devices can be expensive to fabricate they actually make array processors financially viable, as the single IC could hold a large number of individual PE's. A systems architect can then re-cast basic algorithms into a form more suitable for processing, as well as partitioning the algorithm in new ways in order to reduce the amount of working memory required on-chip. This allows for a single IC to have high performance even though it consists of multiple instances of cheap low performance units - there can often be an advantage in using inexpensive technology in seeking high performance.

Algorithms in many areas are well understood and are not expected to be improved upon in the near future; e.g. the fields of mathematical floating-point division and data sorting. However, the systems architect may be limited in computational units and may have to implement algorithms in a non-standard fashion. For instance, if the VLSI device has only fixed-point adders and multipliers how can complex expressions such as $x = ln(n+1)^2$ be calculated? The architect is forced to re-engineer such expressions, devising equivalent functions based upon the mathematical units available. These equivalent functions will be optimised for the parallel architecture that has been designed and will often be faster and more efficiently than the original expression would have been on a more mathematically-capable serial processor.

The work to devise a VLSI architecture suitable for real-time learning of neural networks can be seen to be into three distinct areas; parallel architecture design, parallel algorithm design and mathematical approximation of functions. All areas need to be worked upon with efficiency in mind, as if one stage is inefficient then any gains from the other stages will be lost.

This introductory chapter begins with a brief overview of the evolution of computing machines, followed by the descriptions of the advances made in IC technology. The chapter concludes with a survey of parallel computer architectures and an outline of the structure of the thesis. This introductory chapter, therefore, sets the technological background from which the work presented in the remainder of the thesis was based upon.

1.2 Evolution of Computing Machines

1.2.1 What is a 'Computer' ?

There are two fairly standard definitions of the term 'computer' [OUP90]:

1. *a usually electronic device for storing and processing data (usually in binary form), according to instructions given to it in a variable program*

2. *a person who computes or makes calculations*

The latter definition has fallen out of use since the 1950's, when the former definition became the only one of everyday importance. Until this time the latter definition was a common term, especially within military circles, for people who could perform complex calculations with or without the use of mechanical aids.

The definition given for a computer being a data processing machine makes no preconceptions about the type of machine, nor does it attempt to qualify the type of data that it deals with - the computer could be electronic or mechanical by design, although mechanical computers have been made redundant through technological advances in electronics throughout this century. Computers tend to be either analogue or digital in nature - the latter operates on continuous data values whilst the former uses discrete data values.

Analogue computers can measure precisely the values of physical quantities, such as current and temperature, which makes their use in simulator equipment desirable. Digital computers cannot cope with exact numbers, save where only integer values are required, but their increased speed over their analogue counterparts makes up for their deficiency in accuracy. This lack of accuracy in digital computers has become a moot point, however, as they are accurate to approximately $\pm 1/2^{n-1}$, with $n \geq 64$ becoming commonplace [IEEE85].

Hybrid computers have recently been developed, which incorporate both analogue and digital components, in an attempt to make a computer both speedy and exact in it's calculations. An example of such a VLSI device is discussed further in section 4.1.2.

1.2.2 Historical Development of Computers

1.2.2.1 The Early Years

Computing machines have existed for many thousands of years [BeNe71], albeit in a rather primitive form. The simple abacus, consisting of a number of beads attached to rows of parallel wires embedded inside a rectangular frame, allowed for the rapid calculation of the four basic mathematical operations of addition, subtraction, multiplication and division. Indeed, it is a tribute to the engineering genius behind the abacus in that in the hands of a skilled operator it can still perform basic mathematics faster than a human equipped with a modern-day scientific calculator.

It wasn't until the 17th century that more ambitious calculating machines were built, with scientists such as Gottfried Liebniz and Blaise Pascal designing and building more and more complex machines. Pascal's first machine (1642) was an adding machine, whilst Liebniz' machine (circa 1671) could carry out the four basic arithmetic operations as well as extract square roots. Babbage's *difference engine* (1823) was the first machine to be able to carry out a multi-step calculation; i.e. a calculation requiring more than a single operation. It was designed to calculate tables of functions, such as logarithms and trigonometric functions. In the 1830's Babbage conceived of a much more powerful computer, known as the *analytical* *engine*. It was designed to be a general purpose calculating machine that was theoretically capable of any mathematical operation known at the time; it was very unfortunate for the early development of computers that the analytical engine was never completed, mainly due to the inadequacies of the mechanical technology available at the time. Babbage's design included such innovations as a punched-card reader, printed output, a memory store for 1000 50-digit numbers and an innovative technique that has now come to be known as conditional branching.

The technological jump that came with the frantic scientific activity during World War II pushed forward the bounds of electro-mechanical computers. The first electronic digital computer, ENIAC (Electronic Numerical Integrator and Computer), in 1946, along with International Business Machine's (IBM) Harvard Mark I calculator, marked the beginning of the *first generation* of computer systems.

From this time the technology evolved at a rapid rate. The electronic relays that were used as switching devices in the 1940's became superseded in the 1950's by vacuum tubes. The development of the EDVAC (Electronic Discrete Variable Automatic Computer) in 1950 heralded the first stored-program computer, which marked the beginning of the use of a machine *operating system* to help programmers carry out some tasks.

Despite this development of a software programming environment the costs of the hardware were still predominant, with only rich Universities or large corporations, most notably IBM, being able to afford the development and purchase costs. These costs seemed so prohibitive that IBM issued the now-legendary statement that it believed the world industrial market for computers would be no more than 10 machines. With the development of the transistor by Bell Laboratories this idea changed drastically, and their invention ushered in the *second generation* of computers.

TRADIC (Transistorised Digital Computer) was built by Bell Laboratories in 1954, using 800 transistors in it's construction. Compared to it's predecessor, the vacuum tube, the transistor was more reliable and less power-hungry, as well as being a magnitude smaller; the transistor-based computers also did not require a team of engineers solely to replace blown switching devices within the computer. Printed circuit boards soon followed from the introduction of TRADIC, as did the development of magnetic core memory, both of which subsequently appeared many machines.

The operating system environments available were also enriched greatly; instead of programming the computers in assembly languages three *high level languages* were developed, namely Fortran, Algol and Cobol. These languages greatly improved program design, allowing programmers access to many high-level concepts such as data structures and allowed for the automatic handling of multi-byte variables, both of which were awkward to implement in machine code.

There were also two major advancements in architecture design [HwBr84], with the Larc system from Sperry Rand and the IBM Stretch project. The Larc contained a dedicated I/O processor, which operated in a parallel fashion with one or two other processing units. The Stretch featured instruction lookahead, whereby the execution of the instruction stream is pipelined by overlapping the execution of the current instruction with the fetch/decode/operand-fetch of the subsequent instruction. This breakthrough was so important that it has been implemented in virtually every computer built since this time; this topic is discussed in more detail in section 1.4.2.1.

1.2.2.2 Pre-VLSI Integrated Circuits

The *third generation* of computers began in the early 1960's, and was marked by the increasing use of IC's in the manufacturing process. Many transistors were fabricated onto a single device, with many hundreds of transistors being integrated together into a single 'chip' package. Small- and medium-scale integration (SSI and MSI) computers were capable of operating at a relative high speed and also had a relatively low cost, and these were the first computers to be termed *mainframe*.

Program execution in second generation machines tended to be in batch mode, whereby many programs were executed sequentially until they were completed. Although very popular it became virtually redundant with the introduction of multi-programming, whereby many program segments could be executed simultaneously by interleaving the I/O operations. Soon after this the concept of the time-sharing operating system was born, which increased the real-time multi-user aspect of computers. Note that batch programming has not disappeared completely, with Digital Equipment Corporation (DEC) still supporting it in their VAX family of computers.

Progress was made in the area of memory systems, eventually replacing the core memory with solid state devices. Programming languages also improved, with the introduction of more intelligent compilers to make the task of programming less error-prone. They were also extended so that they had the capability to handle both scalar and vector data, so that languages such as Fortran could be run efficiently in the vector-process machines that began to appear in the early 1970's, which is when the *fourth generation* of computers came into being.

Around the early 1970's large-scale integration (LSI) began to take off, with LSI being used for both logic and memory systems. High-density packaging allowed many thousands of transistors to be in a single chip, heralding the introduction of the *microprocessor*, often termed the 'computer on a chip'. Powerful vectorising compilers appeared for the next generation of vector machines, such as the Cray-1 (1976). Multi-processor mainframe systems also started to appear, such as the Univac 1100/80 (1976) and IBM 360/168 MP (1980).

It was this multi-processing capability, coupled with increased pipelining of program execution and the reduction of price-power ratio, that made computers a commercial success. Without the massive take-up in the data processing industries the funding for further research and development would not have been available, as computers had long passed the stage where the manufacturers commercial customers would accept whatever they were offered. The computing industry at this stage had become demand-led, in that boundaries of technology were being pushed forward because the manufacturers believed that there was a market ready and waiting for their new products.

1.2.2.3 VLSI and Beyond

Microchips were becoming far more complex in the 1980's. The metal and silicon tracks on the chips were still laid out by hand, despite a computer being used. There were no automated layout tools of any worth and the time taken to hand-craft a 1/4 million transistor chip was just too much. In order to increase the size and complexity of computer chips some fundamental changes had to be made in the area of computer-aided design (CAD).

There seems to be two main thrusts to VLSI design. Mead and Conway [MeCo80] advocated what is now commonly known as *structured hierarchial design*, accompanied by a reduced and simplified geometric and electrical rule set. The responsibility on the designer for layout extended right down to full-custom details. The other method, which is largely supported by industry rather than by academia, has placed ASIC design responsibilities solely at the logic level, using gate arrays and standard library cells. This semi-custom method does get devices built fairly quickly, but in order to maximize the potential of the silicon surface the designer needs to do full-custom layouts.

By utilising this structured design method, which is discussed at various levels throughout section 2, the designer can generate chip designs holding (at current levels) up to 8×10^6 transistors. The chip can have built-in self-test capabilities or external connections to other test hardware. The chip can be tested by software simulation through complex CAD packages so that the designer can be very sure of the chances of the chip working upon fabrication. Transistors can be very densely packed due to designing compact bit-slices of functional units and then replicating those slices to the required bit-widths. VLSI in itself is not new technology, but computers created using VLSI design techniques are often known as *fifth generation* computers.

The near future of computers seems to be in extracting the maximum performance from available technology. Chips may begin to be fabricated using gallium-arsenide instead of silicon [Hira90], which will give them the capability of operating at a speed an order of magnitude greater than silicon-based chips. Pipelining and multi-processing will become common even on single-user workstations, allowing them to run processes that would normally require a mainframe computer. Increased techniques at the molecular level, such as using transistors that can be switched with only very few electrons will reduce the physical area required by circuitry. With increasing advances in the field of computer artificial intelligence (AI) it is possible then computers will have the capacity for reasoning above and beyond that programmed into them [RiKn91]; such capabilities will allow computers to adapt to their environment as well as to be able to solve the 'hard' mundane area's of computation (from a person's point of view) such as perception, natural language understanding (both generation and translation) and commonsense reasoning.

However, as section 1.3 illustrates, there seems to be a visible finishing point for the development of silicon or gallium-arsenide chips, as the scaling factors inherent in VLSI design when the features upon the chip become smaller are seen to be limiting factors. A new fundamental breakthrough is required in the next 15 to 20 years, perhaps in optical computing or in physical chemistry and biology, in order to keep the pace of change at the levels that are prevalent today.

1.3 Evolution of VLSI Technology

1.3.1 Development of Integrated Circuit Technology

Integrated circuit technology has advanced greatly since the introduction of the first transistor in the late 1940's by Bell Laboratories. The first device was implemented in germanium, and efforts to implement a transistor in silicon came shortly afterwards. Silicon had advantageous properties over germanium, mostly concerned with cost and switching speed, and a successful implementation was made in the mid-1950's. The dimensions of a transistor fabricated in silicon, known as the *feature size*, was around 37μm; the advancement of VLSI technology now gives us an attainable feature size of 0.25μm, with sizes of 0.18μm and lower currently under development [Camp96] and due for general release in the very near future [TI98].

The so-called *generations* of IC technology (SSI, MSI and LSI) were used to approximately designate the complexity of the IC concerned; as IC fabrication technology developed it became possible to reduce the size of the silicon transistors and, therefore, fit more devices on to a single IC. Table 1.1 shows the approximate bounds of these IC generations; it should be noted that although the figures themselves are not universally accepted the idea of banding the IC generations in this way is deemed acceptable [Burg84]. Complexity was also increasing at a rapid rate; in the mid-1970's it was calculated that IC complexity doubled approximately every 18-24 months [Moor75]. This progression is known as 'Moore's Law' and still holds fairly true today, although the rate of progression is reducing.

Table 1.1 IC Complexity

Generation	Complexity	
SSI	2^6	(64)
MSI	2^{11}	(2048)
LSI	2^{16}	(65,536)

VLSI technology was developed towards the end of the 1970's when it became apparent that existing design techniques were lacking - it was becoming extremely difficult to design large chips (> 150,000 transistors), as computer-based testing of circuits was far from acceptable. The first real text on structured hierarchial design, Mead and Conway's *Introduction to VLSI Systems* [MeCo80], required a new angle on the design of IC's that would grab the attention of designers - the term 'VLSI' was born. At that time it was taking teams of industrial designers several man-years to finish chip designs, but Mead and Conway's methods were focussed on helping teams of non-experts do almost as well - obviously, new design techniques were required.

The impact in the success of using VLSI techniques are very apparent when you compare two microprocessor's that were developed around the same period: the Intel 8086 [Uffe87] and the Motorola 68000 [Whit84]. The differences in style in the two designs was very pronounced; the Intel device did not have the benefits of VLSI design techniques and, thus, it's silicon design looks not unlike spaghetti, with the interconnect between functional units seemingly placed on the silicon wherever it would fit. The Motorola device had several very regular structures, with the interconnects between function blocks being very ordered, and these units relied heavily on iterative design. There is heavy use of iteration in the Motorola device, whereby functional units are made up from multiple instances of identical cells. For instance, the ALU may be 16-bits wide, but only a single 1-bit cell would have been heavily designed and tested; the designer would simply have cascaded 16 instances of the 1-bit cell to create the full-blown 16-bit functional unit.

This method of design has several advantages, as it allows large structures to be designed quickly; if the ALU specification was suddenly changed to 32-bits the Motorola designer would have very little extra work to do (save ensure that the expanded circuit still fits on the available silicon area). The Intel designer would have a lot of work to do; the additional circuitry would have to be added to the device and then extensively tested in software to ensure that some small feature had not been left out.

From the early 1980's every IC manufacturer began to adopt these VLSI structured design techniques, as the benefits they brought was well worth the cost of re-training their designers. As fabrication technology became more sophisticated, with smaller transistor sizes, lower activation voltages and intelligent power usage, designers using VLSI techniques were able to benefit from these gains very quickly. With the imposed rigidity of structured design techniques, along with advances in pre-fabrication design testing, VLSI can help realise chip designs of over 2^{22} (4 million) transistors - today's chip designers are already planning devices of over 2^{23} (8 million) transistors using VLSI techniques.

Further, there is a design methodology known as Ultra Large Scale Integration (ULSI), which is often taken to be 2^5 times the range of VLSI [Burg84] [Shut88]. This, like VLSI before it, is not a technology in itself, but rather a means of identifying devices at the top end of the VLSI devices currently under design. ULSI is not referred to again in this thesis.

The next generation of IC production seems likely to be Wafer Scale Integration (WSI). In this technology the entire silicon wafer is used to construct a single device, with a large number of processing elements fabricated onto the wafer and then connected together. Wafer-scale fabrication of computing structures demands a high yield at fabrication, otherwise the wafer will be full of non-functional units. Figure 1.1 [HeSn82] shows a 4-by-3 lattice of PE's, each interconnected with a number of others by a series of switches. Once fabricated all PE's on a wafer are tested, and then the good PE's are connected together. The wafer is structured so that the presence of the faulty PE's is masked off and only the functional PE's are used.

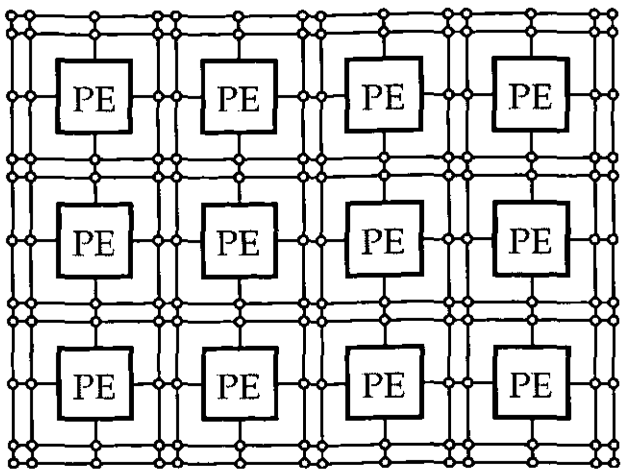


Figure 1.1 Wafer-Scale Integration Lattice

It is currently feasible to fabricate over 300 processing elements per 6-inch wafer. These wafer-scale computers have the capability to be cheaper, faster and more reliable than their counterparts implemented with single-chip components. There remain many problems with WSI, most notably concerned with the practical problems of testing the individual PE's upon the wafer. Other problems, such as the difficulties of routing around faulty PE's and the potential of very high power consumption upon the wafer, still remain to be solved.

With the further development of fabrication technology wafer-scale integration could prove to be the way forward in the design and construction of massive parallel computing structures. Although there are many problems associated with the technology, progress is always being made and this technology looks set to become very powerful in the near future.

1.3.2 VLSI Scalability Constraints

One of the major advantages of using VLSI technology and the related design methodologies is that the basic MOS transistor is inherently scalable [MeCo80] and some simple rules exist to help calculate the effect of scaling on certain fabrication process parameters. By scaling the technology down to a smaller feature size the switching speed of the gates on the device is also reduced, as is the activation voltage required. This enables the VLSI chips to be run at higher and higher clock speeds.

However, although it looks as if there is the possibility of scaling the feature size down to nano-technology levels there are some severe limitations to the technique that could limit the future development of VLSI; system designers are in need of new techniques.

1.3.2.1 Scaling of MOS Transistor Dimensions

The *constant field* model of first-order MOS (metal oxide silicon) scaling theory proposes [Denn73] [Denn74] that the characteristics of a MOS device can be maintained, with the preservation of basic operational characteristics, if certain critical parameters of the device are scaled in accordance to a set scheme. This approach has shown to be very successful when scaling the minimum feature size from the range of 5μm...10μm to the range 1μm...3μm. Although this technique does not give optimal device performance at small dimensions, it is in itself very powerful in providing guidelines as to the improvements that can be expected when a fabrication process is scaled.

The scaling processes works by applying a dimensionless factor α to the following:

- all dimensions on the device
- device voltages
- substrate concentration densities

A schematic of a simple MOS transistor is shown in figure 1.2 [WeEs88]; a fuller description of the general make-up of a

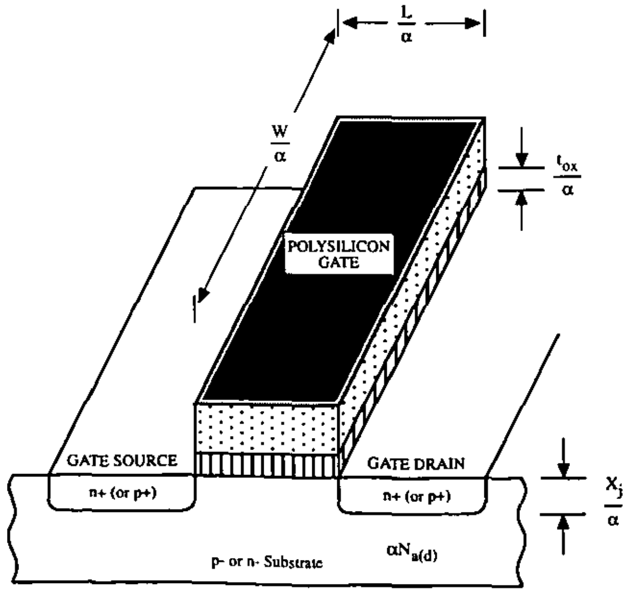


Figure 1.2 Scaled MOS device

MOS transistor, along with the methods employed to fabricate them, is given later in section 2.1. The basic parameters are length (L), width (W), gate oxide thickness (t_{ox}), junction depth (X_j) and substrate doping level (N_a or N_d). Other factors not shown, such as depletion layer thickness (d), transistor threshold voltage (V_t) and drain-to-source current (I_{ds}), are also scaled as a result of the device dimensions being scaled.

The various influences of the α scaling factor are shown in table 1.2. Initially interesting factors are that as the voltage is scaled the electric field (E) in the device remains constant; this has the highly desirable effect that many non-linear functions remain essentially unchanged. As the device dimensions are scaled by 1/α the circuit density scales up by α^2, which results in the current density scaling upwards linearly with α. To cope with this the metal conductors will have to be wider to supply the more densely packed structures. Also, the reduction in the size of the gate oxide thickness (t_{ox}) requires the fabrication process to provide thinner oxides with comparable yields to conventional oxide thicknesses - this is not a trivial problem, and often necessitates re-design of areas of a fabrication plant in order to cope with the new technology.

Table 1.2 Influence of first-order MOS scaling

	PARAMETERS	SCALING FACTOR
DEVICE PARAMETERS	Length: L	$1/\alpha$
	Width: W	$1/\alpha$
	Gate oxide thickness: t_{ox}	$1/\alpha$
	Junction depth: X_j	$1/\alpha$
	Substrate doping: $N_{a\,(\text{or } d)}$	α
	Supply voltage: V_{DD}	$1/\alpha$
	Electric field across gate oxide: E	1
	Depletion layer thickness: d	$1/\alpha$
	Gate delay: (VC/I)	$1/\alpha$
RESULTANT INFLUENCE	DC power dissipation: P_s	$1/\alpha^2$
	Dynamic power dissipation: P_d	$1/\alpha^2$
	Gate area	$1/\alpha^2$
	Power density	1
	Current density	α
	Transconductance: g_m	1

Another characteristic from table 1.2 is that although static and dynamic power dissipation, P_s and P_d, are both scaled by $1/\alpha^2$ the actual amount of power dissipated by the MOS device remains constant, as the circuit density has increased by α^2. As temperature increases the gain of transistors is reduced (due to a reduction of electron carrier mobility) which causes the speed of the circuits to fall. High temperature, high speed circuits require special consideration during the design phase.

The limitation of first-order scaling is that it is a first-order approximation; a more rigorous analysis would provide slightly different values from those shown in table 1.2. It also gives the wrong impression that it is possible to scale to virtually zero dimension or zero threshold voltage. If circuit concentrations reach high figures (larger than 1×10^{19} cm^{-3}) the gate oxide actually breaks down before the transistor channel is formed during fabrication.

Although scaling does reduce the device switching speeds, power consumption per transistor and required input voltages, scaling MOS devices cannot go on forever. There are definite limits on the extent of scaling, but the end of scaling silicon technology is not yet with us; there is still scope for improvement.

1.3.2.2 Scaling of Interconnect Layer

Despite the obvious advantages of first-order scaling it does have a number of undesirable effects; voltage drop, line response and current density all exhibit significant degradation, although in the first two of these the degradation is not entirely apparent; the scaling factors appropriate to interconnect media are shown in table 1.3.

Table 1.3 Scaling of interconnect media

PARAMETERS	SCALING FACTOR
Line resistance: R	α
Line response: RC	1
Normalized line response	α
Line voltage drop: V_d	1
Normalized line voltage drop	α
Current density: J	α
Normalized contact voltage drop: V_c/V	α^2

Both V_d and RC remain constant after first-order scaling. However, with a constant chip size the lengths of signal paths across the chip, as a general rule, do not scale down. This gives the result that voltage drops along communication paths are larger by a factor of α with respect to the scaled voltages. In a similar manner the line response time normalized to the scaled line response is also larger by a factor of α.

Because of these problems it is difficult for the designer to take the maximum advantage of the higher gate switching speeds made available through first-order scaling when signals are required to propagate over lengthy paths. This creates a major problem in the effective distribution of clocking signals.

The fine-line metallisation brought about by first-order scaling also presents it's own problems. As the current density increases then the metal lines within the MOS device must carry a higher current. Because of this electron migration becomes a major problem, whereby the strong current literally chips atoms of aluminium off the metal track (thus increasing the resistance of the track and exacerbating the problem), so new metallisation schemes are required to accommodate the higher current densities that scaling brings.

Also, as the MOS device becomes more densely packed the average line length on a chip tends to increase. Unfortunately, the power dissipation of the scaled gates also decreases with scaling, which makes it harder for them to drive enough power down the wiring, which has a constant resistance through scaling. This implies that as gates are scaled down it is the capacitance on the interconnect that determines gate delay times rather than the gate itself. A good approximation of the maximum length of acceptable interconnect, from a statistical point of view [Keye79], is given by the expression $L_{max} = \frac{\sqrt{A}}{2}$, where A represents the overall silicon area of the chip. Therefore, if scaling occurs and the silicon area increases as well, as is often the case, then the interconnection capacitance problem is exacerbated. Due to this problem designers now use techniques so as to maximise local and minimise global connections in order to design effective VLSI circuits. In certain areas of VLSI design, such as the design of high-speed mathematical units, this optimal use of local connections in order to reduce overall circuit computation time has always been of paramount importance; the techniques used in these specialist areas are bound to spread to all other areas of VLSI design.

1.4 Parallel Computer Architectures

Many areas of science and technology constantly demand more and more powerful computational capabilities. Fields requiring vast amounts of processing power include weather prediction, molecular modelling and astrophysics. Although VLSI technology is getting faster and faster through the use of scaling techniques, scaling alone cannot meet the ever increasing computational requirements of certain scientific fields.

The use of parallel processing in such areas is a sound choice for further improving computer performance. Hwang and Briggs [HwBr84] formally define parallel processing as follows:

> **Definition** *Parallel processing is an efficient form of information processing which emphasizes the exploitation of concurrent events in the computing process. Concurrency implies parallelism, simultaneity and pipelining. Parallel events may occur in multiple resources during the same time interval; simultaneous events may occur at the same time instant; and pipelined events may occur in overlapped time spans. These concurrent events are attainable in a computer system at various processing levels. Parallel processing demands concurrent execution of many programs in the computer. It is in contrast to sequential processing. It is a cost-effective means to improve system performance through concurrent activities in the computer.*

Any computationally expensive field of science that is offered computing power as defined by Hwang and Briggs are bound to accept it. This section describes the various classes of parallel architectures that have evolved over the last three decades.

1.4.1 Historical Legacies

1.4.1.1 Flynn's Classification

Flynn's classifications of computer organizations [Flyn66] divided the basic architecture types up into four categories, depending on the multiplicity of instruction and data streams. An instruction stream is a sequence of instructions as executed by a machine; a data stream is a sequence of data called for by an instruction stream. The data can consist of input, partial results or completed calculations. The four categories are shown in figure 1.3.

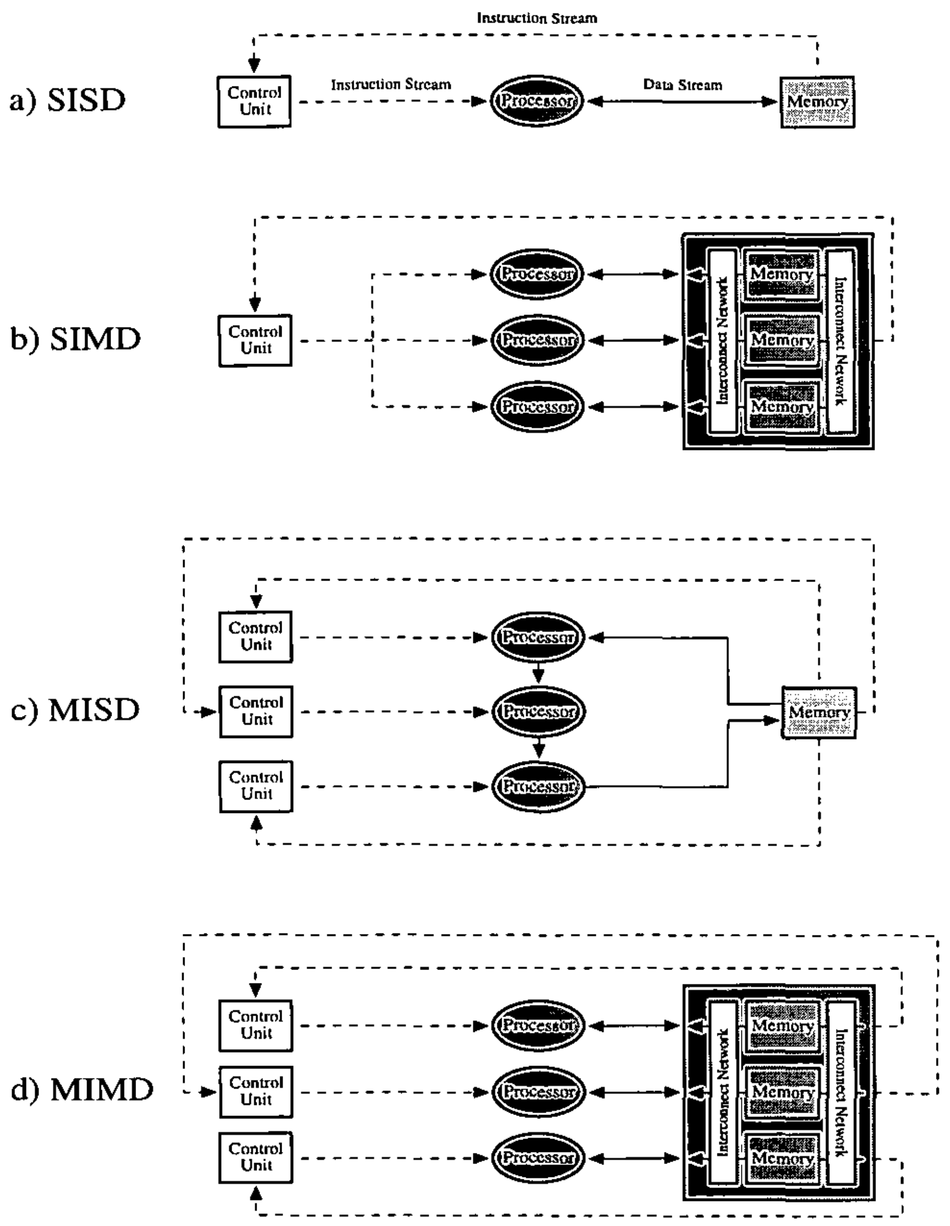


Figure 1.3 Flynn's Classification of Parallel Computer Architecture

The following list describes Flynn's four machine definitions:

- **Single Instruction Single Data** (SISD) machines represent the majority of serial computers available today. Instructions are sequential, but may be overlapped at the execution stage through pipelining, as shown in figure 1.3a

- **Single Instruction Multiple Data** (SIMD) machines correspond to array processors, with multiple processors simultaneously executing the same instruction on different data. The multiple processors are supervised by a single control unit, as shown in figure 1.3b

- **Multiple Instruction Single Data** (MISD) machines contain multiple processors, each receiving different instructions, operating on a single data stream and it's derivatives - the output of one processor becomes the input to the next. A machine of this type is possible but no real examples of it's type exist, which Flynn attributes to the impractical architecture and to the specialized streamlining of data that is required. The schematic is shown in figure 1.3c

- **Multiple Instruction Multiple Data** (MIMD) machines contain multiple independently operating processors attached to a shared memory system, with each processor being able to execute distinct instructions upon distinct data as compared to the other processors. The schematic is shown in figure 1.3d

Flynn's classifications have stood the test of time very well. Although it is useful in getting a rough classification for an architecture, however, it is not flexible enough to be able to classify many of today's newer parallel architectures: vector processors, shared memory machines, distributed architectures, systolic arrays or wavefront arrays are all examples of these. In order to classify these architectures correctly we need to use a different methodology.

1.4.1.2 Feng's Classifications

Realising that there were some problems associated with Flynn's classifications Tse-yun Feng suggested [Feng77] that in order to classify parallel architectures it would be better to use the degree of parallelism inherent within them.

Feng's ideas are based around calculating the maximum degree of parallelism possible within a computer system (P), then further calculating the processor utilisation rate (μ) over a series of processor clock cycles (T). P is defined as being the maximum number of binary bits that the computer system can process at any one time. With P_i being the processor utilisation rate at clock cycle i, the average degree of parallelism, P_{avg}, over a number of processor cycles can be given by:

$$P_{avg} = \frac{\sum_{i=1}^{T} P_i}{T} \tag{1.1}$$

It is clear that the utilisation rate of a computer system is dependant on the application being executed at the time, so if the processor is being fully utilised then we have $P_i = P$ for all i and $\mu = 1$. It is clear that in general $P_i \leq P$; thus, we define the utilisation rate μ as:

$$\mu = \frac{P_{avg}}{P} \tag{1.2}$$

The actual classification method takes the word-length of the computer (n) and the bit-slice length (m). A bit-slice is a string of bits, one from each of the words of data that the processor is holding/processing in any of it's arithmetic and computational pipelines. For example, the Texas Instruments Advanced Scientific Computer (TI-ASC) from 1972 has a word length of 64-bits and four separate pipelines, with each pipe having eight pipeline stages. Thus, the bit-slice length m is given by 4 x 8 = 32. These values are represented as (64, 32), and the maximum degree of parallelism possible is represented by the product of the word-length n by the bit-slice length m.

From Feng's classification guidelines there are four types of processing methodologies:

- **Word-Serial / Bit-Serial** *(WSBS; $n=1$, $m=1$)* This method has often been called *bit-serial processing*, because only one-bit at any time is being processed. This slow process was only done in the first generation machines
- **Word-Parallel / Bit-Serial** *(WPBS; $n=1$, $m>1$)* This is known as *bit-slice processing*, as a single m-bit slice is processed at a time

- **Word-Serial / Bit-Parallel** *(WSBP; n>1, m=1)* Also known as *word-slice processing*, because a single *n*-bit word is processed at a time

- **Word-Parallel / Bit-Parallel** *(WPBP; n>1, m>1)* This is the fastest processing mode of the four, also known as *fully parallel processing* (as opposed to simply *parallel processing* in order to reduce confusion). In this mode an array of n-by-m bits are processed simultaneously

Table 1.4 gives a number of examples of computer systems that were classified using Feng's methods. Different processing needs will require a computer in a different category. Typical small computers of today are of the WSBP type, with a single *n*-bit word being processed at a time. Technology at even this level is changing fast [Taba95] with the advent of processors such as the latest incarnations of Motorola's PowerPC and Intel's Pentium architectures, which are becoming WPBP - as what has happened in the past, technology that was previously used solely for super-computers is now becoming standard at the lower level of personal computers.

Table 1.4 Feng's Computer System Classifications

MODE	COMPUTER SYSTEM	(n, m)
WSPS	MINIMA	(1, 1)
WPBS	Goodyear STARAN	(1, 256)
n=1, m>1	ICL DAP	(1, 4096)
(bit-slice processing)	Goodyear MPP	(1, 16384)
WSBP	DEC VAX 11/780	(32, 1)
n>1, m=1	IBM 370/168 UP	(64, 1)
(word-slice processing)	Cray-1	(64,1)
WPBP	Carnegie-Mellon C.mmp	(16, 16)
n>1, m>1	Texas Instruments ASC	(64, 32)
(fully parallel processing)	Illiac-IV	(64, 64)

1.4.2 High-Level Parallel Architecture Classification

Feng's classifications are very well suited to describing the efficiency of the parallel processing structures within a computer, but do little as to describing how they are carried out. Flynn's classifications provide useful distinctions between the types of computer processor organizations available, but are not very helpful when trying to classify many modern computers. Pipelined vector processors are difficult to accommodate within Flynn's categories, as they lack processors executing the same instruction (SIMD) and also lack the basic asynchronous behaviour of MIMD, yet it clearly is a parallel architecture operating upon multiple tuples of data in parallel.

Duncan proposed a rather different parallel framework [Dunc90] that attempts to place the recent architectural advancements into the broader context of parallel computers, in an attempt to classify all parallel computers using a single coherent framework. In order to do this Duncan's proposals had to satisfy the following set of imperatives, the results of which are shown in figure 1.4:

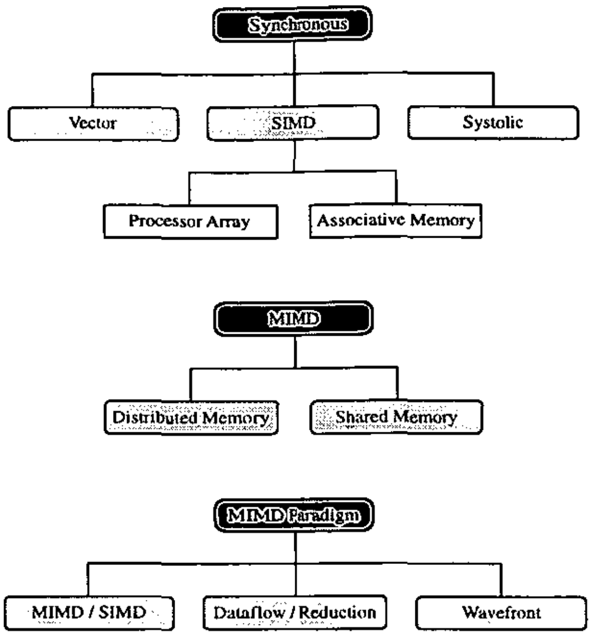


Figure 1.4 Duncan's Classifications

- Architectures that incorporate commonplace low-level parallel features are excluded

- The very useful elements of Flynn's classification, namely the instruction and data stream categorisations, are to be retained and used a sub-classifications

- Any architecture that intuitively merits being classed as a parallel architecture, but which fails to be categorised at all under Flynn's rules, is to be included

1.4.2.1 Synchronous Architectures

Pipelined Vector Processors

The basic idea of a vector processor is to combine two vectors, element by element, in order to produce an output vector. Thus, if A, B and C are vectors with N elements then the vector processor performs the operation $C := A .op. B$. Typical operations are those which implement arithmetic or boolean operators on vector data. A simplified schematic of the dataflow through a vector processor set up for pipelined vector multiplication is shown in figure 1.5.

Most vector computers have a pipelined structure; when one pipeline is not sufficient to achieve the desired performance then designer's occasionally provide multiple pipelines. These multi-pipeline vector processors not only support data streaming through a single pipeline, they also support a fully parallel mode of operation by allowing multiple pipelines to execute concurrently on independent streams of data.

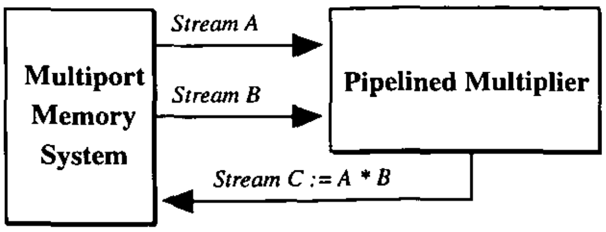


Figure 1.5 Vector Processor Dataflow

The pipeline theory is to divide an operation into several distinct stages, with the results from one stage forming the inputs to the next. The pipeline of stages may be linear or may contain feedback loops, in which case care must be taken as to when new data is provided to the pipeline lest it collide with partial results being fed back into previous stages. Figure 1.6 shows a linear pipeline for floating-pont multiplication (although a similar schema can be imagined for processor fetch/decode/execute cycles). Each stage is controlled by an internal clock, such that each stage is ready to accept new input data at each cycle. As some stages are complex they may be broken down into several other pipelined stages - the *Multiply Mantissas* stage can be broken down into a series of sequential operations, namely *produce partial products* and *add partial products*. All stages, or sub-stages, should be designed so that on each and every internal cycle they are ready to accept new data and output the result from the previous cycle. If every stage requires 25ns of processing time, and there are 10 stages of processing, then the first full result will be available after 250ns. However, as the input data is being fed into the pipeline every clock cycle the second and subsequent results are available after every additional 25ns.

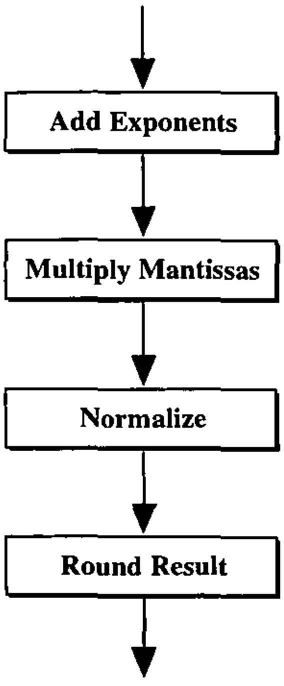


Figure 1.6 Multiplier Pipeline

SIMD Architectures

Under the SIMD architectures there are two sub-categories: *processor array architectures* and *associative memory architectures*. The former has been shown in figure 1.3b, and is simply a synchronous parallel computer with multiple PE's that operate in a lock-step fashion. For a short period of time only a single processor (the control unit) is active: it is busy initialising variables for the calculation and parcelling out the work to be done over the processor array, along with all other computational overheads that are not replicated in other processors within the array.

One problem with this method is that there is a large amount of idle time within the array whilst the control unit is setting up the computation - as the array processors increase in both numbers and in power the time taken for the parallel computation becomes vanishingly small compared to the time taken for the control unit's initialisation. Two methods of reducing this idle time are to overlap the initialisation of computation two with the execution of computation one (as in the Illiac-IV array processor) or to overlap the addition and multiplication operations within the arithmetic pipeline (such as in the Cray II). Pipelines tend not to exceed 10 - 20 stages, and the number of processors within an array also tends not to exceed 10 - 20. There is noticeable inefficiency in the architecture, as the pipelines are empty for some time. Hence, at low levels of parallelism (such as 20 processors) the inefficiency may be tolerable compared to the extra expense of hardware required to keep the pipelines full.

Associative memory (AM) architectures utilise AM instead of the normal random-access memory (RAM). The fundamental difference between AM and RAM is that the former is contents addressable, allowing the parallel access of multiple memory words. This parallel access is used for search and comparison purposes, allowing the controlling processor to carry out a search or compare operation on a large number of memory addresses in a single memory access cycle.

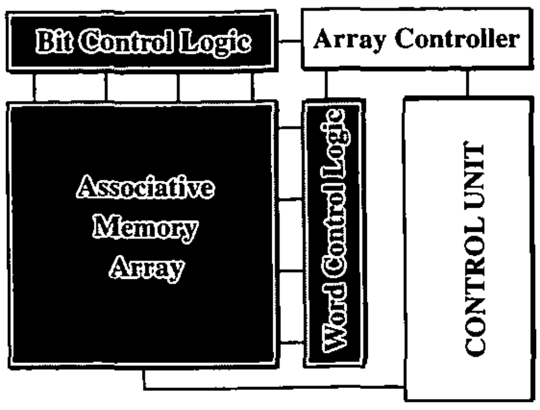


Figure 1.7 Associative Memory Schematic

A basic schematic of an associative memory structure is shown in figure 1.7. To initiate a search, a vector of data is sent into the bit-control logic (BCL) and on into the AM. This vector is simultaneously passed to all words in the AM, and each one compares the vector with its own contents. This operation results in a *match vector* being generated inside the word-control logic (WCL). This vector indicates which of the words in the AM hold data identical to the input search vector. Individual match addresses may be calculated from the match vector and used in further processing, or, as is usually the case, the match vector may be manipulated and then sent back into the AM; a write operation on the AM may then take place, which will then only affect those words in the AM that previously indicated a positive result to the match operation.

Additionally, a *mask vector* may also be sent into the BCL. This indicates that any write or match operation should only operate on the bit columns specified in the vector; it is then perfectly possible to modify the same field or fields within many words of the AM, whilst leaving unaltered those fields within the data words that were masked out from the operation; i.e. the lowest three significant bits could be altered in all data words that indicated positive to a previous match operation, leaving the rest of the bits in the data words unaltered. Normal read/write operations are still possible on the AM, and it is also possible to write a value into all words in the AM in a single operation.

The AM structure is used in areas where a fast look-up for data is required; the search is for whether an item is held in memory and, if so, where. The directory of a processor cache works in this fashion, with a quick check being made to see if data is held in the cache before a request is made to main memory. The AM is searched for a main memory address; if this search is successful then the location in the AM of the address data is used as an offset into another piece of fast memory, where the required data is held, which tends to be 10 - 20 times faster than main memory.

AM structures can also be used [Kane95] where a controlling algorithm needs to know if a particular result has already been generated, as well as a linked-list data structure container. This type of operation requires additional bits of memory to act as *tag fields*, which are manipulated by the AM mask vector. A 10-bit AM, with 8-bits for data and a two tag fields per data item, allows read/write/search requests to be made either of the 8-bit data or of the tag fields; this type of structure can be very powerful.

Systolic Architectures

The concept of systolic architectures was first introduced by Kung and Lieserson [KuLi78] as a high-performance, special purpose VLSI computer system. The term applies to certain computing structures with cellular organisations and pipelined dataflows that are suitable for applications that need to balance intensive computations with a demanding I/O bandwidth [KungHT80]. The architecture consists of a set of interconnected cells, each of which is capable of performing some operation - a simple schematic of a systolic array is illustrated in figure 1.8.

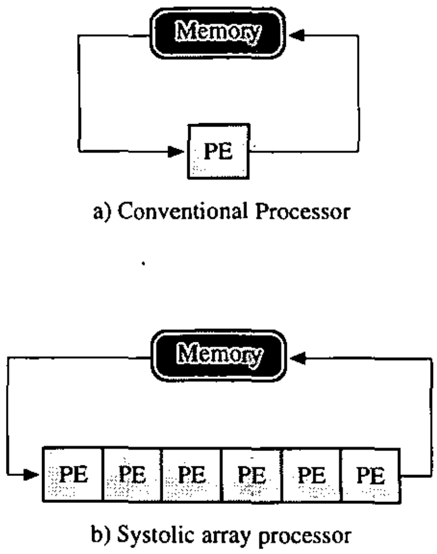


Figure 1.8 Basic Concept of Systolic Arrays

Information flows between the cells in a pipeline fashion, with only those cells at the boundaries of the array acting as I/O ports to the system as a whole. Data interconnections, save for the I/O ports, are entirely local, in that each of the PE's can communicate via direct physical interconnections only with neighbouring PE's. Each PE is controlled with the same internal clock. Between two consecutive clock cycles each PE carries out computations on the data that it received on the last cycle, as well as on any internally held data. Results of the computations are then stored internally and/or made available to neighbouring PE's, which can pick them up at the beginning of the next clock cycle. This 'pulsing' of data resembles the pulsing movement of blood under the contractions of the heart, which are called *systoles* in physiology. This analogy was the reason to give these computer structures the attribute *systolic* [KungHT82]. Further in-depth details of systolic arrays is given in section 2.3.

1.4.2.2 MIMD Architectures

The classic MIMD architecture, as shown in figure 1.3d, supports multiple processors executing independent instructions. The software processes executing on a MIMD architecture are synchronized by passing messages through the interconection network, or by data items stored in shared memory. The architecture is asynchronous, however, due to the decentralized control of hardware resources, and architectures falling under the MIMD label are not classified along with the synchronous architectures from section 1.4.2.1.

Distributed Memory Architecture

Each processor in a distributed memory architecture has registers, arithmetic and logic units, local memory and I/O drivers, each connected to a processor-to-processor interconnection network; this scheme is shown in figure 1.9. No sharing of memory and I/O is permitted in this architecture, with processors sharing information by passing messages via the interconnection network. This system supports communication through a point-to-point exchange of information.

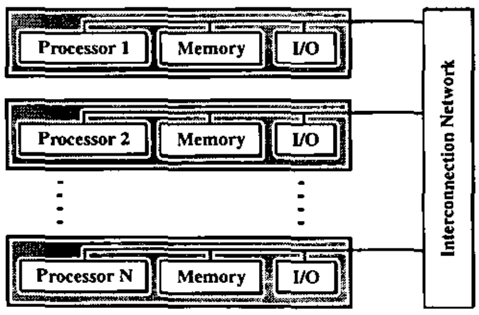
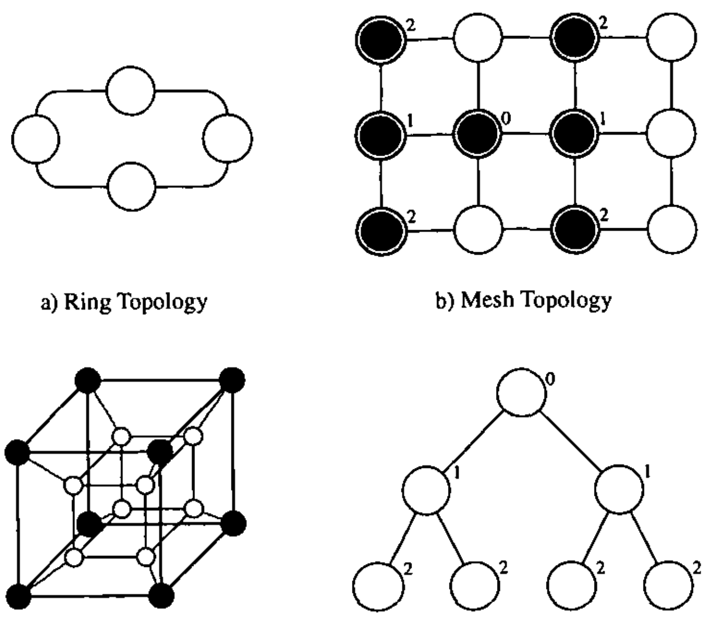


Figure 1.9 Distributed Memory Architecture

Various interconnection strategies have been proposed for this architecture, each of which suits a different class of parallel algorithm; each one has it's own strengths and weaknesses. Examples of distributed memory architectures include ring networks, the mesh, the hypercube and the tree, all of which are illustrated in figure 1.10.

The ring topology, shown in figure 1.10a, is an interconnection structure in which nodes are connected with point-to-point connections between processors and a cyclic structure overall. A transmitting processor places a message on the loop, which is repeated by all intermediate processors until it reaches it's destination; the destination processors takes the message off of the loop by simply failing to repeat it. Although there are many ways in which to operate a ring a common protocol is the *token ring* network. In this protocol a unique message, known as a token, is passed around the processors in the ring. If a processor is waiting to transmit data on to the ring then it takes the token off the ring and transmits it's message instead, otherwise it just repeats the token. The destination processor for any message re-initiates the process by placing a token back on to the network.



a) Ring Topology b) Mesh Topology c) Hyper-Cube Topology d) Tree Topology

Figure 1.10 Various MIMD Interconnection Topologies

The mesh topology, shown in figure 1.10b, is a near-neighbour mesh. It may or may not be recirculating, in that boundary processors can have direct connections to other boundary processors on the opposite edge of the mesh. All inter-PE communications can be specified as a set of routing functions. Whenever a routing function is activated the source PE transmits data to a neighbouring PE; in order to pass data between two PE's that are not directly connected within the mesh the data must be passed through a number of intermediate PE's.

The hypercube topology, shown in figure 1.10c, is a parallel processor whose interconnection structure treats individual processors as the nodes of a multi-dimensional cube. Two processors are interconnected if the corresponding nodes of the cube are neighbours. The hypercube has 2^n processors, each of which is labelled with an n-bit binary tag; in figure 1.10c $n = 4$, implying a four-dimensional hypercube (or four-plane). Each neighbouring processor's label would only be different in a single bit position. At most a message between two processors requires n steps. The route taken by the message is not fixed, yet is simple to realise - a message from processor `0100` to `1110` is only two steps long, as their tags are different in two bit positions. Routing is calculated by taking the exclusive-or of the source and destination node, in this case `1010`; this indicates that the message must traverse plane-1 and plane-3, but the order of traversal is irrelevant.

One of the most influential hypercube designs, the *Cosmic Cube* [Seit85], is a six-dimensional architecture, although the dimensions reflect interconnections useful for algorithmic purposes and does not correspond to physical dimensions. In the Cube each processor was an Intel 8086 processor chip, with an additional Intel 8087 floating-point unit, and all processors were not required to execute the same instruction concurrently; rather, instruction execution was independent at each processor node.

The tree topology, shown in figure 1.10d, is an architecture that is useful where the algorithm suits a master/slave process distribution. If a job can be shared reasonably evenly over many conceptual levels than the leaf-node processors, indicated by a *2* in the figure, carry out some computation and pass their result up the tree. The branch nodes, indicated by a *1* in the figure, carry out further processing, again passing results further up the tree until eventually the root node, indicated by a *0* in the figure, completes the calculation.

The tree topology is difficult to implement in the physical sense; however, by mapping the entire tree on to a reconfigurable mesh, and indicated in figure 1.10b, it is possible to build and implement machines that can utilise the architecture of a binary tree. Each node is mapped in such a way that maximises the potential of the connections available within the mesh; the mesh can never be 100% utilised in this fashion, as sub-branches have to be arranged so that their resultant leaves do not overlap.

Shared Memory Architecture

Similar to the distributed memory architecture, each processor has registers, arithmetic and logic units and I/O units, each being connected to the processor-to-processor interconnection network; this scheme is shown in figure 1.11. Memory is shown as being separate subsystems which is shared amongst all processors, although each processor normally has it's own bank of local cache memory. Again, data is exchanged between processors via the interconnection network. The shared memory architecture conveniently provides a simple means for information interchange and synchronisation, as any two processors can communicate through a shared location. Shared memory computers do not have some of the problems associated with message-passing parallel architectures, such as message latency whilst data is forwarded around the architecture interconnection system by the intermediate PE's. However, other problems, such as data read/write access synchronisation, must be solved.

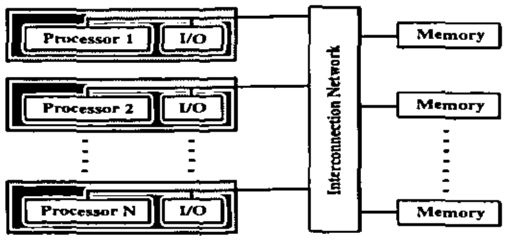


Figure 1.11 Shared Memory Architecture

A crossbar system of interconnect uses multiple crossbar switches to connect several memory units to several processors. It has a low bandwidth and is quite complex to implement, despite being conceptually the simplest shared memory interconnect scheme. A bus system, however, is much better; by replacing the interconnection network in Figure 1.11 with a multi-access bus we have a shared bus method of communication, with each processor and memory unit being connected to it. The cache memory in the processors, along with small amounts of local memory, shortens the effective memory cycle time as well as reducing the use of the bus; one processor does not slow down any others through extensive use of the bus.

As cache/local memory solutions that reduce global bus access by 95% are easily realisable a simple shared memory bus system can support up to 20 processors. Systems with more processors, such as 1000 or more, are unlikely to be realisable without a technological breakthrough that gives a very high bus bandwidth at very low cost.

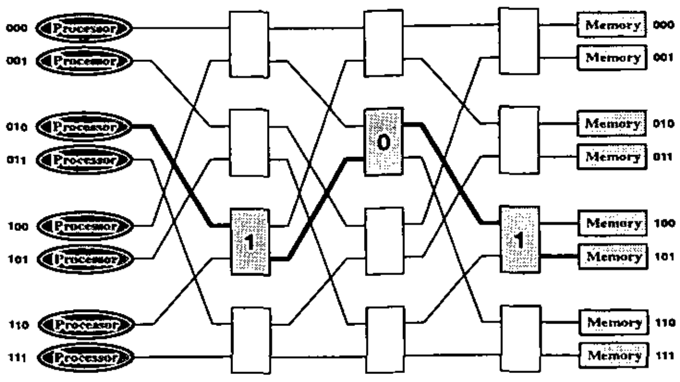


Figure 1.12 Shuffle-Exchange Network

A system of interconnection that lies on an intermediate point in the spectrum of possible interconnect networks [Ston71], yet still provides a good level of performance, is the shuffle-exchange interconnection, as shown in figure 1.12. The figure shows processors at one side, memory units at the other and a series of combining-switch units in between. The combining-switch units can either pass the inputs to the outputs, pass the inputs to the opposite output (reversal) or pass a single input to both outputs (broadcast).

The bandwidth is higher than the bus, but lower than the crossbar. The time-cost for access is *O(N log N)*, as opposed to *O(N)* for the bus and $O(N^2)$ for the crossbar. The architecture also solves the shared-memory exclusive access problem; if access to a shared variable is saturated then there is, normally, no additional speed improvement no matter how many processors are added to the system. However, this does not wholly apply if the exclusive access can be accomplished in part of the switching network and in part of the memory. In the shuffle-exchange the exclusive access is, in effect, done in parallel rather than in serial by making use of the facilities built into the switching mechanisms.

In operations where global memory accesses do not conflict the shuffle-exchange network bandwidth can increase dramatically. It has been shown that if N processors place simultaneous synchronised global requests, such that processor i requests data from memory $i+c$ for any constant c, the requests can be granted simultaneously without any conflict whatsoever [Lawr75]. Further, Lawrie went on to show that if processor i requests data from memory $pi+c$, where p is an odd number, no contention occurs provided N is a power of 2.

Although a powerful interconnection structure for shared memory machines it is far from being perfect. The exclusive access conditions make it unsuitable for some problems, which become unfeasible other than for small values of N. A common algorithm in parallel computing, the fast fourier transform (FFT), has two types of processor-to-processor communication; the butterfly operation, where pairs of processors exchange data and compute further values, and the reverse-binary operation, which transforms the order of the output data. It has been shown [Cvet86] that these two operations are incompatible with the shuffle-exchange network. If the data stored within the processors such that the butterfly operation proceeds without conflict then the reverse-binary operation results in maximum conflict within the network. Conversely, if the reverse-binary operation is made conflict free then the butterfly operation results in maximum conflict.

1.4.2.3 MIMD Paradigm

The architectures within this section all exhibit the normal MIMD characteristics of asynchronous operation and the concurrent manipulation of multiple instruction and data streams. Each architecture has some form of organising principle that is fundamental to it's overall design, as well as the normal MIMD characteristics, so these architectures are classified separately under the heading 'MIMD Paradigm' instead of with the MIMD architectures from section 1.4.2.2.

MIMD/SIMD Architectures

This hybrid architecture is a MIMD machine that has the facility to allow selected portions of the architecture to be controlled in a SIMD fashion. In the MIMD/SIMD hybrid architecture each of the SIMD controlling units have a number of slave SIMD processors under their direct control. Also, each controlling unit has a MIMD operational node which passes them algorithms to be carried out - this structure is shown in figure 1.13a.

The SIMD processor control/slave portion of the architecture is akin to a shared-resource array processor [HwNi80], which consists of two or more controlling units which share a pool of dynamically allocated PE's; such a system is shown in figure 1.13b and is also known as 'Multiple SIMD'. To fit into the MIMD/SIMD architecture it is plain to see that all SIMD control units have a 'master' MIMD controller, such as that shown in figure 1.13a.

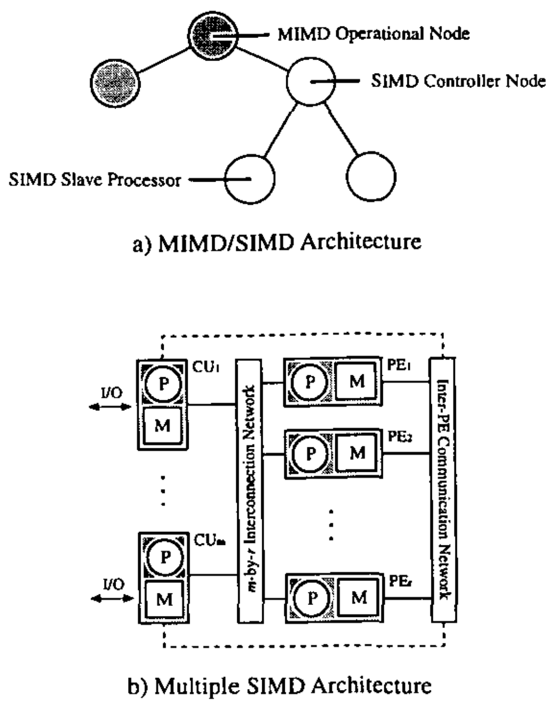

MIMD Operational Node
SIMD Controller Node
SIMD Slave Processor
a) MIMD/SIMD Architecture
I/O
P
M
CU₁
CUₘ
m-by-r Interconnection Network
PE₁
PE₂
PEᵣ
Inter-PE Communication Network
b) Multiple SIMD Architecture

Figure 1.13 MIMD/SIMD Architecture

Dataflow and Reduction Architectures

Dataflow [Denni75] and reduction [Trel82] architectures share a similar operational tenet: the former enables instructions once all operands are available, whilst the latter enables instructions once their results are required elsewhere. The sequence of executed instructions is based on data dependencies, with the reduction architecture also known as a *demand driven* architecture, and these architectures are potentially able to exploit any inherent parallelism at the task, routine and instruction levels. Both architectures make use of tokens to indicate to the machine controllers what a data item is within the machine, with reduction machines also having 'demand' tokens, which indicate to instructions that their results are demanded.

Despite the apparent advantages of these approaches there are a number of disadvantages to such architectures [HwBr84], which are as follows:

- The data driven at instruction level causes excessive pipeline overhead
- Data flow programs tend to waste memory due to the much increased code length
- The packet/token switching network within large data-flow computers becomes cost-prohibitive and is itself a bottleneck in the system

- The architectures have potential in small-scale or very large-scale parallel systems, but compete less favourably at other levels when compared against existing pipeline, array and multiprocessor system

Wavefront Array Architectures

A wavefront approach to designing cellular processor arrays was proposed by Kung et al [KungSY82]. Wavefront processors combines systolic data processing with an asynchronous dataflow, yet retain the features from systolic architectures, such as modular processors and regular, local interconnection networks. The major difference is that wavefront processors do not make much use of a global system clock: they exploit the time-delays used for synchronisation the systolic data pipelines and the associated asynchronous handshaking as the method for coordinating inter-processor data exchange.

Using this method, as illustrated in figure 1.14, a processor informs it's successor in the array that it has completed it's computation and that a result is ready *(1)*. The result is sent *(3)* once the successor indicates that it is ready to receive the data *(2)* and, in turn, sends an acknowledgement to it's predecessor to a state that the data has been received *(4)*. This operation cascades throughout all processors in the array *(5)*. It is clear from figure 1.14 that processors receiving data from wavefront -A- will be able to start operating upon it before processors that were previously concerned with wavefront -A- will be able to start processing new data from wavefront -B-.

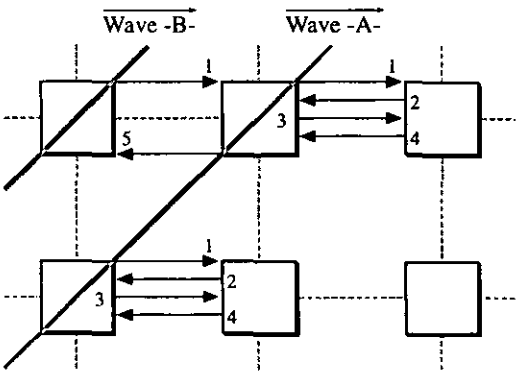


Figure 1.14 Wavefront Array Architecture

Using this method of handshaking means that the flow of wavefront data through the array is very smooth; the wavefronts do not interfere with each other, as the processors holding the wavefront data act as the propagation controllers and cannot, by definition, propagate any data until the destination is ready for it. This is, in principle, a very powerful architecture, as an algorithm can pass through the array as fast as the processors can handle it. There is no set time frame for a stage of the algorithm to be executed within a processor, such as is the case for the global clock in a systolic array. Once a processor has completed its computation it does not have to wait upon a system clock before it can start on the next calculation; it can start as soon as the data values required are available.

1.5 Thesis Organisation

This thesis is concerned with the design and specification of an instruction systolic-array architecture for the VLSI implementation of neural networks with real-time learning. The discussion can be divided into three main parts. Other sections of the thesis consist of the contents lists, references, some details of further work and final conclusions on the ideas put forward throughout this thesis.

The first part is an introduction to the subjects that form the technological background to the discussion; it contains a general overview of computer evolution and architecture, as well as more detailed studies of VLSI technology, systolic arrays and neural networks. The second part consists of the major research undertaken by this thesis; it contains the overview of the VLSI architecture required for successful hardware implementation, as well as details of the systolic algorithms to carry out neural learning using three different learning methodologies. The third part consists of the software and hardware implementations carried out on the basis of the work done in the second part. It consists of the software simulator of the systolic array architecture, along with a few sample applications to demonstrate that the algorithms have been successfully implemented, and the hardware designs carried out towards the end of fabricating a working VLSI device.

1.5.1 Part 1 - Introduction

Part 1 forms the introductory part of the thesis and consists of Chapters 1 to 4 respectively. Chapter 1 gives a general introduction to the areas relating to systolic array processing, notably the evolution of computer architecture, the advancement of integrated circuit technology and the various classifications of parallel computer architectures. Systolic arrays themselves are just one class of parallel computing, but it is one that is very amenable to implementation in VLSI technology.

Chapter 2 contains more detailed descriptions of the areas required for hardware implementations of systolic arrays. Descriptions of the steps required in VLSI fabrication are given, along with the problems associated with the industry. The chapter goes on to give descriptions of computer-aided design techniques currently available, with details of on- and off-chip testing methodologies. The chapter concludes with further details on systolic array processing, along with derivations of the fundamental algorithms associated with the architecture.

Chapter 3 described various neural network learning methodologies, with examples of both supervised and unsupervised techniques. It gives a concise history of the field, from the pioneering work at the beginning of this century, as well as describing the electro-chemical processes involved in biological neural system. Chapter 4 carries on from this with two case studies of recent practical applications of neural networks, along with two research-led applications of neural networks in VLSI devices.

1.5.2 Part 2 - Systolic Array Architecture and Algorithms

Part 2 forms the main theoretical part of the thesis, consisting of Chapters 5 and 6. Chapter 5 contains details on the proposed systolic array architecture for on-chip neural network learning. It describes the device, firstly, in overview, outlining all of the basic requirements of the architecture. It then goes on to describe the reconfigurable instruction set aspect of the systolic array, along with the method developed for approximating any neural activation functions within the array. The chapter concludes with a more in-depth description of the hardware design features that make up the systolic array architecture.

Chapter 6 continues the systolic array architecture theme by describing the systolic algorithms developed for the architecture. There are three different neural learning methodologies that the architecture can currently support, each methodology being of a different type. Each component of the various algorithms is described in detail, along with timing results for one application on each of the three methodologies.

1.5.3 Part 3 - Software and Hardware Implementation

Part 3 forms the main deliverable items of this thesis and consists of Chapter 7 and Appendix A. Chapter 7 describes the implementation in software of a systolic array simulator that closely matches the hardware and software designs from Part 2. This describes the general operation of the software, giving an all-round picture of what the software is capable of. It also gives descriptions of several application examples implemented on the simulator, in order to show that the designed architecture is capable of successfully carrying out neural processing.

Appendix A contains all of the detailed circuit diagrams, and associated VHDL code, required to implement every aspect of the hardware designs from Chapter 5. This section is designed as an appendix, with higher level descriptions of each circuit in Chapter 5 referring to the actual implementations presented in Appendix A.

2 BACKGROUND TECHNOLOGY

INTRODUCTION

This chapter describes in more detail the various technologies that have had the largest influence in the design of a systolic array processor for real-time neural network learning. It begins with a description of the various common methods of IC fabrication, with details on the different methods and some common measures employed to improve the basic fabrication process. Methods for increasing the reliability of IC testing once devices are fabricated are given, along with the available techniques for digital device testing. The chapter concludes with an in-depth discussion on systolic array processing, with details on the various topologies commonly used and a look at the basic algorithms used in systolic array processing.

2.1 VLSI Fabrication Technology

2.1.1 Silicon Semiconductor Technology

2.1.1.1 Silicon Wafer Generation

In it's pure state silicon is a semiconductor, having a bulk electrical resistance somewhere between that of a conductor and an insulator. The resistance of silicon can be varied, over several orders of magnitude, by introducing impurity atoms into the crystal structure: these dopants can either supply free electrons or holes. Impurity elements that use electrons are known as acceptors, since they accept electrons already present in the crystal, leaving vacancies (or holes) behind them, whilst elements that introduce electrons into the crystal are known as donors. Silicon that has a majority of donors is known as *n-type* and that which contains a majority of acceptors is known as *p-type*.

By bringing together n-type and p-type material, where the silicon changes from n-type to p-type, a junction is formed; by arranging these junctions into particular physical structures we can form various kinds of semiconductor devices.

The basic raw material used in fabrication is the silicon wafer, which varies from 75mm to 150mm in diameter and is less than 1mm thick. The wafer is cut from an ingot of single-crystal silicon which has been pulled from a crucible melt of pure, molten polycrystaline silicon. Once the wafer is cut to the required size, usually with diamond blades, and at least one face is buffed and polished to a flat, scratch-free mirror surface.

2.1.1.2 Silicon Oxide Generation

Many of the techniques used in the manufacture of silicon-based IC's rely on the properties of silicon dioxide (SiO_2). Therefore, the reliable manufacture of SiO_2 is extremely important to the entire fabrication process. The oxide is achieved by heating the wafers of silicon in some form of oxidising atmosphere, such as oxygen. The two most common approaches are:

- Dry Oxidation: the oxidising atmosphere is pure oxygen, with temperatures in the region of 1200°C
- Wet Oxidation: the oxidising atmosphere contains water vapour, with temperatures usually between 900°C and 1200°C

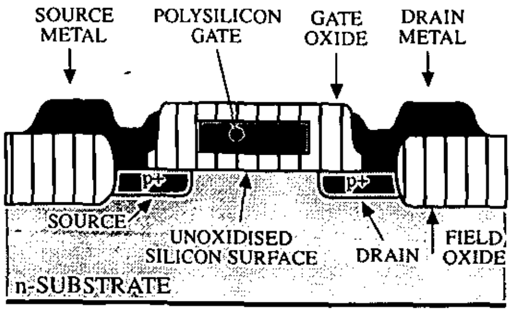


Figure 2.1 Field Oxide Growth

It is important to note that the oxidisation process actually consumes silicon. In the standard fabrication process, as described in section 2.1.2, a layer of silicon is placed on top of the original p-type substrate and is then oxidised - this layer is known as *field oxide*. Since SiO_2 has approximately twice the volume of Si, the SiO_2 layer grows equally in all directions, so the actual SiO_2 layer grows into the base substrate (which was initially polished flat). This can be seen in figure 2.1, which shows a p-type transistor in which the SiO_2 field oxide can clearly be seen to have grown into the substrate. Note, the steps required to create such a transistor are covered in detail in section 2.1.2.

2.1.1.3 Selective Diffusion

The placement of the acceptor and donor areas on the silicon substrate needs to be a very precise process, as is the placement of any associated structures. The ability of SiO_2 to act as a barrier against doping impurities is a vital factor in the placement process, which is known as *selective diffusion*. The process consists of the following stages:

- Grow a layer of SiO_2 on the substrate surface
- Open a window in the SiO_2 using a combination of a photoresistant layer, a glass mask and an ultra-violet (UV) light source
- Remove the SiO_2 not subjected to UV light with a suitable etchant
- Subject exposed silicon substrate to a dopant source

The idea is to cover the SiO_2 with an acid-resistant coating, which is normally a photoresistant organic material. The UV light that passes through the glass mask polymerises the exposed photoresist (PR), thus creating the pattern of SiO_2 that is to remain. The unpolymerised PR, along with underlying SiO_2, can then be easily removed with an organic solvent. This process is shown in figure 2.2.

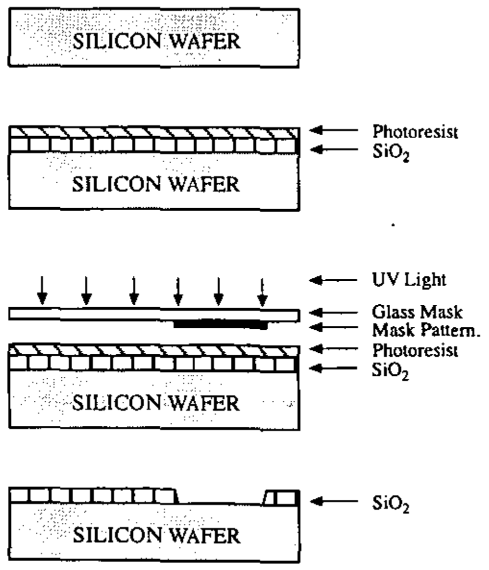


Figure 2.2 Patterning of SiO2

The main problem associated with using PR's in conjunction with UV light sources is that there is a sizeable amount of diffraction around the edges of the pattern masks. The tolerances of the process has limited the line width of around 1.0μm. However, during recent years advances have been made in the field of electron beam lithography (EBL). This is a good technique for pattern generation, allowing line widths down to 0.5μm and beyond with good definition. The disadvantages of EBL are that new plant equipment is required and that a large amount of time is required to access all points on the silicon wafer. Fabrication plants are extremely expensive; manufacturers are reluctant to replace such expensive equipment and research into improving the definition of selective diffusion is still going on.

2.1.2 Standard p-Well Fabrication Process

The p-well fabrication process is fairly well-defined. It starts by taking a moderately doped n-type substrate, creates the p-wells to place n-type devices on and then builds the p-type devices on top of the native n-type substrate. Each of the stages of the process are described in this section [WeEs88]; an overhead view of the mask for each stage is shown along with an idealised plan view of the resultant physical structure of the substrate. The example used throughout this section is the simple CMOS inverter structure.

2.1.2.1 Wells and ThinOx Deposition

Figure 2.3 shows the first two stages of the p-well process. The first mask (a) defines the p-well within the base substrate. Once the field oxide is deposited on to the substrate a small section is etched away. The exposed substrate is then doped with a p-type source, creating a p-type well within the n-type substrate.

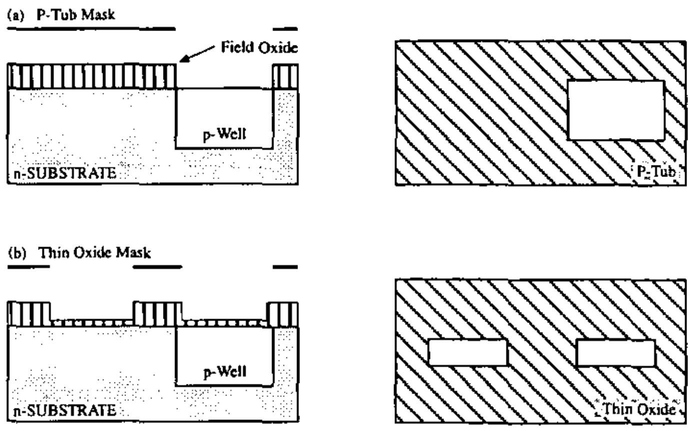


Figure 2.3 Well and ThinOx Deposition

Using the second mask (b) areas of the field oxide are selectively stripped away down to the substrate surface. A very thin layer of SiO_2, known as thin oxide, is then grown on those areas. It is these two thin oxide regions that will form the n- and p-type diffusions for the transistor source/drain regions.

2.1.2.2 Gate Definition

Figure 2.4 shows the next three stages of the p-well process. The first mask (a) defines the surface area to be covered in polysilicon[1], with the pattern required for a CMOS inverter being an inverted 'U'.

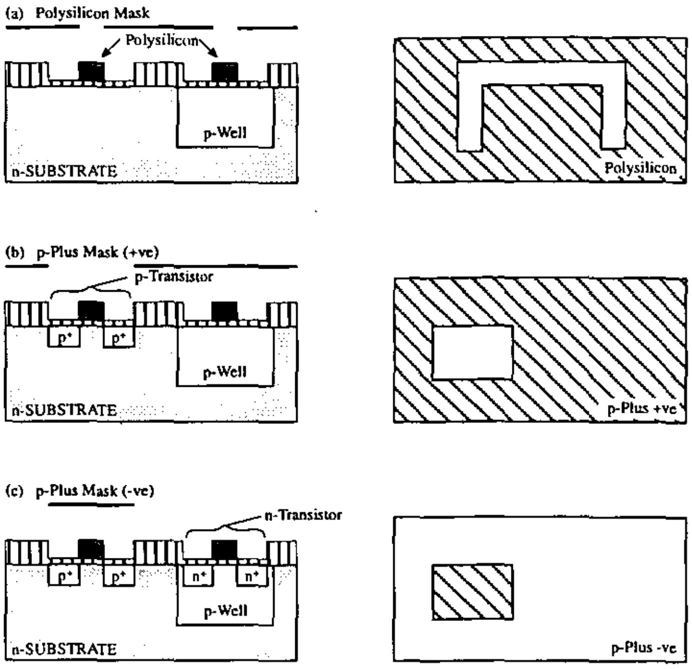


Figure 2.4 Gate Definition

The second mask (b) is used to indicate the thin oxide areas that are to be doped with a p^+ source. These areas will become p^+ diffusion area, which provide self-aligning source and drain areas with the polysilicon that divides the two diffusion regions. A p^+ diffusion region inside an n-substrate allows the construction of a p-type transistor.

The third mask (c) shows a similar process, but which creates an n-diffusion region within the p-well. The mask used is the complement of the mask from step (b). These areas of n^+ diffusion within a p-well, along with the separating section of polysilicon, allows the construction of n-type transistors.

2.1.2.3 Metallisation

Figure 2.5 shows two of the final three stages of the p-well process. The first mask (a) defines the areas required for contact cuts, which allows metal tracks to make contact with the n^+ and p^+ gate areas.

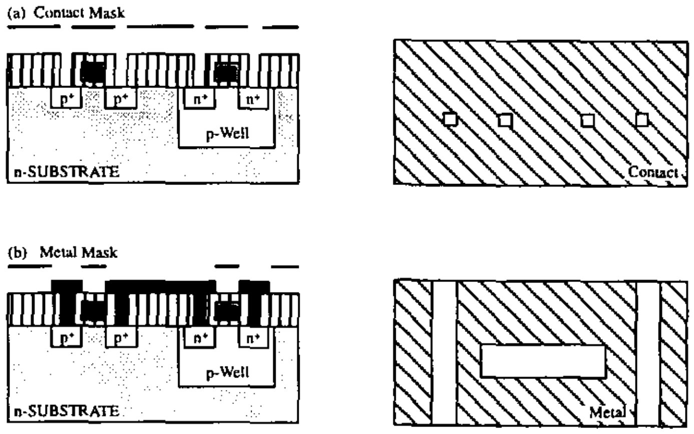


Figure 2.5 Gate Metallisation

[1] Although this shows a polysilicon gate process it should be noted that CMOS devices originally implemented gates with aluminium. This technology was the basis for the majority of the CMOS circuits in the 1970's

A layer of SiO_2 is first grown on to the silicon, and then several patches of gate oxide are etched away as far as the silicon surface. This process leaves all of the n+ and p+ regions exposed, paving the way for the next step.

The second mask (b) shows a layer of metal being added to selective parts of the device. The left and right channels are for the power and ground (V_{DD} and V_{SS}) connections for the inverter, whilst the central channel is for collection of the output signal. The inverter input is driven onto the polysilicon connection.

The final stage of this process, which is not shown, is that once all devices have been created on the silicon wafer the chip packaging bonding pads are etched, in order to allow for the external I/O wire bonds to be attached to the final fabricated chip. The full layout for the CMOS inverter, along with its standard schematic, is shown in figure 2.6.

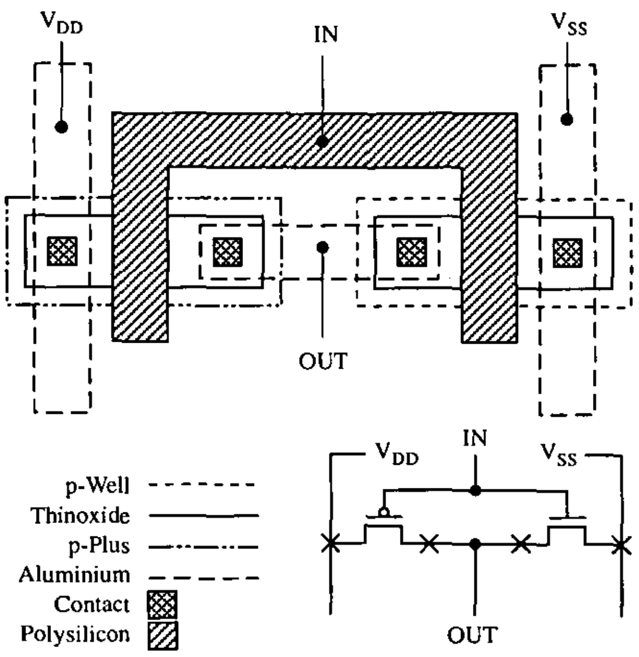


Figure 2.6 Inverter Layout and Schematic

2.1.2.4 Associated Problems

In order to achieve low threshold voltages either deep well diffusions or high well resistivity are required. The latter can accentuate a problem known as *latch-up*, whilst deeper diffusions increase the silicon area due to larger spacings being required between the n- and p-type transistors. The required thresholds are achieved by making the well concentrations one order of magnitude higher than the base substrate doping density. Unfortunately, n-type transistors within the p-well suffer from excessive source/drain capacitance, thus slowing down their switching speeds. Despite the advances made by using CMOS technology n-type transistors are, in general, inferior to those that could be built on their native substrate; i.e. with no well present. Thus, n-type transistor circuits in CMOS will tend to be slower than their counterparts manufactured using the older nMOS process and performance degradation in some CMOS circuits should be expected by any VLSI designer.

