A Novel Architecture for a High Performance Low Complexity Neural Device

Richard Patrick Palmer

A thesis presented for a Doctor of Philosophy
Department of Computer Science
University College, London.

February 1992
This thesis presents a novel architecture on which to implement neural networks and simple digital signal processing models in a compact and low cost manner. The need for this architecture was identified from a particular application; this application requiring a portable real-time device with which to perform neural network and simple digital signal processing models on speech signals. The development of the architecture was driven by this, and other similar applications, and did not start with any pre-conceived ideas. This allowed complete freedom to be expressed in its development, enabling a more novel architecture to be developed.

In this thesis the development of this architecture is reported. This extends from the analysis of current neural models, the identification of the necessary features from these, and the creation of a unified architecture to provide these features. The design of a silicon chip based on this architecture is then presented.

The architecture that has been developed combines the programmability and high efficiency of MIMD arrays, with the low complexity in both communications and control of SIMD arrays. The device built using this architecture incorporates a single processing element that consists of only 5300 transistors. Projections indicate that a totally integrated processor array could be built; incorporating fourteen such processors with on-chip memory, A/D and D/A converters.

An analysis of the architecture, and other architectures that fit into the same classification class, is then presented. This analysis has been able to identify some common features with MIMD/SIMD machines, and to conclude with some projections for the future.
I would like to thank many people in helping me during my time working on this thesis. Mostly I would like to thank Peter Rounce for helping me during the development of this thesis, and for his tireless reading, and rereading, of various drafts. I would also like to thank Chris Clack for volunteering to read this thesis over his Christmas break, and to Mike Brent for allowing me to use the London VLSI Consortium PAD Frame utility.

Many thanks also to my parents, flat mates and Simone for putting up with me during the time spent working on my Ph.D.

Lastly thanks to Steve Wright and the afternoon posse, for keeping a smile on my face during the many afternoons spent working on this thesis.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title Page</td>
<td>1</td>
</tr>
<tr>
<td>Abstract</td>
<td>2</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>3</td>
</tr>
<tr>
<td>Contents</td>
<td>4</td>
</tr>
<tr>
<td>List of Figures</td>
<td>9</td>
</tr>
</tbody>
</table>

### Chapter 1: Introduction

1.1 The Need for Alternative Computational Methods  | 12   |
1.2 Putting Neural Networks to Work             | 13   |
1.3 Objectives of this Thesis                  | 14   |
1.4 Research Contributions                     | 15   |
1.5 Organisation of Thesis                     | 15   |

### Chapter 2: Neural Networks

2.1 Desirable Features of Neural Networks       | 18   |
2.2 Development of Neural Networks              | 19   |
2.2.1 Biological Neuron                        | 19   |
2.2.2 Artificial Neuron                        | 21   |
2.3 Analysis of Neural Networks                 | 21   |
2.3.1 Single Layer Network                     | 22   |
2.3.2 Multilayer Networks                      | 24   |
2.4 Learning in Neural Networks                 | 26   |
2.4.1 Unsupervised Learning                    | 26   |
2.4.2 Supervised Learning                      | 26   |
2.4.2.1 Single Layer Learning Rule (Delta Rule)| 27   |
2.4.2.2 Multilayer Learning Rule (Backpropagation)| 27   |
2.4.3 Other Learning Techniques                | 28   |
2.5 Summary of Neural Networks                  | 29   |

### Chapter 3: Application Areas

3.1 The Development of Speech Processing Systems| 30   |
3.2 Problems to be Addressed                   | 31   |
3.2.1 Speech Features                          | 31   |
3.2.2 Phonemes                                 | 32   |
3.3 Difficulties Encountered by Deaf People     | 33   |
3.4 First Generation of Devices - SiVo          | 33   |
3.4.1 The 'Peakpicker' Chip 36
3.4.2 Results Obtained by using SiVo 37
3.4.3 Limitations of SiVo 38
3.5 Neural Networks and Speech Processing 39
  3.5.1 Use of Neural Networks at UCL 41
  3.5.1.1 Analysis and Resynthesis of Speech 41
  3.5.1.2 Fundamental Period Extraction Network 43
  3.5.2 Real-Time Implementation for Fx Extraction 45
  3.5.3 Requirements for an Alternative Architecture 48
3.6 Conclusion 49

Chapter 4 Architectural Considerations

  4.1 The Diversity of Neural Implementations 50
    4.1.1 Neural Co-processors 51
      4.1.1.1 NETSIM - Texas Instruments 52
      4.1.1.2 DELTA Co-Processor - SAIC Corp. 52
    4.1.2 Neurocomputers and ASIC Neural Devices 53
      4.1.2.1 ETANN - Intel Corporation. 54
      4.1.2.2 DNNA - Neural Semiconductor Ltd. 56
      4.1.2.3 Intelligent Memory Chips - Oxford Computer 57
      4.1.2.4 Hitachi Neural Wafer 59
      4.1.2.5 Digital Snake - Univ. Ancona, Italy. 61
      4.1.2.5 A Flexible WSI Neural Network - Inst Nat Poly 62
    4.2 Assessment of Neural Implementations 64
      4.2.1 Analogue Implementations 65
      4.2.2 Hybrid and Pulse Neural Implementations 66
      4.2.3 Digital Implementations 67
        4.2.3.1 MIMD Arrays 67
        4.2.3.2 SIMD Arrays 69
    4.3 Analysis of SIMD Arrays 72
      4.3.1 Flexibility 73
      4.3.2 Performance 73
      4.3.3 Real-Time Constraints 75
    4.4 Summary of Available Neural Implementations 76

Chapter 5 Proposed Architecture

  5.1 Objectives for the Architecture 78
  5.2 Development of Architecture 79
5.2.1 Desirable Feature of MIMD Arrays
5.2.2 Desirable Feature of SIMD Arrays
5.3 How to Combine these Features
5.3.1 Combining MIMD and SIMD Control
5.3.1.1 Storage of MIMD and SIMD Control Strategies
5.3.1.2 Control Switching & Virtual Neuron Definition
5.3.1.3 Proposed Communications Scheme
5.3.1.4 Synchronisation of Processors
5.4 Proposed Architecture
5.4.1 Architecture of Processing Element
5.5 Programming Strategy
5.5.1. SIMD Program
5.5.2. MIMD Program
5.6 Application Notes and Examples
5.6.1 Programming Techniques
5.6.1.1 Definition of Virtual Neuron
5.6.1.2 Initialisation and Synchronisation
5.6.1.3 Implementation of Input Window
5.6.1.4 Communication of Partial Sums
5.6.2 Application Examples
5.6.2.1 Fully Connected Network
5.6.2.2 State Feedback Network
5.6.2.3 Finite Impulse Response Filter
5.6.2.4 Infinite Impulse Response Filter
5.6.3 Tx Extraction Implementation
5.6.3.1 Silicon Efficiency
5.6.3.2 Reduced Sample Period
5.6.3.3 Reduced Response Time
5.6.4 Summary of Implementation Examples

Chapter 6 Detailed Description of Design
6.1 Design of CMOS Implementation
6.2 Algorithms used for the Arithmetic Functions
6.2.1 Pointer Manipulations
6.2.2 Partial Sum Additions
6.2.3 Multiplication Instructions
6.3 Datapath Construction
6.3.1 Timing Strategy
6.3.2 Arithmetic Unit
6.3.3 Saturated Addition
6.3.4 Double Length Shift
6.3.5 Execution of a Single Multiply Iteration
6.3.6 Simulation of Datapath
6.3.7 Datapath Performance
6.4 Generation of Instruction Decoder
6.4.1 PLA Template
6.4.2 PLA Personality Matrix
6.5 Development of Control Switching
6.5.1 Transition from MIMD to SIMD
6.5.2 Transition from SIMD to MIMD
6.5.3 Implementation of Control Switching
6.6 Development of SIMD Control
6.7 Implementation of Bus Logic Functions
6.7.1 Modulo Address Generation
6.7.2 Sigma Table Bus
6.8 Full Simulation of Processor Array
6.9 Design Conclusion
6.10 Future Large-Scale Integration
6.11 Design Environment

Chapter 7 Analysis of Architecture
7.1 Classification of Parallel Machines
7.2 Other MIMD/SIMD Machines
7.2.1 Non-Von 4 - Columbia University
7.2.2 DADO - Columbia University
7.2.3 XIMD - Carnegie Mellon University
7.3 MIMD/SIMD Machines in the Future
7.4 Postscript

Chapter 8 Conclusion
8.1 Summary of Thesis
8.2 Research Contribution
8.3 Literature Published

Plates
P.1 Confocal Light Microscope
P.2  Operation of a Confocal Light Microscope 181

Appendix

1  LOCAL Finite State Machine 190
2  GLOBAL Finite State Machine 195
3  Instruction Decoder 200
4  Netlist Generation 214
   i)  NET 215
   ii) LABELS 216
   iii) NETLIST 217
5  ESP Program Listing 220
6  PAD Frame Request File 223
7  Performance Analysis 227
8  Literature Published 231
   i)  R. Matthews 232
   ii) P Write 233
   iii) R. Palmer 234

References 244
# List of Figures

## Chapter 2 Neural Networks

<table>
<thead>
<tr>
<th>Diagram 2.1</th>
<th>Biological and Artificial Neurons</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagram 2.2</td>
<td>AND, OR and XOR Pattern Space</td>
<td>22</td>
</tr>
<tr>
<td>Diagram 2.3</td>
<td>Single, Two and Three Layer Networks</td>
<td>24</td>
</tr>
<tr>
<td>Diagram 2.4</td>
<td>Separation Functions for Various Networks</td>
<td>25</td>
</tr>
</tbody>
</table>

## Chapter 3 Application Areas

<table>
<thead>
<tr>
<th>Diagram 3.1</th>
<th>Tx and Fx Extraction</th>
<th>34</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagram 3.2</td>
<td>Signal Flow Through SiVo</td>
<td>35</td>
</tr>
<tr>
<td>Diagram 3.3</td>
<td>Waveforms from the Peakpicker Chip</td>
<td>37</td>
</tr>
<tr>
<td>Diagram 3.4</td>
<td>Input Windows for Different Extraction Levels</td>
<td>40</td>
</tr>
<tr>
<td>Diagram 3.5</td>
<td>State Feedback Network</td>
<td>41</td>
</tr>
<tr>
<td>Diagram 3.6</td>
<td>'Y' Model</td>
<td>42</td>
</tr>
<tr>
<td>Diagram 3.7</td>
<td>Tx Extraction Network</td>
<td>44</td>
</tr>
<tr>
<td>Diagram 3.8</td>
<td>Sample Window Implementations</td>
<td>47</td>
</tr>
</tbody>
</table>

## Chapter 4 Architectural Considerations

<table>
<thead>
<tr>
<th>Diagram 4.1</th>
<th>The Diversity of Neural Implementations</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagram 4.2</td>
<td>ETANN Synapse</td>
<td>55</td>
</tr>
<tr>
<td>Diagram 4.3</td>
<td>DNNA Synapse and Neural Function</td>
<td>56</td>
</tr>
<tr>
<td>Diagram 4.4</td>
<td>Intelligent Memory Chips</td>
<td>59</td>
</tr>
<tr>
<td>Diagram 4.5</td>
<td>Hitachi WSI Structure</td>
<td>60</td>
</tr>
<tr>
<td>Diagram 4.6</td>
<td>Data Access in Hitachi Neurons</td>
<td>61</td>
</tr>
<tr>
<td>Diagram 4.7</td>
<td>Digital Snake Bus Structure</td>
<td>62</td>
</tr>
<tr>
<td>Diagram 4.8</td>
<td>Flexible WSI Structure</td>
<td>63</td>
</tr>
<tr>
<td>Diagram 4.9</td>
<td>Feedforward Network Structure</td>
<td>64</td>
</tr>
<tr>
<td>Diagram 4.10</td>
<td>Communication Strategies for SIMD Arrays</td>
<td>72</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>SIMD Array Performance</td>
<td>74</td>
</tr>
</tbody>
</table>

## Chapter 5 Proposed Architecture

<table>
<thead>
<tr>
<th>Table 5.1</th>
<th>MIMD Array Properties</th>
<th>78</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 5.2</td>
<td>SIMD Array Properties</td>
<td>79</td>
</tr>
<tr>
<td>Diagram 5.1</td>
<td>Methods Proposed for MIMD and SIMD Switching</td>
<td>85</td>
</tr>
<tr>
<td>Diagram 5.2</td>
<td>Outline of Array Architecture</td>
<td>90</td>
</tr>
</tbody>
</table>
Chapter 6

Detail Description of Design

Table 6.1 Multiplication Shift Modes 123
Diagram 6.1 Datapath Schematic 124
Diagram 6.2 Clocking Strategy 126
Table 6.2 Adder Input Selection 126
Diagram 6.3 Arithmetic Unit 127
Table 6.3 Saturated Addition Control 127
Diagram 6.4 Saturated Addition Logic 128
Diagram 6.5 Double Length Shift 129
Diagram 6.6 Multiplication Flow for $\Phi 1$ 131
Diagram 6.7 Multiplication Flow for $\Phi 2$ 132
Table 6.4 MIMD Two Stage Pipeline 138
Table 6.5 SIMD Three Stage Pipeline 139
Diagram 6.8 PERT Chart Showing SIMD and MIMD Control 139
Diagram 6.9  SIMD and MIMD Control Selection in a Processor 145
Diagram 6.10 Local Finite State Machine 146
Diagram 6.11 SIMD Control Block 148
Diagram 6.12 Global Finite State Machine 149
Table 6.6  Modulo Buffer Sizes 151
Diagram 6.13 Address Generation Control 151
Diagram 6.14 Sigma Function Threshold Levels 152
Diagram 6.15 Sigma Threshold Function Logic 154
Listing 6.1 MIMD Program Simulated on Processor Array 156
Diagram 6.16 Cascade of Processor Chips 157
Diagram 6.17 Graph of Die Size for Different Array Configurations 160
Diagram 6.18 Peak Performance of Device 162
Table 6.7  Summary of Performance Efficiency 162
Diagram 6.19 Projected Performance of Device 163
Diagram 6.20 Design Environment 165

Chapter 7  Analysis of Architecture
Table 7.1  Flynn Classification Scheme 169
Diagram 7.1 High-Level Taxonomy of Parallel Computer Architectures 170

Plates
Diagram P.1  Confocal Microscope Principle 181
Plate Ia  Datapath Layout (MAGIC Layout) 183
Plate Ib  Adder Layout
  i)  MAGIC Layout 184
  ii) Photomicrograph 184
Plate Ic  B Input Selection (XOR Function)
  i)  MAGIC Layout 185
  ii) Photomicrograph (Metal 2) 185
  iii) Photomicrograph (Poly & Diff) 186
  iv) Photomicrograph (Metal 1) 186
Plate IIa  Template for Decoder PLA (MAGIC Layout) 187
Plate IIb  PLA Layout (MAGIC LAYOUT) 188
Plate III  Photomicrograph of Single Processor 189
Chapter 1

Introduction

This chapter presents the main motivating factors for research into neural networks, and looks at how neural networks are being introduced into real applications. An overview of this thesis is then given, which includes a breakdown of the contents of each chapter.

1.1 The Need for Alternative Computational Methods

Ever since Von Neumann in the 1940s, computers have been developed using similar methods. The two key points that Von Neumann introduced were:

- Internal Program
- Serial Operation

Today's technology has allowed the computational power of these machines to be fully realised, with increasingly powerful processors being regularly introduced.

Compilers and high-level languages are also developing, further simplifying the task of program writing and allowing increasingly complex problems to be solved. Advances in expert systems have further simplified this task by allowing computers to be programmed by sets of rules. These rules can be generated by an expert in the relevant field without requiring any knowledge about programming.

Despite these advances the two main features of Von Neumann machines remain, and these features now pose limitations to some of the problems that computers are attempting to solve.

- Internal Program
  The requirement of a program involves the development of an algorithm, or set of rules, that very precisely define the operations required. This can present major problems in many tasks such as pattern recognition, signal processing and optimisation,
where algorithms are difficult to determine due to the imprecise nature or knowledge of the task.

• **Serial Operation**

The requirement of serial operation, although slightly relaxed in array and pipeline processors, has limited the performance of computers. Only small sections of hardware can be kept active at any one moment, so limiting the performance that could be obtained with today's large scale integration.

The development of neural networks has been inspired by these drawbacks in conventional computing, and by the way that biological systems process information. Neural networks try to capture the processing techniques used by these biological systems, and to exploit their benefits:

• **Learning**

By using learning rules inspired by biological systems, computers can be made that adapt to their environment, and learn specialised functions. This operation requires no programming, but instead a series of examples are presented along with examples of the required response. This eliminates the requirement for algorithms to be devised before a problem can be solved.

• **Parallel Operation**

The algorithms used in neural networks are naturally parallel, and rely upon localised communications. This allows arrays of processors to be built with an almost limitless expandability, thus making much better use of the processing power that can be obtained from large scale integrated circuits.

1.2 **Putting Neural Networks to Work**

Neural networks have been the subject for research since the 1940s, although most of the major breakthroughs have been more recent. In the last few years neural networks have begun to emerge from the research institutes and be put to use.
The first commercial systems to make use of neural networks were software packages to run on conventional hardware. One such system was the NESTOR Development System[1]. An early project tackled by the NESTOR Development System was bond-trading predictions; a network was taught on recent transactions and their outcomes, and it quickly learnt to replicate these transactions, and even improved upon them.

To enable neural networks to further increase their usefulness required specialised hardware to speed the computations of these networks. SAIC (Science Applications International Corporation) was one of the first sources of specialised hardware on which neural networks could be implemented. SAIC developed several products built around the DELTA processor[2]; these act as co-processors to a conventional host computer, or work in a stand-alone environment.

This increased processing power means that neural networks no longer have to run off-line with pre-prepared data, but instead they can work with fast incoming data in real-time systems. An example of this is the TNA explosive detection system[3]. This system is now being installed at major airports to monitor luggage for explosives. Neural networks are used to perform an analysis on the absorption spectrum of low energy neutrons, from which many different types of explosives can be detected.

1.3 Objectives for This Thesis

The need for still faster hardware, and for a larger network capacity, has continued. To address this problem many ASIC (Application Specific Integrated Circuit) designs have been proposed and built. A common aim in these designs has been to exploit the natural parallelism, and localised communications found in neural networks. This has enabled neurocomputers to be developed with an almost limitless expandability.

The digital architectures that have been developed fall into two classes: fine grained SIMD (Single Instruction Multiple Data) arrays, and coarse grained MIMD (Multiple Instruction Multiple Data) arrays.

SIMD machines tend to show low flexibility, and can also have problems in obtaining a high efficiency if the network topology does not fit
optimally to the array.

MIMD machines offer increased flexibility, and can achieve close to linear speed up with increased numbers of processors. However MIMD arrays achieve these characteristics by the use of complex communications and control mechanisms, adding considerably to the hardware requirements for such a machine.

This thesis highlights these problems, and proposes an alternative. The solution proposed brings together features from SIMD and MIMD arrays, and develops a unified architecture on which to exploit the benefits from each. This architecture is examined in detail, including both its programming, and the design of a CMOS device built implementing it.

1.4 Research Contributions

The main research contributions of this thesis have made to the area of neural architectures and parallel computing. To list the key features that have been presented would include:

• Critical study of the current state in neural implementations

• Development of an hypothesis out of which a new architecture could be developed

• Development of a software methodology combining SIMD and MIMD control

• Development of an architecture to support this software methodology

• Design of a silicon chip incorporating this architecture

• The identification of other similar work

1.5 Organisation of Thesis

The remainder of this thesis covers seven chapters, the organisation of
which is shown below.

**Chapter 2  Neural Networks**

This chapter looks at biological neurons and artificial neural networks. It covers the development of the multilayer perceptron model, with particular emphasis on the pattern recognition ability of different network topologies. Simple learning rules are then presented, with emphasis on the problems encountered by these algorithms.

**Chapter 3  Application Areas**

This chapter looks at an application area to which neural networks have been applied. One specific application is chosen, with the various techniques that have been used in its development being evaluated. This highlights the problems that were encountered, and shows how neural networks were used in solving them.

The problem that remains for this application was how to efficiently implement the neural model that was developed; this problem is further complicated since the proposed use for this network is in a portable real-time device. The problems involved in this implementation are highlighted, and are used as the design criteria for the architecture that has been developed.

**Chapter 4  Architectural Considerations**

In this chapter a survey of the current neural architectures is carried out. This survey takes a few specific examples, each chosen since it highlights a particular technique that is used to implement neural networks. From the analysis of these examples it was possible to identify features from various architectures that would be beneficial for the architecture being developed. An hypothesis is proposed at the end of this chapter: this states that if the beneficial properties of both MIMD and SIMD arrays could be combined in a unified architecture, this would result in suitable architecture for the application given in Chapter 3.
Chapter 5 Proposed Architecture

This chapter covers the development of an architecture to test the hypothesis presented in the previous chapter. This covers some of the design decisions that were made, and presents the overall architecture for an array and an individual processor. Programming and application examples are then given, these illustrate the flexibility of the architecture, and give some ideas as to how it can be programmed.

Chapter 6 Detailed Description of Design

This chapter outlines an implementation using the presented architecture. Major emphasis in this chapter is put upon how both MIMD and SIMD control strategies are used, and how switching between them is performed.

Future work is also covered in this chapter; this investigates the possible development of a fully integrated neural signal processor. This device would operate in a stand-alone environment, with memory, A/D and D/A converters being held on-chip.

Chapter 7 Assessment of Architecture

After presenting an implementation using this architecture, a more general look at the architecture is made. This chapter first classifies this architecture in a number of ways, and then look at other architectures that are similar. This chapter concludes with the possible direction in which this class of architecture might lead.

Chapter 8 Conclusion

This chapter summarises the key points presented in this thesis, and highlights the main research contribution that have been made. A list of publications is given, including those which the author is hoping to complete before the submission of this thesis.
Chapter 2

Neural Networks

Neural networks are biologically inspired forms of computation, that represent a departure from conventional forms of computing. Neural processing involves the use of very simple computational units that are highly interconnected and work in parallel.

The human brain is composed of $10^{11}$ neurons connected by $10^{15}$ interconnections. The brain is made up of 99% interconnections, with only 1% computational units, and works in an analogue manner. The brain is believed to be purely asynchronous with no central control unit. There is instead an homologous distribution of information, thought to be almost entirely held in the interconnections.

2.1 Desirable Features of Neural Networks

Neural networks try to capture a number of features from the brain in order to provide artificial processing systems that exhibit marked differences from conventional computer systems. The main properties that are sought for these artificial processing systems are:

• **Learning**
  Learning is probably the most important feature that the brain exhibits. It allows an individual to experiment with the environment, and using feedback, to modify its behaviour. This process is very complex in the human brain, with both short and long term memories playing important roles. Simplifications of these learning methods can allow learning to be incorporated into neural networks.

• **Generalisation**
  Generalisation allows a system to respond to uncertain data in a predictable manner. It involves using past knowledge to formulate the best response to a new situation or uncertain data. It is this feature that enables humans to carry out many tasks on
uncertain data that computers find extremely difficult. Simple examples of such a task would include speech and visual recognition.

• Redundancy
Redundancy enables a system to continue functioning, even when parts are damaged. This feature can also be called graceful degradation, as it ensures a slow reduction in the performance as more parts become damaged. This property would be beneficial for computer systems, where at present a single defect can be disastrous.

2.2 Development of Neural Networks

An artificial neuron is a mathematical entity that was developed by McCulloch and Pitts 1943[1], Hebb 1949[2] and Rosenblatt 1959[3], to model the functioning of a biological neuron. Diagram 2.1 shows the structure of a biological neuron and its artificial counterpart.

![Diagram 2.1 Biological and Artificial Neurons](image)

2.2.1 Biological Neuron

A biological neuron is composed of two main sections, the cell body and axon. The cell body receives and processes inputs from other neurons, while the axon transmits the output from the cell body to other neurons.
At the end of each axon lie the dendrites; these fan-out allowing each neuron to transmit its output to many other neurons. The mass connectivity obtained from these dendrites is seen to be crucial to the functioning of the brain.