As stated by Weste and Eshraghian [WeEs88] *"...if every silver lining has a cloud then the cloud that has plagued CMOS is a parasitic circuit effect called 'latch-up'" (p.58).* This is the result of shorting the V_{DD} or V_{SS} lines which may result, if luck is on the designer's side, in only system failure but tends to result the complete self-destruction of the CMOS device.

Estreich and Dutton [EsDu82] describe the latch-up effect very well. They state that, effectively, there are two parasitic bipolar transistors inherent in the n-substrate and p-well, each of which has a gain factor associated with it. The first transistor (T_1) has a collector-base-emitter of p-type transistor source, the n-substrate and the p-well itself (pnp transistor), whilst the second transistor (T_2) is n-substrate, p-well and the n-type transistor source within the p-well (npn transistor). The equivalent circuit for this is shown in figure 2.7.

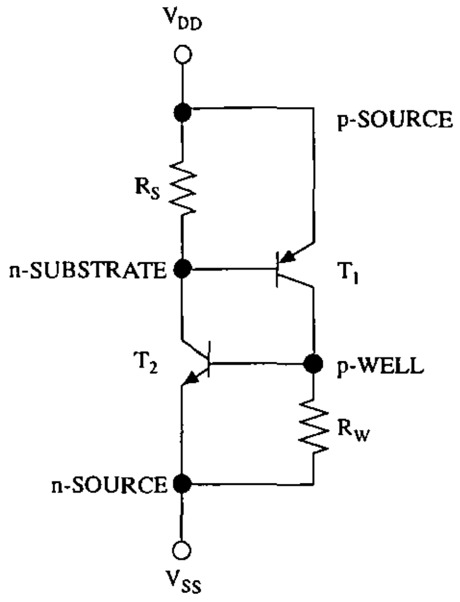


Figure 2.7 Parasitic Latch-Up Equivalent Circuit

There are two areas of resistance within the circuit: one in the n-substrate (R_S) and one in the p-well (R_W). The larger these resistances are then the more susceptible the device will be to latch-up, so the process engineers need to devise methods to keep these resistances low. A different method of reducing the chances of latch-up is to reduce the gain of the bipolar transistors, but this is not as easy as reducing the values of R_S and R_W. A simple method to keep the resistances low is to use multiple additional substrate contacts, which effectively short out the resistors.

A modified inverter cross section, incorporating two substrate contacts, is shown in figure 2.8. The n-substrate is connected to the positive supply V_{DD} through a V_{DD} substrate contact, whilst the p-well is connected to the negative supply V_{SS} through a V_{SS} substrate contact. The contacts themselves are formed when the areas for p-type and n-type transistors are being doped, as described in section 2.1.2.2, with additional p^+ and n^+ diffusion areas being implanted into the p-well and n-substrate respectively. Additional contact cuts are also made, with a further metallic connection made between the relevant power input and the substrate contact.

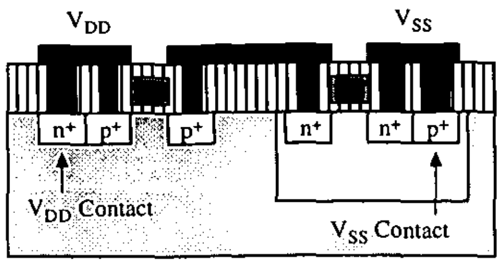


Figure 2.8 Substrate Contacts

With current process technology the possibility of latch-up occurring has been reduced to the point whereby the designer does not have to be concerned about it, so long as the design contains *liberal* substrate contacts. The definition of 'liberal' is not hard and fast, as it is usually acquired from designers who have made successful designs on a particular fabrication process. Although it is possible to synthesise the parasitic effects, and thus calculate the required spread of substrate contacts, it is a difficult (i.e. time-consuming) task.

However, it is possible to reduce the likelihood of latch-up to a great degree by following a small set of rules:

- Every well must have a substrate contact of the appropriate type
- Every substrate contact must be directly connected to the power supply by metal
- Place a substrate contact for every 5 - 10 transistors in the design
- Place substrate contacts as close as possible to the source connection of a transistor connected to a power line - this reduces R_S and R_W
- For the conservative a substrate contact should be placed for every power connection

- At the chip layout stage pack n-transistors towards V_{SS} and p-transistors towards V_{DD}

The most likely point for latch-up to occur is at the I/O structures, as the current flow at those points is large. This can cause quite large parasitics to be present and abnormal voltage levels may be encountered. Although these structures can be created by the designer by following another set of rules [WeEs88] [EsDu82] the simplest method is to use only proven I/O structures that have been designed by experts who fully understand the fabrication process being used at a detailed level.

2.1.3 Other Fabrication Processes

2.1.3.1 n-Well Process

Before the introduction of the CMOS p-well fabrication process the dominant fabrication technique was the nMOS process. Many manufacturers had spent vast sums of money to built plant based on nMOS, and the thought of such plant becoming obsolete overnight was not a pleasant one. An advantage of the n-well process is that it can be fabricated on the same plant equipment as conventional nMOS; because of this the n-well process is often retrofitted to the existing nMOS lines [Ohzo80].

The n-well process is very similar to the p-well process, except that the substrate is p-type, the n-type transistors are directly on the substrate and the p-type transistors are fabricated within a separate n-well within the substrate. Like the p-well process the n-well creates non-optimum transistors, but of the opposite type; p-type transistors perform relatively poorly compared to their n-type counterparts. However, CMOS designs are beginning to incorporate more n-type transistors rather than p-well, so the disadvantages of having non-optimum p-type transistors is partially offset. The n-well process technology is, therefore, very suited to the 'mostly n-type' CMOS designs and provide distinct performance advantages over the p-well process.

2.1.3.2 Twin-Tub Process

In both the p-well and n-well process only one type of transistor has near-optimum operational characteristics. The twin-tub process attempts to optimise both types of transistor separately, making it possible to modify threshold voltages and other parameters of both n- and p-type transistors independently [Parr80].

The process begins with either an n^+ or p^+ substrate which has a lightly doped epitaxial layer placed on top of it, which is used to protect against latch-up. The object of having the epitaxial layer is so that highly pure silicon layers can be grown, each with a controlled thickness and accurate doping levels.

The process sequence is similar to that for p-well, save for the actual formation of the twin -tubs where both n-wells and p-wells are created. The finished cross-section for a standard inverter is shown in figure 2.9. Substrate contacts are still used in the process, as the epitaxial layer does not eliminate latch-up altogether.

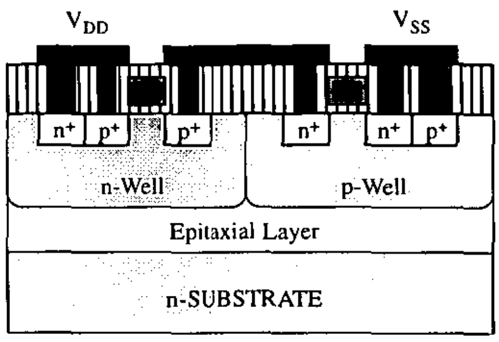


Figure 2.9 Twin-Tub Process Cross-Section

The two wells are defined and implanted individually, which is followed by the formation of field oxide and gate oxide. At this stage the thresholds of future p-type transistors are modified by further implants into the relevant sections of gate oxide within the n-well. Polysilicon is then deposited as required, and the substrate contact p^+ and n^+ regions within the wells are implanted. Finally, contact cuts are made, the device is metallised and then finished off by the cutting the areas for I/O bonding pads.

2.1.3.3 Silicon on Insulator Process

The silicon on insulator CMOS process (SOI) has several potential advantages over the traditional CMOS processes [MaSi64]: higher circuit density, no latch-up problems and lower parasitic capacitances. There are various mask and doping steps required to form n-type and p-type devices, but the extra steps involved in standard CMOS processes to create additional implanted wells simply do not exist with this technology.

Snapshots of several stages of the SOI fabrication process are shown in figure 2.10. Initially, a layer of thin very lightly doped silicon is grown over an insulator, such as sapphire or magnesium aluminate spinel. The silicon is then etched away, except for the where a diffusion area is required. This forms a number of n$^-$ islands on the insulator surface, and is shown in figure 2.10a.

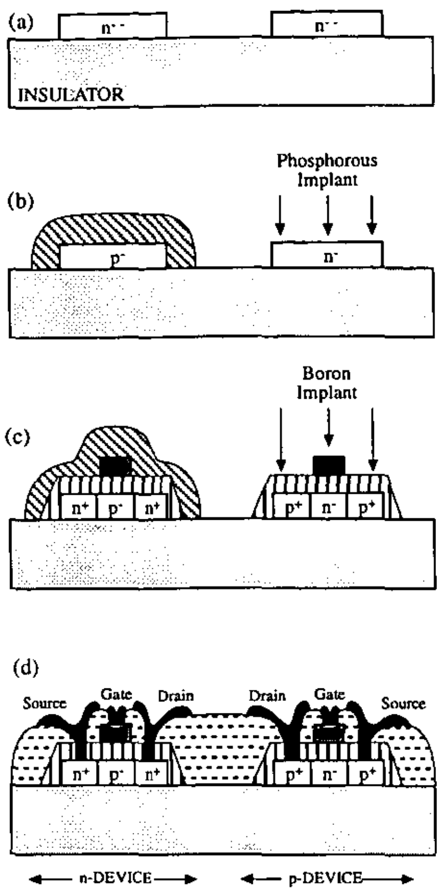


Figure 2.10 Selection of SOI Process Steps

The actual diffusion areas are created by masking one island with photoresist and then implanting a dopant into the other. This creates lightly doped p-type and n-type diffusion regions, with boron being used for the p-channel and phosphorus used for the n-channel devices; this is shown in figure 2.10b.

After the polysilicon is placed over the diffusion areas, using standard CMOS techniques, the diffusion areas are changed into n-p-n or p-n-p transistor islands. Photoresist masks off one island and the polysilicon masks off the base diffusion areas to create the junction. Again, phosphorus and boron are used to create n-type and p-type areas respectively, and this is shown in figure 2.10c.

To finish off the SOI process a layer of phosphorus glass, or some other insulator, is then deposited over the entire structure. Contact cuts are then etched from the top insulator and metallisation then occurs. This is shown in figure 2.10d, where the n-type and p-type transistors are also labelled.

Despite the advantages that the SOI process has over the other CMOS techniques outlined in this section there are some disadvantages. The single-crystal sapphire or spinal substrates are much more expensive than silicon and, because of this, the processing techniques themselves are less advanced than bulk silicon processes. The device gains are smaller than those associated with CMOS devices, so the I/O structures have to be quite large to be able to cope with these. Although SOI has the potential to be the fastest and most reliable process technology for CMOS fabrication it is also the most expensive; it's take-up by manufacturers has been, therefore, less than enthusiastic.

2.1.3.4 Process Enhancements for CMOS

There are a number of process enhancements to the standard CMOS process that increases the routability of designs and decrease resistance between devices. The simplest method of reducing the resistance of circuits [Chow83] is to reduce the resistance of the polysilicon lines by combining it with some refractory metal. Chow outlines four different approaches to this problem, two of which are shown in figure 2.11, each of which uses a different type of gate material. Each approach uses a standard substrate or well, with appropriately doped diffusion regions for the transistor being created. The gate and field oxide shown is, as always, SiO_2.

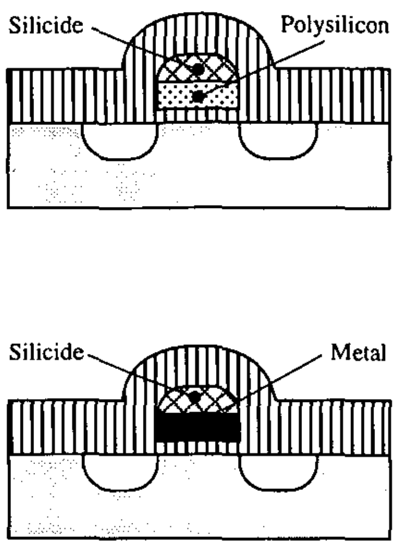


Figure 2.11 Silicide Gate

The first approach replaces the standard polysilicon gate with a sandwich of silicide upon polysilicon. The silicide is a mechanically strong structure; tantalum silicide is also stable throughout standard CMOS process stages and can also be retrofitted onto existing process lines. This approach is called the *polyside* approach. The second approach shown uses what is called a molybdenum gate, which is a sandwich of silicide and metal; this is called a *heart of moly* structure. The benefit of all of Chow's methods is that they reduce the overall interconnect resistance, which allows the gate material to be used over reasonably long distances; an additional benefit is that it can achieved with minimum disturbance to any existing CMOS process line, as it does not require any additional masks.

A simple approach to increase the overall routability of circuits is to use a second layer of metal. Additional layers of metal, as a rule, have a courser pitch due to the topology of the silicon surface being more varied. Connections between first level and second level metal (metal-1 and metal-2) is achieved by a *via* contact, as shown in figure 2.12. If further connection is required to either diffusion or polysilicon then a separation between the via and the contact cut is not normally required, even though it is shown in figure 2.12, but in less advanced or underdeveloped processes a separation gap may be needed. Such as gap would require an intermediate connection between metal-2 and the polysilicon or diffusion region; such connections are normally made using metal-1. If a separation gap is not required then the via can be made directly over the contact cut, giving a direct connection to the underlying layers.

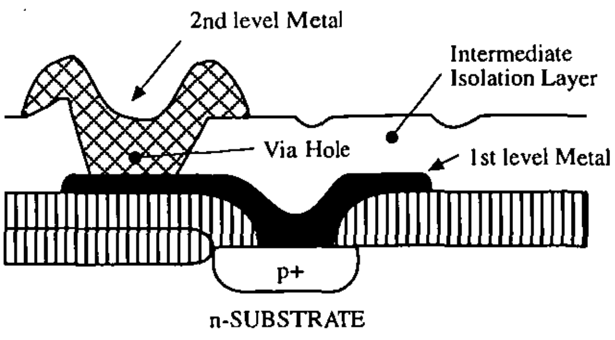


Figure 2.12 Second Level Metal

A novel method for preventing latch-up [IEDM83] [Yama83] is to introduce deep trenches of SiO_2 into the base substrate material. This methodology improves the n-to-p spacing of transistors, thus reducing the chances of latch-up occurring. The trenches are placed between n-type and p-type transistors, thus separating completely an injected well from other substrate material on the horizontal plane; a cross-section for this process in shown in figure 2.13. In practice, however, a p^+ substrate with injected n^+ diffusion regions will have a p^- epitaxial layer at around the level of the bottom of the trenches, with additional p^+ diffusion regions above the epitaxial layer wherever n-type transistors are required.

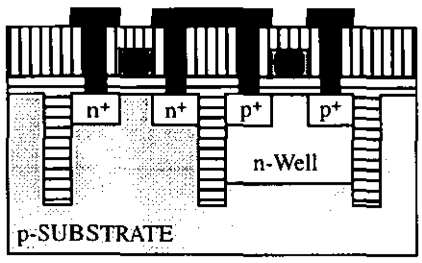


Figure 2.13 Trench Isolation Cross Section

One other innovation in the field is the use of 3-dimensional structures in the fabrication process. The ideas behind this are to increase the performance and to reduce the overall area by utilising the vertical dimension [GiLe80] [Kawa83]. The SOI process modification proposed by Kawasura et al in shown figure 2.14. It places transistors in the vertical plane within the phosphorus glass layer, with the vertical transistor being formed using n-p-n or p-n-p techniques. It is mounted within a different gate oxide from SiO_2, otherwise parasitic circuit effects can occur between two areas of identical and active gate oxide; these transistors use silicon nitride (Si_3N_4) instead. However, although this technique seems to have a good future ahead of it the fact that it requires the SOI process as a base means that it is still prohibitively expensive for all bar experimental uses.

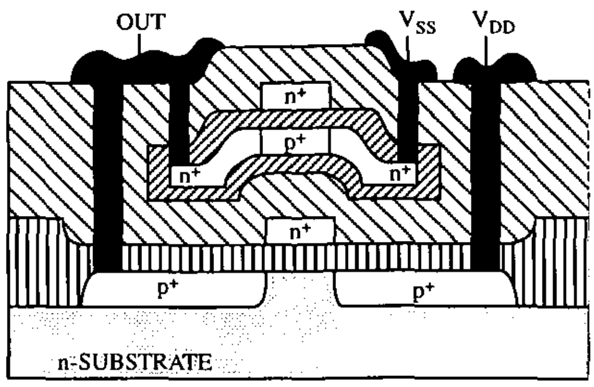


Figure 2.14 3-D CMOS Cross Section

2.2 Digital Logic Testing Techniques

2.2.1 Introduction to Digital Logic Testing

2.2.1.1 The Need For Test Strategies

As outlined in section 1 the technology of VLSI design is advancing at a rapid rate. This advancement, however, has led to many problems in the field of digital logic testing. Such problems have been around since digital logic devices first came into existence, but the growing number of circuits placed on individual chips has exacerbated them. The increase in circuit size and complexity has, in turn, led to little or no increase in the number of I/O pins on the device, which creates a further bottleneck for digital testing; more logic must be tested with the same number of I/O pins, thus making it much more difficult to actually test the chip.

Testing is assuming a larger proportion of the total product cost as a result of the growing circuit complexity, which is ironic when you consider that the software design tools that make it possible to cram more circuits on to a single chip at a reduced physical fabrication cost are, effectively, increasing the cost of circuit test. New testing strategies are emerging to cope, but the design of these is adding to the overall time required to get a product out of the door and into the marketplace. Manufacturers must balance the consequences of releasing a product that is not fully tested into the market; the time-lead gained over competitors through early release can be terribly expensive should the product suffer wholesale failure, with the return of large quantities of defective products.

The scale of the problems involved in testing complex components can be seen in that it was quoted *"...that 30% of the cost of production is spent on the testing of some LSI circuits."* [Wate82]. As device complexity increased over the years so did the scale of the problem: *"...the cost of testing can exceed 60% of the total device costs."* [AmMu88].

The field of digital logic testing is a very volatile and dynamic field. As new design techniques and architectures are introduced the corresponding testing environment needs to evolve further in order to efficiently test the new designs. Existing test methodologies also need to be refined just to try and increase their efficiency and to reduce their overall cost. Section 2.2 describes a number of different test strategies and their usefulness but it is in no way an exhaustive list of available techniques, which is beyond the scope of this section.

2.2.1.2 What is a Test ?

The idea behind testing any device is to construct an experiment to either confirm or deny a particular hypothesis or to distinguish between two conflicting hypotheses. A number of stimuli are applied to the device and the resultant responses are evaluated. By knowing in advance what the correct responses ought to be the test engineer can determine whether or not the device is working correctly.

When working with digital devices the test stimuli are known as *test vectors*. They are normally applied to the device at the input pins. The responses are observed either at the output pins or, in some test configurations, from certain test points within the device; results can be gathered in the latter case from either probes inserted into the device or from using built-in test circuitry which can route certain internal signals to the output pins if the device is currently under test. The results are compared against known good values, either by recording results from a known good device or from a computer simulation of the device.

A problem with digital testing is that if a test response indicates that the circuit has an error then all that implies is that their are one or more errors on the circuit concerned with the given test. Further testing is normally required in order to identify which fault on the device caused the error response. This process is accomplished by using what are known as *fault models* and is, essentially, the same in that test vectors are simulated against a fault-aware computer model of the device. The model is modified to contain a particular fault, or set of faults, and output responses between the fault model and the faulty digital device are compared; a duplicate or approximate response indicates that the digital device contains the faults simulated within the current fault model.

If the digital device responds correctly to all test vectors then we cannot conclude that it is fault-free; the best that can ever be said is that it does not contain any faults that the test engineer has tested for. It may well contain other faults for which no effective test vector had been applied. Even if the computer models used in simulation are extremely accurate they will not uncover defects unless effective stimuli have been applied.

2.2.1.3 The Economics of Testing

It should be possible to test every individual circuit on a digital device with every possible combination of test vectors. However, time and cost constraints mean that is simply not possible to test a product so that the final customer can be 100% certain of receiving a fully functional device. The test engineer has to find an answer to the following question: "How much testing is enough?". If a faulty product is returned then the designer incurs the cost of replacement, as well as suffering a loss of good will, which is a vital commodity in the marketplace.

Additional testing also requires additional cost, from the development of the test plan and also for the application of the test on to the actual devices - these test costs must be minimised as much as possible. Test costs are reduced so that at a reduced level of test, i.e. non-100% test coverage, the cost of any additional testing exceeds the savings gained through the reduced service and replacement of faulty products. Obviously, exceptions must be made for devices which are placed in safety-critical systems, whereby the manufacturers must only release perfect products to its clients.

It is far easier to spend more on testing if the product is to be a large-volume consumer product, where the test costs can be amortised over a large number of units. In this case a thorough and efficient test is worth creating, as it can reduce the amount of time spent testing each unit. In low-volume products testing can take up a disproportionately large amount of the overall cost and it may not be possible to justify such extensive testing. However, in safety-critical applications such additional costs for testing just have to be borne.

The economics of testing imply that we want to test for the faults that are most likely to occur. The test engineer must know which faults occur and how frequently they occur. By knowing these facts the effectiveness of a test can be measured. It is then required to get an estimate of the defect level (DL) of the device, which is the fraction of devices shipped that are still defective. This is done using the following equation [WiBr81]:

$$\mathrm{DL} = 1 - \mathrm{Y}^{(1-\mathrm{T})} \qquad (2.1)$$

This uses the yield level of the fabrication process (Y) and the test effectiveness (T). If it is possible to test for all defects (T=1) or if the fabrication process yields no defective units (Y=1) then the overall defect level equals zero - this perfect situation almost never happens.

Equation 2.1 can be restated to express the test fraction *T* in terms of the yield and defect level:

$$\mathrm{T} = 1 - \frac{\log(1-\mathrm{DL})}{\log(\mathrm{Y})} \qquad (2.2)$$

This allows the test engineer to enter the known yield level of the fabrication process along with the desired defect level, thus giving the test effectiveness level required in order to meet the requirements. By increasing the yield level we reduce the required test effectiveness level, so simply by improving the fabrication process the test engineer can reduce the task of testing the device.

Another important aspect of the economics of testing is the cost of locating and replacing a defective unit. In the case of digital devices it is vital that faulty devices are found and replaced as early as possible; this is an acknowledgement of the so-called power-of-ten rule, whereby the costs increase by an order of magnitude at each stage of integration. If it costs *N* to detect a faulty chip at incoming inspection then it will cost *10xN* if the chip escapes the inspection stage and survives to be soldered on to a circuit board. Again, if it not detected at the board-test stage then it may cost *100xN* if the defect is not detected until the board is placed in a complete system. If a defective system makes it as far as a customer the cost becomes incredible. There is a large economic incentive, therefore, for the test strategy to find defects as early as possible.

2.2.2 Combination and Sequential Logic Test Strategies

2.2.2.1 Stuck-At Fault Model

The two most common defects in digital electronics are shorts and opens; the former is where a connection exists where one should not exist and the latter is a lack of connection between two points. The application of test vectors can test the functionality of the device, but the time required for such test can be too high; an n input device requires 2^n test vectors, so with n=40 at 10 million test vectors per second we still require over 30 hours to test the circuit. Additionally, if the circuit contains any sequential logic then there is no assurance that such a technique will test every function.

Eldred took the approach that instead of testing every function capable by the device you should instead test the hardware itself [Eldr59]. The common faults are modelled using a computer simulator, whereby input stimuli are used that could differentiate between fault-free and faulty circuits. This has the advantages that not only can the effectiveness of the test strategy be evaluated but groups of tests can be assigned to specific defects so that if a test pattern returns a faulty result it is pinpointing a specific component or set of components.

The stuck-at fault model checks for the inputs and outputs of digital logic gates that are either shorted or left open. An n-input combinational circuit can implement 2^{2^n} different logic functions. In order to verify with 100% accuracy that a particular circuit implements the desired function requires all 2^n input patterns to be applied - this could take an inordinate amount of time.

Figure 2.15 shows the standard logic symbol circuit for the simple function $e = \overline{\overline{a \cdot b} + c}$. If the output of the NAND gate (signal d) is stuck-at-1 then, regardless of the second input to the NOR gate (signal c), it is simply not possible to drive the output of the NOR gate (signal e) to logic-1; so long as there is one logic-1 input to a NOR gate the output will always be logic-0. It can be seen from this example that each circuit can have a number of possible of stuck-at faults - the simple circuit in figure 2.15 has a total of 10 possible stuck-at faults, consisting of a stuck-at-0 and a stuck-at-1 fault for each and every signal in the circuit (inputs, outputs and intermediate).

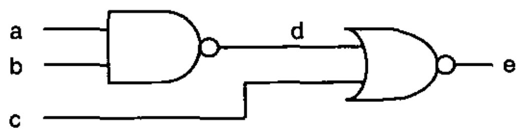


Figure 2.15 Simple Function Schematic

2.2.2.2 Sensitised Path Fault Model

A test can be defined [LePr92] as comprising of a set of input values, together with the resulting true and faulty output values. A sensitised path is where the fault can be controlled from the circuit inputs and the result is observable at the output. By testing for d stuck-at-1 ($d/1$), which results in the output $e/0$ regardless of the inputs, the path a-b-d-e is sensitised to the test. However, the faulty result $e/0$ also covers other faults along the sensitised path, namely $a/0$ and $b/0$. Hence, a single successful test can cover a number of different faults.

There is a requirement, therefore, for an efficient method of generating test patterns. A general procedure for this can be expressed as follows:

```
Prepare fault list
Repeat
•     Select a fault from the fault list
•     Generate a test
•     Check the fault coverage for the test
•     Delete these faults from the fault list
Until a satisfactory fault cover is obtained
```

In an ideal situation an acceptable fault coverage level is 100%, but a figure of between 90%-95% is often deemed acceptable. This method only really works where the internal structure of the device being tested is known but, as in many commercial devices, this information is not always available to the end-user. In these cases a number of sequential tests would be required in order to determine if a fault has occurred; i.e. a status register could be checked by loading it with specific data, carrying out some arithmetic operation on it and reading it back. The biggest problem with this method of testing is that it is difficult to determine whether or not an adequate fault cover level has been established.

2.2.2.3 D-Algorithm Fault Model

An additional problem with the stuck-at fault model arises when a single output has multiple paths, as shown in figure 2.16. Here, reconvergent fan-out exists between the first and last NAND gate, which means that it is not possible to construct a viable sensitive path in order to test for *e/1*; this is due to the fact that the point of failure on signal *e* fans out to another element of circuitry and, as it reconverges with the test path later regardless of whether the chosen sensitive path encompasses *f* or *g*, the effect of the original fault may be masked.

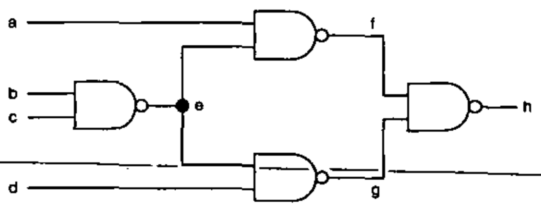


Figure 2.16 Reconvergent Fan-Out

The answer to this problem is to simultaneously sensitise all possible paths from a point of failure to the circuit outputs. This approach, known as *n-dimensional sensitisation*, is as follows [Roth66]:

(i) *D-Drive:* For each pass through the circuit all possible paths from a chosen fault site to all outputs are generated simultaneously, cancelling any convergent fan-out paths that may occur

(ii) *Consistency Operation:* Using a backward-trace procedure, the primary input conditions required to generate the static inputs for the *D-Drive* are derived

The procedures described above are based upon what is known as the *calculus of D-cubes*, which allows a formal mathematical model of a network (including fault conditions) to be set up. However, a full description of this method is beyond the scope of this section.

2.2.2.4 Boundary Scan Testing

The D-algorithm has withstood the test of time and is accepted as a good method of testing circuits; the method has even been extended so that it can cope with feedback loops within circuits that are inherent in any sequential circuit design [PuRo71]. However, as the complexity of VLSI devices increase it has become impossible, to all practical purposes, to generate test sequences that would test the complete system within the VLSI device - it became inevitable that circuits would have to be designed and placed on the device in such a way so that testing of individual circuits could take place.

The development of a procedure known as the *scan-path technique* slowly evolved that required a system to be partitioned in such a way that all bi-stable devices can be tested separately from the normal combinational logic, which could be tested using normal methods. The bi-stable devices were organised as a single long shift register, which specially designed logic on-chip could shift-in test values and shift-out test results whilst holding the device in a pause mode.

In 1985 a consortium of European and US companies established the Joint Test Action Group (JTAG). This was done with the aim of standardising a design-for-test hardware structure which was suitable for board-level testing, which, in itself, required new test strategies to be developed. The final report from the JTAG Committee, issued in 1989, was formally adopted as an IEEE standard [IEEE89].

The JTAG boundary scan technique places what is known as a *boundary scan cell* in series with each functional component pin. This cell is a multi-function circuit that can either shift-in or shift-out values to and from each pin, or can be placed in a transparent-mode in order to allow the device to operate normally. Figure 2.17 shows the outline structure of a board containing two devices: device (a) contains the required test logic but device (b) does not, and must be supplemented by external buffers containing the boundary scan cells. Not shown on figure 2.17 is that all devices on a board also contain a common *Test Mode Select* input and a *Test Clock* input.

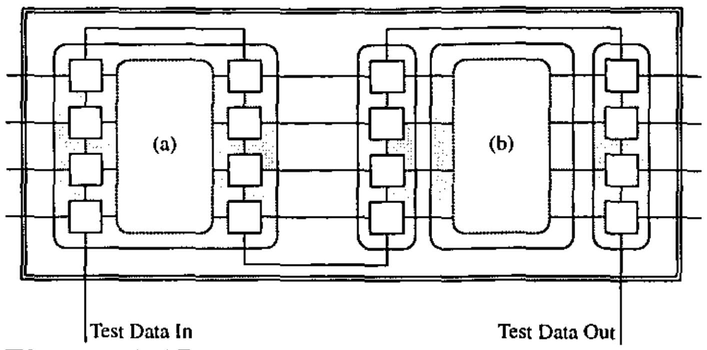


Figure 2.17 Outline of Boundary Scan Structure

The internal structure of a device that contains the boundary scan test logic is shown in figure 2.18. All devices that comply with the IEEE 1149.1 standard must contain the following components:

i) boundary scan register chain
ii) instruction register
iii) bypass register
iv) test access port controller

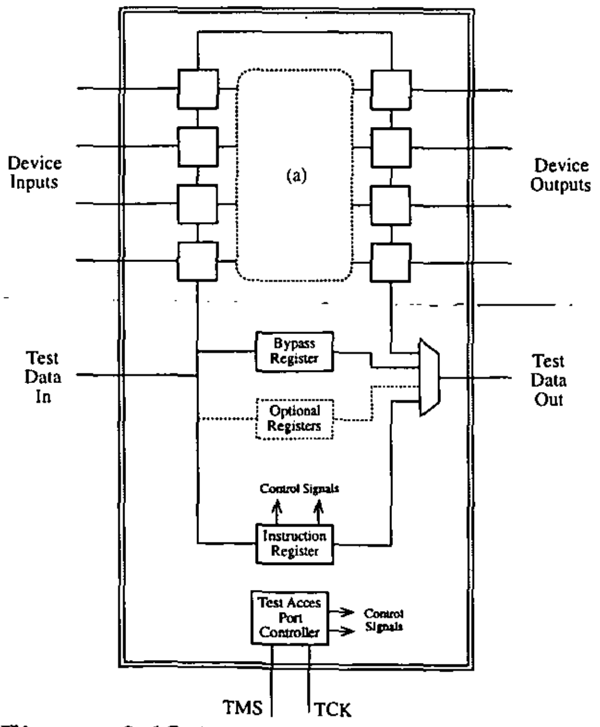


Figure 2.18 JTAG Device Architecture

The contents of the instruction register are decoded in order to provide the control signals that activate the various test facilities on-board the device. The bypass register allows the internal scan-path to be bypassed, which removes the device entirely from the board scan path - this has the side effect of shortening the overall length of the scan path and simplifies test access to the remaining devices on the board. The test access controller is a sequential circuit that is controlled and clocked by the TMS and TCK signals respectively. It generates the control signals for both the instruction and the data register.

Although the application of boundary scan techniques appears complex it provides a structured, high-level and now widely-accepted standard for design-for-test methodology at the system level.

2.3 Systolic Array Architectures

2.3.1 Introduction

2.3.1.1 Architecture Background

The study of systolic arrays involves a combination of skills from the realms of both Computer Science and Electronic Engineering. They were first proposed by Kung and Leiserson [KuLi78] as a method of implementing a simple parallel processing system, which would consist of a nearest-neighbour connected array of fairly simplistic processors or cells. A simple definition is that a systolic array is a network of processors which rhythmically compute and pass data through the system. This transfer of data from processor to processor, rather than from main memory to processor, allows for a high degree of pipelining and synchronised multi-processing, thus avoiding the classic memory access bottleneck that commonly occurs in normal Von Neumann machines.

Kung also stated [KungHT82] that computational tasks can classified into two distinct families: compute-bound and I/O-bound. If, in a computation, the total number of operations is larger than the total number of I/O operations then the computation is compute-bound, otherwise it is I/O-bound. For instance, matrix-multiplication is compute-bound, whereas matrix-addition is I/O-bound. Speeding up I/O-bound computations is difficult using current technologies, as it requires an increase in memory bandwidth. Compute-bound computations, however, can be speeded up through the use of systolic arrays, which can give a higher computation throughput without increasing memory bandwidth. The basic configuration of a systolic array compared to a normal Von Neumann machine is shown in figure 2.19, which clearly shows the potential increase in computational throughput by using systolic arrays.

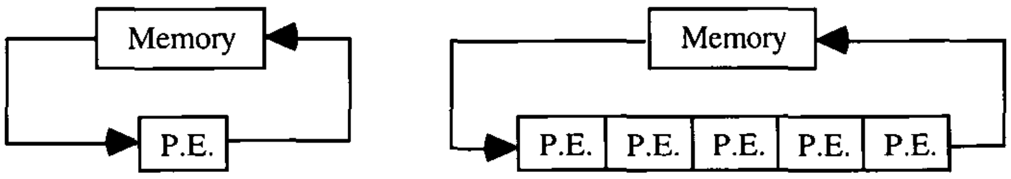


Figure 2.19 Von Neuman .v. Systolic Array Architecture

These arrays allow for the implementation of high-performance parallel algorithms, using VLSI technology to provide inexpensive implementation. Such systems can also be implemented on a software-based system, although such a system would have lower performance figures when compared against a direct VLSI implementation. The major benefit of a VLSI implementation over a software implementation is clearly in the throughput of data that a VLSI device could achieve, but such a solution, however, may be difficult to turn into a generic systolic system - interconnections within VLSI devices can be very expensive in terms of silicon area, more so than the processing elements themselves [Ston90]. Although much slower a software solution allows for easier research into connection strategies suited for specific applications.

2.3.1.2 Soft-Systolic Research - A Brief History

The field of systolic processing first came about after 1978, with the publication of the paper by Kung and Leiserson. The driving force behind all research at that time was the idea of using VLSI techniques to produce low cost devices that could be used for computationally intensive tasks, such as those in signal and image processing. The performance of such devices was not measured in the normal sense of algorithm performance, but in the data throughput obtainable between a host computer and the systolic VLSI device.

Parallel machines existed long before the advent of VLSI - these machines, however, tended to have a prohibitively high cost of manufacture. One of the major benefits of using VLSI techniques [MeCo80] is that a device with simple and regular interconnections could allow for cheap implementations and high densities - this implies both high performance and a low overhead for support components. Hence, most parallel machine designers were interested in designing parallel algorithms that had simple and regular data-flows, such that they could easily be implemented in hardware using VLSI techniques.

Unfortunately, VLSI implementation has its own problems. Interconnections between individual units within a VLSI device has, as mentioned previously, a high price - the larger the number of units on a device the larger the price. As well as data communication links multiple power, ground and system clock signals need to be distributed to every unit, with the arrival of the system clock to each unit being synchronised. Chip area is also a limiting factor, thus reducing the number of active processing units per device [Micz87] - a small device has a statistically smaller chance of containing a flaw due to the VLSI fabrication process than a large device, as it is a VLSI design rule of thumb that 50% of working ASIC's do not work in target systems first time [Ambl92]. Because of this general instability of the fabrication process few systolic designs have been implemented in silicon, with most systolic work dealing with theoretical systolic algorithms implemented in software simulators - a list of devices from the early 1980's can be found in [Megs92] and a collection of simulation works in [Evan91].

2.3.2 Systolic Architecture Definitions

2.3.2.1 What is a Systolic Array ?

There are a number of definitions of systolic arrays within the literature [KuLi78] [KungSY84] [Ullm84]. For the sake of clarity during further discussion we have adopted the definitions found in [KungSY88]. A systolic array can be defined as a computing network possessing the following features:

- *Synchrony* The data elements are rhythmically computed, via a global clock, and passed through the network

- *Modularity and regularity* The array consists of modular processing units with homogenous interconnections, whereby the network may be extended indefinitely

- *Spatial locality and temporal locality* The array manifests a locally-communicative interconnection structure, i.e. spatial locality. There is at least one unit-time delay allotted so that signal transactions from one node to the next can be completed, i.e. temporal locality

- *Pipelinability* (i.e. O(M) execution-time speedup) The array exhibits a linear rate pipelinability, i.e. it should achieve an O(M) speedup, in terms of processing rate, where M is the number of PE's. The efficiency of the array is measured by a speedup factor, T_S / T_P, where T_S is the processing time within a single processor and T_P is the processing time within the array processor

2.3.2.2 Properties of Systolic Arrays

The major factors affecting the adoption of systolic array architectures are as follows: simple and regular design, concurrency and communication, and balancing computation with I/O [KungHT82].

As has already been mentioned one of the most efficient ways at exploiting the power of VLSI design is to use a regular and simple design and replicating it wherever possible. Great savings in design and testing can be achieved by using this technique. Another advantage is that simple and regular designs are more likely to be modular and will probably be adjustable to various performance targets and goals.

The use of concurrency is dependant upon the underlying algorithms employed by the system, as the use of concurrency is a major factor in the potential speedup of computer systems. Inter-PE communication becomes significant as the number of PE's increase, as in VLSI design it is the routing costs that dominate the power, time and silicon area required to implement any computation [MeCo80].

The ultimate performance goal of an array processor system is to have a computation rate that balances the available I/O bandwidth. Systolic arrays processors are typically attached to some host system, through which it receives data and outputs results. With a low-bandwidth system, which is typical of today's technology, an array processor must carry out multiple calculations per I/O access. This requires data items for computations to remain within the array processor for a number of internal clock cycles. Therefore, I/O bandwidth not only affects the speed of external data transfers but also the amount of on-board local memory that the array processor requires.

This problem of I/O is accentuated when we have a large dimension problem being calculated on a small array. This inevitably requires the problem to be decomposed, the partitioning of which is in itself a non-trivial task. Hence, the decomposition methods used and the decisions on how to use buffer memory to minimize I/O problems are critical to the practical implementation of an array processor system.

2.3.2.3 Inner Product Step Processor

The so-called inner product step (IPS) is the single operation most common to all processors within the processor array. It is a very simple operation, and is given by $C' \leftarrow C + A \times B$ Such an expression implies that each processor must contain at least three registers, denoted R_a, R_b and R_c respectively. Each register will require two connections, one for both input and output, and a schematic for the geometries for both rectangular and hexagonal elements is shown in figure 2.20. The first type of geometry, the square processor, is used for matrix-vector multiplication and for the solution of triangular linear systems, whilst the second type of geometry, the hexagonal processor, is used for matrix-matrix multiplication and LU-decomposition [MeCo80].

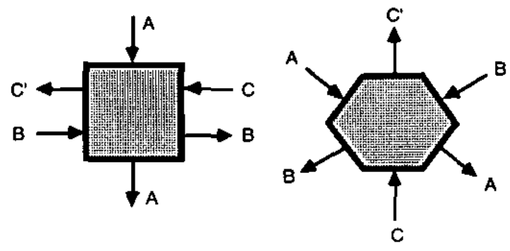


Figure 2.20 Geometry for IPS Processors

2.3.3 Linear Connected Systolic Array

IPS cells in a linear topology are arranged in a linear array of processing units, as shown in figure 2.21, using instance of the 'square' geometry shown in figure 2.20. All processing units receive external data to form their 'A' data, with the units at either end of the linear array receiving external data to form the 'B' and 'C' - internal 'B' and 'C' data is generated by the processing elements and then passed to adjoining elements.

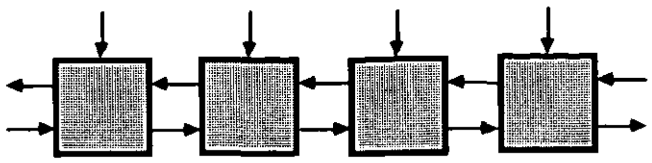


Figure 2.21 Linearly Connected Systolic Array

Each of the processors in the systolic array operate in discrete time units. During that time unit each processor shall input values into its three registers (R_a, R_b and R_c) and calculate the new value for R_c. This new value shall be made ready for output on the following time unit. All output values are latched, such that the changing of an output does not affect the input of any other processor during a single time unit.

This type of processing unit has a very modest hardware demand in VLSI terms - the units themselves are uniform, the inter-process connections are regular between adjacent units and there are very few external connections. The construction of a VLSI device containing an array of such processors should prove to be very cost effective.

2.3.3.1 Convolution Problem

Perhaps the simplest algorithm for a linear systolic array is one to perform basic mathematical convolution - it is also convenient in that it shows many of the features of systolic designs. This problem can be stated simply: compute the recurrence

$$y_i = w_1 \cdot x_i + w_2 \cdot x_{i+1} + \ldots + w_n \cdot x_{i+n-1} \tag{2.5}$$

for $i = 1 \ldots n$ given the sequence $w_1, w_2, \ldots, w_k$ and $x_1, x_2, \ldots, x_n$, where n and k are positive integers - the w_i values are often referred to as weights. There are many different ways to construct a systolic array to perform this task, but only a single variation is looked at with the configuration shown in figure 2.22, although there are other examples [KungHT82].

This design consists of four cells, each with two inputs and two outputs - the *x* values move to the right, the *y* values move to the left and the *w* values (weights) are pre-loaded into their respective cell where they remain for the duration of the calculation. The definition for each cell, along with the input-output relationship and calculations performed, is shown in figure 2.22.

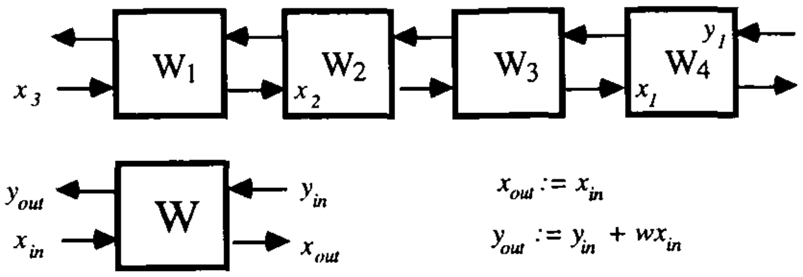


Figure 2.22 Bi-Directional Systolic Array for Convolution

The algorithm works in a series of cycles. During such a cycle data is transferred along the arcs on the array - this will involve either inter-cell transfer or I/O from the entire array. Data values for *x* are input to the array on alternate cycles, with result values *y* also being output by the array on alternate cycles. Note, valid values for both input and output do not occur on the same cycle, with zero being input to the array if an *x* value is not scheduled for input and the *y* values output on nonscheduled cycles are just ignored. On every cycle, however, a zero value is input on the right side of the array to form initial values for the cell summation calculation. All cells in the array are assumed to operate in parallel, with some form of synchronous timing method being adopted throughout the entire array (although asynchronous timing is possible).

2.3.3.2 Matrix-Vector Multiplication Problem

Kung and Leiserson (1978) proposed that a simple linear array such as that shown in figure 2.22 is sufficient for a large number of important algorithms, one of which is the commonly used matrix-vector multiplication algorithm, which occurs abundantly in neural network applications.

The basic problem [MeCo80] is that we wish to multiply a matrix denoted $A = (a_{ij})$ by a vector denoted $x = (x_1, \dots, x_n)^T$ - the result formed is denoted $y = (y_1, \dots, y_n)^T$. This result can be formed by calculating the following recurrence:

$$\begin{aligned} y_i^{(1)} &= 0 \\ y_i^{(k+1)} &= y_i^{(k)} + a_{ik} \cdot x_k \\ y_i &= y_i^{(n+1)} \end{aligned} \tag{2.6}$$

Figure 2.23 shows an example set of data. The matrix A is an *n-by-n* band matrix with a band width $w = p + q - 1$. The recurrence equations above can easily be calculated by sending the x and y values through w linearly connected processors, with each processor being pre-loaded with the correct 'weight' value a_{ij} from the matrix for the calculation to be performed.

$$\begin{array}{c} \leftarrow p \rightarrow \\ \begin{array}{c} \uparrow \\ q \\ \downarrow \\ \\ \\ \\ \end{array} \begin{bmatrix} a_{11} & a_{12} & & & & \mathbf{0} \\ a_{21} & a_{22} & a_{23} & & & \\ a_{31} & a_{32} & a_{33} & a_{34} & & \\ & a_{42} & a_{43} & a_{44} & a_{45} & \\ & & a_{53} & a_{54} & a_{55} & a_{56} \\ \mathbf{0} & & & a_{64} & a_{65} & a_{66} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \\ \qquad\qquad A \qquad\qquad\qquad\qquad\qquad x \qquad\quad y \end{array}$$

Figure 2.23 Matrix-Vector Multiplication Problem Data

The key to the whole process is the spatial location of all data at each step of the systolic process. The connections between all of the cells, along with the direction of all data transfers, is shown in figure 2.24. All y_i values, which are initially set to zero, move to the left, while the x_i values are moving to the right and the a_{ij} values are moving down - all moves are synchronised by use of a single global clock signal. Due to the arrangement of the data values it turns out that each y_i term is able to accumulate all of its product terms before it leaves the linear array of processors. Note that in the example

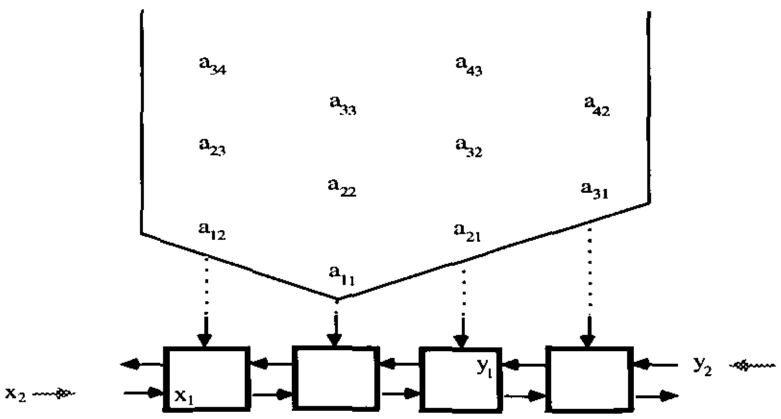


Figure 2.24 Data Connection Map for Matrix-Vector Multiplication

shown in figure 2.24 the first product term is created in the second processor, and that all unspecified values are assumed to be zero.

During any particular cycle alternating processors are idle - the effective y_i values within them are zero, as are the a_{ij} values. Hence the cell performs no useful operation during that cycle. It can be surmised, therefore, that it may well be possible to reduce the number of processors required for this operation on a band matrix with band width w from w to $w/2$.

As mentioned previously the synchronisation of the data transfers is of vital importance, as is the rendezvous in the cells of the correct data at the correct time. Each IPS cell has three registers, which hold the values for A_{ij}, x_i and y_i respectively, and all are initialised to zero. All cells are numbered by integers from left to right, starting from 1. Each step of the operation consists of just two operations, but for odd numbered time steps only odd numbered cells are active, and vice versa.

The basic algorithm can be stated as follows:

Operation #1 - Shift Register Values

- R_a receiving a new element from the band in matrix A
- R_x receiving the contents of register R_x from the left neighbouring node, with R_x in the first processor receiving a new component from the vector x
- R_y receives the contents of register R_y from the right neighbouring node, with R_y in the first processor outputting its value and R_y in the w^{th} processor receiving a zero value

Operation #2 - Multiply & Add

- $R_y' \leftarrow R_y + R_a \times R_x$

The operation of the algorithm over a series of operations can be seen in figure 2.25.

It can easily be seen that the three shift components of operation #1 can all be done simultaneously and that the completion of both operations constitutes a single cycle of the systolic algorithm. The figure show the traversal of the data values during the initial cycles, along with the cumulative values inside the *y* registers in each processor. Given that the bandwidth of any matrix *A* is denoted by *w* it can be seen

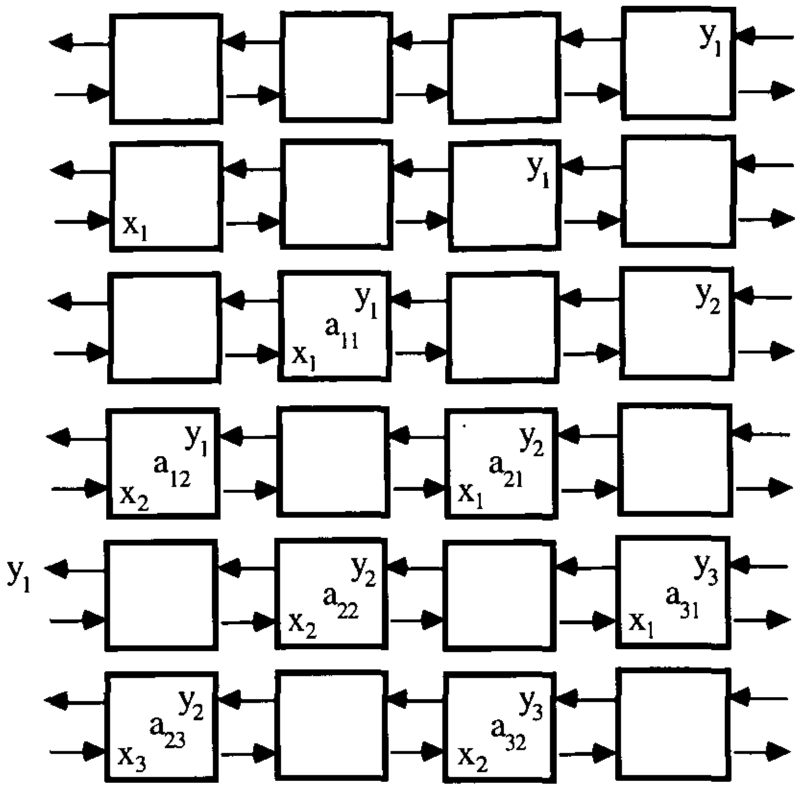


Figure 2.25 Initial Cycles of Matrix-Vector Multiplication

that after *w* cycles the components of the product $y = Ax$ will start shifting out of the left-most processor - the rate of output is one valid data item per 2 cycles. Hence it can be seen that all *n* product values can be output in $2n + w$ cycles, which is a large improvement on the *O(wn)* time requirement of a sequential algorithm implemented on a uni-processor system.

2.3.4 Dense Matrix Systolic Arrays

Two-dimensional systolic arrays have provided the mechanism to carry out computations on two dense matrices. Both rectangular and triangular arrays are feasible, both for the computation of matrix products and solving systems of linear equations. Both methods are efficient in their use of execution time, being far more attractive than the normal linear array.

2.3.4.1 Rectangular Array : Matrix-Matrix Multiplication

The product of two matrices of order *n* is nothing more than n^2 scalar products, each consisting of *n* terms. Thus, when calculating **C=A.B**, the element c_{IJ} is derived from the scalar product of the first row of **A** and the first column of **B**. There are two simple solutions to this problem: the rectangular array and the square array.

For the rectangular array [KuLe80] this can be achieved using multiple instances of the standard multiply-and-accumulate (MAC) cells, as shown in figure 2.20. For *n=3* three MAC cells are required to do the job, as shown in figure 2.26. This will calculate a single scalar product. By placing two identical networks to the right of figure 2.26 the array can also compute the scalar products of the first row of **A** and all three rows of **B**, giving c_{12} and c_{13}. By pipelining the computation, sending the other rows of **A** after the first, the array can calculate c_{22}, c_{23} and c_{33}.

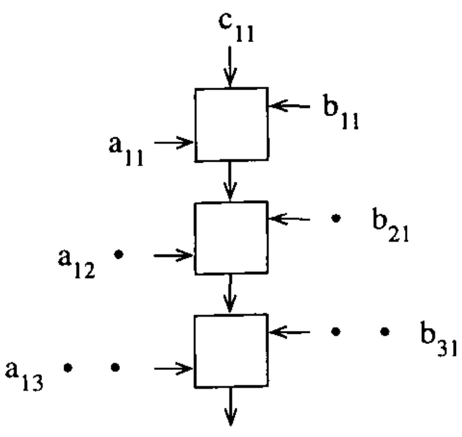


Figure 2.26 Rectangular Array for Scalar Products

The final elements of **C**, namely c_{21}, c_{31} and c_{32}, require two more copies of figure 2.26 to be added to the left of the array. As in algorithms for the linear array a delay needs to be introduced into the network between every pair of consecutive elements. The final systolic array matrix for this operation is shown in figure 2.27.

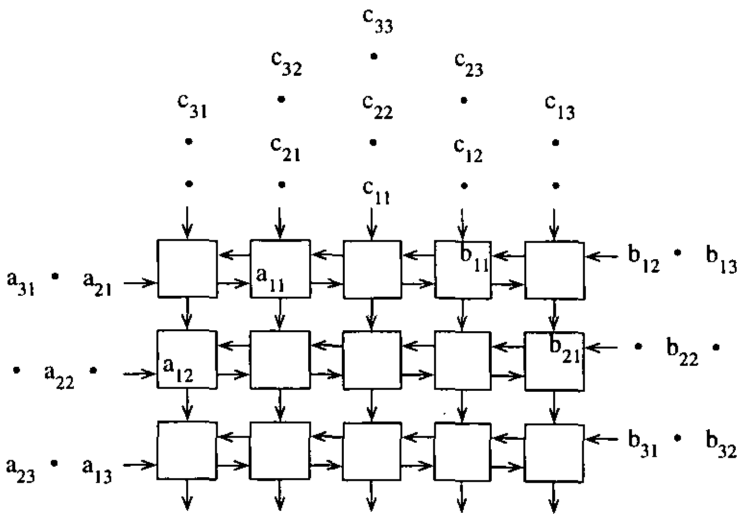


Figure 2.27 Full Rectangular Array for 3x3 Scalar Products

This type of systolic array can carry out a matrix-matrix multiplication of two n-by-n matrices in *4n-3* cycles. There are a total of *2n-1* scalar product arrays, with the processor matrix for the operation being *(2n-1) x n* cells in size.

Implementation in a square array requires a different version of the standard MAC cell, which calculates the scalar product in a completely different manner. The same calculation occurs as in the original MAC cell, but the flow of data is different.

The MAC cell for required for this operation is shown in figure 2.28. The accumulated components for **C** remain within the cell, with components of **A** and **B** being passed through the cell. After a number of internal accumulations the cell holds the final value for a single component of **C** inside an internal register, ready for further computations or for output from the systolic array.

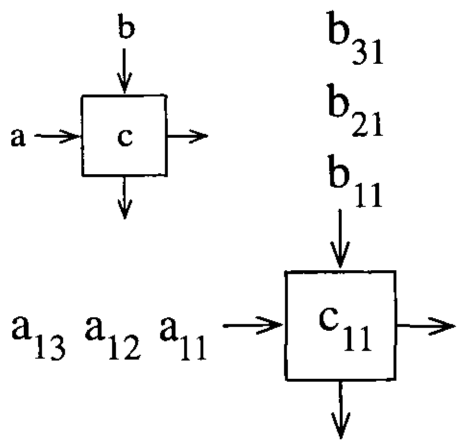
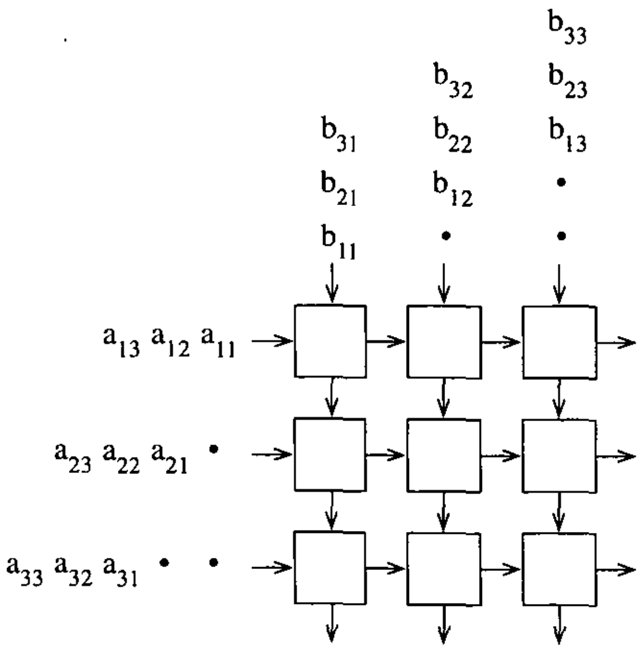


Figure 2.28 Square Array MAC Cell Configuration

The systolic array required to compute the scalar product of two n-by-n matrices consists of n^2 cells [PrVu80]. It takes precisely $3n$-2 time steps for the scalar product computation to complete; the full systolic array for this operation is shown in figure 2.29. The one drawback with the square array method is that the results of the computation reside within the internal registers of the processors. It is not possible to output all n^2 components of **C** in a single cycle, as only the MAC cells at the edge of the systolic array have any external I/O capability. Additional processing is required to gain access to the final values of **C**.



b_{33}
b_{32} b_{23}
b_{31} b_{22} b_{13}
b_{21} b_{12} •
b_{11} • •
a_{13} a_{12} a_{11}
a_{23} a_{22} a_{21} •
a_{33} a_{32} a_{31} • •

Figure 2.29 Full Square Array for 3x3 Scalar Products

The simplest solution to this problem [QuRo89] is to add some control structures to the cells so that once an accumulation in any cell has completed it is instructed to pass the contents of the internal register holding the component of **C** to the cell on it's right. Another solution, which is especially useful for systems that repeatedly require access to the data within **C** [KaEv96], is to simply hold the result matrix **C** within the cells, accessing it as and when desired by whatever computation is required by the systolic array. This type of solution requires additional internal registers in the cells to hold results of short-term and long-term calculations. It has the benefit, however, of drastically reducing I/O costs when dealing with the matrix **C**.

2.3.4.2 Triangular Arrays : Solution of Linear Systems

A common problem to be solved with triangular arrays is the solution of linear systems such as $\mathbf{A}x = b$, where the matrix **A** and the vectors x and b both have the order n. With **A** being a lower triangular matrix then the solution vector to the system, $x = (x_1, x_2, \ldots, x_n)$, can be computed with the recursive equations

$$\begin{cases} x_i^{(0)} = 0 \\ x_i^{(k)} = x_i^{(k-1)} + a_{ik} \cdot x_k \\ x_i = \left(b_i - x_i^{(i-1)}\right) / a_{ii} \end{cases} \tag{2.7}$$

for all $1 \le k \le i-1$. A systolic array similar to that used for the calculation of matrix-vector products can be used to solve this system, but with cells relating to the upper triangular portion of the matrix **A** being deleted from the array.

The array has two types of cell [KuLi80]: the standard square multiply-and-accumulate MAC cells, and additional 'round' cells at the edge of the array which calculate the final components of the solution vector x using a subtraction/division operation instead of an addition/multiplication operation.

These additional round cells also re-route the calculation back into the array, in order to allow further processing to be carried out. The operation of the round processor is shown in figure 2.30. The components of x pass through the array to accumulate the partial products with the array components a and then the round cells at the edge of the linear array computes the final value for the component x. The entire linear array for the process, with $n=4$ is shown in figure 2.31.

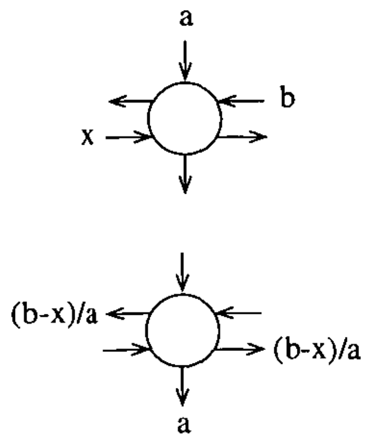


Figure 2.30
Triangular System Cells

In the case of solving a linear system where the matrix is not of a lower triangular type then some additional computations are required.

Given the linear system $\mathbf{A}x=b$, where $\mathbf{A}$ is a dense matrix system of order n then in order to solve it on the linear array shown in figure 2.31 the system has to be transformed into the following equivalent triangular system:

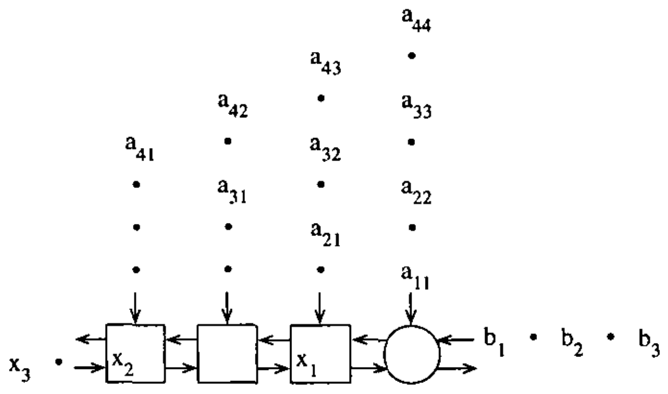


Figure 2.31 Triangular System Linear Array

$$\mathbf{T}x=b' \tag{2.8}$$

The translation from the dense-matrix array to the triangular array is a complex process and is covered in the extensive literature on the subject [GeKu81] [Ahme82] [Cosn86a]; the remainder of this section will briefly describe the methods and architecture proposed by Gentleman and Kung [GeKu81].

The translation to the equivalent system show in equation 2.8 is done with the help of a triangularisation schema which is applied to matrix $\mathbf{A}$ augmented on the right hand side with b, using either Gaussian or Givens elimination matrices. The system matrix is reset for the processes to be $\mathbf{A} \leftarrow (\mathbf{A}, b)$, giving a matrix of size *n x (n+1)*.

This schema can be represented as follows:

```
for $k = 1$ to $n-1$
    for $i = k+1$ to $n$
        $\begin{pmatrix} row\ k \\ row\ i \end{pmatrix} = \boldsymbol{M}_{ik} \cdot \begin{pmatrix} row\ k \\ row\ i \end{pmatrix}$
```

The matrix $\mathbf{M}_{ik}$ is chosen in a way so as to zero the coefficient in position *(i,k)*. At step *k* the row *k* is known as the pivot row, and is combined with all lower rows to zero all elements in the k^{th} column which are below the diagonal. Thus, after step *n-1* the resultant system is triangular. The method for choosing the matrix also depends on the properties of the original system matrix **A**. The method is not allowed to introduce any pivoting techniques, as that would destroy the regularity of the systolic array.

The Gaussian method is used for positive or diagonal matrices, whilst in the more general case orthogonal factorisation matrices (called Givens matrices) are used:

Gaussian Matrix
$$\mathbf{M}_{ik} = \begin{pmatrix} 1 & 0 \\ -\frac{a_{ik}}{a_{kk}} & 1 \end{pmatrix} \tag{2.9}$$

Givens Matrix
$$\mathbf{M}_{ik} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \quad where\ \theta = arctan\left(\frac{a_{ik}}{a_{kk}}\right) \tag{2.10}$$

A rather useful compromise solution [Sore85] is to modify the Gaussian method, using a technique known as *neighbour pivoting*: if the ratio $|a_{ik}| / |a_{kk}| < 1$ then the matrix is generated as given in equation 2.9; if not then the rows *i* and *k* are exchanged before the combination factor is applied, thus modifying the task of generating the matrix to the following:

$$\mathbf{M}_{ik}=\begin{pmatrix}1 & -\frac{a_{kk}}{a_{ik}}\\ 0 & 1\end{pmatrix} \tag{2.11}$$

An important point using this technique is to only generate combinations of those rows whose coefficients are of absolute value less than one, which was proven by Sorenson [Sore85] to be mathematically stable.

Gentleman and Kung [GeKu81] used a triangular array of processors connected orthogonally, containing $\frac{n\cdot(n+1)}{2}+n$ processors, using a combination of two different types of cell. Each processor carries out two distinct operations: an initialisation process and a computation process. During the initialisation process all internal registers in the processor are set to zero. The two cells required for Gaussian elimination with neighbour pivots is shown in figure 2.32 and their respective programs are shown below.

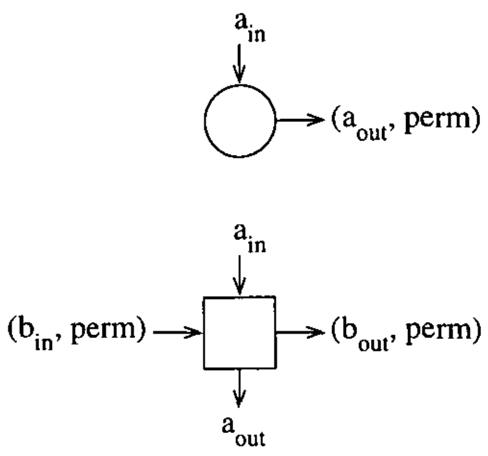


Figure 2.32 Cell Topology for Gaussian Elimination

Round Cell

```
case init
{
TRUE:   $r := a_{in}$;
        init := false;
        break;

FALSE:  if   $|a_{in}| \le |r|$
        then $a_{out} := -a_{in}/r$;
             perm := false;
        else $a_{out} := -r/a_{in}$;
             $r := a_{in}$;
             perm := true;
        endif;
        break
}
```

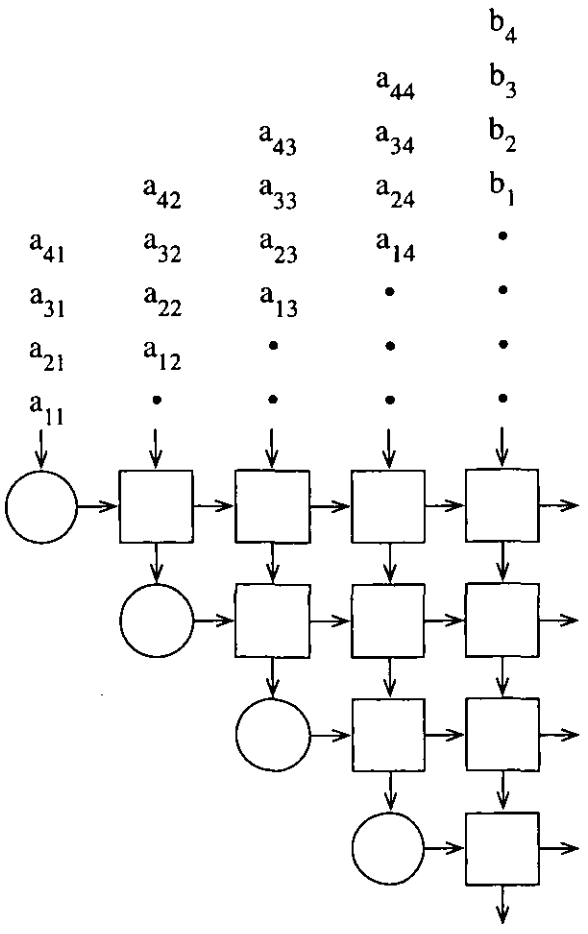
Square Cell

```
case init
{
TRUE:   $r := a_{in}$;
        init := false;
        break;

FALSE:  if   perm
        then $a_{out} := r+a_{in}.b_{in}$;
             $r := a_{in}$;
             $b_{out} := b_{in}$;
        else $a_{out} := a_{in}+r.b_{in}$;
             $b_{out} := b_{in}$;
        endif;
        break;
}
```

The full array used for this process is shown in figure 2.33. Details of the algorithm and intermediate register values are not given here, but an informal description of such things can be found in [QuRo89].

At the end of the algorithm the values for **T** and *b'* are held within the systolic array, and some arbitrary method of emptying the array is needed. The actual contents of the array are as follows:

$$\begin{pmatrix} t_{11} & t_{12} & t_{13} & t_{14} & b_1' \\ - & t_{22} & t_{23} & t_{24} & b_2' \\ - & - & t_{33} & t_{34} & b_3' \\ - & - & - & t_{44} & b_4' \end{pmatrix}$$



b_4 b_3 b_2 b_1
a_{44} a_{34} a_{24} a_{14}
a_{43} a_{33} a_{23} a_{13}
a_{42} a_{32} a_{22} a_{12}
a_{41} a_{31} a_{21} a_{11}

Figure 2.33 Triangularisation of a Dense Matrix *(n=4)*

Once the triangular system has been generated there remains the task of solving the triangular system. Although this can be done on the linear array described earlier in this section the row-wise output format of the triangularisation process is incompatible with the diagonal-wise input of the linear array.

There exist methods, however, that solve the triangular system on-the-fly [Cosn86b] using a slightly different array to that shown in figure 2.33, thereby making the solution of dense matrix linear systems a relatively simple process for a systolic array architecture.

3

ARTIFICIAL NEURAL NETWORKS

INTRODUCTION

This chapter presents a description of the field of neural network systems. It begins with a discussion of neural networks themselves, with biological information and details of the early attempts to create artificial neural networks. It goes on to describe several of the more common learning algorithms in use today, giving examples of both supervised and unsupervised learning. The chapter concludes by describing some of the less common algorithms, which normally have more specialised uses and are not normally used for everyday problems.

3.1 Neural Network Overview

3.1.1 What are Neural Networks ?

Neural networks systems are biologically inspired, meaning that they are composed of elements that perform in a manner that are analogous to the most elementary functions of the biological neuron. These elements are organised in a manner that may (or, probably, may not) be related to the anatomy of the human brain. Despite the fact that the resemblance to real neurons is, at best, only superficial they still manage to exhibit a surprising number of the brain's characteristics: they can learn from experience, they can abstract the essential characteristics from an input data set that may contain irrelevant data (such as random or periodic noise) and they have the capacity to generalise from previous known inputs to new unknown information.

More than any other factor it is probably the learning capability of neural networks that has been responsible for the interest that the field has received. The networks has the ability to modify their behaviour; by responding to an environment, which is normally a set of input vectors, the network can self-adjust in order to produce consistent results. There are many different algorithms available that dictate how the networks modify their behaviour; they are known as *training algorithms*, and each of these have their own strengths and weaknesses.

Thus, with the idea of a neural network system being an adaptable machine we can use the following definition of a neural network [Hayk94]:

A neural network is a massively parallel distributed processor that has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:

1. *Knowledge is acquired by the network through a learning process*
2. *Interneuron connection strengths, known as synaptic weights, are used to store the knowledge*

3.1.2 Biological and Physiological Background

The basic unit of the nervous system is the individual nerve cell or *neuron* [Vand86], which occur in a variety of sizes and shapes. Nevertheless, as shown in figure 3.1, most of them consist of four basic parts: the cell body, the dendrites, the axon and the axon terminals. The dendrites form a series of highly branched outgrowths from the cell body. The dendrites and the cell body are the sites of most of the specialised junctions where signals are received from other neurons. The dendrites, in effect, increase the surface area of the cell body membrane, thus increasing the room available for incoming signals from other neurons.

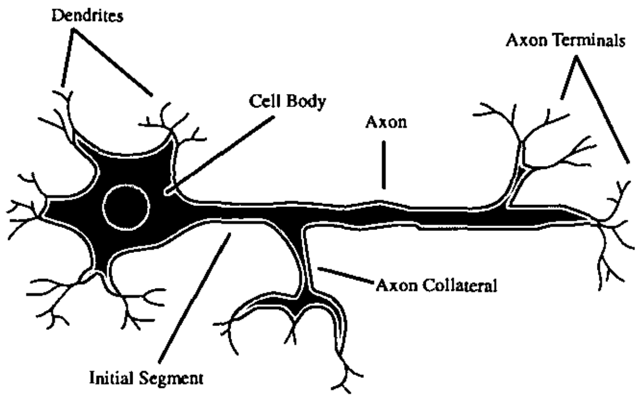


Figure 3.1 Diagrammatic Representation of a Biological Neuron

The axon, or nerve fibre, extends from the cell body. The first part of the axon, along with the part of the cell body where the axon is joined, is known as the initial segment. It is here that the electric signals are initiated in many neurons, where they then propagate away from the cell body. The axon may give off branches, known as collaterals, along it's course and, near their ends, the main axon and associated collaterals undergo considerable branching. Each branch ends in an axon terminal, and it is these terminals that are responsible for transmitting signals, which are electro-chemical in nature, from the neuron to the cells contacted at the axon terminals. Note that not all neurons behave in this manner - this is just an illustration of a general case.

The axon can be treated as a cylinder, with a difference in electro-chemical potential between the outside and the inside. The normal level of approximately -70mv is the resting potential. Whenever a local change in potential occurs a current will flow between that region and an adjacent region which is at it's resting potential. Current will always flow between two points if there is a difference in potential and there is a conducting material in between.

The local current is carried by ions such as potassium (K^+), sodium (Na^+) and chlorine (Cl^-). The flow is much like water flowing through a leaky hose - charge is lost as the current flows along the axon membrane, because the membrane is permeable to the ions that carry the electro-chemical charge. The result of this is that the current magnitude will decrease with distance away from the initial site of the potential charge.

These potentials can also be summed, either temporally or spatially - figure 3.2 illustrates this point, showing membrane potentials (E*m*) and various stimuli (^). The three graphs show that potentials can vary in strength (i), that they are conducted in a decremental fashion over the length of the axon (ii) and that they can be summed in two different ways (iii). Note, although it is not shown on figure 3.2 potentials can be lower than the resting potential; increases towards and above 0mv are known as depolarisations whilst those that reduce the potential are known as hyper-polarisations. These potentials are known as *graded potentials.*

The other type of potential is an *action potential*, which allows for transmission of signals over much greater distances, although only a few cells can operate in this manner. If the potential is raised above a certain threshold, normally within 10-15mv of the resting potential, then a very rapid alteration in the membrane potential will occur, typically lasting only 1ms. The membrane potential may change from -70mv to +30mv before it re-polarises back to it's original resting potential value of -70mv.

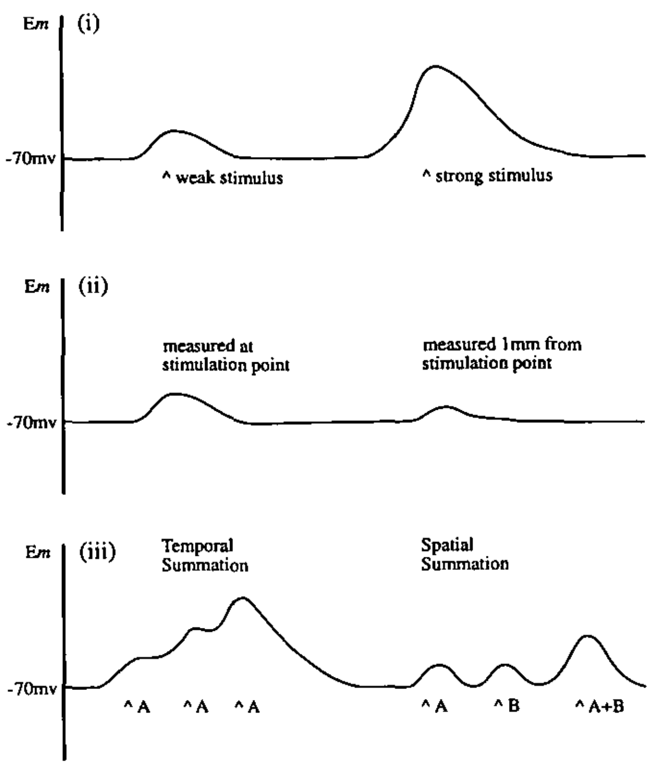


Figure 3.2 Graded Potential Possibilities

The output of any cell that can support action potentials is an all-or-nothing-response [Darn86] and, as such, cannot be summed. Once the depolarisation of the membrane crosses the threshold the output is constant for a particular type of cell; it is independent of the initiating event. Once the action potential finishes the membrane drops below the resting potential and there is a slight time-lag before the resting potential is regained; further action potentials cannot occur during this time.

Between the source and destination of potentials there is a structure known as a synapse, which acts as a chemical transmitter. Thus, the pre-synaptic cell generates a wave and the post-synaptic cell receives a wave. The synapse also alters the potential depending on whether or not it is an excitory or inhibitory synapse. Each synapse in a group which share post-synaptic cells can have different accentuation or attenuation levels. Although not drawn in figure 3.1 each axon terminal attaches to the pre-synaptic side of a synapse and the beginning of each dendrite attaches to the post-synaptic side.

In summary, each neuron cell body has a number of dendrites which act as inputs to the cell by carrying an electro-chemical charge, which can be either an action or a graded potential. Each of the potentials are scaled by a value dependant on the type of synapse that the relevant dendrite is attached to. If the resultant summed signal over all of the dendritic inputs is above a set threshold value for the neuron cell body then the cell will fire an action potential of around 30mv down its axon, else it will remain at its resting potential level of -70mv. It is this basic model that the majority of artificial neural networks are based on.

3.1.3 Historical Perspective of Neural Networks

3.1.3.1 Early Artificial Neural Network Research

The struggle to understand the brain owes much to the pioneering work of Ramón y Cajál [Ramó11] who introduced the idea of neurons being the primary constituents of the brain. The brain is a highly complex, non-linear parallel computer, which has the capability of organising neurons in order to perform certain computations, such as pattern recognition or motor control. However, the brain performs such operations many magnitudes faster than even the most powerful of today's computers. Shepherd and Koch estimated [ShKo90] that there is somewhere in the order of 10 billion neurons in the human cortex, with around 60 trillion associated synapses or connections. Neural network research has some way to go before such a parallel neural machine can be constructed.

It was not until the 1940's that the first real workable paper was published by McCulloch and Pitts [McPi43]. This paper presented the first sophisticated study of what they termed neuro-logical networks. In their work they combined the new ideas of linear threshold decision elements, finite state machines and some representations of various forms of memory and behaviour.

It was from their ideas that the field of cybernetics emerged which attempted to combine the essential concepts from fields such as biology, psychology, engineering and mathematics. In this field researchers attempted to find network architectures which could perform some specific functions, although this goal was later dropped in favour of the idea of creating a machine that had the capability to learn.

3.1.3.2 Hebbian Learning

Unfortunately, the concept of learning is not very well defined, and practical neural network research had to wait for the psychologists to come up with a model for human learning. Donald Hebb proposed a learning law [Hebb49] that became the starting point for artificial neural network training algorithms. In essence, Hebb proposed that a synapse connecting two neurons is strengthened whenever those two neurons fire. This may be thought of as strengthening a synapse according to the correlation between the excitation levels of the two neurons that it connects. For this reason so-called Hebbian learning is sometimes called correlation learning. The idea can be expressed as follows:

$$w_{ij}[t+1] = w_{ij}[t] + \alpha \cdot \mathrm{NET}_i \cdot \mathrm{NET}_j \tag{3.1}$$

where

$w_{ij}[t]$ =	the synaptic strength from neuron *i* to neuron *j* prior to adjustment
$w_{ij}[t+1]$ =	the synaptic strength from neuron *i* to neuron *j* after adjustment
α =	the learning rate coefficient
NET_i =	the excitation of the source neuron
NET_j =	the excitation of the destination neuron

Hebb's idea managed to answer the question of how learning could take place without a teacher. In this system learning takes place on a local level, involving only two neurons and a single synapse; no other feedback systems are required for the neural patterns to develop. Many successes were obtained using this method, but some patterns just could not be learned. there have been numerous extensions to this training algorithm and many more training algorithms have since been developed, most of which owe a great debt to Hebb's work. In particular, Rosenblatt [Rose62], Widrow [Widr59], Widrow and Hoff [WiHo60] and many others developed supervised learning algorithms, producing networks that learned a broader range of input patterns, and at higher learning rates, than could be accomplished by using simple Hebbian learning.

Progress in the 1950's and 1960's was rapid. A number of different researchers combined biological and physiological insights to produce artificial neural networks, which were initially implemented in electronic circuits. These successes produced a burst of activity and optimism, with researchers such as Marvin Minsky, Bernard Widrow and Frank Rosenblatt all developing artificial neural networks consisting of a single layer of neurons. It seemed that the key to intelligence had been found; perhaps all one had to do in order to produce a mechanical human brain was to construct an artificial neural network with enough neural cells. It did not take long to dispel this belief.

3.1.3.3 Minsky's Perceptron

It is well known that in order for a machine to recognise the pattern X then it must possess, at least potentially, the capability to represent X. The model commonly used for this in the 1960's was the *perceptron* model. Like the model previously proposed by Hebb the perceptron model simply could not represent certain things; if a failure occurred during the training process it seemed that neither prolonging the training experiments nor building a larger network would be of any help. The moral of this seemed simple - you cannot learn enough simply by studying learning itself; you also have to understand the nature of what it is you are trying to learn.

In the landmark book *Perceptrons* [MiPa69] the authors proved that single-layer artificial neural networks were theoretically incapable of solving many problems, including the function performed by the simple exclusive-or logic gate. They were also not very optimistic about the future: they could not see how the benefits of single-layer networks could be carried forward to multi-layer networks, stating that *"Perhaps some powerful convergence theorem will be discovered, or some profound reason for the failure to produce an interesting 'learning theorem' for the multi-layered machine will be found."* (pp 232).