Each neuron has many thousand inputs, coming from the dendrites of other neurons, or from sensory organs. These inputs are called synapses, and consist of a 20-30nm gap between the incoming nerve terminal and the input receptor on the cell body of the neuron. It is through the synapses that the incoming information is processed.

Nerve signals are transmitted as variable frequency pulses through the axon. These are integrated by the nerve ending, resulting in a variable quantity of 'Neurotransmitter' being released across the synapse. This neurotransmitter is picked up by the input receptor on the cell body, and results in a change in the membrane potential at that point on the cell membrane. This change can occur in two ways, either as an increase in membrane potential, excitatory, or as a decrease in membrane potential, inhibitory. The combined effect of the many thousands of input receptors results in overall changes to the cell membrane potential.

At equilibrium a potential exists across this cell membrane, which is the result of charged ions flowing through the membrane. Due to the membrane having varying permeability to the different ions this potential is about -60mV at equilibrium. This is called the resting potential.

The effect of the different neurotransmitters results in the membrane potential varying across the cell body. The amount of change in this membrane potential is dependent upon the strengths of the excitatory and inhibitory incoming impulses, and their locality.

If at any point on the cell body the membrane potential changes by as much as +20mV, a change occurs in the structure of the membrane. This action is called depolarisation, and effects the membrane by suddenly allowing certain ions to pass more freely. This sudden movement of ions creates an action potential (+40mV), which has the effect of
depolarising the surrounding area. The outcome of this is an impulse that spreads across the cell membrane, and down the axon. This impulse is then transmitted along the axon to the dendrites, resulting in this signal being transferred to further neurons. After each impulse has passed the resting potential is quickly reinstated, and this whole action can then be repeated.

Although the impulses transmitted down each axon are of fixed amplitude, and vary only in frequency, the operation of the biological neuron is fundamentally analogue. The cell bodies main processing task being the temporal and spatial summation of the input pulses across the cell body.

2.2.2 Artificial Neuron

The important features of the above description have been used to formulate the model most widely used for the artificial neuron. Structurally the artificial neuron can be made to look very similar, with several inputs, a neural function and an output that feeds to many other neurons. The mathematical function of an artificial neuron is shown in Diagram 2.1.b. and consists of applying a function to the weighted sum of each input.

The function most commonly used is a sigma function, since this solves many problems encountered by early research. It allows small signals to pass reasonably unaffected, while stopping large signals saturating the neuron's output. This is especially important in multilayer networks, where the outputs from one layer provide the inputs to the next.

This very simplistic model of a biological neuron does not attempt to imitate every feature. However it does mimic very well the important feature of mass connectivity, and distributed processing found in biological systems.

2.3 Analysis of Neural Networks

The following sections looks at the properties of various network types, and considers both their beneficial features and their limitations.
2.3.1 Single Layer Network

The simplest network that can be used is a single layer of neurons. This type of network was widely researched by the developers of the artificial neuron, and simple learning rules were developed. The Delta Rule was developed by Rosenblatt in 1959\(^3\) and is the most common.

After extensive research into this class of network it was shown by Minsky and Papert in 1960\(^4\), that these networks lacked any real power in pattern recognition. The problem used to show this was the XOR, or parity problem. This problem is discussed below - in this example a single neuron is used for the analysis. Minsky and Papert were able to extend this simple analysis onto any single layer network, and hence showed the fundamental limitations of this type of network.

Diagram 2.2 AND, OR and XOR Pattern Space

The XOR Problem

By taking a single neuron with two inputs (A & B) there are 4 possible binary patterns that can be represented. These can be plotted in a two dimensional pattern space, with the four input patterns being represented by the corners of a square. Several functions can be performed on these input patterns: these are the Boolean logic functions. Diagram 2.2 shows three such Boolean functions: AND, OR and XOR.
Minsky and Papert used a series of simple equations to show the possible separations that a single layer network could perform. The first two Boolean functions can be easily expressed by using the following equations, where:

\[
Wa = \text{Weight for Input A} \\
Wb = \text{Weight for Input B}
\]

Neural Function is a Threshold at 0.5

**AND**

\[
\begin{align*}
0Wa + 0Wb &< .5 \\
0Wa + 1Wb &< .5 \\
1Wa + 0Wb &< .5 \\
1Wa + 1Wb &> .5
\end{align*}
\]

Solution: \( .25 < Wa < .5 \) \\
\( .25 < Wb < .5 \)

**OR**

\[
\begin{align*}
0Wa + 0Wb &< .5 \\
0Wa + 1Wb &> .5 \\
1Wa + 0Wb &> .5 \\
1Wa + 1Wb &> .5
\end{align*}
\]

Solution: \( Wa > .5 \) \\
\( Wb > .5 \)

However the third function, XOR, cannot be expressed so easily. The four equations below describe the line used to determine the XOR pattern space. These four equations are clearly inconsistent.

**XOR**

\[
\begin{align*}
0Wa + 0Wb &< .5 \\
0Wa + 1Wb &> .5 \\
1Wa + 0Wb &> .5 \\
1Wa + 1Wb &< .5
\end{align*}
\]

This example shows the limitations of a single neuron. The only pattern recognition functions that can be performed by a single neuron are those that can be separated by a single line. This class of functions is termed as having the property of linear separability.
Linear separability can be extended into many dimensions, with hyperplanes being used instead of linear lines to determine the separation. Minsky and Papert showed that although many neurons can be used in a single layer network, only linearly separable functions can be expressed.

2.3.2 Multilayer Networks

Minsky and Papert concluded that the only useful class of neural networks were those that incorporated two or more layers, this class is known as multilayer networks.

However the problem that remained was in teaching these multilayer networks. Unless the weights can be taught, the complex patterns that these networks can represent can never be realised. It was this problem of teaching multilayer networks that caused the research in neural networks to die back. It was not until 1986 when Rumelhart and McClelland[5] developed a learning rule for multilayer networks, that research expanded and developed rapidly.

Diagram 2.3 shows a single layer, two layer and a three layer network. The connectivity between the layers in multilayer networks is normally 100%, that is each neuron is connected to every neuron in the preceding layer (fan in), and is also connected to every neuron in the following layer (fan out). The layers are conventionally labelled as:

Diagram 2.3 Single, Two and Three Layer Networks
• **Input Layer**
This layer does no processing, and is only used to represent the number of inputs to the network.

• **Hidden Layers**
The layers inside the network are labelled hidden layers, due to their outputs being hidden. If there is more than one hidden layer then they are numbered, starting with the layer nearest the input layer.

• **Output Layer**
This layer provides processing, and represents the output from a network. In a single layer network the output layer represents the only processing layer in the network.

The power that multilayer networks provide depends upon the number of layers in the network. The addition of each layer can be likened to ability of drawing more separation lines in the pattern space. Diagram 2.4 shows the possible separations with single layer networks, two layer networks and three layer networks, each with reference to a two-input network.

![Diagram 2.4 Separation Functions for Various Networks](image-url)
2.4 Learning in Neural Networks

There are two basic classes of learning algorithm used in neural networks: unsupervised and supervised learning.

2.4.1 Unsupervised Learning

In unsupervised learning the network is encouraged to experiment, and to discover for itself the required response. This form of learning is particularly suitable for cases where no model answers are available, since the network will determine its own set of rules to suit the application.

To teach the network it has to be presented with an input set of conditions. At the start of learning the network produces a random response, this is judged according to the suitability of this response. The score produced helps the network to determine whether the response was good or bad. Learning is then carried out by either adjusting the connections to increase this response on a good score, or to decrease the response on a bad score. By repeating this a series of times acceptable performance may be achieved, and the network can be put to work.

While the network is being used the connections can either be frozen to stop future learning, or learning can be allowed to continue. This continuation of learning can be beneficial as it allows the network to adjust to changes in the environment.

2.4.2 Supervised Learning

In this form of learning a network is provided with an input set of conditions, and is shown the required response to this input set. By examining the difference between the network's output and the required response, a set of error values can be calculated. The network then modifies its connections to minimise these error values. By repeating this process through many input and output pairs a network will be taught how to respond to a variety of conditions.
The following algorithms have been developed for supervised learning in single and multilayer networks.

2.4.2.1 Single Layer Learning Rule (Delta Rule)

The teaching of a single layer network is fairly simple, since each neuron's error can be directly calculated. Learning is done by repeating the following two stage procedure until an acceptable performance is obtained.

- **Forward Stage**
  The forward stage involves presenting an input set to the network, and allowing the network to iterate. It is this phase of the network that is used in recognition mode when learning is complete.

- **Reverse Stage**
  During the reverse stage the weights are altered so that the network output will better match the required output set. A $\delta$ value is calculated for each neuron by taking the product of the error and a derivative of the sigma function. Each weight for that neuron is then updated by adding the product of the neuron's $\delta$ term and each input value to the relevant weight.

2.4.2.2 Multilayer Learning Rule (Backpropagation)

This learning rule is an extension of the Delta Rule and is used in multilayer networks. As stated above the main problem encountered in teaching multilayer networks is in generating error terms for the hidden units. This is obtained by the following procedure:

- **Forward Stage**
  The forward stage involves presenting an input set to the network, and allowing the network to iterate. It is this phase of the network that is used in recognition mode when learning is complete.
•Reverse Stage

This reverse stage is executed one layer at a time, taking the output layer first. The same operation is carried as in the Delta Rule to calculate a δ term for each neuron in the output layer, and to adjust the weights in this layer.

To continue this operation for the hidden layers requires the generation of the error terms for each hidden layer neuron. This is achieved by using the δ terms calculated in the output layer, and backpropagating these to the preceding hidden layer. In this operation the weights in the output layer work in reverse. Summation is then performed on all backpropagated errors in each hidden layer neuron, providing an error term for that neuron. Each weight is then modified by taking this error term and using the Delta Rule in the normal manner.

Many hidden layers can be trained using this algorithm. By repeating this operation back through the layers, error terms can be derived for every neuron.

2.4.3 Other Learning Techniques

There are other features that have been introduced to improve the speed and efficiency of learning. These are as summarised below:

•Training Constant

A training constant can be used to modify the speed of learning. This is a value (typically 0.1 to 1.0) that is multiplied with the δ term before modifying the weights, hence varying the rate at which the network learns. This value can be continuously changed during learning to optimise the rate of learning.

•Neuron Bias

To speed the convergence of a network it can be beneficial to add a bias to each neuron. This bias shifts the sigma function to increase the gain where it is most needed. To calculate this offset an additional input is made to each neuron: this is fixed to the highest positive value (+1). Learning is then carried out
on this input in the normal manner. After learning this weights value is used as the neuron's bias, and is added to the partial sum for each iteration.

• Momentum
This is another method used to speed up the learning in multilayered networks. By adding a momentum value to the offset of each weight it is possible to speed the convergence for the network. This momentum value is determined by taking the last weight offset and multiplying it by some factor.

2.5 Summary of Neural Networks

This chapter has outlined some of the problems and techniques encountered in single layer and multilayer networks. The capabilities of these networks are extensive, with theoretical studies showing very high performance in pattern recognition. However, as previously mentioned, the real power of networks can only be realised by the teaching of these networks to represent complex pattern spaces. It is this task of teaching networks that present one of the major challenges in neural network research.
Chapter 3

Application Areas

The application areas for neural networks are extensive[1]. A list of just a few successful examples would include optimisation, forecasting, control, and signal processing. To obtain a greater knowledge of how neural networks are being used, and the work involved in developing working neural systems, this chapter takes one specific application. The application chosen is taken from work that has been carried out by the Phonetics Department at University College, London.

This chapter shows the alternative methods that have been used in developing a speech processor that can be used to simplify a speech signal, and then to re-present it in this simplified manner to a deaf person. The discussion covers the traditional methods that have been used in solving this problem, and presents a neural network solution that has been developed. The neural network developed for this application provides a more robust and elegant solution than the alternative techniques that had previously been used.

This chapter finishes with a discussion on a real-time implementation that had been built to execute this neural network. This implementation highlights the problems that exist in developing real-time neural systems, with particular reference to the small size, and portability requirements of this application. This chapter concludes with some requirements that I have isolated for an architecture that would provide an alternative, and possibly better implementation for this class of problem.