Unfortunately, the conclusions of Minsky and Papert were unassailable. Discouraged researchers left the field for areas of greater promise and, more importantly, greater funding. Government agencies redirected their funding and the field of artificial neural networks lapsed into obscurity for nearly two decades. The dedicated few who continued, such as Stephen Grossberg or Teuvo Kohonen, were often underfunded and unappreciated. They found it difficult to find publishers, which is why work published during the 1970's and early 1980's is found scattered amongst a wide range of journals. Gradually a new theorem emerged, upon which the more powerful multi-layered networks of today are being built upon - these networks now routinely solve the problems that Minsky and Papert proposed in *Perceptrons.*

It can be said that Minsky's excellent work led to an unfortunate recess in the progress in the field. There is also no doubt, however, that the field was dogged by unsupported optimism and an inadequate theoretical basis. Perhaps the shock provided by *Perceptrons* allowed a period for the necessary maturation of the field.

3.2 Common Neural Network Learning Methodologies

3.2.1 Perceptron Learning

The first real practical work on neural networks [McPi43] was based around the simple Σ model: an artificial neuron receives a number of inputs x, each scaled by an associated weighting factor w, and if the sum of these scaled inputs $x.w$ exceeds a set threshold value then the neuron fires (outputting a logic-1) else it remains inactive (outputting a logic-0). Networks utilising this type of structure usually consist of a number of such neural units, each arranged as a single layer and connected to the same set of inputs, became known as *perceptrons.*

3.2.1.1 Neuron Configuration

A typical perceptron neuron has the structure as shown in figure 3.3. It shows a number of inputs being scaled by some factor, followed by a

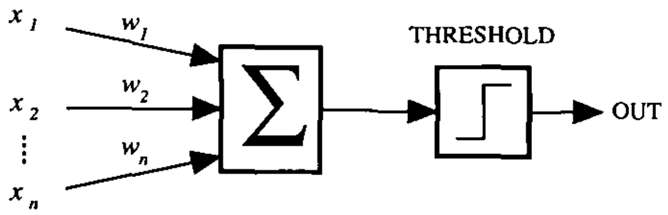


Figure 3.3 Perceptron Neuron Structure

summation of all scaled values, the result of which is fed into a threshold unit which determines whether or not the neuron fires. The output of the neuron is either a logic-0 or a logic-1.

3.2.1.2 Learning Algorithm

The learning process for perceptrons was originally defined by Rosenblatt [Rose62]. It is a *supervised learning* process, meaning that for every input pattern to be learned there is a predetermined output pattern. The training algorithm uses this *a priori* knowledge to guide the weight adjustment process within the network. The training process is as follows:

1. Apply an input pattern and calculate the output Y
2. a) If the output is correct (i.e. as expected) then go to step 1
   b) If the output is incorrect, and is zero, add each input to the corresponding weight
   (c) If the output is incorrect, and is one, subtract each input from the corresponding weight
3. Go to step 1

If the network can deduce the correct output of an input pattern then nothing is changed. If the output is incorrect then the weights are adjusted in such a manner as to reduce the error; weights are increased or decreased in an attempt to force the neuron to fire or not to fire given a particular input pattern.

In a finite number of steps the network will learn how to separate a number of different input patterns. The training process itself is global; the network learns all of the patterns simultaneously. This raises the question that there may be an optimum order in which to present the input patterns to the network training algorithm, so that the number of training iterations required in order to learn all patterns is reduced. Unfortunately, there is little theory to guide this determination.

3.2.1.3 Linear Separability

The major problem with perceptron-based networks, as was correctly deduced by Minsky and Papert in the 1960's [MiPa69], is that of linear inseparability. A perceptron network, like any other neural network architecture, can learn any set of input patterns so long as it has the capacity to represent such patterns internally. However, perceptron networks cannot represent all functions; in fact, as the number of inputs to the network increases the proportion of representable functions amongst the functions available decreases at an exponential rate.

The simplest problem that a perceptron simply cannot learn is the exclusive-or function. This function accepts two binary inputs x and y, outputting a logic-1 if and only if both inputs are different, otherwise the output is logic-0. Perceptron representation of a two-input function can be visualised by plotting all possible outputs on a graph, as shown in figure 3.4. The threshold function is, effectively, a line that bisects the graph; if the summation unit in the neuron produces a value on one side of the bisecting line then the neuron will fire, otherwise it will not. The setting of the threshold function is such that the bisecting line isolates the logic-1 outputs from the logic-0 outputs. It takes no more than a cursory glance at the graph in figure 3.4 to realise that there is no possible way in which a single bisecting line can isolate the exclusive-or logic-1 outputs (B_0 and B_1) from the logic-0 outputs (A_0 and A_1); no combination of w_1 or w_2 can produce such a line. A perceptron neural network cannot represent, and therefore cannot learn, the exclusive-or function.

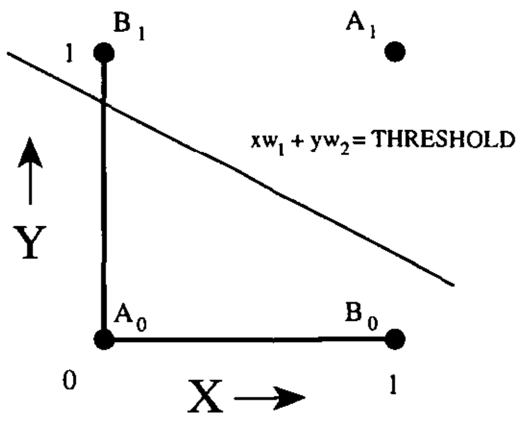


Figure 3.4 Exclusive-Or Threshold on the X-Y Plane

As the number of inputs to the network increases the situation gets worse. For a three-input function the separation is performed by a flat plane cutting through the resulting three-dimensional input space. For an n-input function, where $n>3$, visualisation breaks down and one must mentally generalize a space of n dimensions divided by some hyperplane. The actual number of representable functions is well known [Wind60] and shown in table 3.1.

Table 3.1
Linearly Separable Functions

n	2^2^n	Separable Functions
1	4	4
2	16	14
3	256	104
4	65,536	1,882
5	$4.3\text{x}10^9$	94,572
6	$1.8\text{x}10^{19}$	5,028,134

As table 3.1 shows the probability of any randomly selected function being linearly separable becomes vanishingly small with even a modest number of variables. For this reason single-layer perceptron networks are normally limited to simple problems. In order for more complex functions to be learned the network was required to consist of multiple layers, being a simple cascade of multiple single-layer networks. An associated training methodology was required that was able to cope with training neurons that did not receive inputs from the input training set but rather received the outputs of neurons in the previous layer. At the time of Minsky and Papert's work in the late 1960's such a training algorithm simply did not exist.

3.2.2 Backpropagation Learning

The invention of backpropagation a heralded the first theoretically sound algorithm for training multi-layered neural networks. It led to the resurgence of interest in the field after many years or near eclipse. Despite it having a number of limitations it dramatically increased the range of problems to which artificial neural networks could be applied.

The history of the invention of backpropagation is quite novel, having been 'discovered' on at least three separate occasions. Rummelhart et al believed that they had published the first clear and concise description of the algorithm [Rumm86]. It was soon after this that Parker was shown to have anticipated this work [Park82], whilst Werbos was found to have described the method earlier still [Werb74]. If Rummelhart et al and Parker had been aware of Werbos's much earlier work then they could have saved themselves a great deal of effort!

3.2.2.1 Neuron Configuration

The basic configuration for a neuron is similar to that for a perceptron network, and is shown in figure 3.5. A set of inputs x is applied to the neuron, either being external inputs or outputs from

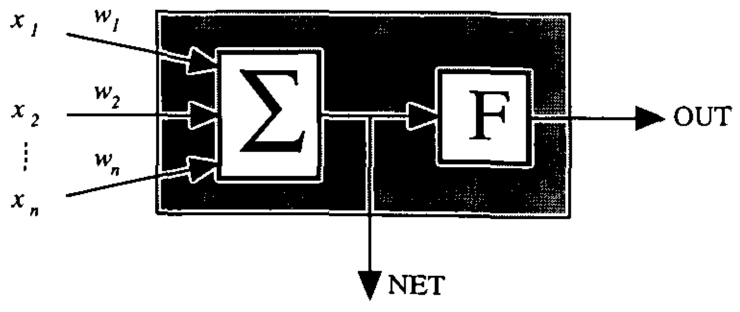


Figure 3.5 Backpropagation Neuron Schematic

neurons in previous layers of the network. Each of these is multiplied by a weight *w* and the results are summed to form the value *NET*; this value must be calculated for every neuron in the network. After *NET* is calculated a function *F* is applied to it, producing the value *OUT*.

The activation function *F* required for backpropagation has the requirement that it must be differentiable everywhere. The standard function used is

$$\begin{aligned} F(\mathrm{NET}) &= \mathrm{OUT} \\ &= 1/\left(1+e^{-x}\right) \end{aligned} \tag{3.2}$$

and is called a *sigmoid* function. This sigmoid compresses the range of *NET* so that *OUT* lies somewhere in the range of 0...1. It also has the very desirable feature in that it has a very simple derivative, which is used during the backpropagation algorithm

$$\frac{\delta \mathrm{OUT}}{\delta \mathrm{NET}} = \mathrm{OUT} \cdot (1 - \mathrm{OUT}) \tag{3.3}$$

This sigmoid also manages to introduce some form of automatic gain control. For small values of *NET* (i.e. *NET* near zero) the gradient of the sigmoid is steep, thus producing high gain. For larger values of *NET* the gain decreases, which allows large input signals to be accommodated in the network without causing saturations whilst allowing smaller signals to pass through with an excessive amount of attenuation. The activation function *F* is shown in figure 3.6.

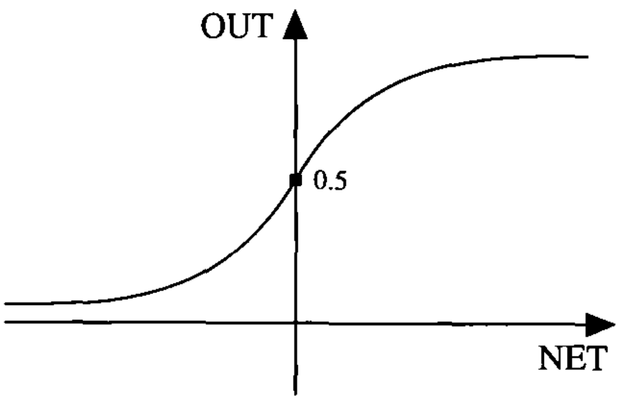


Figure 3.6 Backpropagation Activation Sigmoid Function

3.2.2.2 Learning Algorithm Overview

As in the perceptron network backpropagation networks are trained with sets of training pairs: each pair consists of an input vector and a desired output or target vector. A whole group of training pairs is called a training set. Before starting the training process all weights in the network are initialised to small random values, which prevents initially large weights from saturating the network. Also, if the weights are initialised to the same value (such as zero) and the network requires unequal weights in order to represent the training set then the network will not learn.

The basic steps required by the backpropagation learning algorithm are as follows:

1. Select the next training pair from training set and apply it to the network
2. Calculate the output of the network
3. Calculate the error between the network output and the desired target output associated with the training pair
4. Adjust the weights in the network in such a way that minimises the error
5. Repeat steps 1...4 for each training pair until the error for the entire training set reaches an acceptably low value

This is a supervised learning process. Steps 1 and 2 are the manner in which the network will ultimately be used; inputs are applied to the network and then the outputs are calculated. These steps can also be considered to be the *forward pass* of the training process, in that inputs are being passed through the network. In step 3 the error for the input data is calculated and, in step 4, this is passed back through the network in order to adjust the network weights. These two steps can be considered to be the *reverse pass* of the training process, as data is being passed from the output layer of the network back through the previous layers.

In the forward pass an input-target vector pair **X** and **T** come from the training set. An input vector **X** is applied to the network and an output vector **O** is produced. The calculation is done on a layer by layer basis beginning with the first layer of the network, which normally accepts inputs solely consisting of components of the input vector **X**. Each neuron has a *NET* value calculated for it and then the activation function then squashes this value to form *OUT*. Once all neurons in a layer have had this calculation performed the entire set of output values is used as inputs to the second and subsequent layers. This is repeated layer by layer until the final set of network outputs is produced.

This process can be stated very succinctly in vector form. The weights between neurons can be considered to be the matrix **W**, with the weight between neuron 8 in layer 2 to neuron 5 in layer 3 being designated $w_{82,53}$. The *NET* vector **N** may be expressed as the product of **X** and **W**, giving the expression **N** = **XW**. The activation function F is then applied to **N** in order to produce the output vector **O**. Therefore, for any given layer the following expression applies:

$$\mathbf{O} = F(\mathbf{XW}) \tag{3.4}$$

Simply, in order to calculate the output of the network equation 3.4 must be applied to each layer in the network, from input to output, with the output vector from one layer forming the input vector to the next.

It should be noted, however, that this applies only to fully-connected networks; it is perfectly feasible with the backpropagation learning algorithm to have input data applied to neurons in layers other than the first, and outputs from one layer can form inputs to neurons in more than one subsequent layer. Outputs in neurons in any layer can form components of the final network output vector. All descriptions of the backpropagation algorithm throughout this section make the assumption that the network is fully connected.

3.2.2.3 Layered Training Process

The reverse pass of the network can be split into two distinct sections; adjusting the weights in neurons in the output layer and adjusting the weights in the neurons in all other layers; non-output layers are referred to henceforth as *hidden* layers. Neurons in the output layer of the network have an associated target vector, so this layer is fairly straightforward to train. The hidden layers, however, require some form of target that is generated from the error values of neurons in layers closer to the output layer; these neurons are much harder to train.

Before a weight adjustment can be calculated for any neuron a pre-calculation is required that is different for the output and hidden layers; once this pre-calculation has been made then the remainder of the adjustment process, which is common to all layers, can then be applied.

(i) Output Layer Pre-Calculations

In order to adjust all the weights between neuron *p* in hidden layer *j* with neuron *q* in the output layer *k* a value known as the neuron δ value needs to be generated. This is done by subtracting the output for neuron *q* from the target output for the input pattern, which gives the error for neuron *q* for the current input value. The connections between neurons and weights for this operation can be seen in figure 3.7.

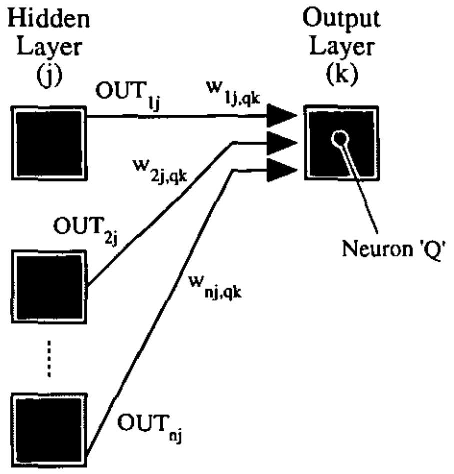


Figure 3.7 Connections for Output Layer Training

This error value is then multiplied by the derivative of the neuron activation function, thus giving the required δ value. This is used to adjust all weights for one particular neuron, in this case neuron *q* in output layer *k*. In summary, δ-generation for output layer neurons is as follows:

$$\delta_{qk} = OUT_{qk} \cdot (1 - OUT_{qk}) \cdot (Target_{qk} - OUT_{qk}) \tag{3.5}$$

(ii) Hidden Layer Pre-Calculations

It can be seen that δ-generation for non-output layer neurons cannot be done in this manner, as there is no available target value for the input pattern currently being learned; the target value is for the entire network and, thus, only applies to neurons in the output layer. Equation 3.5 needs modifying for neurons in hidden layers as follows:

$$\delta_{pj} = OUT_{pj} \cdot (1 - OUT_{pj}) \cdot \left(\sum_q \delta_{qk} \cdot w_{pj,qk} \right) \tag{3.6}$$

The δ-values from the output layers are used to generate the δ-values in the hidden layers. They act as an error indicator, attempting to get neurons in the hidden layers to reduce the error produced in the output layer neurons.

The weights connecting neurons in the hidden layer to those in the output layer can be seen to act in reverse at this point; during the forward pass they propagate the *OUT* signals from neurons in the preceding layer, scaling them as they go, whilst in the reverse pass they propagate δ-values from neurons in subsequent layers, also scaling them as they go. The neuron and weight connections for training neurons in the hidden layers can be seen in figure 3.8.

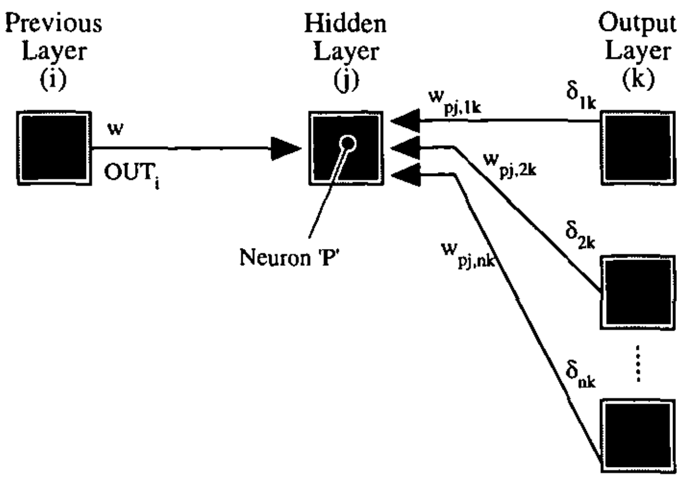


Figure 3.8 Connections for Hidden Layer Training

(iii) Weight Modification

Once a neuron has it's δ-value, regardless of what layer it is in, the training process is identical. Two further equations are required by the training process: one to calculate the weight adjustment for each connection in the neuron and one to actually modify the weight. These equations are as follows:

$$\Delta w_{pj,qk} = \eta \cdot \delta_{qk} \cdot OUT_{pj} \qquad (3.7)$$

$$w_{pj,qk}[n+1] = w_{pj,qk}[n] + \Delta w_{pj,qk} \qquad (3.8)$$

where

$w_{pj,qk}[n]$ = value of weight from neuron *p* in layer *j* to neuron *q* in layer *k* before any weight adjustment has taken place

$w_{pj,qk}[n+1]$ = value of weight from neuron *p* in layer *j* to neuron *q* in layer *k* after the weight adjustment has taken place

$\Delta w_{pj,qk}$ = value of the impending weight adjustment between the two neurons

δ_{qk} = the δ value for neuron *q* in layer *k*

η = training rate coefficient, typically in the range 0.01 to 1.00

OUT_{pj} = the value of OUT for neuron *p* in layer *j*

This process of applying either equation 3.5 or 3.6 to every neuron in the network, followed by equations 3.7 and 3.8 for every inter-neuron weight connection in the network, is carried out for every input-target pair in the training set until the errors are acceptably low. Note, the values for *OUT* in the hidden layer when the first network layer is being updated are, effectively, the input values from the training set.

3.2.2.5 Enhancements to the Learning Algorithm

There are many additions that can be made to the standard backpropagation algorithm in order to make it more efficient. A very simple improvement is the addition of a trainable neuron bias. This permits more rapid convergence by offsetting the activation function, giving a similar effect to adjusting the threshold in a perceptron neuron. It is achieved by connecting an additional trainable weight to each neuron that has a permanently wired logic-1 input. It is trained in an identical manner to other weights and can significantly reduce convergence times.

Many researchers [Rumm86] [SeRo87] have described methods of momentum, whereby previous weight changes affect future weight changes. They both add terms to the weight adjustment equations 3.7 and 3.8 that add a proportion of the previous weight adjustment to the current one. Although the methods differ they both help the algorithm to follow narrow gullies in the error space rather than crossing rapidly from side to side. These methods work well on some problems but, unfortunately, have little or even a negative effect on others.

Another powerful convergence speed-up [StHu87] takes into account the fact that the standard range of inputs 0...1 is not optimum. The weight adjustment $\Delta w_{pj,qk}$ is proportional to the output level of the source neuron output OUT_{pj}. In a binary system the OUT_{pj} value of zero results in no weight modifications, implying that half of the weights will not be modified! By changing the input range to ±1/2, and by adding a bias to the activation function of -1/2, convergence times can be reduced by between 30% and 50%.

Although a very powerful algorithm the backpropagation method of neural network training has it's share of problems. Close examination of the convergence proof by Rummelhart et al [Rumm86] shows that infinitesimally small weight adjustments are assumed, which implies that the training time is infinite. It is necessary to select a training rate step size η, but there is little theory to guide the network designer. A small training rate can result in convergence taking impractical lengths of time, whilst a large training rate may result in permanent instability in the network, with the network being unable to learn the problem being presented to it.

3.2.3 Kohonen Self-Organised Learning

3.2.3.1 Network and Neuron Configuration

The Kohonen unsupervised learning methodology is effectively a "winner takes all" network. For a given input vector one, and only one, neuron in the network will fire, with all other neurons remaining dormant. The layout for a layer of Kohonen neurons is shown in figure 3.9.

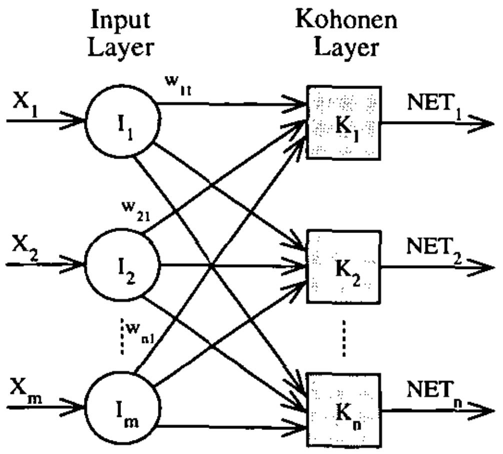


Figure 3.9 Kohonen Neural Network Layer

As in the backpropagation network there is a set of neurons associated with each neuron in the network. In figure 3.9 the neuron K_1 has weights w_{11}, w_{21}, ... , w_{m1}, which makes up the weight vector W_1. These weights connect the input data vector **X** to the neurons, where the *NET* output is based upon the weighted sum of the input vector. This may be expressed as follows:

$$NET_j = \sum_m (w_{mj} \cdot x_m) \tag{3.9}$$

Once the *NET* values for each neuron have been evaluated then the neuron with the highest value is declared the "winner"; it sets it's output value to logic-1 and all other neurons in the layer have their output value set to logic-0. It is, therefore, difficult to predict in advance which specific Kohonen neuron will fire for a particular input pattern. Indeed it is unnecessary to know this information in advance, as the main requirement of the training process is to separate dissimilar input vectors.

3.2.3.2 Input Vector and Weight Initialisation

It is beneficial to normalize all of the input vectors for a given training run. This is easily done by dividing each component of the input vector by the vector's length, which is the square root of the sum of the squares of all components. The vector length is given by:

$$x_i' = X_i / \sqrt{\sum_n x_n^2} \tag{3.10}$$

Equation 3.10 converts the input vector into a vector pointing in the same direction in n-dimensional space but of unit length. With vectors of 2-dimensions it can be seen that all normalised vectors terminate on a circle of radius one; this can be seen in figure 3.10. With vectors of 3-dimensions the vectors terminate on the inner surface of a sphere. This idea can be extended to an arbitrary number of dimensions, where the vectors terminate on the surface of an n-dimensional hyper-sphere.

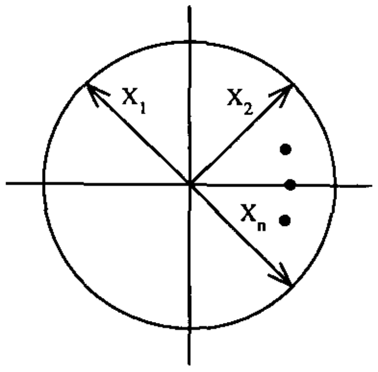


Figure 3.10 Unit Length Vectors

With the input vectors normalised the initially random weights within the network should also be normalised. The idea behind the training process, as is described later in this section, is to have weight vectors equal to normalised input vectors. If the weights are normalised before the training process begins then they start off closer to their desired values. If weight vectors are not normalised then some neurons in the network may never get the opportunity to fire, effectively wasting some capacity in the network. Also, the neurons that remain 'operational' may not have the capability to discern between the various categories of input vectors.

3.2.3.3 Learning Algorithm

During the training process an input vector is applied and its dot product with the weight vectors in each neuron are calculated and the neuron with the highest dot product being declared the winner and firing. The winning neuron is the one whose weight vector most matches the input vector. This neuron then has its weight vector slightly adjusted so that it is even more like the input vector. The change is proportional to the difference between the weight vector and the input vector.

The equation used during the training process to adjust the weights is as follows:

$$w_{new} = w_{old} + \alpha\left(x - w_{old}\right) \qquad (3.11)$$

where

w_{old} = value of the weight before adjustment

w_{new} = value of the weight after adjustment

α = training rate coefficient (may vary during training process)

The training rate coefficient α usually starts out at around 0.7 and may be gradually reduced during the training process.

In a geometric fashion firstly the vector $\mathbf{X}$ - $\mathbf{W}_{old}$ (a) is calculated by generating a vector from the end of $\mathbf{W}_{old}$ (c) to the end of $\mathbf{X}$ (d). This is then scaled by α, which is always less than one, producing a change vector δ (b). The new vector $\mathbf{W}_{new}$ (e) is then formed from the point of origin in the n-dimensional input space to the end of the change vector δ. This series of steps is shown in figure 3.11.

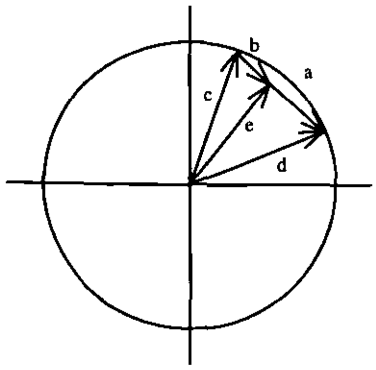


Figure 3.11
Kohonen Weight Changes

3.2.3.4 Enhancements to the Learning Algorithm

The simple act of randomising the weights within a Kohonen network can cause the training process some trouble. Once randomised the weights are uniformly distributed around the n-dimensional hyper-sphere. However, if the density of weight vectors is too low or too high for the given distribution of input vectors then either no neurons may be mapped to particular inputs or more than one may be.

The most desirable solution to this problem is to distribute the weight vectors in relation to the density of the input vectors. This has the effect of placing the correct number of weight vectors in the vicinity of the input vectors. Although this is impractical to implement directly there are several techniques available that approximate the effects.

A good method [Wass89] is the *convex combination method* which sets all weights in the network to the same value: $1/\sqrt{n}$, where *n* is the is the dimensionality of the hyper-sphere. This has the feature of normalising the weights to unit length. Each component of the input vectors is also scaled before input to the network in the following manner:

$$x_i' = \alpha \cdot x_i + \left\{ \left(1/\sqrt{n} \right) \cdot \left(1 - \alpha \right) \right\} \tag{3.12}$$

The scaling factor α is initially set to a small value, which causes all input vectors to have a length close to $1/\sqrt{n}$ and also coincident to the weight vectors. As the network is trained the scaling factor α is gradually increased to 1. This allows the input vectors to separate and resume their original values. The weight vectors, in turn, follow one or a small group of input vectors and, at the end of the training process, produces the desired pattern of outputs across the network. This method works well but is slow, as the weight vectors are being adjusted to a set of moving input vectors.

An interesting method [Desi88] is to add a conscience to each neuron. If a winning neuron has been winning more than its fair share of input vectors (approximately 1/m, where *m* is the number of neurons in the Kohonen network) then it temporarily raises a threshold value internal to the neuron, thus reducing its chance of winning. This then allows other neurons in the network to have the opportunity to train.

The problem of weight distribution within a Kohonen network is still a cause for concern, as it can seriously affect the accuracy of the resultant trained network. Unfortunately, the effectiveness of the various solutions is most certainly problem dependent and no hard and fast rule exists that works for all problems.

Another method of training proposed by Kohonen involves allowing more than one neuron to fire for a given input pattern. The normal training method, with only one neuron firing, is known as *accreditive mode*, whilst having more than one neuron firing is known as *interpolative mode*. The interpolative mode allows a group of neurons to fire. Their outputs are again normalised to unit length, which is done by dividing each *NET* value by the square root of the sum of the squares of all *NET* values; all neurons not in the firing group have their outputs set to zero. The benefits of this method is that in accreditive mode the accuracy of the network is limited in that the output of the network is a function of just a single neuron in the network. In interpolative mode, more complex mappings are possible, thus producing more accurate results, but no conclusive evidence on when to use this method yet exists.

3.3 Alternative Neural Network Learning Methodologies

3.3.1 Counter Propagation Learning

The counter propagation network was initially developed by Hecht-Nielsen [Hech87] in an attempt to provide solutions for neural network problems that could not tolerate the long training times that were required under the backpropagation network architecture. It has the ability to reduce training times by 99%,but the training algorithm is not as general as backpropagation and cannot be used on all problems.

3.3.1.1 Network and Neuron Configuration

It is a combination of two different network training architectures: the unsupervised self-organizing map of Kohonen, as described in section 3.1.2.3, and the Outstar network of Grossberg [Gros69]. Used together in this fashion they produce a training architecture that possesses properties that is not available in either one alone. The outputs of the Kohonen network layer are used as input to the Grossberg layer. The network is fully connected, in that the output of every Kohonen neuron is an input to every Grossberg neuron, but note that the sizes of the Kohonen and Grossberg layers do not have to be the same. The topology of the network is shown in figure 3.12.

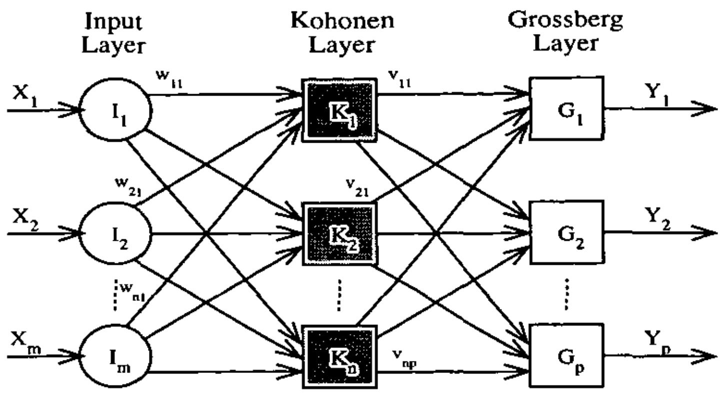


Figure 3.12 Counter Propagation Network

In operational mode the counter propagation network is very simple. An input vector **X** is presented to the Kohonen layer. Each neuron outputs a results vector **K**, whereby only one component of **K** contains non-zero data. This vector is presented to each of the neurons in the Grossberg layer, via the relevant weight vector **V** associated with the neuron, and each neuron forms its own *OUT* value as follows:

$$NET_p = \sum_n \left(x_n \cdot v_p\right) \tag{3.13}$$

or simply **Y=KV**. Note that as only one element of **K** is non-zero the calculation is actually very simple; the action taken by the Grossberg neurons is to simply pass the value of the weight connected to the non-zero element of **K** to its output.

3.3.1.2 Grossberg Layer Training

The Grossberg layer is equally simple to use during the training phase. An input vector is applied and the Grossberg neuron outputs are calculated. Weights are then only adjusted if they are connected to the non-zero Kohonen layer output. The adjustment is proportional to the difference between the Grossberg weight and the desired output of the Grossberg neuron to which it connects. This is as follows:

$$v_{np}^{new} = v_{np}^{old} + \beta \cdot k_n \cdot \left(y_p - v_{np}^{old}\right) \tag{3.14}$$

where

k_n = output of Kohonen neuron *n*

y_p = component *p* of the vector of desired outputs

β = training rate (initially 0.1 then gradually reduced during training)

v_{np} = weight between Kohonen and Grossberg layer neuron

It is clear from the training methods employed for both layers that the weights of the Grossberg layer will converge to the average values of the desired outputs whilst the weights of the Kohonen layer converge to the average values of the inputs. The Kohonen layer training is unsupervised whilst the Grossberg layer training is supervised. The effect is that the Kohonen layer produces outputs at indeterminate positions, which are then mapped by the Grossberg layer to the desired outputs.

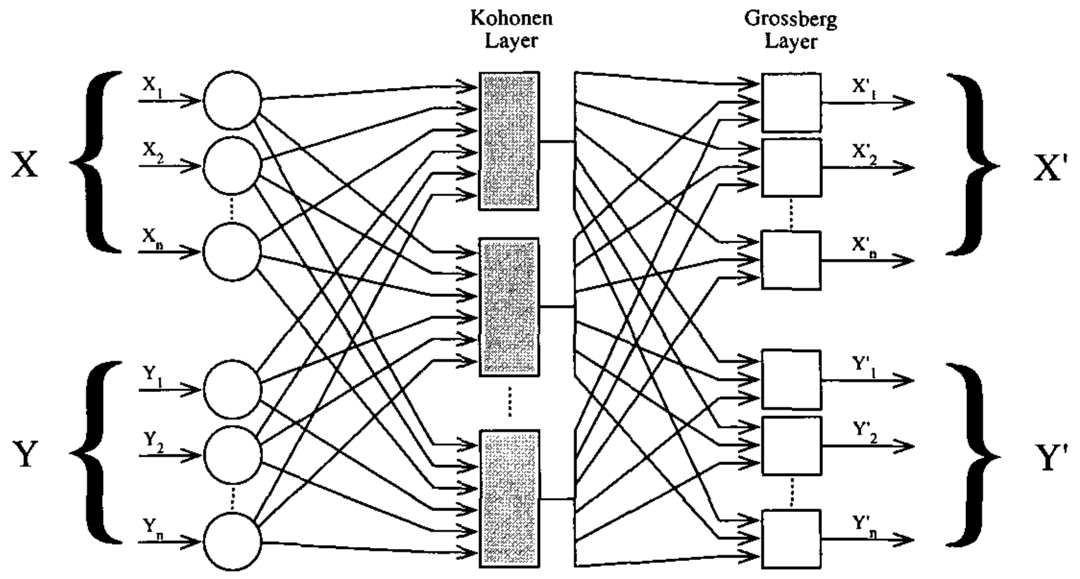


Figure 3.13 Full Counter Propagation Network

One of the most powerful aspects of the counter propagation network is that it can be used for vector mapping; it can generate functions of input data as well as generate inverse functions. Figure 3.13 shows a full counter propagation network, which has the input and output vectors split into two sections; **X** and **Y** as input vector tuples and **X'** and **Y'** as desired output vectors. Dividing the input and output vectors like this has no effect on the training algorithm, as they are indistinguishable from a single vector. Note that the desired outputs **X'** and **Y'** are identical to the input vectors - the network trains to recognise itself.

After successful training the network can perform identity mappings; applying an **X** and **Y** vector on the network inputs will result in the same values appearing at the outputs. Although not very useful in itself it becomes particularly interesting when one realises that by applying only the **X** vector to the network, leaving the **Y** vector as zero, the network will still produce the relevant **X'** and **Y'** output vectors. This is, effectively, a function mapping from **X** to **Y'**, with the network approximating the function. Additionally, if the inverse function exists then applying **Y** to the network, setting **X** to zero, will produce **X'**. This powerful mapping ability is the main strength in the counter propagation network.

3.3.2 Hopfield Learning

3.3.2.1 Recurrent Networks

One of the major features in the networks discussed up until this point is that they are all non-recurrent; there are no feedbacks from the outputs of the network back to the inputs. Without any feedback the networks are unconditionally stable in that they can never enter a mode in which the output wanders from state to state, never actually producing a usable output. However, this comes at a price; non-recurrent networks have a limited repertoire of behaviour when compared against recurrent networks.

Once an output has been routed back on to the network inputs a new output is calculated. and then fed back in again to modify the output. This process is repeated again and again and, in a stable network, successive outputs differ in value by less and less until the output remains constant.

It was the problem of stability that held back research, as it was not possible to predict which networks would become stable and which would remain, effectively, in a state of chaos. It wasn't until the work of Cohen and Grossberg [CoGr83] that a network theorem emerged that defined a subset of recurrent networks whose outputs eventually reached some stable state. Other important contributions to the field have been made by Hopfield, whose work was so influential that certain network configurations have become known as *Hopfield Networks.*

3.3.2.2 Network and Neuron Configuration

A Hopfield recurrent network has a layer of neurons, each of which accepts an input value from the training set. Each neuron also accepts the previous outputs of all neurons, scaled by some weighting factor. Each neuron goes on to produce a weighted sum of the recurrent inputs plus the input of the training set and then applies some activation function F in order to produce an *OUT* signal. This operation, save for the recurrent aspect, is similar to other networks discussed so far. Figure 3.14 shows a recurrent network consisting of two layers, with the dashed weight lines indicating weights fixed at zero.

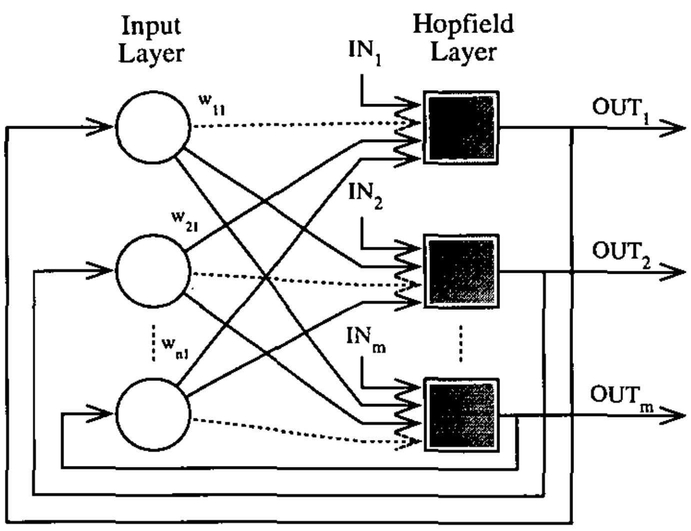


Figure 3.14 Single Layer Recurrent Network

In Hopfield's early work [Hopf82] the activation function F was a simple threshold function, such as that found in the perceptron networks. The *OUT* value of a neuron is set to one if the weighted sum of the *OUT* values of all other neurons, along with the current input value, is greater than a preset threshold value *T*; the *OUT* is set to zero otherwise. This is calculated as follows:

$$\mathrm{NET}_j = \mathrm{IN}_j + \sum_{i \neq j} (w_{ij} \cdot \mathrm{OUT}_i)$$

$$\mathrm{OUT}_j = \begin{cases} 1 \text{ if } \mathrm{NET}_j > T_j \\ 0 \text{ if } \mathrm{NET}_j < T_j \\ \text{unchanged if } \mathrm{NET}_j = T_j \end{cases} \qquad (3.15)$$

The operation of this network can be visualised quite easily. Figure 3.15a shows the case for a 2-neuron system, which gives rise to four system states (00, 01, 10, 11), each of which labels a vertex of a square. Figure 3.15b shows the case for a 4-neuron system, which gives rise to eight system states, each of which labels a vertex on a cube. This can be generalised to n-neurons, where an n-bit binary number labels a vertex on an n-dimensional hypercube. When a new input vector is applied, the network moves from state to state until it stabilises. The stable state is determined by the inputs, the weights and the threshold. If the input vector is only partially complete then the network will stabilise on the state closest to the desired state.

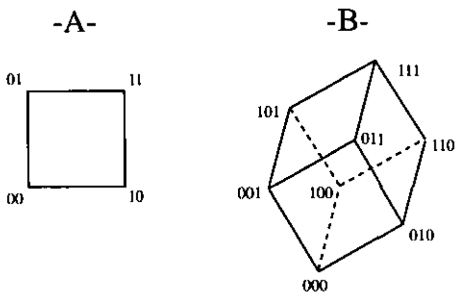


Figure 3.15 Hopfield Network State Space

3.3.2.3 Learning Algorithm

The method of finding a stable state can be expressed mathematically. If a function could be found that always decreases whenever the network changes state then, eventually, such a function must reach a minimum value and then remain constant; at this point the network would be classed as stable. Such a function is called a *Liapunov* function and works as follows:

$$E = -\tfrac{1}{2}\sum_i \sum_j (w_{ij} \cdot \mathrm{OUT}_i \cdot \mathrm{OUT}_j) - \sum_j (I_j \cdot \mathrm{OUT}_j) - \sum_j (T_j \cdot \mathrm{OUT}_j) \quad \forall\, i \neq j \qquad (3.16)$$

where

E = artificial network energy value

w_{ij} = weight between output of neuron *i* and the input to neuron *j*

OUT_j = output of neuron *j*

I_j = external input to neuron *j*

T_j = threshold of neuron *j*

It has been shown [CoGr83] that recurrent networks are stable if the weight matrix **W** between layers in the network is symmetrical with zeros on the leading diagonal, hence the $\forall i \neq j$ term in equation 3.16. The change in energy *E* due to the change in the state of neuron *j* is:

$$\begin{aligned} \delta E &= -\delta OUT_j \cdot \left[\sum_{i \neq j} (w_{ij} \cdot OUT_i) + I_j - T_j \right] \\ &= -\delta OUT_j \cdot [NET_j - T_j] \end{aligned} \tag{3.17}$$

There are three different possibilities regarding the change in the state of a neuron:

(i) $NET_j > T_j$ This causes the term in brackets to be positive, implying that the output of $neuron_j$ must either change in a positive direction or remain constant, as given in equation 3.15. This means that δOUT_j can only be positive or zero and that δE must be negative. Hence, network energy must remain constant or decrease

(ii) $NET_j < T_j$ In this case δOUT_j can only be negative or zero, implying again that δE must be negative. Hence, network energy must remain constant or decrease

(iii) $NET_j = T_j$ In this case δOUT_j can only be zero implying again that δE must also be zero. Hence, network energy remains constant

It is because the energy function shows this continuous downward trend that eventually the network must find a minimum value and stop. This type of network is, by definition, stable.

3.3.2.4 Continuous Systems and Associative Memories

Hopfield continued this work into continuous systems [Hopf84]. A common shape for the activation function *F* is an s-shaped sigmoid function, similar to that used in the backpropagation network learning algorithm. The function used by Hopfield is

$$F(x)=1/\left(1+e^{-\lambda\cdot NET}\right) \tag{3.18}$$

where λ is a coefficient that determines the steepness of the sigmoid function. As in the binary system stability is ensured if the weight matrix is symmetrical. The energy function is conceptually similar to the discrete case, so is not discussed in any more detail here. However, if the value of λ is large then a continuous system operates much like a discrete binary system, with the network stabilising near a vertex of the hypercube, with outputs being close to either zero or one. As λ is reduced the stable points move away from the vertices, disappearing one by one as λ approaches zero.

It is possible to make the continuous model of the Hopfield network to act as an associative memory in a similar fashion to the Kohonen networks. In order to do this on a recurrent network the weights must be selected to produce energy minima as the correct vertices of the hypercube. The outputs in this model lie in the range -1...+1, which correspond to the binary values 1 and 0 respectively. The memories themselves are encoded as binary vectors and stored in the weights according to the following equation:

$$w_{ij}=\sum_{d=1}^{m}\left(OUT_{id}\cdot OUT_{jd}\right) \tag{3.19}$$

where

m = number of desired memories (output vector set size)

d = index number of a desired memory (output vector)

OUT_{id} = component *i* of output vector *d*

In order to perform as an associative memory the outputs of the network have to be forced to the values of the input vector, even if the input vector is only partially complete. The input vector is them removed from the network inputs, which allows the system to "relax" and find it's nearest deep minimum value. However, as the network follows the local slope of the energy function it is possible for the network to become stuck in a localised minimum point and not find the best overall solution.

3.3.2.5 Hopfield Networks and the Boltzmann Machine

Hopfield networks tend to stabilize to local minimum of the energy function rather than to a global minimum, as outlined in the previous section. It is possible to solve this problem through the use of a class of networks know as *Boltzmann machines*. These networks have neurons that change in a statistical rather than a deterministic fashion, and there is a close analogy between these methods and in the way in which metal in annealed; these methods are often known referred to as simulated annealing [Kirk83].

A metal is annealed by heating it to a high temperature, above it's melting point, and then letting it cool gradually. The laws of thermodynamics state that at such high temperatures the atoms within the metal possess very high energies and can move about freely, randomly assuming every possible configuration. As the temperature is reduced, however, these energies decrease until the system has settled into some minimum energy configuration; the system energy global minimum is reached once the cooling process is complete.

The Boltzmann probability factor determines the probability distribution of system energies at a given temperature. This distribution is defined approximately as follows:

$$P(E) \propto \exp\left(\frac{-E}{k \cdot T}\right) \qquad (3.20)$$

where

E = system energy

P(E) = probability of system being in a state with energy *E*

k = Boltzmann's constant

T = temperature (degrees Kelvin)

The state-change rules for the continuous network must be changed so that they act statistically rather than deterministically. This is done by using the amount by which a neuron *NET* output exceeds it's threshold value *T* as the probability of a weight change occurring, as defined by

$$\begin{aligned} E_i &= NET_i - \theta_i \\ P_i &= 1 / \left(1 + \exp\left(\frac{-\delta \cdot E_i}{T}\right)\right) \end{aligned} \tag{3.21}$$

where

E_i = the network energy

NET_i = the NET output of neuron *i*

ϕ_i = the threshold of neuron *i*

T = the artificial temperature of the system

During operation the artificial temperature *T* is set to a high value and the neurons are clamped to some state as determined by an input vector. The network then attempts to seek a minimum energy state using the following two steps:

(i) For each neuron with a probability equal to P_i, set state to one, else set state to zero

(ii) Gradually reduce artificial temperature T and then repeat step (i) until a minimum energy state is reached

3.3.3 Adaptive Resonance Theory

In a real world environment the human brain is exposed to a constantly changing world; it may never be presented with an identical set of sensory inputs twice in a lifetime. Under such circumstances networks which have a fixed training set, such as backpropagation, simply would not be able to cope; it would often learn nothing in such a situation, constantly modifying it's weights to no avail.

The adaptive resonance theory, or *ART*, is the result of research into the problem of temporal instability [CaGr87] [Gros87]. ART networks maintain the idea of plasticity, in that they have the capacity to learn new patterns whilst preventing the modification of patterns that have already been learned. Although the theory can be difficult to understand the ART networks have generated a great deal of interest in networks of this type. Details of the complex mathematics behind the ART networks is beyond the scope of this discussion, so this section will concentrate solely on the fundamental ideas of the network, along with descriptions of the three major phases of the network: recognition, comparison and search.

3.3.3.1 The Architecture of ART

There are two forms of the ART paradigm; ART1 accepts only binary input data whilst ART2 is more general in that it can accept both binary and continuous inputs [CaGr87b]. For the sake of brevity only ART1 is discussed in this section, being referred to as simply *ART* for the duration of this section.

The ART network is, basically, a vector classifier that accepts an input vector and then classifies it into one of a number of categories, depending upon which of the patterns stored within the network that it most closely resembles. A new category is created if the input pattern does not resemble any of the stored patterns, whilst if a match is made within a specified vigilance tolerance then the stored pattern is modified (or trained) so that it is more like the input pattern.

Under no circumstances are any stored patterns ever modified if they do not match the input pattern within the vigilance tolerance; the only way in which a stored pattern can be changed is if the input pattern resembles it closely.

A simplified version of the architecture is shown in figure 3.16. The network consists of two layers of neurons, denoted *comparison* and *recognition* respectively. The other units provide some control functions that are required for training and classification within the network.

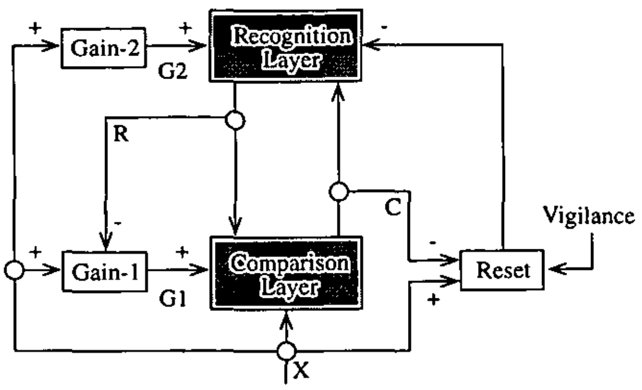


Figure 3.16 Simplified ART Architecture

The comparison layer receives a binary input vector **X** and initially passes a copy through to the recognition layer as **C**. The vector **R** is received from the recognition layer, which modifies the vector **C**. Each neuron in this layer receives three inputs: a component x_i from **X**, a feedback signal P_j (weighted sum of all components of **R**) and the input from the Gain-1 unit *G1*. Each neuron also has a binary weight vector **T** which connects the feedback vector **R** to the neuron (T $\Rightarrow$ top-down). In order for a neuron to output a one then at least two of the inputs must also be one, otherwise the neuron will output a zero. The gain signal *G1* is initially set to one and **R** is set to zero, meaning that in the first step **X** is effectively copied onto the output vector **C**. However, **C** changes over time as the feedback vector **R** from the recognition layer takes effect.

The task of the recognition layer is to classify the input vector. Each of the neurons in the layer has a weight vector **B** (B $\Rightarrow$ bottom-up) and accepts the input vector **C**. Only the neuron whose weight vector best matches the input vector will output a one, with all other neurons in the layer outputting a zero. The weight vector **B** essentially constitutes a stored pattern; this is an idealised pattern, which represents a category, but the components are continuous numbers and not binary. Each neuron carries out a dot product between **B** and **C**, with the neuron having the largest result being declared the winner. The recognition layer also operates a lateral inhibition network, ensuring that only one neuron in the recognition layer fires at any one time. It operates on a winner-takes-all basis, with outputs of all neurons being connected to themselves (via a positive weight) and to all other neurons in the layer (via a negative weight). This enables neurons with large outputs to inhibit the output of other neurons in the layer.

Gain-1 is used just to ensure that the comparison layer initialises correctly by passing on a copy of **X** to the recognition layer as **C**. *G1* is set to one if any component of **X** is one and all components of **R** are zero, otherwise *G1* is set to zero. The logic for this is shown in table 3.2.

Table 3.2 G1 Calculation

Or of **R**	Or of **X**	G1
0	0	0
0	1	1
1	0	0
1	1	0

Gain-2 is used to enable or disable the neurons in the recognition layer; if it is zero then the layer is effectively disabled. The actual value of *G2* is one if and only if any component of input vector **X** is one; i.e. it is the logical-or of all members of **X**.

The reset unit takes in **X** and **C** as inputs and measures the similarity between them. If they differ by more than the *vigilance* parameter then a reset signal is sent to disable the firing neuron in the recognition layer. It works by calculating the ratio of the number of ones in **C** to the number of ones in **X**; if this ratio is below the level of the vigilance parameter then the reset signal is sent to the recognition layer.

3.3.3.2 Recognition Phase

At the beginning of the recognition phase no input vector is applied; **X** is zero across all components. *G2* is set to zero as a result, disabling all neurons in the recognition layer and causing the initial value of **R** to be set to zero. This primes all neurons in the recognition layer, giving all an equal chance of success once the input vector is applied to the network.

The input vector **X** is then applied, which must have at least one non-zero component in order to set *G1* and *G2* to one. This primes all neurons in the comparison layer, forcing the vector **C** to initially be an exact duplicate of the input vector **X**.

A dot product is them formed for all neurons in the recognition layer between **C** and the weight vector **B** associated with the neuron. The neuron with the largest dot product, which is determined through the lateral inhibition network within the recognition layer, is the one whose weight vector **B** best matches **C**. This neuron outputs a one, whilst all other neurons in the recognition layer output a zero. The outputs from all neurons in the layer form the vector **R**, which has only a single non-zero component corresponding to the winning neuron.

3.3.3.3 Comparison Phase

The comparison phase starts once the recognition layer passes it an **R** with a single non-zero component. As **R** is input to each neuron in the comparison layer it can be seen that component $\mathbf{R}_i$ is fanned out to every neuron in the layer. Therefore, neuron *j* in the recognition layer receives an input $p_j = t_{ij} \cdot \mathbf{R}_i$, which is equal to either zero or t_{ij} (which itself can only have the values zero or one). This vector **P** is a representation of the closest stored pattern in the recognition layer.

As **R** is no longer all-zero *G1* is inhibited and set to zero. Thus, only neurons in the comparison layer that receive simultaneous ones from both **X** and p_j will fire. The feedback from the recognition layer to the comparison layer forces components of **C** to zero where the original input **X** does not match the stored pattern; i.e. where **X** and **P** do not match.

In the case where there are few matches between **X** and **P** few neurons in the comparison layer will fire, resulting in **C** having many non-zero constituents. This implies that the 'matched' pattern being fed to the comparison layer by **P** is not correct and that the neuron in the recognition layer which holds this matched pattern should be inhibited. This process is carried out by the reset block, which compares **X** and **C** to check for a degree of similarity within a specified tolerance level. This action will force the output of the winning neuron in the recognition layer to zero, disabling that neuron for the remainder of the current classification.

3.3.3.4 Search Phase and Performance Issues

If the reset signal does not activate then the network has found a match and the classification process is complete. If the reset signal activates then other stored patterns within the network need to be searched in order to find a better match. The act of inhibiting the firing neuron in the recognition layer causes all components of **R** to be reset to zero, *G1* goes to one and the input pattern **X** is again effectively copied to **C**. Also, during the recognition phase, a different neuron in the recognition layer will win and be able to fire, causing a different pattern **P** to be presented to neurons in the comparison layer. This process repeats until one of the following two events occur:

(i) a stored pattern is found that matches **X** within the limits specified by the vigilance parameter. A training cycle is then entered that modifies the relevant weight vectors **T** and **B** associated with the firing neuron in the recognition layer

(ii) all patterns stored in the recognition layer have been tried and none match the input vector **X**. At this stage all neurons in the recognition layer have been inhibited. In this case a previously unused neuron in the recognition layer is assigned the pattern, with associated weight vectors **B** and **T** set up accordingly

The network effectively performs a sequential search through all of the stored patterns in the network. This can be a very time consuming process on a normal serial digital computer, so simulations can take a large amount of time. However, on an analogue computer, or on a parallel digital computer, all of the dot-products calculated in the recognition phase can be performed simultaneously, resulting in a very rapid search.

Another drawback with implementation on a serial digital computer is with the lateral inhibition network in the recognition layer. During the time when a neuron wins the battle for the right to fire all neurons are involved in simultaneous calculations, requiring global broadcast communications with all other neurons in the layer. Again, implementation on a parallel digital computer or, even better, on an analog computer, substantially reduces the time required.

4

IMPLEMENTATION OF NEURAL NETWORKS

INTRODUCTION

This chapter describes some proposals for the VLSI implementation of neural networks, covering both digital and analogue logic designs. Each section describes the chip architecture and the implementation of neural networks on the VLSI system. The chapter continues with details on some recent practical examples of neural network applications, with the discussion being based on the application rather than on the network.

4.1 VLSI Neural Network Systems

4.1.1 Backpropagation in Linear Arrays

4.1.1.1 Background Information

This research [Nayl94] looked into the feasibility of implementing neural networks in hardware for the purpose of image processing. It is based upon an existing linear systolic array chip design known as *HANNIBAL* [Myer91], developed in cooperation by British Telecom and Loughborough University.

Rather than modify the existing HANNIBAL design the work concentrated on mapping the backpropagation learning algorithm on to the chip. Although the chip itself had already been fabricated the work simulated an array of HANNIBAL devices on a SUN Sparc workstation, rather than being based upon designing a circuit board containing an array of such devices.

4.1.1.2 Chip Architecture and Neural Mapping

The HANNIBAL architecture is a linear systolic array, consisting of four PE's per device, that theoretically may be cascaded to any length. Each PE within the device has 256 words of 16-bit local memory, as well as a 32-bit fixed-point multiplier and adder.

The PE has the capability of carrying out a standard multiply-and-accumulate (MAC) operation in a single clock cycle. The internal databus is 16-bit wide, and may be operated in a feed forward or feed back mode; this reversal of the data pipe proves to be useful during the back propagation of errors through the network. The neurons in the hidden and output layers of a network are mapped on to the PE's in HANNIBAL on a 1:1 basis.

The data pipe is divided into two separate 8-bit streams, with the upper stream holding the input data to the neurons the lower stream holding the outputs for the neurons; this scheme shown in figure 4.1. In this figure a network with an output layer of two neurons and a hidden layer of three neurons can be mapped directly onto the device, with neurons 1-3 being mapped on to hidden layer PE's 1-3, and neurons 4-5 being mapped on to output layer PE's 1-2.

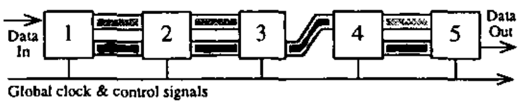


Figure 4.1 Feed Forward Mode

Input data is piped directly into the input layer PE's, with subsequent results being held in the PE's until processing in the layer has finished. Once all results are ready they are sent out of the PE's and routed back into the PE's used to represent the output layer; this method prevents output data from the hidden layer PE's being overwritten by results from the output layer PE's.

The feed back mode is shown in figure 4.2. The data pipe is configured as a single 16-bit stream, passing data to all PE's in the linear array. This mode of operation is only required when the network is in training mode and the errors produced during training are propagated back through the network.

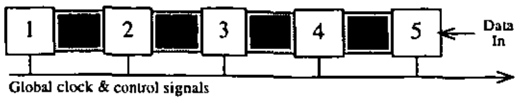


Figure 4.2 Feed Back Mode

4.1.1.3 Implementation of Backpropagation

The neural network implementation on the HANNIBAL architecture requires the device to be in one of several *modes*. A controller within each PE is provided with control information that enables it to execute the algorithm almost autonomously, with an external array controller handling the infrequent mode changes and all data I/O synchronisation and storage.

Stage #1 : Network Recall

Network recall utilises the feed forward mode of the data pipe. The activation value of any neuron in the network is given by a non-linear function of the sum of its weighted synaptic inputs, as is standard in the backpropagation algorithm.

The array controller sets the linear array into an *activation* mode. The process-time graph in figure 4.3 shows, cycle by cycle, the processes that are occurring within each PE in the network for a single 2-element input vector. Waiting cycles are shown as blank spaces, with layer indices on inputs only being shown at the top and bottom of the figure and not on each parameter.

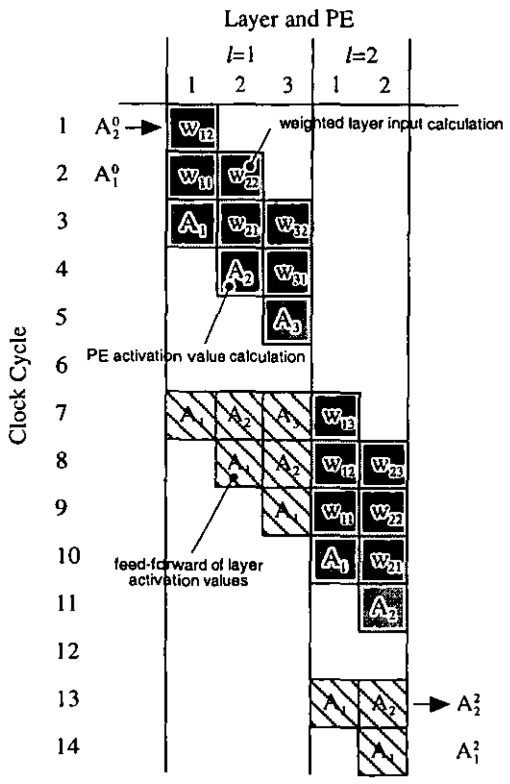

Layer and PE
l=1
l=2
1 2 3 1 2
Clock Cycle
1 2 3 4 5 6 7 8 9 10 11 12 13 14
A_2^0
A_1^0
weighted layer input calculation
PE activation value calculation
feed-forward of layer activation values
A_2^2
A_1^2

Figure 4.3 Recall Timing Graph

In operation PE 1 receives the first element of the input vector on the first clock cycle, performs a MAC operation and passes the input vector element to PE 2. This is repeated for all PE's in the layer for all elements of the input vector. Once a PE has completed the calculation of it's activation value it waits for all other PE's in the layer to complete. After another cycle of delay, all activation values are transferred onto the lower 8-bits of the data pipe. Data is then redirected from the output of PE 3 in the hidden layer to the input of PE 1 in the output layer, and the above operation is repeated so that the activation values for PE's in the output layer can be calculated.

Stage #2 : Network Learning

The neural learning process on the HANNIBAL device consists of three distinct stages: forward propagation of the activation values, error backpropagation and modification of the weight values. The first step is identical to the recall stage previously described, except that the activation values for the output layer neurons are not clocked out of the device; they are used internally during the second stage of error backpropagation.

The second phase of error backpropagation requires the device to be put into *calculation* mode by the array controller. The error values for each neuron for the given input pattern are calculated, with a different algorithm being used for output layer and hidden layer neurons. The equations used are equivalent to equations 3.5 and 3.6 given previously in section 3.2.2.3 and are defined as follows for the output layer and hidden layer respectively:

$$\delta_j^l = A_j^l \cdot \left(1 - A_j^l\right) \cdot \left(E_j - A_j^l\right) \tag{4.1}$$

$$\delta_j^l = A_j^l \cdot \left(1 - A_j^l\right) \cdot \sum_{i=0}^{I-1} \left(\delta_i^{l+1} \cdot W_{ij}^{l+1}\right) \tag{4.2}$$

where

$E =$ expected output value

$i =$ index of PE in layer above current layer

$I =$ number of PE's in layer above current layer

The process-time graph for these calculations is shown in figure 4.4. Whereas the calculation in equation 4.1 is fairly simple for the output layer the calculation for the hidden layers, given in equation 4.2, is a more complex vector-matrix operation.

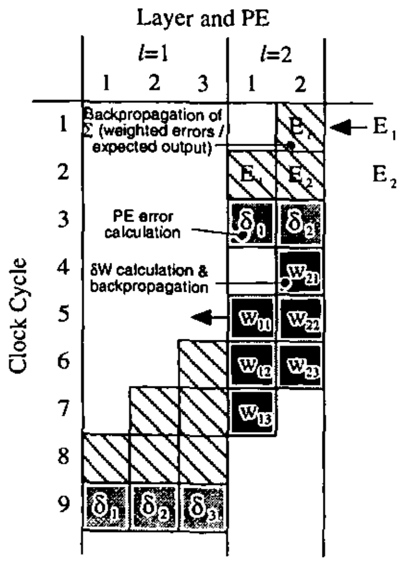

Layer and PE
l=1
l=2
1 2 3 1 2
Clock Cycle
1 2 3 4 5 6 7 8 9
Backpropagation of Σ (weighted errors / expected output)
PE error calculation
δW calculation & backpropagation
E_1
E_2
δ_0 δ_2
W_{21} W_{11} W_{22} W_{12} W_{23} W_{13}
δ_1 δ_2 δ_3

Figure 4.4 Calculation Timing Graph

Before the operation commences the value $A_j^t.(1-A_j^t)$ is precalulated by each PE, which is not shown in figure 4.4. The components of the expected output vector E are clocked into the appropriate PE's in the output layer. Error values δ_1^2 and δ_2^2 are then calculated and stored within the PE. These error values are then passed back to PE's in the hidden layer in order to calculate the error values for the hidden layer, as shown in equation 4.2. Note that this graph only shows the process for a single hidden layer; if additional hidden layers existed then the algorithm shown in equation 4.2 would be repeated until the error values for all neurons in all hidden layers had been calculated.

The third phase of the network learning stage requires the HANNIBAL device to be put into *update* mode by the array controller. In this phase the error values are used to calculate the weight adjustments that have to be made to all weights in the network, as well as to update the trainable neuron bias values present in each neuron. The calculation to be performed is equivalent to equations 3.7 and 3.8 and is as follows:

$$W_{jk}^l(n+1) = W_{jk}^l(n) + \eta \cdot \delta_j^l \cdot A_k^{l-1} \quad (4.3)$$

Note, for the trainable bias value the neuron input A_k^{l-1} is always hard-wired to +1. The process-time graph for this phase is shown in figure 4.5. The process is based upon a two cycle read/modify operation; this begins with the neuron biases, which are the simplest values to modify, as they do not require any input vectors or activation values for the calculation.

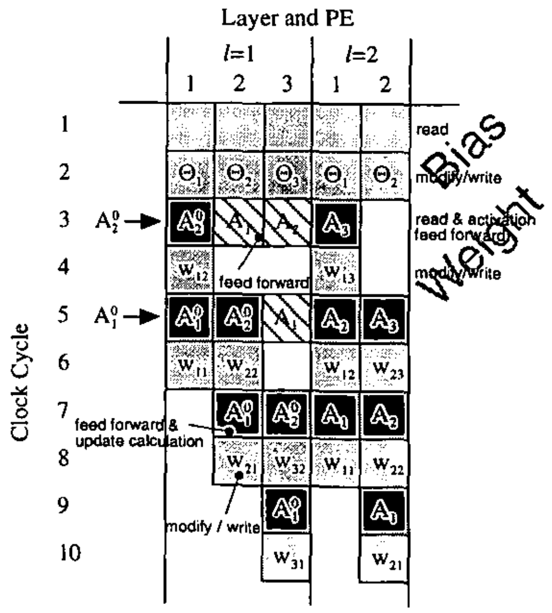


Figure 4.5 Update Timing Graph

Equation 4.3 shows that for the weight adjustments the original input vectors need to be presented to the linear array again. The input vectors are presented to the input layer PE's, where the weight are adjusted, and then the activation values and input vectors are fed forward towards the output layer. The pipeline structure allows these transfers to be made in parallel. The two-cycle requirements of this operation is because the PE requires one clock cycle to read the data values and calculate the weight adjustment, with a second cycle being required to write the modified weight value back to local memory.

4.1.2 Real-Time Clustering Neural Engine

4.1.2.1 Background Information

The project to implement a clustering device in hardware [SeLi96] was based upon a desire to cluster in real-time, being able to cluster together patterns at their speed of arrival on the microchip. It was also to overcome the problem that is associated with many existing clustering algorithms [Pao89] [Rodr93] in that they often need to be trained off-line in order to build up the categories.

The work also attempts to modify the clustering algorithm, based upon the ART-1 model [CaGr87a], in order to make it more VLSI-friendly; this should produce a more efficient hardware design for the final circuit. In many places the algorithm was simplified, at a cost of possible performance degradation, for the sole reason of making it possible to implement the algorithm in VLSI form. These simplifications were always hardware-oriented.

4.1.2.2 VLSI-Friendly ART-1 Algorithm

In their original work on ART, Carpenter and Grossberg introduced two different types of architectures for the neural memory using time domain non-linear differential equations: short term memory (STM) and long-term memory (LTM). This gave rise to three different levels of possible ART-1 implementation:

(i) *Full Model Implementation*, with both STM and LTM equation realised

(ii) *STM Steady State Implementation*, with LTM equations realised and STM governed by some non-linear algebraic equations, requiring STM events to be artificially sequenced

(iii) *Fast Learning Implementation*, where both LTM and STM events are artificially sequenced

The computational overheads decrease with each model and hence it is the preferred choice for implementation in VLSI form.

From a hardware point of view [1] one of the first issues that comes into consideration is that there are two sets of weights within the network: recognition (bottom up) layer weights **B**, which are continuous, and comparison (top down) layer weights **T**, which are binary. The physical implementation of **B** presents the first difficulty, as these weights require either an analog or digital memory of sufficient precision such that the digital discretisation of the weights does not affect the system. The original work on ART stated that **T** effectively stored copies of binary input patterns, with **B** being a scaled continuous representation of the same data; therefore, **B** is a normalized version of **T**. Also, bottom-up weights **B** may take any real value in the range 0...K, where$K=\frac{L}{L-1+n}$ for L>1 [CaGr87a].

It would be physically desirable to implement only the binary-valued weight set **T**, then let the hardware do the normalization of the bottom-up weights **B** during the computation of **C**. In this way the two sets of weights can be replaced by a single binary valued set **Z**, with the calculation of **C** modified to take into account the normalization effect of the original bottom-up weights. The calculation for **C**, after modification by the bottom-up weights **B**, can be translated to the following:

$$\mathbf{C}_j=\frac{L\cdot\mathbf{C}_{Aj}}{L-1+\mathbf{C}_{Bj}}\quad\left(\equiv\frac{L\cdot\sum_{i=1}^{N}(z_{ij}\cdot X_i)}{L-1+\sum_{i=1}^{N}z_{ij}}\right) \tag{4.4}$$

This rather minor implementation modification results in a much more VLSI friendly algorithm, although it still requires a division operation for each node in the recognition layer. It would be very desirable to remove this operation completely. In previous work [SeLi95] the authors

[1] The notation used in this section is taken from section 3.3.3.1 and not from [SeLi96]

showed that this division operation can be replaced by a subtraction operation, whilst preserving all of the computational properties of the ART-1 algorithm; although the observed behaviour is different for some sequences of patterns with respect to ART-1 the overall clustering behaviour is still equivalent. This further modification is as follows:

$$\mathbf{C}_j = L_A \cdot \mathbf{C}_{Aj} - L_B \cdot \mathbf{C}_{Bj} + L_M \tag{4.5}$$

where L_A and L_B are positive parameters that play the roles of L in equation 4.4, save that $L_A > L_B$. The constant parameter $L_M > 0$ is required to ensure that $\mathbf{C}_j \geq 0$ for all possible values of T_{Aj} and T_{Bj}. This additional hardware simplification is very important, as it provides the potential for a significant performance improvement, as well as making the algorithm even more VLSI friendly.

4.1.2.3 VLSI Implementation

The device is designed to communicate with the outside world via digital I/O structures, allowing it to act with an asynchronous digital nature. However, the internal circuitry consists of current-mode analog microelectronics.

There are five different operations that the analog circuitry has to perform:

(i) Generation of the terms $\mathbf{C}_j$: as the terms z_{ij} and $\mathbf{I}_j$ are binary then binary multiplication, addition and subtraction is required

(ii) A winner-take-all (WTA) operation to select the maximum $\mathbf{C}_j$

(iii) Comparison of the vigilance factor with the winning element of $\mathbf{C}$

(iv) De-selection of the terms of the winning element $\mathbf{C}$ if it lies outside the vigilance tolerance

(v) Update of the weights

The first three operations require a certain amount of precision, whilst the last two operations are not precision dependent.

A possible hardware schematic diagram for a circuit that could obtain precision between 1-2% for 100-pixel binary input patterns is shown in figure 4.6. This shows an 18x100 array of synapses ($S_{1,1} \ldots S_{18,100}$), a 1x100 array of controlled current sources ($CC_1 \ldots CC_{100}$), two 18-element vectors of current mirrors ($CMA_1 \ldots CMA_{18}$ and $CMB_1 \ldots CMB_{18}$), an 18-element vector of current comparators ($COM_1 \ldots COM_{18}$), an 18-input WTA device, two 18-output current mirrors (CMM and CMC) and an adjustable current mirror (ρ). The registers $R_1 \ldots R_{18}$ and the final NOR output are optional units. Note that the output from the WTA network, $\overline{y_1} \ldots \overline{y_{18}}$ are also used as inputs to the array of synapses, but has been omitted from the figure. Other omitted signals are RESET and $\overline{\text{LEARN}}$ (inputs to the synapse array) and ER (input to WTA circuit).

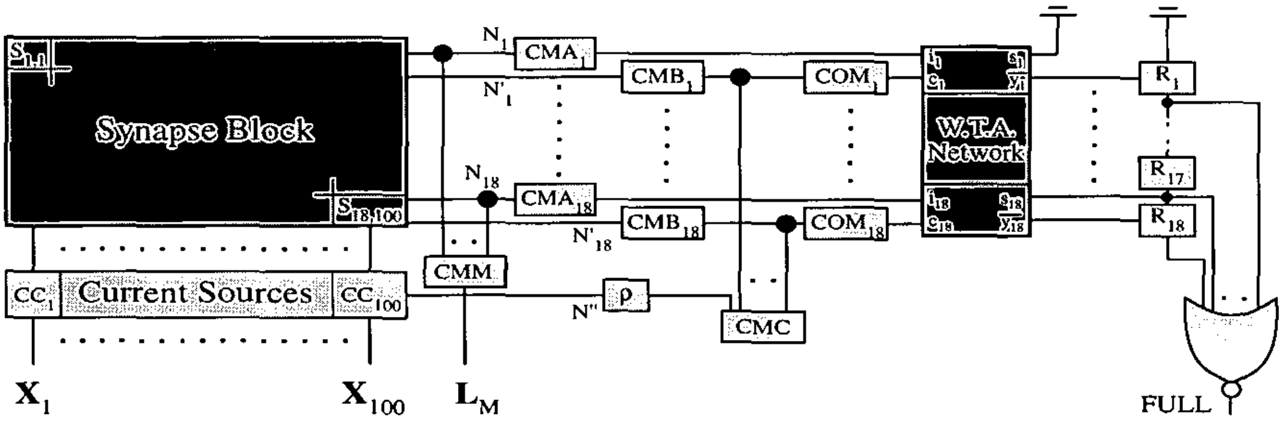


Figure 4.6 Hardware Block Diagram for VLSI-Friendly ART-1 Algorithm

Further breakdown of the circuit units is not necessary, but full details are given in the original work [SeLi96].

4.1.2.4 Implementation of ART-1

Each synapse in the array receives two input signals, $\overline{y_i}$ and X_j, two global control signals, RESET and $\overline{\text{LEARN}}$, holds the value z_{ij} and generates two output currents:

$$\begin{aligned} N_i &= L_A \cdot z_{ij} \cdot X_i - L_B \cdot z_{ij} \\ N'_i &= L_A \cdot z_{ij} \cdot X_i \end{aligned} \tag{4.6}$$

All synapses on row *i* share the same $\mathbf{N}_i$ and $\mathbf{N'}_i$ line into which the generated currents are injected. $\mathbf{N}_i$ is sent to current mirror CMA_i and $\mathbf{N'}_i$ is sent to current mirror CMB_i. The initial current on the $\mathbf{N}_j$ lines is set to L_M, which is replicated 18 times by current mirror CMM; the actual value for L_M is arbitrary so long as it ensures that the terms of $\mathbf{C}_j$ are positive.

Each element of the controlled current source $\mathbf{CC}_j$ shares the same output node N'', which is the generated current

$$N'' = L_A \cdot \left(\sum_{i=1}^{N} \mathbf{X}_i \right) \tag{4.7}$$

This reaches the input of the adjustable gain control ρ current mirror, and is later replicated 18 times by the current mirror CMC onto the output lines of each CMB_i. The current on these lines, which is the input to COM_i, is then set to be:

$$\mathrm{COM}_i = L_A \cdot \left(\sum_{j=1}^{n} z_{ij} \cdot \mathbf{X}_j \right) - L_A \cdot \left(\sum_{i=j}^{N} \mathbf{X}_j \right) \tag{4.8}$$

The comparator compares this value with zero; if the current is positive then the output falls, whilst if the current is negative then the output rises. This current is sent to the input c_i of the WTA, along with the input i_i (which is a mirrored copy of $\mathbf{C}_i$ from the synapse array). If c_i is high then the input i_i will not compete for the winning node in the ART-1 network. Conversely, a low c_i implies that the input i_i (or $\mathbf{C}_i$) will compete for the winning node. The outputs of the WTA, $\overline{y_i}$, are all high, except for that node which wins the winning node contest, which has a low $\overline{y_i}$.

A problem with this approach is that with a number of uncommitted rows in the synapse, which is bound to happen even after a number of patterns have been learned and stored, there is a chance that these rows will generate currents equal to or greater than the node that ought to win. In order to avoid these problems a number of D-type registers [Roth92] may be added to the circuit. These registers are initially set to logic-1, so that the WTA inputs $s_2 \ldots s_{18}$ are high.

These inputs have the same effect as the c_i inputs: if s_i is high then Ci does not compete, but if s_i is low then it enters the WTA competition.

Initially, only $\overline{c_1}.C_1$ will compete. As soon as $\overline{y_i}$ rises (i.e. goes to zero) the input of R_1 is transmitted to its output, making s_2 equal zero. Now both $\overline{c_2}.C_2$ and $\overline{c_2}.C_2$ will compete for the winning node. As soon as $\overline{c_2}.C_2$ wins then the input of R_2 is transmitted to its output, making s_3 equal zero, and the process continues.

This method ensures that only those synapse rows that have previously won, implying that they hold a stored pattern, and one additional uncommitted row, which can hold any new uncategorisable patterns, can compete in the learning process, with all other uncommitted synapse rows effectively masked out of the process. Once all synapse rows are involved in the competition then the output signal FULL indicates that all synapse rows are storing a category. The register process is enabled and disabled via the ER input on the WTA.

4.2 Neural Network Applications

4.2.1 Traffic Management of a Satellite Communication Network

4.2.1.1 Background Information

Satellites, from their geostationary orbits 22 300 miles above the earth, can view over one third of the earth and can instantly connect any two points within their coverage [Camp87]. This, coupled with their record of high reliability, makes them the most attractive multiple access communication medium.

It is known that, for circuit-switched networks, that a non-hierarchial switching methodology performs better than hierarchial static switching [Schw87]. It has also been shown that by reserving some portion of network capacity, in order to have alternate routing possibilities, overcomes instability in the network at high load levels [Akin84].

There are many routing algorithms available to telecommunication companies, but this project [Ansa96] proposes a new traffic management scheme to improve the efficiency of a circuit-switched satellite communications network of the geostationary orbital type. It uses simulated mean field annealing (MFA), a neural network technique [Hayk94], to carry out the proposed management scheme. It includes the idea of dynamically adapting the networks as well as dynamically routing each message arrival. This allows the network itself to change due to traffic conditions, thus improving the level of service, and also to continually organise itself to minimize the cost for varying traffic conditions.

4.2.1.2 Traffic Management Scheme

A satellite communications network can be viewed as a mesh topology, with each node representing either a satellite or an earth-based ground station. Each link between nodes may have any number of circuits, but the total capacity of the network is fixed. Traffic is generated by Poisson-based random sources characterized by two parameters: the average rate of message generation and the average length of a message. The satellite communications system acts as a server system, providing a transmission service to the generated traffic.

A system model of the proposed scheme is shown in figure 4.7. The object of the scheme is to dynamically route each call, as traffic conditions change from time to time, thus maximizing the throughput of the network. This requires four different functional modules: map generation, router, controller and arbitrator.

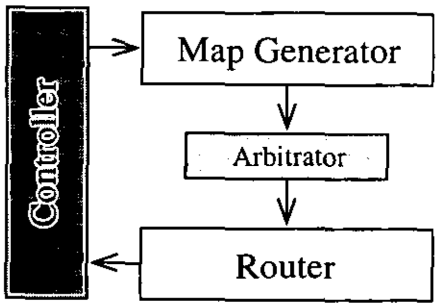


Figure 4.7 Network System Model

a) Map Generator

This unit generates a map of the best configuration for the prevalent traffic conditions. Such maps differ by two parameters **C** and **R**; the former denotes the link capacities of the network and the latter denotes the number of circuits that can be used for alternately routed calls. Therefore, in a particular link j, there are $c_j - r_j$ circuits reserved for direct calls; this calculated parameter is referred to as the *reservation* parameter. The generator also takes as inputs the current status of the network, the total capacity of the network and the average arrival rates of each origin-destination (O-D) pair.

With this information in hand an optimization technique is used to try and find an optimal map which will minimize the rate of unconnectable calls, known as the *block rate*. Two different techniques are employed in the task: simulated annealing and MFA.

b) Router

The router performs dynamic routing for every call arriving on the network as follows:

- If the direct link has an idle circuit then the arriving call is routed onto it
- If no direct link is free then a randomly selected alternate route is tried; this will be blocked if either or both links in that particular O-D pair is in a reserved state (i.e. at least *r* circuits in the link are busy)
- If direct and alternate routing fail then the call is blocked and lost from the network

This simple routing algorithm requires little computation, thus reducing the processing delay of each call.

c) Controller

The task of the controller is to keep track of the state and performance of the network. It decides whether or not a new map is necessary based upon several parameters, the most important of which are the arrival rates of the O-D pairs and the load balance of the network. The load imbalance *d* is calculated by the following set of equations:

$$\begin{aligned} \Delta &= F / C \\ \delta_{ij} &= f_{ij} / c_{ij} \\ d &= \frac{1}{C} \cdot \sum_{(i,j)} \left(\Delta - \delta_{ij}\right)^2 \end{aligned} \qquad (4.9)$$

where

F = total flow of the network

C = total capacity of the network

The controller calculates the amount by which the network's current load balance deviates from that of a fully balanced network; this measure is taken to be the indication of premature saturation of the network. The actual threshold value for this imbalance at which the controller deems the current map to be inefficient, denoted d_t, is defined as:

$$d_t = 0.1 \cdot \Delta \tag{4.10}$$

When d is larger than d_t the network is considered to be inefficient and the controller calls up the map generator module to request a better map for the current traffic conditions. Problems arise, however, when d_t is small and close to zero. In this case even a very small deviation from the ideal cannot be tolerated and the map generation process is called up too frequently; this may not be cost effective in the long run. Hence, the parameter d is only updated after a number of network status updates, so that any generated map remains in operation for a minimum number of network status updates; in the proposed traffic management scheme d is updated after 10 network status updates, with up to 100 network status updates being used in judging the overall pattern to the network traffic.

d) Arbitrator

The arbitrator decides whether or not using a modified map will actually be beneficial to the network performance; thus, it is used as a cost-saving measure. As the routing of calls must be uninterrupted, and the optimisation of such must be done in real time, there may be some instances where a map configured from the most recent network experiences may not actually reflect the optimal performance for present traffic conditions. This process can be eliminated, to a certain extent, by properly choosing both d_t and the update rate for d.