3.1 The Development of Speech Processing Systems

A group of about 10 members from the Phonetics Department at University College London, Guys Hospital and Cambridge University have joined together to research in the same area. They have formed what they call the EPI (External Pattern Input) Group. The objective of this group is to provide aids to help deaf people understand speech. Their view is
that it is possible to represent speech as several separable patterns. Their aim is to extract these basic patterns, and to reconstitute them in a form suitable for those people who find difficulty in understanding normal speech. This project is aimed at profoundly deaf people, and even the totally deaf. Work has been done to provide the totally deaf with electrical implants, which give an electrical stimulus to the auditory nerve, resulting in some sensation of hearing.

Both the profoundly deaf, and the totally deaf with electrical implants, suffer from the same problems when attempting to understand speech. The cause of this is that their hearing can only occur in a small frequency band, usually at the lower end of the audio frequency range. The frequency analysis carried out by the ear also becomes less accurate, making it difficult to distinguish between similar frequencies. These hearing problems result in these people being able to hear noises, but not with a wide enough spectrum to understand speech fully.

The objective for the EPI group is to develop a system that can identify various speech patterns, and then re-present these to the deaf person, using only the limited frequency ranges available and in a much simplified form.

3.2 Problems to be Addressed

Before examining the project in detail we must first study the basic properties of speech. Since speech exhibits a highly complex arrangement of signals, a hierarchy is used in its analysis. This helps to classify the signals, and to provide a uniform representation.

This hierarchy consists of three levels:

- Speech Features
- Phonemes
- Words and Sentences

3.2.1 Speech Features

Speech can be said to be represented by several basic features, which
can be attributed to the differing methods used in their production. They do not have clear cut properties, and differ between different speakers. Pitch variation, relative power and time duration all modify these features slightly. The three main speech features are:

- **Frication**
  Frication is produced by passing air over the tongue and lips. This produces the high frequencies as used in /s/ and /z/.

- **Voicing**
  Voicing is produced by the vocal cords, which provides a full spectrum of excitation. This is then subjected to selective filtering as it passes up through the vocal chambers, producing a wide variation of sounds. For each sound the excitation is the same, it is the shape of the throat and the mouth which causes the variations to the sounds. The voicing feature is used to produce all the vowels.

- **Nasal**
  Nasal sound is produced by vocal cord excitation, and the forcing of this through the nasal passage and out through the nose. This can be done either by pressing the tongue to the roof of the mouth, or by closing the lips. The sounds with the nasal feature are /n/ and /m/. Variations to these sounds are caused by the shape of the nasal passage, and tend to be higher in pitch than vocal sounds, due to the smaller size of the nasal chambers.

3.2.2 Phonemes

Speech features are combined to produce the next level in the speech hierarchy. This hierarchical level is represented by phonemes.

When attempting to recognise phonemes several problems occur; the same phoneme can sound very different depending upon its context, and they exhibit wide speaker-to-speaker variations. The effect of the context on the sound of a phoneme is due to neighbouring phonemes 'blurring' together; this is called coarticulation. Coarticulation occurs between any two adjacent phonemes, and even across word boundaries. This causes
major problems when attempting to recognise speech: word boundaries become difficult to pinpoint, and the same word will sound different depending on the preceding and following words.

These problems combine to make speech recognition a highly complex task. Very subtle distinctions separate similar sounding words, with the whole meaning of the speech relying upon fine discriminations between these patterns.

3.3 Difficulties Encountered by Deaf People

To help understand speech, it is useful to distinguish the relative strengths of each of these features. A high level of separation is required as many sounds may differ in just one speech feature. For example the difference between /p/ and /b/ is made only by the presence or absence of voicing: voicing being present in /b/ and not in /p/.

The voicing feature had been concentrated on first by the EPI Group. This had been decided because voicing is almost invisible to a lipreader, as all the movements are carried with in the throat. The voicing feature is also very useful since it provides information about timing and stress, and it can be used as feedback to the deaf person while he or she is speaking.

3.4 First Generation of Devices - SiVo

From their analysis of these problems the EPI Group were able to develop their first generation of aids; Microstim and SiVo (Sinusoidal Voice Aid)\[2,3\]. These are very similar in design: Microstim was designed for totally deaf, while SiVo was for the partially deaf. SiVo is described below; what is said applies just as well to Microstim.

The basic aim of SiVo is to extract the peaks from the voiced fundamental frequency, and to construct a sine wave to represent this information. This sine wave is constructed so that its frequency maps that of the fundamental frequency peaks, giving the patient a representation of the speaker's pitch contour.
Diagram 3.1 TX and FX Extraction

Diagram 3.1 shows the abstraction of speech, first to provide the fundamental period (Tx), and then the fundamental frequency (Fx). Finally the sinusoidal output is provided, with its peaks corresponding with those of the original voice waveform, and the Tx instances.

SiVo operates by picking out the peaks in the audio waveform, which are used to represent the voiced fundamental period (Tx). This peak picking is done by a threshold function on the speech waveform. Diagram 3.2 shows the signal flow through the various stages of SiVo.

The speech signal is first preprocessed to filter out the higher frequencies in the waveform; this has the effect of removing the background noise and the unwanted voiceless speech sounds.

The peaks that are extracted are then digitally processed to estimate the voiced fundamental frequency (Fx). An optional frequency mapping can be applied to give either a 50 Hz or an 80 Hz frequency drop, this feature is selected by the 'MAPITCH' switch. This MAPITCH switch ensures that the output from the device falls within the audio frequency range of the listener.

The amplitude of the required sine wave is calculated by use of a lookup table, this ensures that a comfortable listening volume is achieved at all frequencies that the device can offer (31 Hz to 707 Hz). The amplitude lookup is necessary since different patients have different hearing abilities at varying frequencies. If uniform amplitude were
Diagram 3.2  Signal Flow Through SiVo

provided, considerable discomfort would be experienced in some frequency bands where the patient may have good hearing, and no hearing may be present in particularly bad frequency bands. The resultant waveform is generated in an analogue fashion, and presented to the patient via an earphone.

The difference between SiVo and Microstim devices is only in the resultant waveform. A square wave is produced in Microstim, which is applied, via an ear moulded electrode, directly to the inner ear.

The first implementation of SiVo uses a mixture of analogue and digital techniques. The preprocessing and the threshold function is carried out by analogue techniques, while the actual sinusoidal output is produced by a microprocessor and a D-A converter. The amplitude lookup table is held in an EPROM, which enables each device to be customised to the patient, by the use of a PC at the clinic.

This device is constructed from individual components, and is about the size of a packet of cigarettes. A large amount of this volume is taken
up with the batteries that are required to make the device portable.

This original analogue version suffers in the large numbers of discrete components that are required, and from the large power consumption of these analogue devices. To cut down on this number of components a digital version of the threshold function has been developed on a silicon chip. This is called the Peakpicker, since its main function is to indicate the position of each peak in the speech waveform.

3.4.1 The 'Peakpicker' Chip

The 'Peakpicker' Chip was designed with the help of the Electrical Engineering Department, at University College London[4]. It provides a digital implementation of the threshold function and the low pass filter, hence cutting down on the number of individual components that are required. An A-D converter is needed to work with this chip, but future versions may have full integrated into the Peakpicker Chip.

The Peakpicker algorithm is based on a comparator, which compares the present input with the previous output from the Peakpicker algorithm. This allows the Peakpicker to pick out the first peak. To enable subsequent peaks to be picked out, a decay is used on the Peakpicker output. The rate of this decay determines how closely the peaks can be detected, and is adjusted by external control on the chip. The output from the peakpicker algorithm is a saw tooth waveform, which is then passed through a high pass filter to accentuate the peaks. This whole algorithm is then repeated to ensure maximum performance.

Diagram 3.3 shows the various waveforms that are produced by this algorithm. The preprocessing is performed by applying a logarithmic amplifier, and then low pass filtering. To generate the Tx instances spikes are required - these are produced by use of a comparator and a monostable after the Peakpicker algorithm.

The Peakpicker was built around a single bit adder, with data being multiplexed through it. This single bit adder can implement all the required stages, including the peakpicker algorithm, filter and comparator.
The chip was designed using SOLO 1200 purchased from European Silicon Structures. The standard 2 micron CMOS cell library was used throughout this design, resulting in a fully static synchronous circuit. The circuit area is 20mm² including the pads, with the core of the design occupying 12mm². The total number of transistors in the design is 2500.

The chip is packaged in a 40 pin device, using a CMOS compatible PAD library. The full use of CMOS is made possible since most modern A/D and D/A converters are now CMOS compatible, ensuring that the power consumption was as low as possible. The resulting power consumption of the device is 10 micro Watts when running at 12.8 KHz.

3.4.2 Results Obtained by using SiVo

The results obtained from testing the first generation of SiVo aids can be separated into three groups:
• Stone Deaf
These people had such bad hearing that they could not hear the output from the device. The Microstim research project is aimed at helping these people so this was as expected.

• Profoundly Deaf
These people found a great improvement with the use of the SiVo. In conjunction with lipreading, patients could distinguish between voiced and voiceless consonants with about 70% accuracy, compared with the 49% accuracy obtained when tested with a conventional aid.

• Lightly Deaf
These people found that not enough information was provided, and preferred normal aids.

The results from this design were very encouraging for the EPI group. Out of the 14 patients provided with SiVo aid 7 patients found it a real benefit and started using the devices regularly.

It is envisaged that the performance of SiVo and Microstim will be very similar to the above when the Peakpicker chip is utilised. This is because the basic algorithm remains the same; only the computational style is changed.

3.4.3 Limitations of SiVo

The main limitation to SiVo is in dealing with excessive background noise. The device is not able to extract accurately the fundamental frequency from noisy signals; many spurious signals are given, and correct signals are often missed.

A further limitation is in the specificity of the device. It is designed for this one model only, and cannot be easily extended to handle other models to extract further speech features. These problems led the EPI group to look at alternative methods to extract speech features. Neural networks were then chosen, and studies were carried out into their
possible use.

3.5 Neural Networks and Speech Processing

Much recent research into speech recognition has used neural networks[5]. Neural networks have been used to recognise, and label, the various patterns which occur in speech. There have been several approaches, each using different types of networks, and achieving a variety of results.

The first attempts to use neural networks in speech recognition had been carried out using Boltzmann Machine implementations. Early work done by Bridle and Moore in 1984[6] introduced this model to the area of speech processing. They had been able to recognise simple static patterns by the use of simulated annealing. However, they soon realised that to recognise real speech, some form of time context would have to be incorporated into their models.

Later work done at Cambridge University in 1986[7], extended this Boltzmann Machine model with the introduction of state feedback. With this temporal context, they had been able to recognise some short sentences of continuous speech.

The use of the Boltzmann Machine in speech recognition has not developed much since. This is mainly due to the slow speed of the simulated annealing, and the uncertain response time for the recognition of patterns. Other neural models, such as the multilayer perceptron, have taken over providing much faster implementations[8], allowing real-time systems to be developed.

To include temporal context into multilayer perceptron networks, the use of time windows was introduced by the TRACE model in 1986[9]. Each window is made up from several time slices, built up from the information extracted from the previous extraction layer.

Diagram 3.4 shows a typical arrangement of four such extraction levels: frequency spectrum, speech features, phonemes and words. Each of these extraction levels is composed of a number of units - with each unit
representing a particular item in the relevant extraction level. For example in the speech feature extraction level an individual unit is required for each feature to be recognised.

By building up the number of slices composed of these units a dynamically changing signal can be represented, allowing both forward and backwards context to be utilised when recognising patterns.

Further work had also been carried out at Cambridge University during 1987, extending their state feedback model onto multilayer perceptron networks[10]. In this model part of a network’s output is feedback though a time delay, and added to the network’s input pattern. This feedback creates some internal storage in a network, allowing it to recognise dynamic patterns. Such a network can be made to exhibit Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) features, opening up new areas for neural signal processing. Diagram 3.5 shows the simplest form of a state feedback network.
3.5.1 Use of Neural Networks at University College London

The first work carried out using neural networks by the Phonetics Department at University College London, was during 1986. This work had been a comparison between the performances of a multilayer perceptron network and a Bayes Classifier in speech classification. The Bayes Classifier is a conventional method of classifying pattern vectors, and is based on statistical decision theory, relying upon normal distribution of pattern clusters for optimum classification. The tests performed showed a considerable advantage in the use of a multilayer perceptron algorithm over that of the Bayes Classifier. This result is due primarily to the non-normal distribution of the pattern clusters in speech signals.

Since this time considerable work has been carried out using the multilayer perceptron model in a variety of projects. Two of the main projects that have been worked on are discussed below.

3.5.1.1 Analysis and Resynthesis of speech

The objective of this work is to look into the possibility of forming an internal representation of speech. This internal representation would be used both as a target for speech pattern recognition, and as a starting point for speech resynthesis. It can also be used to compress speech signals, with analysis and resynthesis of the speech being carried out at the transmitter and receiver.
Previous attempts at performing speech analysis and resynthesis have had difficulties. These attempts have analyzed the frequencies and energies of signals instead of considering the inherent pattern property of speech: the latter can be used to form a better representation of speech. The use of phonetic knowledge in the construction of an internal representation can be viewed as a 'Y' model. Diagram 3.6 shows schematically the interaction of this phonetic knowledge in the creation of an internal representation (I.R.).

The ideal properties of such an internal representation are that it should be easily obtainable from speech analysis, and that it should contain sufficient information to enable natural sounding speech to be produced. Ideally some form of speaker independence should also be possible - this would then show that a true internal representation has been made.

A multilayer perceptron network had been developed for these experiments. This network has 45 units in the analysis hidden layer, 38 units in the resynthesis hidden layer, and 15 internal representation units between these two layers. These internal representation units are used to detect the degree of presence of a selection of speech features, such as:

- SIL: Absence of speech
- FRIC: Presence of frication
- VOC: Presence of voicing
- NAS: Presence of nasality
- VFRIC: Presence of voicing and frication
- S: Presence of /s/ quality frication

The allocation of internal representation units allows for some redundancy in the representation, enabling generalisation to occur on
new untaught speech, and to provide a more robust performance.

The results of this work concluded that, with a limited set of taught data (digits 0-9), a good internal representation had been produced. The speech resynthesised was intelligible, if a little simplified, and the network was also able to generalise onto new untaught data.

3.5.1.2 Fundamental Period Extraction Network

A Fundamental Period Extraction Network had been developed, this was taken up by the EPI Group to provide a basis for the next generation of hearing devices, aimed at superseding SiVo and Microstim. The design of this network was to enable an estimate of the voiced fundamental period (Tx) to be made, and for it to work under non-ideal situations, with good background noise rejection[15].

This has been carried out using the multilayer perceptron algorithm, with a network being made up of four layers: an input layer, two hidden layers and an output layer. Diagram 3.7 shows this topology, and the input window arrangement used.

This input window is used to form the temporal context required for accurate Tx extraction. It is 20.5ms wide, and is constructed from 41 0.5ms wide slices. Each slice is made up of six energy values, corresponding to six different frequency bands, resulting in a total of 246 input units.

The two hidden layers both consist of six units. This was decided upon by simulation: ten units had first been used, but it was found that six units showed a very similar performance. This method of deciding upon the number of units to use in a network shows up the rather ad hoc methods still employed in this area of neural networks.

Only one unit is required in the output layer - this provides the Tx information. This information is then further processed to generate the required sinusoidal output for the device.

The training for this network was done on speech, with the fundamental
period being recorded by use of a Laryngograph. A Laryngograph is a pick-up that is attached to the throat, which enables a recording to be made of the vocal cord activity. This recording of the vocal cord activity was presented to the output layer during learning.

The initial results obtained showed good recognition, and great robustness when external noise was introduced to the input. It was discovered that the best noise rejection occurs when noise is added to learning data, since this enables a better generalisation of a noisy signal to be constructed.

The network proved to be so robust that it was even able to improve on the Laryngograph recording of the initial speech. When tested, using the same data it was taught on, the network could pick out extra vocal cord closures that were originally missed on the Laryngograph recording.
3.5.2 Real-Time Implementation for Tx Extraction Network

Work has been carried out to provide a real-time implementation of this multilayer perceptron network[16]. The system developed consists of a TMS320C25 Digital Signal Processor (with on-chip RAM), EPROM, A/D converter and various support components. The device fits on a printed circuit board measuring 7cm by 6cm. The board executes at 32 MHz and has a power consumption of 400mW. This gives about 12 hours life with lithium batteries.

The key points in the choice of the processor are defined as below:

- Single cycle Multiply-Accumulate and Data-Move Instruction
- Integer processing (To cut down on power consumption)
- Must be CMOS for low power consumption
- As much on-chip memory as possible

These points need no further explanation, with the exception of the Single cycle Multiply-Accumulate and Data-Move instruction. This instruction is used to implement the input window that is required in the Tx Extraction Network.

Most digital signal processing chips use a variety of different addressing modes to simplify the implementation of signal processing models, and to ensure the high throughput of data through the processing unit. Since digital signal processors now incorporate multiply and accumulate instructions that can occur in a single clock cycle, they require this processing power to be matched with the speed of the operand fetches. This puts a very high demand on the address generation that is required for many signal processing models that are commonly used.

For the implementation of input windows, two methods are widely used:

Modulo Address Generation
Fetch - Operate - Data Move Instructions

Both these methods use specialised hardware in their implementation,
thus ensuring the maximum throughput of data through the processing units. These two methods can be seen on the two most common families of digital signal processors:

Motorola DSP56000

In the Motorola DSP56000 a separate hardware block is used for address generation, this is termed the address generation unit (AGU). This unit operates in parallel with the multiplier, and can be used to implement a variety of address modes. The address mode that would be most suitable for the implementation of the Tx Extraction Input Window would be the Modulo Address Generation.

- **Modulo Address Generation**
  This address mode allows circular buffers to be created inside a memory block. These buffers must fall on a binary base address, and be of a binary divisible length. This mode allows FIFO buffers, sample windows and delays to be easily programmed.

Diagram 3.8.a shows the implementation of an input window using this address mode. It involves the setting up of a modulo buffer within the memory and the use of an input pointer to keep track of the most recent data sample. The pointer increments each time a new data sample is entered - with wrap around occurring if it increments beyond the end of the modulo buffer. This new data sample is labelled TO in the diagram, with the previous data samples being labelled as T-1, T-2... As each new sample is entered the remaining samples will recede back in time, (T-1 will become T-2 etc.) with the least recent being over-written by the new data sample.

This method of creating an input window has a drawback in that the value that represents a particular time, (ie TO) will be continuously moving through memory as new samples are entered. This requires special pointers to always keep track of the position of the input window within the modulo buffer.
Diagram 3.8  Sample Window Implementations

Texas Instruments TMS320

The TEXAS Instruments Digital Signal Processor range TMS320[18] uses a fundamentally different method in its address generation. By incorporating RAM on-chip the processor can achieve much faster operations on this memory, allowing special instructions to be used with no time overheads. One such instruction is the MACD instruction.

• MACD (Multiply-Accumulate and Data-Move)
This instruction fetches the next operand, performs the multiplication and accumulation, and moves the operand value to the next memory location. This data move has the effect of implementing FIFO buffers, sample windows and delays within a fixed memory area.

Diagram 3.8.b shows the implementation of an input window. It involves creating a buffer the same size as the input window. Using the MACD instruction once in a program loop performing the network update, results in the whole input window being shifted one place back in time. (The value at T0 will be moved to T-1 etc.) This operation will result in a space being created at the beginning of the input window, where the next sample can be placed.
The benefit of using this method to implement the input window is that each sample will remain fixed at the same location. (ie T0 will always be at the same position, likewise T-1, T-2 ..) This removes the necessity of having to keep track of where in memory each sample is positioned.

3.5.3 Requirements for an Alternative Architecture

This thesis studies an alternative architecture that is hoped will provide a better solution for the implementation of the Tx Extraction Network and other similar tasks. This architecture is to build on the features seen in the TMS320, while being optimised for minimal size and power consumption.

To achieve this objective I have isolated the following criteria on which to judge any prospective architecture.

- **High Flexibility**
  The architecture must exhibit enough flexibility to enable a variety of network types to be implemented. However the trade-off between flexibility and hardware complexity also needs considering, to ensure that an overly flexible architecture does not incur an undue cost in complexity.

- **Soft-Programmability**
  Soft-programmability allows the full flexibility of an architecture to be exploited in a convenient manner. In the following chapter it will be seen that many architectures achieve a very low complexity by a trade-off against soft-programmability. These architectures will be examined to discover whether this trade-off is always valid.

- **High Efficiency**
  To ensure that the two main objectives of minimal size and power consumption is achieved requires the efficient use of all sections of a particular architecture. This criterion is used vigorously in the analysis of any prospective architecture.
These three criteria are used in the critical analysis of currently available neural architectures presented in the following chapter. Following this analysis a new architecture is developed - again taking these criteria as design goals, and as a means of measuring its success.

3.6 Conclusion

This chapter has introduced a typical application area to which neural networks have been applied, and has studied the implementation of a specific problem related to speech processing.

This study has shown the benefits in using neural networks for this task; neural networks not only provide a more robust performance, but offer a more general solution that can be mapped onto a multitude of different problems.

The remainder of this thesis is devoted to developing an architecture that can implement the style of networks presented in this chapter. The criteria laid down in the previous section are used during this development, and are discussed in more detail in the later chapters.
Chapter 4

Architectural Considerations

This chapter considers the requirements that were identified in Chapter 3, and attempts to match these to an existing neural architecture. This chapter concludes, that at present, there is no single architecture that meets all these requirements. However it is possible to formulate an hypothesis that could result in suitable architecture; this hypothesis is taken up by the remainder of the thesis, which presents a suitable architecture, and an analysis of its success.

4.1 The Diversity of Neural Implementations

The hardware on which to implement neural networks is developing as neural networks become larger, and require faster execution. Originally conventional computers were used to simulate neural networks, with the first specialised neural hardware being developed to act as neural co-processors to speed up the neural operations.

To achieve even greater speed, and higher network capacity, special purpose computers are now being developed. These machines are called neurocomputers. The technology required to build these machines provides an almost limitless expandability by using highly specialised neural ASIC devices.

Diagram 4.1 gives some indication as to the power, in both speed and network capacity, that a variety of implementations can achieve. This diagram also includes conventional computers and some low intelligence animals, so that comparisons can be made. The values given in this diagram are for single devices only, the performance of many of these implementations can be increased by utilising arrays of these devices.

The rest of this chapter looks at the different architectures that have been proposed for neural co-processors and neural ASICs. After presenting these architectures some analysis is carried out, with particular reference to the criteria laid down in Chapter 3. From this
Diagram 4.1 The Diversity of Neural Implementations

Analysis it is possible to see that there is no single architecture, from those currently available, that meet all these requirements.

4.1.1 Neural Co-processors

Neural co-processors are a fairly mature technology with several products now being commercially available. Co-processors can be obtained for several hosts - PCs, SUNs and other UNIX workstations. They tend to be board based, and consist of memory, input/output buffers and fast computational units. The technology used in these co-processors ranges from digital signal processors to bit-slice technology, and even custom built devices. The parallelism incorporated in most co-processor boards is low: the speed is obtained by using extremely fast multipliers and pipelined architectures.
4.1.1.1 NETSIM - Texas Instruments

NETSIM\[3\] is a board-based co-processor built around two custom designed chips. These are a ‘Solution Engine’ and a communications chip.

The Solution Engine has just three instructions, these being specially designed for neural operations. These are:

- Repeat-Multiply-Sum  (Update Synaptic Value)
- Read-Write
- Repeat-Multiply-Sum-Write  (Update Weight Values)

The speed obtainable from the Solution Engine is $4 \times 10^6$ interconnections per second or $1.3 \times 10^6$ learning interconnections per second.

NETSIM boards can be connected into arrays, of either two or three dimensions. The communications chip performs all the necessary data transfers, at a rate of 10 Mbit per second. The theoretical maximum number of NETSIM boards that can be connected in one array is 27000, with up to 256 neural nodes per board.