4.2.1.3 Map Generation Using Simulated Annealing

The average probability of a call being blocked by the network, $\overline{B}$, must be minimized in order to increase the performance of the network. This probability depends upon the capacity of a link and on the number of circuits that can be used by alternately routed calls in the link. These two independent variables, *c* and *r*, make the solution space of such a problem very large; selection of the optimally configured map from the solution space is computationally time consuming, and some optimization technique must be applied in order to find the required map. One powerful neural network-based technique, as described in section 3.3.2.5, is referred to as simulated annealing.

The idea of simulated annealing is to reduce the system energy and to find a state of minimum energy. For the map generation process the energy cost function is simply the total block rate of the network. The cost function for any link *s* can be written as follows:

$$E(s) = \overline{B}\,(\mathbf{C}, \mathbf{R}) \tag{4.11}$$

The artificial temperature of a simulated annealing network is reduce over time during the optimisation process. This *cooling schedule* is specific to each application and requires four parameters to be defined [GeGe84].

- *Initial Temperature* is defined so that virtually all transitions are accepted (but not all)
- *Stopping Criterion* is based on the argument that the execution of the algorithm can be terminated if the improvement in cost achieved through continuation is small; hence, if two consecutively generated maps do not vary significantly in cost then the algorithm is terminated and the first map is considered to be near-optimal
- *Number of Transitions at Each Temperature* is defined as being the level at which the generation of new states ceases and the current temperature value is modified. This is down after either a certain number of accepted states have been generated or after the generation of a specific number of new states. Due to the experimental nature of the project many different levels of *accepted* or *new* state-based rules were used

- *Temperature Updating Rule:* the difference in two step temperatures, and their relative costs, are required in determining the rule for decreasing the temperature. The rule is related to the capacity of the network, *CP*, and is given in equation 4.12, where *a* is a constant and *j* is the iteration index (which is linearly incremented)

$$T = a \cdot \frac{CP}{\ln(j)} \tag{4.12}$$

The final requirement of the simulated annealing process is to define some form of neighbourhood structure for the neural network. This work defined three different structures, each of which was used during the simulation study.

Case 1) Varying Reservation Parameter of the Link: any link has *c* circuits, with any number of circuits in the link being reserved in the range [0, *c*]. When looking for a neighbour a random number in this range is chosen and assigned as the reservation parameter for that link

Case 2) Varying Link Capacities Only: one link is chosen at random and has a random circuit deducted from it. This circuit is then added to another link that will benefit from the extra capacity. For reasons of practicality each link is also assigned an upper and lower bound capacity of circuits

Case 3) Varying Both Link Capacity and Reservation: as this has the potential to have a large number of combinations a similar control measure to that in case #2 is used

4.2.1.4 Map Generation Using Mean Field Annealing

Although simulated annealing is a powerful optimization technique it is very computationally expensive, especially when the problem search space is large. MFA is a trade-off between performance and computational complexity, and can be used effectively to minimize the call blocking probability. Full details of the theory of MFA are not given here but information and further references can be found in the literature [Pete87] [PeHa89].

Two operations from the simulated annealing algorithm are still required in MFA: the operation to reduce the temperature and the process to search for the optimal solution at each temperature. In MFA, however, the relaxation process is replaced by a search for the mean value of the solution; the equilibrium can be reached faster by using the mean and, thus, the MFA algorithm speeds up the computational process. The same three different cases of map generation from simulated annealing are considered during the MFA process.

The energy function for the network is based on an energy function by Hopfield and Tank [HoTa85] with two constraint terms, with each map generation case having a slightly different function. The base cost of the function is based around the energy costs of direct blocking, alternate blocking, the probability of alternately routing a call and the probability of an alternately routed call being blocked given the number of circuits available in the link. Full derivations of the base cost and constraint terms of the energy function, as well as the derivation of the mean field equations, are given in [Ansa96].

4.2.1.5 Performance and Conclusions

Many different simulations were carried out during this study, using both simulated annealing and MFA methods of optimization. The first set of simulations set out to show the effects of having a permanently static map; no matter what proportion of circuits were reserved for alternately routed calls, the network would go into a state of near-overload at high loads (~87% throughput) despite some capacity remaining in the network.

By varying the reservation parameter per link, yet retaining the same circuit capacity, optimization by both methods proved to be beneficial by a few points, with simulated annealing giving slightly better results. Varying the link capacities proved to be even better, giving greater network throughput. Again, simulated annealing proved to be better than MFA, but the computation times required were

Table 4.1 Annealing Simulation Results

Varying Parameter	SA Time	MFA Time	SA Thru	MFA Thru
Reservation	264.5	12.3	91%	90%
Capacity	412.4	16.7	93%	92%

prohibitive. Table 4.1 shows the computation times (in arbitrary time units) and throughput levels for both cases and both optimization techniques.

From the simulation results it can be verified that by implementing a control scheme where some portion of the link capacity is reserved for direct calls the performance of the network can be improved. It also manages to avoid instability in overloaded traffic conditions and improves the throughput of the network.

The MFA algorithm fine-tunes the network configuration and improves the network performance. Even though the results of the more common simulated annealing optimization technique are better the computation times for MFA are at less than 5% of those for simulated annealing. Therefore, mean field annealing is the optimization method of choice.

4.2.2 Prediction of a Continuous Stirred Tank Reactor

4.2.2.1 Background Information

An important area of neural computing research is the investigation into the parallelisation of neural network training algorithms and their implementation onto existing parallel computer architectures. Two possibly fruitful architectures are special purpose systems based upon transputers [DeBl90] and the linking together of workstations using PVM [Begu93].

The development of a 5-step ahead neural predictor for a continuous stirred tank reactor (CSTR) [McIr95], a typical piece of non-linear equipment in a chemical plant, is used as a case study for the efficiency of a parallel implementation of a multi-layer perceptron neural network. It is used to assess the performance of the parallel algorithms developed for both the transputer and PVM systems.

4.2.2.2 Hardware Implementations

PVM (Parallel Virtual Machine) is a software package that allows parallel programs to be run on a heterogeneous network of UNIX computers. It consists primarily of two parts:

- a daemon process that resides on all machines making up the virtual machine
- PVM library containing callable 'C' language functions for all functions regarding message passing, process spawning, task coordination and modify the virtual machine

Communication between machines on the network is slow when compared against that of dedicated concurrent processing hardware, and is also variable depending upon network and machine load. Because of this parallel programs running under PVM are only beneficial when the problem granularity is very high, thus allowing the speed and size of local processing power to be fully exploited.

Implementation on a PVM-based system is fairly straightforward once the parallel processing algorithms have been developed. The network of machines is set up and a central control process sends out remote processing calls to each machine as required. Although the setup can be (and should be) optimised by the user, there little work is required to get the algorithms up and running; many of the communication problems associated with such a heterogeneous system are handled by PVM and the machine networking protocols being used.

A transputer network, on the other hand, is a dedicated programming architecture that offers very high speed communication between adjacent processors, with each processor on the network having a direct serial connection to at most four other processors. This is dissimilar to the PVM architecture, which, by means of the inherent network protocols, allows direct communication between all processors on the network. Consequently, a suitable network for the transputers needs to be developed for each application; typical configurations include the tree, pipeline, ring and rectangular array. For the chemical plant application the most favoured architectures are the ring and the pipeline, which are best suited to neural algorithms.

A ring architecture was chosen over a pipeline, as it allows the last transputer in the network to be connected back to the first transputer; a schematic for the network is shown in figure 4.8. A ring network allows a unidirectional flow of data, which can be advantageous.

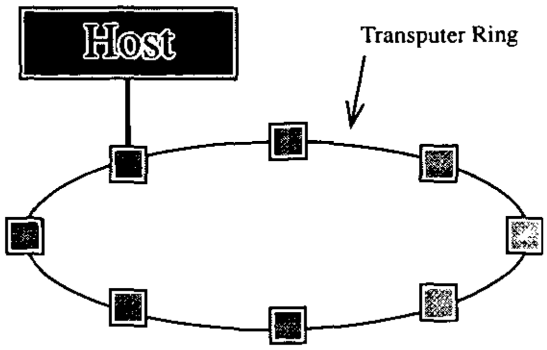


Figure 4.8 Transputer Ring Network

The structure of the parallelised algorithms devised in [McIr95] gives the following communication timing:

$$T_c = n \cdot \tau \cdot (2 \cdot N_w + N_c) + \tau \cdot (N_W + N_C) \tag{4.13}$$

where

n = number of parallel processes

τ = transmission time per floating-point value

N_c = number of function evaluations during neural algorithm

N_w = number of weights in the network

Once the transputer ring host controller has placed data on the network it does not have to wait until it has gone to the last transputer on the ring network. This reduces the communication time T_c in equation 4.13 by $(n-1) \cdot N_W \cdot \tau$.

4.2.2.3 Neural Network Implementation of CSTR Predictor

The Continuous Stirred Tank Reactor is a highly non-linear plant and, as such, is a useful example for testing neural networks. The predicted system is a single-input/single-output system: the input is the flow rate of a coolant $q_c(t)$ and the output is the concentration of a product compound $C(t)$. The delay between the tank and the product output is approximately 30 seconds. The reaction within the system is exothermic, which raises the temperature and hence reduces the reaction rate. The coolant allows the manipulation of the temperature, hence allowing the product concentration to be controlled.

A schematic of such a system is shown in figure 4.9. Although there are many parameters involved in the operation of the plant these and the related control equations are omitted from the figure.

A general non-linear k-step ahead prediction model takes the form

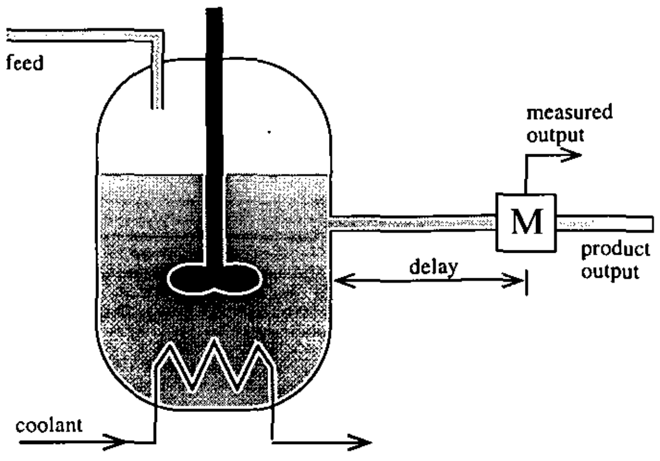


Figure 4.9 Continuous Stirred Tank Reactor

$$\hat{y}(t+k)=F\{u(t+k-1),\ldots,u(t-m),y(t),\ldots,y(t-n)\} \tag{4.14}$$

where

y(t...t-n) = past *n* outputs of the CSTR plant

u(t-1...t-m) = past *m* controls of the CSTR plant

u(t...t+k-1) = future *k* controls of the CSTR plant

F = non-linear function mapping CSTR plant variables to predicted output

The function of the neural network is to approximate the mapping *F*. For control purposes it would be useful to be able to predict across the range of the 30 second delay, using a sample time of 0.1 minutes results in a 5-step ahead predictor. Assuming the CSTR plant is of order five or less then the desired predictive model is shown in figure 4.10.

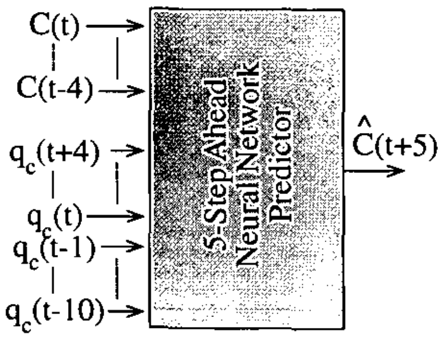


Figure 4.10 CSTR Plant Neural Predictor

4.2.2.4 Performance and Conclusions

The multi-layer perceptron, with layer sizes of 20-20-1 respectively, was trained using the full memory (FM) BFGS algorithm [McKe90], with 800 training vectors. The trained network was then tested by comparing the predictor output, delayed by five time units, with the actual output of the CSTR plant. The network performed very well, with over a 95% correlation between the predicted and actual output at all times.

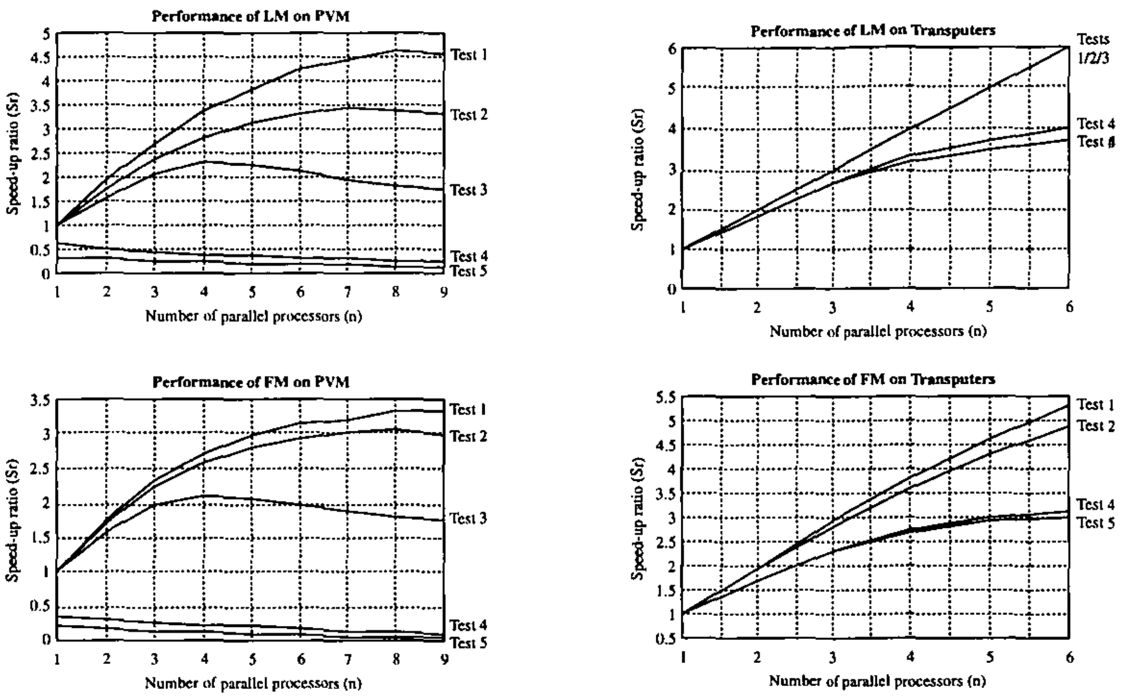
Parallel versions of the FM algorithm, along with the limited memory (LM) BFGS algorithm [Gill92], were implemented for both the PVM system and a transputer system; the PVM network consisted of up to nine idle Sun workstations and the transputer ring consisted of up to six processors.

Table 4.2 shows the test problems used on the target parallel architectures. The result graphs are shown in figure 4.11. From the graphs it can be seen that the transputer implementation performs much better than those running under PVM. This is mainly a reflection of the difference in speed in inter-processor communication, with PVM being around 20 times slower than the transputers.

Table 4.2 Dimensions of Test Problems

Name	Problem	MLP Structure	Training Set Size	N_W
Test 1	CSTR	20, 5, 1	800	111
Test 2	CSTR	20, 10, 1	800	221
Test 3	CSTR	20, 20, 1	800	441
Test 4	y=(x-2)(x+1)	1, 5, 1	25	16
Test 5	y=(x-2)(x+1)	1, 10, 1	25	31

The number of training vectors and, to a lesser extent, the number of weights N_W determine the achievable speed-up for a given problem. On the PVM system the algorithms performed reasonably well for the CSTR problem where the training set is large but performs terribly where the training set is small. Thus, a PVM implementation of neural networks should only be considered for large problems.



Performance of LM on Transputers

Figure 4.11 Algorithm Performance

The speed-ups achievable for the CSTR problem on the transputer system approach the theoretical maximum for the number of processors involved. Performance is also reasonable for small problems, although far from optimal. Test 3 of the FM algorithm failed completely due to insufficient memory, showing that a PVM-based solution for such a large problem is a feasible alternative system architecture to the transputer network.

5

PROPOSED ARCHITECTURAL DESIGN

INTRODUCTION

This section introduces the architecture for an instruction systolic array processor, optimised for on-chip learning of artificial neural networks. It goes on to give detailed descriptions of the processing elements within the array processor, with high-level circuit diagrams for each of the functional units. The section concludes with details of the instruction processing methodology used by the array processor, with high-level circuit diagrams for the functional units.

5.1 Architectural Overview

5.1.1 Array Structure

There are many different possibilities for the implementation of VLSI chips that attempt off- and on-line learning for neural networks [Trel89]. With the digital devices the implementation method is normally to dedicate processing elements to being neurons in the network, with each neuron having the circuitry required to carry out a matrix-vector product operation. The neuron circuit is also responsible for holding, normally in local memory, the set of weights associated with it within the network.

As the kernel of the majority of artificial neural network computations is the matrix-vector product it was decided to design an architecture which took advantage of this fact, yet was still able to carry out neural network learning for a variety of network training methodologies [KaEv96]. Each element of the matrix-vector product can be seen to be the result of a neuron input scaled by a synaptic weighting value. By dedicating an individual PE to the processing required for an individual synapse, but also allowing the PE to perform other arbitrary mathematical calculations, a system can be designed that can perform both synaptic and neural computations.

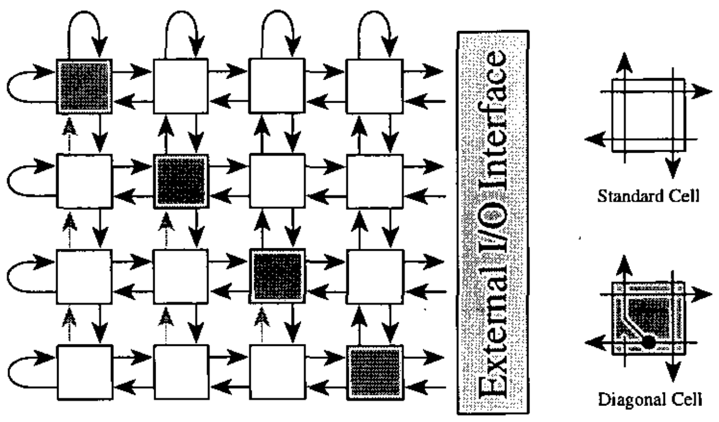


Figure 5.1 Neural VLSI Array Architecture

The basic array structure, as shown in figure 5.1, is similar to some previous work [Lehm93], but the similarity ends there; the internal workings of the PE's are vastly different. The direction of data traversal is indicated with arrows joining input and output connections between neighbouring PE's. The actual array size used for this work is a 6–by–6 array but, for the sake of clarity, figure 5.1 only shows a 4–by–4 array.

Each PE is joined to each of it's four neighbouring PE's with an input and output connection, although there are some exceptions depending on the PE's location in the array: west- and north-edge PE's wrap the relevant connections out back onto themselves, east-edge PE's connect directly to the array I/O controller. South-edge PE's have no southerly neighbour connections at all, implying data output to the south from a south-edge PE is effectively lost. Any data transmitted to the north from a south-edge PE is meaningless, as there is no corresponding southerly input - this is illustrated in figure 5.1 by light-shaded data arrows. Each PE can carry out some computation, with the results being made available on the east or south output of a PE, overwriting any data originally input from the west or north of the PE. Leading diagonal PE's copy the east-west input data on to their south-north output, effectively routing the external I/O data to the array north-south data path (after a short delay).

5.1.2 Instruction Systolic Array Processing

Section 2.3 described the architecture of the systolic array, as well as describing several of the more common algorithms associated with it. It may be desirable, however, to have a procedure that requires more than one type of operation; PE's within the systolic array have an option as to what type of operation they can carry out in any given clock cycle. The option to be realised is determined by an instruction tag that is input to the cell along with a data item. Using this technique individual PE's within a systolic array can carry out a number of different operations; a systolic array in which instruction flows are involved as well as data flows as known as *instruction systolic arrays* [Kund86].

In an instruction systolic array the instruction and data streams can be considered within a common framework. Both streams are input to a PE, with values on the output data stream possibly being replaced as a result of some computation, and both streams are then propagated by the PE throughout the rest of the systolic array. The instructions are actually considered as data which is first processed by the PE, in that each cell first analyses what kind of instruction it has received - processing proceeds according to the result of this analysis.

An example of an instruction systolic array algorithm, back substitution in triangular linear systems of algebraic equations [Petk89], is shown in figure 5.2. A sequence of instructions is input to the right-most PE of the array and passed to the left on successive clock cycles. Data is passed into the array along with the instruction and also on the top input to each PE. In

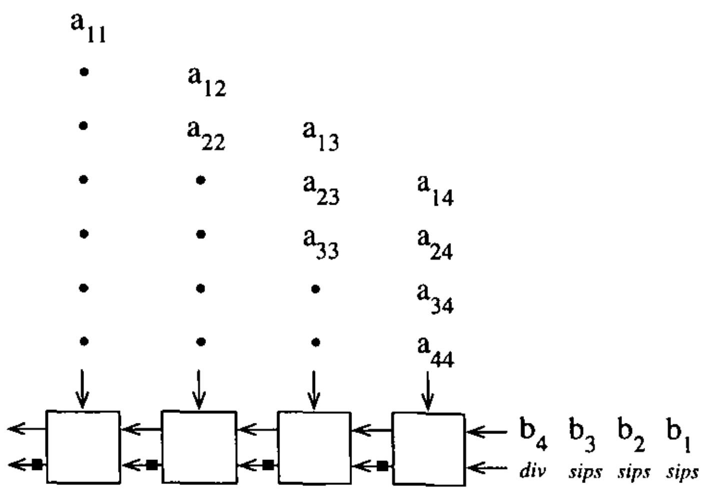


Figure 5.2 Back Substitution using Instruction Systolic Arrays

this case, the sequence of instructions consists of one division instruction followed by a number of inner product step instructions. In any given clock period each PE executes one single instruction out of the many that they are capable of processing.

In any implementation of instruction systolic arrays the PE's might be rather powerful processing nodes, with considerable local program and data storage with little or no direct connections to the host computer. In such a system the array itself must synchronise the inter-PE exchange of data and instructions, which may lead to more complex PE's. However, such a system also has the capability to be a very powerful parallel array processor.

5.1.3 Array I/O Requirements and Processing

There are two types of PE in the array, both being identical save for one feature. PE's on the leading diagonal of the array have their south → north data connection (SN) shorted, with data from the east → west data connection (EW) being copied onto it. Data on the north → south connection (NS) and west → east connection (WE) are identical in both types of PE. This method allows data to be sent from the I/O controller on the EW datapath to be fed, after a certain time delay, to the WE and NS datapaths using the array west- and north-edge PE wrap-around feature. When data enters a PE on it's NS datapath then an operation is carried out within the PE. Note that 'data' here implies a numeric data item and an instruction operation code, which is decoded within the PE and acted upon.

One major difference between this architecture and that normally associated with two-dimensional systolic array processors [KungSY88] is that external data I/O exists solely on one edge of the array. Normally, at least two edges of the array are used for data input, with a third edge used for the collection of result output values. By only using a single edge of the array for both input and output, letting the array itself to distribute the data to the relevant PE's, only a fairly simple I/O controller is required [Lehm93]. The controller only has to concern itself with a single two-way communication channel.

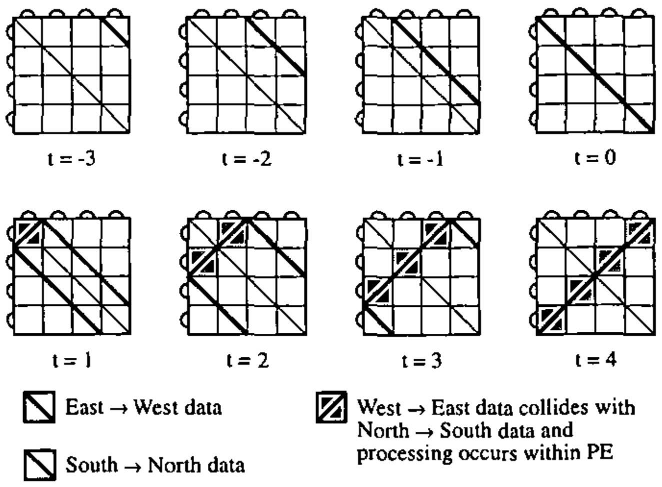

t = -3
t = -2
t = -1
t = 0
t = 1
t = 2
t = 3
t = 4
East → West data
South → North data
West → East data collides with North → South data and processing occurs within PE

Figure 5.3 Internal Datapath Timings

During a normal run of an algorithm data is passed in on the east edge of the array, and this data proceeds westwards over several cycles until data reaches the diagonal cells, where a copy is then placed on the SN datapath. As in other systolic designs the east-edge input data is staggered in a diagonal fashion, and this entire process is illustrated in figure 5.3. Data that reaches the west or north edges of the array is redirected onto the WE and NS datapaths of the respective PE's. Whenever a PE finds data on it's NS datapath it extracts the operation code from the data, decodes it and then carries out the appropriate instruction data as required. This system allows the operation codes and data to be embedded into a single data stream.

A matrix-vector multiplication is straightforward to show as an example operation, as it can be done within the array using a single replicated instruction. Assume that each PE has been set up to process the instruction [MVM] as

[MVM] *col 1* NS * REG → WE′

[MVM] *other* (NS * REG) + WE → WE′

where REG denotes the contents of a private register within each PE. The register within each PE initially holds a value from a matrix, and an input vector is sent into the array. The PE's process the instructions, once they receive them, by multiplying the contents of the internal register with the data on the NS datapath; if the PE is not on column 1 (the west edge of the array) then it also adds any value on the WE datapath to the result of the multiplication[1]. The final result is placed on the PE's WE datapath, ready to be output on the next cycle.

[1] By having two versions of the same instruction in the array the I/O controller does have to concern itself with the awkward task of ensuring that PE's in the first column of the array receive a zero on their WE input, in order to start the accumulation process, at the same time as the [MVM] instruction arrives on their NS input

It can be seen that each PE in the array calculates a partial result and passes it along it's WE datapath to the PE on it's east edge, where it is used in future calculations. After $2n$ cycles (where n is the dimensionality of the array) the first row of the array outputs a result, with other rows outputting their results on subsequent cycles; each component of the result vector is output from the array after $3n$ clock cycles.

The instructions and data arc sent to the array in a staggered fashion, and most algorithms presented in this thesis follow this pattern. It allows each row in the array to act as an n-stage pipeline, with each PE calculating partial results for use by the PE on it's east side. If an additional vector is input to the array immediately following the first the result of this second matrix-vector multiplication is available on the cycle after the first result is output. This allows for y matrix-vector multiplications, all utilising the same matrix, to be processed in only $3n+y-1$ cycles; this method of pipelining is vital to the overall speed of the array and is utilised to the maximum extent possible.

5.1.4 Data Format and Precision

There are a number of mathematical and storage units within each PE, all of which are described in detail in section 5.2. They consist of a fixed-point adder, a fixed-point multiplier and a bank of local memory registers. Each of these units are 12-bits in size and arranged in a standard representational format; the nominal representation of a number is shown in figure 5.4.

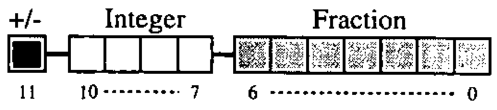


Figure 5.4 Data Format

Due to the fairly low precision used in the data representation a system to capture errors will need to be implemented. With a 12-bit fixed-point multiplier of the format shown in figure 5.4 the generated result will be 23-bits in size, with the integer and fractional parts of the result being 8- and 14-bits in size respectively. The fractional part will need to be reduced down to just 7-bits, and the integer part will need to be checked for overflow or underflow; there are problems associated with both of these schemes.

Truncating the fractional part of the result gives a maximum absolute error of 1/128. By using the bit-value in the 1/256 position of the fractional result [Mano82] it is possible to correctly round the fractional result up or down by 1/128, thus reducing the maximum absolute error to 1/256; a '1' in the 1/256 position indicates that rounding up is required. This method cannot be employed, however, as it would require either an extra adder unit per PE or the extension of the execution time for instructions by an additional cycle in order to re-use the existing adder. The former option is undesirable, as although a single adder is not very large the systolic array is designed to consist of an array of 6-by-6 PE's, so the addition of a single large circuit in each PE is quite expensive in terms of circuit area. The latter option is also undesirable, as the systolic array is being designed so that every instruction is processed in a single cycle; this option would practically double the execution time of every algorithm. The method of truncation, therefore, is a straight cut-off and loss of all bit values less than 1/128.

In a normal serial processor all mathematical operations have an error flag associated with them in order to indicate events such as underflow, overflow or division-by-zero. Although this is a tried and tested method it cannot be used in the systolic array. If the first PE in a row indicates that an operation resulted in overflow then the result being fed to the next PE in the row is garbage. Such a result cannot be operated upon at all by subsequent PE's, as it cannot be assured how much the result is in error; subsequent processors should effectively ignore such data values.

Hence, each mathematical unit in a PE will output an *integrity* flag along with any result, in order to indicate if it is safe to use the data associated with it. The units will also replace the generated result value with a number representing the maximum or minimum representable value. The mathematical units will inspect the integrity flags associated with each input value; if either flag is set then it is safe to assume that any result generated by the unit will also be in error, even if the result is technically correct[2]. This method, known as saturation, is fairly

[2] If an overflowed-maximum value is added to an underflowed-minimum value then the actual result is -1/128. Such a value cannot be used, however, as it cannot be guaranteed that one of the inputs was not meant to be of greater value by several magnitudes. Hence, such results must also be flagged as being in error at all times

common amongst pipelined processors, where saturated additions and multiplications are necessary in order for an intermediate result to hold on to some integrity. The only exception to this technique is that the correct sign for the multiplication result is always used, whereas with an addition operation the sign of the input in error is used for the result (or an arbitrary choice if both are in error).

5.1.5 Processing Element Structure

Figure 5.5 shows a schematic diagram of an individual PE within the array. It shows the four main data paths, although associated input registers and output buffers are omitted for the sake of clarity. It shows two calculation units, a data comparator, a result range limiter and an internal register storage block. An instruction opcode memory block is shown, containing fast static memory cells and the circuitry necessary to re-program the memory; note that for the sake of clarity no control signals from the instruction memory to other units are shown in figure 5.5. A number of multiplexors are also shown, which route data to the correct area of the PE during instruction execution. The internal data routing unit is simply a large area of interconnect, arranged so as to connect each possible data bus to the relevant input to the PE functional units.

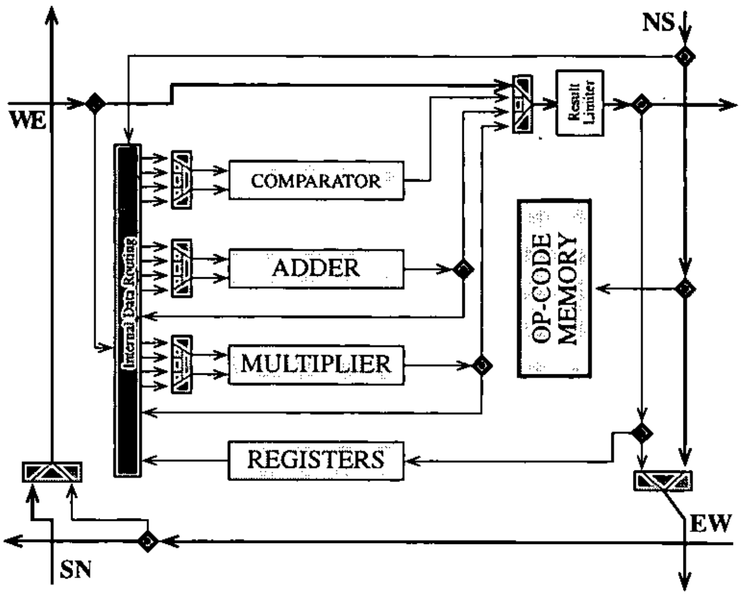


Figure 5.5 Processing Element Schematic

The multiplexor in the lower-left area of the schematic is used to redirect data from the EW databus onto the SN databus. If the PE is on the leading diagonal of the array then this circuit is present, otherwise it is removed completely, allowing the original SN input to be passed on unmolested to other PE's in the array.

5.2 Neural Network Hardware Features

5.2.1 On-Chip Learning Methodology

Of particular interest to the neural network training algorithms is the internal register block within each PE. This holds four 12-bit words of data, and each word can be addressed individually by using an *active* switch within the block. It has been shown [Pao89] that the backpropagation network requires at most only three layers of neurons to represent any arbitrary function. As this algorithm has been proven to converge and learn any function that the network can represent [Rumm86] any implementation of backpropagation only has to provide three layers of neurons in order to operate correctly. Although more layers can be added Pao's work showed that this isn't necessary.

The proposed architecture sets aside three register slots per PE in order to store a network weighting value. As each PE represents a synapse rather than an entire neuron these register weights can be seen to represent the weighting factor between one neuron output and another neuron synaptic input. By activating all layer-1 registers within the PE all weights for neurons in that layer are visible to the processing routines; weights for neurons in other layers are still held within the PE but are not directly accessible[3]. This method allows all weights for all neurons in the network to be held on-chip at all times, thus dramatically reducing the communication overheads typically associated with hardware implementations of neural networks that hold the weights off-chip in a separate data store.

Another benefit from the on-chip learning methodology, besides the decrease in required I/O bandwidth, is that any other operations that require weight values in the calculations do not need to load the values on to the chip: they are already present in the PE's. The flowing nature of systolic algorithms means that as the weight values remain fixed within the PE's themselves instruction opcodes can pick up the values as the instructions flow through the array.

[3] PE's in the array have their own private *active* register indicator - it is not a global value, but the routines used in the neural network algorithms tend to set all PE's to have the same active register indicator

The fourth register within the PE is used as a general purpose register on non-neural operations. It is loaded with constant values required during a pass of an algorithm, or perhaps as an accumulator of partial results. This allows other mathematical operations to be carried out with having to remove the weight values from the array, as there is still some spare local memory capacity within each PE.

5.2.2 Reconfigurable Instruction Set

5.2.2.1 Implementation Overview

The normal method of decoding instructions within a PE is to use a small programmable logic array [WeEs88], with the control signals resulting from an instruction being hard-wired into it. This results in a function similar to a ROM device. However, it has the problem that the effects of instructions cannot be varied at a later date, so any instructions implemented would have to be very general in nature or optimised for a specific algorithm.

The preferred method for the instruction set decoding is to use a small block of fast static RAM inside each PE. This will hold the relevant control information for the number of instructions required. The RAM address range is to be limited to just 4-bits, giving a capacity for 16 unique instructions, giving a system bus width of 16-bits (including 12-bits for data accompanying the instruction opcode). Each PE in the array should hold the same instruction set for a single algorithm, although if an algorithm requires more than 16 instructions then the PE's could be programmed with slightly different instruction sets; i.e. if instruction `[0110]` is only required in the first row of the array then it need not be duplicated in the PE's in other rows, which are free to carry out a different task when executing instruction `[0110]` (although different implementations of the same opcode should be conceptually similar)

The opcode number itself is used as the address for the RAM, with data being read from or written to it as required. Each opcode has an associate mnemonic code, which is used as an aid in designing and interpreting the algorithms.

5.2.2.2 Fixed Instructions

There are a number of instructions that are fundamental to the operation of the array processor and need to be hard-wired into the instruction set. These cannot be overwritten with any other data and can be treated as constant values. This part of the static RAM is actually implemented as a ROM circuit: all write operations to these instructions stores are ignored, but read operations are carried out as normal, as the individual cells within the memory still conform to the protocols required by the reset of the memory unit. A schematic layout of such cells is given in figure 5.6, which shows the layout for a hard-wired logic-1 (a logic-0 has the V_{DD} and V_{SS} connections reversed). A full description of the operation of the RAM and ROM units of the instruction set memory is given in section 5.3 along with descriptions of the other hardware elements of the architecture. A list of the instruction opcodes that are fixed in ROM are given in table 5.1.

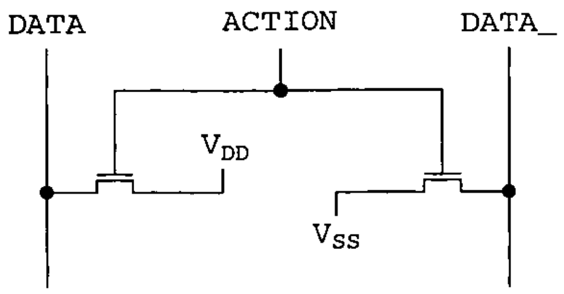


Figure 5.6 Opcode ROM Bit-Slice

Table 5.1 Instructions fixed in ROM

Opcode	Mnemonic	Description
[0000]	[PASS]	No processing occurs, but data passed in all directions
[0001]	[SWITCH]	Switches currently active register within specified PE
[0010]	[LOCK]	Sets lock status of specified PE
[0011]	[PROG]	Programs a new instruction into the opcode memory Also used to set values in result range limiter

[PASS] a simple no-operation instruction; no processing occurs and all input data is propagated to the respective outputs

[SWITCH] sets the currently active register within the PE to that specified by the data on the WE input stream. This value can be in the range 0...3

[LOCK] locks the instruction that appears on the subsequent clock cycle on the address input into a local memory buffer area, but only if the PE is on the row specified by the value on WE - this can be in the range 0...5. Locked instructions are processed on every cycle within that PE regardless of the instruction present on the NS datapath until another [LOCK] instruction is received for a PE in that same row

[PROG] nothing occurs until the subsequent clock cycle: the instruction then present on NS is programmed with the data associated with it in the opcode memory, thus re-programming that instruction. If the instruction to be re-programmed is a [SWITCH] or [LOCK] then nothing happens; if the instruction is [PASS] or [PROG] then the result range limiter (as described in section 5.3) are set up instead

As well as the specialised functions carried out by the ROM-fixed instructions all of them process an effective [PASS] on all other input data. Unfortunately, the 12-bits of data accompanying an opcode is not sufficient to handle all of the control signals required for an instruction. On the clock cycle that the [PROG] instruction is received the data that resides in NS[0:9] is stored in a temporary register. On the subsequent clock cycle this 10-bits of stored data is used along with the entirety of the new NS data in order to program the instruction control signals. Hence, the PE requires 22-bits of control signals per instructions, implying that the instruction memory unit is 16x22-bits in size.

5.2.2.3 Internal PE Control Signals

Table 5.2 shows all of the mathematical operations are possible within a PE[4], all of which have been implemented in order to give the array processor a useful instruction set. All of the five active data areas can be operated on in any combination; an addition can follow a multiplication in the context of a single instruction (and vice versa). The inputs to the adder and multiplier can accept any combination of inputs, although the arithmetic units cannot feed their outputs back onto their inputs. Input selection is done via several 4-to-1 multiplexors, each of which require 2-bits of control data. The adder can also act as a subtractor, requiring a single additional control signal to switch between addition and subtraction mode. Hence the arithmetic units require a total of 9-bits of control data.

Table 5.2 Possible Mathematical Operations

MULTIPLIER	ADDER	WE Output
REG * NS	MULT + WE	WE Input
WE * NS	MULT + REG	Comparator
WE * REG	WE + REG	Adder
REG * ADDER	WE + NS	Multiplier
WE * ADDER	MULT + NS	
ADDER * NS	REG + NS	

[4] The table does not show where an arithmetic unit uses the same input variable on both input ports

There are four possible values to be output from the PE, as indicated in table 5.2, and this is implemented using an additional 4-to-1 multiplexor. This requires 2-bits of control data. Although a compound instruction such as NE*(REG+WE) implies that the value to be output is the result from the multiplication this must be made explicit to the PE, as there is no way of telling directly which value to output. The multiplexors simply open up pathways for the data within the PE; there is no inherent intelligence as to which pathway is routed to the WE output of the PE. Hence, the result to be output must be specified.

The comparator unit carries out three different functions, all of which are listed in table 5.3. All of these functions operate on pairs of data, and a list of all possibilities are also shown in table 5.3. Two control signals are required for the comparator unit to put it into either equality mode or comparator mode; comparator mode is split into two further functions, allowing the unit to return either the maximum or minimum number from its two inputs.

Table 5.3 Comparator Data

Function	Upper Input	Lower Input
Equality (x,y)	NS Input	Multiplier
Maximum (x,y)	WE Input	WE Input
Minimum (x,y)	Register	Register
	Adder	Adder

In order to implement an ABS function, whereby the absolute positive value of some input is returned, it is required to multiply the input value by 1.0 or -1.0, depending on whether or not the input is positive or negative. This method takes advantage of the simple fact that a multiplication of two numbers with the same sign always results in a positive number.

The lower input to the multiplier contains an ABS unit, which requires two control signals. The first signal enables the unit and the second instructs it as to what operation to carry out; the options are ABS(x) and -ABS(x). If the unit is not enabled then the second input to the multiplier passes through unchanged. If the unit is enabled then the ABS unit replaces the second value by ±1.0, depending on the operation required. The relation

Table 5.4 ABS Unit Logic

Input Sign	Control	ABS Out
+ve	1	1.0
+ve	0	-1.0
-ve	1	-1.0
-ve	0	1.0

between the operation control signal and the output of the unit is shown in table 5.4. Note that this operation changes the sign of the value on the first multiplier input value depending on the sign of the data value on the second input; the sign change can be on the basis of a different value, although it is more usual to base a sign change using the same value as that on the first input.

There are four different combinations of destination for the chosen result value, and these options are shown in table 5.5. Regardless of the current instruction the result is always sent to the WE output. Results can also be copied to the NS output, in order to propagate a result to PE's in other rows. The result can also be stored in the currently active register within the PE. This can be done even if the currently active register was a source of data for the current instruction, as the registers are designed to be read from and written to safely in the same clock cycle - this is discussed further in section 5.3. Two control signals are required for this process, as separate signals are used to copy the result to the NS output and also into the currently active register.

Table 5.5
Result Destinations

Destination
WE
WE, NS
WE, REGISTER
WE, NS, REGISTER

Section 5.1.4 stated that the accuracy of data in the system is ±1/128. The neural approximation function used in backpropagation, as shown in equation 3.2, has this result given an input of approximately ±4.844. Hence, any input values outside of this range to the activation function will give the result of 0.0 or 1.0; this is to be avoided, as the result of the function can never equal these values (a result of 0.0 is particularly unwanted). The overflow units described in section 5.1.4 can be modified to accept an additional pair of maximum/minimum numbers, which can be loaded in via a [PROG] instruction. A single control signal is required to indicate to the mathematic units which set of numbers to use: the system maximum/minimum numbers or a set relevant to a particular function, such as ±4.844 in the case of the sigmoid activation neural function.

Table 5.6 Summary of instruction-based control signals

Unit	Signals	Value	Implication
Adder Control	1	0	Addition operation (Upper + Lower)
		1	Subtraction operation (Upper - Lower)
Adder (Upper Input)	2	00	WE input data
		01	NS input data
		10	Currently active register
		11	Multiplier unit output
Adder (Lower Input)	2	as above	as above
Multiplier (Upper Input)	2	00	WE input data
		01	NS input data
		10	Currently active register
		11	Adder unit output
Multiplier (Lower Input)	2	as above	as above
Comparator Control	2	00	Return maximum value
		01	Return minimum value
		1x	Equality operation
Comparator (Upper Input)	2	00	NS input data
		01	WE input data
		10	Currently active register
		11	Adder unit output
Comparator (Lower Input)	2	00	Multiplier unit output
		01	WE input data
		10	Currently active register
		11	Adder unit output
Result Selector	2	00	WE input data
		01	Comparator output
		10	Adder unit output
		11	Multiplier unit output
ABS Unit	2	0x	ABS unit disabled
		10	Negative ABS(x) result required
		11	Positive ABS(x) result required
Result Destinations	2	00	WE output only
		01	WE output and NS output
		10	WE output and currently active register
		11	WE output, NS output and currently actuve register
Result Range Limit	1	0	Use standard maximum/minimum overflow values
		1	Use user-defined limiting values

Table 5.6 gives a full summary of the control signals present within the PE. This gives rise to a large number of possible mathematical instruction combinations, and these combinations are given in table 5.7. There are a total of 11 basic mathematical operations, given that $2ax$ is equivalent to $2xy$. Other instructions can be made equivalent by variable substitution, but are otherwise distinct ($2x \equiv x+y$ if $x = y$). These 11 instructions can have any combination of WE,

NS and REGISTER as the x, y and a variables, and can have the final output re-directed to several different locations.

Table 5.7 Instruction Combinations

Main Input Type	Single Op	Combined #1	Combined #2
ADDER $x+y$	$x + y$	$a(x + y)$	$(x + y)^2$
MULTIPLIER $x*y$	xy	$a + xy$	$2xy$
ADDER $x+x$	$2x$	$2ax$	$4x^2$
MULTIPLIER $x*x$	x^2	$a + x^2$	$2x^2$

An instruction consisting of just an addition or multiplication operation, all of which are shown in the *Single Op* column in table 5.7, uses either two input variables x and y or uses a single variable x as both inputs to the operation. An instruction consisting of an addition and a multiplication (in any order) can either introduce a third variable a into the second operation (shown as *Combined #1* in table 5.7), or use the result of the first operation as both inputs in the second (shown as *Combined #2* in table 5.7).

Note that as the adder can also act as a subtractor, the applicable range of these 11 basic operations is quite extensive, especially as none of them make reference to either the basic comparator or the result range limiter, which increases the operation range even further.

5.2.3 Activation Function Approximation

5.2.3.1 Standard Approximation Methods

Many of the feed-forward networks, including Backpropagation, use a non-linear activation function to evaluate the output value of a neuron. This function takes in some value *NET*, which is the sum of the products of the inputs to the neuron and the associated weights, and performs the operation *OUT = F(NET)* on it. The sigmoidal activation function used in backpropagation, shown in figure 5.7, was first given in equation 3.2 and is defined as:

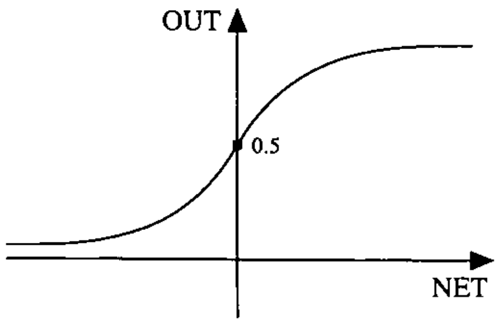


Figure 5.7 Sigmoidal Activation Function

$$\mathrm{OUT} = 1/\left(1 + e^{-x}\right) \tag{5.1}$$

Each PE in the processor array contains a single fixed-point multiplier and adder. The activation function cannot be calculated in a single PE nor, due to the nature of the exponential function, can be calculated exactly; hence, some form of function approximation must be used. A number of PE's can be used in the approximation algorithm in order to chain together a number of related calculations.

The approximation of equation 5.1 can be divided up into two separate operations; the calculation of the exponential followed the calculation of its reciprocal. This method is preferable to approximating the whole function at once, as such a complex function may be hard to implement in a simple fashion. Using Taylor's formula [Swok88] the initial approximation of the exponential can be calculated as

$$e^x \approx 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} \tag{5.2}$$

which can be implemented on the processor array fairly easily. An algorithm that requires just a single pass of the array is devisable to solve equation 5.2 with terms up to x^7 when using a 6x6 array of processors - this is reasonably accurate for an approximation of the function.

Problems arise, however, when the reciprocal of $1+e^x$ is carried out. The standard method for approximating the reciprocal function is the iterative calculation

$$x_{t+1} = x_t \cdot (2 - N \cdot x_t) \tag{5.3}$$

where $x_i \to 1/N$. This requires a good approximation for the initial value for x_1, which must lie in the range $0 \ldots 2/N$. With this initial approximation in hand the iterative algorithm converges to a good value for x within four iterations, but this is not guaranteed - it is dependant on the accuracy of the initial value for x. A single iteration of the algorithm can be done with a single pass through the array, so a total of six or seven passes to calculate the activation function seems quite feasible. The attractiveness grows with the realisation that the activation functions for all neurons within a single layer of the neural network can be carried out in parallel.

The success of the reciprocal approximation is based entirely on the closeness of the initial value for x to the desired solution. Not much processing time can be devoted to this task, but so long as the value falls within the range 0...*2/N* then it should be acceptable for use. Given the data precision inherent in the processor array, as discussed in section 5.1.4, a good initial approximation has been found to be given by the following method:

(i) Find top set bit in *N*; ie if $2^{y+1} > N \geq 2^y$ then top set bit is 2^y

(ii) Reset all bits in *N* to 0, save top set bit which is left as 1

(iii) Reverse all bit values to form fraction; ie 4 becomes 1/4

The value from this action always lies in the range *1/N...2/N*. However, as only 7-bits of accuracy is used for the fractional part of the data the iterative routine to calculate the reciprocal introduces and error which results in intermediate values in the calculation being occasionally 'lost'. Values internal to a PE will be set to zero as a result of a calculation. This results in x_t being set to 0 at some stage, implying that the final result for *1/x* being set to 0. As well as a few values between $x=1$ and $x=95$ having this problem all values of $x>96$ do it at some stage of the algorithm, even when the number of passes is restricted to four.

Examples of execution of the algorithm are shown in table 5.8 for $64 \leq x \leq 55$. Data in the shaded cells are too small to be representable in only 7 bits of data and result in a value of zero. Such zero-results always happen on the first iteration of the algorithm, which led to the conclusion that an iterative method for calculating the reciprocal cannot be used, given the level of data precision within the processor array.

Table 5.8 Iterative Reciprocal Calculation Errors

X	1/X	2/X Est	Pass 1	Pass 2	Pass 3	Pass 4	Error
55	0.01818	0.03125	0.00879	0.01333	0.01689	0.01809	0.00009
56	0.01786	0.03125	0.00781	0.01221	0.01607	0.01768	0.00018
57	0.01754	0.03125	0.00684	0.01101	0.01511	0.01721	0.00034
58	0.01724	0.03125	0.00586	0.00973	0.01397	0.01662	0.00062
59	0.01695	0.03125	0.00488	0.00836	0.01260	0.01583	0.00112
60	0.01667	0.03125	0.00391	0.00690	0.01094	0.01470	0.00197
61	0.01639	0.03125	0.00293	0.00534	0.00893	0.01300	0.00339
62	0.01613	0.03125	0.00195	0.00367	0.00650	0.01039	0.00574
63	0.01587	0.03125	0.00098	0.00189	0.00356	0.00632	0.00955
64	0.01563	0.01563	0.01563	0.01563	0.01563	0.01563	0.00000

Hastings [Hast55] has shown an alternative method for generating approximations to functions. This method is based loosely on Chebyshev polynomials, and tends to be a simple sum of products: various power terms of *x* are multiplied with pre-determined weights, then summed with or without other weighting factors, then possibly having another function applied to it. Hastings' approximation for e^{-x} is

$$e^{-x} \approx \frac{1}{\left(1 + a_1 x + a_2 x^2 + a_3 x^3\right)^4} \tag{5.4}$$

$$\left(a_1 = .2507213 \qquad a_2 = .0293732 \qquad a_3 = .0038278\right)$$

which also shows the values of the three weighting factors. The maximum absolute error for this approximation is given as 0.0002, but this can be further reduced by increasing the number of terms in *x* used in the approximation and by modifying the associated weights. This increase in accuracy comes at the cost of an increase in required processing power.

Dispensing with the problem of implementing the final reciprocal operation it can be seen that some of the numbers involved in this approximation require a large amount of accuracy. The weights require more accuracy than the processor array can supply, although this just results in the maximum error increasing; the resulting error is unacceptable, as it can be seen that a_3 cannot be represented at all, thus nullifying the effects of x^3. Also, the small value of the summation is taken to the fourth power, which will again lose a large degree of accuracy.

Although the Hastings method is not directly applicable to the required function approximation the basic idea of finding a simple polynomial expression that approximates the function is still very appealing. The only restriction is that the intermediate values within the expression do not become either too large or too small for the PE's to cope with.

5.2.3.2 Other VLSI Approximation Methods

In an analog device the non-linear sigmoidal activation function is easy to implement, as it can be based on the non-linearity of a simple device, such as a transistor or diode. However, the shape of the sigmoid cannot be controlled in any way, as it is dependant on a physical device. Also, difficulties can arise in the calculation of the derivative of the activation function which, as outlined in section 3.2.2.1, is required for the learning phase of the backpropagation learning algorithm.

Digital solutions to the sigmoidal activation function fall into two main trends: by using look-up tables and ROM tables [Nigr91] and by summing a truncated Taylor series expansion. This second trend can be sub-divided into two additional sub-classes:

(i) sum of steps approximation [Beiu92]

(ii) piece-wise linear approximation [AlSt91] [Myer89]

although there are a few dedicated approximation methods specifically for the solution of the sigmoidal activation function [Pesu90]. Examination of the literature shows that the look-up table approach falls short of the goal of having a good performing algorithm in a small silicon area; this performance/price goal is often an important practical consideration when it comes to develop such approximations in hardware. The approach of having some form of Taylor-series expansion, i.e. a polynomial based expression, is superior to such look-up table methods. The methods presented, however, either require additional hardware in each PE, which would be particularly expensive, or the algorithms assume large-precision floating-point mathematical capability in the target hardware system. Although these methods are successful in their approximation they are not particulary suitable for implementation in a systolic array.

5.2.3.3 The Bézier Curve

The Bézier form [Bézi70] of the cubic polynomial curve segment plots the intermediate points between a start point P_0 and an end point P_3, using the set tangent vectors between those points and two further points P_1 and P_2; this is shown in figure 5.8. Note that these two additional points in the figure do not lie on the curve itself, although it is possible for them to do so. The Bézier curve interpolates between the start and end points and approximates between the other two points using a set of expressions known as *Bernstein polynomials*, which act as weighting functions for the curve. The derivation of the Bernstein polynomials is not given here but can be found in a more concise and readable form in [Watt89], with the relationships between this formulation and other cubics in [Fole90].

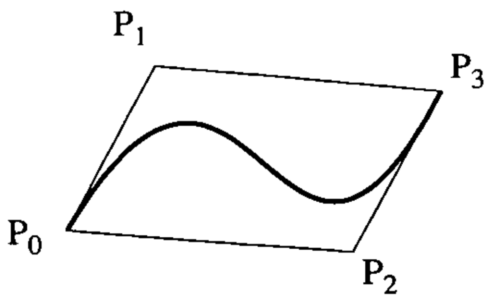


Figure 5.8 Bézier Curve Segment

Figure 5.8 shows that a single Bézier curve segment can represent a simple curve with two inflexions along its length. The backpropagation neural activation function *F(NET)* has such a curve, so if it is possible to find a Bézier representation of the curve then there exists a good approximation of the function that requires just the evaluation of a polynomial expression. The calculation required to find the *x*- or *y*-coordinate of a point on a Bézier curve is simply a four-element vector-product, which is easily realisable on the systolic array. The equation required is given by

$$B(t) = (1-t)^3 \cdot P_0 + 3t(1-t)^2 \cdot P_1 + 3t^2(1-t) \cdot P_2 + t^3 \cdot P_3 \qquad (5.5)$$

or in vector notation

$$B(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} \qquad (5.6)$$

One of the properties of the Bézier curve is that neither an *x*- or *y*-coordinate is supplied to the polynomial expressions, rather a location is specified that is a percentage along the curve's length between P_0 and P_3. The expressions return either the *x*- or *y*-coordinate, depending on whether the *x*- or *y*-coordinates of the control points are used in the calculation; i.e. to calculate the *x*-coordinate for a point along the curve then the *t*-position along the curve is supplied along with the *x*-coordinate of all four control points.

The output of the backpropagation activation function *F(NET)* lies in the range 0...1, although neither limit is ever actually reached. As the minimum representable value with a PE is 1/128 all values for the input to *F(NET)* that produce smaller results than 1/128 (or larger than 127/128) should be modified so that they produce these limiting results. Hence, the inputs to *F(NET)* should be restricted to ±4.844. Section 5.2.2.3 described the result range limiter unit within the PE, which can easily be programmed with these limits in order to accomplish this range restriction on the calculation of the *NET* input to the activation function.

5.2.3.4 Activation Function Approximation

As the inputs to the *F(NET)* function have a fixed range of ±4.844 then they can be translated directly to the range 0...1 required by the Bézier curve expressions. By taking the result on the curve at these limits as the start and end point on the Bézier curve then a first approximation to the curve can be evaluated. The control points for this initial approximation are shown in table 5.9. The maximum error for any point on the curve is 1/210, which is smaller than the minimum representable value within the systolic array. Because of this low error the given curve can be deemed to be a good approximation of the activation function *F(NET)*.

Table 5.9
Initial Bézier Control Points

Control	X-Pos	Y-Pos
P_0	0.0000	0.0078
P_1	0.6000	0.0000
P_2	0.4000	1.0000
P_3	1.0000	0.9922

The data that is to be supplied to the function approximation must be translated from the range ±4.844 to the range 0...1 along the curve. However, this cannot be done perfectly accurately, as the x-coordinate represents points on the linear x-axis, whilst the required t-pos values are positions actually on the polynomial curve a certain percentage between the control points P_0 and P_3; a t-pos value of 1.0 represents the final point on the curve at the P_3 control point. This difference in scaling, along a 20 segment curve, is shown in figure 5.9.

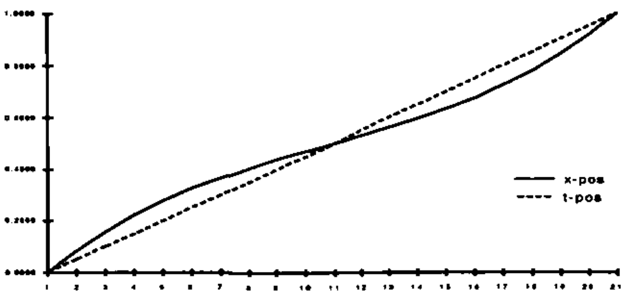


Figure 5.9 X-pos and t-pos Relationship

Once the data has been simply scaled from ±4.844 to 0...1 it has to be squashed so that it lies closer to the required t-pos value. This squashing function must be done in a single pass of the processor array, as it is a very common operation in the backpropagation learning algorithm and is likely to be executed fairly frequently. The expressions

$$\begin{aligned} \text{mid.dist} &= x - \tfrac{1}{2} \\ \text{end.dist} &= \tfrac{1}{2} - \text{ABS}\left(\text{mid.dist}\right) \\ t.\text{pos} &= x + \text{mid.dist} * \text{end.dist} \end{aligned} \tag{5.7}$$

manage to scale the x-values in this manner. However, values for $x \to 0$ are not decreased by enough and $x \to 1$ are not increased by enough - the maximum error for translation using this method is approximately 1/54. Because of this a 'magic' scaling factor must be introduced, so that the correction factor applied to each x-value is increased in magnitude. By modifying the squashing expressions to

$$\begin{aligned} \text{magic} &= 1.21 \\ t.\text{pos} &= x + \text{magic} * \text{mid.dist} * \text{end.dist} \end{aligned} \tag{5.8}$$

we can achieve a maximum translation error of approximately 1/121.

Table 5.10
Final Bézier Control Points

Control	X-Pos	Y-Pos
P_0	0.0000	0.0078
P_1	0.6000	-0.0159
P_2	0.4000	1.0159
P_3	1.0000	0.9922

Using these modified x-values with the previously discussed Bézier curve approximation, using the control points in table 5.9, we get a maximum approximation error of 1/81. By moving the control points slightly, to those shown in table 5.10, we reduce the maximum error to just 1/118: approximately 10% of the values are in error, with the error being just the least significant bit of the data.

For a fast method of approximating the *F(NET)* activation function these errors are acceptable, and much more accurate than any linear approximation method. The algorithm can be implemented directly on the systolic array without the need for any additional circuitry, and can also be executed in a single pass of the array. Although slightly less accurate than some of the existing Taylor-series based approximations it has the benefit of being implementable at no extra cost in terms of hardware.

5.3 Hardware Design and Implementation

5.3.1 Miscellaneous Circuits

5.3.1.1 Signal Multiplexors

There are several occasions where a functional unit in the PE needs to choose between a number of different input values, usually one from two or one from four. Figure 5.10 shows schematic diagrams for both of these units, which are described in more detail in sections A.1.1 and A.1.2 respectively.

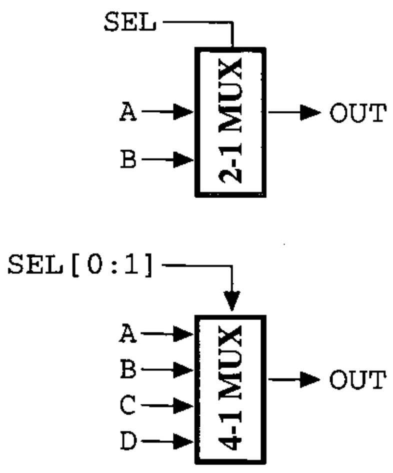


Figure 5.10 Multiplexor Schematics

The units work by selecting between the signals using the input control signal SEL or SEL[0:1]. All inputs are connected to the single output line OUT, with a transmission gate assigned to each input. The control signal(s) enable just one of the transmission gates in the multiplexor, thus driving just a single input onto the output and blocking all other input signals. The control signal logic for the two multiplexors is shown in table 5.11.

Table 5.11 Multiplexor Control

SEL	OUT
0	A
1	B

SEL[2]	OUT
00	A
01	B
10	C
11	D

5.3.1.2 ABS Control Unit

The multiplier unit is used to provide an arithmetic ABS function, by driving ±1.0 onto the second multiplier input in order to ensure that after multiplication with the first input the result is of a specified sign. The result can be forced to be either positive or negative in this fashion. Figure 5.11 shows the schematic diagram for this unit, which is described in more detail in section A.4.1.

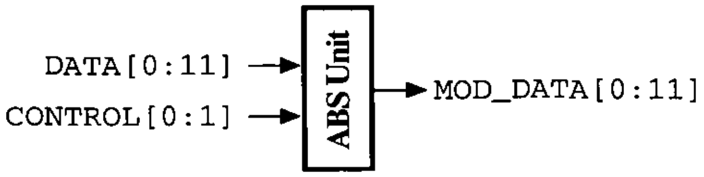


Figure 5.11 ABS Unit Schematic

The ABS unit does not carry out the ABS calculation, rather it ensures that the multiplier receives the correct inputs. Assertion of CONTROL[1] activates the unit. CONTROL[0] and the sign of the input DATA (which lies on DATA[11]) are used to determine the output of the unit, which is either +1.0 or -1.0. If the sign of the input DATA is already of the required sign then the output is +1.0, implying that the multiplier does not alter the sign of its other input. If the sign of the input DATA is incorrect then the output is -1.0, implying that the multiplier alters the sign of its other input. The control signal logic for the ABS control unit is shown in table 5.12.

Table 5.12 ABS Unit Control

DATA[11]	CONTROL[0]	MOD_DATA
0	0	-1.0
0	1	+1.0
1	0	+1.0
1	1	-1.0

5.3.1.3 Majority Function

Certain functions on the PE require calculation of some form of majority function, whereby the assertion of at least two from three

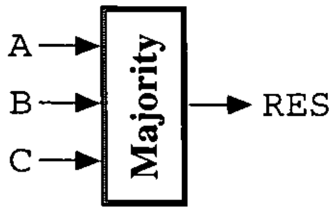


Figure 5.12 Majority Function Schematic

inputs results in a positive input. Figure 5.12 shows the schematic diagram for this unit, which is described in more detail in section A.1.6.

The unit itself is fairly simple, consisting of a few combinational logic gates. The control signal logic for the majority function is shown in table 5.13.

Table 5.13 Majority Function Control

A	B	C	RES
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

5.3.1.4 Result Range Limiter

The output of the PE can be made to be restricted within a certain range, which can be specified for each individual PE within the processor array. These ranges can be programmed by any algorithm that is currently being processed. It works by using a set of parallel comparators, which compare the result output for the PE with the pre-programmed ranges, and replacing the result with the relevant range limit if the result lies outside this range.

Figure 5.13 shows a schematic diagram for this unit, which is described in more detail in section A.4.2. The unit operates in two modes; program and limit. In program mode the limits are programmed by using a combination of WRITE_OP and ADDR[0:3], with ADDR of [0011] resulting in the maximum limit being stored and [0000] in the minimum limit being stored - these limits should be present on NS[0:11].

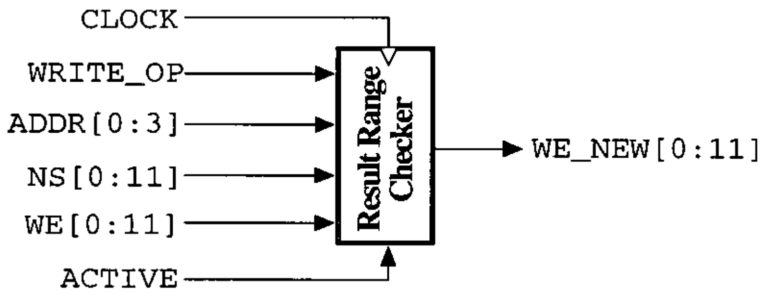


Figure 5.13 Result Range Limiter Schematic

Table 5.14 Range Limiter Multiplexor Control

CTRL[1]	CTRL[1]	WE_NEW
0	0	WE
0	1	MIN
1	0	MAX
1	1	WE

In limit mode, whereby ACTIVE is asserted, the input on WE[0:11], which is the result of any instruction processed on the PE, is compared with the two limits currently stored within the unit. A set of multiplexors are used to select the correct value to place on WE_NEW depending on the results of the two comparisons. If no limit was exceeded (or both were, due to incorrect programming of the limiting values) then the original WE is passed on to WE_NEW, otherwise the limiting value that was exceeded is passed on to WE_NEW. The two comparators produce an internal signal CTRL[0:1], with CTRL[0] being the result of the lower-limit comparator. The control signal logic for the multiplexed output selected is shown in table 5.14.

The actual comparators themselves are more complex. They are parallel in nature, using a bit-slice technique whereby only a single instance of the comparator needs to be designed, with an n-bit comparator being just n cascaded instances of a 1-bit comparator. The 1-bit and n-bit comparators are described in more detail in sections A.4.2.2 and A.4.2.3 respectively. The single bit comparator schematic is shown in figure 5.14.

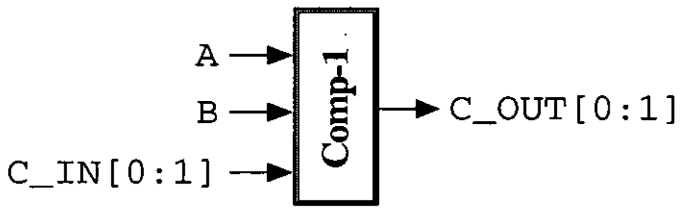


Figure 5.14 1-Bit Comparator Schematic

The comparator takes in two signals, A and B, and a pair of control signals C_IN[0:1]. This control input indicates whether or not *á priòri* information exists as to the relative sizes of A and B, as indicated in table 5.15. By using this information, and the values of A and B, the output control signals C_OUT[0:1] indicate the relative sizes of A and B; the implications of C_OUT are identical to those of C_IN. For the comparison of two 1-bit numbers, or for the comparison of the least significant bits of larger numbers, the C_IN value is 10, indicating that the A and B inputs are assumed to be equal until proved otherwise.

Table 5.15 Comparator Input Control Values

Cond	C_IN[1]	C_IN[0]
A < B	1	1
A = B	1	0
A > B	0	0

The transfer function from C_IN to C_OUT is easily realised, and table 5.16 shows the relevant derivations. It clearly shows that if the inputs for A and B are identical then the control inputs are passed directly to the outputs. Hence, for a 2-bit number, the C_IN for the most significant bit comparison is the C_OUT for the least significant bit comparison.

Table 5.16 C_OUT Transfer Function

A	B	C_OUT[1]	C_OUT[0]
0	0	C_IN[1]	C_IN[0]
0	1	B	B
1	0	B	B
1	1	C_IN[1]	C_IN[0]

If a comparison reveals an equality then the comparison result from the previous bit comparison is passed on. For example, comparing 0110 with 1100 leads to a chain of C_OUT values from each step of the comparison operation, as shown in table 5.17. It shows that although it is the most significant bits that have the most effect on the result, if the most significant bits are equal then the results of comparisons with bits of lower significance have an effect on the final result.

Table 5.17 Comparison Example

A	B	C_OUT
...0	...0	A = B
..10	..00	A > B
.110	.100	A > B
0110	1100	A < B

The comparators cannot handle the comparison of sign bits, as taken at face value the sign bit value for negative numbers (1) is larger than that for positive numbers (0). Hence, comparison of the sign bits needs to be done separately. However, if the sign bits are different then the result of the comparison of the entirety of A and B can be deduced from the result of the comparison of the sign bits.

5.3.1.5 General Purpose Comparator

The general purpose comparator provides the facility to carry out comparison and equality functions on any two different internal data values, and is based on the comparison unit described in section 5.3.1.4. The equality function returns either 1.0 or 0.0, depending on whether or not the inputs are equal. The comparison function has two sub-functions: maximum and minimum. Given two inputs the unit can return either the maximum value or the minimum value. This unit is described in more detail in section A.4.3.

5.3.2 Register Units

5.3.2.1 Flip-Flop Registers

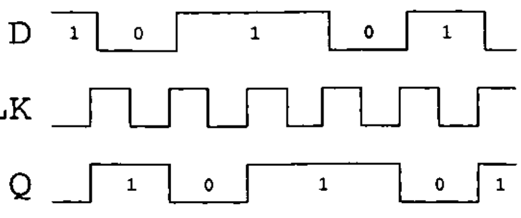


Figure 5.15 Flip-Flop Timing Diagram

Flip-flop registers store data on the inputs on the transition of the clock from one period to the next; changes to the inputs outside of this transition time have no effect on the value stored within the flip-flop. This type of register is used for the PE input registers, as described in section A.2.1. In these registers the data present on the four PE inputs upon the transition from a negative to positive clock (on the rising edge) is stored and used within the PE throughout the clock cycle, regardless of any changes to the inputs. The timing relationship between the input D, the output Q and the clock period CLK is shown in figure 5.15.

5.3.2.2 Half-Latch Registers

Half-latch registers store data on their inputs during the entirety of a clock period, normally when an additional LOAD input has also been asserted. Changing the input during the high clock edge also changes the value stored, but does not affect the output of the register until the following clock period. This type of register is used for the PE output registers and the internal register block, as described in sections A.2.2 and A.2.3.1, where the values on the inputs are valid up until the transition to the next clock cycle. Hence, values on the input are stored on the negative period of the clock cycle, then made available at the register outputs throughout the next clock cycle.

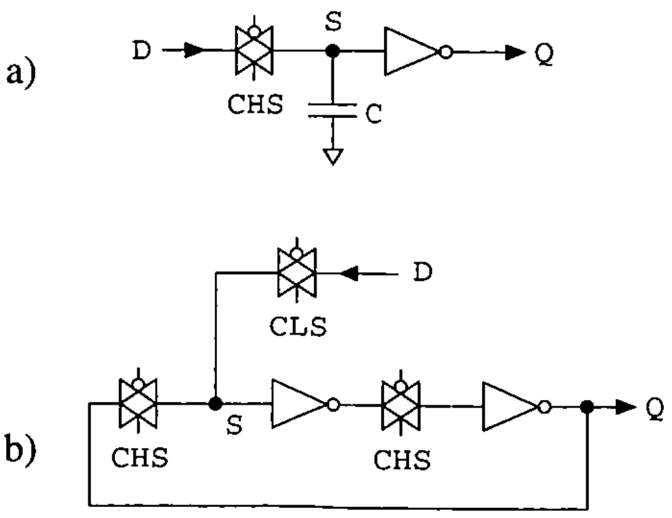


Figure 5.16 Latch-Based Register

The basic half-latch circuit is shown in figure 5.16a, which shows a simple transmission gate, inverter and capacitor. When the clock is high and stable (CHS) the value on D is transferred to S. This is then inverted and output at Q at all times. The value S is stored at C for a reasonable length of time, but will begin to dissipate beyond acceptable values after approximately 100ns [WeEs88][5]. Cascading two half-latches together creates a self-restoring 1-bit temporary storage area, and this is shown in figure 5.16b. By driving in a new data signal when the half-latch transmission gates are closed, which is when the clock is low and stable (CLS), then a new value can be stored on S. This value is not propagated to the output Q until the next clock cycle, as this action requires the clock to be high and stable. Note, CLS is often combined with a separate LOAD signal, so the value stored within the register can remain unaltered indefinitely.

5.3.3 Instruction Set Memory

5.3.3.1 RAM/ROM Instruction Store

The RAM/ROM section of the instruction set memory is a fairly standard piece of memory. It contains 12 words of 22-bit RAM and 4 words of 22-bit ROM. All words can be read from the memory and the contents of the RAM can be re-programmed at any time. The circuitry for all sections of the instruction store can be found in section A.3.4.

A schematic of the instruction store is shown in figure 5.17. It works like any standard RAM unit, in that it contains a number of static RAM or

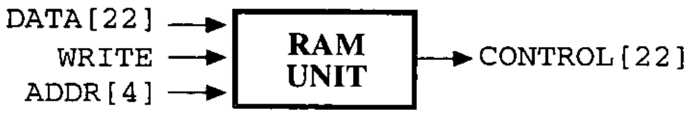


Figure 5.17 Instruction Set Store Schematic

ROM cells. A 4-bit address is placed on ADDR and the WRITE flag is set to indicate that a read or write operation is required. On a read operation the values from the specified word in memory is driven onto the CONTROL output. On a write operation the values on DATA are stored in the specified word in memory, although if this word is one of those implemented in ROM rather than RAM then the write operation has no discernable effect.

[5] When using a 5v system - as the voltage is scaled down, along with the width of the aluminium tracks on the device, then so does the time taken for a driven signal to dissipate beyond usable levels

There are just three operations possible in the instruction set memory unit, which are summarised in table 5.18. It shows that a read operation is valid for any memory address, but only valid on a write operation for a limited range of memory addresses.

Table 5.18 Instruction Set Store Operations

WRITE	ADDR	Effect
0	xx	Values at (ADDR) → CONTROL
1	0000-0011	No effect - invalid address range
1	0100-1111	Values on DATA → (ADDR)

5.3.3.2 Read/Write Memory Controller

In normal operational circumstances the memory is read from at all times. When a [PROG] operation is received then on the next clock cycle the memory is written to. By generating the read-write flag based on the current operation, and then delaying it for a clock cycle using a pair of half-latches, it will arrive at the instruction set store at the correct time. This circuit is described further in section A.3.1, which also shows that upon receipt of a system reset signal the read-write signal is forced to indicate a read operation for the current and subsequent cycles. A schematic of the circuit block is shown in figure 5.18.

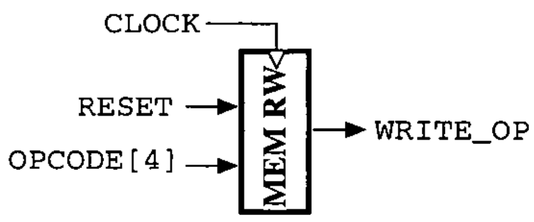


Figure 5.18 Read/Write Schematic

5.3.3.3 Instruction Lock/Unlock Unit

The lock circuitry within the instruction set memory is used to lock an opcode into a PE so that it is processed on every clock cycle regardless of any other opcode input. There are a number of possible lock states, which are summarised in table 5.18. Note that each PE has a row number hard-wired into it, and a [LOCK] operation is only considered to be valid within a PE if the data on NS matches this hard-wired value. This allows individual PE's anywhere in the processor array to be locked and unlocked by a processing algorithm. Each lock state implies the following:

Table 5.19 Possible PE Lock States

RESET	Opcode	State	Next State
1	[any]	[any]	Unlock
0	[LOCK]	Unlock	StoreOp
0	[LOCK]	Lock	Unlock
0	[LOCK]	StoreOp	Unlock
0	[other]	Unlock	Unlock
0	[other]	Lock	Lock
0	[other]	StoreOp	Lock

Unlock	PE input opcode on NS datapath processed normally
StoreOp	As *Unlock*, except that the current opcode is also stored in a local register
Lock	The opcode stored in the local register during the *StoreOp* state is executed until another [LOCK] opcode is input to the PE

When a PE has an opcode locked in to its local register the current input opcode on the NS datapath is not processed. It is, however, passed through the PE to neighbouring PE's as if it had been processed.

This allows for powerful systolic algorithms to be designed, but with individual PE's in the processor array carrying out some constant task oblivious to the operations going around it.

A schematic of the lock circuitry is shown in figure 5.19 and more detailed circuits are given in section A.3.2. Depending on the lock status the multiplexor selects either the OPCODE data or the register data as the instruction store memory address. Data is only written into the register during a *StoreOp* state. The state machine unit handles the transition from state to state, and the value on IP is used to indicate which row of the processor array any [LOCK] operation is destined for.

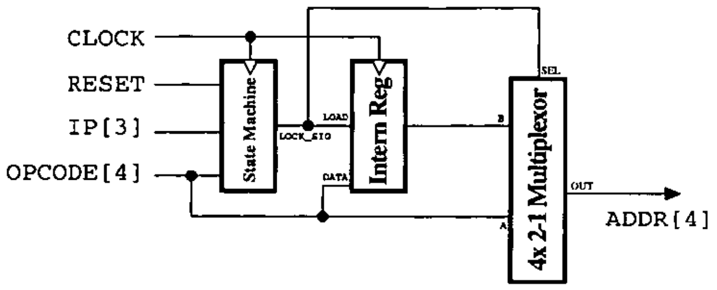


Figure 5.19 Opcode Lock Schematic

5.3.4 Addition Unit

5.3.4.1 Addition Unit Overview

The entire adder unit takes in two 12-bit numbers, their associated integrity flags and a control signal to indicate whether the operation to carry out is addition or subtraction. The output from the adder is the result of the addition or subtraction operation, along with an associated integrity flag to

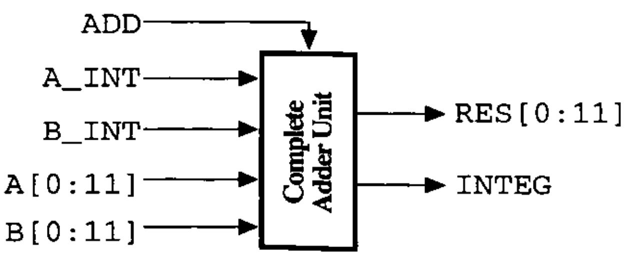


Figure 5.20 Complete Adder Unit Schematic

indicate the correctness of the result. This unit is described in section A.4.4, and a schematic of the entire unit is shown in figure 5.20.

5.3.4.2 Radix-4 Adder

A basic half-adder unit adds two 1-bit values, taking into account a carry-in signal, and produces two outputs, a sum and a carry-out. A simple *n*-bit adder cascades many such devices together, with the carry-out from one unit being the carry-in for the next. Although such an adder works perfectly well it is fairly slow in operation; the delay from the generation of the first carry-out to the final carry-out is very long, being of *O(n)*. Such a basic adder unit is unacceptable for use in the systolic processor array.

A standard radix-4 adder speeds up the process in two ways. Firstly, this method prioritises the calculation of the carry-out over the calculation of the sum, which reduces the overall carry propagation time within the adder. Secondly, by adding a pair of 2-bit values at once the time taken to produce the adder sum is approximately halved. The overall time requirement of a radix-4 adder is $O(\sqrt{n})$. The radix-4 adder is described further in section A.4.3.1, and a schematic of the circuit is shown in figure 5.21.

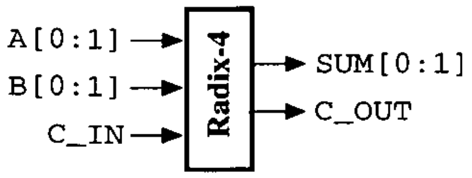


Figure 5.21 Radix-4 Adder Schematic

5.3.4.3 Carry Select Adder

A method of implementing a fast adder is to use a carry-select adder [Uya84]. This increases the silicon area required for the adder, but drastically reduces the number of gate delays required to go from initial carry-in to final carry-out. It uses blocks of 4-bit adders,

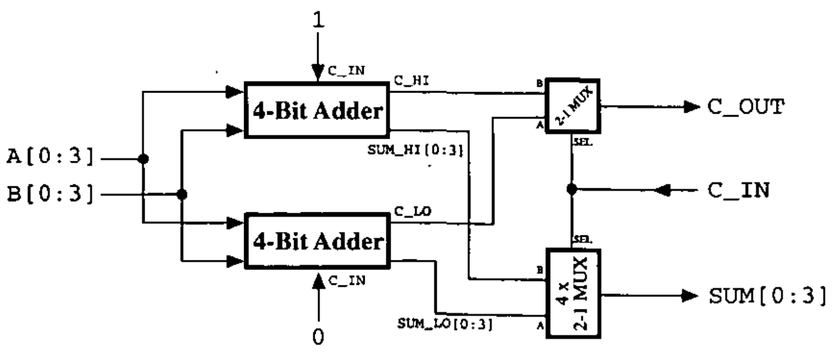


Figure 5.22 Carry Select Adder Block

which are simply a pair of cascaded radix-4 adders. After evaluation of the first 4-bit addition the carry-out is used as a carry-in to the next 4-bit block. However, this next block is implemented twice, each with a different hard-wired carry-in value. The carry-out of this first block then *selects* which of the next two blocks has their sums and carry-outs propagated to the next pair of 4-bit adders. This scheme is shown in figure 5.22.