A major feature in this design is the development, and eventual expansion into arrays of NETSIM boards. For development purposes a replica NETSIM board is inserted into a PC host. High level languages can be used to develop and debug applications using this replica NETSIM board. Upon completion the software can be downloaded onto an array, with the host PC providing input and output as well as control over the array.

NETSIM provides a versatile implementation for neural networks. The speed achieved is adequate for many applications, but this may require complex and bulky arrays of NETSIM boards to be built.

4.1.1.2 DELTA Co-Processor - SAIC Corp.

The DELTA\[4\] co-processors provides a solution to a wide range of applications. It allows the full development and eventual production of systems built around DELTA processor boards, and the DELTA operation
system.

The development tools are standard across the range of products, and use the DELTA operating system and ANSim development tools. This facility enables the development of software to be carried out with a consistent interface. The range of processor boards available allows a variety of systems to be built: they can be used in conjunction with a host processor, or in a stand-alone environment. The stand-alone processors include A/D and D/A converters so real-time control systems can be built with minimal support hardware.

The DELTA processor architecture consists of a 4 stage pipelined Harvard Architecture providing maximum parallelism for the operations required:

- Operand Fetch (4 Address Modes)
- Multiply
- Accumulate
- Write Result

The multiplication and addition are performed by BIT's ECL floating point multiplier and ALU chips. The performance achieved is 10^7 interconnections per second with a capacity of 10^6 connections.

Input and output is performed by FIFO Buffer links capable of 40MB/sec, allowing up to 32 devices to be connected using a shared bus.

The DELTA processor and ANSim development tools provide a commercial means of incorporating neural networks into many applications. By providing a range of compatible hardware and software products, a variety of solutions can be achieved from this system.

4.1.2 Neurocomputers and ASIC Neural Devices

The next generation of neural implementations is composed of neurocomputers using neural ASIC devices. These implementations can offer a 10^3 increase in performance over neural co-processors by utilising the latest technologies and by building specialised devices. This area of neural device is still undergoing considerable research.
with very few commercial products out on the market yet: most products now available are Neural ASICs out of which neurocomputers can be made. Hitachi is one of the first companies to produce a prototype neurocomputer, but this will not be marketed for the next couple of years.

Below several neural ASICs are discussed. The key points of each design are mentioned, giving a quick review of each implementation, and not necessarily a full description of its operation. The examples chosen indicate the variety of possible solutions, and the different goals and objectives used by the respective research groups. After presenting these examples, assessments are made of the technologies used, and possible drawbacks to these architectures are highlighted.

4.1.2.1 ETANN - Intel Corporation.

By the development of 'Floating Gate Non-Volatile Analogue Memories' Intel have been able to produce a true analogue neural device, the Electrically Trainable Artificial Neural Network (ETANN)[5]. The architecture consists of 64 neurons, and two 4096 synaptic matrices. The first matrix provides an input synapse array, and the second provides a feedback synapse array. This arrangement allows both Hopfield and two layer feedforward networks to be implemented. Sample/hold units are provided to enable the analogue signals to be clocked through the circuitry.

Each synapse is composed of two EEPROM memory cells, an excitatory and an inhibitory cell. The difference between the floating gate voltages (Vfg) of the two EEPROM cells represents the weights value. Diagram 4.2 shows a synaptic cell.

The operation of each synapse is as below:

\[
\text{Weight} = (V_{fg+} - V_{fg-}) \\
\text{Input} = (V_{in+} - V_{in-}) \\
\text{Output} = ((V_{in+} - V_{in-}) \times (V_{fg+} - V_{fg-})) \\
= (I_{out+} - I_{out-})
\]
Summation for each neuron is performed by current summation. The synaptic outputs from each synapse in a neuron are connected, resulting in the total current being the sum of the post-synaptic currents. (Kirchhoff's First Law.) The neural function is performed by a non-linear amplifier on the summated output current.

Learning is performed in a two cycle operation. The ETANN executes a forward iteration, and external hardware then calculates the error terms and modifies the weights. The weights are programmed by a technique known as Fowler-Nordheim Tunnelling. By sending a pulse of 10-12 Volts to the floating gate, tunnelling of electrons occurs between the floating gate and the diffusions. The charge deposited is then used by the transistor to produce the floating gate voltage. By using two memory cells each weight can be increased or decreased by sending pulses to either the excitatory or inhibitory memory cell.
This device provides a low accuracy - 4 bit accuracy - high speed analogue neural implementation. Its speed is $10^{10}$ interconnections/sec, and capacity $8 \times 10^3$ connections.

4.1.2.2 DNNA - Neural Semiconductor Ltd.

The DNNA Chip[6] (Digital Neural Network Architecture) is based on digital pulse trains, allowing very simple operating units, and high speed. The architecture used is very similar to analogue devices with the network topology being hardwired in the form of a matrix. However the use of pulse trains does avoid some of the limitations presented by analogue devices - the stability is improved, and the performance is more predictable. Most important is the facility of inter-chip communications. This poses very large problems with analogue devices since board level capacitance and resistance are very hard to match with on-chip circuitry.

Diagram 4.3 Pulse Trains in DNNA

Diagram 4.3 shows the operating style of this pulse train implementation. Each synapse is composed of an AND gate, requiring only 4 transistors. The summation is performed by a wired-OR; this functions in a manner similar to analogue devices with their current summation. The effect of this wired-OR function also acts as a neural function.
**Low Activation Levels**

The post-synaptic pulses overlap infrequently, so the OR function performs a simple addition.

**High Activation Levels**

The post-synaptic pulses occur more often resulting in more overlaps. The OR function now no longer performs the sum of these pulses, but instead saturates in a manner similar to a sigma function.

Due to the almost limitless expandability of pulse trains the design is packaged as a cascadable synaptic array of 32 x 32 elements. Neuron chips are also supplied which consist of 32 neurons that can be wired to a synaptic array. These neuron chips operate by taking the pulse train, integrating it over a period of time, and producing an output pulse train with the use of a stochastic pulse generator.

DNNA chips provide a hardwired solution to neural implementations. The speed obtained can be traded-off against accuracy - the more pulses used for each iteration allow greater accuracy, at a cost of slower performance. Using 256 pulses per iteration a performance of $2 \times 10^8$ interconnections per second can be achieved on a single synapse array.

**4.1.2.3 Intelligent Memory Chips - Oxford Computer**

The Intelligent Memory Chip[7] presents a very neat solution to matrix and vector operations. They can be used in number of applications that require such operations since they are fully programmable and configurable.

Conventional memory chips require the transfer of data to and from memory to let the processor carry out operations on the data. This operation creates a bottleneck between the memory and processor. The Intelligent Memory Chip has made a radical step by moving the processing from the processors into the memory chips themselves.

Intelligent Memory Chips can perform matrix/vector multiplication on their contents. Each Intelligent Memory Chip consists of two dual-ported
256 x 256 memory arrays - the matrix array and the vector array. The two blocks of data to be multiplied are loaded into these arrays.

For neural network implementations the weights are loaded into the matrix array - with each intelligent memory chip holding a single bit for each weight value. Up to 64Kbits can be held in each intelligent memory chip. The weights for different neurons should be held in different rows, since multiplication for each neuron occurs across a whole row at the same time.

The input values are loaded into the vector array, this array is dual-ported so that this operation can overlap the execution of the intelligent memory chips. Each input value is loaded so that it occupies the same column in memory, with the different bits being held consecutively over a number of rows.

The multiplication is performed simultaneously between an two entire rows, one taken from each memory core. This requires 256 single bit multipliers (AND Gates). The product terms are then summed by an adder with 256 inputs, before being accumulated by the single bit partial product accumulator. To carry out a multiplication between two rows requires n cycles (where n = no. bits in the vector data), and m Memory Chips (where m = no. bits in matrix data).

Over these n cycles each intelligent memory chip presents a single row of the matrix array to n successive rows in the vector array. After n cycles a single bit partial product for each input value in the vector array multiplied by each weight value is obtained in each intelligent memory chip. To obtain the complete product each single bit partial product is added together. Before this addition can occur the bit position for each single bit partial product has to be taken into consideration. This relies upon the weighting of each single bit partial product according to its bit position (ie 2^1, 2^2, 2^3 ...). These weighted partial products are then added to produce a single product value for that matrix/vector multiplication.

Diagram 4.4 shows the internal operation of both the Intelligent Memory Chip, and the Weighted Summer.
The chip set supplied by Oxford Computers consists of the Intelligent Memory Chips, Weighted Adder, controllers and input/output buffers. These can be used in a variety of configurations and are fully programmable. The operation speed obtained varies according to the arrangement used - for 8 bit data $4 \times 10^9$ connections/sec. can be achieved. This configuration would require 8 Intelligent Memory Chips, one for each bit in the weights table.

4.1.2.4 Hitachi Neural Wafer

As mentioned above Hitachi is one of the first company to propose a full scale commercial neurocomputer[8]. This neurocomputer is based on wafer scale integration, with each wafer holding 49 Neural chips connected by a hierarchical bus structure. Each chip holds 12 neurons, giving a total of 588 neurons. Redundancy is built in: out of these 588 neurons only 576 are used. This increases the yield since on average there are
about 10 defects per wafer.

The bus structure is shown in Diagram 4.5. This is hierarchical in structure to allow distribution and buffering, but is logically connected and must be shared by all neurons. This sharing limits the bus bandwidth to a single value per cycle. This allows high performance to be obtained for Hopfield networks since all neurons can operate on the same data. Feedforward networks and reduced connectivity networks however will show a reduced performance since not all neurons will be able to operate with the data on the bus.

Diagram 4.5 Hitachi WSI Structure

The bus arbitration is performed by a central control, which signals the address of the next neuron to output its partial sum onto the bus. This partial sum is output onto the bus with the address of the source neuron. The sigma function is performed on the main bus using shared hardware, before the neural output is sent as input to all neurons.

Each neuron has a local memory of 64 weights only. After learning, which occurs off-chip, only the largest 64 weight values for each neuron are used. To identify that the input value on the bus is relevant for a neuron, special data access hardware is used in each neuron. This is shown in Diagram 4.6. This hardware takes the address of the source neuron for each input value, and by use of a lookup table decides if there is a weights value corresponding to it. This weight is output if
required, otherwise zero is used to indicate that the input value should be ignored.

This WSI (Wafer Scale Integration) implementation shows the ease of expandability of neural devices, and how redundancy can be built in to ensure a high yield. However the communication structure in this design is limited in complexity, which will severely reduce the performance in certain network types. The maximum performance from a wafer is \(2 \times 10^9\) interconnections per second with a capacity of \(3 \times 10^3\) connections.

![Diagram 4.6 Data Access in Hitachi Neuron](image)

**Diagram 4.6 Data Access in Hitachi Neuron**

4.1.2.5 Digital Snake - Univ. Ancona, Italy.

The Digital Snake Implementation[9] is typical of many research level implementations. It presents a device that can be customised to a particular network, thus bringing about a reduction in the complexity and size of the implementation. This implementation is intended for feedforward networks, and can result in a very high performance if the network topology is of a regular structure. A study of the performance
of this type of architecture is carried out in greater detail toward the end of this chapter.

In this example each neuron is represented by a single processor. These processors are linked by linear buses that can be used to shift input and output values through an array.

Three buses are provided, input, output and error propagation. These are linked as shown in Diagram 4.7. Input is provided at one end of the linear array, and is shifted through the array as a computational wavefront. When each processor has processed all the input data, the outputs can then be made available. These are shifted through the output bus and, by a crossover, are made available as input to the lower layers. The operations of both inputs and outputs are overlapped to ensure high performance.

![Diagram 4.7 Digital Snake Bus Structure](image)

4.1.2.5 A Flexible WSI Neural Network - Inst. National Polytechnique

This research project\[10\] has worked on silicon compilation methods that are used to cascade a number of elementary blocks to build a customised neural processor. The individual processors in this design are generated according to a set of parameters, which are then 'tiled' in a regular manner to build the required size array on a wafer.

The global structure of a wafer containing these processors is shown in Diagram 4.8. From this diagram it is possible to see how the use of a
regular structure allows any number of such processors to be tiled together. According to the precision used up to 400 - 800 such processors can be placed on a 4 inch wafer.

To enable a variety of network topologies to be mapped onto a wafer, special bus switches are placed on each of the buses. These switches are defined before fabrication, and will be implemented as either software controlled switches or hard wired switches. These switches allow the feedforward of input values and the backpropagation of errors to be implemented for a wide range of network topologies.

Diagram 4.9 shows how the switches would be arranged for a two layer feedforward network. The hard switches define each layer, and allow the input values for that layer to be shifted through and presented to each processor. Once each layer has completed its computation the soft switches between each layer closes, allowing the outputs to feedfoward from one layer to the next.
This architecture allows the fast customising of an array of neural processors. Using silicon compilation techniques ensures that each implementation matches the required network topology, thus providing an optimum solution.

4.2 Assessment of Neural Implementations

This section carries out an analysis on each type of neural architecture that has been presented. Particular attention is made to the criteria given in Chapter 3 when selecting a suitable architecture. The discussion starts off with analogue architectures, and then follows with hybrid architectures. It will be seen that the properties found in these two are very similar; it is only their computational style that differs.

Digital architectures are then examined, these are taken in more depth since they represent a more probable solution to the criteria given. The
digital architectures are split into two groups, the MIMD arrays and the SIMD arrays. The analysis shows the very different properties of these two groups, and their respective advantages and disadvantages.

4.2.1 Analogue Implementations

Analogue implementations can provide very high speeds in their computations, however they do exhibit several limitations\[11\]. Programmability is hard to include in analogue devices without large overheads\[12\], this leads to most implementations having a fixed connectivity, onto which a network must be mapped. Device variations, electrical instability to both noise and heat, and low accuracy in weights storage are other problems found.

Power consumption and amplifier stability are major problems that limit the scaling of analogue implementations\[13\]. To ensure the stability of a device, the neural function amplifiers should be kept in their active region. This ensures that the power consumption is minimised, and that saturation does not occur in the network. To keep the amplifiers in their active regions, the sum of the input currents must be kept below the threshold level. This becomes more difficult to ensure as the number of synapses entering a neural node increases. To ensure that the sum of these currents does not put the amplifier into saturation, the input currents must be scaled down by the same factor as the increase in the number of synapses. With this reduction in the signal current the stability and accuracy of the system falls, as it becomes more susceptible to temperature, noise and device variations.

Analogue implementations of neural networks are difficult to simulate due to the device variations, and complex characteristics of the analogue components. This can result in off-chip learning becoming inaccurate, requiring the incorporation of on-chip learning\[14\] which in turn adds considerable complexity to the device. This learning problem is neatly tackled by Intel in the ETANN\[5\]. Here the analogue device is used only for the forward phase, with external logic performing the reverse phase, and adjusting the weights. This allows accurate calculations of the errors to be made, increasing the network's performance.
Other limitations of analogue devices are the difficulties in matching off-board capacitance and resistances. This tends to limit a network to a single chip, restricting the size of network that can be implemented.

It can be concluded that analogue neural implementations provide good solutions in small, noise free applications. Here their lower accuracy can be counteracted by their high speeds and asynchronous nature\cite{15}. Visual pre-processing is a good application area for such devices\cite{16,17}.

These factors indicate that an analogue implementation would not be suitable as an architecture to meet the requirements given in Chapter 3. The main objection is the lack of the soft-programmability that is required to allow a device to be configured for a variety of different networks. Other problems relate to the difficulties encountered in off-chip learning, making it difficult to mass produce large numbers of such devices.

A final point to mention is the technology required to fabricate analogue VLSI devices. Two polysilicon layers are required to build the capacitors. This was not available to me, since the ES2 process that London VLSI Consortium uses only incorporates a single polysilicon layer. This would limit any proposed design to metal/metal or metal/poly capacitors. These have a very small unit capacitance, resulting in the use of large chip areas on which to build the capacitors.

4.2.2 Hybrid and Pulse Neural Implementations

To solve some of the problems encountered in analogue implementations, hybrid and pulse implementations have been proposed. These offer very similar properties to analogue implementations, but with some considerable advantages.

Many earlier hybrid implementations used digital memory to store the weights\cite{18}. This was primarily due to analogue memories being unavailable at the time, and also to their increased stability, and the possibility of using conventional fabrication processes.
The most common hybrid implementations now are pulse train devices \cite{6,19,20}. These devices are constructed in a very similar manner to analogue implementations, and differ only in the computation and connection style. Instead of using variable voltages or currents, variable frequency pulses are used, in a manner very similar to biological nervous impulses. These pulse trains avoid the noise and temperature problems encountered in analogue implementations. They also enable expandability, since digital pulses can be communicated between chips without affecting their values.

This class of neural architectures was also eliminated from the possible solutions for the target application area. The reason again being the lack of soft-programmability that was seen to be crucial in the requirements for the target architecture.

### 4.2.3 Digital Implementations

Digital implementations can solve many of the problems associated with analogue techniques. Digital devices benefit from increased stability, the arbitrary accuracy that can be achieved, and most importantly their soft-programmability. All of these features point to the benefits of using digital techniques for the application areas that this device is proposed for.

When considering a possible architecture with which to build a neural processor, a look at the different ways parallel arrays are constructed is required. The key features to any parallel array is how the different processors are controlled, and how the communications between the processors occur. Both these features weigh very heavily in the design, and to the resulting properties that different arrays exhibit. The two main control types for parallel arrays are considered below.

#### 4.2.3.1 MIMD Arrays

The neural co-processors discussed in this chapter [NETSIM and DELTA] both incorporate MIMD processors. These processors make use of parallel multipliers and pipelined architectures to obtain a very high
Due to their high performance it is possible to time multiplex the execution of a network across a number of such processors. The ability to do this can considerably reduce the number of processors required for an implementation. By careful calculations the number of processors can be estimated to achieve the required run time performance for any network. In MIMD arrays the speed up obtained from the use of multiple processors can be almost linear with the number of processors used, provided that care is taken to ensure that the communication network does not become overloaded.[21]

The technique of mapping neurons onto an arbitrary size array I have termed in this thesis as 'Virtual Neurons', since each neuron is not represented by a physical processor, but instead by a logical mapping onto the array. It is this factor which gives MIMD arrays their versatility, and their ability to implement almost any network topology.

The communications carried out in MIMD array involves message passing.[21] This requires the packaging of each value to be transmitted, these packets contain the source address, destination address and the actual value. This packet is then transmitted from the source onto the communications bus. The technique used in actually transmitting this packet of data varies, and will depend upon the connectivity between the processors. For neural processing there are two suitable forms of connectivity, which are mentioned below.

- **Shared Bus**[4]

The DELTA co-processor is an example of a shared bus communication strategy. In this case up to 30 processors can be connected onto a high bandwidth bus. The message passing on this bus is carried out by an asynchronous communications manager for each processor. This monitors the activity on the bus, transmitting a packet of data only when the bus is free, and picking up any packets addressed to it. These packets are then be held in a buffer until required by the processor.
Multi-Dimension Array\(^{(22,23)}\)

This type of communication strategy is seen in the NETSIM co-processor. In this case the processors are arranged in two or three dimensional arrays, with each processor providing links to its neighbours. Messages are passed from processor to processor until they reach their destination. This form of communication strategy can match that required from neural networks very efficiently, since local connections are predominant in neural networks.

To obtain high performance, complex routing algorithms can be used to ensure the most efficient route is taken for a packet. These algorithms also ensure that bottlenecks do not occur by rerouting packets around blocked channels.

The key point to note in both these communications techniques is the fact that any processor can communicate with any other processor. This property I have termed in this thesis as being a single level communication strategy. This has the effect of totally freeing the network mapping, allowing almost limitless possibilities in the assignment of neurons to processors. It will be seen that this feature of MIMD arrays plays a large part in giving MIMD arrays their flexibility and high performance.

One drawback to this style of communications strategy is the requirement for some form of host processor to provide the input and output from the array. This host must fit into the communication strategy, and must use the same communications protocol as that used by the array. This requirement for a host processor can add to the overall complexity of a final implementation.

4.2.3.2 SIMD Arrays

SIMD arrays use very large numbers of processors to produce a fine granularity array. This approach can be seen to be very similar to that used in biological systems, with very small individual processing elements working in unison.
There are two basic methods used in assigning the workload of a network across an array. These two methods I have termed as:

Variable Network Mapping
Fixed Network Mapping

Variable Network Mapping

In variable network mapping no fixed mapping of neurons to processors is carried out. Examples of this are the Hitachi Wafer and Intelligent Memory Chips. Both implementations use a fixed number of processors, capable of different mappings of a network across this pool of processors.

In the Hitachi Wafer each processor is used for a single neuron, with the mapping of each neuron being defined by the weights in each processors weights block. In this example external global control is required to arbitrate the common communications bus.

The Intelligent Memory Chips provides an array of very low granular processors. In this case there is no assignment of an individual processor to each neuron, but instead a time multiplexing of each neuron through the array. The control of this process is by software, which defines the vector and matrix blocks that are to be multiplied, and where the results are to be placed.

Both these implementations require external hardware. The Hitachi Wafer requires a host processor and bus arbitration unit, while the Intelligent Memory Chips require Weighted Summers, micro-controllers and vector input/output interfaces. These neural implementations provide good solutions for neural ASICs building blocks, out of which neurocomputers can be made. They are highly versatile, offer very high speed, and can be expanded to an arbitrary size. However their suitability for a small size device is considerably inhibited by the requirement for the control and arbitration logic.
Fixed Network Mapping

This class of neural implementation can provide a very compact, and efficient device. Network mapping is defined by dedicated bus structures, which removes the requirement for external arbitration or global control. Each processor can work autonomously with a minimal control strategy thus exploiting the distributed nature of neural networks and their local interconnections, by the use of these dedicated bus structures. This considerably reduces the device complexity, and can bring down the size of an implementation to a single chip.

The following array topologies have been proposed for SIMD neural arrays.

• **Shift Arrays**[9]
  The processors are arranged in a linear array, with the output from one layer being shifted, and used as input to the following layer.

• **Broadcast Bus**[24]
  The processors are arranged in layers with the outputs from one layer being connected to all inputs in the following layer via a broadcast bus.

• **Single and Multi Dimension arrays**[25]
  Communications are performed by message passing via processor-to-processor links. Such a strategy requires the incorporation of routing hardware within each processor.

Diagram 4.10 shows an outline of how each of these communication methods would be used to implement a network of 4, 3 and 4 units in the 1st hidden, 2nd hidden and output layers respectively.

The communication strategies in these examples tend to mimic the network topology fairly closely, so that each neuron is mapped onto a single processor. This mapping of neurons to processors is determined by the processors' physical position within the communication structure. For example in the broadcast bus communication structure, each layer of
processors will be used to represent a layer of neurons, with each processor implementing a single neuron.

The simple communications style of SIMD arrays opens up the possibility of running an array in a stand alone configuration. By the use of simple logic the input and output to the array can be interfaced directly to external hardware. For example A/D and D/A converters could be used to provide direct input and output without the need for an intermediate host processor. Such a scheme could dramatically reduce the overall complexity of an implementation.

4.3 Analysis of SIMD Arrays

Two good studies have been carried out on the implementation of neural networks on SIMD arrays[26,27]. These studies show a small variation in performance between different communication strategies, with some strategies being better suited to certain network topologies than others. I will not recover the work done in these papers, but instead I will generalise all SIMD implementations into a single group. This generalisation is valid since it uses the best theoretical performances of these SIMD arrays.

When used to implement a multilayer networks a SIMD Array will operate in a manner similar to a pipeline. This is because each
layer of processors uses the previous layer's output for its input. This arrangement will exhibit all the properties of a pipeline processor with regard to throughput (pipeline iteration time) and latency (time from input affecting output).

Using this assumption it is now possible to isolate various problems that SIMD arrays present, and also to predict their performance in implementing specific networks.

4.3.1 Flexibility

The first problem found in SIMD arrays is due to the inflexibility of these implementations. The requirement for the bus structure in each implementation to match the network topology results in a customised device having to be built for each network. This factor seriously limits the flexibility of these implementations, and means that the soft programming of a network onto a device is not possible.