The full 12-bit adder consists of a single 4-bit adder (for bits 0...3) and two 4-bit carry select blocks (for bits 4...7 and 8...11). Each adder and carry select block generates their sum and carry values in parallel, all being available after 4 gate delays. Each block requires an additional gate delay in order to propagate the carry values to the next block, which selects the correct sum outputs for one of the bit ranges. As there are two carry select blocks this carry propagation requires just two additional gate delays, resulting in a total of just six gate delays for the complete 12-bit addition operation.

5.3.4.4 Subtractor Control

It is possible for the main adder unit to carry out the operation A-B instead of A+B, which is a featured often utilised in some of the neural training algorithm stages. It can be done by simply inverting the sign of the adder input B. This is achieved using standard two's complement techniques by inverting all of the bits of the value B and then adding one to the result. However, instead of adding one to the result (which requires an adder) it is sufficient to set the initial carry-in to the addition operation to one instead of zero, which has the effect of adding one to the result. This control unit are described further in section A.4.4.4, and a schematic of the circuit is shown in figure 5.23.

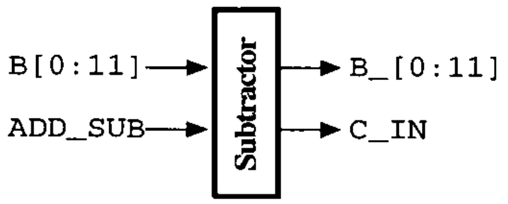


Figure 5.23 Subtraction Control Schematic

5.3.4.5 Overflow/Underflow Unit

Addition overflow/underflow has to be handled within the adder unit. As well as the simple arithmetic errors the unit must also cope with the integrity status of the original inputs to the adder. The circuits used to implement this unit are described further in section A.4.4.5, and a schematic of this is shown in figure 5.23.

If the integrity of any adder input has failed, as indicated by A_INT and B_INT, then the adder output is automatically set to the value of the failed input[6] and the integrity state is set accordingly; the result of the addition RES_IN is simply discarded.

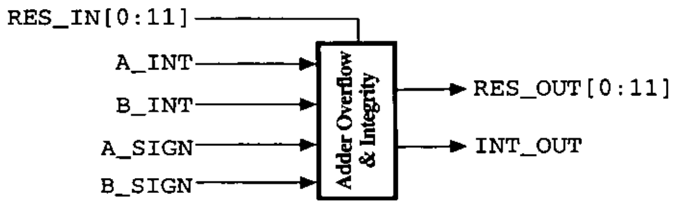


Figure 5.24 Adder Overflow Schematic

The overflow/underflow state of the addition can be inferred from the signs of the input, A_SIGN and B_SIGN, and the sign of the result, RES_IN[11]. Possible addition overflow results are shown in table 5.20. If the two adder input values are of different sign then it is simply not possible for the addition to overflow or underflow; overflow and underflow only occurs in the input signs are identical but the adder result sign is different. In these cases the result is replaced by the maximum or minimum representable value and the integrity flag is set to indicate failure.

Table 5.20
Overflow Possibilities

A	B	RES	OK ?
+ve	+ve	+ve	√
+ve	+ve	-ve	x
+ve	-ve	+ve	√
+ve	-ve	-ve	√
-ve	+ve	+ve	√
-ve	+ve	-ve	√
-ve	-ve	+ve	x
-ve	-ve	-ve	√

5.3.5 Multiplier Unit

5.3.5.1 Multiplier Unit Overview

An efficient implementation of a parallel multiplier normally some modified version of the standard Booth algorithm [Boot51]. Such a circuit tends to occupy the largest block on a VLSI device, especially if the design cannot include any pipelining due to the requirement of the multiplication operation being executed in a single machine cycle. The selection of the Booth algorithm was made because it is fairly simple, which means that a good VHDL synthesis optimisation tool is able to reduce a behaviourial model VHDL routine into a fairly efficient

[6] This can be deduced from the sign of the value, allowing the setting of the maximum or minimum representable value

silicon design. However, due to the sheer physical size of a 12-bit parallel multiplier the example circuit shown in section A.4.5.1 gives a 12-bit VHDL code module but only shows an optimised 2-bit circuit.

The entire multiplier unit takes in two 12-bit numbers, their associated integrity flags and a clock signal to indicate to begin the multiplication operation. The output from the multiplier is the result of the multiplication operation, along with an associated integrity flag to indicate the correctness of the result. This unit is described in section A.4.4, and a schematic of the entire unit is shown in figure 5.25.

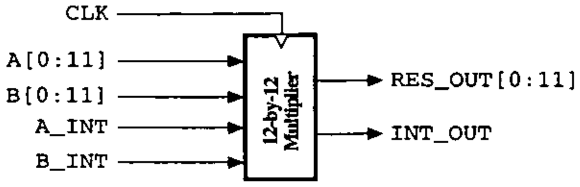


Figure 5.25 Complete Multiplier Unit Schematic

5.3.5.2 Booth Multiplier Scheme

A graphical synopsis of the Booth multiplier scheme [NaJo97] is shown in figure 5.26. The operations of the multiplier can be summarised as follows:

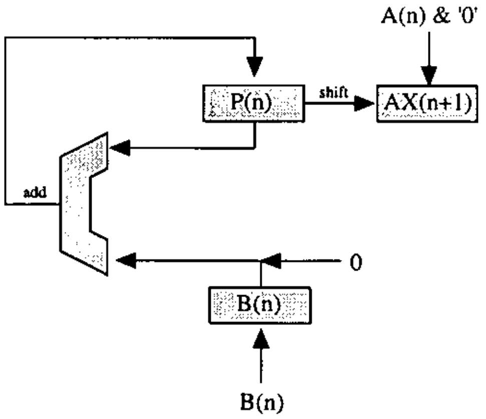


Figure 5.26 Booth Multiplier Functional Overview

i) Clear P, load inputs A and B into the registers AX[n+1..1] and B, setting AX[0] to logic-0

ii) Examine AX[1] and AX[0] and carry out an operation based upon it: on 00 or 11 do nothing (add 0 to P), on 01 perform P=P+B, on 10 perform P=P-B. Ignore any errors in the addition/subtraction operation

iii) Arithmetically shift right the concatenation of P and AX (thus preserving the sign of P)

iv) Repeat from step (ii) for each bit in the original A input

v) The final result can be extracted from both P and AX - concatenate P and AX together and then discard the most significant and least significant bits of the combined value.

The basic 12-bit multiplier unit, which does not include any integrity checking of the inputs, is shown in schematic form in figure 5.27. It shows two 12-bit inputs A and B, a clocking input CLK and a 23-bit output RESULT.

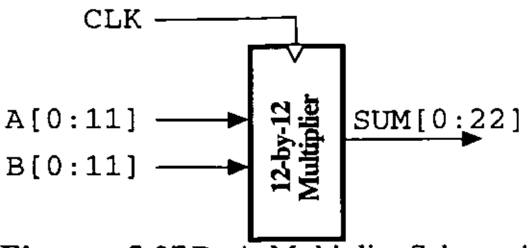


Figure 5.27 Basic Multiplier Schemati

5.3.5.3 Multiplier Integrity Checking

As is standard in arithmetic theory a parallel 12-bit multiplier will produce a 23-bit result in raw form; it is up to the implementation to take the result and produce a usable value from it. The internal number format used within the systolic array architecture is given in section 5.1.4. The difference in format between these standard internal numbers and the results generated by the multiplier is shown in figure 5.28. This shows that the effective width of both the integer and fractional part of the number is doubled in size by the multiplier unit, with the shaded bands indicating where equivalent number representations lie within both the internal format and the multiplier output format (the sign bit is always equivalent).

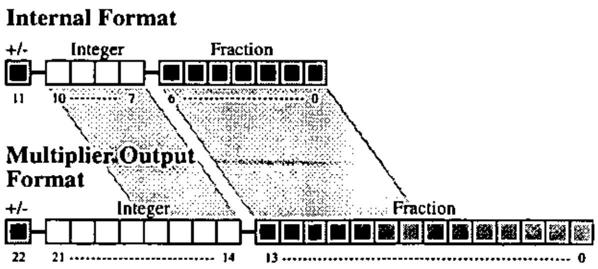


Figure 5.28 Relationship Between Internal and Multiplier Data Formats

The multiplier output must be restricted to the ranges laid down in section 5.1.4. The fractional part of the result can be handled very simply - only the top seven bits of the fractional part of the multiplier result is used, with the rest being discarded. For the integer part a check on the top four bits is made; if any of these are logic-0 (if the result is positive) of logic-1 (if the result is negative) then the result is out of the internal representable range. In this case the result is modified to either the minimum or maximum representable value and the integrity flag associated with the output is set to indicate the overflow error.

6 SYSTOLIC NEURAL ALGORITHMS

INTRODUCTION

This section describes the three neural network training algorithms that have been implemented in the systolic array processor. It describes the individual stages of each algorithm, along with any requirements for constant data values within the PE registers, as well as details of any instructions that are 'locked' into a PE for perpetual execution during a single stage of an algorithm. Dataflow diagrams within the systolic array for each stage of each algorithm, showing the implications of each calculation and the implied direction of the execution flow, as well as showing timing information for each stage of each algorithm.

6.1 Backpropagation Learning Algorithm

6.1.1 Introduction

6.1.1.1 Backpropagation Instruction Set

The backpropagation algorithm requires all 12 available instruction slots in order to be implemented on the systolic array processor. All instructions use the default result truncation range of the system maximum and minimum numbers, except for the [MVM] operation on the right-most column of PE's, which uses the range ±4.844. Most of the instructions do not have an obvious mnemonic, so they have them based on the arithmetic unit using both external inputs, the arithmetic unit concerned and the other input to the second arithmetic input (if there is one). Hence, the mnemonic [NW.ADD.R] implies the operation (NS+WE)*REGISTER, although the redirection of the final result is not implied in the mnemonic.

Table 6.1 Backpropagation Instruction Set

Opcode	Mnemonic	Description
[0100]	[MVM] *col 1*	NS * REGISTER → WE'
	[MVM] *other*	WE + (NS * REGISTER) → WE'
[0101]	[V.MVM] *row 1*	WE * REGISTER → WE' & NS'
	[V.MVM] *other*	NS + (WE * REGISTER) → WE' & NS'
[0110]	[N.NABS.R]	NEG.ABS(NS) + REGISTER → WE'
[0111]	[NR.ADD]	NS + REGISTER → WE'
[1000]	[NW.MUL]	NS * WE → WE' & NS'
[1001]	[NW.ADD.R]	(NS + WE) * REGISTER → WE' & NS'
[1010]	[NW.MUL.R]	(NS * WE) + REGISTER → WE' & NS'
[1011]	[NW.MUL.R2]	(NS * WE) + REGISTER → WE' & REGISTER'
[1100]	[WR.MUL]	WE * REGISTER → WE'
[1101]	[WR.ADD.N]	(WE + REGISTER) * NS → WE' & NS'
[1110]	[LOAD]	WE → WE' & REGISTER'
[1111]	[MAT-ADD]	REGISTER + WE → WE' & REGISTER'

Table 6.1 shows the complete programmable instruction set for the backpropagation neural learning algorithm. Note that some instructions, such as the matrix-vector multiplication operation [MVM], have multiple definitions depending upon which row or column the PE is in the systolic array. This feature is used where results of some operation have to be accumulated across a row or column; this ensures that the first PE in the row or column begins the accumulation process, with other PE's in the row or column picking up previous partial results and including them in their own internal calculations.

6.1.1.2 Algorithm Summary

The learning algorithm can be split into three distinct sections: forward pass, reverse pass and weight update. The forward pass concerns itself with generating the output values for each neuron in the network for a particular input pattern. It contains two components:

i) Generate NET Value for Neurons

Carries out a simple matrix-vector multiplication operation using the weight matrix and an input vector, producing a series of neuron *NET* values (as described by equation 3.4) *ii) Bézier Approximation*

Converts individual *OUT* values to a Bézier t-score (as described by equations 5.7 and 5.8), then approximates the neural activation function for the backpropagation learning algorithm using the converted *OUT* value (as described by equation 5.6)

The reverse pass modifies the weights in the neurons in response to the difference between the actual neuron outputs and the desired neuron outputs. It contains three components:

i) Output Layer: F'(NET), δ and Δw Calculation

Calculates weight updates for each output layer neuron weight and input-vector combination (as described by equations 3.3, 3.5 and 3.7)

ii) Hidden Layer: False Target for δ Generation

Calculates a false target vector for use in the training of hidden layer neurons (as described within equation 3.6)

iii) Hidden Layer: F'(NET), δ and Δw Calculation

Calculates weight updates for each hidden layer neuron weight and input-vector combination (as described by equations 3.3, 3.6 and 3.7)

The weight-update pass updates the weights held in the PE internal register blocks with the adjustment values that have been calculated; this is based on equation 3.8.

6.1.2 Forward Pass Components

6.1.2.1 Generate *NET* Value for Neurons

This operation is a simple matrix-vector multiplication. The correct register set for the network layer being processed is selected and all processors are unlocked. An input vector is presented to the array and each PE performs the operation [MVM], with PE's in the first column of the array starting off the accumulation process. The internal values, such as $W_{2,3}$, are the contents of the internal registers representing the layer of the network currently being processed. The subscript denotes neuron/input, with PE's in a single row representing a single neuron. Each row of the array, each of which represents an individual neuron in the network layer, generates a single *NET* value for the given input vector. The algorithm process is shown in figure 6.1.

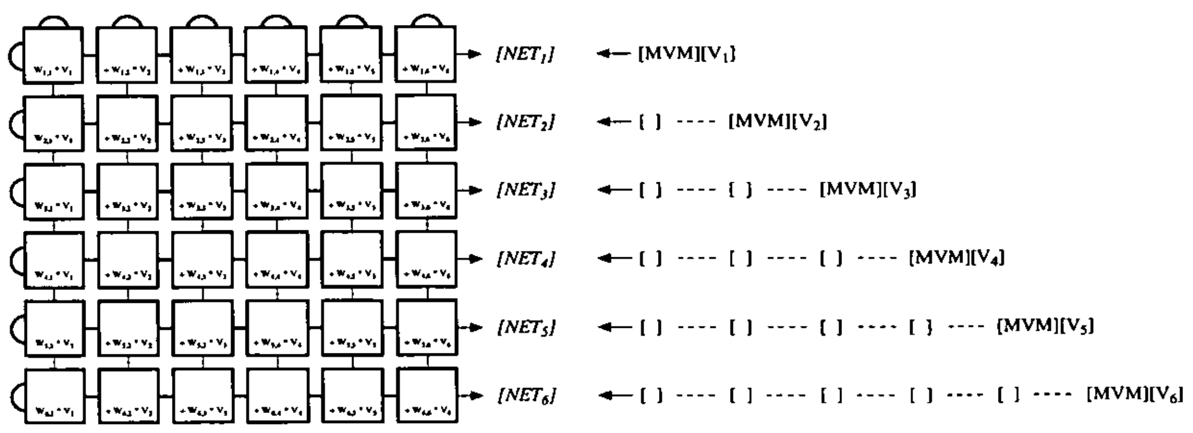

[NET1]
[NET2]
[NET3]
[NET4]
[NET5]
[NET6]
[MVM][V1]
[ ] ---- [MVM][V2]
[ ] ---- [ ] ---- [MVM][V3]
[ ] ---- [ ] ---- [ ] ---- [MVM][V4]
[ ] ---- [ ] ---- [ ] ---- [ ] ---- [MVM][V5]
[ ] ---- [ ] ---- [ ] ---- [ ] ---- [ ] ---- [MVM][V6]

Figure 6.1 Algorithm for *NET* Value Calculation

The array is not always utilised at 100% efficiency for every algorithm[1]. Active PE's in the array have a highlighted border (although in figure 6.1 all PE's are active). Result redirections also have this highlight to indicate the flow of results; in figure 6.1 all WE datapaths carry a partial result value, but no NS datapaths do so. Partial results calculated within a PE are described with the PE, as are any locked instruction (although there are no locked instructions in figure 6.1).

Note that in figure 6.1 the input algorithm is entered in a staggered fashion. All algorithms work in this manner, as this allows results from PE's in one column to be used in PE's in the next column on the next clock cycle with the next instruction in the algorithm. Multiple algorithms can be entered into the array on successive clock cycles; the delays shown in figure 6.1 are present just to show the staggered nature of the algorithm input into the array.

The results generated by a single algorithm are staggered in an identical, but reversed, manner to the inputs, although figure 6.1 does not show this for the sake of clarity. PE's in the final column of the array will output a result on the clock cycle following the PE in the previous row; the first row will output its first result after 12 clock cycles, with results on successive instances of the same algorithm appearing on successive clock cycles. Hence, the generation of results from algorithms tend to have an initial delay, but are then generated on successive clock cycles.

[1] All algorithms shown throughout section 6 assume that the systolic array has a dimension of 6x6, and that the neural network being utilised on-chip has no layer larger than six

6.1.2.2 Bézier Approximation

This algorithm carries out two separate tasks: conversion of a neuron *NET* value to a Bézier t-score and then the Bézier approximation of *F(NET)* based upon this t-score. PE's in the first three rows in the array carry out the first operation (based upon equations 5.7 and 5.8), with the next two rows carrying out the approximation (based upon equation 5.6).

There is a large number of pre-loaded constant values and locked instructions present in this algorithm. All of the pre-loaded data required for this algorithm is shown in table 6.2. The entire algorithm, along with all dependent data and instructions, is shown in figure 6.2, with locked instructions and pre-loaded data indicated through a different typeface and a shaded background.

Table 6.2
Bézier Approximation Constants

Token	Value
α	1 / 9.6875
MAX-NUM	4.84375
a	$-3P_0 + 3P_1$
b	$-3P_0 - 6P_1 + 3P_2$
c	$-P_0 + 3P_1 - 3P_2 + P_3$
d	$P_1 - 1$

The conversion of a neuron *NET* value to a Bézier t-score shown in figure 6.2 does not use the nomenclature specified by equation 5.7. The value *u* is a modified version of *x*, scaled to the range 0.0...1.0 from ±4.844. The modified workings to the Bézier approximations are given in table 6.3.

Table 6.3
Bézier Approximation Workings

Figure 6.2	Equation 5.7
u	n/a
u'	mid.dist
u''	magic * mid.dist
(...)	end.dist

The final calculation of the modified t-score is carried out using the following equation:

$$\text{t.pos} = \text{u} + \text{u}'' \cdot \left(\tfrac{1}{2} - \text{ABS}(\text{u}')\right) \qquad (6.1)$$

Processing of the approximation within the array has no relation to the position of the neuron in the network that produced the *NET* value. The neural weight values are not used at all in this process, with the algorithms using the fourth (or alternate) register within each PE instead.

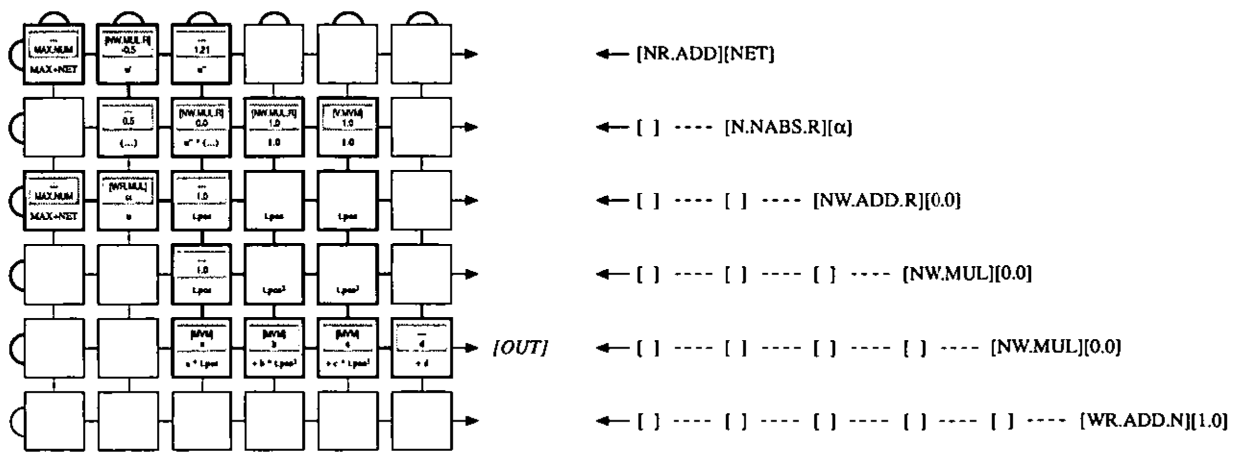

[OUT]
← [NR.ADD][NET]
← [ ] ---- [N.NABS.R][α]
← [ ] ---- [ ] ---- [NW.ADD.R][0.0]
← [ ] ---- [ ] ---- [ ] ---- [NW.MUL][0.0]
← [ ] ---- [ ] ---- [ ] ---- [ ] ---- [NW.MUL][0.0]
← [ ] ---- [ ] ---- [ ] ---- [ ] ---- [ ] ---- [WR.ADD.N][1.0]

Figure 6.2 Algorithm for Bézier *F(NET)* Approximation

Some PE's obviously create a result, such as that in PE (1,2) (column/row order), but as the value is not used anywhere within the array the PE border is not highlighted, indicating effective inactivity. If a PE output is not used then it is simply output into an inactive PE, such as the WE output for the third PE on the top row of the array, which is denoted PE(3,1).

6.1.3 Reverse Pass Components

6.1.3.1 Output Layer: F'(NET), δ and Δw Calculation

There are two different versions of the reverse pass algorithm: one for the output layer and one for the hidden layers. The difference in the two is in the calculation of the neuron δ value. The output layer version is simple, requiring a single calculation, whilst the hidden layer version is more complex, requiring a matrix-vector operation. The entire reverse pass, excluding the actual weight update process, can be completed in a single pass for the output layer neurons and in two passes for the hidden layer neurons. The three variables that have to be calculated for the output layer are

$$\begin{aligned} F'(\mathrm{NET}) &= \mathrm{OUT}(1-\mathrm{OUT}) \\ \delta &= F'(\mathrm{NET}) \cdot (\mathrm{Target}-\mathrm{OUT}) \\ \Delta w &= \eta \cdot \delta \cdot \mathrm{OUT}_{\mathrm{source}} \end{aligned} \tag{6.2}$$

where

η = training rate coefficient, typically in the range 0.01...1.00

Target = desired output for neuron / input-vector pair

Perhaps the most important of these three variables is δ, as the hidden layer neurons use the output layer neurons δ values in order to calculate their own δ values. This algorithm outputs both the weight adjustment with respect to the current neuron / input-vector pair, as well as outputting the δ value for the neuron. This is not dependent on any weights, so the δ is the same for all weights in a particular neuron and is repeatedly calculated when each weight adjustment within a neuron is calculated.

As batch processing is more efficient than pattern-by-pattern processing a single weight will have the adjustments calculated for each input pattern summed within the array. Once the summation is complete the Δw output will contain the sum of all modifications, which means that only a single update per weight is required regardless of the number of input patterns the neural network is attempting to learn or classify. In order to facilitate this the PE that sums the various Δw values for a particular weight, PE (4,3) in the array, initially loads its alternate register with 0.0. The adjustments are added to this alternate register and, when all adjustments have been calculated, the final Δw value for the weight in question is output from the array.

The accumulation PE will require 0.0 to be loaded in at the beginning of processing for every weight in every neuron. If momentum is required [Rumm86] [SeRo87] then PE (4,3) should be loaded with its own Δw from the previous training pass, scaled by the momentum coefficient α. This will offset the cumulative weight update by a value proportional to the previous weight update.

The entire algorithm, along with all dependent data and instructions, is shown in figure 6.3, with locked instructions and pre-loaded data also indicated. Row #2 of the array outputs the neuron δ value, which is the same for all weight/input-pattern combinations within a single neuron, and row #3 outputs the current accumulation of weight adjustments that has been calculated; once the last adjustment is accumulated then the output from this row is the total calculated adjustment.

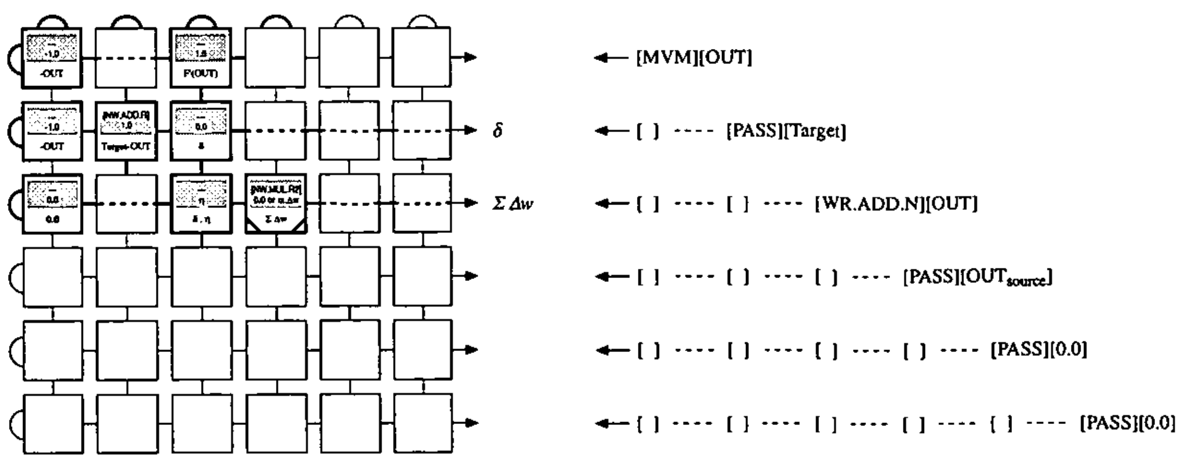

δ
ΣΔw
[MVM][OUT]
[ ] ---- [PASS][Target]
[ ] ---- [ ] ---- [WR.ADD.N][OUT]
[ ] ---- [ ] ---- [ ] ---- [PASS][OUT$_{source}$]
[ ] ---- [ ] ---- [ ] ---- [ ] ---- [PASS][0.0]
[ ] ---- [ ] ---- [ ] ---- [ ] ---- [ ] ---- [PASS][0.0]

Figure 6.3 Algorithm for Output Layer Weight Adjustments

As the [PASS] instruction causes no additional processing within a PE no outputs are ever indicated, except when a previous PE result is being passed through a number of PE's for use either later in the algorithm, as indicated by PE (2,1), or for eventual output from the array, as indicated by PE (3,2). An additional indication is required when an instruction causes the result to be stored in the currently active PE register, as indicated in PE (4,3).

6.1.3.2 Hidden Layer: False Target Calculation

Neurons in the hidden layers of the network have no specified target value to train for, so the value in equation 6.2 *(Target-OUT)* must have some equivalent value calculated before any weight adjustments can be ascertained. The standard method for the Backpropagation algorithm, as described in section 3.2.2.4, is to use the δ-values of the neurons in the following layer, along with the weight values that connect those neurons to the current hidden layer neuron. This is shown graphically in figure 6.4[2]. Once the δ-value is calculated the weight adjustments for the hidden layer neuron can be carried out in a similar fashion to the output layer neurons.

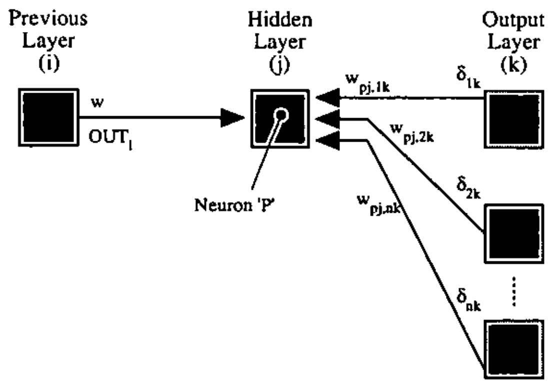


Figure 6.4 Connections for Hidden Layer Training

The calculation required for neuron *p* in hidden layer *j* is show in equation 6.3:

[2] Originally shown in section 3 as figure 3.8

$$\text{False.Target}_{p,j} = \sum_{q=1}^{n} \left(\delta_{q,k} \cdot w_{pj,qk}\right) \tag{6.3}$$

where

$\delta_{q,k}$ = δ-value for neuron q in hidden layer k

$w_{pj,qk}$ = weight connection neuron p in hidden layer j to neuron q in output layer k

Due to the design of the architecture of the array all weights from neuron p in hidden layer j to all neurons in output layer k lie in the same column of PE's and are in the PE internal register set associated with layer k. Hence, equation 6.3 can be calculated by switching in this register set and carrying out a simple vector product operation.

The entire algorithm, along with all dependent data and instructions, is shown in figure 6.5, indicating that no locked instructions or pre-loaded data is required.

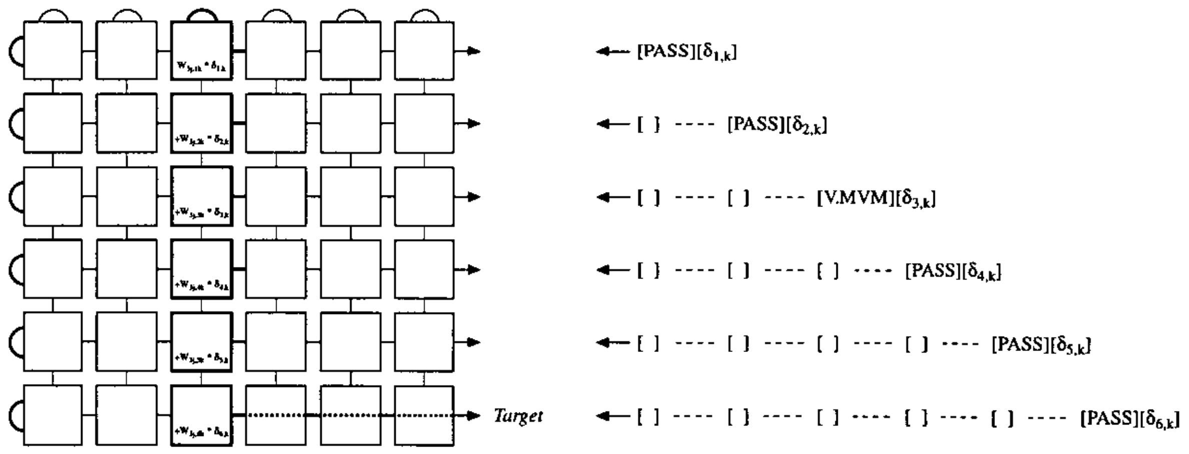

Target
[PASS][$\delta_{1,k}$]
[ ] ---- [PASS][$\delta_{2,k}$]
[ ] ---- [ ] ---- [V.MVM][$\delta_{3,k}$]
[ ] ---- [ ] ---- [ ] ---- [PASS][$\delta_{4,k}$]
[ ] ---- [ ] ---- [ ] ---- [ ] ---- [PASS][$\delta_{5,k}$]
[ ] ---- [ ] ---- [ ] ---- [ ] ---- [ ] ---- [PASS][$\delta_{6,k}$]

Figure 6.5 Algorithm for False Target Generation

Figure 6.5 shows the calculation of the false target for neuron #3 in hidden layer j. Output layer k is six neurons in size, so all rows of the array are used and the false target value is generated on row #6 of the array. If the output layer k consisted of only four neurons then only four rows of the array would have been used, with the fifth and sixth operations in the algorithm becoming `[PASS][0.0]`, and the false target would have been generated on row #4.

To calculate the false target for a different neuron in hidden layer *j* the algorithm is modified so that the [V.MVM] instruction lies on the same instruction row as the neuron; i.e. for neuron #3 in hidden layer *j* the [V.MVM] instruction appears as instruction 3 in the algorithm, thus utilising all weights in output layer *k* associated with that hidden layer neuron, all of which lie in column #3 of the array. All other operations in the algorithm are [PASS] operations, but the placement of the δ-values for the output layer *k* always remains as shown in figure 6.5.

One pass of this algorithm is made per neuron in hidden layer *j* using the same output layer *k* δ-values. The algorithm changes for each neuron only by the shifting of the [V.MVM] instruction, as well as resetting any instructions to [PASS] [0.0] as necessary due to a difference in sizes between layers *j* and *k*. As neuron δ-values are input-pattern independent this process only has to be carried out once per neuron in any of the hidden layers in the network.

6.1.3.3 Hidden Layer: F'(NET), δ and Δw Calculation

This process is virtually identical to the that used for neurons in the output layer, as described in section 6.1.3.1, except that all references to *(Target-OUT)* are replaced by *False.Target.*

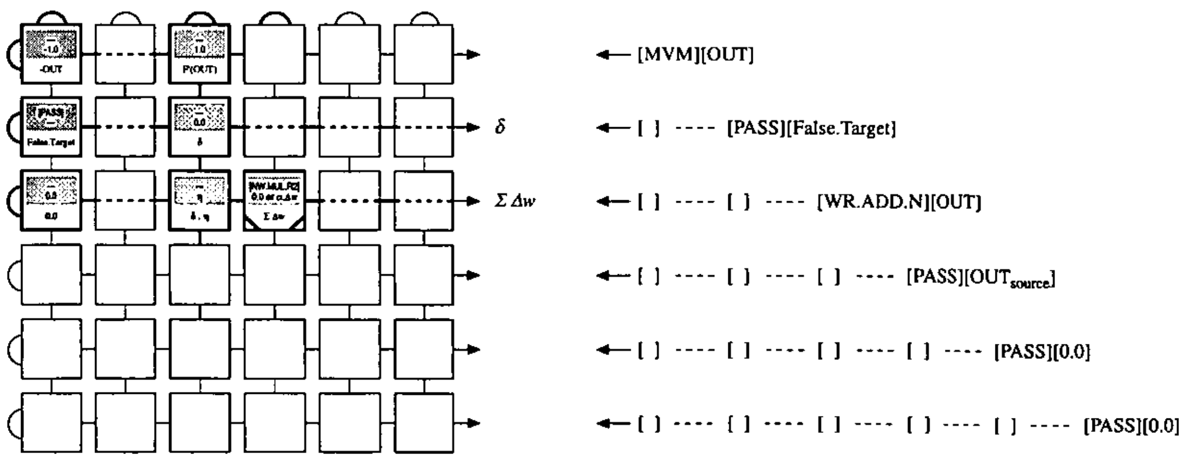
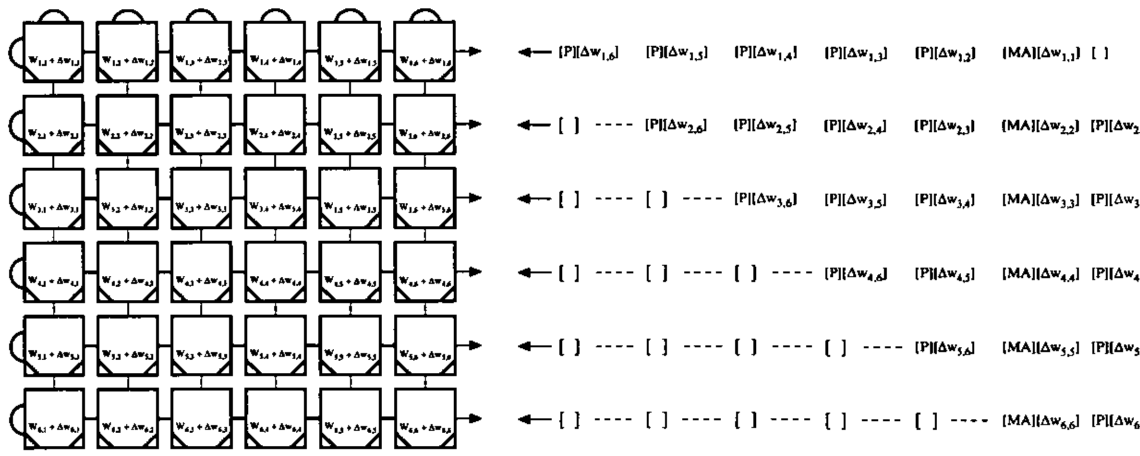

-OUT
F(OUT)
False.Target
δ
ΣΔw
[MVM][OUT]
[ ] ---- [PASS][False.Target]
[ ] ---- [ ] ---- [WR.ADD.N][OUT]
[ ] ---- [ ] ---- [ ] ---- [PASS][OUT$_{source}$]
[ ] ---- [ ] ---- [ ] ---- [ ] ---- [PASS][0.0]
[ ] ---- [ ] ---- [ ] ---- [ ] ---- [ ] ---- [PASS][0.0]

Figure 6.6 Algorithm for Hidden Layer Weight Adjustment

The entire algorithm, along with all dependent data and instructions, is shown in figure 6.6, with locked instructions and pre-loaded data also indicated; note that these are slightly different to those required for the output layer, as shown in figure 6.3. Row #2 of the array outputs the neuron δ value, which is the same for all weight/input-pattern combinations within a single neuron, and row #3 outputs the current accumulation of weight adjustments that has been calculated; once the last adjustment is accumulated then the output from this row is the total calculated adjustment.

6.1.4 Update Network Weights

The weights held in the register blocks within the PE's must be updated at the end of each training pass. As there are three registers set aside for neural network weights, one for each of the three possible layers, a [SWITCH] operation must precede any attempt to update the weights. Figure 6.7 shows the majority of the algorithm required to update the weights in a network layer consisting of six neurons, although for the sake of clarity the operations [PASS] and [MAT-ADD] have been truncated to [P] and [MA] respectively.



$w_{i,j}+\Delta w_{i,j}$
[P][$\Delta w_{1,6}$] [P][$\Delta w_{1,5}$] [P][$\Delta w_{1,4}$] [P][$\Delta w_{1,3}$] [P][$\Delta w_{1,2}$] [MA][$\Delta w_{1,1}$] [ ]
[ ] ---- [P][$\Delta w_{2,6}$] [P][$\Delta w_{2,5}$] [P][$\Delta w_{2,4}$] [P][$\Delta w_{2,3}$] [MA][$\Delta w_{2,2}$] [P][$\Delta w_{2,1}$]
[ ] ---- [ ] ---- [P][$\Delta w_{3,6}$] [P][$\Delta w_{3,5}$] [P][$\Delta w_{3,4}$] [MA][$\Delta w_{3,3}$] [P][$\Delta w_{3,2}$]
[ ] ---- [ ] ---- [ ] ---- [P][$\Delta w_{4,6}$] [P][$\Delta w_{4,5}$] [MA][$\Delta w_{4,4}$] [P][$\Delta w_{4,3}$]
[ ] ---- [ ] ---- [ ] ---- [ ] ---- [P][$\Delta w_{5,6}$] [MA][$\Delta w_{5,5}$] [P][$\Delta w_{5,4}$]
[ ] ---- [ ] ---- [ ] ---- [ ] ---- [ ] ---- [MA][$\Delta w_{6,6}$] [P][$\Delta w_{6,5}$]

Figure 6.7 Algorithm for Weight Update

The weight updates are fed into the array rotated 90° clockwise, each of which are contained within a [PASS] instruction. They are arranged so that all weight update values are present in the correct PE's in row #1 of the array on the same clock cycle, with subsequent rows having the updates correctly placed on subsequent cycles. The [MAT-ADD] operations are placed so that when all PE's in row #1 of the array contain the correct weight adjustment value then the current operation within them is [MAT-ADD]; at this point the weights are updated across the entire row simultaneously. On the next cycle the block of [MAT-ADD] instructions are passed to row #2 of the array, where the weights in that row are updated. This is repeated until all weights in the neural network layer are updated. In order to update all weights in the network this algorithm is simply repeated once per layer of neurons in the network, each pass of which is preceded with a [SWITCH] command to each column of the array in order to activate the correct internal register within the PE register block.

6.1.5 Backpropagation Timings

Timings for a number of backpropagation neural networks have been evaluated, with the networks being chosen to reflect performance rather than any particular application. Two networks were chosen to be implemented on a standard 6-by-6 array processor, one being a small network and one utilising the array more efficiently. The third and final test network assumes an array processors of size 24x24 PE's has been generated, either by fabricating a VLSI device of that size or by using a technique such as wafer-scale integration to connect together multiple smaller devices. The setup information for these test networks is given in table 6.4.

Table 6.4 Backpropagation Example Neural Network Setups

	6 x 6		24 x 24
Layer 1	4	6	24
Layer 2	3	5	20
Layer 3	2	3	12
Patterns	25	125	500

Throughout these example network timings sufficient start and end delays have been allocated to each algorithm in order to ensure that all results from one algorithm have been output before the next algorithm is initiated. Time overheads of the system control hardware have not been taken into account, which may be a non-trivial figure, as the controller has to collate results and prepare them for future use. However, no algorithm requires its own output values as its input, and any variable data required in an algorithm is fixed into the same position in the algorithm; i.e. the output layer δ-value and Δw calculation in section 6.1.3.1 uses the output layer neuron *OUT* values in instruction rows #1 and #3 of the algorithm at all times, and there are no cycles of the algorithm where *OUT* is present on any other instruction rows or where *OUT* is not present on rows #1 and #3.

The timing charts show the number of cycles required for each particular algorithm for each particular network. They show the time required to setup an algorithm, the time required for the first result to be made available and the time required for each additional result. Note that in the first algorithm, the generation of neural *NET* values, has a different timing requirement for the output of the first result per layer of the network. This is because an *n*-neuron layer only requires *n* rows of the array in order to output all *NET* values, as each row in the array contains weights for one neuron in each layer. Therefore, the timing for this element is averaged over all three layers, which is why it is shown to sometimes take a non-whole number of cycles in order to carry out the task. The timing for each additional weight update follows the same pattern, as each layer requires a slightly different execution time. The final value for total cycles required is the value required over all three layers of each network.

The algorithm names have been truncated as follows:

NET =	Generation of neural *NET* values
OUT =	Generation of *OUT* values through approximation of *F(NET)* function
HLn-a =	Hidden layer-*n* generation of false target values
HLn-b =	Hidden layer-*n* generation of δ- and Δw values
OL =	Output layer generation of δ- and Δw values
Update =	Update of weights in network

Table 6.5 6x6 Array Timing (using non-optimal network)

Direction	**Forward Pass**		**Reverse Pass**					**Update**
Algorithm	*NET*	*OUT*	*HL1-a*	*HL1-b*	*HL2-a*	*HL2-b*	*OL*	*Update*
Setup	1	22	9	11	9	11	13	11
First	15	16	15	15	14	15	15	8
Additional	1	1	2	1	2	1	1	4.5
Cycles Req'd	120	336	30	425	27	325	177	25

Table 6.5 shows timings for a three layer neural network consisting of four, three and two neurons respectively, with a training set size of 25 patterns. Each training pass takes a total of 1,468 clock cycles, with 456 clock cycles for the forward pass and 1,012 clock cycles for the reverse pass and associated weight updates.

Table 6.6 6x6 Array Timing (using optimal network)

Direction	**Forward Pass**		**Reverse Pass**					**Update**
Algorithm	*NET*	*OUT*	*HL1-a*	*HL1-b*	*HL2-a*	*HL2-b*	*OL*	*Update*
Setup	1	22	9	11	9	11	13	11
First	16.67	16	18	15	17	15	15	12
Additional	1	1	2	1	2	1	1	6.5
Cycles Req'd	425	1864	37	4525	34	3775	1902	36

Table 6.6 shows timings for a three layer neural network consisting of six, five and three neurons respectively, with a training set size of 125 patterns. Each training pass takes a total of 12,598 clock cycles, with 2,289 clock cycles for the forward pass and 10,309 clock cycles for the reverse pass and associated weight updates.

Table 6.7 24x24 Array Timing

Direction	**Forward Pass**		**Reverse Pass**					**Update**
Algorithm	*NET*	*OUT*	*HL1-a*	*HL1-b*	*HL2-a*	*HL2-b*	*OL*	*Update*
Setup	1	22	27	11	27	11	31	29
First	66.67	16	72	15	68	15	15	48
Additional	1	1	2	1	2	1	1	17
Cycles Req'd	1700	7035	145	72025	133	60025	30045	111

Table 6.7 shows timings for a three layer neural network consisting of 24, 20 and 12 neurons respectively, with a training set size of 500 patterns. Each training pass takes a total of 171,219 clock cycles, with 8,735 clock cycles for the forward pass and 162,484 clock cycles for the reverse pass and associated weight updates.

These timing figures give rise to the figures shown in table 6.8, which show the number of weight update calculations per second during a training run, at a nominal clock speed of 20MHz. Table 6.8 also shows the total number of neuron activations per second during a forward pass (i.e. when the system is in recognition mode), as well as the number of clock cycles required per neuron activation.

Table 6.8 Final Backpropagation Timings

	WU / Sec	**Act / Sec**	**Cycles / Act**
4-3-2 Network	11.57×10^6	9.86×10^6	2.026
6-5-3 Network	16.06×10^6	15.28×10^6	1.308
24-20-12 Network	75.16×10^6	64.12×10^6	0.311

6.2 Kohonen Learning Algorithm

6.2.1 Introduction

6.2.1.1 Kohonen Instruction Set

The Kohonen algorithm requires 8 instruction slots in order to be implemented on the systolic array processor. All instructions use the default result truncation range of the system maximum and minimum numbers. Instruction mnemonics are defined in a similar way to those for the backpropagation algorithm, as described in section 6.1.1.1.

Table 6.9 Kohonen Instruction Set

Opcode	Mnemonic	Description
[0100]	[MVM] *col 1*	NS * REGISTER → WE'
	[MVM] *other*	WE + (NS * REGISTER) → WE'
[0101]	[LOAD]	WE → WE' & REGISTER'
[0110]	[MIN]	MIN (NS, REG) → WE', NS' & REGISTER'
[0111]	[EQUAL]	EQ (NS, REG) → WE' & NS'
[1000]	[SCALE]	NS * REGISTER → WE'
[1001]	[MULT]	WE * REGISTER → WE'
[1010]	[ADD]	NS + REGISTER → WE'
[1011]	[REG-ADD]	NS + REGISTER → WE' & REGISTER'

Table 6.9 shows the complete programmable instruction set for the Kohonen neural learning algorithm. As in the backpropagation learning algorithm some instructions have multiple definitions depending upon which row or column the PE is in the systolic array.

6.2.1.2 Algorithm Summary

In order to ease implementation all weights stored in the network have their signs reversed. All equations are modified in order to take this fact into account. Hence, all references to maximum and minimum values from section 3.2.3 have been swapped.

The learning algorithm can be split into three distinct sections: forward pass, reverse pass and weight update. The forward pass concerns itself with generating the output values for each neuron in the network for a particular input pattern. It contains three components:

i) Generate OUT Value for Neurons

Carries out a simple matrix-vector multiplication operation using the weight matrix and an input vector, producing a series of neuron *OUT* values (as described by equation 3.9)

ii) Search for Minimum OUT Value

Checks each of the *OUT* values generated and finds the smallest value, which represents the winning neuron value, as described in section 3.2.3.1.

iii) Modify OUT Values to 1.0/0.0

Changes each of the *OUT* values to either 1.0 or 0.0, depending on whether the *OUT* value equals the minimum *OUT* value or not, as described in section 3.2.3.1.

The reverse pass calculates the weight update Δw in the winning neuron. Only the neuron that won the forward pass has it's weights modified, with other modifications being set to 0.0; this is based on equation 3.11.

The weight-update pass updates the weights held in the PE internal register blocks with the adjustment values that were calculated in the reverse pass; this is based on equation 3.11.

6.2.2 Forward Pass Components

6.2.2.1 Generate OUT Value for Neurons

This operation is a simple matrix-vector multiplication. An input vector is presented to the array and each PE performs the operation [MVM], with PE's in the first column of the array starting off the accumulation process. The internal values, such as $W_{2,3}$, are the contents of the internal registers representing the weights in the network. The subscript denotes neuron/input, with PE's in a single row representing a single neuron. Processors in the right-most column do not contain weight values and are not used in this algorithm. Each row of the array, each of which represents an individual neuron in the network layer, generates a single *OUT* value for the given input vector. The algorithm process is shown in figure 6.8.

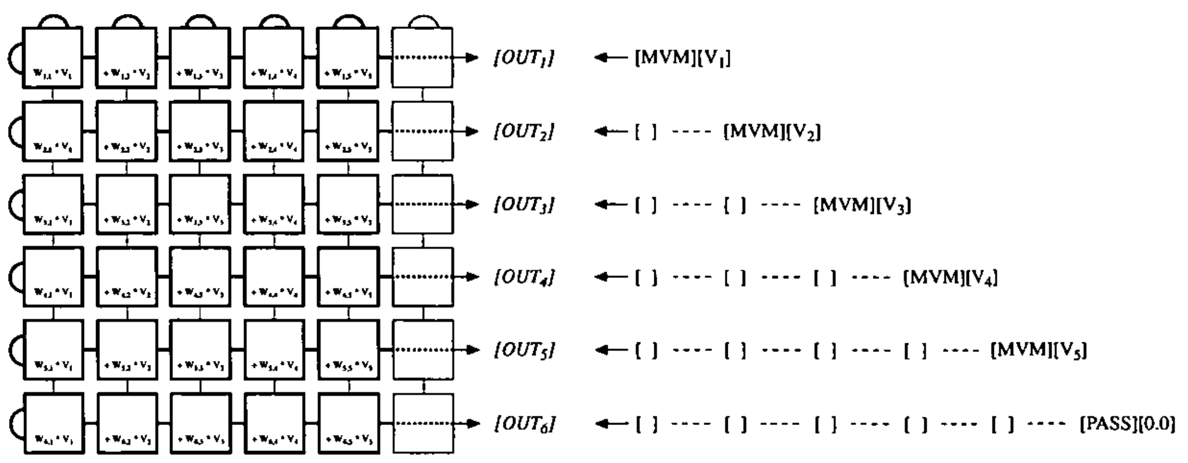

[OUT1] ← [MVM][V1]
[OUT2] ← [ ] ---- [MVM][V2]
[OUT3] ← [ ] ---- [ ] ---- [MVM][V3]
[OUT4] ← [ ] ---- [ ] ---- [ ] ---- [MVM][V4]
[OUT5] ← [ ] ---- [ ] ---- [ ] ---- [ ] ---- [MVM][V5]
[OUT6] ← [ ] ---- [ ] ---- [ ] ---- [ ] ---- [ ] ---- [PASS][0.0]

Figure 6.8 Algorithm for OUT Value Calculation

PE's in the second-last column of the array will output a result on the clock cycle following the PE in the previous row; the first row will output its first result after 12 clock cycles, with results on successive instances of the same algorithm appearing on successive clock cycles.

6.2.2.2 Search for Minimum OUT Value

This operation is a data search operation. It scans through each of the generated *OUT* values and finds the smallest one, which represents the output of the winning neuron. Note that due to the fact that the signs of the weights are reversed the minimum *OUT* is searched for rather than the maximum. A linear search through an output vector is made in the top-right PE, which is initialised to the maximum representable number before the algorithm begins. Each of the *OUT* values from the output vector are applied in turn, and if a value is less than that stored in the PE then it is written into the internal register, thus becoming the minimum value. This algorithm has no discernable outputs, although the minimum *OUT* value is left stored in the top-right PE in the array. The algorithm process is shown in figure 6.9

6.2.2.3 Modify OUT Values

This operation modifies all *OUT* values, as well as producing training rate α values that are specific to each neuron - these α values are required when calculating the weight updates in the reverse pass. It scans through each of the generated *OUT* values and checks them against the stored minimum *OUT* value. If an *OUT* value equals the stored minimum value then it replaced with the value 1.0, else it is replaced by the value 0.0. This is carried out in the top-right PE in the array. The PE directly underneath this, which has the operation [SCALE] locked into it and the training rate parameter -α stored within it, takes these new *OUT* values and produces either 0.0 or -α as a result. This new value is then used later in calculation of the weight updates, acting as the α_x value for the neuron with the corresponding *OUT* value. The algorithm process is shown in figure 6.10.

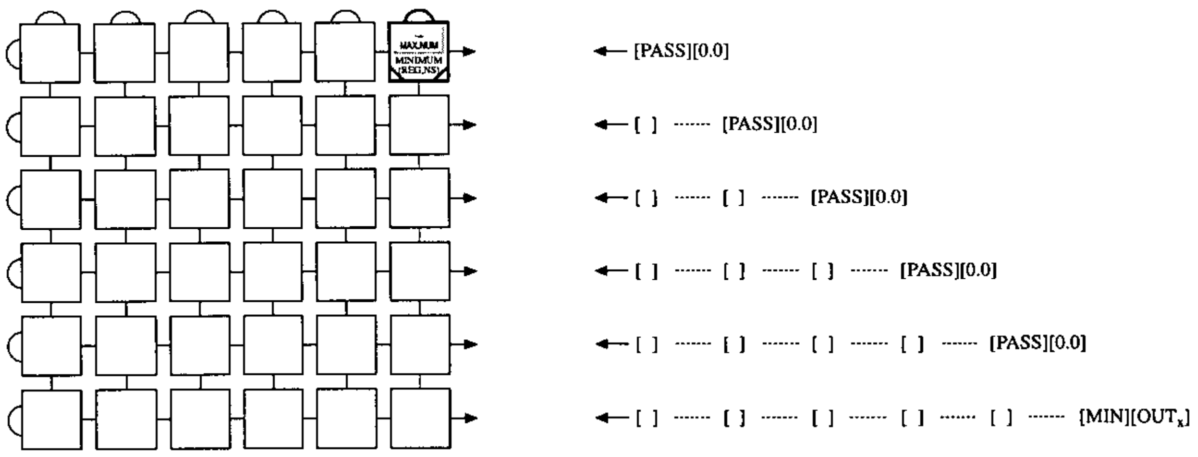


Figure 6.9 Algorithm for Minimum OUT Search

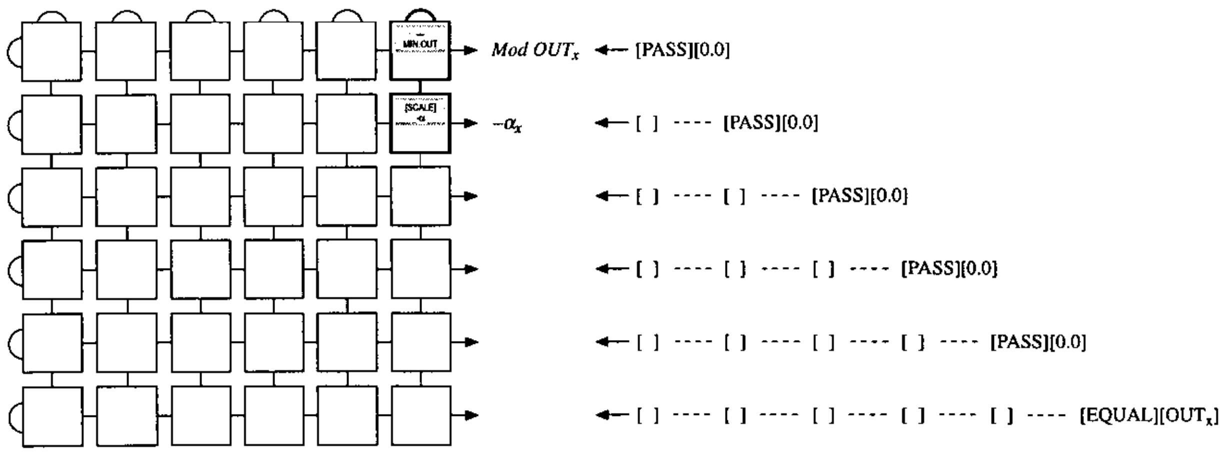

MIN.OUT
[SCALE]
-α
Mod OUT_x
$-\alpha_x$
[PASS][0.0]
[ ] ---- [PASS][0.0]
[ ] ---- [ ] ---- [PASS][0.0]
[ ] ---- [ ] ---- [ ] ---- [PASS][0.0]
[ ] ---- [ ] ---- [ ] ---- [ ] ---- [PASS][0.0]
[ ] ---- [ ] ---- [ ] ---- [ ] ---- [ ] ---- [EQUAL][OUT$_x$]

Figure 6.10 Algorithm for Modified OUT Generation

The two algorithms shown in figures 6.9 and 6.10 are designed to be run back-to-back with no intermediate delay. The stored value in the top-right PE in the second algorithm is generated by the first algorithm and does not have to be loaded in by the second. Also, the locked information for α_x generation in the second algorithm could be loaded in during the setup phase of the first; the results for this operation can just be ignored during the run of the first algorithm and read off as required during the second algorithm.

6.2.3 Reverse Pass: Δw Calculation

This operation calculates the individual weight adjustments required per neuron for the current input pattern. The calculation required is

$$\Delta w_{x,n} = \alpha_x \left(V_n - w_{x,n} \right) \tag{6.4}$$

where

α_x = training rate for neuron x

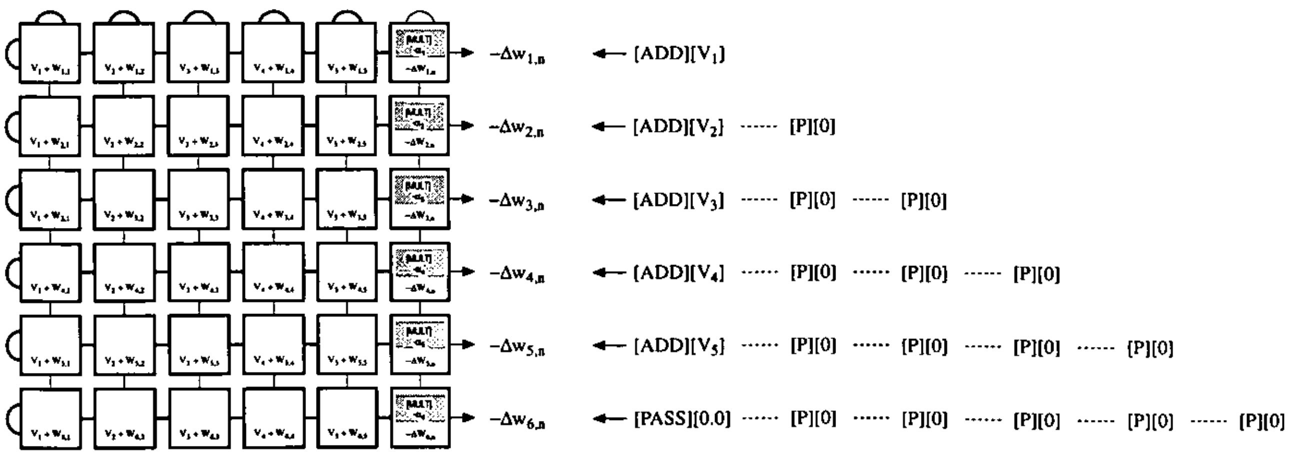
V_n = n-th component of input vector

$w_{x,n}$ = n-th weight in neuron x

The α_x value is either α or 0.0, depending on whether or not neuron x was the winning neuron. As the weights held in the array are of the incorrect sign the actual calculation carried out by this step of the algorithm is

$$-\Delta w_{x,n} = -\alpha_x \left(V_n + w_{x,n} \right) \tag{6.5}$$

Every weight in the array has an adjustment calculated, although weights that are not associated with the winning neuron have an adjustment value of 0.0. The algorithm process is shown in figure 6.11.



$-\Delta w_{1,n}$
← [ADD][V_1]
$-\Delta w_{2,n}$
← [ADD][V_2] ······ [P][0]
$-\Delta w_{3,n}$
← [ADD][V_3] ······ [P][0] ······ [P][0]
$-\Delta w_{4,n}$
← [ADD][V_4] ······ [P][0] ······ [P][0] ······ [P][0]
$-\Delta w_{5,n}$
← [ADD][V_5] ······ [P][0] ······ [P][0] ······ [P][0] ······ [P][0]
$-\Delta w_{6,n}$
← [PASS][0.0] ······ [P][0] ······ [P][0] ······ [P][0] ······ [P][0] ······ [P][0]
MULT

Figure 6.11 Algorithm for Weight Adjustment Calculation

The input algorithm to figure 6.11 is entered in a vertical fashion rather than staggered. This is so that every PE in a single row receives the [ADD] instruction, along with the associated input vector component, on the same clock cycle. This allows adjustments for all five weights in a neuron to be passed along to the right-most column one by one, in order to be scaled by the relevant training parameter α_x. Hence, each row outputs five results, one for each weight in the neuron.

Although all non-winning neuron weight adjustments are 0.0 this method requires no additional processing time over calculating just the winning neuron weight adjustments. No complex control is required, as the most time consuming element of this algorithm is the setting up of the locked instructions and the storing of the training parameters in the right-most column of PE's.

6.2.4 Update Network Weights

The weights held in the register blocks within the PE's must be updated at the end of each training pass. Figure 6.12 shows the majority of the algorithm required to update the weights in a network layer consisting of six neurons, although for the sake of clarity the operations [PASS] and [REG-ADD] have been truncated to [P] and [RA] respectively.

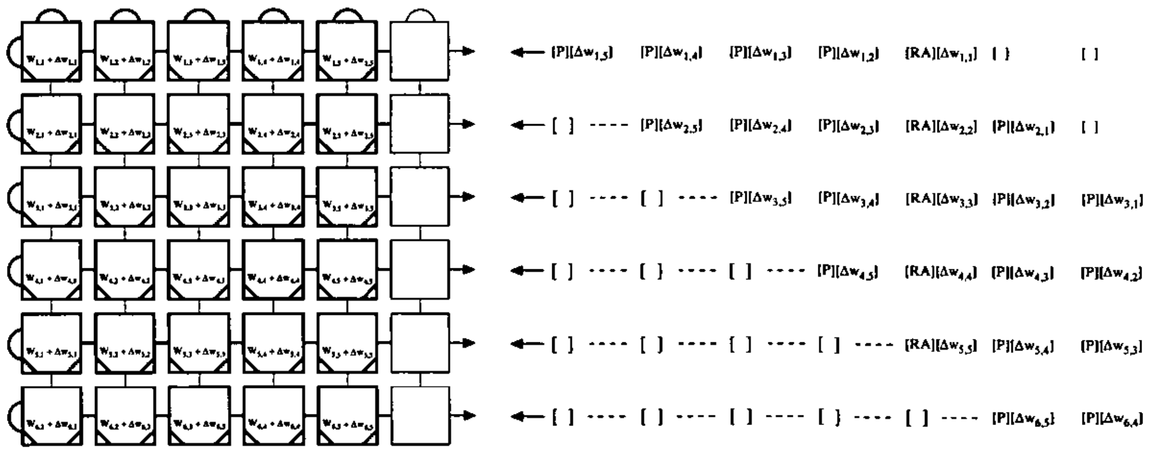

[P][Δw1,5] [P][Δw1,4] [P][Δw1,3] [P][Δw1,2] [RA][Δw1,1] [ ] [ ]
[ ] ---- [P][Δw2,5] [P][Δw2,4] [P][Δw2,3] [RA][Δw2,2] [P][Δw2,1] [ ]
[ ] ---- [ ] ---- [P][Δw3,5] [P][Δw3,4] [RA][Δw3,3] [P][Δw3,2] [P][Δw3,1]
[ ] ---- [ ] ---- [ ] ---- [P][Δw4,5] [RA][Δw4,4] [P][Δw4,3] [P][Δw4,2]
[ ] ---- [ ] ---- [ ] ---- [ ] ---- [RA][Δw5,5] [P][Δw5,4] [P][Δw5,3]
[ ] ---- [ ] ---- [ ] ---- [ ] ---- [ ] ---- [P][Δw6,5] [P][Δw6,4]

Figure 6.12 Algorithm for Weight Update

The weight updates are fed into the array rows in reverse order, each of which are contained within a [PASS] instruction. They are arranged so that the weight updates are present in the correct PE's in row #1 of the array on the same clock cycle, with subsequent rows having the updates correctly placed on subsequent cycles. The [REG-ADD] operations are placed so that when all PE's in row #1 of the array contain the correct weight adjustment value then the current operation within them is [REG-ADD]; this is identical to the method used to update the weights in a backpropagation network, as described in section 6.1.4.

6.2.5 Network Timings

Timings for several Kohonen neural networks have been evaluated, with the networks being chosen to reflect performance rather than any particular application. Two networks were chosen to be implemented on a single 6-by-6 array of processors, both with neurons accepting five inputs; one holds the neural weights in one internal register whilst the other utilises two internal registers per PE in order to double the effective number of neurons. Two other networks assumes that a number of array processors have been connected together in order to form a 12-by-12 array of PE's. One of these larger networks emulates the larger of the two 6-by-6 array networks, whilst one uses a larger network architecture. The setup information for these test networks is given in table 6.10.

Table 6.10 Kohonen Example Neural Network Setups

	6 x 6		12 x 12	
Neurons	6	12	12	12
Inputs	5	5	5	11
Patterns	25	25	25	125

The network timings are arranged in the same manner as those for the backpropagation algorithm in that sufficient start and end delays exist at each stage of the algorithm. Although the Kohonen neural training algorithm is unsupervised there is still a training mode, which is where weights are adjusted in response to input patterns. Because of this the timing charts show two versions of the forward pass elements of the algorithm. In training mode the patterns are trained one by one, whereas in operational mode the patterns can be applied in batches and thereby reduce the effective time for a classification to be made.

The algorithm names have been truncated as follows:

OUT = Generation of neural *OUT* values

MIN = Search for minimum *OUT* value

WIN = Modify *OUT* values to 1.0 or 0.0

Δw = Calculate weight update for winning neuron

UPD = Update weights in network

Table 6.11 6x6 Array Timing (small network)

Mode	**Training Mode**					**Operational Mode**		
Algorithm	*OUT*	*MIN*	*WIN*	Δw	*UPD*	*OUT*	*MIN*	*WIN*
Setup	1	8	-	4	-	1	8	-
First	18	7	9	18	12	18	7	8
Additional	-	1	1	-	-	1	1	1
Cycles Req'd	475	500	350	550	300	43	500	325

Table 6.11 shows timings for a network consisting of six neurons, with five inputs per neurons and a training set size of 25 patterns. A training pass for 25 patterns takes 2,175 cycles, with the forward recognition pass taking 1,325 cycles. In operational mode, where recognition is done using batches of patterns, the recognition pass takes only 868 cycles.

Table 6.12 6x6 Array Timing (large network)

Mode	**Training Mode**					**Operational Mode**		
Algorithm	*OUT*	*MIN*	*WIN*	Δw	*UPD*	*OUT*	*MIN*	*WIN*
Setup	1	8	-	4	-	1	8	-
First	20	7	9	20	15	20	7	8
Additional	-	1	1	-	-	4	1	1
Cycles Req'd	525	650	500	600	375	117	650	475

Table 6.12 shows timings for a network consisting of twelve neurons, with five inputs per neurons and a training set size of 25 patterns. A training pass for 25 patterns takes 2,650 cycles, with the forward recognition pass taking 1,675 cycles. In operational mode, where recognition is done using batches of patterns, the recognition pass takes only 1,242 cycles.

Table 6.13 shows timings for a network consisting of twelve neurons, with five inputs per neurons and a training set size of 25 patterns. This is identical to the previous network, except that it is implemented on a 12-by-12 array of processors rather than using two layers of internal registers per PE on a 6-by-6 array of processors. A training pass for 25 patterns takes 3,775 cycles, with the forward recognition pass taking 2,075 cycles. In operational mode, where recognition is done using batches of patterns, the recognition pass takes only 646 cycles. Note, the algorithms for finding the minimum *OUT* value and for modifying the *OUT* values to 1.0 or 0.0 can be split over two 6-by-6 array processors in operational mode, each processing half of the batch of input patterns.

Table 6.13 12x12 Array Timing (small network)

Mode	**Training Mode**					**Operational Mode**		
Algorithm	*OUT*	*MIN*	*WIN*	*Δw*	*UPD*	*OUT*	*MIN*	*WIN*
Setup	1	8	-	4	-	1	8	-
First	36	7	9	36	28	36	7	8
Additional	-	1	1	-	-	1	1	1
Cycles Req'd	925	650	500	1000	700	61	338	247

Table 6.14 12x12 Array Timing (large network)

Mode	**Training Mode**					**Operational Mode**		
Algorithm	*OUT*	*MIN*	*WIN*	*Δw*	*UPD*	*OUT*	*MIN*	*WIN*
Setup	1	8	-	4	-	1	8	-
First	36	7	9	36	36	36	7	8
Additional	-	1	1	-	-	1	1	1
Cycles Req'd	4625	3250	2500	5000	4500	161	1638	1197

Table 6.14 shows timings for a network consisting of twelve neurons, with eleven inputs per neurons and a training set size of 125 patterns. A training pass for 125 patterns takes 19,875 cycles, with the forward recognition pass taking 10,375 cycles. In operational mode, where recognition is done using batches of patterns, the recognition pass takes only 2,996 cycles. Again, the algorithms for finding the minimum *OUT* value and for modifying the *OUT* values have been speeded up by splitting the input patterns over two 6-by-6 array processors whilst the system is in operational mode.

These timing figures give rise to the figures shown in table 6.15, which shows the number of patterns that the network can train per second and the rates of neuron activation during both training mode and operational mode.

Table 6.15 Final Kohonen Timings

	Patterns / Sec	Train-Act / Sec	Op-Act / Sec
6x6 (small)	2.29×10^5	2.26×10^6	3.45×10^6
6x6 (large)	1.88×10^5	3.58×10^6	4.83×10^6
12x12 (small)	1.32×10^5	2.89×10^6	9.28×10^6
12x12 (large)	1.25×10^5	2.89×10^6	10.01×10^6

The timings in table 6.15 assume a nominal clock speed of 20MHz. It is interesting to note that it is faster to train a twelve neuron / five input network on a 6-by-6 array and then use it on a 12-by-12 array. This is due to the more efficient use of pipelining in the smaller array when just a single pattern is being processed in training mode. When a batch of patterns require processing in operational mode the larger array has a lower additional pattern overhead than the smaller, which more than cancels out this increase, as the figures in table 6.15 show.

The number of neural activations in training mode is independent of the number of neural inputs if the number of neurons in the network remains constant, as in the two 12-by-12 array examples. This is due to the inputs to the neurons being presented to the PE array in columns from left to right in the array. The time taken for a one-input neuron vector product to reach the array I/O controller on the right-most edge is not affected if, on the way, additional products for additional neural inputs are also accumulated. Hence, a one-input neuron will present it's output from the array in the same number of clock cycles as a 12-input neuron.

6.3 Counter Propagation Learning Algorithm

6.3.1 Introduction

6.3.1.1 Counter Propagation Instruction Set

The counter propagation algorithm requires 8 instruction slots in order to be implemented on the systolic array processor. All instructions use the default result truncation range of the system maximum and minimum numbers. Instruction mnemonics are defined in a similar way to those for the backpropagation algorithm, as described in section 6.1.1.1.

Table 6.16 shows the complete programmable instruction set for the counter propagation neural learning algorithm. As in the backpropagation learning algorithm some instructions have multiple definitions depending upon which row or column the PE is in the systolic array. This instruction set is identical to that required for the Kohonen learning algorithm. This is not surprising, as one half of the counter-propagation is a direct implementation of the Kohonen learning algorithm, with the other half being the Grossberg outstar. The Grossberg training algorithms, as shown in section 3.3.1.2, are remarkably similar to those used in the Kohonen networks, so no additional instructions over and above the Kohonen instructions are required.

Table 6.16 Counter Propagation Instruction Set

Opcode	Mnemonic	Description
[0100]	[MVM] *col 1*	NS * REGISTER → WE'
	[MVM] *other*	WE + (NS * REGISTER) → WE'
[0101]	[LOAD]	WE → WE' & REGISTER'
[0110]	[MIN]	MIN (NS, REG) → WE', NS' & REGISTER'
[0111]	[EQUAL]	EQ (NS, REG) → WE' & NS'
[1000]	[SCALE]	NS * REGISTER → WE'
[1001]	[MULT]	WE * REGISTER → WE'
[1010]	[ADD]	NS + REGISTER → WE'
[1011]	[REG-ADD]	NS + REGISTER → WE' & REGISTER'

6.3.1.2 Algorithm Summary

The learning algorithm can be split into three distinct sections: forward pass, reverse pass and weight update. The forward pass concerns itself with generating the output values for each neuron in the network for a particular input pattern, which themselves are outputs from the Kohonen layer of neurons. This pass is just a simple matrix-vector multiplication operation using the Grossberg weight matrix and the input vector, producing a series of neuron *OUT* values (as described by equation 3.13). As only a single input is non-zero, due to only a single output from the Kohonen layer being non-zero, this algorithm results in all Grossberg neuron weights connected to a non-zero input to be passed directly to the Grossberg neuron outputs.

The reverse pass calculates the weight update Δw for all neurons in the network, although only those neurons connected to the non-zero input are actually modified, and this operation is based upon equation 3.14. These updates take into account the target value for the input to the Grossberg layer, which is required due to the Grossberg network being a supervised learning neural model. The weight-update pass updates the weights held in the PE internal register blocks with the adjustment values that were calculated in the reverse pass; this also is based on equation 3.14.

6.3.2 Forward Pass

This operation is a simple matrix-vector multiplication. An input vector is presented to the array and each PE performs the operation [MVM], with PE's in the first column of the array starting off the accumulation process. The internal values, such as $W_{2,3}$, are the contents of the internal registers representing the weights in the network. The subscript denotes neuron/input, with PE's in a single row representing a single neuron. Processors in the right-most column do not contain weight values and are not used in this algorithm. Each row of the array, each of which represents an individual neuron in the network layer, generates a single *OUT* value for the given input vector. The algorithm process is shown in figure 6.13.

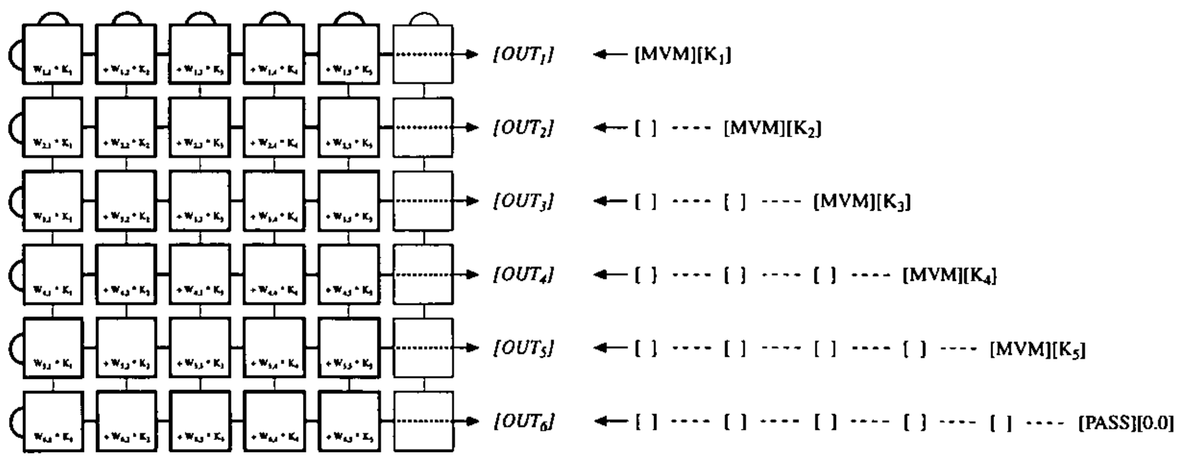

[OUT1] ← [MVM][K1]
[OUT2] ← [ ] ---- [MVM][K2]
[OUT3] ← [ ] ---- [ ] ---- [MVM][K3]
[OUT4] ← [ ] ---- [ ] ---- [ ] ---- [MVM][K4]
[OUT5] ← [ ] ---- [ ] ---- [ ] ---- [ ] ---- [MVM][K5]
[OUT6] ← [ ] ---- [ ] ---- [ ] ---- [ ] ---- [ ] ---- [PASS][0.0]

Figure 6.13 Algorithm for OUT Value Calculation

The PE's in the second-last column of the array will output a result on the clock cycle following the PE in the previous row; the first row will output its first result after 12 clock cycles, with results on successive instances of the same algorithm appearing on successive clock cycles.

6.3.2 Reverse Pass

This operation calculates the individual weight adjustment required per neuron for the current input pattern. As only a single component of the input to each Grossberg neuron is non-zero the calculation is not relevant for most weights in the network. The calculation required is

$$\Delta w_{x,n} = \beta \cdot (y_x - w_{x,n}) \cdot K_n \qquad (6.6)$$

where

$\beta =$ training rate

$y_x =$ target vector for Grossberg neuron x

$K_n =$ output of Kohonen neuron n

Only those weights attached to the winning Kohonen neuron are modified, so a method of extracting

$$-y_x + w_{x,n} \qquad (6.7)$$

for just one weight per Grossberg neuron needs to be devised; i.e. a conditional [PASS] | [ADD] operation must be created.

The Kohonen layer has to be trained before the Grossberg layer, as otherwise the Grossberg neurons do not receive the same input/target pair and may never train. Hence, during the training of the Grossberg neurons only the forward pass of the Kohonen algorithm is required. When the Kohonen *OUT* values are being modified to 1.0 or 0.0, as described in section 6.2.2.3, a value known as α_x is calculated for use later in the reverse training pass. This is not required for training the Grossberg neurons, but the processing required for the production of α_x can be used to calculate the instruction opcode required in each column of PE's in order to carry out equation 6.7. Each column in the array is required to process either [ADD] or [PASS], depending on the value of K_n. Hence, in the PE where the α_x value is normally carried out in the Kohonen forward pass the instruction [MULT] is locked and the data value 9.0 is stored. The integer result from this calculation is the opcode value for the [ADD] or [PASS] instruction, which is then used in the associated column of weights in the Grossberg layer. This simulates multiplying by K_n and the operation can be seen in figure 6.14.

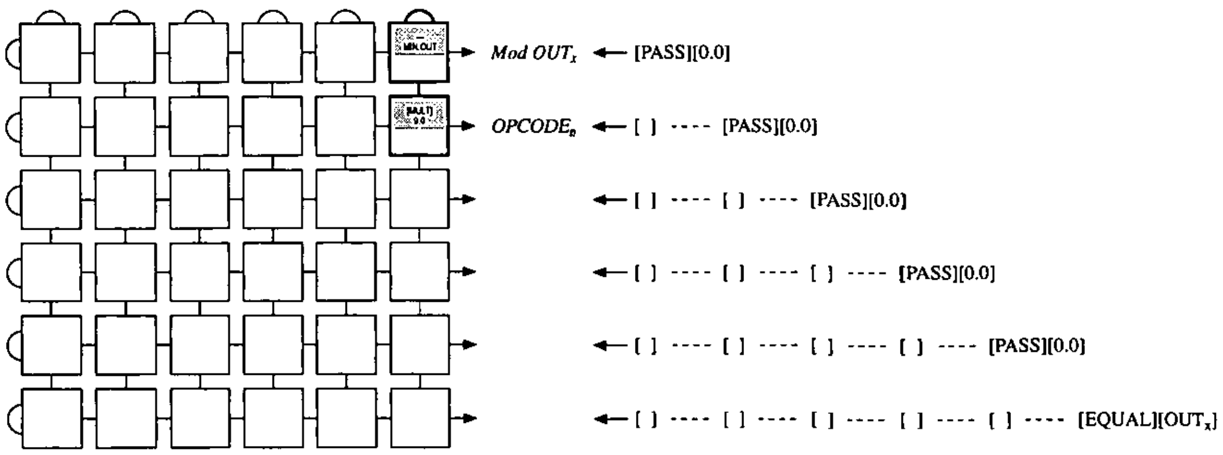

MIN OUT
MULT 9.0
Mod OUT_x ← [PASS][0.0]
OPCODE_n ← [ ] ---- [PASS][0.0]
← [ ] ---- [ ] ---- [PASS][0.0]
← [ ] ---- [ ] ---- [ ] ---- [PASS][0.0]
← [ ] ---- [ ] ---- [ ] ---- [ ] ---- [PASS][0.0]
← [ ] ---- [ ] ---- [ ] ---- [ ] ---- [ ] ---- [EQUAL][OUT_x]

Figure 6.14 Modified Algorithm for Kohonen Modified OUT Generation

With the correct opcodes evaluated the rest of equation 6.6 can be easily calculated, and is shown in figure 6.15, where it is assumed that the second neuron in the Kohonen network was the winning neuron (hence the location of the [ADD] and [PASS] opcodes).

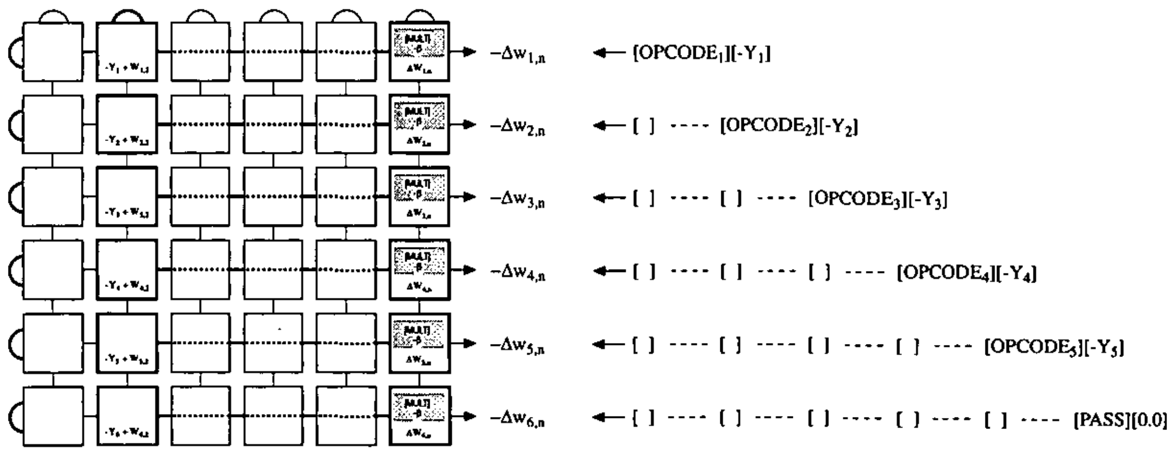

-Y1+W1,2
-Y2+W2,2
-Y3+W3,2
-Y4+W4,2
-Y5+W5,2
-Y6+W6,2
MULT
ΔW
-Δw1,n ← [OPCODE1][-Y1]
-Δw2,n ← [ ] ---- [OPCODE2][-Y2]
-Δw3,n ← [ ] ---- [ ] ---- [OPCODE3][-Y3]
-Δw4,n ← [ ] ---- [ ] ---- [ ] ---- [OPCODE4][-Y4]
-Δw5,n ← [ ] ---- [ ] ---- [ ] ---- [ ] ---- [OPCODE5][-Y5]
-Δw6,n ← [ ] ---- [ ] ---- [ ] ---- [ ] ---- [ ] ---- [PASS][0.0]

Figure 6.15 Algorithm for Weight Adjustment Calculation

6.3.3 Update Network Weights

The weights held in the register blocks within the PE's must be updated at the end of each training pass. Figure 6.16 shows the majority of the algorithm required to update the weights in a network layer consisting of six neurons, although for the sake of clarity the operations [PASS] and [REG-ADD] have been truncated to [P] and [RA] respectively.

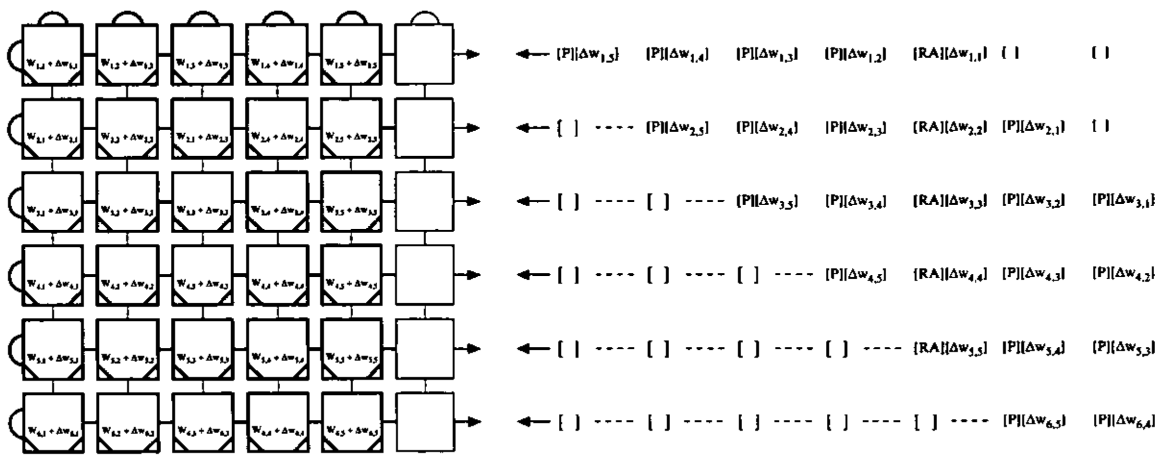

[P][Δw1,5] [P][Δw1,4] [P][Δw1,3] [P][Δw1,2] [RA][Δw1,1] [ ] [ ]
[ ] ---- [P][Δw2,5] [P][Δw2,4] [P][Δw2,3] [RA][Δw2,2] [P][Δw2,1] [ ]
[ ] ---- [ ] ---- [P][Δw3,5] [P][Δw3,4] [RA][Δw3,3] [P][Δw3,2] [P][Δw3,1]
[ ] ---- [ ] ---- [ ] ---- [P][Δw4,5] [RA][Δw4,4] [P][Δw4,3] [P][Δw4,2]
[ ] ---- [ ] ---- [ ] ---- [ ] ---- [RA][Δw5,5] [P][Δw5,4] [P][Δw5,3]
[ ] ---- [ ] ---- [ ] ---- [ ] ---- [ ] ---- [P][Δw6,5] [P][Δw6,4]

Figure 6.16 Algorithm for Weight Update

The weight updates are fed into the array rows in reverse order, each of which are contained within a [PASS] instruction. They are arranged so that the weight updates are present in the correct PE's in row #1 of the array on the same clock cycle, with subsequent rows having the updates correctly placed on subsequent cycles. The [REG-ADD] operations are placed so that when all PE's in row #1 of the array contain the correct weight adjustment value then the current operation within them is [REG-ADD]; this is identical to the method used to update the weights in a backpropagation network, as described in section 6.1.4.

6.3.4 Counter Propagation Timings

6.3.4.1 Introduction

All timings for the counter propagation network have been evaluated using the same parameters as for the Kohonen networks used in section 6.2.5. This setup information has been shown previously in table 6.10, using two networks on a 6-by-6 array of PE's and two networks on a 12-by-12 array of PE's. This allows for timings for just the Grossberg layer to be evaluated as well as for the counter propagation network as a whole, which also uses the timings for the Kohonen networks.

6.3.4.2 Grossberg Layer Timings

Network timings for the Grossberg network layer are arranged in the same manner as those for the backpropagation algorithm in that sufficient start and end delays exist at each stage of the algorithm. The timing charts show two versions of the forward pass element of the algorithm; in training mode the patterns are trained one by one, whereas in operational mode the patterns can be applied in batches and thereby reduce the effective time for a classification to be made.

The algorithm names have been truncated as follows:

OUT = Generation of neural *OUT* values

Δw = Calculate weight updates for all neurons

UPD = Update weights in network

Table 6.17 shows the timings for a network consisting of six neurons, with five inputs per neuron and a training set size of 25 patterns. A training pass for 25 patterns takes 1,325 cycles, with the forward recognition pass taking 475 cycles. In operational mode, where recognition is done using batches of patterns, the recognition pass takes only 42 cycles.

Table 6.17 6x6 Array Timing (small network)

Mode	**Training Mode**			**Op Mode**
Algorithm	*OUT*	*Δw*	*UPD*	*OUT*
Setup	1	4	-	-
First	18	18	12	18
Additional	-	-	-	1
Cycles Req'd	475	550	300	42

Table 6.18 shows the timings for a network consisting of twelve neurons, with five inputs per neuron and a training set size of 25 patterns. A training pass for 25 patterns takes 1,500 cycles, with the forward recognition pass taking 525 cycles. In operational mode, where recognition is done using batches of patterns, the recognition pass takes only 117 cycles.

Table 6.18 6x6 Array Timing (large network)

Mode	**Training Mode**			**Op Mode**
Algorithm	*OUT*	*Δw*	*UPD*	*OUT*
Setup	1	4	-	1
First	20	20	15	20
Additional	-	-	-	4
Cycles Req'd	525	600	375	117

Table 6.19 shows the timings for a network consisting of twelve neurons, with five inputs per neuron and a training set size of 25 patterns. This is identical to the previous network, except that it is implemented on a 12-by-12 array of processors rather than using two layers of internal registers per PE on a 6-by-6 array of processors. A training pass for 25 patterns takes 2,625 cycles, with the forward recognition pass taking 925 cycles. In operational mode, where recognition is done using batches of patterns, the recognition pass takes only 61 cycles.

Table 6.19 12x12 Array Timing (small network)

Mode	**Training Mode**			**Op Mode**
Algorithm	*OUT*	*Δw*	*UPD*	*OUT*
Setup	1	4	-	1
First	36	36	28	36
Additional	-	-	-	1
Cycles Req'd	925	1000	700	61

Table 6.20 shows the timings for a network consisting of twelve neurons, with eleven inputs per neuron and a training set size of 125 patterns. A training pass for 125 patterns takes 14,125 cycles, with the forward recognition pass taking 4,625 cycles. In operational mode, where recognition is done using batches of patterns, the recognition pass takes only 161 cycles.

Table 6.20 12x12 Array Timing (small network)

Mode	**Training Mode**			**Op Mode**
Algorithm	*OUT*	*Δw*	*UPD*	*OUT*
Setup	1	4	-	1
First	36	36	36	36
Additional	-	-	-	1
Cycles Req'd	4625	5000	4500	161

Table 6.21 Final Grossberg Timings

	Patterns / Sec	**Train-Act / Sec**	**Op-Act / Sec**
6x6 (small)	3.77×10^5	2.26×10^6	71.42×10^6
6x6 (large)	3.33×10^5	11.42×10^6	51.28×10^6
12x12 (small)	1.90×10^5	6.48×10^6	98.36×10^6
12x12 (large)	1.76×10^5	6.48×10^6	186.33×10^6

These timing figures give rise to the figures shown in table 6.21, which shows the number of patterns that the network can train per second and the rates of neuron activation during both training mode and operational mode.

During training the Grossberg layer timings are of roughly the magnitude as those for the Kohonen layer; the pattern throughput per second is no more than double and the number of neuron activations per second is slightly more than triple.

In operational mode, however, the Grossberg layer excels. Neuron activations in the Kohonen layer requires three steps, two of which do not use pipelining to any degree and slow the process down drastically. In the Grossberg layer the neuron activations only require a simple matrix-vector multiplication, which is an area in which systolic arrays have always had an impressive record, and give the network layer an impressive turn of speed.

6.3.4.3 Counter Propagation Timings

It is not feasible to produce combined learning times for the full counter propagation network. The Kohonen and Grossberg layers of the network train independently, and the Kohonen layer must be fully trained before Grossberg layer training can commence. This is due to the fact that the input pattern set for the Grossberg layer is the output from a fully trained Kohonen neural network, and it is simply not possible to train the Grossberg layer until it can be presented with a coherent and consistent input set from a trained Kohonen layer.

Combined times for the counter propagation network in operational mode are possible, as only the forward pass elements of both networks are utilised. The timings have used the same four network configurations that have been used throughout sections 6.2 and 6.3. Timings have been given for two operational environments: one for pattern processing, where one pattern is processed at a time, and one for batch processing, where a number of patterns are processes simultaneously.

Table 6.22 shows the times for a counter propagation network when used in single pattern processing mode. It shows the times required for each pattern in the training set to be processed, along with the number of patterns that can be processed and the number of neurons that can be activated per second at a nominal clock speed of 20MHz. Clearly, the times are affected by the reduction in pipelining available due to only single patterns being presented to the network at any one time. This greatly affects networks implemented on the larger 12-by-12 array of PE's, as the time lag between input and output is quite excessive. It is because of the increased latency of the 12-by-12 networks that leads to a 12-neuron 5-input network being more efficient in single pattern mode when implemented on a 6-by-6 array of processors.

The times in table 6.22 also show that when two networks have an identical number of neurons, albeit with a different number of inputs, the processing time required per input pattern is identical. This shows that there is no additional cost when adding network inputs to a network, so long as the number of inputs is less than the number of neurons.

Table 6.22 Counter Propagation Single Pattern Timings

	Kohonen	Grossberg	Totals	Patt / Sec	Act / Sec
6x6 (small)	1325	475	1800	2.77×10^5	1.60×10^6
6x6 (large)	1675	525	2200	2.27×10^5	2.72×10^6
12x12 (small)	2075	925	3000	1.66×10^5	2.00×10^6
12x12 (large)	10375	4625	15000	1.66×10^5	2.00×10^6

Table 6.23 shows the times for a counter propagation network when used in batch pattern processing mode. It shows the same information as table 6.22, but for the network in batch processing mode instead of pattern processing.. These timings show that the increased use of pipelining within the networks, by virtue of processing a batch of patterns together instead of just one, have increased the overall efficiency of the networks.

Table 6.23 Counter Propagation Batch Pattern Timings

	Kohonen	Grossberg	Totals	Patt / Sec	Act / Sec
6x6 (small)	868	42	910	5.49×10^5	3.29×10^6
6x6 (large)	1242	117	1359	3.67×10^5	4.41×10^6
12x12 (small)	646	91	737	6.78×10^5	8.14×10^6
12x12 (large)	2996	161	3157	7.96×10^5	9.50×10^6

NEURAL NETWORK SOFTWARE SIMULATION

INTRODUCTION

This section describes the soft systolic system written to simulate the systolic array processor. The software system itself is described, along with details of the various options available within the software. It goes on to describe several test applications that were developed for the simulator, with applications available for each of the neural learning algorithms developed in section 6. These applications were chosen to demonstrate that the systolic array processor and neural algorithms described in sections 5 and 6 were capable of carrying out some of the standard neural learning procedures, rather than advanced applications of neural network use in the real world.

7.1 Soft Systolic Simulator

7.1.1 Simulator Overview

The soft systolic simulator was written in the 'C++' programming language on an Intel Pentium computer running the Microsoft Windows95 operating system. The code was developed using the Microsoft *Developer Studio* environment using Visual C++ V5. The simulator will run on any Windows95 or WindowsNT Intel-based computer. It consists of several thousand lines of program code, split into several distinct segments: human-computer interaction, data acquisition/generation and neural simulation.

The software allows the user to set up a set of program instruction codes to use in any simulation run, which consists of 12 instructions capable of carrying out any of the operations specified in section 5. An additional two alternate instructions, depending on the placement of a PE in the array, can be set up per opcode slot.

A project for simulation can be set up for any of the three neural learning methodologies supported, with training parameters relevant to the network specified by the user on-screen. Data files containing training vectors, and any associated target vectors for supervised learning neural networks, are imported into the simulator and then used in a simulation run.

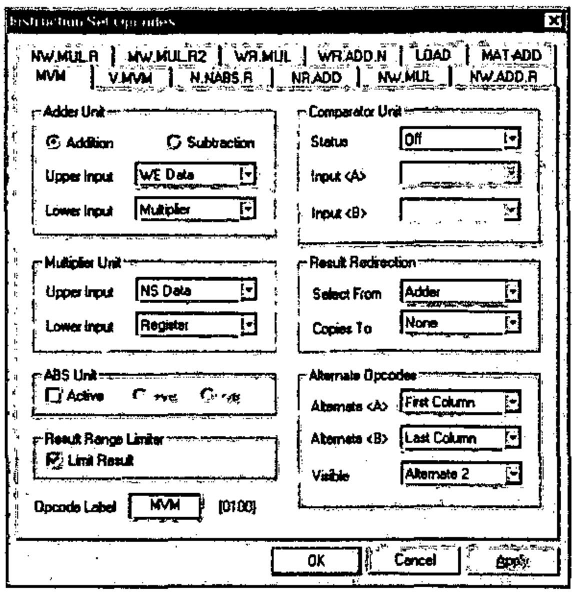
The simulator can be used in three different modes. The simplest mode, known as *direct* mode, takes a set of neural data, as specified above, and uses it as the basis for a simulation run. The user has very little control over the simulation run, bar being able to pause or stop the run if convergence looks unlikely (or likely to take too long). In *step* mode the user can process a single pass of the training algorithm, visualising the outputs of the network in a separate window. In *detailed* mode a virtual representation of the array is displayed, with details of the opcode being processed and the results of the opcode being displayed for each PE in the array. The user can step through each single clock cycle of the process, seeing all instruction results on-screen at all times. The user is able to switch between the three modes at any time, so long as the simulation run has been paused.

The results generated by any training run are simply text files usable in any computer spreadsheet package, with values relating to the network convergence criteria being arranged so that the user can create any representational graph required within the spreadsheet package itself. All network weights are also saved from the simulator, so that they can be used at a later date in a fully trained network that is being used in operational mode rather than training mode.

7.1.2 Simulator Data Definitions

7.1.2.1 Instruction Set Definitions

The opcodes for each supported neural learning methodology can be set up using the single screen shown in figure 7.1. It shows the various options available for each individual instruction. The user can select which instruction is displayed by clicking on the relevant tab-mark at the top of the display - this will bring forward the selected instruction dialog box into the display. All possible options described in table 5.7 are implementable in each dialog, although only two alternate opcode definitions per instruction are possible. The dialog shows that it is a fairly straightforward process to set up an instruction. Some controls are marked as being inactive, such as the sign of the *ABS Unit* operation, and are only activated when other enabling controls are selected. The user is also able to modify the name of the instruction in this dialog, and there is a constant [xxxx] marker by the instruction name to remind the user of the actual opcode number that they are editing.



Adder Unit: [✓ Addition] Subtraction
Upper Input: WE Data
Lower Input: Multiplier

Comparator Unit
Status: Off
Input <A>:
Input <B>:

Multiplier Unit
Upper Input: NS Data
Lower Input: Register

Result Redirection
Select From: Adder
Copies To: None

ABS Unit: Active +ve -ve

Alternate Opcodes
Alternate <A>: First Column
Alternate <B>: Last Column
Visible: Alternate 2

Result Range Limiter: [✓ Limit Result]

Opcode Label: MVM [0100]

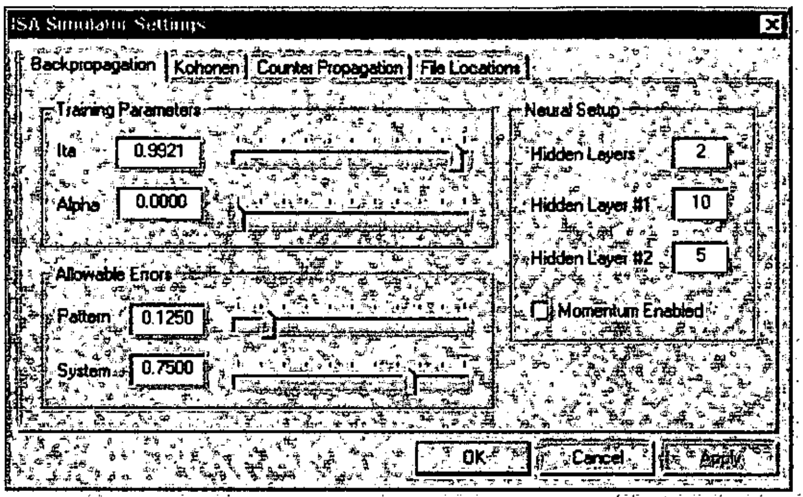
Figure 7.1 Opcode Definition Dialog

The simulator supports three different instruction sets, one for each of the supported learning methodologies, each of which is fully modifiable by the user. Each set of opcodes is stored in a separate data file, which is loaded in by the application automatically. The user may freely modify these instruction sets and save them for later use, but they are not able to overwrite the default instruction sets for each of the three neural learning methods. The user may select to save the current instruction set via a menu selection; if there is an un-saved modified instruction set in memory then the application will prompt the user to save it whenever it is about to be destroyed (such as when the application terminates).

7.1.2.2 Neural Parameter Definitions

In order to carry out a simulation the user must select a the type of neural network that they wish to use and then configure it for the systolic application that they have in mind. The user is able to modify the training parameters for all three neural network types independent of any systolic application that is loaded (if any), and this allows the simulator to be used as a general purpose instruction set editor.

A single tab-marked display is used modify the parameters of each of the three neural network types. This allows the user to modify all of the supported training parameters for each of the three network architectures, although it should be noted that this display does not have any knowledge of the neural training sets themselves. The dialog for the backpropagation network is shown in figure 7.2. The user is able to specify the network training rates, the error bounds for the neural training process and dictate the number of neurons in the hidden layers. It should be noted that the number of neurons in the output layer cannot be specified through this dialog, as this value is directly dependant on the number of elements in the target output vectors defined in the neural training sets.



Training Parameters
Ita 0.9921
Alpha 0.0000

Allowable Errors
Pattern 0.1250
System 0.7500

Neural Setup
Hidden Layers 2
Hidden Layer #1 10
Hidden Layer #2 5
☐ Momentum Enabled

Figure 7.2 Neural Parameter Definition Dialog

For the Kohonen and counter propagation networks a similar scheme is implemented, with the physical layout of their dialog boxes being modified from that shown in figure 7.2 due to the differences in the nature of the training algorithms. The fourth tab-mark on this display is used to select the file locations for the instruction sets, both the default ones for each neural architecture and any others defined by the user. This allows them to load in any previously stored instruction set, as well as to locate a default instruction set.

7.1.2.3 Neural Text File Definition

Each systolic application has an associated text data file, which contains information regarding training set data and the types of network that it is valid for. The format for this data file is as follows:

`<string>`	*Title of neural network (may be left blank)*
`<int> .....`	*Type of valid neural networks*
`<int>`	*Number of training patterns in the set*
`<int>`	*Number of distinct training patterns in the set*
`<int> <int>`	*Number of inputs / required outputs per pattern*
`<float> .....`	*All inputs for pattern 1*
`<float> .....`	*All required outputs for pattern 1*
`...... .....`	....
`<float> .....`	*All inputs for pattern N*
`<float> .....`	*All required outputs for pattern N*

It is clear that the back- and counter propagation networks can share common training files, as they are both supervised networks. As the Kohonen algorithm is unsupervised it would not expect any *required output* values in the text files; hence, any *required output* data is treated as an additional input pattern. The Kohonen and counter propagation networks both use the *distinct patterns* value in order to guess at the number of neurons required in the Kohonen layers of both networks; if the value is 0 then the simulator makes it equal to the number of patterns in the training set.

7.2 Backpropagation Learning Applications

7.2.1 Exclusive-Or Problem

7.2.1.1 Application Overview

The exclusive-or problem is the one that almost halted research for 20 years into the field of artificial neural networks. In their landmark book *Perceptrons* [MiPa69] the authors, proved that single-layer artificial neural networks were theoretically incapable of solving many problems, including the function performed by the simple exclusive-or logic gate. They also were not very optimistic about the future:

> *The Perceptron has shown itself worthy of study despite (and even because of!) it's severe limitations. It has many features that attract attention: its linearity, it's intriguing learning theorem and it's clear paradigmatic simplicity as a kind of*

parallel computation. There is no reason to suppose that any of these virtues carry over to many-layered version. Nevertheless, we consider it to be an important research problem to elucidate (or reject) our intuitive judgement that the extension is sterile.

Perhaps some powerful convergence theorem will be discovered, or some profound reason for the failure to produce an interesting 'learning theorem' for the multi-layered machine will be found.
(pp 231-232)

Unfortunately, Minsky and Paperts conclusions were unassailable. Discouraged researchers left the field for areas of greater promise and, more importantly, greater funding. The breakthrough in the field was to produce a learning algorithm that could train a multi-layer network, which is now known as the backpropagation learning algorithm. A very simple test of any backpropagation system is the exclusive-or network, as this is the type of linearly inseparable application that backpropagation was intended to solve.

7.2.1.2 Network Structure

The architecture of the network to be used is shown in figure 7.3. It shows just two layers of neurons, with two neurons in the hidden layer and a single neuron in the output layer. This is acceptable as a solution for the exclusive-or problem [Lipp87], where only a simple convex open or closed region is required to discriminate between the training set input patterns.

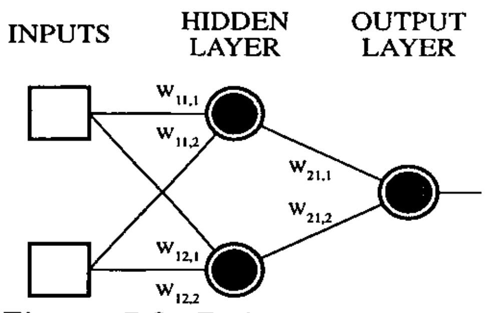


Figure 7.3 Exclusive-Or Network

The training set used is a continuous version of the binary exclusive-or problem, but with the values optimised for the mathematical accuracy available in the array processor. Normally, the continuous values 0.1 and 0.9 are used

Table 7.1 Exclusive-Or Training Set

Pattern	Input A	Input B	Target
1	$^{13}/_{128}$	$^{13}/_{128}$	$^{13}/_{128}$
2	$^{13}/_{128}$	$^{115}/_{128}$	$^{115}/_{128}$
3	$^{115}/_{128}$	$^{13}/_{128}$	$^{115}/_{128}$
4	$^{115}/_{128}$	$^{115}/_{128}$	$^{13}/_{128}$

instead of the binary values for 0 and 1. The nearest equivalences for these values in the array processor are $^{13}/_{128}$ (0.1015625) and $^{115}/_{128}$ (0.8984375) respectively. The complete training set is given in table 7.1.

7.2.1.3 Simulation Run

Table 7.2 shows all of the training parameters used in the simulation run, along with the number of clock cycles required per iteration. The network trained after 814 iterations, with a maximum pattern error of $^{11}/_{128}$ and a maximum system error of $^{21}/_{128}$. These are acceptable errors for the given application, as it allows for full differentiation between all of the target output vectors. A graph showing the reduction in the pattern error over the training run is shown in figure 7.4.

Table 7.2 Exclusive-Or Training Parameters

Parameter	Value
Training Rate	$^{96}/_{128}$ (0.75)
Momentum	OFF
Max Iterations	2,500
Max System Error	$^{26}/_{128}$ (~0.2)
Max Pattern Error	$^{13}/_{128}$ (~0.1)
Cycles / Iteration	317

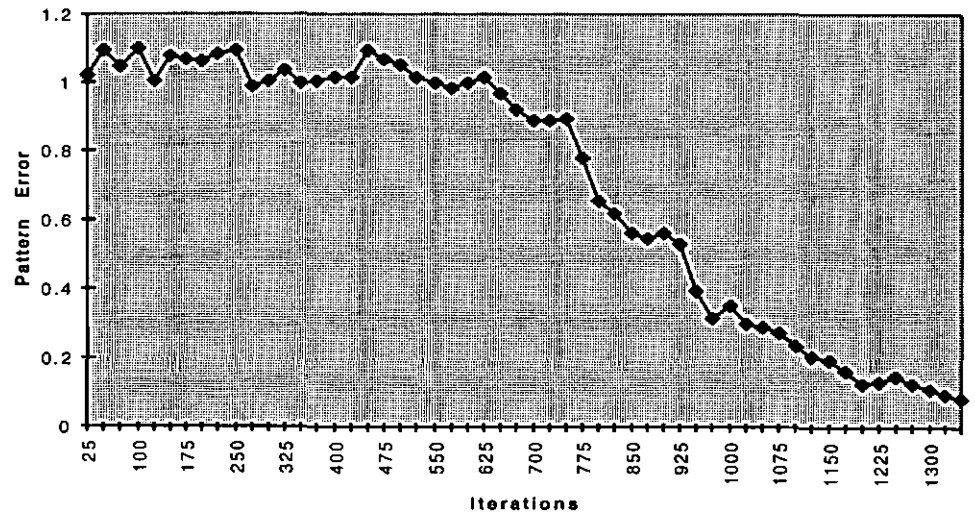


Figure 7.4 Exclusive-Or Training Run Progress

7.2.2 Parity Problem

7.2.2.1 Application Overview

A common approach when looking for problems to test the backpropagation algorithm is to find one which both Minsky and Papert deemed to be a 'hard' problem for a multi-layer perceptron to be able to learn. A classical example amongst such problems is the so-called *parity problem*, in which a network outputs a logic-1 if the total number of logic-1 inputs to the network is odd and logic-0 if the total number of logic-1 inputs is even (or zero).

7.2.2.2 Network Structure

The simplest network that can show this example [AlMo90] consists of just two layers: a single neuron in the output layer and n neurons in the only hidden layer, where n is the number of inputs to the network. The architecture of the network used for this problem is shown in figure 7.5. The object of the training process is to have the neurons in the hidden layer to learn to recognise the number of inputs that are set to logic-1 independent of the inputs that are actually set to logic-1; i.e. neuron 1 in the hidden layer can recognise if one input is logic-1, neuron 2 can recognise if two inputs are logic-1, and so on. The single output layer neuron just has to distinguish between n numbers, discerning whether any particular number is odd or even.

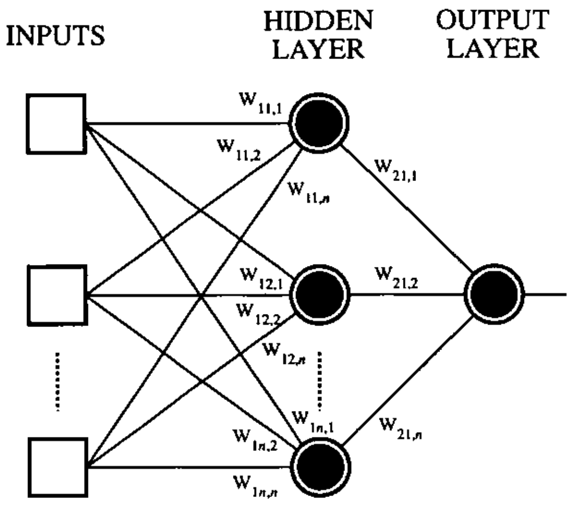


Figure 7.5 Parity Network

This application used the value $n = 6$ for the training run, as processing a neural layer of six neurons is the most optimal setup for the 6-by-6 array processor. This gives rise to 2^n input patterns, each with six components, representing all possible 6-bit binary values; for the sake of brevity these input patterns are not listed here. As in the exclusive-or simulation the binary inputs 0 and 1 have been replaced by the closest representable equivalences of the dynamic values 0.1 and 0.9, as described in section 7.2.1.2. This method could easily be extended for much larger n, although this would dramatically increase the processing time required for each additional neuron in the hidden layer, as it doubles the possible number of input pattern variations.

7.2.2.3 Simulation Run

Table 7.3 shows all of the training parameters used in the simulation run. Two separate training runs were carried out: one containing the entire training set and one containing a random selection of 80% of the training set. This was to see if the neural network was able to generalise the principle behind the parity problem, given the limited mathematical precision available on the array processor. The number of cycles per iteration required when using the entire set and when using the reduced set are both shown in table 7.3.

Table 7.3
Parity Training Parameters

Parameter	Value
Training Rate	$^{77}/_{128}$ (~0.6)
Momentum	$^{32}/_{128}$ (0.25)
Max Iterations	10,000
Max System Error	$^{32}/_{128}$ (0.25)
Max Pattern Error	$^{19}/_{128}$ (~0.15)
Cycles / Iteration	2,243 (100%) 1,847 (80%)

When using the complete training set the network trained after 6,833 iterations, with a maximum pattern error of $^{19}/_{128}$ and a maximum system error of $^{27}/_{128}$. These errors are sufficient, given that the inputs are meant to be equivalent to binary values. A graph showing the reduction in the pattern error over the training run is shown in figure 7.6.

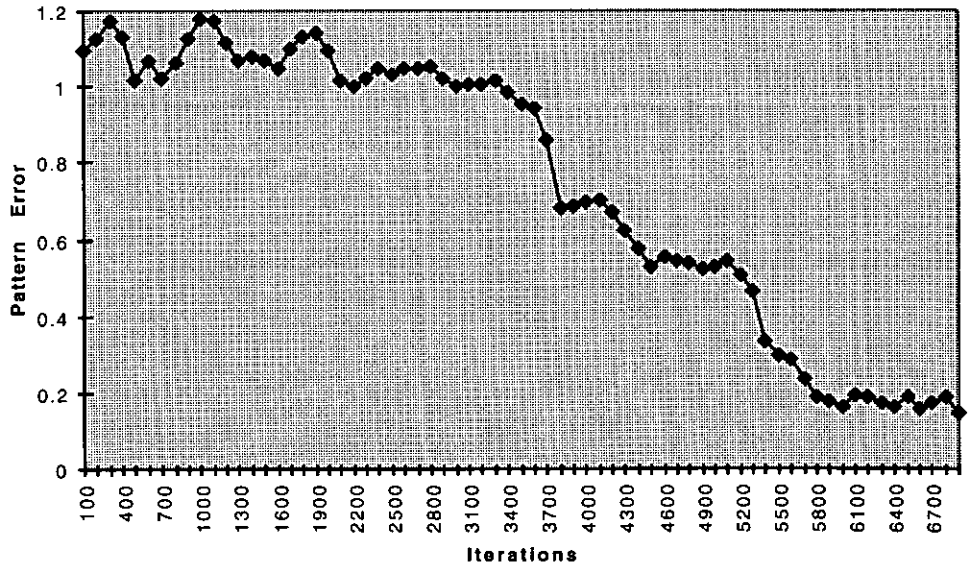


Figure 7.6 Parity Training Run Progress (whole set)

When using 51 of the 64 members of the training set the network trained after 4,822 iterations, with a maximum pattern error of $^{16}/_{128}$ and a maximum system error of $^{32}/_{128}$. Again, these final error values are sufficient for the problem. A graph showing the reduction in the pattern error over the training run is shown in figure 7.7.

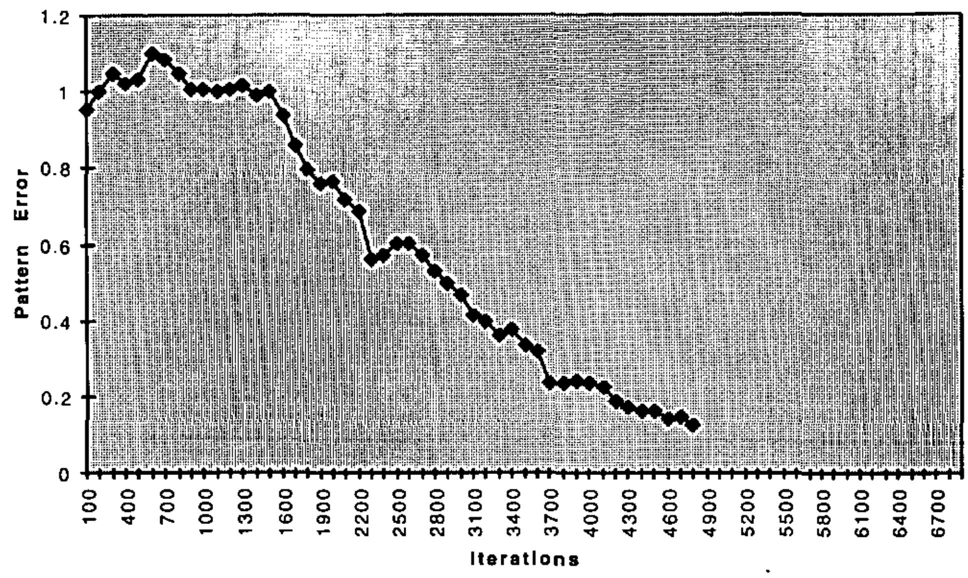


Figure 7.7 Parity Training Run Progress (limited set)

7.2.3 Simple Pattern Recognition Problem

7.2.3.1 Application Overview

One common use of the backpropagation learning algorithm is in the field of pattern recognition and feature extraction. The network is trained on certain known data, normally taken from a near-perfect source, and used in an environment where the input data may be far from perfect; an example of this is in a text recognition neural network that has learnt to recognise the alphabet but has to deal with character images from an imperfect source, such as a fax machine.

7.2.3.2 Network Structure

The network is to be trained on representations of the numeric characters 0 to 9 inclusive. Data for the characters is extracted from a 5-by-3 grid of pixels, each of which is either black (on) or white (off). The entire training set is shown in figure 7.8. This method of using a grid of pixels, with the normalised characters

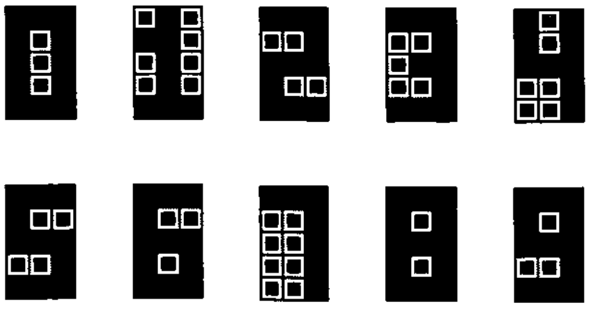


Figure 7.8 Pattern Recognition Training Set Images

projected onto it, has been used successfully in many applications. Burr used this method to recognise series of characters which, when used in conjunction with a dictionary filtering program, led to recognition of complete English words at an accuracy level of 99.7% [Burr87].

The network structure for this problem can be done in several ways. A simple method is to have a single hidden layer, with the number of neurons equal to the number of constituent parts of each input vector, and an output layer with enough neurons to implement some pre-defined encoding strategy. For this application the hidden layer requires 15 units and the output layer requires six units - one to indicate the input is in either the range 0...4 or 5...9 and five more to indicate which of the five values in the recognised range matches the input. Table 7.4 shows the target data values for each input pattern (0.9 and 0.1 are shown in the table instead of their nearest representable equivalents for the sake of clarity).

Table 7.4 Pattern Recognition Training Set Targets

Pattern	Range	Indices				
'0'	0.1	0.9	0.1	0.1	0.1	0.1
'1'	0.1	0.1	0.9	0.1	0.1	0.1
'2'	0.1	0.1	0.1	0.9	0.1	0.1
'3'	0.1	0.1	0.1	0.1	0.9	0.1
'4'	0.1	0.1	0.1	0.1	0.1	0.9
'5'	0.9	0.9	0.1	0.1	0.1	0.1
'6'	0.9	0.1	0.9	0.1	0.1	0.1
'7'	0.9	0.1	0.1	0.9	0.1	0.1
'8'	0.9	0.1	0.1	0.1	0.9	0.1
'9'	0.9	0.1	0.1	0.1	0.1	0.9

7.2.3.3 Simulation Run

Table 7.4 shows all of the training parameters used in the simulation run, along with the number of clock cycles required per iteration. The network trained after 11,617 iterations, with a maximum pattern error of $^{13}/_{128}$ and a maximum system error of $^{17}/_{128}$. These are acceptable errors for the given application, as it allows for full differentiation between all of the target output vectors. A graph showing the reducing pattern error over the training run is shown in figure 7.9.

Table 7.5 Pattern Recognition Training Parameters

Parameter	Value
Training Rate	$^{96}/_{128}$ (0.75)
Momentum	$^{32}/_{128}$ (0.25)
Max Iterations	25,000
Max System Error	$^{19}/_{128}$ (~0.15)
Max Pattern Error	$^{13}/_{128}$ (~0.1)
Cycles / Iteration	14,317

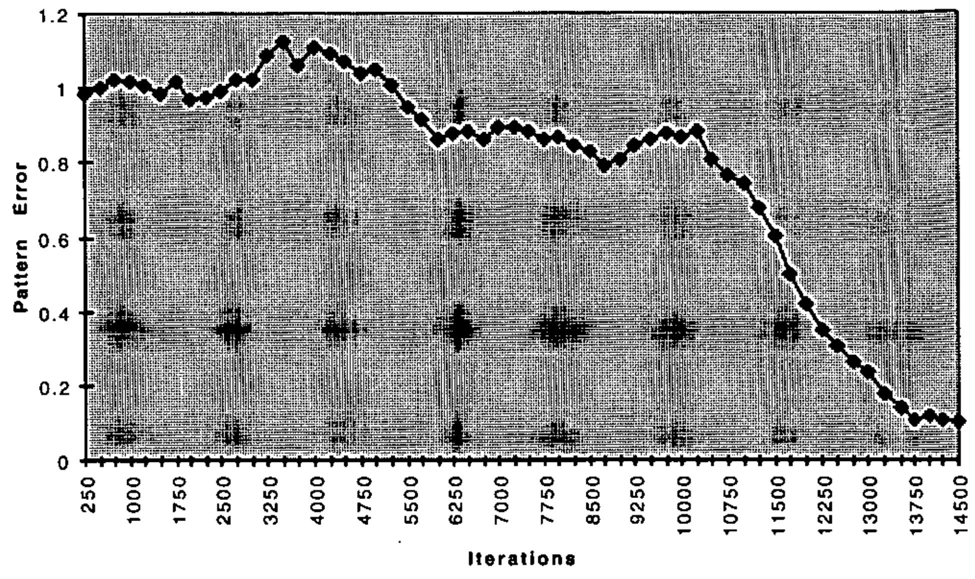


Figure 7.9 Pattern Recognition Training Run Progress

Although this is a fairly trivial example of the backpropagation learning algorithm it does show that the architecture and algorithms designed for the backpropagation learning methodology are able to successfully cope with such a common problem.

7.3 Kohonen Application - Pattern Recognition

7.3.1 Problems Associated with Kohonen Learning

The most useful property of the Kohonen network is that it has the ability to extract the statistical properties of the input data set. Kohonen showed [Koho88] that in a fully trained network the probability of a randomly selected input vector being closest to any given weight vector is *1/k*, where *k* is the number of Kohonen neurons. This, of course, is the optimal distribution of the weights on the hypersphere; this also assumes that all weight vectors are in use, which is a situation that can be difficult to realise.

As described in section 3.2.3.4 it is desirable to distribute weight vectors about the hypersphere according to the density of the input vectors that must be separated. As this is impractical to implement directly, requiring the placing of more weight vectors near a high density region of the input space, approximation techniques must be used.

The solution used in this pattern recognition application is to randomise all of the weight vectors and then normalise them. This will uniformly distribute the weight vectors around the hypersphere, but is unlikely to give an near-optimal distribution for reasons that have already been discussed. All input vectors are also normalised, which should help alleviate the problems caused by non-optimal hypersphere placement of the weight vectors. The randomisation and normalisation processes are not done on the systolic array processor, and should be carried out by the host computer before any neural processing is initiated.

7.3.2 Simulation Strategy

Due to the problems with weight distribution it was realised that the network may fail to train simply due to unfortunate initial values within the network. Therefore, several training runs were made with different initial values until a trained network was obtained, and it is this successful network that is discussed throughout the rest of section 7.3.

In order to test for convergence each of the input patterns in the training set are passed through the network and the winning neurons for each input are noted. Convergence is deemed to occur when each input pattern results in a different neuron firing.

7.3.3 Network Structure

The network structure for this application consists of a single layer of Kohonen neurons (n = 10), and this is shown in figure 7.10. It was anticipated that the weights within each neuron would align themselves to a single input pattern, firing only when that input pattern is present on the inputs to the neuron; this means that the network has to contain one neuron per input pattern. No neurons within the network are wasted, as each has an input pattern that they are expected to train upon.

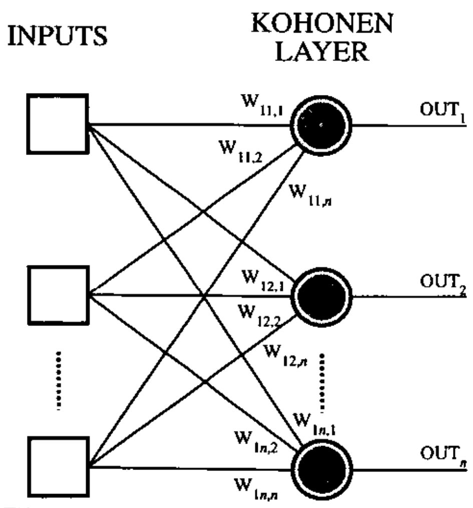


Figure 7.10 Kohonen Network

The input patterns are identical to those specified in section 7.2.3.2, although these inputs do not have an associated target vector. The values of the inputs themselves have been normalised, which converts each input into a unit vector pointing in the same direction as the original vector, using equation 3.10.

7.3.4 Simulation Run

Table 7.6 shows all of the training parameters used in the simulation run, along with the number of clock cycles required per iteration. The network trained after 3,121 iterations, which occurred after a number of failed training runs; note, this failure is a problem inherent in the Kohonen learning algorithm rather that a problem with the systolic simulator.

Table 7.6 Kohonen Training Parameters

Parameter	Value
Training Rate	$^{90}/_{128}$ (~0.7)
Max Iterations	10,000
Cycles / Iteration	2,190

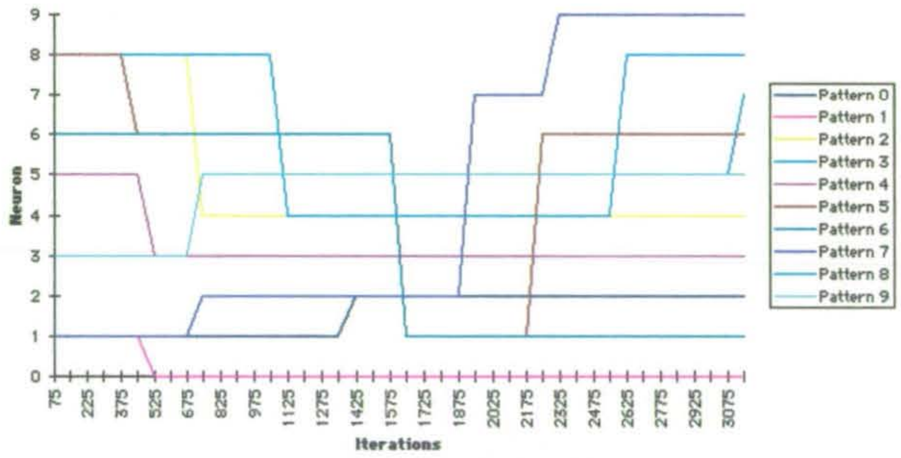


Figure 7.11 Kohonen Training Run Progress

A graph showing which neurons win for each input pattern is shown in figure 7.11. It shows how each neuron changes it's internal weights over the training run, with some neurons winning for multiple input patterns. At certain times during the training run some neurons do not win for any of the input patterns. During the course of the training run the network settles down, with each neuron in the network winning for just a single input pattern in the training set. At this point the network is deemed to have converged.

7.4 Counter Propagation Application - Pattern Recognition

7.4.1 Application Overview

The idea behind the counter propagation application is to be generate a network that is capable of differentiating between a series of images, each of which is allocated an index value. Upon presentation of this index value to the network (when the network is in *operational* mode rather than *training* mode) the actual image bitmap for the appropriate image is output. This identity mapping feature of the counter propagation network was previously discussed in more detail in section 3.3.1.2.

Although this bears a strong resemblance to the Kohonen application discussed in section 7.3 the Kohonen part of the network is different, as it has to take into account the index value inputs. Therefore, the Kohonen layer of the counter propagation network cannot use the weights from the Kohonen network from section 7.3.

7.4.2 Network Structure

The index value for the ten input patterns is just a simple binary code in the range 0...9. This will require four additional inputs to each of the neurons in the Kohonen layer, which will still consist of ten neurons. The Grossberg layer will require the same number of neurons as there are inputs to the Kohonen layer, as this is a requirement of the identity mapping process. The entire network details required for the counter propagation application is shown in table 7.7, which has not been shown in a figure for the sake of clarity; a detailed diagram of the layout of a counter propagation network was given in figure 3.13.

Table 7.7
Counter Propagation Network Structure

Parameter	Size
Network Inputs	19
Kohonen Neurons	10
Grossberg Neurons	19

The input patterns are identical to those specified in section 7.2.3.2, with the addition of the four index inputs. The values of the input vectors themselves have been normalised, which converts each input into a unit vector pointing in the same direction as the original vector, using equation 3.10.

7.3.3 Kohonen Simulation Run

Table 7.8 shows all of the training parameters used in the simulation run, along with the number of clock cycles required per iteration. It also shows the number of training iterations allowed before a test for convergence is made. The network trained after 5,117 iterations, which occurred after a number of failed training runs.

Table 7.8
Counter Propagation Kohonen Layer Training Parameters

Parameter	Value
Training Rate	$^{90}/_{128}$ (~0.7)
Max Iterations	10,000
Cycles / Iteration	2,790

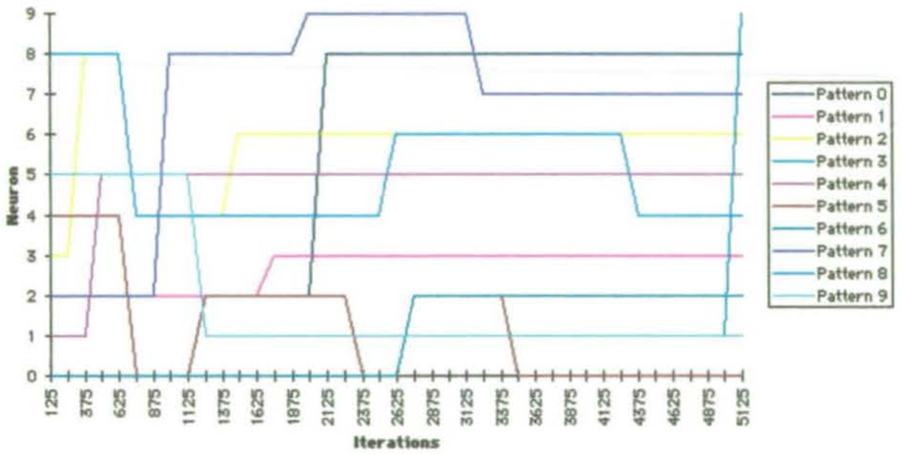


Figure 7.12 Counter Propagation Kohonen Layer Progress

A graph showing which neurons win for each input pattern is shown in figure 7.12. It shows how each neuron changes it's internal weights over the training run, with some neurons winning for multiple input patterns. Note that although the problem is similar to the that described in section 7.2 the neuron-pattern mapping shown in figure 7.12 is different from that shown in figure 7.11 However, the progress of the training run is fairly similar with regards to multiple-pattern winning neurons, which shows that the addition of some index variable inputs did not significantly alter the problem with respect to the Kohonen network. Indeed, it is still patterns 8 and 9 that are the last to converge, which indicates that the basic nature of the problem has remained unchanged.

7.3.4 Counter Propagation Simulation Run

Table 7.9 shows all the training parameters used in the simulation run, along with the number of clock cycles required per iteration. The training rate is gradually reduced over the training process, and this application reduced it every 500 iterations of the training algorithm; each rate reduction was by $^{1}/_{128}$. The network finally trained after 7,718 iterations, with a maximum pattern error of $^{13}/_{128}$ and maximum system error of $^{28}/_{128}$. At the point of convergence the training rate was equal to $^{6}/_{128}$.

Table 7.9
Counter Propagation Grossberg Layer Training Parameters

Parameter	**Value**
Training Rate	$^{13}/_{128}$ (~0.1)
Max System Error	$^{32}/_{128}$ (0.25)
Max Pattern Error	$^{13}/_{128}$ (~0.1)
Max Iterations	10,000
Rate Reduction	1,000 iterations
Reduce By	$^{1}/_{128}$ (~0.01)
Cycles / Iteration	2,210

The inputs to the Grossberg layer each have an associated target vector, which is the input to the Kohonen layer that created the Grossberg layer input. Before the training process for the Grossberg layer can begin a mapping between each Kohonen input to Grossberg input must be made in order to know what the target vector is for each neuron in the Grossberg layer. This is due to the unsupervised nature of the Kohonen algorithm, as there is no way to determine in advance of training which neuron in the Kohonen layer will win for each input vector. A graph showing the reducing pattern error over the training run is shown in figure 7.13.

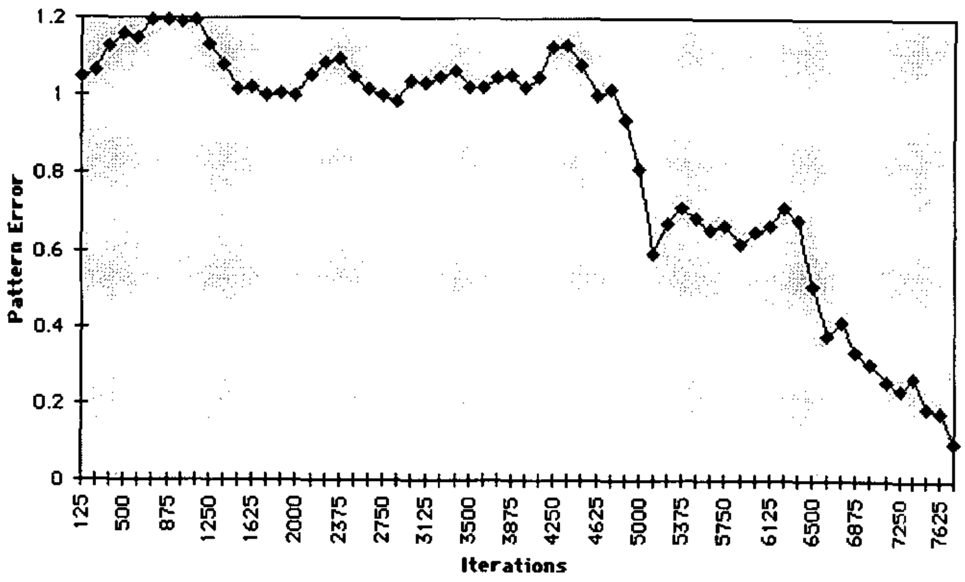


Figure 7.13 Counter Propagation Grossberg Layer Progress

8 SUMMARY AND FURTHER RESEARCH

INTRODUCTION

This thesis has introduced a number of new systolic algorithms for neural network learning, under the framework of being suitable for implementation on a custom designed two-dimensional systolic array processor. This final chapter summarises the work that has been presented in this thesis, discussing the more important aspects of the work in more detail. It concludes by briefly discussing some areas where further research and expansion on this work could prove fruitful.

8.1 Thesis Summary

The work presented in this thesis has previously been summarised in section 1.5 on a chapter by chapter basis. However, a re-iteration of this will help to highlight the achievements made during the course of this study. A different partitioning scheme is used throughout section 8.1 as compared to section 1.5, which may show the success of the work in a slightly different light.

8.1.1 Increasing Computational Capacity

Chapter 1 gave a general overview on the history of the field of computing, with special reference to the field of parallel architectures, whilst Chapter 2 described the techniques behind various fabrication methodologies, with the associated methods for testing digital logic, and also discussed systolic array architectures in more detail.

These two sections showed that there are, basically, two ways in which computer performance can be improved in terms of computational capacity. The first, and most costly, is to increase the efficiency of the VLSI fabrication process, by increasing circuit density, reducing feature sizes and minimising switching time. Although it seems that the ever-increasing performance of VLSI must eventually end, due to the inherent difficulties with the silicon process, IBM have recently introduced [IBM-98] a copper-tracked fabrication process, as opposed to aluminium-tracked, which has drastically reduced switching times for little additional fabrication cost; this does not take into account the massive non-recurrent research expenditure and the cost of re-tooling fabrication plants in order to cope with this fundamental change. Although silicon techniques are reaching the frontiers of their capacity there is still plenty of life left in the technique.

The other method for increasing computational capacity is to use some form of parallel architecture that employs multiple processing units working simultaneously on a single task. This may involve exploiting pipelining, utilising multiple identical PE's in a MIMD array or having a client-server approach based around a multiple-SIMD architecture where PE's are allocated to multiple tasks on demand; a mixture of these techniques is not unusual on the same device, with the TMS320C8x family of digital signal processing chips from Texas Instruments being configurable in several different parallel methodologies [TI-95].

In order to maximize the computational capacity of any device the design would have to keep track of current fabrication technology in order to maximize the exploitation of silicon, as well as embrace parallelism as much as possible; the chip architecture presented in this thesis encompasses both of these areas. The parallel array architecture used provides a reconfigurable array that allows MIMD, multiple-SIMD and isolated SIMD paradigms. The use of a re-programmable instruction systolic architecture keeps the instruction set per PE very small and efficient, with all instructions completing within a single cycle. By choosing to implement all of the silicon design in VHDL the design is, effectively, fabrication process independent - all that is required is to keep pace with advances in silicon (or other) fabrication is that a fabrication process has an efficient VHDL compiler for the design to be implemented on. VHDL compiler libraries are now released in much the same way that full- and semi-custom silicon CAD packages release libraries for new fabrication processes. Although some re-design work may be required in order to fully optimize a VHDL design for a particular process this is not as time consuming as re-designing the whole system. Because of this a VHDL design can always be implemented using the most up-to-date fabrication methodologies.

8.1.2 Hardware Implementation of Neural Networks

Chapter 3 gave a general history and overview of the neural network learning methodologies, discussing various algorithms and some aspects of the biological neural system upon which all of the field is based upon. Chapter 4 went on to describe some practical implementations of neural networks, both in software and hardware VLSI systems.

These two sections describe the wide variety of possible implementations of neural networks, some of which bear more of a resemblance to current models of the biological nervous system than others. The learning algorithms can be split into two distinct camps: supervised and unsupervised. Both types have their own relative advantages and disadvantages, both in terms of functionality and reliability. For example, although the back-propagation algorithm can be proven to be able to distinguish between all members of a training set it relies on infinitesimally small adjustments to the neural weights between passes. This is obviously impractical, so larger steps must be made in order to make an implementation possible; this, however, leads to the possibility of the learning process entering a state of paralysis, being unable to learn beyond a certain point. Unfortunately, this learning algorithm is by far the most frequently occurring in industry, mainly due to its simplicity - implementations of it are all too often far from perfect.

It would seem that for a closed system, where the neural network must operate in a real-time environment and modify its behaviour to suit, an unsupervised network would be the way forward. In an embedded system the network would be able to recognise when its environment had changed substantially and would be able to re-learn (or update) its environment. This would require some external network supervision and the use of a data dictionary in order to recognise when the environment had changed, but the network itself would be entirely responsible for the learning process. Unfortunately, unsupervised learning algorithms are not suited to all problems, and the level of suitability can only be discovered through experimentation.

One consideration that must be taken into account when designing a neural network system is speed: software is slow, hardware is fast. Unfortunately, hardware solutions tend to be fixed to a specific learning methodology and also have practical limits on the number of artificial neurons that they contain. Software solutions can be extended to, modified or just plain replaced as time goes by, but a standard hardware solution cannot be modified at all. However, the speed of a hardware system is often without comparison; whilst a software system may be able to carry out a particular computation in a few milli-seconds, a logically equivalent hardware system could carry it out in micro-seconds. Some compromise between hardware and software must be sought.

A MIMD parallel architecture implemented in hardware using VHDL, with each PE in the device having re-programmable micro-code, would allow a software neural network to be implemented in high-speed hardware. With the PE's being optimised for matrix-vector multiplications the neural algorithms will be able to take full advantage of the performance of its hardware environment whilst retaining the flexibility of a software-based system.

8.1.3 Practical Advantages of Study

A number of practical advantages in the field of hardware-based neural networks have been gained through the work presented in this study. The choice of architecture for the neural networks in itself as described in the previous two sections is a major benefit in its own right, as it could allow low-cost high-performance hardware neural networks, so will not be discussed in any more detail here. However, the other major benefits of this study will now be looked at in a little more detail.

i) Reprogrammable Instruction Sets

The micro-code within each of the PE's on the systolic array device can be re-programmed to suit any particular task, be it neural network based or otherwise. This vastly increases the scope of the architecture as it is not fixed to just a single learning paradigm - many different training methods can be employed on a single hardware device.

The novel feature of allowing each PE to be uniquely programmable, in that each PE can have an entirely different micro-code set from any other PE in the device, further enhances this re-programmability, as the restriction of allowing only 16 instructions per PE is greatly alleviated. In this manner more complex algorithms could be implemented on the architecture, even though they may require many more than 16 unique instructions.

ii) Multiple Result Outputs

The result from an operation within any PE can be sent in many directions; to the PE to the east, to the south, to the local registers or to a combination of the three. This allows the intelligent routing of partial results around the systolic array, implying that part of the array can be used as the main processing path for an operation yet other PE's within the array can be utilised in calculating partial results that may be required later in a calculation.

As each row and column of PE's within the systolic array can be processing at any given clock cycle it is possible for every row of the array to output a result to the outside world on every clock cycle. This allows for multiple calculations to be carried out at any time, albeit with the restriction that they most likely have to share some common data. This facility to have a single instruction/data input stream generate multiple results further increases the processing capacity for the architecture. Unfortunately, the problem with all such parallel architectures remains, in that in order to take full advantage of this type of multi-processing the software designer must pay close attention whilst implementing any algorithm so that it fully utilises such features.

iii) Single Cycle Multiply-and-Accumulate

One of the most common operations for many neural network learning paradigms is the matrix-vector multiplication operation. This consists of a repeated multiply-and-accumulate instructions (MAC) for each element of the result vector, and this operation is one of the most highly suited to implementation in a systolic array.

As it was likely to be an integral part to any hardware implementation of a neural network the systolic array architecture described in this study was designed such that a MAC operation could be carried out simultaneously in every PE in just a single clock cycle. Although the architecture is far from unique in this respect it was fundamental in the design of the architecture, as without a single cycle MAC operation the architecture may not have been viable.

iv) Function Unit Configurability

The three functional units within each PE, the comparator, the adder and the multiplier, had few restrictions on the source of either of their two inputs or of the destination for their result output; i.e. an instruction could use any of the functional units in practically any combination. This increases the flexibility of each of the 16 operations possible within each of the PE's, as each operation can be made fairly complex with respect to the calculation carried out and on the destination of the result.

Both the multiplier and the adder contain in-built saturation of the results, whereby any result generated is replaced by either the maximum or minimum value representable within a PE if the result has overflowed or underflowed. Results can be further restricted between a range of two numbers, programmable on a PE-by-PE basis, and this limit can be enabled or disabled per instruction. Provision of both saturation and conditional range limiting allows an algorithm to keep close track of a calculation and ensure that limiting values are never exceeded.

v) Program Independent Operation

Although implicit in the architecture of a systolic array device a software program running on the architecture requires no external controlling influence regarding the internal operation of the program. The external controller provides only data values and an algorithm, and the algorithm itself is fixed in advance; the non-existence of internal loops, as is normal in systolic arrays, implies that such control is not required. Hence, the complexity of the external controller does not have to be too advanced.

8.2 Further Work

There are a number of enhancements to the architecture presented in this study that could increase the complexity of problems that the device is able to handle. These enhancements, although potentially fruitful, were not fully investigated in this study as they were considered secondary in importance to finding an architecture that suited the soft-systolic implementation of neural networks. However, if any work were carried out in the future that might lead to the fabrication of a VLSI device then these ideas should be investigated more fully, as their incorporation into the architecture presented here would increase the commercial viability of any devices produced.

The first of these, the increase in capacity of the on-board PE micro-code, is a benefit in its own right. However, the other enhancements proposed in this section would not be possible unless this increase was made. Hence, although the increase itself is not a great technology feat, it is discussed first in order to show other benefits that would arise from it.

8.2.1 Increased Micro-code Capacity

It would be advantageous to increase both the depth and breadth of the capacity of the PE micro-code. This would increase the complexity of the existing instructions, as well as increase the number of instructions as a whole.

One of the main reasons for not carrying out this work at this stage is that the current functional units within a PE do not require a more complex instruction word, nor do the neural systolic algorithms require any more complexity. Another reason is that with 12-bits of data and 4-bits of instruction code the input and output for each row of the systolic array was 16-bits - this was very convenient in the design of the neural systolic algorithms, and also served to keep the number of external device I/O pins to a manageable number.

Other neural network training methodologies that may benefit from hardware implementation, such as the Hopfield network or the ART paradigms, are more complex than those implemented in this work. Practical implementations would probably require a larger and more varied instruction set than that implemented here.

The field of signal processing, which now encompasses the fields of audio and visual processing, also benefits from hardware implementations. Some of the popular algorithms, such as the discrete cosine transforms used in many image compression techniques, are possible on the systolic architecture, but are only really plausible when using one dimension of data; such techniques require two-dimensional data as input and these would not be possible on the architecture unless the micro-code was increased in both depth and breadth.

8.2.2 Index Registers

The four registers within each of the PE's can be used for either instruction source data or result destination data. Only a single register is ever active during one clock cycle, but this register can be read from and written to in the same cycle. During the operations used in the systolic algorithms that have been developed there has only been the need to access a single register per clock cycle, with the active register being selected via the [SWITCH] operation. However, this operation has the drawback in that all PE's in the same column process this operation.

Hence, it may beneficial to any algorithms developed in the future for the systolic architecture to include the facility to allow an instruction op-code to specify both the source and destination register that it requires. This may be hard-wired into the micro-code itself, such that a particular instruction always uses the same source and destination register pair. It could also be linked in to a set of multiplexors and temporary store so that the result of the previous instruction selects the register pair for the next instruction.

As well as allowing for the implementation of more complex algorithms this enhancement to the systolic architecture would also speed up some existing algorithms - they would not have to be liberally sprinkled with [SWITCH] instructions. Although this may only save one cycle per pass of an algorithm stage these savings will build up, eventually amounting to a saving of a few percent.

8.2.3 Conditional Instructions

In the systolic architecture all instructions are processed as the PE receives them on its North-South input (unless it is locked). There is no capacity for having any functional units within the PE operate only if a value is less than, greater than, equal to or not equal to any particular value, as is possible in more conventional processors.

It would be advantageous to some algorithms to conditionally transform some operations to [PASS] instructions dependant on some other PE input condition. This could be linked directly to the comparator unit, so that an addition (and result storage) only occurred if some externally requested condition were met. It may also be possible to split an instruction so that a choice between two operations occurred; i.e. a multiply if the condition was met and an addition if the condition was not met.

Conditional instructions, however, introduce an element of uncertainty into the systolic architecture and into the algorithms that run on it. The external device controller would no longer be absolutely certain how long a particular stage of an algorithm will take to execute if it included conditional instructions. Care would need to be taken with the external controller, as it may have to modify its instruction stream in order to react to a conditional instruction, whereas without conditional instructions the external controller does not have to react to any results in this manner. However, as long as sufficient care is taken at the algorithm design stage the use of conditional instructions could prove very beneficial to some algorithms, and this could lead to extremely powerful algorithms being implemented in relatively inexpensive VLSI devices.

8.2.4 Multiple Processing Paths

The functional units within each PE are, effectively, independent of one another, although they all share a common final output port. There is no technical reason why the results from the functional units cannot be diverted along different output paths; e.g. the adder result goes to the West-East port, the multiplier result goes to the North-South port and the comparator result goes into some destination index register.

This complicates the internal PE architecture and the micro-code by a large degree, although the bulk of the re-design effort would be in the internal data routing, multiplexing and control rather than in the functional units themselves (which would remain virtually untouched). Other internal units, such as the result range limiter, may have to be duplicated around each functional unit rather than act as a shared common resource.

This enhancement would vastly increase the computational power of the systolic architecture, but it would greatly complicate the algorithm design process. The combination of conditional instructions linked with multiple processing paths would make the timing calculations for an algorithm stage rather complex and, unfortunately, extremely critical. However, the rather large amount of effort required to implement this is likely to be justified given the improvement in raw processing power that it would produce.

8.3 Final Comments

Since the development of the digital computer the technology associated with it has evolved at an extremely rapid rate. However, although performance has improved tremendously in the last fifty years the demands of the end-user have increased also. Engineers, scientists and researchers are constantly striving to both improve computing performance whilst still reducing the overall cost of the computer system.

Systolic array architectures are a very powerful approach to exploiting massive parallelism with an absolute minimum of communication overheads. Due to their regularity of structure and heavy use of local communication, they are very amenable to VLSI implementation. Drawbacks with such implementations, notably the prohibitive cost and the inflexibility of some fixed hardware solutions, have led to systolic arrays VLSI implementations to be few and far between. By using VHDL as the target hardware description language, and testing a design before final fabrication using inexpensive field-programmable gate array devices, the cost aspect has become less of a burden. Also, by having a systolic array that is completely re-programmable the systolic designs can be made extremely flexible.

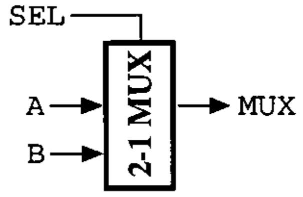
This thesis has shown that systolic arrays are capable of handling computationally intensive applications and that they are a powerful form of massively parallel processing. It also has shown that the field of artificial neural networks can benefit hugely from implementation in hardware. The work presented in this thesis has shown that systolic arrays can be adapted to implement neural networks very effectively. When this implementation is in a re-programmable VLSI device its performance surpasses any software system and is on a par with, if not superior to, any other fixed hardware implementation of an equivalent neural algorithm.

A CIRCUIT DESIGNS AND VHDL CODE

A.1 Basic Circuit Elements

A.1.1 2-to-1 Multiplexor

This is a data selector. It takes in two separate 1-bit data values, A and B, and outputs one of the them on the output line OUT after a short delay. The signal to be output is chosen based on the value of the third input SEL. As the circuit is not clocked in any way it's output is dependent on changes on all three input signals. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.1.



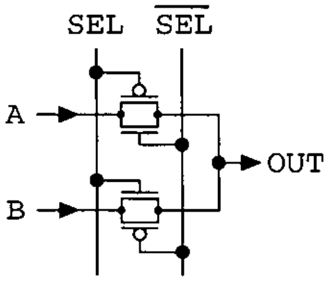


Figure A.1
2-to-1 Multiplexor

```
--------------------
-- SEL: MUX switch
-- A,B: Input values
-- OUT: Output value
--------------------

entity MUX_2 is
   port (SEL:  in  BIT;
         A, B: in  BIT;
         OUT:  out BIT);
end MUX_2;

architecture MUX_UNIT_2 of MUX_2 is
begin
   process (SEL, A, B)
      variable TEMP: BIT;
   begin
      case SEL is
         when '0' => TEMP := A;
         when '1' => TEMP := B;
      end case;
      OUT <= TEMP after 500 ps;
   end process;
end MUX_UNIT_2;
```

A.1.2 4-to-1 Multiplexor

This is a data selector. It takes in four separate 1-bit data values, A, B, C and D, and outputs one of the them on the output line OUT after a short delay. The signal to be output is chosen based on the value of the fifth input SEL(0:1). As the circuit is not clocked in any way it's output is dependent on changes on all five input signals. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.2.

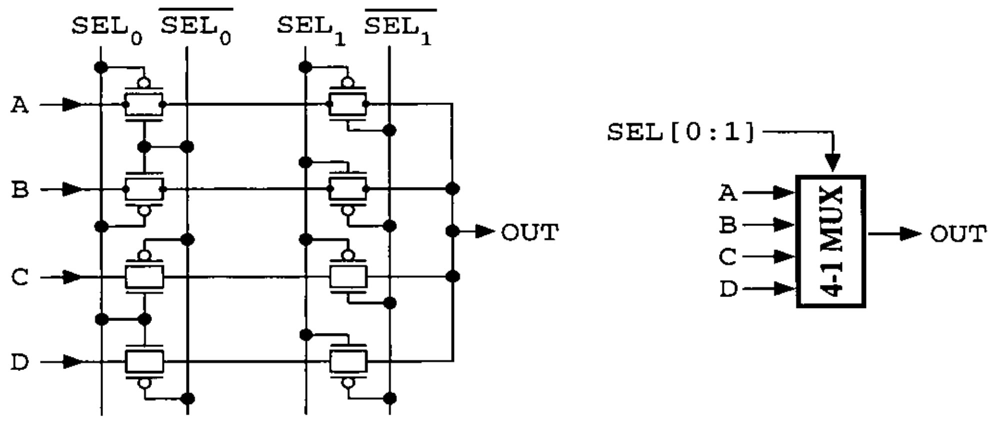


Figure A.2 4-to-1 Multiplexor

```
------------------------
-- SEL(2):  UX switches
-- A,B,C,D: Input values
-- OUT:     Output value
------------------------

entity MUX_4 is
   port (SEL:        in  BIT_VECTOR(0 to 1);
         A, B, C, D: in  BIT;
         OUT:        out BIT);
end MUX_4;

architecture MUX_UNIT_4 of MUX_4 is
   process (SEL, A, B, C, D)
      variable TEMP: BIT;
   begin
      case SEL is
         when "00" => TEMP := A;
         when "01" => TEMP := B;
         when "10" => TEMP := C;
         when "11" => TEMP := D;
      end case;
      OUT <= TEMP after 1 ns;
   end process;
end MUX_UNIT_4;
```

A.1.3 D-Type Flip-Flop

This is a 1-bit data store. Upon a CLOCK event if the CLOCK is high then the data on the input data D is read and stored in the internal store Q after a short propagation delay. The value stored in Q is always available at the output of the flip-flop. The circuitry within the shaded area is a standard S-R flip-flop. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.3 - note that the simple VHDL code, which is the standard definition of a D-type flip-flop, belies the complexity of the circuit.

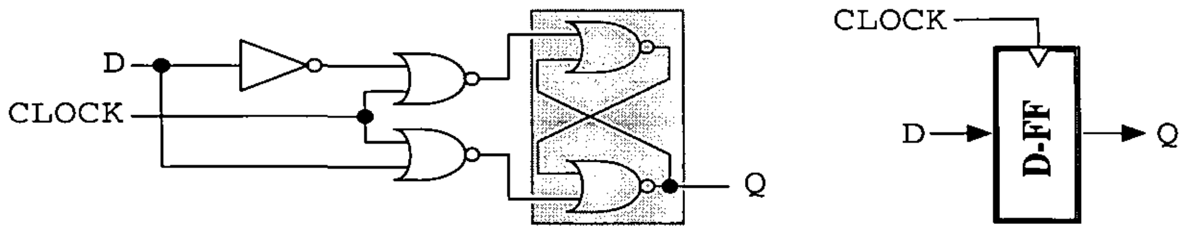


Figure A.3 D-Type Flip-Flop

```
--------------------------
-- CLOCK: System clock
-- D:     Input data
-- Q:     Output data
--------------------------

entity D_FLIP_FLOP is
   port (CLOCK, D: in  BIT;
         Q:        out BIT)
end D_FLIP_FLOP;

architecture SYNC_DFF of D_FLIP_FLOP is
   process (CLOCK)
   begin
      if ((CLOCK = '1') and CLOCK'EVENT) then
         Q <= D after 2 ns;
      end if;
   end process;
end SYNC_DFF;
```

A.1.4 D-Type Flip-Flop with Reset

This is a 1-bit data store. Upon a CLOCK event if the CLOCK is high then some data is stored in the internal store Q; if the input RESET is not set then the data from the input D is stored after a short propagation delay, else a logic-0 is stored after an even shorter delay (as the value is sourced directly from the power supplies) - in an S-R flip flop the input pair "01" guarantees the output to be zero regardless of it's previous value. The value stored in Q is always available at the output of the flip-flop. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.4.

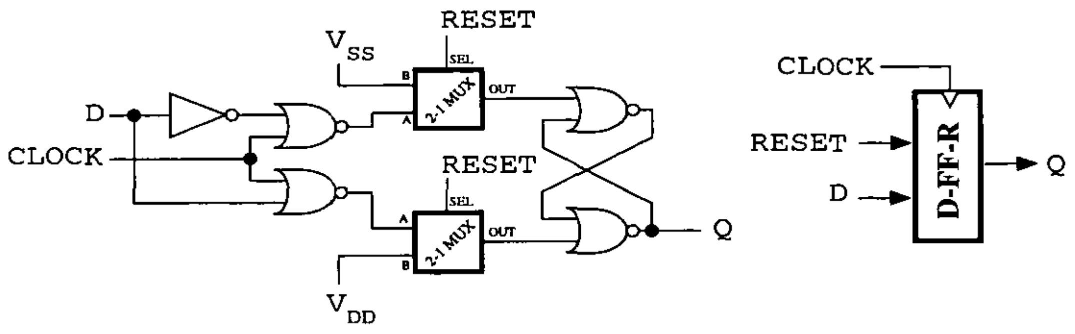
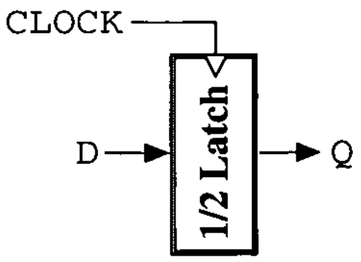


Figure A.4 D-Type Flip-Flop with Instant Reset

```
--------------------------
-- CLOCK: System clock
-- RESET: Hard reset
-- D:     Input data
-- Q:     Output data
--------------------------

entity D_FLIP_FLOP_RES is
   port (CLOCK:     in  BIT;
         RESET, D: in  BIT;
         Q:         out BIT)
end D_FLIP_FLOP_RES;

architecture SYNC_DFF_R of D_FLIP_FLOP is
   process (CLOCK)
   begin
      if ((CLOCK = '1') and CLOCK'EVENT) then
         if (RESET = '0') then
            Q <= D after 2 ns;
         else
            Q <= 0 after 500 ps;
         end if;
      end if;
   end process;
end SYNC_DFF_R;
```

A.1.5 Half Latch

This is a pseudo- 1-bit data store. Any data on the input line D is stored in the internal value S so long as the CLOCK signal is both high and stable (as indicated by the internal signal CHS). The process is dependent on a change in either D or CLOCK. The negation of the stored value S is made available at the circuit output Q at all times. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.5. Note, this circuit does not take into account power leakage at the capacitor C - as long as the clock phase in use is regular and has a period of 100ns or less then the effect of the leakage should be negligible.



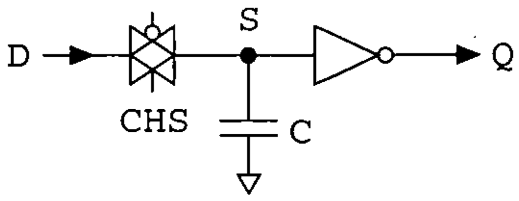


Figure A.5 Half Latch

```
---------------------
-- CLOCK: Clock phase
-- D:     Input data
-- Q:     Output data
---------------------

entity HALF_LATCH is
   port (CLOCK, D: in  BIT;
         Q:        out BIT)
end HALF_LATCH;

architecture TRANSP_HALF_LATCH of HALF_LATCH is
   process (CLOCK, D)
      signal S: BIT;
   begin
      if ((CLOCK = '1') AND CLOCK'STABLE) then
         S <= D after 500 ps;
      end if;
      Q <= not S after 500 ps;
   end process;
end TRANSP_HALF_LATCH;
```

A.1.6 3-Input Majority Function

This circuit takes three binary inputs, A, B and C, producing a single value RES. If at least two inputs are set high then RES is also high, whereas if at most one input is set high then RES is set low. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.6.

```
----------------------
-- A,B,C: Input Values
-- RES:   Output Value
----------------------

entity MAJORITY is
   port (A,B,C: in  BIT,
         RES:   out BIT);
end MAJORITY;

architecture MAJORITY_3 of MAJORITY is
   component NAND2
```

```
      port (A, B: in BIT; RES: out BIT);
   end component;
   component OR2
      port (A, B: in BIT; RES: out BIT);
   end component;
   signal T1,T2,T3: BIT;
begin
   N1: NAND2(A, B, T1);
   O1: OR2(A, B, T2);
   N2: NAND2(C, T2, T3);
   N3: NAND2(T1, T3, RES);
end MAJORITY_3;
```

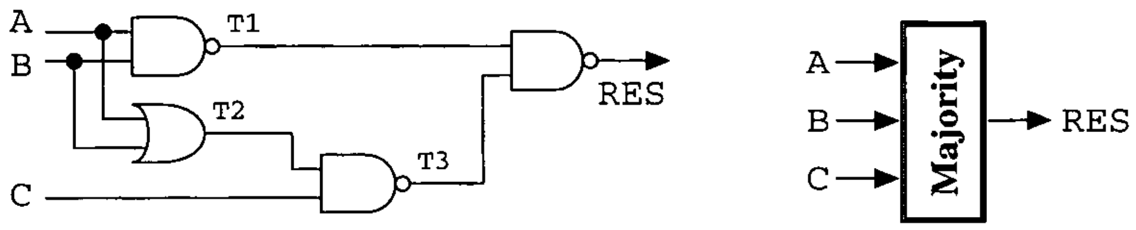


Figure A.6 3-Input Majority Function

A.1.7 Negative Clock Open Latch

This is a 1-bit data store. Throughout the stable negative edge of the CLOCK input (as indicated by the internal signal CLS) the data present on the input D is stored within the cell. The contents of the store are always available at the output Q, although newly stored inputs are not output until the following clock cycles (implying that read/write in a single cycle is possible) Upon the stable positive edge of the CLOCK input (as indicated by the internal signal CHS) the data within the cell cannot be altered and is cycled around the circuit. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.7.

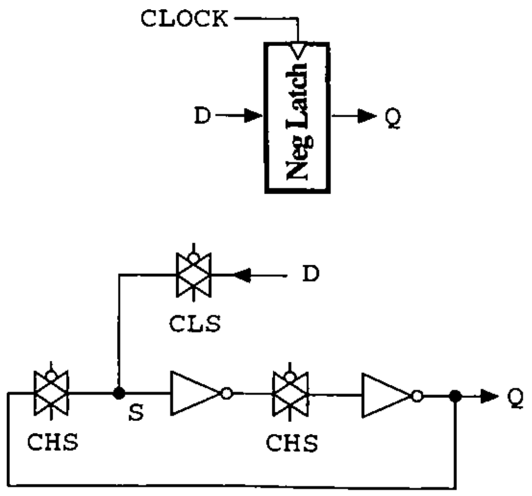


Figure A.7 Negative Edge Open Latch

```
----------------------
-- CLOCK: System Clock
-- D:     Input data
-- Q:     Output data
----------------------

entity NEG_OPEN_LATCH is
   port (CLOCK, D: in  BIT;
         Q:        out BIT)
end NEG_OPEN_LATCH;

architecture NEG_LATCH of
NEG_OPEN_LATCH is
   process (CLOCK, D)
   begin
      if (CLOCK'STABLE) then
         if (CLOCK = '0') then
            S <= D after 500 ps;
         else
            S <= Q after 500 ps;
         end if;
      end if;
      Q <= S after 1 ns;
   end process;
end NEG_LATCH;
```

A.2 Data Storage Units

A.2.1 PE Input Register

This is a 16-bit data storage register. It is used to hold the operation code and data input to the PE from all four cell edges. It stores the data on the input IP on the rising edge of the clock in a series of D-type flip-flops. This data is made available at all times on the output OP. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.8.

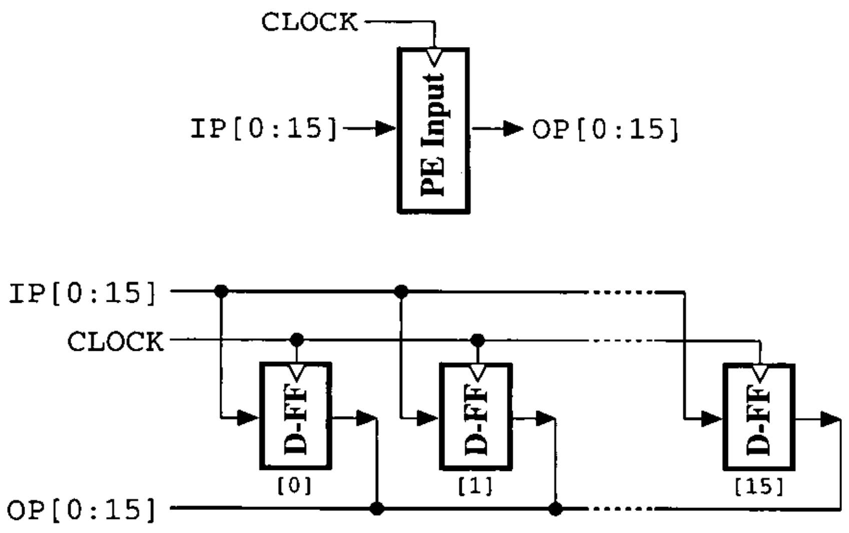
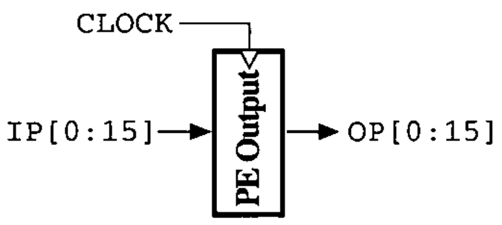


Figure A.8 PE Input Register

```
-----------------------
-- CLOCK:  System clock
-- IP(16):  Input data
-- OP(16): Output data
-----------------------

entity PE_INPUT_REG is
   generic (START: INTEGER := 0; STOP : INTEGER := 15);
   port (CLOCK: in  BIT;
         IP:    in  BIT_VECTOR (START to STOP);
         OP:     out BIT_VECTOR (START to STOP));
end PE_INPUT_REG;

architecture OP_INPUT_REG of PE_INPUT_REG is
   component D_FLIP_FLOP
      port (CLOCK, D: in IT; Q: out BIT);
   end component;
begin
   process (CLOCK)
   begin
      INST_ALL: for K in START to STOP generate
         DFF: D_FLIP_FLOP port map (CLOCK, IP(K), OP(K));
      end generate INST_ALL;
   end process;
end OP_INPUT_REG;
```

A.2.2 PE Output Register

This is a 16-bit data storage register. It is used to hold an operation code and a data value to be output from the PE from all four cell edges. It stores the data on the input IP throughout the negative edge of the clock in a series of negative edge open latches - these are used instead of the D-type flip-flops of the input register as the output of this register is the input to the other, so the date on the output OP must be stable before the rising edge of the clock. This data is made available at all times on the output OP. The process is dependent on the value of CLOCK as well as the value of IP. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.9.



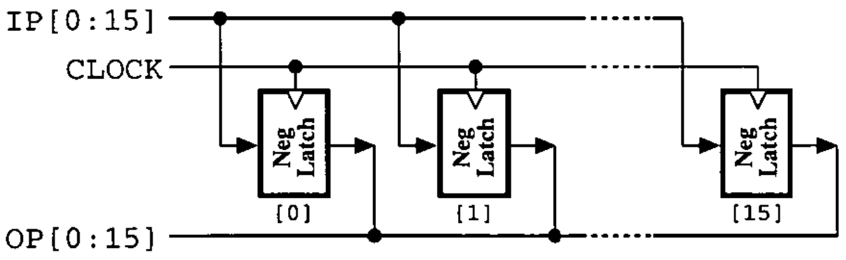


Figure A.9 PE Output Register

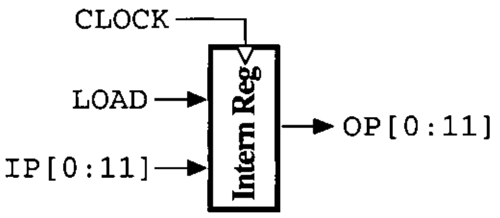
```
-----------------------
-- CLOCK:  System clock
-- IP(16): Input data
-- OP(16): Output data
-----------------------

entity PE_OUTPUT_REG is
   generic (START: INTEGER := 0; STOP : INTEGER := 15);
   port (CLOCK: in  BIT;
         IP:    in  BIT_VECTOR (START to STOP);
         OP:    out BIT_VECTOR (START to STOP));
end PE_OUTPUT_REG;

architecture OP_OUTPUT_REG of PE_OUTPUT_REG is
   component NEG_OPEN_LATCH
      port (CLOCK, D: in BIT; Q: out BIT);
   end component;
begin
   process (CLOCK, IP)
   begin
      INST_ALL: for K in START to STOP generate
         DFF: NEG_OPEN_LATCH port map (CLOCK, IP(K), OP(K));
      end generate INST_ALL;
   end process;
end OP_OUTPUT_REG;
```

A.2.3 PE Internal Register Block

A.2.3.1 Register Unit

This is a 12-bit data storage register. It is used to hold a data value within the PE for use in the execution of any current or future instructions. It uses a series of negative-edge latches to store data on the input IP internally when the CLOCK is low and the LOAD signal is high. The data within the register is available at all times on the output OP. The process is dependent on the value of CLOCK as well as the value of IP; it is not dependent on LOAD as this signal is stable throughout a clock cycle, changing only at the beginning of a clock cycle in response to a new instruction code. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.10.

```
-----------------------
-- CLOCK:  System clock
-- IP(12): Input data
-- OP(12): Output data
-----------------------

entity INTERNAL_REG is
   generic (START: INTEGER := 0; STOP : INTEGER := 11);
   port (CLOCK: in BIT;
         LOAD:  in BIT
         IP:    in  BIT_VECTOR (START to STOP);
         OP:    out BIT_VECTOR (START to STOP));
end INTERNAL_REG;

architecture PE_INTERNAL_REG of INTERNAL_REG is
   component NEG_OPEN_LATCH
      port (CLOCK, D: in BIT; Q: out BIT);
   end component;
   signal MOD_CLOCK: BIT;
begin
   process (CLOCK, IP)
   begin
      MOD_CLOCK <= (not CLOCK) nand LOAD after 1 ns;
      wait on MOD_CLOCK;
      INST_ALL: for K in START to STOP generate
         DFF: NEG_OPEN_LATCH port map (MOD_CLOCK, IP(K), OP(K));
      end generate INST_ALL;
   end process;
end PE_INTERNAL_REG;
```



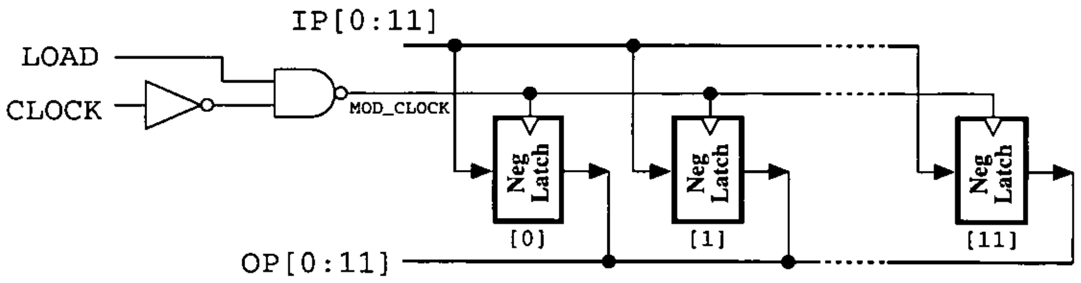


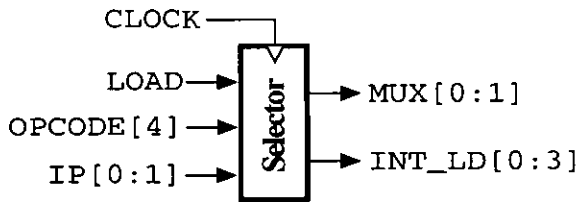
Figure A.10 Register Block Internal Register

A.2.3.2 Active Register Selector

This is an internal control signal generator and data store. The selector holds a 2-bit value to indicate which of the four internal registers is currently active; the *load* signals on the registers and an output multiplexor block takes data from the selector. Data on IP is stored in the internal registers whenever a [SWITCH] instruction is received by the PE. The outputs from the regusters are available on OP at all times. These signals are later sent directly to the multiplexor block without any further processing. They are also combined with the LOAD input to ensure that only the active register receives any external LOAD instruction. The VHDL code for this circuit is shown below (minus the combinational logic for the generation of the internal signal SWITCH_OP), and a schematic of the circuit itself is shown in figure A.11.

```
---------------------------------
-- CLOCK:      System Clock
-- IP[2]:      Current input data
-- LOAD:       RegLoad-Op received
-- OPCODE[4]: Current Opcode
-- MUX[2]:     Output MUX signals
-- INT_LD[4]: Load output signals
---------------------------------

entity REG_SELECT is
   port (CLOCK:  in  BIT;
         LOAD:   in  BIT;
         IP:     in  BIT_VECTOR (0 to 1);
         OPCODE: in  BIT_VECTOR (0 to 3);
         MUX:    out BIT_VECTOR (0 to 1);
         INT_LD: out BIT_VECTOR (0 to 3));
end REG_SELECT;

architecture PE_REG_SELECT of REG_SELECT
   component INTERNAL_REG
      generic (START, STOP: INTEGER)
      port (CLOCK, LOAD: in BIT: IP: in BIT_VECTOR (START to STOP);
            IO: out BIT_VECTOR(START to STOP));
   end component;
   component MUX_2
      port (SEL, A, B: in BIT; OUT: out BIT);
   end component;
   signal SELECT: BIT_VECTOR (0 to 1);
   signal T1, T2,. SWITCH_OP: BIT;
begin
   --------
   -- See if current opcode is [SWITCH]
   --------
   T1 <= (not OPCODE(0)) nor OPCODE(1);
   T2 <= OPCODE(2) nor OPCODE(3);
   SWITCH_OP <= T1 and T2;

   --------
   -- Store new 'reg-no' in register
   --------
   LTC: INTERNAL_REG generic map (0, 1)
                     port map (CLOCK, SWITCH_OP, IP, SELECT);

   --------
   -- Generate reg-block load signals
   --------
   MX0: MUX_2 port map (LOAD,0,SELECT(0) nor SELECT(1), INT_LD(0));
   MX1: MUX_2 port map (LOAD,0,SELECT(0) nor not(SELECT(1)), INT_LD(1));
   MX2: MUX_2 port map (LOAD,0,not(SELECT(0)) nor SELECT(1), INT_LD(2));
   MX3: MUX_2 port map (LOAD,0,SELECT(0) and SELECT(1), INT_LD(3));

   --------
   -- Copy output of flip-flops to multiplexor block
```

```
    --------
    MUX(0) <= SELECT(0);
    MUX(1) <= SELECT(1);
end PE_REG_SELECT;
```



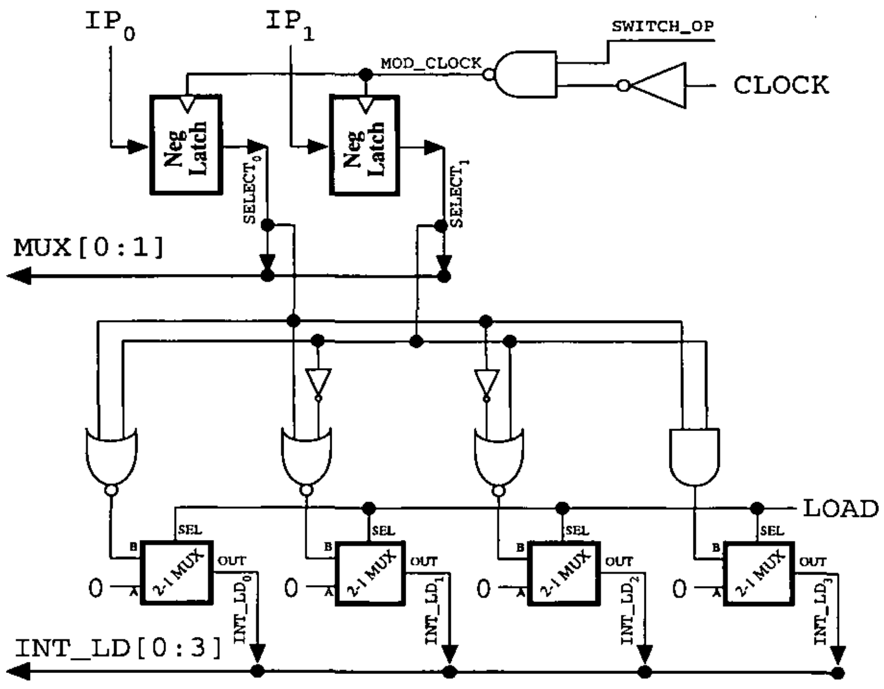


Figure A.11 Register Block 'Active' Selector

A.2.3.3 Main Register Block

This is a multiple storage unit with a multiplexed output. The internal PE register block consists of four 12-bit register units, as shown in figure A.10. Each register unit receives identical copies of the input data IP, but with individual instances of INT_LD, as only one register unit is designated as being 'active'. The 12-bit outputs from each of the register units are fed into 12 instances of a 4-to-1 multiplexor, arranged and controlled such that the 12-bit output from the register unit that is currently active is forwarded on out of the register block. The output OP(12) from the multiplexor block is available at all times. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.12.

```
---------------------------------------------
-- CLOCK:     System clock
-- IP(12):    Input data to be stored
-- INT_LD(4): Load signal for active register
-- SEL(2):    MUX control signal
-- OP(12):    Output of active register
---------------------------------------------

entity REG_STORAGE is
   port (CLOCK:  in  BIT;
         IP:     in  BIT_VECTOR(0 to 11);
         INT_LD: in  BIT_VECTOR(0 to 3);
         SEL:    in  BIT_VECTOR(0 to 1);
         OP:     out BIT_VECTOR(0 to 11));
end REG_STORAGE;

architecture PE_REG_STORAGE of REG_STORAGE
   component INTERNAL_REG
      generic (START, STOP: INTEGER);
      port (CLOCK, LOAD: in BIT; IP: in BIT_VECTOR(START to STOP);
            OP: out BIT_VECTOR(START to STOP));
   end component;
   component MUX_4
      port (SEL: in BIT_VECTOR(0 to 1); A, B, C, D: in BIT; OUT: out BIT);
   end component;
   type BLOCK_ADDR is array (0 to 3, 0 to 11) of BIT;
   signal REG_OUT: BLOCK_ADDR:
begin
   --------
   -- Generate the register block
   --------
   REG_BLK: for K in 0 to 3 generate
      RG1: INTERNAL_REG generic map (0, 3)
                        port map (CLOCK, INT_LD(K), IP, REG_OUT(K));
   end generate REG_BLK;

   --------
   -- Multiplex the output
   --------
   GEN_MUX: for K in 0 to 11 generate
      MX1: MUX_4 port map (SEL, REG_OUT(0)(K), REG_OUT(1)(K),
            REG_OUT(2)(K), REG_OUT(3)(K), OP(K));
   end generate GEN_MUX;
end PE REG_STORAGE;
```

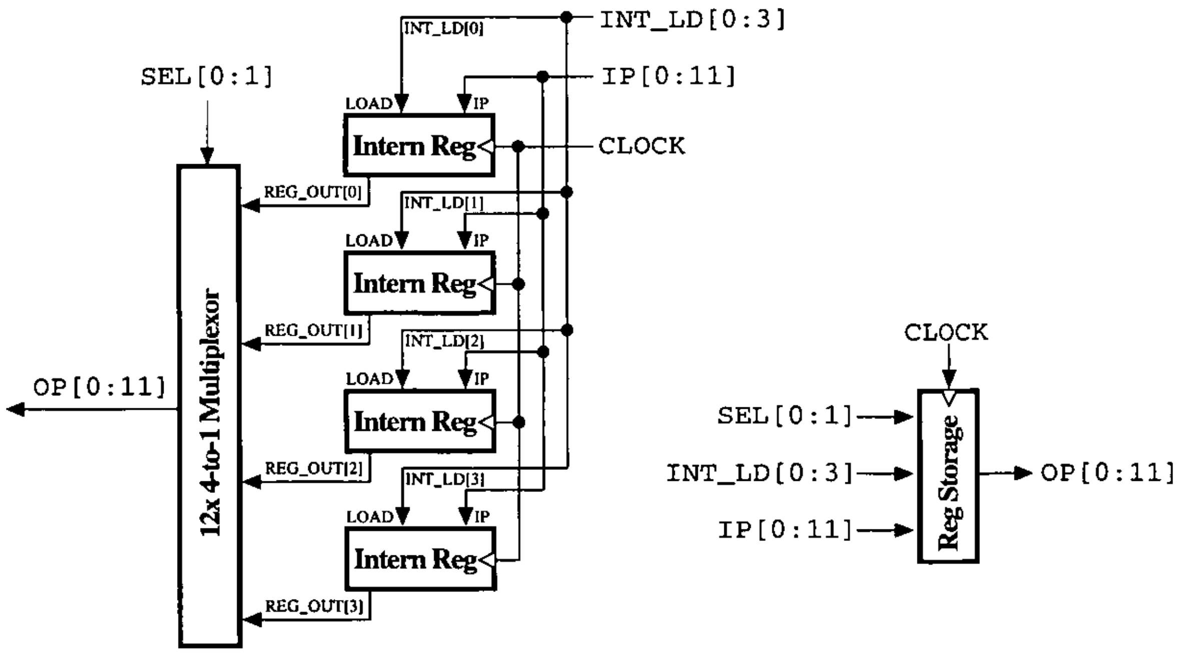


Figure A.12 Register Block Storage

A.2.3.4 Complete Register Block

This is the entire register storage block within a single PE. It links together the units described in both sections A.2.3.2 and A.2.3.3. It adds no additional functionality, save for providing a single external interface for all signals. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.13.

```
---------------------------------------
-- CLOCK:      System Clock
-- LOAD:       Data load required
-- OPCODE[4]:  Switch-Op being processed
-- IP[12]:     Input data/switch index
-- OP[12]:     Output of active register
---------------------------------------

entity REGISTER_BLOCK is
   port (CLOCK:  in  BIT;
         LOAD:   in  BIT;
         OPCODE: in  BIT_VECTOR(0 to 3);
         IP:     in  BIT_VECTOR(0 to 11);
         OP:     out BIT_VECTOR(0 to 11));
end REGISTER_BLOCK;
```

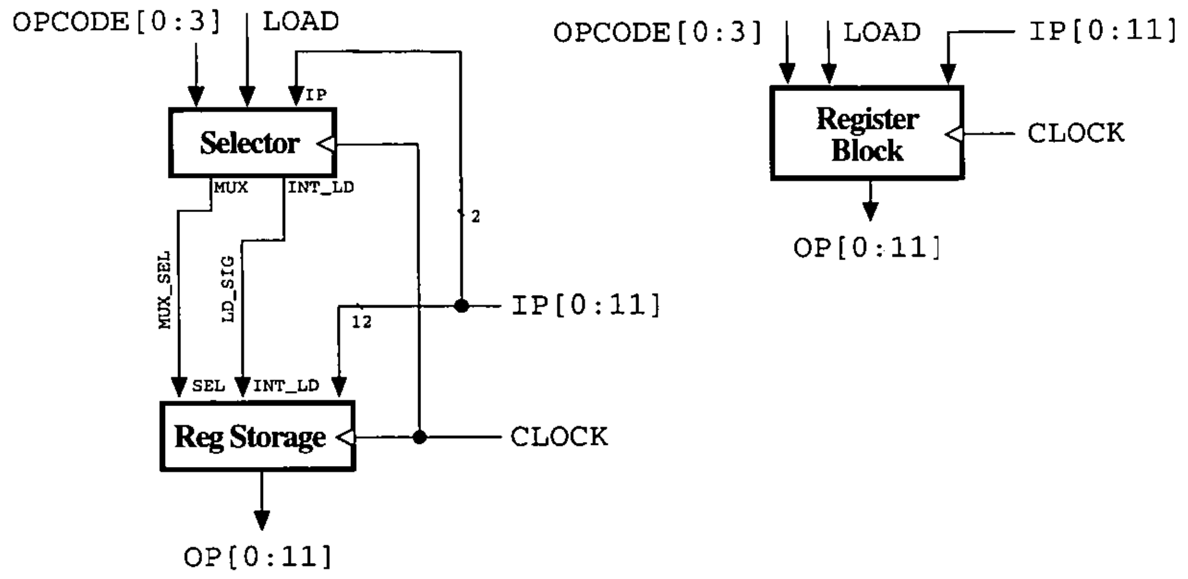


Figure A.13 Entire Internal Register Block

```
architecture PE_REGISTER_BLOCK
   component REG_SELECT
      port (CLOCK, LOAD: in BIT; IP: in BIT_VECTOR(0 to 1),
            OPCODE: in BIT_VECTOR(0 to 3); MUX: out BIT_VECTOR(0 to 1);
            INT_LD: out BIT_VECTOR(0 to 3));
   end component;
   component REG_STORAGE
      port (CLOCK: in BIT; IP: in BIT_VECTOR(0 to 11);
            INT_LD: in BIT_VECTOR(0 to 3); SEL in BIT_VECTOR(0 to 1);
            OP: out BIT_VECTOR(0 to 11));
   end component;
   signal MUX_SEL: BIT_VECTOR (0 to 1);
   signal LD_SIG: BIT_VECTOR (0 to 3);
begin
   UPPER: REG_SELECT port map (CLOCK, LOAD, IP(0 to 1), OPCODE, MUX_SEL,
                               LD_SIG);
   LOWER: REG_STORAGE port map (CLOCK, IP, LD_SIG, MUX_SEL, OP);
end PE_REGISTER_BLOCK;
```

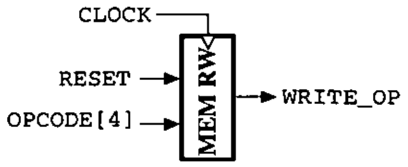
A.3 Instruction Set Storage

A.3.1 Memory Read/Write Generator

This is a control signal generator. It takes in the binary value for an instruction operation and generates the correct read/write signal for the static memory that holds the instruction set data. On a RESET the unit outputs a logic-0 (read) regardless of the other input data for the current and subsequent clock cycle. If the operation is a [PROG] instruction then the a read is generated on the current clock cycle, with a logic-1 (write) being generated on the subsequent clock cycle. Any other instruction results in a read being generated on the subsequent clock cycle. The

VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.14.

```
--------------------------------
-- CLOCK:    System clock
-- RESET:    System reset
-- OPCODE:   Current instruction
-- WRITE_OP: Write operation
--------------------------------

entity MEM_RW is
   port (CLOCK, RESET: in  BIT;
         OPCODE:       in  BIT_VECTOR(0 to 3);
         WRITE_OP:     out BIT);
end MEM_RW;

architecture MEM_RW_CONTROL of MEM_RW is
   component NAND2
      port (A, B: in BIT; RES: out BIT);
   end component;
   component OR2
      port (A, B: in BIT; RES: out BIT);
   end component;
   component NOR2
      port (A, B: in BIT; RES: out BIT);
   end component;
   component HALF_LATCH
      port (CLOCK, D: in BIT; Q: out BIT);
   end component;
   component MUX_2
      port (SEL, A, B: in BIT; OUT: out BIT);
   end component;
   signal GEN_VAL, NEXT_OP, T1, T2: BIT;
   signal L1_OUT, L2_OUT, L3_IN: BIT;
begin
   --------
   -- Check for [PROG]/[0011] & RESET
   --------
   NA1: NAND2 port map (OPCODE(0), OPCODE(1), T1);
   OR2: OR2 port map (OPCODE(2), OPCODE(3), T2);
   NR1: NOR2 port map (T1, T2, GEN_VAL);
   MX1: MUX_2 port map (RESET, GEN_VAL, 0, NEXT_OP)
   --------
   -- Send result into LATCH chain
   --------
   LT1: LATCH port map (NEXT_OP, CLOCK, L1_OUT);
   LT2: LATCH port map (L1_OUT, not CLOCK, L2_OUT);
   MX2: MUX_2 port map (RESET, L2_OUT, 0, L3_IN);
   LT3: LATCH port map (L3_IN, CLOCK, WRITE_OP);
end MEM_RW_CONTROL;
```



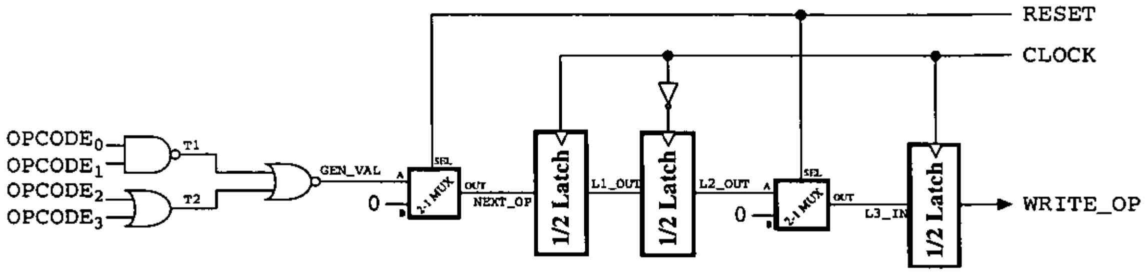


Figure A.14 Opcode Memory Read/Write Generator

A.3.2 Opcode Lock Unit

A.3.2.1 Inner Block Circuit

This is the inner section of the opcode memory lock-status controller. It keeps track of the current state of the lock circuitry: locked, locking or unlocked. Depending on the input OPCODE and the current status of the circuitry, as given by the two negative-edge latches, the status is changed to a different value. Note, no circuitry for the generation of REG_ROW is given: this simply indicates if the data on IP is identical to the row of the PE in the processor array. This is done by a single multi-input NOR gate with some inverted inputs - PE's in each row of the array, therefore, it has a different set of inputs to the NOR gate. The VHDL code for the generation of REG_ROW for a PE in row #3 of the array is given. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.14.

```
-----------------------------------
-- CLOCK:      System clock
-- RESET:      System reset signal
-- OPCODE[4]:  Current PE opcode
-- NS[3]:      Current NS data input
-- LOCK_SIG:   Output Lock signal
-----------------------------------

entity LOCK_INNER is
   port (CLOCK:     in  BIT;
         RESET:     in  BIT;
         OPCODE:    in  BIT_VECTOR(0 to 3);
         NS:        in  BIT_VECTOR(0 to 2);
         LOCK_SIG : out BIT);
end LOCK_INNER;
```

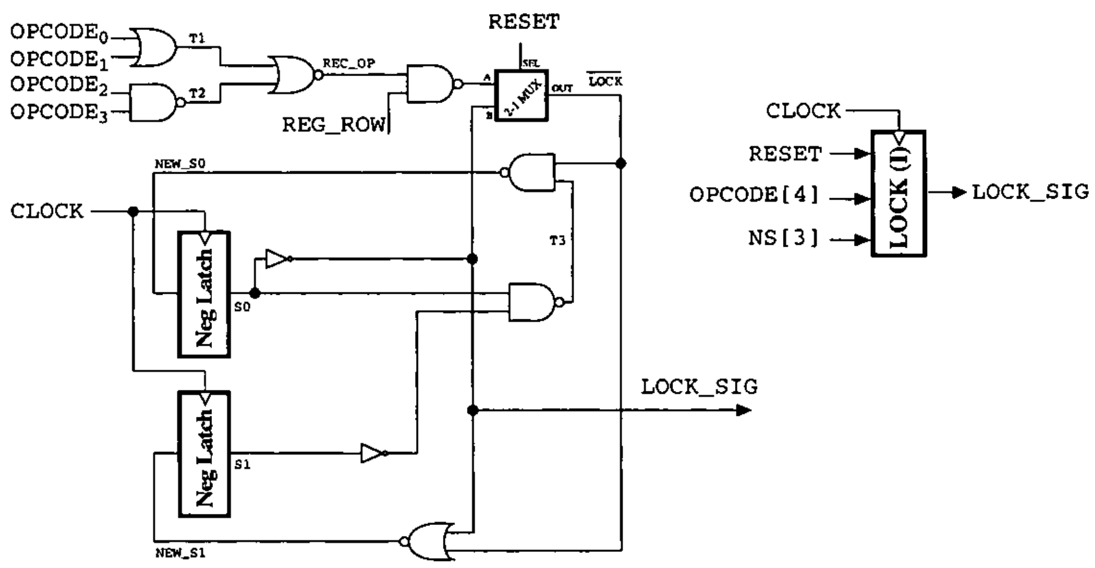


Figure A.15 Opcode Lock Inner Block

```
architecture OP_LOCK_INNER of LOCK_INNER is
   component NAND2
      port (A, B: in BIT; OUTPUT: out BIT);
   end component;
   component AND2
      port (A, B: in BIT; OUTPUT: out BIT);
   end component;
   component NOR3
      port (A, B, C: in BIT; OUTPUT: out BIT);
   end component;
   component NOR2
      port (A, B: in BIT; OUTPUT: out BIT);
   end component;
   component NOR
      port (A, B: in BIT; OUTPUT: out BIT);
   end component;
   component MUX_2
      port (SEL, A, B: in BIT; OUT out BIT);
   end component;
   component NEG_OPEN_LATCH
      port (CLOCK, D: in BIT; Q: out BIT);
   end component;
   signal T1, T2, T3: BIT;
   signal REG_ROW, REC_OP, PRE_LOCK: BIT;
   signal LOCK_, S0, S1: BIT;
   signal NEW_S0, NEW_S1: BIT;
begin
   --------
   -- Work out if a LOCK has happened for this row
   --------
   OR1: OR2 port map (OPCODE(0), OPCODE(1), T1);
   NA1: NAND2 port map (OPCODE(2), OPCODE(3), T2);
   NR1: NOR2 port map (T1, T2, REC_OP);
   NR2: NOR3 port map (NS(2), not NS(1), not NS(0), REG_ROW);
   NA2: NAND2 port map (REC_OP, REG_ROW, PRE_LOCK);
   --------
   -- Override PRE_LOCK during RESET if req'd
   --------
   MX1: MUX_2 port map (PRE_LOCK, not S0, LOCK_);
   --------
   -- Keep the latches up to date
   --------
   RG1: NEG_OPEN_LATCH port map (CLOCK, NEW_S0, S0);
   RG2: NEG_OPEN_LATCH port map (CLOCK, NEW_S1, S1);
   --------
   -- Modify the two latches to indicate current state
   --------
   NA3: NAND2 port map (S0, not S1, T3);
   NA4: NAND2 port map (LOCK_, T3, NEW_S1);
   NR3: NOR3 port map (not S0, LOCK_, NEW_S1);
   LOCK_SIG_ <= S0;
end OP_LOCK_INNER;
```

A.3.2.2 Outer Block Circuit

This is the outer section of the opcode memory lock-status controller. This unit stores an opcode in a set of registers on the cycle after a [LOCK] instruction is processed (during the *locking* state). It also routes either this stored opcode or that present on the OPCODE input to the control signal memory unit; the relevant control signals are then read from or written to that address. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.16.

```
-----------------------------------
-- CLOCK:      System clock
-- RESET:      System reset signal
-- OPCODE[4]: Current PE opcode
-- NS[3]:      Current NS data input
-- ADDR[4]:    RAM address location
-----------------------------------

entity LOCK_OUTER is
   port (CLOCK, RESET: in BIT;
         OPCODE:  in  BIT_VECTOR(0 to 3);
         NS:      in  BIT_VECTOR(0 to 2);
         MEMADDR: out BIT_VECTOR(0 to 3));
end LOCK_OUTER;

architecture OP_LOCK_OUTER of LOCK_OUTER is
   component LOCK_INNER
      port (OPCODE: in BIT_VECTOR(0 to 3);
            CLOCK, RESET: in BIT; NS: in BIT_VECTOR(0 to 2);
            LOCK_CTRL: out BIT);
   end component;
   component INTERNAL_REG
      generic (START, STOP: INTEGER);
      port (CLOCK, LOAD: in BIT; IP: in BIT_VECTOR(START to STOP);
            OP: out BIT_VECTOR(START to STOP));
   end component;
```

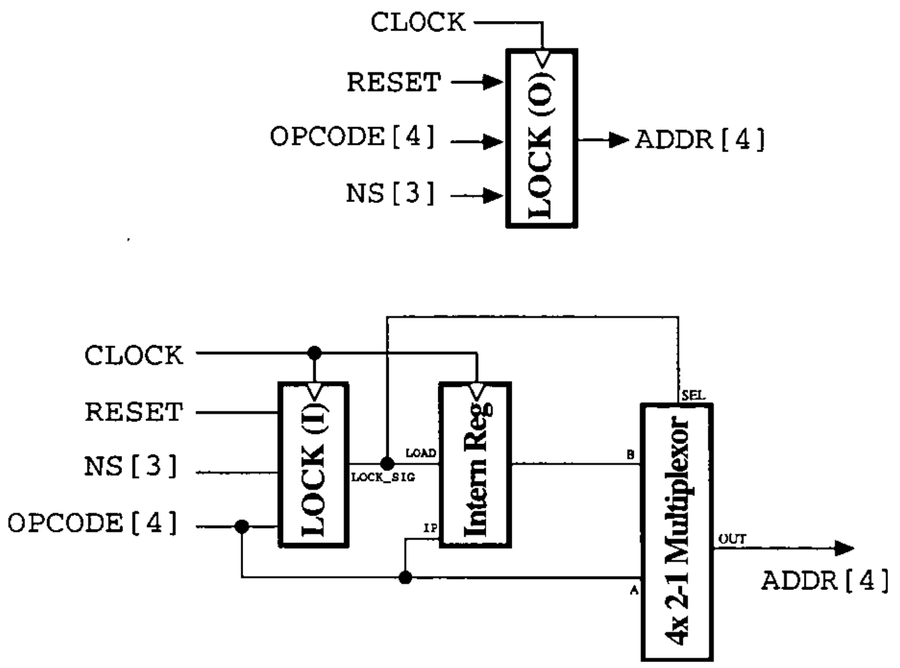


Figure A.16 Opcode Lock Outer Block

```
   component MUX_2
      port (SEL, A, B: in BIT; OUT: out BIT);
   end component;
   signal REG_OP: BIT_VECTOR(0 to 3);
   signal LOCK_SIG: BIT;
begin
   --------
   -- Ensure correct address in register block
   --------
   CL: LOCK_INNER port map (OPCODE, CLOCK, RESET, NS, LOCK_SIG);
   ST: INTERNAL_REG generic map (0, 3)
                    port map (CLOCK, LOCK_SIG, OPCODE, REG_OP);
   --------
   -- Send correct address to RAM unit
   --------
   MX: for K in 0 to 3 generate
      GEN: MUX_2 port map (LOCK_SIG, REG_OP(K), OPCODE(K), MEMADDR(K));
   end generate MX;
end OP_LOCK_OUTER;
```

A.3.3 Additional Control Signal Store

This is a data storage area. Section 5.2.2.2 stated that a temporary 10-bit register is required when programming the instruction set: this data accompanies the [PROG] instruction and is combined with all data on the NS datapath on the following clock cycle in order to create the full 15-bit word of control signals. This is accomplished via three negative edge latches, which store data on NS whenever a WRITE_OP is not occurring. This data is always made available at the circuit output, so that when a WRITE_OP does occur the data on NS on the previous clock cycle is forwarded to the instruction set memory along with the data currently on NS[12]. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.17.

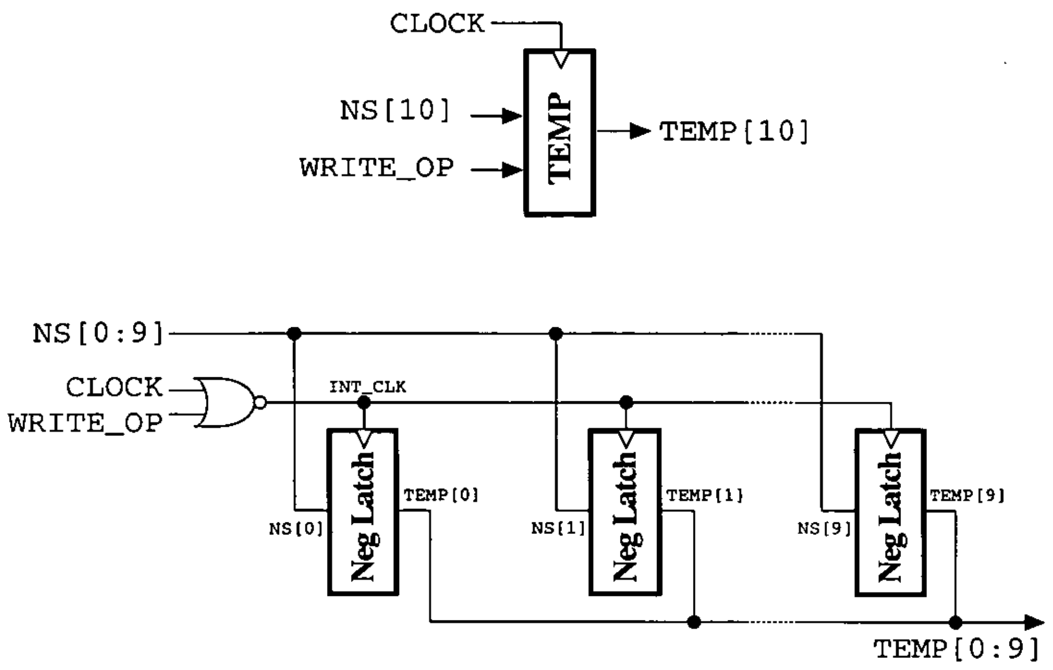
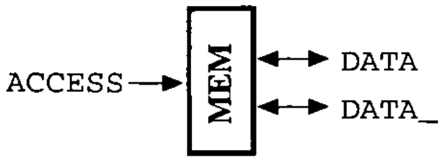


Figure A.17 Control Signal Temporary Storage

```
----------------------------------
-- CLOCK:     System clock
-- WRITE_OP:  Current RAM R/W mode
-- NS:        Current NS data input
-- TEMP:      Stored temp data
----------------------------------

entity TEMP_REG is
   port (CLOCK, WRITE_OP: in BIT;
         NS:   in  BIT_VECTOR(0 to 9);
         TEMP: out BIT_VECTOR(0 to 9));
end TEMP_REG;

architecture OP_TEMP_REG of TEMP_REG is
   component NEG_OPEN_LATCH
      port (CLOCK, D: in BIT; Q: out BIT);
   end component;
   signal INT_CLK: BIT;
begin
   --------
   -- Simple and straightforward register block
   --------
   INT_CLK <= CLOCK nor WRITE_OP;
   REG_BLK: for K in 0 to 9 generate
      LCH1: NEG_OPEN_LATCH port map (INT_CLK, NS[K], TEMP[K]);
   end generate REG_BLK;
end OP_TEMP_REG;
```

A.3.4 Instruction Set RAM Unit

A.3.4.1 Base Memory Storage Cell



This is a data storage unit. It is a standard 6-transistor static memory cell. The outside two transistors restrict access to the cell via the ACCESS input. The pair of back-to-back inverters hold the data value. If the values on DATA and DATA_ are different then they are written into the cell memory upon a memory access. If the values on DATA and DATA_ are both logic-1 then the memory is being read from. The inverters within the cell then pull one of either DATA or DATA_ down to logic-0, such that they match the values stored internally at D and D_. Values of logic-0 on both DATA and DATA_ are invalid for a memory access. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.18.

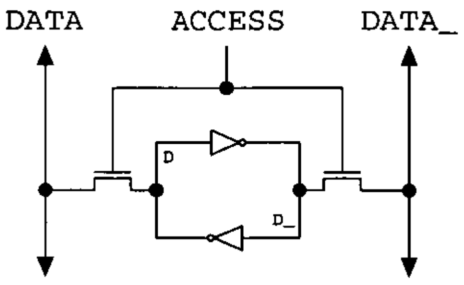


Figure A.18 Base Memory Cell

```
----------------------------------------
-- DATA:    Input/output data
-- DATA_:   Negative input/output data
-- ACCESS: Memory operation in progress
----------------------------------------

entity BASE_MEMORY is
   port (DATA:    buffer BIT;
         DATA_:   buffer BIT;
         ACCESS: in     BIT);
end BASE_MEMORY;

architecture OP_BASE_MEM of BASE_MEMORY is
   signal D, D_: BIT;
begin
   process (ACCESS, DATA)
   begin
      --------
      -- Memory access event
      --------
      if (ACCESS = '1') then
         --------
         -- Write Memory
         --------
         if (DATA = DATA_) then
            D <= DATA after 500 ps;
            D_ <= DATA_ after 500 ps;
         --------
         -- Read Memory
         --------
         else if ((DATA = '1') and (DATA_ = '1')) then
            DATA <= D after 1 ns;
            DATA_ <= D_ after 1 ns;
         end if
      end if;
   end process;
   --------
   -- Keep the memory cycling
   --------
   D <= not D_ after 500 ps;
   D_ <= not D after 500 ps;
end OP_BASE_MEM;
```

A.3.4.2 Fixed Instruction Memory Cells

This is a constant data storage cell. It functions in a similar way to the base memory cell except that no write operations are possible, implying that ACCESS is only granted on read operations. Instead of having a pair of back-to-back inverters the cell has a direct connection to V_{DD} and V_{SS}, the order of which depends on the generic input DATA_VAL: if DATA_VAL is logic-0 then the shaded power connections are used and the non-shaded power connections are not used. If DATA_VAL is logic-1 then the converse is true. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.19.

```
---------------------------------------
-- DATA:     Output data
-- DATA_:    Negative utput data
-- ACCESS:   Read operation in progress
-- DATA_VAL: Type of constant cell
---------------------------------------

entity CONST_MEM is
   generic (DATA_VAL: BIT);
   port (ACCESS:   in  BIT;
         DATA:     out BIT;
         DATA_:    out BIT);
end CONST_MEM;

architecture OP_CONST_MEM of CONST_MEM is
begin
   process (ACCESS)
   begin
      --------
      -- Read Memory only valid op
      --------
      if ((ACCESS = '1') and (DATA = '1') and (DATA_ = '1')) then
         DATA <= DATA_VAL after 1 ns;
```

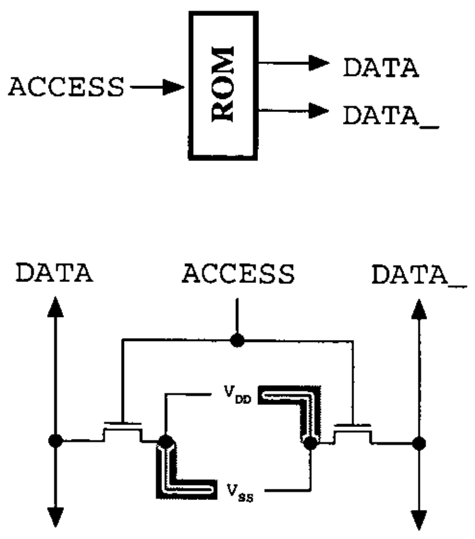


Figure A.19
Constant Memory Cell

```
         DATA_ <= (not DATA_VAL) after 1 ns;
      end if;
   end process;
end OP_CONST_MEM;
```

A.3.4.3 Memory Address Decoder

This is a data selector unit. The signal ADDR indicates which of the 16 memory locations are to be accessed. The decoder unit then outputs 16 ACCESS signals, one per word of memory, indicating whether or not memory cells in a particular word allow data to be read to or written from them. When a read operation is in progress, as indicated by WRITE_OP, only one of the ACCESS signals are non-zero. During a write operation one or none of the ACCESS signals are non-zero; writing to the four fixed memory words, from [0000] to [0011] respectively, is meaningless and if such an operation is attempted then all of the ACCESS signals are set to zero via the multiplexors. The circuitry to decode addresses [0000], [0001], [0010] and [1111] is shown in figure A.20; each one simply identifies if the location in ADDR is a particular number, shown inside the gate, outputting a logic-1 if so. This is done with a mixture of NOR gates, AND gates and INVERTERS, and circuits to decode the other addresses are similar to those shown. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.20.

```
-------------------------------------
-- ADDR:     Memory location required
-- WRITE_OP: Write operation required
-- ACCESS:   Word select lines
-------------------------------------

entity ADDR_DECODER is
   port (WRITE_OP: in  BIT;
         ADDR:     in  BIT_VECTOR(0 to 3);
         ACCESS:   out BIT_VECTOR(0 to 15));
end ADDR_DECODER;

architecture OP_ADDR_DEC of ADDR_DECODER is
begin
   process (ADDR, WRITE_OP)
      variable TEMP_LINES: BIT_VECTOR(0 to 15);
   begin
      --------
      -- Blank all access lines initially
      --------
      TEMP_LINES := (others => '0');
```

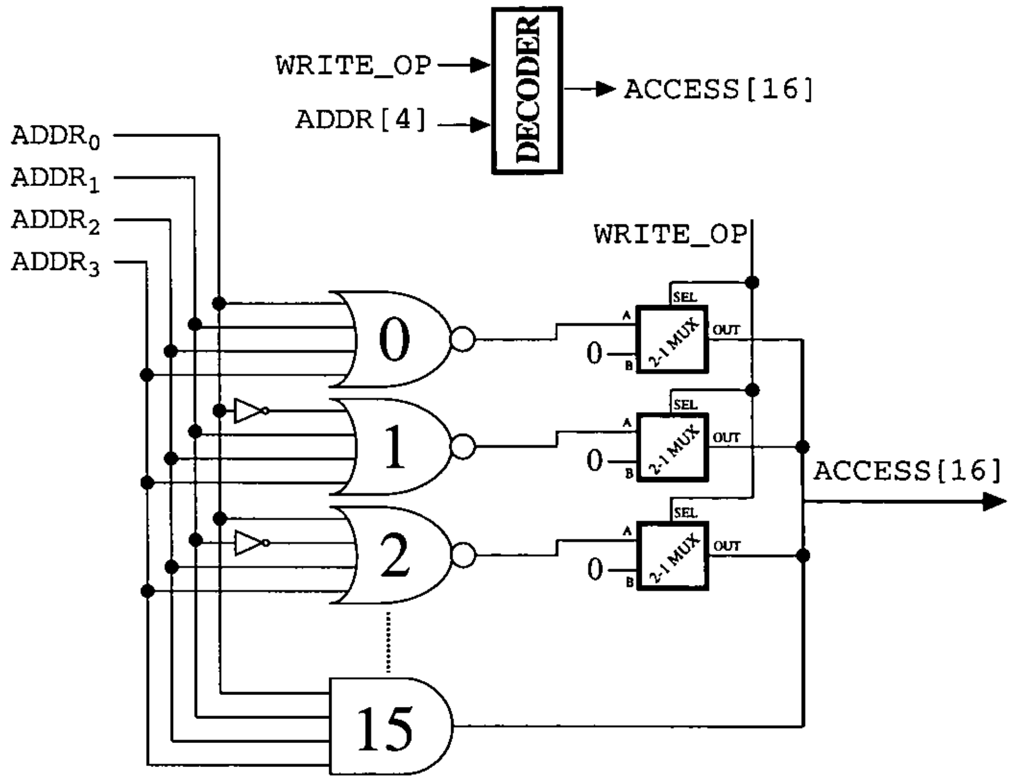


Figure A.20 Memory Address Decoder

```
    --------
    -- Mark the correct one
    --------
    case ADDR is
       when "0000" => TEMP_LINES[0] := 1;
       when "0001" => TEMP_LINES[1] := 1;
       when "0010" => TEMP_LINES[2] := 1;
       when "0011" => TEMP_LINES[3] := 1;
       when "0100" => TEMP_LINES[4] := 1;
       when "0101" => TEMP_LINES[5] := 1;
       when "0110" => TEMP_LINES[6] := 1;
       when "0111" => TEMP_LINES[7] := 1;
       when "1000" => TEMP_LINES[8] := 1;
       when "1001" => TEMP_LINES[9] := 1;
       when "1010" => TEMP_LINES[10] := 1;
       when "1011" => TEMP_LINES[11] := 1;
       when "1100" => TEMP_LINES[12] := 1;
       when "1101" => TEMP_LINES[13] := 1;
       when "1110" => TEMP_LINES[14] := 1;
       when "1111" => TEMP_LINES[15] := 1;
    end case;
    --------
    -- Make exception for writing to fixed ops
    --------
    if ((ADDR[3] = '0') and (ADDR[2] = '0') and (WRITE_OP = '1')) then
       TEMP_LINES := (0 to 3 => '0');
    end if;
    --------
    -- Copy them all across
    --------
    ACCESS <= TEMP_LINES;
 end process;
```

```
end OP_ADDR_DEC;
```

A.3.4.4 Bit Control Logic Driver

This is a control generation unit. It ensures that the correct signal is on the DATA and DATA_ lines for each bit-column in the memory; bits at the same position in different words share the same DATA and DATA_ values, but at most one bit cell in this column is activated via it's ACCESS value. On a write operation the data from NS and TEMP is driven onto DATA, with the complement placed on DATA_. On a read operation both DATA and DATA_ have logic-1 placed on them. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.21.

```
-------------------------------------------
-- NS:        Current NS data
-- TEMP:      Previously stored NS data
-- WRITE_OP:  Type of operation required
-- DATA:      Data for the bit columns
-- DATA_:     More data for the bit columns
-------------------------------------------

entity MEM_BCL is
   port (NS:        in BIT_VECTOR(0 to 11);
         TEMP:      in BIT_VECTOR(0 to 9);
         WRITE_OP:  in BIT;
         DATA:      buffer BIT_VECTOR(0 to 21);
         DATA_:     buffer BIT_VECTOR(0 to 21));
end MEM_BCL;

architecture OP_MEM_BCL of MEM_BCL is
   component MUX_2
      port map (SEL, A, B: in BIT; OUT out BIT);
   end component MUX_2;
begin
   process (WRITE_OP, NS, TEMP)
   begin
      --------
      -- Generate NS related lines first
      --------
      GEN_NS: for K in 0 to 11 generate
         MX1: UX_2 port map (WRITE_OP, 1, NS[K], DATA[K]);
         MX2: MUX_2 port map (WRITE_OP, 1, (not NS[K]), DATA_[K]);
      end generate GEN_NS;
      --------
      -- Then generate old TEMP lines
      --------
      GEN_TEMP: for K in 0 to 9 generate
         MX3: MUX_2 port map (WRITE_OP, 1, TEMP[K], DATA[K+12]);
         MX4: MUX_2 port map (WRITE_OP, 1, (not TEMP[K]), DATA_[K+12]);
      end generate GEN_TEMP;
   end process;
end OP_MEM_BCL;
```

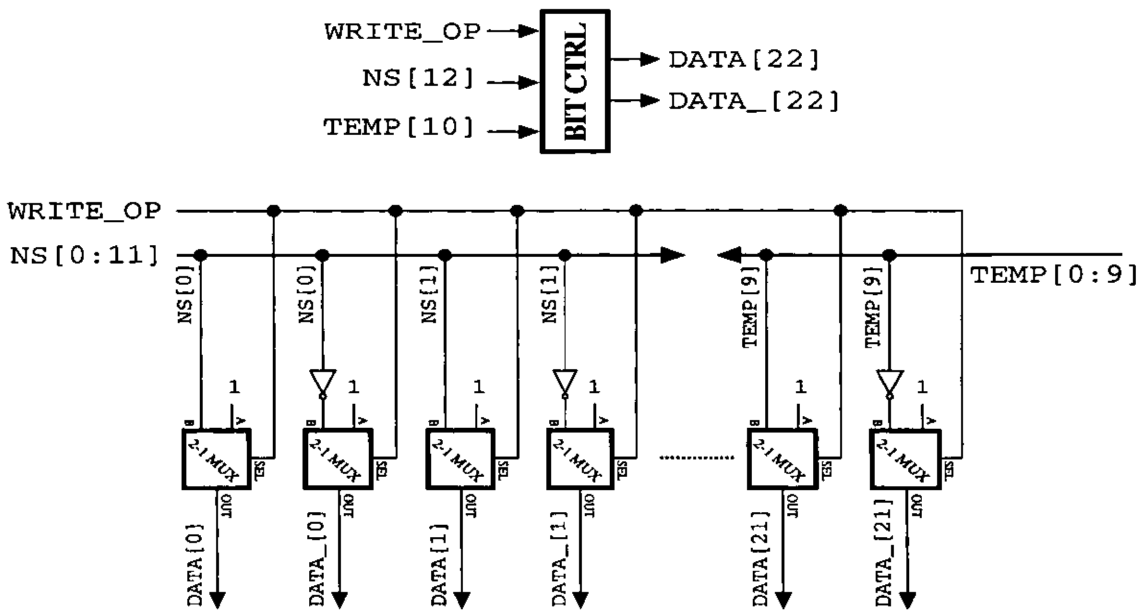
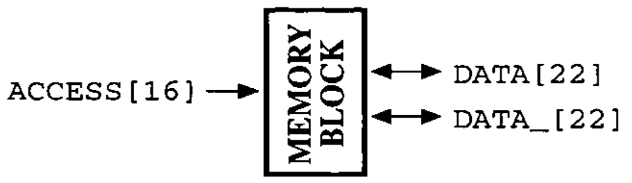


Figure A.21 RAM Unit Bit Control Logic

A.3.4.5 Instruction Memory Block

This is a data storage unit. It holds an array of base memory storage cells, which share common data I/O lines amongst the bit columns. Read/Write control is handled by the DATA and DATA_ lines, which is also where any output data is placed by the memory cells. Word selection is handled by the ACCESS signals, which activate a single row of cells within the array. The first four words of memory are ROM-based for the fixed instructions, whilst the rest are RAM-based for the programmable instructions. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.22.

```
----------------------------------------
-- DATA:   Bit-control data and data I/O
-- DATA_:  Bit control data and data I/O
-- ACCESS: Word control data
----------------------------------------

entity MEM_BLOCK is
   port (ACCESS: in BIT_VECTOR(0 to 15)
         DATA:   buffer BIT_VECTOR(0 to 21);
         DATA_:  buffer BIT_VECTOR(0 to 21));
end MEM_BLOCK;
```



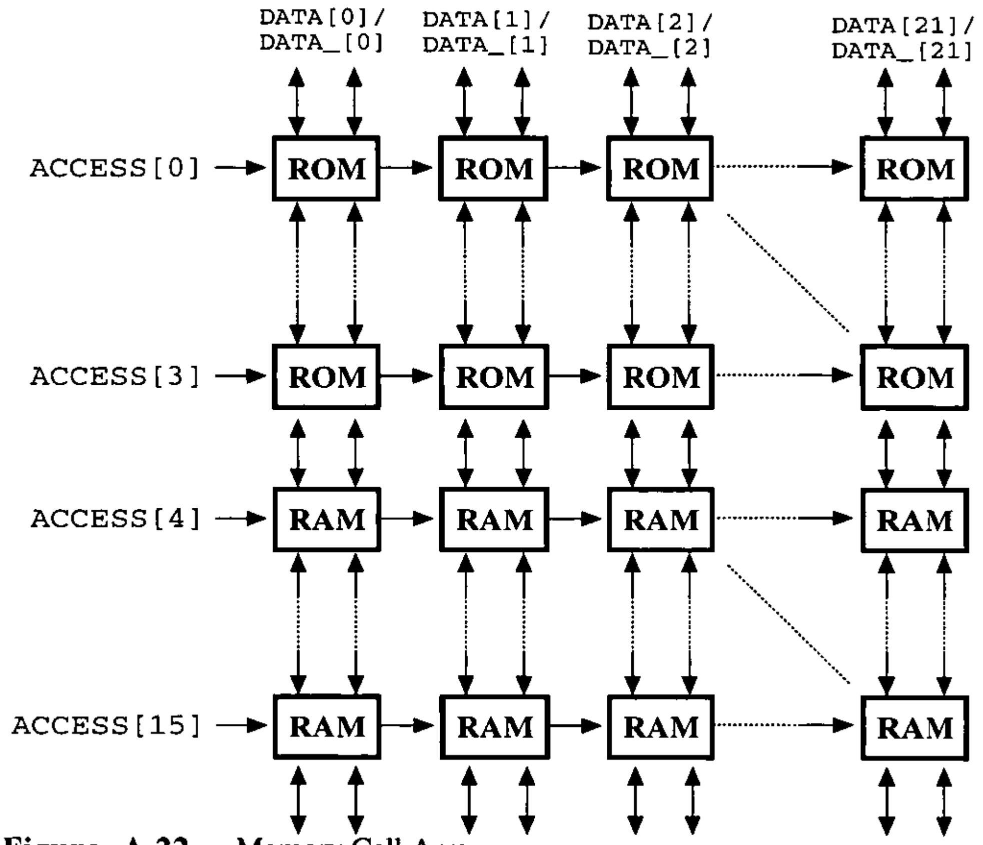


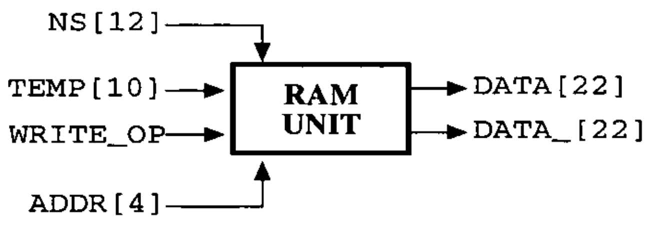
Figure A.22 Memory Cell Array

```
architecture OP_MEM_BLOCK of MEM_BLOCK is
   component BASE_MEM
      port (DATA, DATA_: buffer BIT; ACCESS: in BIT);
   end component;
   component CONST_MEM
      generic (DATA_VAL: BIT);
      port (ACCESS: in BIT; DATA, DATA_: buffer BIT);
   end component;
begin
   --------
   -- Simple large array of elements
   --------
   GEN_BLK1: for K in 0 to 21 generate
      --------
      -- Fixed instructions
      --------
      GEN_BLK2: for L in 0 to 3 generate
         MEM: CONST_MEM generic map (0)
                        port map (DATA[K], DATA_[K], ACCESS[L]);
      end generate GEN_BLK2;
      --------
      -- Programmable instructions
      --------
      GEN_BLK3: for L in 4 to 15 generate
         MEM: BASE_MEM port map (DATA[K], DATA_[K], ACCESS[L]);
      end generate GEN_BLK3;
   end generate GEN_BLK1;
end MEM_BLOCK;
```

A.3.4.6 Complete Memory Block

This is the entire memory block, consisting of the memory array block, address decoders and bit-control logic. No logic in addition to that already described in section A.3.4 exists in this circuit. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.23.

```
---------------------------------------------
-- NS:       Current NS data
-- TEMP:     Previous NS data
-- WRITE_OP: Memory operation
-- ADDR:     Memory address (OPCODE) required
-- DATA:     Memory I/O data
-- DATA_:    Memory I/O data
---------------------------------------------

entity RAM_UNIT is
   port (NS:       in BIT_VECTOR(0 to 11);
         TEMP:     in BIT_VECTOR(0 to 9);
         WRITE_OP: in BIT;
         ADDR:     in BIT_VECTOR(0 to 3);
         DATA:     buffer BIT_VECTOR(0 to 21);
         DATA_:    buffer BIT_VECTOR(0 to 21));
end RAM_UNIT;

architecture OP_RAM_UNIT of RAM_UNIT
   component MEM_BLOCK
      port (ACCESS: in BIT_VECTOR(0 to 15);
            DATA, DATA_: buffer BIT_VECTOR(0 to 21));
   end component;
   component ADDR_DECODER
   port (WRITE_OP: in  BIT; ADDR: in  BIT_VECTOR(0 to 3);
         ACCESS: out BIT_VECTOR(0 to 15));
   end component;
   component MEM_BCL
      port (NS in BIT_VECTOR(0 to 11); TEMP: in BIT_VECTOR(0 to 2);
         WRITE_OP: in BIT; DATA, DATA_:buffer BIT_VECTOR(0 to 21));
   end component;
   signal WORD_ACCESS: BIT_VECTOR(0 to 15);
begin
   --------
   -- Simply link everything together
   --------
   COMP1: MEM_BCL port map (NS, TEMP, WRITE_OP, DATA, DATA_);
   COMP2: ADDR_DECODER port map (NS, ADDR, WORD_ACCESS);
   COMP3: MEM_BLOCK: port map (WORD_ACCESS, DATA, DATA_);
end OP_RAM_UNIT;
```



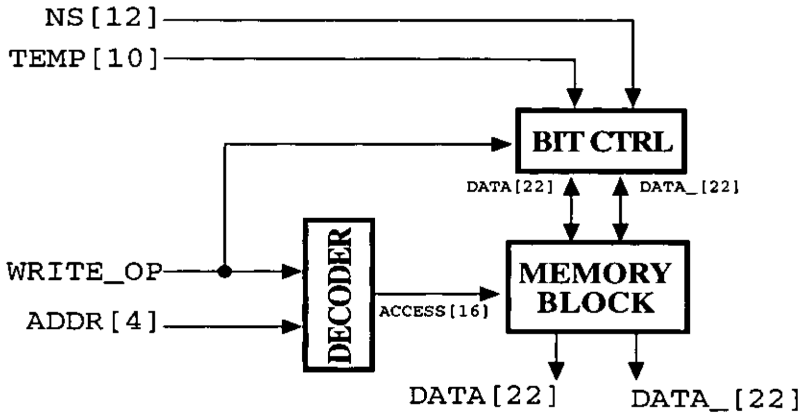


Figure A.23 Enitre RAM Unit

A.3.5 Complete Instruction Set Store

This is a data storage unit. It contains all of the units required for the entire instruction set memory unit, including read/write control, opcode locking, physical memory array and temporary data storage. It adds no additional functionality other than that already described. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.24.

```
-----------------------------------
-- CLOCK:    System clock
-- OPCODE:   Current opcode
-- RESET:    System reset signal
-- NS:       Current NS input data
-- WRITE_OP: Memory operation
-- CTRL[22]: Control signal outputs
-----------------------------------

entity OPCODE_MEMORY is
   port (CLOCK, RESET: in  BIT;
         OPCODE:   in  BIT_VECTOR(0 to 3);
         NS:       in  BIT_VECTOR(0 to 11);
         WRITE_OP: out BIT;
         CTRL:     out BIT_VECTOR(0 to 21));
end OPCODE_MEMORY;

architecture OP_INSTR_SET of OPCODE_MEMORY is
```

```
   component RAM_UNIT
   port (NS: in BIT_VECTOR(0 to 11); TEMP: in BIT_VECTOR(0 to 3);
         WRITE_OP: in BIT; ADDR: in BIT_VECTOR(0 to 3);
         DATA, DATA_: buffer BIT_VECTOR(0 to 21));
   end component;
   component TEMP_REG
   port (CLOCK, WRITE_OP: in BIT; NS: in  BIT_VECTOR(0 to 9);
         TEMP: out BIT_VECTOR(0 to 9));
   end component;
   component LOCK_OUTER
   port (OPCODE: in BIT_VECTOR(0 to 3); CLOCK, RESET: in BIT;
         MEMADDR: out BIT_VECTOR(0 to 3));
   end component;
   component MEM_RW is
   port (CLOCK, RESET: in  BIT; OPCODE: in  BIT_VECTOR[0 to 3];
         WRITE_OP: out BIT);
   end component;
   signal MEMORY_WRITE: BIT;
   signal MEM_ADDR: BIT_VECTOR(0 to 3);
   signal TEMP_NS: BIT_VECTOR(0 to 9);
   signal PX, NX: BIT_VECTOR(0 to 21);
begin
   --------
   -- Simply map everything together
   --------
   COM1: MEM_RW port map (CLOCK, RESET, OPCODE, MEMORY_WRITE);
   COM2: LOCK_OUTER port map (OPCODE, CLOCK, RESET, MEM_ADDR);
   COM3: TEMP_REG port map (CLOCK, MEMORY_WRITE, NS[0 to 9], TEMP_NS);
   COM4: RAM_UNIT port map (NS, TEMP_NS, MEMORY_WRITE, MEM_ADDR, PX, NX);
   WRITE_OP <= MEMORY_WRITE;
   CTRL <= PX;
end OP_INSTR_SET;
```

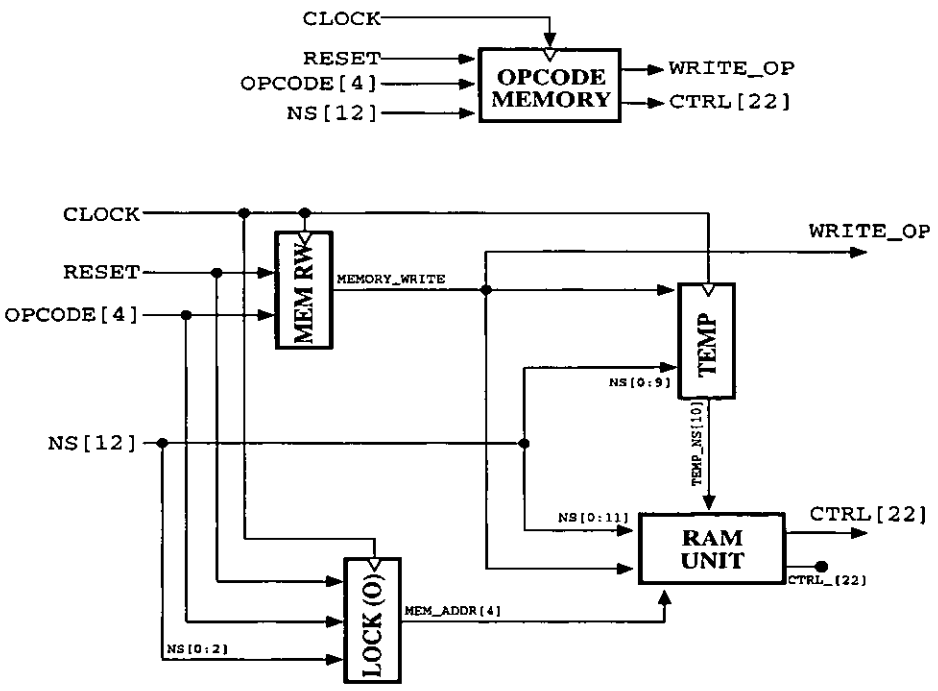


Figure A.24 Complete Instruction Set Memory

A.4 Mathematical Units

A.4.1 ABS Unit

This is a subsidiary unit to the multiplier. It modifies one of the inputs to the multiplier to be ±1.0, such that the other input to the multiplier is of a specified sign after the multiplication. The values and implications of the signal CONTROL were specified in table 5.5. If the unit is not active (CONTROL[1]) then the input DATA is allowed to pass through unaltered. If the unit is active then the sign of this input (DATA[11]) and CONTROL[0] determines whether or not DATA is replaced by either +1.0 or -1.0. The final output of the unit is placed in MOD_DATA. Note, the binary representations within the PE for values +1.0 and -1.0 are 000010000000 and 111110000000 respectively. The VHDL code for this circuit is shown overleaf, and a schematic of the circuit itself is shown in figure A.25.

```
---------------------------------
-- CONTROL:  Unit control signals
-- DATA:     Input data
-- MOD_DATA: Output data
---------------------------------

entity ABS_UNIT is
   port (CONTROL:  in  BIT_VECTOR(0 to 1);
         DATA:     in  BIT_VECTOR(START to STOP);
         MOD_DATA: out BIT_VECTOR(START to STOP));
end ABS_UNIT;

architecture MULT_ABS_UNIT of ABS_UNIT is
   component MUX_2
      port (SEL, A, B: in BIT; OUT: out BIT);
   end component;
   signal SELECT: BIT;
begin
   --------
   -- Generate MUX's that are the same regardless of input sign
   --------
   BLK1: for K in 0 to 6 generate
      MX1: MUX_2 port map (CONTROL[1], DATA[K], 0, MOD_DATA[K]);
   end generate BLK1;
   BLK2: MUX_2 port map (CONTROL[1], DATA[7], 1, MOD_DATA[7]);
   --------
   -- Generate sign-dependent MUX's
   --------
   SELECT <= CONTROL[0] xor DATA[11];
   BLK3: for K in 8 to 11 generate
      MX2: MUX_2 port map (CONTROL[1], DATA[K], SELECT, MOD_DATA[K]);
   end generate BLK3;
end MULT_ABS_UNIT;
```

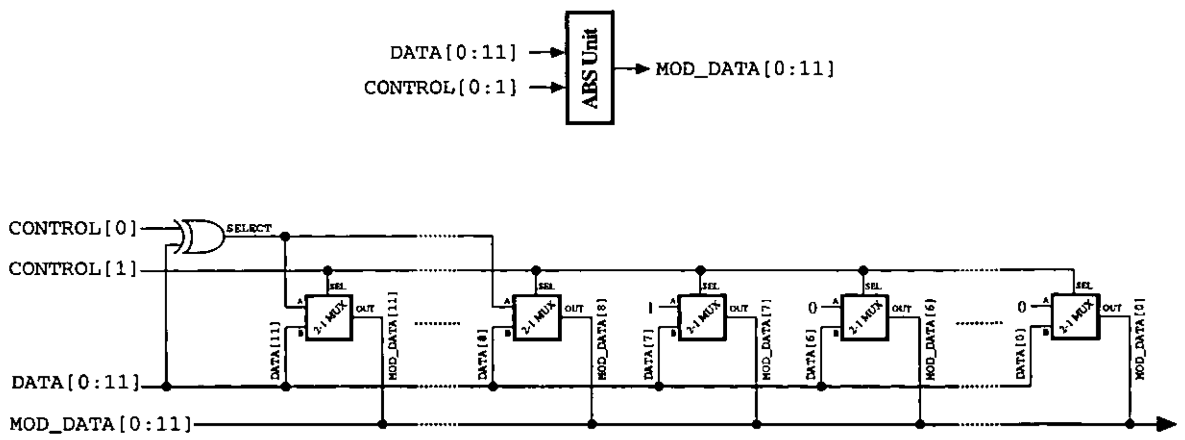


Figure A.25 ABS Unit

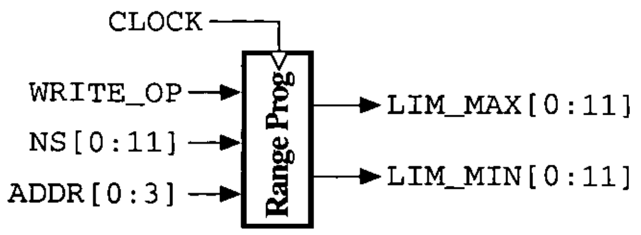
A.4.2 Result Range Limiter

A.4.2.1 Range Limit Programmer

This is a data storage unit. It stores and continually outputs a pair of 12-bit values, LIM_MAX and LIM_MIN. These are stored within two 12-bit register blocks. The upper block, used to store LIM_MAX, has the value NS stored if a write operation is in progress (WRITE_OP) and the value on ADDR is [0011]. If the value on ADDR is [0000] then the data on NS is stored in the lower register block, which holds LIM_MIN, during a write operation. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.26.

```
------------------------------------------
-- CLOCK:    System clock
-- NS:       Data to be stored
-- ADDR:     Indicates which limit is on NS
-- WRITE_OP: Data on NS is to be stored
-- LIM_MAX:  Current maximum limit
-- LIM_MIN:  Current minimum limit
------------------------------------------

entity LIMIT_STORE is
   port (CLOCK:    in  BIT;
         NS:       in  BIT_VECTOR(0 to 11);
         ADDR:     in  BIT_VECTOR(0 to 3);
         WRITE_OP: in  BIT;
         LIM_MAX:  out BIT_VECTOR(0 to 11);
         LIM_MIN:  out BIT_VECTOR(0 to 11));
end LIMIT_STORE;

architecture RL_LIMIT_STORE of LIMIT_STORE is
   component AND2
      port (A, B: in BIT; RES: out BIT);
   end component;
   component NOR2
```

```
      port (A, B: in BIT; RES: out BIT);
   end component;
   component INTERNAL_REG
      generic (START, STOP: INTEGER);
      port (CLOCK, LOAD: in BIT; IP: in BIT_VECTOR(START to STOP);
            OP: out BIT_VECTOR(START to STOP));
   end component;
   signal T1, T2, T3, T4: BIT;
   signal LOAD_HIGH, LOAD_LOW: BIT;
begin
   --------
   -- Generate control signals
   --------
   AN1: AND2 port map (ADDR[1], ADDR[0], T1);
   NR1: NOR2 port map (ADDR[3], ADDR[2], T2);
   NR2: NOR2 port map (ADDR[1], ADDR[0], T3);
   AN2: AND2 port map (T2, WRITE_OP, T4);
   AN3: AND2 port map (T1, T4, LOAD_HIGH);
   AN4: AND2 port map (T4, T5, LOAD_LOW);
   --------
   -- Instantiate both register blocks
   --------
   RB1: INTERNAL_REG port map (CLOCK, LOAD_HIGH, NS, LIM_MAX);
   RB2: INTERNAL_REG port map (CLOCK, LOAD_LOW, NS, LIM_MIN);
end RL_LIMIT_STORE;
```



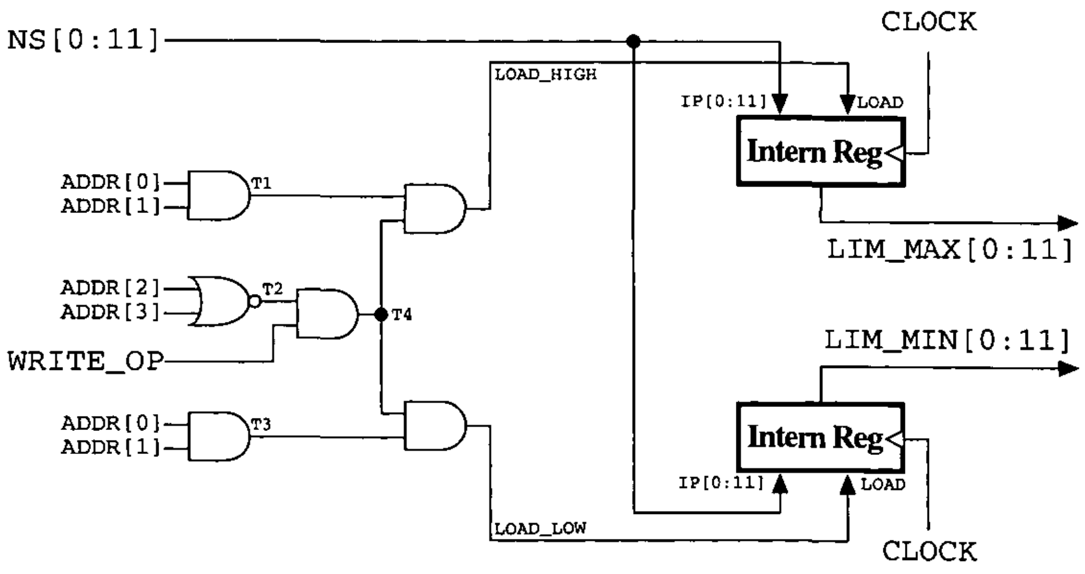
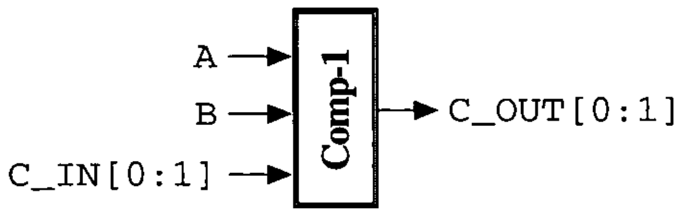


Figure A.26 Range Limit Data Stores

A.4.2.2 Single Bit Data Comparator

This is a control signal generator. This is a bit-slice circuit, designed to be implemented with a cascade of multiple instances. It takes in two 1-bit values, A and B, along with a pair of control signals, C_IN, and indicates via C_OUT whether A=B, A>B or A<B by setting C_OUT to [10], [00] or [11] respectively. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.27.



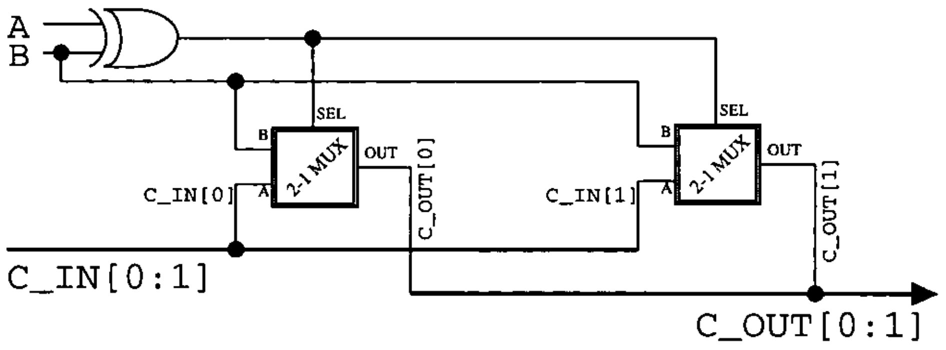


Figure A.27 1-Bit Comparator

```
--------------------------------------------
-- A:     First input value
-- B:     Second input value
-- C_IN:  Initial state of n-bit comparison
-- C_OUT: Comparison result
--------------------------------------------

entity BIT_COMPARE is
   port (A, B:  in  BIT;
         C_IN:  in  BIT_VECTOR(0 to 1);
         C_OUT: out BIT_VECTOR(0 to 1))
end BIT_COMPARE;

architecture RL_BIT_COMPARE of BIT_COMPARE is
   component MUX_2
      port (SEL, A, B: in BIT; OUT out BIT);
   end component;
   signal DIFF: BIT;
begin
```

```
   --------
   -- Simple multiplexor function
   --------
   DIFF <= A xor B;
   GEN: for K in 0 to 1 generate
      MUX: MUX_2 port map (DIFF, C_IN[K], B, C_OUT[K]);
   end generate GEN;
end RL_BIT_COMPARE;
```

A.4.2.3 n-Bit Comparator

This is a control signal generator. It compares two n-bit data words, indicating which of the two are the largest (or if they are equal). Multiple instances of the *1-Bit Comparator* are used, with the C_OUT output of one being the C_IN input to the next. The final C_OUT value indicates the relative sizes of the two n-bit data words. The VHDL code for this circuit is shown below and a schematic of the circuit, instantiated as an 11-bit comparator, is shown in figure A.28.

```
--------------------------
-- A:    First input
-- B:    Second input
-- COMP: Comparator result
--------------------------

entity WORD_COMPARE is
   generic (START: INTEGER := 0; STOP: INTEGER := 10);
   port (A:    in  BIT_VECTOR(START to STOP);
         B:    in  BIT_VECTOR(START to STOP);
         COMP: out BIT_VECTOR(0 to 1));
end WORD_COMPARE;

architecture RL_WORD_COMPARE of WORD_COMPARE is
   component BIT_COMPARE
      port (A, B: in BIT; C_IN:  in  BIT_VECTOR(0 to 1);
            C_OUT: out BIT_VECTOR(0 to 1))
   end component;
   signal C_INT: array [START to STOP] of BIT_VECTOR(0 to 1);
begin
   --------
   -- Starting values
   --------
   C_INT[START] <= "10";
   --------
   -- Simply cascade lots of 1-bit comparators
   --------
   CASC: for K in START to (STOP-1) generate
      CMP: BIT_COMPARE port map (A[K], B[K], C_INT[K], C_INT[K+1]);
   end generate CASC;
   FIN: BIT_COMPARE port map (A[STOP], B[STOP], C_INT[STOP], COMP);
end RL_WORD_COMPARE;
```

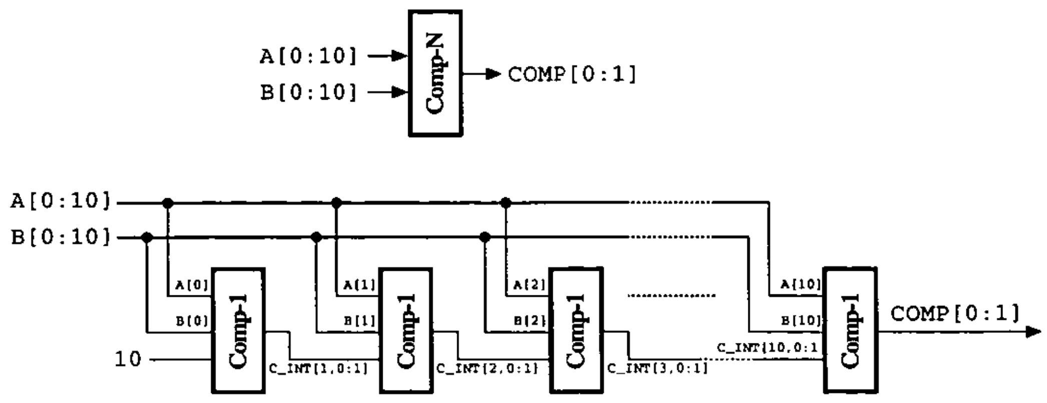


Figure A.28 N-Bit Comparator

A.4.2.4 Result Range Checker

This is a data interrogation unit. It takes in a 12-bit input NS and compares it with two other inputs HIGH and LOW. The unit uses a mixture of 1-bit and n-bit comparators to see if the NS input is greater than HIGH or less than LOW. This is indicated via the 2-bit control output CTRL. The VHDL code for this circuit is shown overleaf, and a schematic of the circuit itself is shown in figure A.29.

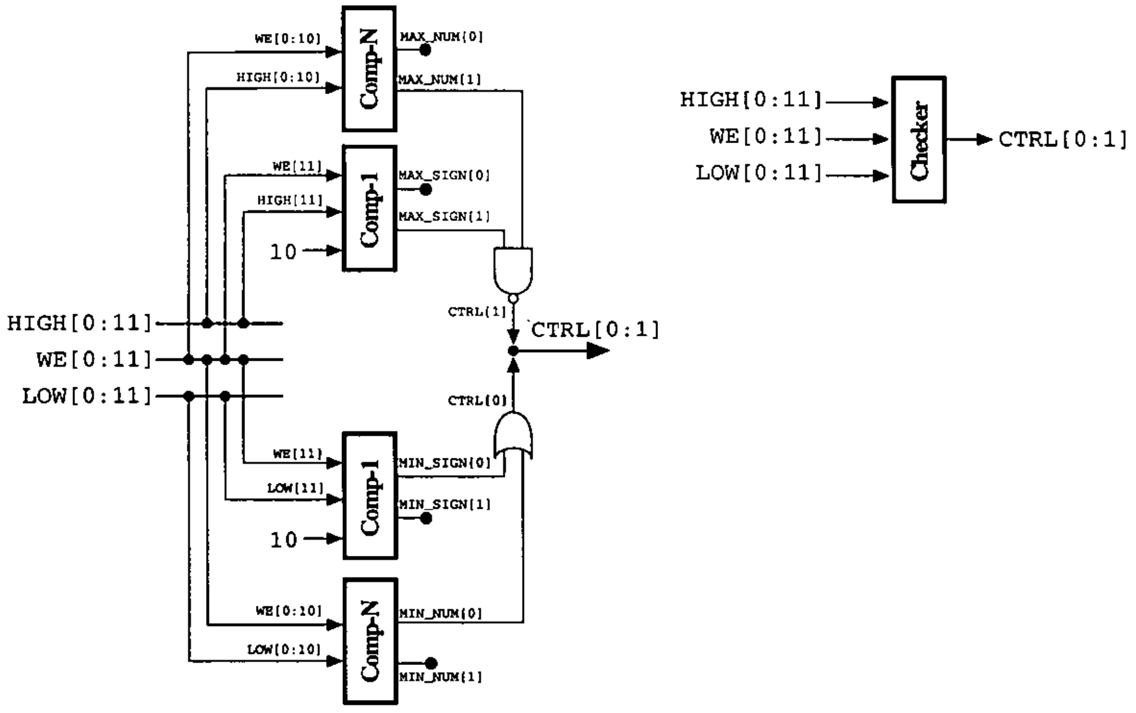


Figure A.29 Dual-Value Range Checker

```
-----------------------------
-- WE:   Data input
-- HIGH: Upper limiting value
-- LOW:  Lower limiting value
-- CTRL: Check result output
-----------------------------

entity RANGE_CHECK is
   port (WE:   in  BIT_VECTOR(0 to 11);
         HIGH: in  BIT_VECTOR(0 to 11);
         LOW:  in  BIT_VECTOR(0 to 11);
         CTRL: out BIT_VECTOR(0 to 1));
end RANGE_CHECK;

architecture RL_RANGE_CHECK of RANGE_CHECK is
   component WORD_COMPARE
      generic (START, STOP: INTEGER);
      port (A, B: in BIT_VECTOR(START to STOP);
            COMP: out BIT_VECTOR(0 to 1));
   end component;
   component BIT_COMPARE
      port (A, B: in BIT; C_IN: in BIT_VECTOR(0 to 1);
            C_OUT: out BIT_VECTOR(0 to 1))
   component BIT_COMPARE
   end component;
   signal MAX_NUM, MAX_SIGN: BIT_VECTOR(0 to 1);
   signal MIN_NUM, MIN_SIGN: BIT_VECTOR(0 to 1);
begin
   --------
   -- Maximum value check
   --------
   MX1: BIT_COMPARE port map (WE[11], HIGH[11], 10, MAX_SIGN);
   MX2: WORD_COMPARE port map (WE[0:10], HIGH[0:10], MAX_NUM);
   CTRL[1] <= MAX_SIGN[1] nand MAX_NUM[1];
   --------
   -- Minimum value check
   --------
   MN1: BIT_COMPARE port map (WE[11], LOW[11], 10, MIN_SIGN);
   MN2: WORD_COMPARE port map (WE[0:10], LOW[0:10], MIN_NUM);
   CTRL[0] <= MIN_SIGN[0] or MIN_NUM[0];
end RL_RANGE_CHECK;
```

A.4.2.5 Complete Result Range Checker

This is a data integrity checker. It combines the *Range Programmer* and *Range Checker*, along with a separate multiplexor, to determine if the data currently on the WE datapath lies within a specified range; if it is outside this range then the value on WE is replaced by the relevant range limit. The whole unit is activated through the assertion of the ACTIVE input. Note, that if the input on WE is both greater than the upper limit and less than the lower limit then the chosen value is WE, as such a situation implies that the range limits have been incorrectly specified.

The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.30.

```
---------------------------------------
-- CLOCK:    System Clock
-- ACTIVE:   Unit activation input
-- WRITE_OP: Memory operation required
-- WE:       Current WE data
-- NS:       Current NS data
-- ADDR:     Current opcode
-- NEW_WE:   New WE data after check
---------------------------------------

entity RESULT_RANGE_CHECKER is
   port (CLOCK:    in  BIT;
         ACTIVE:   in  BIT;
         WRITE_OP: in  BIT;
         NS:       in  BIT_VECTOR(0 to 11);
         WE:       in  BIT_VECTOR(0 to 11);
         ADDR:     in  BIT_VECTOR(0 to 3);
         NEW_WE:   out BIT_VECTOR(0 to 11));
end RESULT_RANGE_CHECKER;

architecture RL_RESULT_CHECKER of RESULT_RANGE_CHECKER is
   component RANGE_CHECK
      port (WE, HIGH, LOW: in BIT_VECTOR(0 to 11);
            CTRL: out BIT_VECTOR(0 to 1));
   end component;
   component LIMIT_STORE
      port (CLOCK: in  BIT; NS: in BIT_VECTOR(0 to 11);
            ADDR: in BIT_VECTOR(0 to 3); WRITE_OP: in BIT;
            LIM_MAX, LIM_MIN: out BIT_VECTOR(0 to 11));
   end component;
   component MUX_4
      port (SEL: in BIT_VECTOR(0 to 1); A, B, C, D: in BIT; OUT: out BIT);
   end component;
   signal MAX_VAL, MIN_VAL: BIT_VECTOR(0 to 11);
   signal CHECK_RES, MUX_CTRL: BIT_VECTOR(0 to 1);
begin
   --------
   -- Map inputs to functional units
   --------
   PRG: LIMIT_STORE port map (CLOCK, NS, ADDR, WRITE_OP, MAX_VAL, MIN_VAL);
   CHK: RANGE_CHECK port map (WE, MAX_VAL, MIN_VAL, CHECK_RES);
   --------
   -- Put everything through multiplexor
   --------
   MUX_CTRL[0] <= ACTIVE and CHECK_RES[0];
   MUX_CTRL[1] <= ACTIVE and CHECK_RES[1];
   BLK: for K in 0 to 11 generate
      MPX: MUX_4 port map (MUX_CTRL, WE[K], MIN_VAL[K], MAX_VAL[K], WE[K],
                           NEW_WE[K]);
   end generate BLK;.
0end RL_RESULT_CHECKER;
```

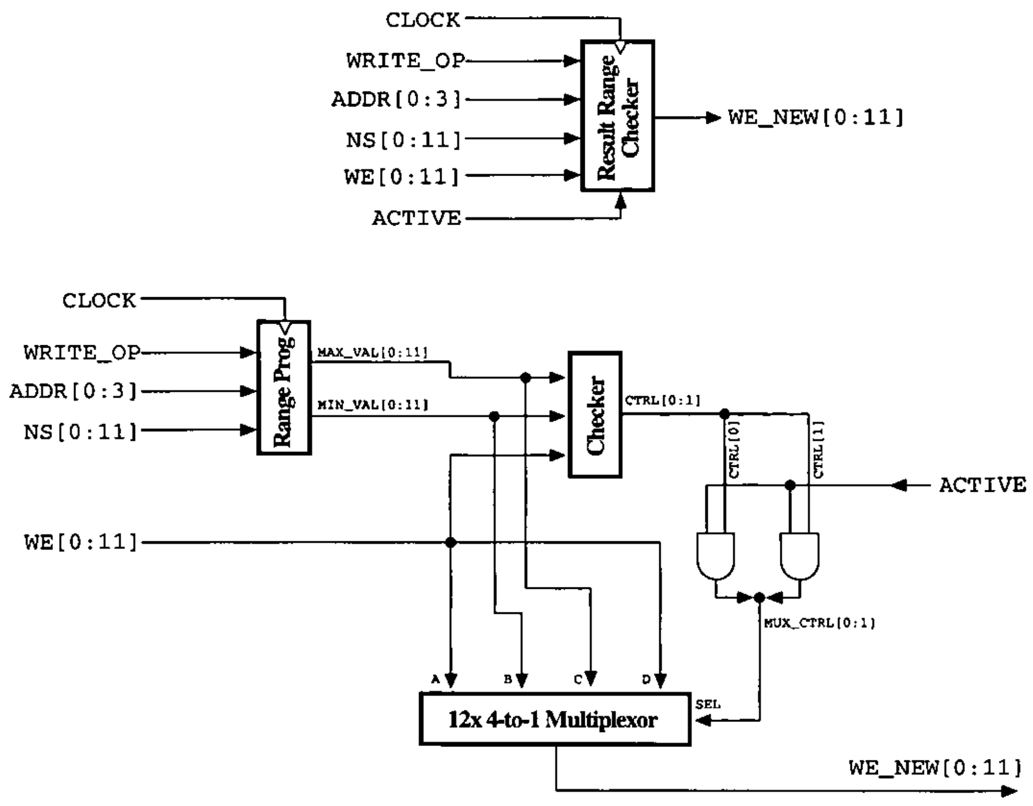


Figure A.30 Complete Result Range Checker Unit

A.4.3 General Purpose Comparator

This is a data comparator. It takes in two 12-bit data items, A and B, and performs some comparison operation on them. The unit can carry out either an equality check, indicated by the input EQUALITY, or can choose the maximum or minimum of the two inputs, as indicated by the input MAXIMUM. A comparison operation is automatically done when an equality operation is not required. During an equality check the 12-bit output of the unit, COMP_RES, contains either 1.0 or 0.0 depending upon a positive or negative result to the check. During a comparison operation the maximum or minimum of the two inputs, depending on the comparison required, is copied onto the COMP_RES output. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.31.

```
---------------------------------------------
-- A[12]:          First input input
-- B[12]:          Second data input
-- EQUALITY:       Equality/Comparison switch
-- MAXIMUM:        Comparison value required
-- COMP_RES[12]:   Result of comparison operation
---------------------------------------------
```

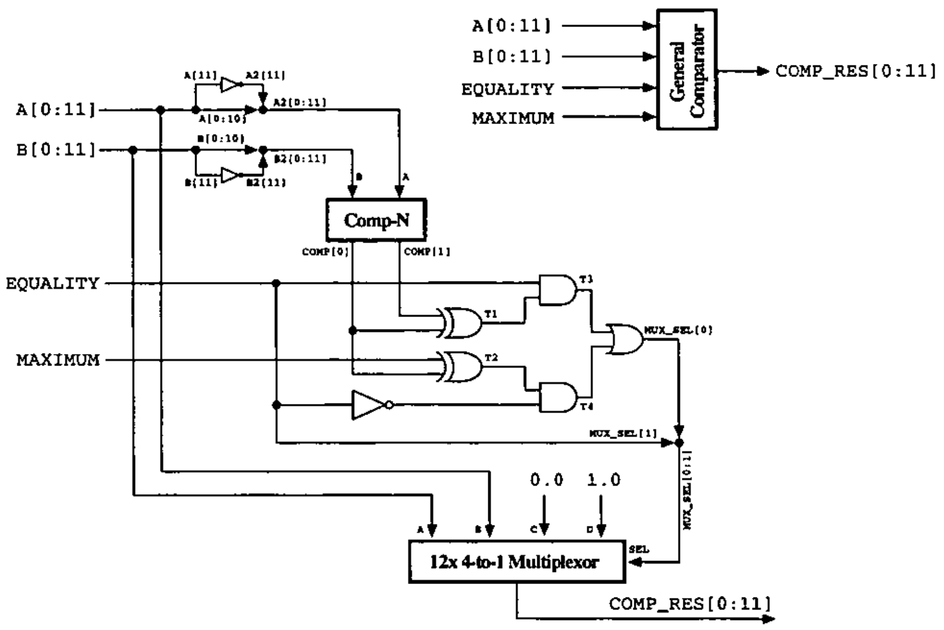


Figure A.31 General Purpose Comparator Unit

```
entity GP_COMPARATOR is
   port (A, B:      in  BIT_VECTOR(0 to 11);
         EQUALITY:  in  BIT;
         MAXIMUM:   in  BIT;
         COMP_RES:  out BIT_VECTOR(0 to 11))
end GP_COMPARATOR;

architecture GP_DATA_COMPARATOR of GP_COMPARATOR is
   component WORD_COMPARE
      generic (START, STOP: INTEGER);
      port (A, B: in BIT_VECTOR(START to STOP);
            COMP: out BIT_VECTOR(0 to 1));
   end component;
   component MUX_4
      port (SEL: in BIT_VECTOR(0 to 1); A, B, C, D: in BIT; OUT out: BIT);
   end component;
   component OR2
      port (A, B: in BIT; RES: out BIT);
   end component;
   component AND2
      port (A, B: in BIT; RES: out BIT);
   end component;
   component XOR2
      port (A, B: in BIT; RES: out BIT);
   end component;
   signal A2, B2: BIT_VECTOR(0 to 11);
   signal CTRL: BIT_VECTOR(0 to 11);
   signal MUX_SEL: BIT_VECTOR(0 to 1);
   signal T1, T2, T3, T4: BIT;
begin
   --------
   -- Get copies of inputs with inverted signs
   --------
   A2[0 to 10] <= A[0 to 10];
   A2[11] <= not A[11];
   B2[0 to 10] <= B[0 to 10];
   B2[11] <= not B[11];
   --------
   -- Feed these into the comparator
   --------
   CMP: WORD_COMPARE generic map (0, 11) port map (A2, B2, CTRL);
   --------
   -- Work out which result to use
   --------
   MUX_SEL[1] <= EQUALITY;
   XR1: XOR2 port map (CTRL[0], CTRL[1], T1);
   XR2: XOR2 port map (CTRL[0], MAXIMUM, T2);
   AN1: AND2 port map (T1, EQUALITY, T3);
   AN2: AND2 port map (T2, not EQUALITY, T4);
   OR1: OR2 port map (T3, T4, MUX_SEL[0]);
   --------
   -- Then use it
   --------
   RES1: for K in 0 to 6 generate
      MUX_4 port map (MUX_SEL, B[K], A[K], 0, 0, COMP_RES[K]);
   end generate RES1;
   RES2: MUX_4 port map (MUX_SEL, B[7], A[7], 0, 1, COMP_RES[7]);
   RES3: for K in 8 to 11 generate
      MUX_4 port map (MUX_SEL, B[K], A[K], 0, 0, COMP_RES[K]);
   end generate RES3;
end GP_DATA_COMPARATOR;
```

A.4.4 Fixed-Point Adder Unit

A.4.4.1 Radix-4 Adder Unit

This is a standard radix-4 adder. It adds a pair of 2-bit inputs, A and B, taking into account the value of any carry-in, as indicated by C_IN. The result of the addition, along with the resultant carry-out, is made available at SUM and C_OUT respectively. The VHDL code for this circuit is shown overleaf, and a schematic of the circuit itself is shown in figure A.32.

```
----------------------------------
-- A:     First input
-- B:     Second input
-- C_IN:  Carry-in to calculation
-- C_OUT: Carry-out of calculation
-- SUM:   Result of calculation
----------------------------------

entity RADIX_4 is
   port (A:     in  BIT_VECTOR(0 to 1);
         B:     in  BIT_VECTOR(0 to 1);
         C_IN:  in  BIT;
         C_OUT: out BIT;
         SUM:   out BIT_VECTOR(0 to 1));
end RADIX_4;
```

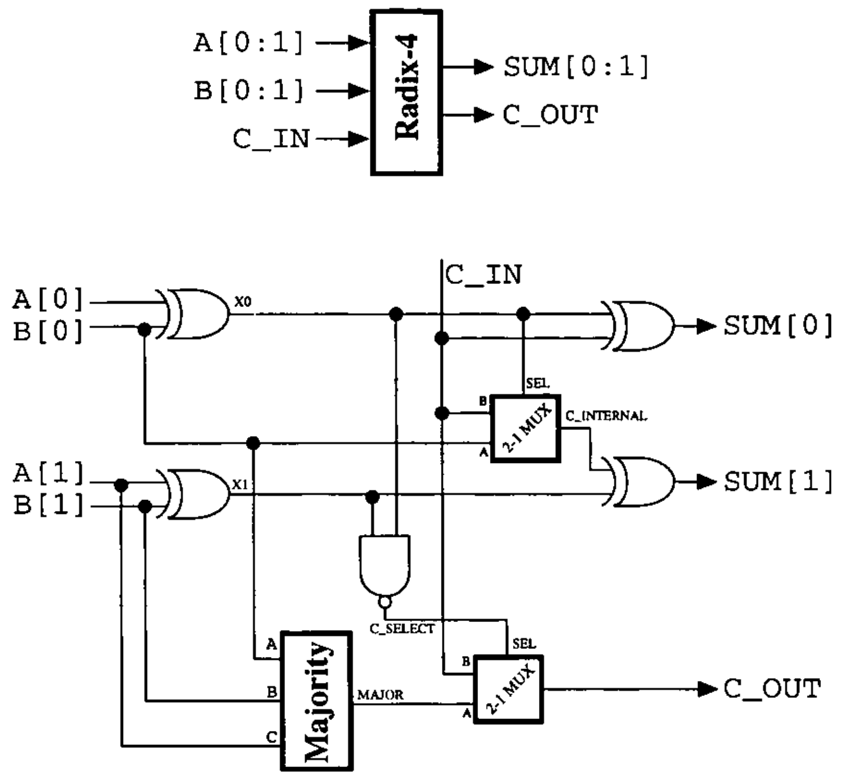


Figure A.32 Radix-4 Adder

```
architecture BASIC_RADIX_4 of RADIX_4 is
   component XOR2
      port (A, B: in BIT; RES: out BIT);
   end component;
   component NAND2
      port (A, B: in BIT; RES: out BIT);
   end component;
   component MAJORITY
      port (A, B, C: in BIT; RES: out BIT);
   end component;
   component MUX_2
      port (SEL, A, B: in BIT; OUT: out BIT);
   end component;
   signal X0, X1, MAJOR: BIT;
   signal C_INTERNAL, C_SELECT: BIT;
begin
   --------
   -- Generate SUM[2]
   --------
   XR1: XOR2 port map (A[0], B[0], X0);
   XR2: XOR2 port map (A[1], B[1], X1);
   MX1: MUX_2 port map (X0, B[0], C_IN, C_INTERNAL);
   XR3: XOR2 port map (X0, C_IN, SUM[0]);
   XR4: XOR2 port map (X1, C_INTERNAL, SUM[1]);
   --------
   -- Generate C_OUT
   --------
   MAJ: MAJORITY port map (A[1], B[1], B[0], MAJOR);
   NA1: NAND2 port map (X0, X1, C_SELECT);
   MX2: MUX_2 port map (C_SELECT, MAJOR, C_IN, C_OUT);
```

```
end BASIX_RADIX_4;
```

A.4.4.2 Adder Block Pair

This is a 4-bit adder. By cascading together two *Radix-4* adders allows creation of a 4-bit adder, which is a useful circuit in the generation of the full 12-bit carry-select adder. It simply joins two instances of a radix-4 adder together, with no further intermediate circuitry. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.33.

```
----------------------------------
-- A:      First input
-- B:      Second input
-- C_IN:   Carry-in to calculation
-- C_OUT:  Carry-out of calculation
-- SUM:    Result of calculation
----------------------------------

entity ADDER_PAIR is
   port (A:      in  BIT_VECTOR(0 to 3);
         B:      in  BIT_VECTOR(0 to 3);
         C_IN:   in  BIT;
         C_OUT:  out BIT;
         SUM:    out BIT_VECTOR(0 to 3));
end ADDER_PAIR;

architecture ADDER_4 of ADDER_PAIR is
   component RADIX_4
```

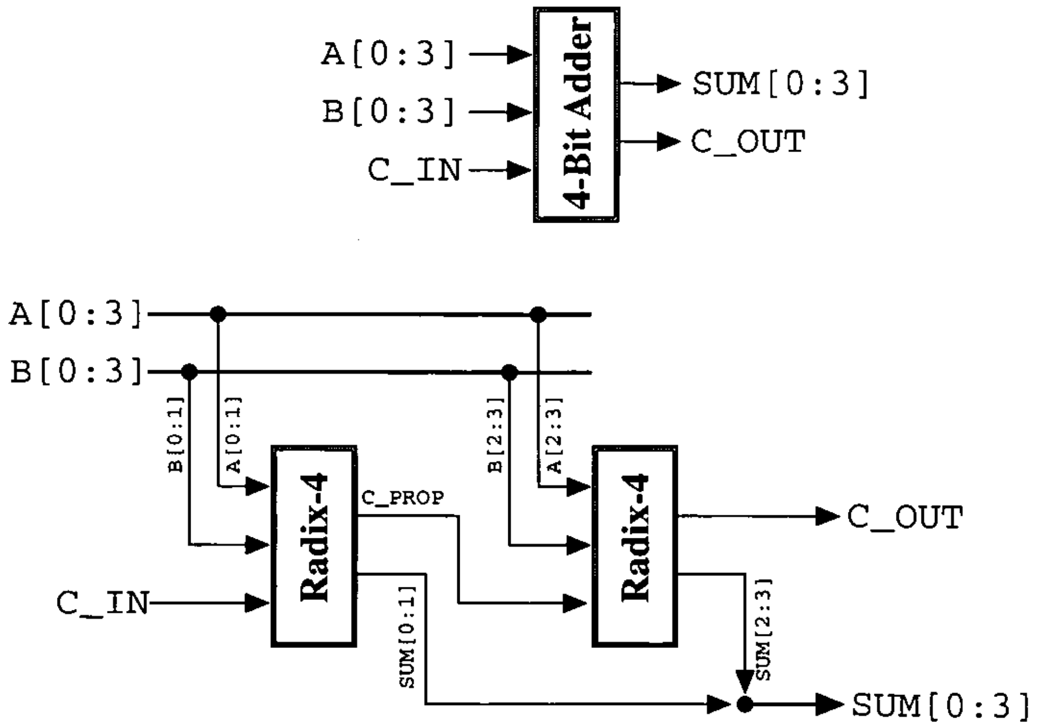


Figure A.33 4-Bit Basic Adder Block

```
      port (A, B: in BIT_VECTOR(0 to 1); C_IN: in BIT; C_OUT: out BIT;
            SUM: out BIT_VECTOR(0 to 1));
   end component;
   signal C_PROP: BIT;
begin
   --------
   -- Simply join two of them together
   --------
   AD1: RADIX_4 port map (A[0 to 1], B[0 to 1], C_IN, C_PROP, SUM[0 to 1]);
   AD2: RADIX_4 port map (A[2 to 3], B[2 to 3], C_PROP, C_OUT,SUM[2 to 3]);
end ADDER_4;
```

A.4.4.3 Carry-Select Adder Block

This is a section of a 4-bit carry-select adder. It consists of two instances of the *Adder Block Pair*, with one given `C_IN = 0` and the other `C_IN = 1`. The blocks add the same `A` and `B` data values. The block sum and carry outputs `SUM` and `CARRY` are selected depending on the value of `C_IN` for the entire block. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.34.

```
-------------------------------
-- A:     First input
-- B:     Second input
-- C_IN:  Carry-in to block
-- C_OUT: Carry-out of block
-- SUM:   Result of calculation
-------------------------------

entity CARRY_SELECT is
   port (A:      in  BIT_VECTOR(0 to 3);
         B:      in  BIT_VECTOR(0 to 3);
         C_IN:   in  BIT;
         C_OUT:  out BIT;
         SUM:    out BIT_VECTOR(0 to 3));
end CARRY_SELECT;

architecture BLOCK_CARRY_SELECT of CARRY_SELECT is
   component ADDER_PAIR
      port (A, B: in BIT_VECTOR(0 to 3); C_IN: in BIT; C_OUT: out BIT;
            SUM: out BIT_VECTOR(0 to 3));
   end component;
   component MUX_2
      port (SEL, A, B: in BIT; OUT out BIT);
   end component;
   signal C_LO, C_HI: BIT;
   signal SUM_LO, SUM_HI: BIT_VECTOR(0 to 3);
begin
   --------
   -- Setup a pair of adders with opposite C_IN
   --------
   AD1: ADDER_PAIR port map (A, B, 0, C_LO, SUM_LO);
   AD2: ADDER_PAIR port map (A, B, 1, C_HI, SUM_HI);
   --------
   -- Output the correct C_OUT and SUM with multiplexors
   --------
   CAR: MUX_2 port map (C_IN, C_LO, C_HI, C_OUT);
   BLK: for K in 0 to 3 generate
      SUM: MUX_2 port map (C_IN, SUM_LO[K], SUM_HI[K], SUM[K]);
   end generate BLK;
end BLOCK_CARRY_SELECT;
```

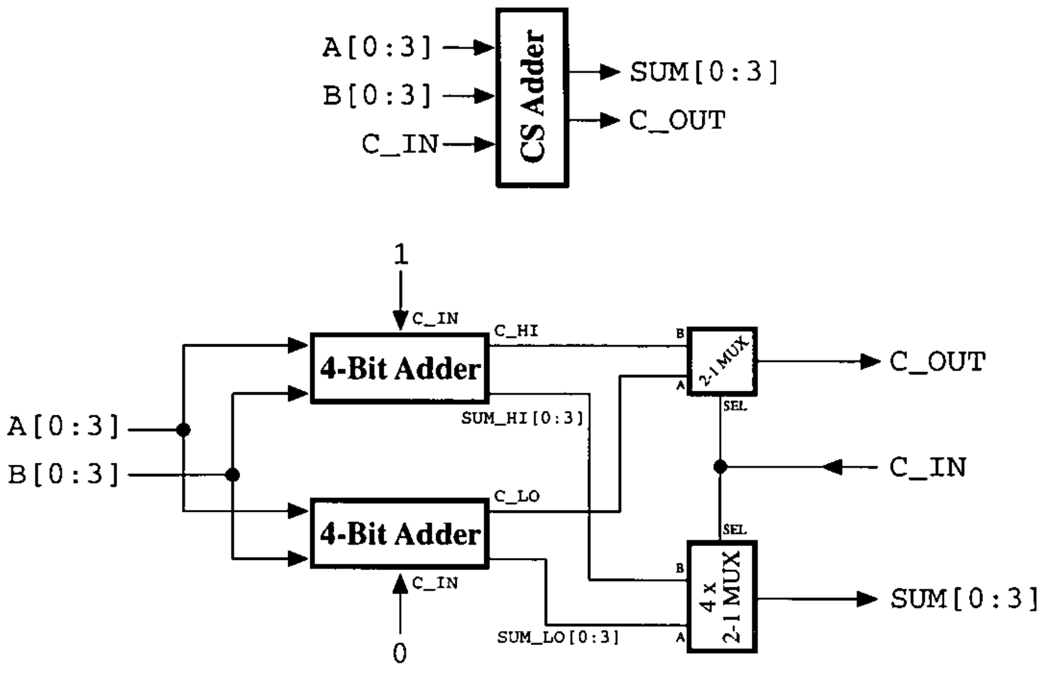


Figure A.34 Carry-Select Adder 4-Bit Block

A.4.4.4 Subtraction Control Unit

This is a control generator and data selector unit. It takes in the second input to the adder unit, B, and produces either a copy of it or the inverse of it in the output B_. This is done on the basis of the input ADD_SUB, which determines the operation carried out by the adder unit (1 => addition). The initial carry-in to the adder unit, C_IN, is also generated; if a subtraction operation is required then it is set to logic-1, otherwise it is set to logic-0. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.35.

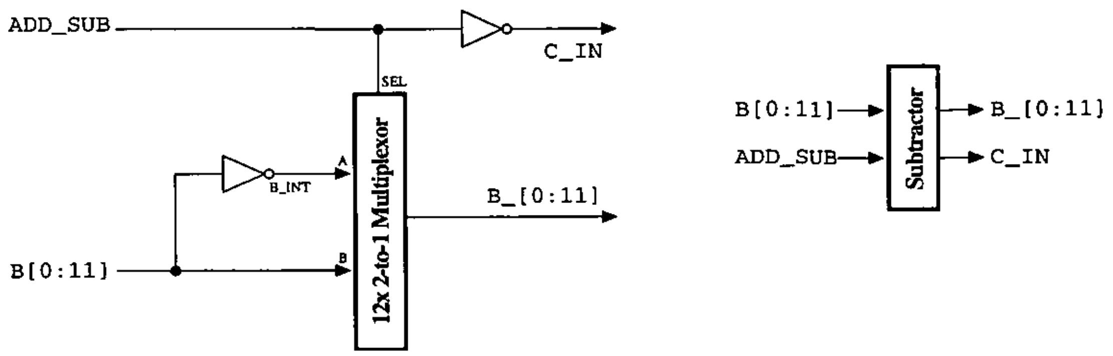


Figure A.35 Subtraction Control Unit

```
----------------------------------------------
-- B:       Second input to the adder unit
-- B_:      Modified second input to the adder
-- ADD_SUB: Required adder operation
-- C_IN:    Initial carry-in to the adder
----------------------------------------------

entity SUBTRACT_CONTROL is
   port (B:         in  BIT_VECTOR(0 to 11);
         ADD_SUB:   in  BIT;
         B_:        out BIT_VECTOR(0 to 11);
         C_IN:      out BIT);
end SUBTRACT_CONTROL;

architecture ADD_SUB_CONTROL of SUBTRACT_CONTROL is
   component MUX_2
      port (SEL: in BIT; A, B: in BIT; OUT: out BIT);
   end component;
   signal B_INT: BIT_VECTOR(0 to 11);
begin
   --------
   -- Invert all of B
   --------
   SGN: for K in 0 to 11 generate
      B_INT(K) <= not B(K);
   end generate SGN;
   --------
   -- Select correct B or B_INT
   --------
   BLK: for K in 0 to 11 generate
      MX: MUX_2 port map (ADD_SUB, B_INT(K), B(K), B_(K));
   end generate BLK;
   --------
   -- Finally, do the carry
   --------
   C_IN <= not ADD_SUB;
end ADD_SUB_CONTROL
```

A.4.4.5 Overflow/Underflow and Integrity Handler

This is a data selector unit. This unit will output either the addition result or the maximum/minimum representable number. This is done using the integrity flags for both inputs (A_INT and B_INT), the sign values of both inputs (A_SIGN and B_SIGN) and the sign value of the result of the addition (RES[11]). The results of various pieces of combinational logic are used to drive the selectors of a set of *4-to-1 Multiplexors*, which selects the correct output. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.36.

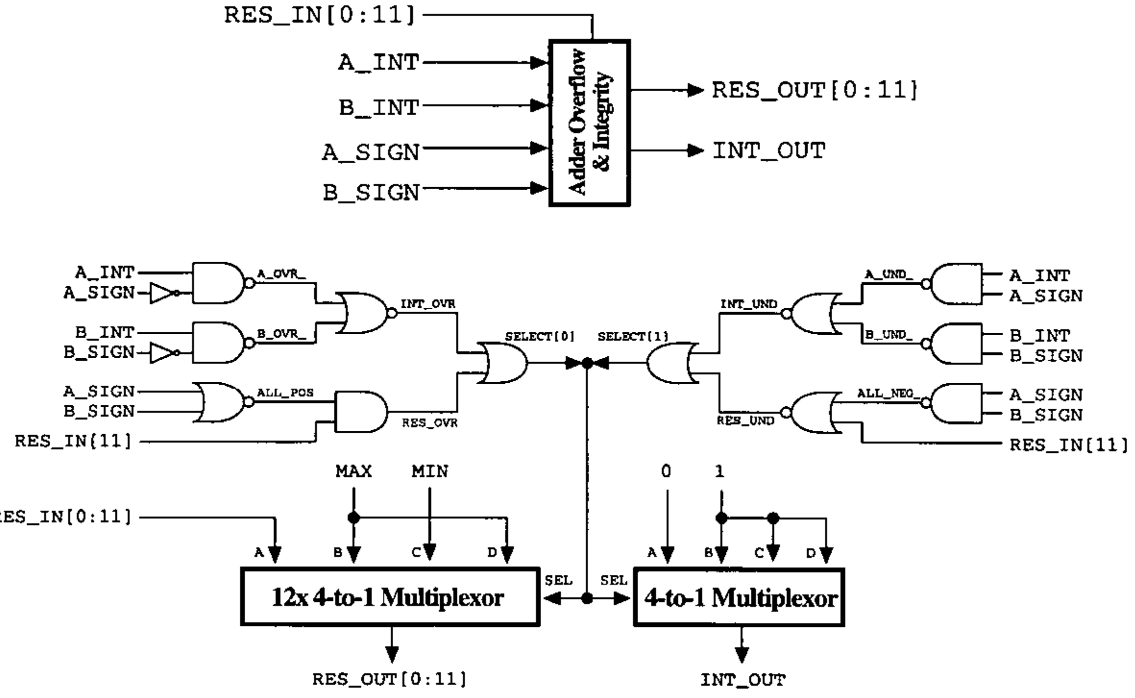


Figure A.36 Adder Overflow/Underflow & Integrity Handler

```
---------------------------------------------
-- A_INT:   Integrity status of adder input A
-- B_INT:   Integrity status of adder input B
-- A_SIGN:  Sign value of adder input A
-- B_SIGN:  Sign value of adder input B
-- RES_IN:  Result of the addition
-- RES_OUT: Output of unit
-- INT_OUT: Integrity of the unit output
---------------------------------------------

entity ADDER_OVERFLOW is
   port (A_INT:    in  BIT;
         B_INT:    in  BIT;
         A_SIGN:   in  BIT;
         B_SIGN:   in  BIT;
         RES_IN:   in  BIT_VECTOR(0 to 11);
         RES_OUT: out BIT_VECTOR(0 to 11);
         INT_OUT: out BIT);
end ADDER_OVERFLOW;

architecture ADDITION_OVERFLOW of ADDER_OVERFLOW is
   component MUX_4
      port (SEL: in BIT_VECTOR(0 to 1); A, B, C, D: in BIT;
            OUT: out BIT);
   end component;
   component NAND2
      port (A, B: in BIT; RES out BIT);
   end component;
   component AND2
      port (A, B: in BIT; RES out BIT);
   end component;
   component NOR2
      port (A, B: in BIT; RES out BIT);
   end component;
   component OR2
      port (A, B: in BIT; RES out BIT);
   end component;
   signal A_OVR_, B_OVR_, A_UND_, B_UND_: BIT;
   signal INT_OVR, INT_UND, RES_OVR, RES_UND: BIT;
   signal ALL_POS, ALL_NEG_: BIT;
   signal SELECT: BIT_VECTOR(0 to 1);
begin
   --------
   -- Check for overflow possibilities
   --------
   NA1: NAND2 port map (A_INT, not(A_SIGN), A_OVR_);
   NA2: NAND2 port map (B_INT, not(B_SIGN), B_OVR_);
   NR1: NOR2 port map (A_OVR_, B_OVR_, INT_OVR);
   NR2: NOR2 port map (A_SIGN, B_SIGN, ALL_POS);
   AN1: AND2 port map (ALL_POS, RES[11], RES_OVR);
   OR1: OR2 port map (INT_OVR, RES_OVR, SELECT[0]);
   --------
   -- Check for underflow possibilities
   --------
   NA3: NAND2 port map (A_INT, A_SIGN, A_UND_);
   NA4: NAND2 port map (B_INT, B_SIGN, B_UND_);
   NR3: NOR2 port map (A_UND_, B_UND_, INT_UND);
   NA4: NAND2 port map (A_SIGN, B_SIGN, ALL_NEG_);
   NR4: NOR2 port map (ALL_NEG_, RES[11], RES_UND);
   OR2: OR2 port map (INT_UND, RES_UND, SELECT[1]);
   --------
   -- Now select an integrity and a result sign/value
   --------
   MX1: MUX_4 port map (SELECT, 0, 1, 1, 1, INT_OUT);
   MX2: MUX_4 port map (SELECT, RES_IN[11], 0, 1, 0, RES_OUT[11]);
   SEL: for K in 0 to 10 generate
      MX3: MUX_4 port map (SELECT, RES_IN[K], 1, 0, 1, RES_OUT[K]);
   end generate SEL;
end ADDITION_OVERFLOW;
```

A.4.4.6 Complete Adder Unit

This is the entire 12-bit fixed-point adder unit. It takes in two 12-bit numbers, A and B, and their respective integrity flags, A_INT and B_INT, and produces a 12-bit output RES and an associated integrity flag INTEG. The operation carried out is A+B, unless the input ADD is logic-0, in which case the operation is A-B. The integrity circuit can be bypassed entirely by setting the input RAW to logic-1; this ignores the bypass function (and sets INTEG to logic-0), but means that the VHDL compiler would not have to generate the unit at all if this input was a constant logic-1 at the instantiation of the adder unit. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.37.

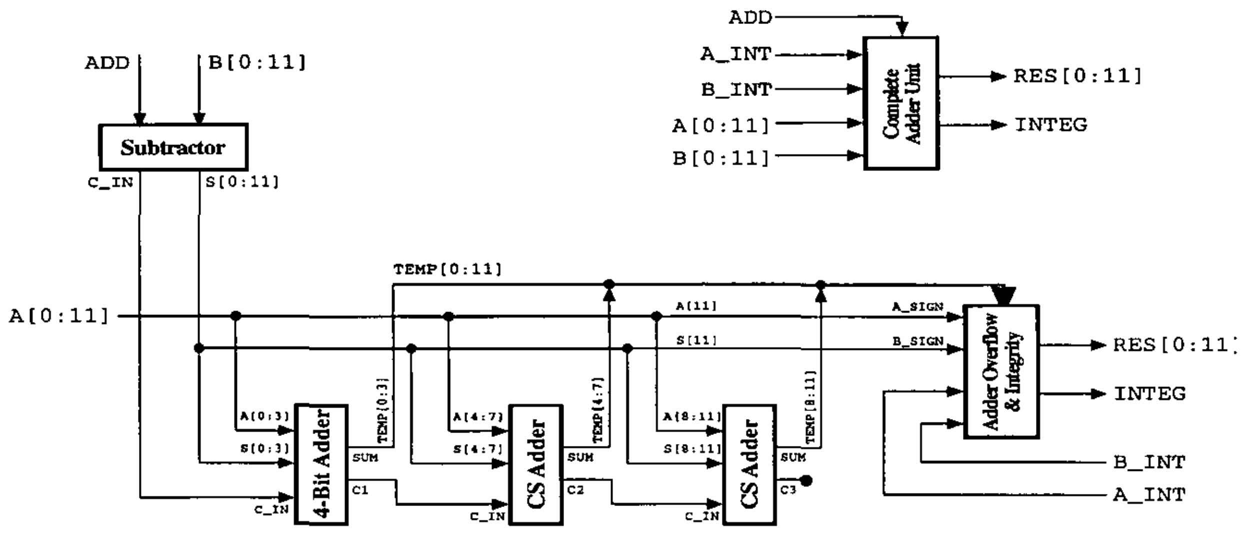


Figure A.37 Complete Adder Unit

```
-----------------------------------------
-- A:     Adder input A
-- B:     Adder input B
-- ADD:   Indicates if op is subtraction
-- RAW:   Pay no attention to integrity
-- A_INT: Integrity status of input A
-- B_INT: Integrity status of input B
-- RES:   Final result of A+B operation
-- INTEG: Integrity status of adder output
-----------------------------------------

entity WHOLE_ADDER is
   port (A:     in  BIT_VECTOR(0 to 11);
         B:     in  BIT_VECTOR(0 to 11);
         ADD:   in  BIT;
         RAW:   in  BIT;
         A_INT: in  BIT;
         B_INT: in  BIT;
         RES:   out BIT_VECTOR(0 to 11);
         INTEG: out BIT);
```

```
end WHOLE_ADDER;

architecture WHOLE_ADDER_UNIT of WHOLE_ADDER is
   component ADDER_OVERFLOW
      port (A_INT, B_INT, A_SIGN, B_SIGN: in BIT;
            RES_IN: in BIT_VECTOR(0 to 11);
            RES_OUT: out BIT_VECTOR(0 to 11); INT_OUT: out BIT);
   end component;
   component SUBTRACT_CONTROL is
      port (B: in BIT_VECTOR(0 to 11); ADD_SUB: in BIT;
            B_: out BIT_VECTOR(0 to 11); C_IN: out BIT);
   end component;
   component CARRY_SELECT
      port (A, B: in BIT_VECTOR(0 to 3); C_IN: in BIT;
            C_OUT: out BIT; SUM: out BIT_VECTOR(0 to 3));
   end component;
   component ADDER_PAIR
   port (A, B: in BIT_VECTOR(0 to 3); C_IN: in BIT;
         C_OUT: out BIT; SUM: out BIT_VECTOR(0 to 3));
   end component;
   signal TEMP: BIT_VECTOR(0 to 11);
   signal S: BIT_VECTOR(0 to 11);
   signal C_IN, C1, C2, C3: BIT;
begin
   --------
   -- Sort out carry-in and subtraction control
   --------
   S1: SUBTRACT_CONTROL port map (B, ADD, S, C_IN);
   --------
   -- Feed all of the adder units
   --------
   A1: ADDER_PAIR port map (A[0 to 3], S[0 to 3], C_IN, C1, TEMP[0 to 3]);
   A2: CARRY_SELECT port map (A[4 to 7], S[4 to 7], C1, C2, TEMP[4 to 7]);
   A3: CARRY_SELECT port map (A[8 to 11], S[8 to 11],C2,C3, TEMP[8 to 11]);
   --------
   -- Sort out the correct output and integrity
   --------
   if (RAW = "1") then
      OVR: ADDER_OVERFLOW port map (A_INT, B_INT, A[11], S[11], TEMP,
                                   RES, INTEG);
   else
      RES := TEMP;
      INTEG := "0";
   end if;
end WHOLE_ADDER_UNIT;
```

A.4.5 Fixed Point Multiplier Unit

A.4.5.1 Modified Booth's Multiplier

Unlike the majority of the VHDL units the multiplier is described purely in behavioural terms rather than in structural terms. A full description of an optimised multiplier circuit would take many pages of VHDL (and many pages of circuit layout). The VHDL code below gives a generic multi-bit multiplier description, but the schematic shown in figure A.38 shows an optimised circuit for a 2-bit multiplier [NaJo97] based upon the same algorithm

```
--------------------------------
-- A:   Multiplier input A
-- B:   Multiplier input B
-- RES: Result of A * B operation
-- CLK: Clock input
--------------------------------

entity BOOTH_MULT is
   port (A:   in  BIT_VECTOR(0 to 11);
         B:   in  BIT_VECTOR(0 to 11);
         CLK: in  BIT;
         RES: out BIT_VECTOR(0 to 22));
end BOOTH_MULT;

architecture BOOTH_MULT_UNIT of BOOTH_MULT is
   component WHOLE_ADDER
      port (A, B: in  BIT_VECTOR(0 to 11); ADD, RAW, A_INT, B_INT: in  BIT;
         RES: out BIT_VECTOR(0 to 11); INTEG: out BIT);
   end component
   --------
   -- Reverse variable ranges (so later arithmetic shift makes sense)
   --------
   variable P:        BIT_VECTOR(11 downto 0);      -- same size as 'A'
   variable AX:       BIT_VECTOR(12 downto 0);      -- size is 'A' + 1
   variable S:        BIT_VECTOR(24 downto 0);      -- size is 'P' + 'AX'
   variable ACTION:   BIT_VECTOR(1 downto 0);
   variable COUNT:    INTEGER range A;
   signal IGNORE:     BIT;
begin
   --------
   -- Wait until low edge of clock
   -- (i.e. until inputs valid)
   -- and initialise circuit
   --------
   wait until (CLK = '0');
   P := (others => 0);
   AX := A'reverse_range & '0';
   COUNT := 0;

   --------
   -- Do algorithm for each bit in 'A'
   --------
   BOOTH: loop
      --------
      -- Work out if we add, subtract or do nothing
      --------
      case AX(1 downto 0) is
         when "01" =>   ACTION := "11";   -- addition required
         when "10" =>   ACTION := "10";   -- subtraction required
         when others => ACTION := "00";   -- no action required
      end case;
      --------
      -- Do addition/subtraction as required (ignoring integrity)
      --------
      if ACTION(1) = "1" then
         ADD1: WHOLE_ADDER (P'reverse_range, B'reverse_range,
                                          ACTION(0), 1, 0, 0, P, IGNORE);
      end if;
      --------
      -- Shift internal values for next cycle
      --------
      S := P & AX;                    -- contatenate P and AX
      S := ARITH_SHIFT_RIGHT(S, 1);   -- shift right (keeping sign)
      P := S(24 downto 13);           -- reassign P
      AX := S(AX'range);              -- reassign AX
      --------
      -- Finished ?
      --------
      COUNT := COUNT + 1;
      exit BOOTH when COUNT = A'length;

   end loop;

   --------
   -- Assign final result
   --------
   RES <= S(23 downto 1)'reverse_range;
end BOOTH_MULT_UNIT;
```

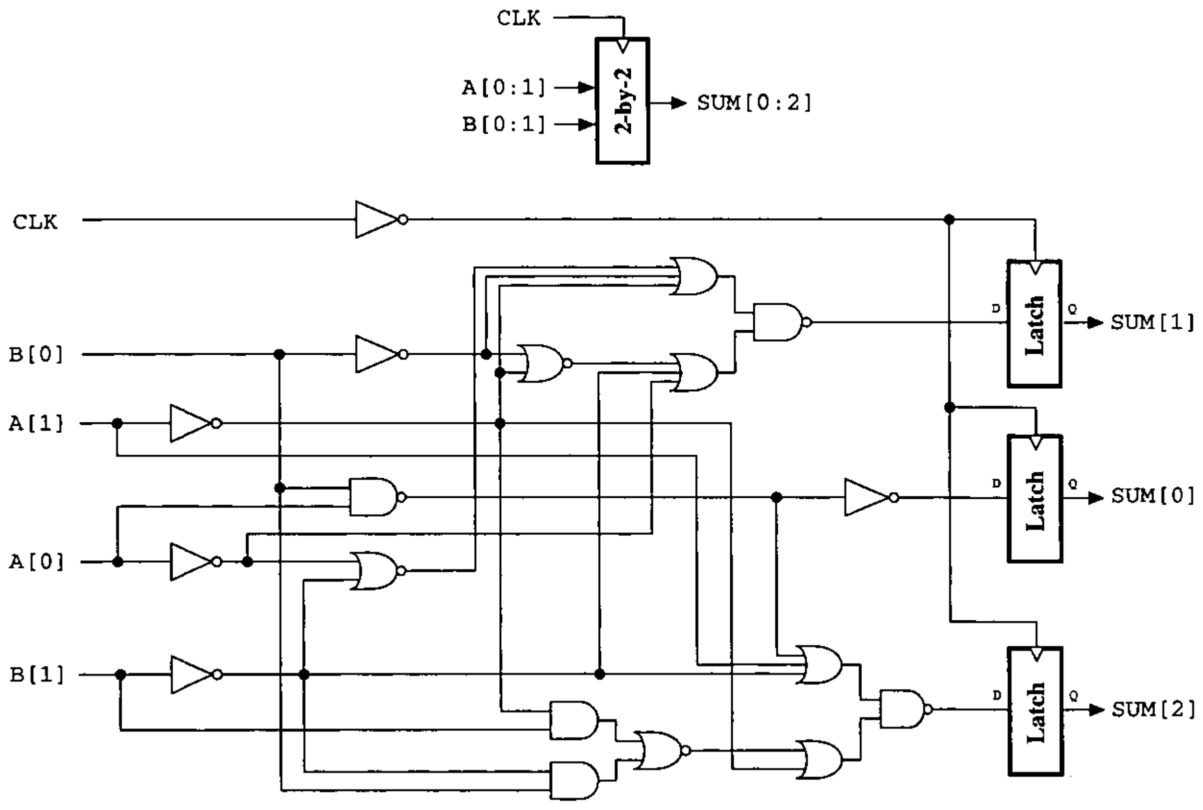


Figure A.38 2x2 Optimised Booth Multiplier

A.4.5.2 Result Integrity Handler

Due to the number format used within the ISA processor the 23-bit output from the basic multiplier is defined as a sign bit {22}, eight integer bits {21..14} and 14 fractional bits {13..0}. A valid 12-bit number must be extracted from this and an integrity rating assigned to it - this rating is based upon the two input vectors A and B, the result RES and the integrity flags associated with A and B. The VHDL for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.39.

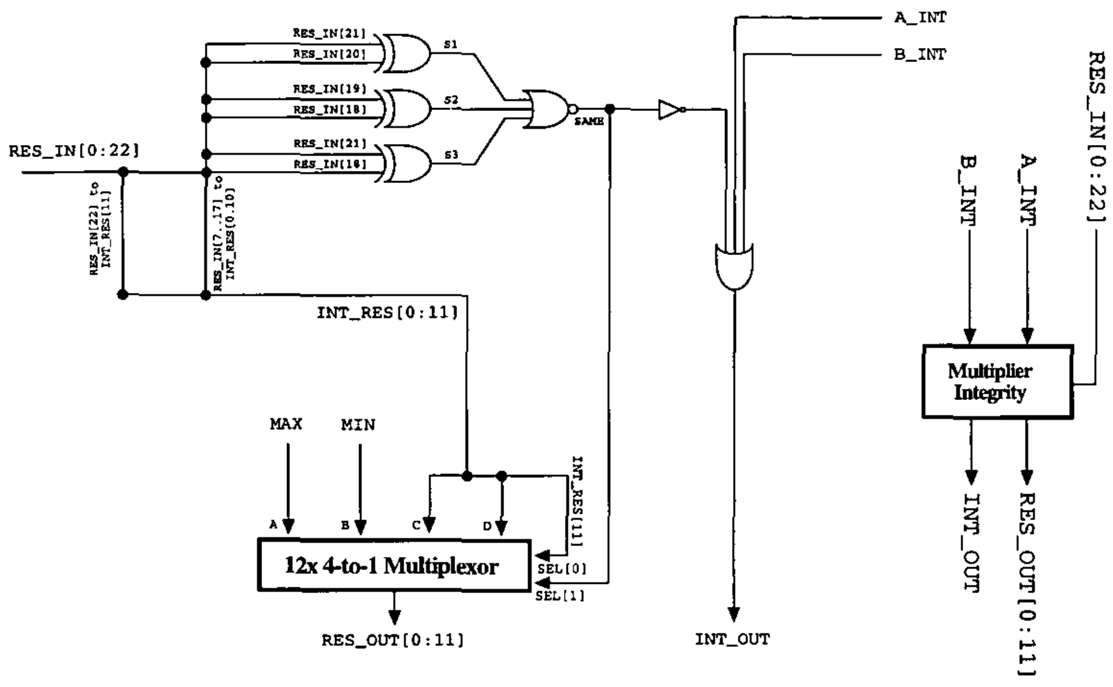


Figure A.39 Multiplier Integrity Checker

```
-----------------------------------------------
-- A_INT:    Integrity state of multiplier input A
-- B_INT:    Integrity state of multiplier input B
-- RES_IN:   Full Result of the multiplication
-- RES_OUT:  Output of unit
-- INT_OUT:  Integrity of the unit output
-----------------------------------------------

entity MULT_INTEGRITY is
   port (A_INT:    in  BIT;
         B_INT:    in  BIT;
         RES_IN:   in  BIT_VECTOR(0 to 22);
         RES_OUT:  out BIT_VECTOR(0 to 11);
         INT_OUT:  out BIT);
end MULT_INTEGRITY;

architecture MULT_INTEGRITY_UNIT of MULT_INTEGRITY is
   component MUX_4
      port (SEL: in BIT_VECTOR(0 to 1); A, B, C, D: in BIT;
            OUT: out BIT);
    end component;
    component XOR2
        port (A, B: in BIT; RES out BIT);
    end component;
    component NOR3
        port (A, B, C: in BIT; RES out BIT);
    end component;
    component OR3
        port (A, B, C: in BIT; RES out BIT);
    end component;
    signal S1, S2, S3: BIT;
    signal SAME: BIT;
    signal SEL: BIT_VECTOR(0 to 1);
begin
    --------
    -- See if any upper bits are different
    --------
    XR1: XOR2 port map (RES_IN[21], RES_IN[20], S1);
    XR2: XOR2 port map (RES_IN[19], RES_IN[18], S2);
    XR3: XOR2 port map (RES_IN[21], RES_IN[18], S3);
    NR1: NOR3 port map (S1, S2, S3, SAME);
    --------
    -- Map 23-bit to 12-bit and set integrity
    --------
    INT_RES[11] <= RES_INT[22];
    INT_RES[0 to 10] <= RES_INT[7 to 17];
    OR1: OR3 port map (not(SAME), A_INT, B_INT, INT_OUT);
    --------
    -- Now select a valid (max/min/res/res) result
    --------
    SEL: for K in 0 to 10 generate
        MX1: MUX_4 port map (SEL, 1, 0, INT_RES[K], INT_RES[K], RES_OUT[K]);
    end generate SEL;
    RES_OUT[11] <= INT_RES[11];
end MULT_INTEGRITY_UNIT;
```

A.4.5.3 Complete Multiplier Unit

This is the entire 12-bit fixed-point multiplier unit. It takes in two 12-bit numbers, A and B, along with their associated integrity values A_INT and B_INT, and produces a 12-bit output RES and an associated integrity value INTEG. The VHDL code for this circuit is shown below, and a schematic of the circuit itself is shown in figure A.40.

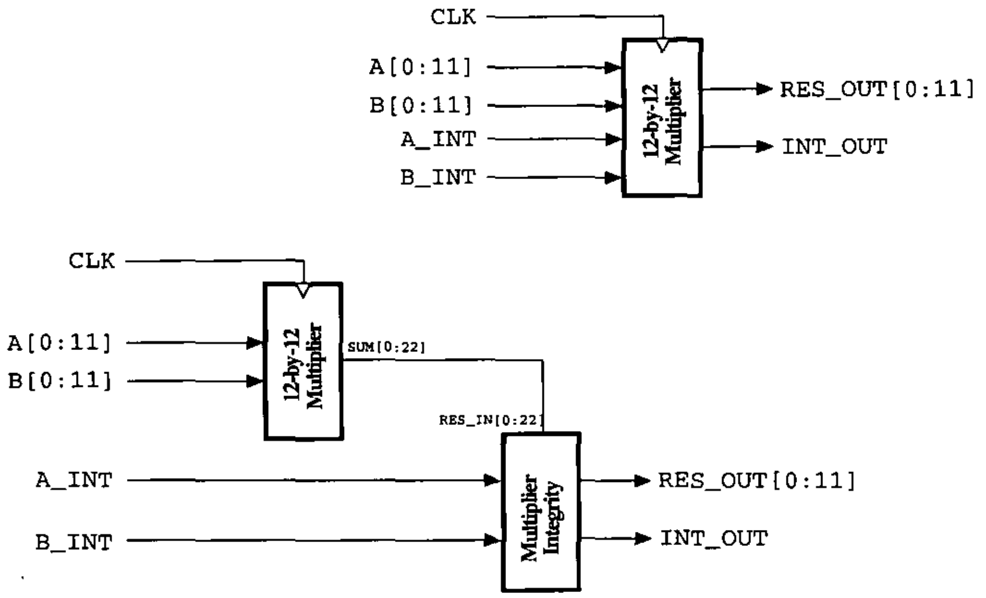


Figure A.40 Complete 12x12 Parallel Multiplier

```
---------------------------------
-- A:     Multiplier input A
-- B:     Multiplier input B
-- A_INT: Integrity of input A
-- B_INT: Integrity of input B
-- RES:   Result of A * B operation
-- INTEG: Integrity of result
-- CLK:   Clock input
---------------------------------

entity WHOLE_MULT is
   port (A:     in  BIT_VECTOR(0 to 11);
         B:     in  BIT_VECTOR(0 to 11);
         A_INT: in  BIT;
         B_INT: in  BIT;
         CLK:   in  BIT;
         RES:   out BIT_VECTOR(0 to 11);
         INTEG: out BIT);
end WHOLE_MULT;

architecture WHOLE_MULT_UNIT of WHOLE_MULT is
   component BOOTH_MULT
      port (A, B: in  BIT_VECTOR(0 to 11); CLK: in BIT;
            RES: out BIT_VECTOR(0 to 22));
   end component
   component MULT_INTEGRITY is
   port (A_INT, B_INT: in  BIT; RES_IN: in BIT_VECTOR(0 to 22);
         RES_OUT: out BIT_VECTOR(0 to 11); INT_OUT: out BIT);
   end component;
   signal TEMP_RES: BIT_VECTOR(0 to 22);
begin
   --------
   -- Simple mapping of two components
   --------
   PT1: BOOTH_MULT port map (A, B, CLK, TEMP_RES);
   PT2: MULT_INTEGRITY port map (A_INT, B_INT, TEMP_RES, RES, INTEG);
end WHOLE_MULT_UNIT;
```

REFERENCES

[Ahme82] H.M. Ahmed, J.M. Delosme and M. Morf, 'Highly concurrent computing structures for matrix arithmetic and signal processing', in: *Computing*, **15**, pp 65-82, 1982

[Akin84] J.M. Akinpela, 'Overload performance of engineering networks with non-hierarchial and hierarchial routing', in: *AT&T Bell Labs Technical Journal*, **63**(7), pp 1261-1281, 1984

[AlMo90] I. Aleksander and H. Morton, *An Introduction to Neural Computing*, UK : Chapman and Hall, 1990

[AlSt91] C. Alippi and G. Storti-Gajani, 'Simple approximation of sigmoidal functions: realistic design of digital neural networks capable of learning', in: *Proceedings ISCAS '91 (Singapore)*, USA : IEEE Press, pp 1505-1508, 1991

[Ambl92] A.P. Ambler, *CAD-I*, Unpublished Lecture Notes, Dept. Electrical Engineering, Brunel University, Uxbridge, England, 1992

[AmMu88] A.P. Ambler and G. Musgrave, 'Design for testability in the digital environment', in: *New Electronics*, pp. 43-44, February 1988

[Ansa96] N. Ansari, A. Arulambalam and S. Balasekar, 'Traffic management of a satellite communication network using stochastic optimisation', in: *IEEE Transactions on Neural Networks*, **7**(3), pp 732-744, 1996

[Begu93] A. Beguelin et al, *PVM 3 User's Guide and Reference Manual*, Oak Ridge National Laboratory, Oakridge, Tennessee, USA, 1993

[Beiu92] V. Beiu, J.A. Peperstraete and R. Lauwereins, 'Using threshold gates to implement sigmoidal non-linearity', in: *Proceedings of the International Conference on Artificial Neural Networks*, Amsterdam : Elsevier Science Publications, **2**, pp 1447-1450, 1992

[BeNe71] C.G. Bell and A. Newell, *Computer Structures : Readings and Examples*, USA : McGraw Hill, 1971

[Bézi70] P. Bézier, *Emploi des Machines á Commande Numérique*, France : Masson et Cie, 1970. Translated by A.R. Forrest and A.F. Pankhurst *Numerical Control - Mathematics and Applications*, UK : Wiley, 1972

[Boot51] A.D. Booth, 'A signed multiplication technique', *Quarterly Journal of Mathematics*, **4**, Part 2, 1951

[Burg84] R. Burger, R.K.Calvin III, W.C. Holton and L.W. Sumney, 'The impact of IC's on computer technology', in: *IEEE Computer*, **17**, pp 88-95, October 1984

[Burr87] D.J. Burr, 'Experiments with a connectionist text reader', in: *Proceedings of the IEEE First International Conference on Neural Networks*, (M. Caudill and C. Butler eds.), **4**, pp 717-724, USA : SOS Printing, 1987

[CaGr87a] G. Carpenter and S. Grossberg, 'A massively parallel architecture for a self-organizing neural pattern recogition machine', in: *Computer Vision, Graphics and Image Processing*, **37**, pp 54-115, 1987

[CaGr87b] G. Carpenter and S. Grossberg, 'ART2 : self organisation of stable category recognition codes for analog input patterns', in: Applied Optics, **26**(23), pp 4919-4930, 1987

[Camp87] S.J. Campanella, 'Small, low-cost earth stations : a major trend', in: *IEEE Spectrum*, **24**, pp 42-43, January 1987

[Camp96] S.A. Campbell, *The Science and Engineering of Microelectronic Fabrication*, UK : Oxford University Press, 1996

[Chow83] T.P. Chow, 'A review of refractory gates for MOS VLSI', in: *International Electronic Devices Meeting (Technical Digest)*, pp 513-517, December 1983

[CoGr83] M.A. Cohen and S.G. Grossberg, 'Absolute stability of global pattern formation and parallel memory storage by competitive neural networks', in: *IEEE Transactions on Systems, Man and Cybernetics*, **13**, pp 815-826, 1983

[Cosn86a] M. Cosnard, M. Daoudi, J.M. Muller and Y. Robert, 'On parallel and systolic Givens factorisation of dense matrices', in: *Parallel Algorithms and Architectures*, (M. Cosnard, P. Quinton, Y. Robert and M. Tchuente eds.), France : North Holland, 1986

[Cosn86b] M. Cosnard, Y. Robert and M. Tchuente, 'Matching parallel algorithms with parallel architectures : a case study', in: *IFIP Working Conference on Highly Parallel Computers for Numerical and Signal Processing Applications*, IFIP WG 10-3, France, 1986

[Cvet86] Z. Cvetanovic, 'Performance analysis of FFT algorithm on a shared-memory parallel architecture', in: *IBM Journal of Research and Development*, 1986

[Darn86] J. Darnell, H. Lodish and D. Baltimore, *Molecular Cell Biology*, USA : American Scientific Books, 1986

[DeBl90] M. DeBlasi, *Computer Architecture*, UK : Addison Wesley, 1990

[Denn73] R.H. Dennard et al., in: *Semiconductors Silicon Electrochemical Society*, H.R. Duff and R.R. Burgess, Editors, 1973

[Denn74] R.H. Dennard et al., *IEEE Journal of Solid-State Circuits*, **SC-9**, 1974

[Denni75] J.B. Dennis and D.P. Misunas, 'A preliminary architecture for a basic data flow processor', in: *Proceedings of the Second Annual Synopsium on Computer Architecture*, pp 126-132, USA : IEEE Press, January 1975

[Desi88] D. DeSieno, 'Adding a conscience to competitive learning', in: *Proceedings of the IEEE International Conference on Neural Networks*, pp 117-124, USA : SOS Printing, 1988

[Dunc90] R. Duncan, 'A survey of parallel computer architectures', in: *IEEE Computer*, **23**(Feb), pp 5-16, 1990

[Eldr59] R.D. Eldred, 'Test routines based on symbolic logic statements', in: *Journal of ACM*, **6**(1), pp 36-39, January 1959

[EsDu82] D.B. Estreich and R.W. Dutton, 'Modelling latch-up in CMOS integrated circuits and systems', in: *IEEE Transactions on CAD*, **CAD-1**(4), pp 157-162, October 1982

[Evan91] D.J. Evans, *Systolic Algorithms*, UK : Gordon and Breach Science Publications, 1991

[Feng77] T.Y. Feng, ed., 'Parallel Processors and Processing', special issue, *ACM Computing Surveys*, **9**(1), 1977

[Flyn66] M.J. Flynn, 'Very high speed computing systems', in: *Proceedings of IEEE*, **54**, pp 1901-1909, 1966

[Fole90] J.D. Foley, A. van Dam, S.K. Feiner and J.F. Hughes, *Computer Graphics - Principles and Practice*, 2nd Edition, USA : Addison Wesley, 1990

[GeGe84] S. Gemen and D. Gemen, 'Stochastic relaxation, Gibbs distributions and Beyesian restoration of images', in: *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **6**, pp 721-741, 1984

[GeKu81] W.M. Gentleman and H.T. Kung, 'Matrix triangularization by systolic arrays', in: *Proceedings of the Society of Photo-optical Instrumentation Engineers*, **298**, pp 19-26, 1981

[GiLe80] J.F. Gibbons and K.F. Lee, in: *Electron Device Letters*, **EDL-1**, p 117, 1980

[Gill92] P.E. Gill, W. Murray and M.H. Wrights, *Practical Optimization*, UK : Academic Press, 1992

[Gros69] S. Grossberg, 'Some networks that can learn, remember and reproduce any number of complex mathematical space-time patterns', in: *Journal of Mathematics and Mechanics*, **19**, pp 53-91, 1969

[Gros87] S. Grossberg, 'Competitive learning : from interactive activation to adaptive resonance', in: *Cognitive Science*, **11**, pp 23-63, 1987

[Hamm91] D. Hammerstrom, 'A highly parallel digital architecture for neural network emulation', in: *VLSI for Artificial Intelligence and Neural Networks*, (J.G. Delgado-Frias and W. Moore eds.), pp 357-366, USA : Plenum Press, 1991

[Hast55] C. Hastings, *Approximations for Digital Computers*, USA : Princetown University Press, 1955

[Hayk94] S. Haykin, *Neural Networks : A Comprehensive Foundation*, USA : Macmillan College Publishing, 1994

[Hebb49] D.O. Hebb, *Organisation of Behaviour*, USA : Science Editions, 1949

[Hech87] R. Hecht-Nielsen, 'Counterpropagation networks', in: *Proceedings of the IEEE First International Conference on Neural Networks*, (M. Caudill and C. Butler eds.), pp 19-32, USA : SOS Printing, 1987

[HeSn82] K.S. Hedlund and L. Snyder, *Proceedings of International Conference on Parallel Processing*, August 1982

[Hira90] M. Hirayama, M. Ohmori and K. Yamasaki, 'GaAs LSI Fabrication and Performance', in: *Semiconductors and Semi-Metals*, **29**, (T. Ikoma ed.), also known as *Very High Speed Integrated Circuits : Gallium Arsenide LSI*, UK : Academic Press, 1990

[Hopf82] J.J. Hopfield, 'Neural networks and physical systems with emergent collective computational abilities', in: *Proceedings of the National Academy of Science*, **79**, pp 2554-2558, 1982

[Hopf84] J.J. Hopfield, 'Neurons with graded response have collective computational properties like those of two-state neurons', in: *Proceedings of the National Academy of Science*, **81**, pp 3088-3092, 1984

[HoTa85] J.J. Hopfield and D.W. Tank, 'Neural computation of decisions in optimization problems', in: *Biological Cybernetics*, **52**, pp 141-152, 1985

[HwBr84] K. Hwang and F.A. Briggs, *Computer Architecure and Parallel Processing*, USA : McGraw Hill, 1984

[HwNi80] K. Hwang and L.M. Ni, 'Resource optimisation of a parallel computer for multiple vector processing', in: *IEEE Transactions on Computers*, **C-29**, pp 831-836, September 1980

[IEDM83] _________, 'Device technology - isolation and dielectrics', in: *International Electronic Devices Meeting (Technical Digest)*, pp 19-46, December 1983

[IEEE85] _________, *IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*, 1985

[IEEE89] _________, *IEEE Standard 1149.1 - Standard Test Access Port and Boundary Scan Architecture*, 1989

[KaEv96] A.J. Kane and D.J. Evans, 'An instruction systolic array architecture for neural networks', in: *International Journal of Computer Mathematics*, **61**(1-2), pp 63-89, 1996

[Kane95] A.J. Kane, *A Neural Network Analyser I.C. for the Evaluation of Boolean Network Cyclic Lengths*, Master's Thesis, Dept. Electrical Engineering, Brunel University, England, 1995

[Kawa83] S. Kawamura et al., '3-Dimensional SOI/CMOS IC's fabricated by beam recrystalisation', in: *International Electronic Devices Meeting (Technical Digest)*, pp 364-367, December 1983

[Keye79] R.W. Keyes, 'The evolution of digital electronics towards VLSI', in: *IEEE Transactions on Electron Devices*, **26**(4), pp 271-278, 1979

[Kirk83] S. Kirkpatrick, C.D. Gellat Jr. and M.P. Vecchi, 'Optimization by simulated annealing', in: *Science*, **220**, pp 671-680, 1983

[Koho88] T. Kohonen, *Self-Organization and Associative Memory*, 2nd Edition, USA : Springer Verlag, 1988

[KuHw89] S.Y. Kung and J.N. Hwang, 'A unifying algorithm/architecture for artificial neural networks', in: *Proceedings of the International Conference on Application Specific Signal Processors*, pp 2505-2508, Edinburgh, 1989

[KuLe80] H.T. Kung and P.L. Lehman, 'Systolic VLSI arrays for relational database operations', in: *Proceedings of the ACM/SIGMOD International Conference on Management of Data*, (P.P. Chen and R.C. Sprowls eds.), pp 105-116, 1980

[KuLi78] H.T. Kung and C.E. Lieserson, 'Algorithms for VLSI processor arrays', in: *Sparse Matrix Proceedings*, Siam Press, pp 256-282, 1978

[KuLi80] H.T. Kung and C.E. Lieserson, 'Systolic arrays for VLSI', in: [MeCo80], section 8.3

[Kund86] M. Kunde, H.-W. Lang, M. Schimmler, H. Schmeck and H. Schröder, 'The instruction systolic array and its relation to other models of parallel computers', in: *Parallel Computing 85*, pp 491-497, Amsterdam, 1986

[KungHT80] H.T. Kung, 'The structure of parallel algorithms', in: *Advances in Computers*, **19**, pp 66-112, 1980

[KungHT82] H.T. Kung, 'Why systolic architectures ?', in: *IEEE Computer*, **15**(1), pp 37-45, 1982

[KungSY82] S.Y. Kung, K.S. Arun, R.J. Galezer and D.V.B. Rao, 'Wavefront array processor : language, architecture and application', in: *IEEE Transactions on Computers*, **C-31**(11), pp 1054-1066, November 1982

[KungSY84] S.Y. Kung, 'On supercomputing with systolic/wavefront processors', in: *Proceedings of the IEEE*, **72**(7), July 1984

[KungSY88] S.Y. Kung, *VLSI Array Processors*, USA : Prentice Hall, 1988

[Lawr75] D.H. Lawrie, 'Access and alignment of data in an array processor', in: *IEEE Transactions on Computers*, **C-24**, pp 496-503, 1975

[Lehm93] C. Lehmann, M. Viredaz and D. Blayo, 'A generic systolic array building block for neural networks with on-chip learning', in: *IEEE Transactions on Neural Networks*, **4**(3), pp 400-407, 1993

[LePr92] D. Lewin and D. Protheroe, *Design of Logic Systems*, 2nd Edition, UK : Chapman and Hall, 1992

[Lipp87] R.P. Lippman, 'An introduction to computing with neural nets', in: *IEEE ASSP Magazine*, **4**, pp 4-22, 1987

[Mano82] M.M. Mano, *Computer Systems Architecture*, USA : Prentice Hall, 1982

[MaSi64] H.M Mansevit and W.I. Simpson, 'Single crystal silicon on a sapphire substrate', in: *Journal of Applied Physics*, **35**, pp 1349-1351, 1964

[McIr95] S. McCloone and G. Irwin, 'Parallel off-line training of multilayer perceptrons', in: *Proceedings of the 3rd IFAC/IFIP Workshop on Algorithms and Architectures for Real-Time Control*, Ostend : Belgium, 1995

[McKe90] J.J. McKeown, D. Meegan and D. Sprevak, *An Introduction to Unconstrained Optimization*, UK : Adam Hilger, 1990

[McPi43] W. McCulloch ad W. Pitts, 'A logical calculus of the ideas imminent in the nervous system', in: *Bulletin of Mathematical Biophysics*, **5**, pp115-137, 1943

[MeCo80] C.A. Mead and L.A. Conway, *Introduction to VLSI Systems*, USA : Addison Wesley, 1980

[Megs92] G.M. Megson, *An Introduction to Systolic Algorithm Design*, UK : Oxford Science Publications, 1992

[Micz87] A. Miczo, *Digital Logic Testing and Simulation*, USA : Wiley, 1987

[MiPa69] M.L. Minsky and S.A. Papert, *Perceptrons : An Introduction to Computational Geometry*, USA : MIT Press, 1969

[Moor75] G.E. Moore, 'Progress in digital integrated electronics', in: *International Electronic Devices Meeting*, December 1975, pp 11-13, 1975

[Myer89] D.J. Myers and R.A. Hutchinson, 'Efficient implementation of piece-wise linear activation function for digital VLSI neural networks', in: *Electronic Letters*, **25**(24), pp 1662-1663, 1989

[Myer91] D.J. Myers, J.M. Vincent and D.A. Orrey, 'HANNIBAL : A VLSI building block for neural networks with on-chip backpropagation learning', in: *Proceedings of the Second International Conference on Microelectronics for Neural Networks*, pp 171-181, Munich, 1991

[Nayl94] D. Naylor, S. Jones and D. Myers, 'Backpropagation in linear arrays - a performance analysis and optimisation', in: *IEEE Transactions on Parallel and Distributed Computing*, 1994

[NaJo97] D. Naylor and S. Jones, *VHDL : A Logic Synthesis Approach*, UK : Chapman & Hall, 1997

[Nigr91] M.E. Nigri, P. Treleaven and M. Vellasco, 'Silicon compilation of neural networks', in: *Proceedings CompEuro '91*, USA : IEEE Press, pp 541-546, 1991

[Ohzo80] T. Ohzone et al., 'Silicon-gate n-well CMOS process by full ion implantation technology', in: *IEEE Transactions on Electronic Devices*, **ED-27**, pp 1789-1795, September 1980

[OUP90] ______, *The Concise Oxford Dictionary of Current English*, 8th Edition, UK : Oxford University Press, 1990

[Pao89] Y.H. Pao, *Adaptive Pattern Recognition*, USA : Addisson Wesley, 1989

[Park82] D.B. Parker, *Learning Logic*, Invention Report S81-64, File 1, Office of Technology Licensing, Stanford University, Stanford, USA

[Parr80] L.C. Parillo et al., 'Twin-tub CMOS - a technology for VLSI circuits', in: *IEEE International Electronic Devices Meeting*, pp 752-755, 1980

[PeHa89] C. Peterson and E. Hartman, 'Explorations of the mean field theory learning algorithm', in: *Neural Networks*, **2**, pp 475-494, 1989

[Pesu90] E.E. Pesulima, A.S. Pandya and R. Shankar, 'Digital implementaion issues of stochastic neural networks', in: Proceedings of the IJCNN (Washington DC), USA : IEEE Press, **2**, pp 187-190, 1990

[Pete87] C. Peterson, 'A mean field theory learning algorithm for neural networks', in: *Complex Systems*, **1**, pp 995-1019, 1987

[Petk89] N. Petkov, *Systolic Parallel Processing*, Netherlands : Elsevier Science, 1989

[PrVu80] F.P. Preparata and J.E. Vuillemin, 'Area-time optimal VLSI networks for multiplying matrices', in: *Information Processing Letters*, **11**, pp 77-80, 1980

[PuRo71] G.R. Putzolu and J.P. Roth, 'A heuristic algorithm for the testing of asynchronous circuits, in: *IEEE Transactions on Computers*, **C-20**, pp 1286–1283, 1971

[QuRo89] P. Quinton and Y. Robert, *Algorithmes et Architectures Systoliques*, France : Masson, 1989, Translated by I. Craig, *Systolic Algorithms and Architectures*, UK : Masson/Prentice Hall, 1991

[Ramó11] S. Ramón y Cája, *Histologie du système nerveux de l'homme et es vertébrés*, Maloine : France, 1911

[RiKn91] E. Rich and K. Knight, *Artificial Intelligence*, 2nd Edition, USA : McGraw Hill, 1991

[Rodr93] A. Rodríguez-Vázquez et al, 'Current-mode techniques for the implementation of continuous- and discrete-time cellular neural networks', in: *IEEE Transactions on Circuits and Systems II*, **40**, pp 132-146, March 1993

[Rose62] F. Rosenblatt, *Principles of Neurodynamics*, USA : Spartan Books, 1962

[Roth66] J.P. Roth, 'Diagnosis of automata failures - a calculus and method', in: *IBM Journal of Research and Development*, **10**, pp 278-291, 1966

[Roth92] C.H. Roth, *Fundamentals of Logic Design*, 4th Edition, USA : West Publishing Company, 1992

[Rumm86] D.E. Rummelhart, G.E. Hinton and R.J. Williams, 'Learning internal representations by error propagation', in: *Parallel Distributed Processing*, Vol 1, pp 318-362, USA : MIT Press, 1986

[Schw87] M. Schwarz, *Telecommunications Networks : Protocols, Modelling and Analysis*, USA : Addison Wesley, 1987

[Seit85] C.L. Seitz, 'The Cosmic Cube', in: *Communications of the ACM*, **28**(1), pp 22-33, 1985

[SeLi95] T. Serrano-Gotarredona and B. Linares-Barranco, 'A VLSI-friendly "fast learning" ART1 algorithm', in: *Proceedings of the 1995 World Congress on Neural Networks*, Washington D.C., **1**, pp 27-30, 1995

[SeLi96] T. Serrano-Gotarredona and B. Linares-Barranco, 'A real-time clustering microchip neural engine', in: *IEEE Transactions on VLSI Systems*, **4**(2), pp 195-209, 1996

[SeRo87] T.J. Sejnowski and C.R. Rosenberg, 'Parallel networks that learn to pronounce English text', in: *Complex Systems*, **1**, pp 145-168, 1987

[ShKo90] G.M. Shepherd and C. Koch, 'Introduction to synaptic circuits', in: *The Synaptic Organization of the Brain*, (G.M. Shepherd ed.), pp 3-31, USA : Oxford University Press, 1990

[Shut88] M.J. Shute, *Fifth Generation Wafer Architecture*, UK : Prentice Hall, 1988

[Sore85] D.C. Sorenson, 'Analysis of pairwise pivoting in gaussian elimination', in: *Report MCS-TM-26*, Argonne National Laboratory, 1985

[StHu87] W.S. Stornetta and B.A. Huberman, 'An improved three-layer backpropagation algorithm', in: *Proceedings of the IEEE First International Conference on Neural Networks*, (M. Caudill and C. Butler eds.), USA : SOS Printing, 1987

[Ston71] H.S. Stone, 'Parallel processing with the perfect shuffle', in: *IEEE Transactions on Computers*, **C-20**, pp 153-161, 1971

[Ston90] H.S. Stone, *High Performance Computer Architecture*, 2nd Edition, USA : Addison Wesley, 1990

[Swok88] E.W. Swokowski, *Calculus with Analytical Geometry*, 4th Edition, USA : PWS-Kent, 1988

[Taba95] D. Tabak, *Advanced Microprocessors*, 2nd Edition, USA : McGraw Hill, 1995

[TI-95] *TMS320C80 (MVP) Technical Brief*, Texas Instruments Inc, USA, 1995

[TI-98] *TMS320C62xx Technical Brief (Rev 3.0)*, Texas Instruments Inc, USA, 1998

[Trel82] P.C. Treleaven, D.R. Brownbridge and R.P. Hopkins, 'Data-driven and demand-driven computer architecture', in: *ACM Computing Surveys*, pp 93-144, March 1982

[Trel89] P. Treleaven, M. Pacheco and M. Vellasco, 'VLSI architectures for neural networks', in: *IEEE Micro*, pp 8-27, 1989

[Uffe87] J. Uffenbeck, *The 8086/8088 Family - Design, Programming and Interfacing*, USA : Prentice Hall, 1987

[Ullm84] J.D. Ullman, *Computational Aspects of VLSI*, USA : Computer Science Press, 1984

[Uya84] M. Uya, K. Kaneko and J. Yasui, 'A CMOS floating point multiplier', in: *IEEE Transactions on Electronic Devices*, **C-30**(5), pp 305-311, 1984

[Vand86] A.J. Vander, J.H. Sherman and D.S. Luciano, *Human Physiology : The Mechanism of Body Functions*, 4th Edition, USA : McGraw Hill, 1986

[Wass89] P.D. Wasserman, *Neural Computing : Theory and Practice*, USA : Van Nostrand Reinhold, 1989

[Wate82] D.G.P. Waters, 'The problems of testing large-scale integrated circuits', in: *British Telecoms Engineering*, **1**, pp 64-69, July 1982

[Watt89] A. Watt, *Fundamentals of 3D Computer Graphics*, UK : Addison Wesley, 1989

[WeEs88] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design : A Systems Perspective*, USA : Addison Wesley, 1988

[Werb74] P.J. Werbos, Beyond Regression - New Tools for Prediction and Analysis in the Behavioural Sciences, Masters thesis, Harvard University, USA, 1974

[Whit84] I.R. Whitworth, *16-Bit Microprocessors*, UK : Granada Publishing, 1984

[WiBr81] T.W. Williams and N.C. Brown, 'Defect level as a function of fault coverage', in: *IEEE Transactions on Computers*, **C-30**(12), pp 987-98, December 1981

[Widr59] B. Widrow, 'Adaptive sampled-data systems - a statistical theory of adaptation', in: *1959 IRE WESCON Convention Record*, Part 4, pp 88-91, 1959

[WiHo60] B. Widrow and M. Hoff, 'Adaptive switching circuits', in: *1960 IRE WESCON Convention Record*, pp 96-104, 1960

[Wind60] R.O. Windner, 'Single state logic', in: *Proceedings of the AIEE Fall General Meeting*, 1960

[Yama83] T. Yamaguchi et al., 'High-speed latchup-free 0.5µm channel CMOS using self-aligned $TiSi_2$ and deep-trench isolation technologies', in: *International Electronic Devices Meeting (Technical Digest)*, pp 522-525, December 1983