4.3.2 Performance

As with any pipeline, the time required between each stage in the pipeline is determined by the slowest stage. This can cause major problems in some network topologies.

If the number of units in the different layers varies, then the time taken to compute the layer with largest number of inputs will determine the pipeline iteration time. Idle cycles will have to be inserted into the shorter layers to keep them in synchronisation. This problem is particularly acute in signal processing applications, where the input layer tends to be extremely wide. This width is required to capture time and frequency components of the signal being processed.

The method that I have used in the analysis of the performance of different architectures involves looking at the number of cycles required to implement each neuron, and the number of idle cycles that a particular architecture will impose.

For this analysis I shall take the Tx Extraction Network presented in
Chapter 3 as an example. The key points in this network are:

- 246 Input Units
- 6 Neurons in 1st Hidden Layer
- 6 Neurons in 2nd Hidden Layer
- 1 Neuron in Output Layer

When implemented on a SIMD array each layer of the network is treated as a pipeline stage, requiring a three stage pipeline for the network. The pipeline iteration time is 246 Time Units, this representing the widest layer (246 Units in Input Layer). To synchronise the array 240 idle cycles are inserted for the 2nd hidden layer and the output layer. The processors in both these layers operate at a considerably reduced efficiency.

Table 4.1  SIMD Array Performance

<table>
<thead>
<tr>
<th>Mapping</th>
<th>1st Hidden Layer</th>
<th>2nd Hidden Layer</th>
<th>Output Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 PEs</td>
<td>6 PEs</td>
<td>1 PE</td>
</tr>
<tr>
<td>Iteration 1</td>
<td>246</td>
<td>6 + 240 Idle</td>
<td>6 + 240 Idle</td>
</tr>
<tr>
<td>Iteration 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iteration 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From this analysis it is possible to compute the average efficiency of the processor array:

- 1st Hidden Layer: 246 Synapses \( \frac{246}{246} \times 100 = 100\% \)
- 2nd Hidden Layer: 6 Synapses \( \frac{6}{246} \times 100 = 2.4\% \)
- Output Layer: 6 Synapses \( \frac{6}{246} \times 100 = 2.4\% \)

The total efficiency can be calculated by taking the sum of the efficiency of each neuron, and finding the mean of this value.

- 1st Hidden Layer: 6 Neurons \( 6 \times 100 = 600 \)
- 2nd Hidden Layer: 6 Neurons \( 6 \times 2.4 = 14.4 \)
- Output Layer: 1 Neuron \( 1 \times 2.4 = 2.4 \)
Average for Whole Network: 13 Neurons

\[ \frac{616.8}{13} = 47.5\% \]

This calculation shows the low efficiency obtained from an array of SIMD neural processors arranged in a Fixed Network Mapping. The mean utilisation is 47.5% and more importantly, is that 7 out of the 13 processors are only operating at 2.4% utilisation.

### 4.3.3 Real-Time Constraints

Achieving the required real-time performance can also present problems for such an implementation. Since the speed of operation is determined by the pipeline iteration time and the number of layers, there are limited methods that can be used to adjust the performance of a network implementation. In any real-time system there are two primary constraints that have to be met:

- **Sample Period**
  This determines the speed that data is input to a system. The sample period can be critical in many digital signal processing applications. If it is not obtained then frequency folding will occur in the input frequency spectrum, causing great inaccuracies in the models being implemented.

- **Response Time**
  This represents the time that occurs between a change on the input affecting the output. This can often be critical, for example in speech processing, the delay must be short to avoid any mismatch between audio and visual signals.

It is important to achieve both these constraints as closely as possible. If the device is too slow then problems such as those indicated above will occur. If it is too fast then the device must be more complex than necessary, representing wasted silicon area and power consumption. Therefore it is important to be able to adjust the real-time performance of an implementation to match that which is required.
4.4 Summary of Available Neural Implementations

From the assessments carried out in this chapter, it has been possible to highlight the relevant advantages, and disadvantages of a variety of neural implementations. For the very specialised requirements laid down in Chapter 3, none of the above neural implementations are entirely suitable. It is however possible to suggest that digital architectures present the best solution. The properties from the two main classes of parallel digital architectures have been discussed, and form an important feature in remainder of this thesis. These feature are summarised below:

- **MIMD Arrays**
  MIMD arrays rely upon a software mapping of a network across an array. Each processor is programmed individually to enable this, with the number of processors in the array being considerably less than the number of neurons in the network. This scheme gives MIMD arrays considerable flexibility, with the benefit of soft programming of a network onto an array.

  However these benefits come at a cost in increased complexity. The requirement for each processor to have its own control, local memory and communications support all add considerably to the size, cost and power consumption of an implementation.

- **SIMD Arrays**
  SIMD arrays can offer much simpler methods in the implementation of neural networks. The devices seen in this chapter make use of hard wiring to define network topologies, eliminating the need for complex control and communications, but at a cost in reduced flexibility, and no soft programmability.

By considering the beneficial features of each of these types of architecture a hypothesis can be formed. This hypothesis states:

If an architecture could be developed that combines the properties exhibited by both MIMD and SIMD arrays, then a device could be
built that exhibits all the following properties:

- High Flexibility
- Soft Programmable
- High Performance
- Low Complexity
- Low Power Consumption
- Low Cost of Implementation

This hypothesis drove the development of an architecture, which is presented in the remainder of this thesis.
Proposed Architecture

This chapter takes the hypothesis presented in Chapter 4, and expands the ideas behind it further. By looking at MIMD and SIMD arrays it has been possible to extract the architectural features that are attributed to their respective properties, and from this study to develop an architecture to combine these features.

5.1 Objectives for the Architecture

The development of the proposed architecture was initiated by the requirements for the target application presented in Chapter 3. A study of this, and related applications, allowed the identification of the desirable properties that an architecture should exhibit to support these applications efficiently. From the study of current neural architectures none were identified as having these properties. However this study was able to identify the beneficial and detrimental properties from the two main classes of digital neural architectures - MIMD and SIMD arrays. These are summarised in Table 5.1 and Table 5.2.

<table>
<thead>
<tr>
<th>Table 5.1</th>
<th>MIMD Array Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advantages</td>
<td>Disadvantages</td>
</tr>
<tr>
<td>High Flexibility</td>
<td>High Complexity</td>
</tr>
<tr>
<td>Software Programmable</td>
<td>Complex Communications</td>
</tr>
<tr>
<td>High Efficiency</td>
<td>Requires Host Processor</td>
</tr>
</tbody>
</table>

The conclusion from this study was that if the beneficial features from both types of arrays could be combined, the resulting device would represent an ideal solution for the application areas discussed in Chapter 3.
### Table 5.2 SIMD Array Properties

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low Complexity</td>
<td>Low Flexibility</td>
</tr>
<tr>
<td>Simple Communications</td>
<td>Hard Programmability</td>
</tr>
<tr>
<td>Stand Alone Operation</td>
<td>Possible Low Efficiency</td>
</tr>
</tbody>
</table>

### 5.2 Development of Architecture

It was this combination of the advantageous features from both MIMD and SIMD arrays that represented the main objective for the work presented in this thesis. The following sections in this chapter outline the steps that were taken in the development of an architecture that could fulfil this. A summary of these steps is presented below:

**Step 1**
- **Extract Desirable Features from MIMD and SIMD Arrays**
  This involved looking at MIMD and SIMD arrays, and deciding what features in their respective architecture could be attributed to each one of their beneficial properties.

**Step 2**
- **Develop Methods to Implement these Features**
  Methods had to be derived to enable the features from step 1 to be implemented in the same architecture. This stage was crucial in the development of the architecture, since the features isolated from MIMD and SIMD arrays tended to be inconsistent with each other.

**Step 3**
- **Develop an Architecture to Combine these Features**
  The final step was the development of a unified architecture to include all these features.
5.2.1 Desirable Features of MIMD Arrays

The three main beneficial properties of MIMD arrays that this architecture is attempting to capture are:

- High Flexibility
- Soft Programmability
- High Efficiency

All three of these properties can be attributed to the same feature in MIMD arrays, that is the ability to perform a logical mapping of a network onto an array, by the use of Virtual Neurons.

The use of Virtual Neurons allows the assignment of neurons onto the MIMD array to be done in a multitude of ways - with either several Virtual Neurons to a processor or the splitting of a Virtual Neuron across several processors. This arbitrary assignment of Virtual Neurons to each processor has several benefits:

- High Flexibility
  Virtual Neurons allow much greater flexibility in the mapping of neurons onto processors. It is possible to implement many different network topologies on the same device by simply reconfiguring the network mapping.

- Soft Programmability
  Since each processor has its own control, and individual program, the network mapping is fully programmable. This factor considerably simplifies the development of a network implementation, and allows greater versatility to be obtained.

- High Efficiency
  Virtual Neurons allow the workload of a network to be more evenly distributed across the available processors, ensuring that the number of idle cycles in an implementation is kept to a minimum. Virtual Neurons can also be used in several ways to help achieve the real-time constraints that an application demands. It enables close to linear speed-up with the number of processors, thus
allowing the number of processors to be altered to achieve the required performance.

The implementation of Virtual Neurons requires two special features in an architecture:

• **MIMD Control**
  MIMD control provides the ability for each processor to execute a different program, enabling different virtual neurons to be implemented on each processor.

• **Single Level Communications**
  The use of a single level communications scheme allows complete flexibility to be exploited in the network mapping.

5.2.2 Desired Features of SIMD Arrays

The main beneficial properties of SIMD arrays that this architecture is attempting to capture is:

• Low Complexity
• Simple Communications
• Stand Alone Operation

It is more difficult to attribute these properties in SIMD arrays to any single feature of their architecture. Probably the most important single feature in these arrays is that they are controlled by a single central controller, to which all these beneficial properties can be attributed:

• **Low Complexity**
  Removing program control hardware and complex communications controllers considerably reduces the complexity of SIMD arrays.

• **Simple Communications**
  SIMD control can bring about considerably reduced complexity in the communications when compared to MIMD control. This is achieved because each processor is synchronised by the SIMD control, eliminating the requirement for buffering and synchronisation.
• Stand-Alone Operation
The use of SIMD control, and simple communications strategies, allows such a device to operate without a host processor. It is possible to provide input and output directly from I/O devices, for example an A/D and D/A converter can be directly connected to an array with the minimum of additional hardware.

5.3 How to Combine these Features

The architectural features extracted from MIMD and SIMD arrays that it is intended to combine are:

• MIMD Control
• Single Level Communications Strategy
• SIMD Control
• SIMD Communication

The next stage in the development of this architecture was to find a means to combine these features into a single device. It was envisaged that the bulk of the processing, and communications, would be carried out under SIMD control, with MIMD control being associated with the scheduling of virtual neurons onto each processor.

5.3.1 Combining MIMD and SIMD Control

The combination of both MIMD and SIMD control strategies presents several problems:

• Where to store the two different control strategies
• How to select between the control strategies
• How to represent each virtual neuron
• How to implement the communications scheme
• How to synchronise between processors
• How to do all the above with the minimum of complexity

The processes involved in solving these problems, and the options that were available, are discussed in the sections below. The solutions
eventually used are then presented as an overall architecture, bringing all the ideas presented together.

5.3.1.1 Storage of MIMD and SIMD Control Strategies

The solution proposed for the storage of MIMD and SIMD control strategies involves the use of two different program stores. This is necessary since the hardware requirements for each of these memory blocks are different.

The SIMD program has to be broadcast to the array, adding extra complexity to the SIMD program store. This is due to the additional sequencing and synchronisation that needs to be performed during the broadcasting of the SIMD program.

The requirements for the MIMD program store are simpler since each processor controls its own program flow, allowing conventional memory to be used. It was decided to hold the MIMD program local to each processor, an alternative method would have involved a central MIMD store with bus arbitration and memory protection schemes. However the use of this central store was rejected due to bus bandwidth problems.

5.3.1.2 Control Switching and Virtual Neuron Definition

Once the storage requirements of the two control strategies were defined, it was possible to consider how to control the switching between the control types, and how to define each virtual neuron. These two problems are related, and are considered together.

The representation of each virtual neuron involves loading the various pointers, the definition of the weights, computation of the sigma function, and the writing of the neuron's output. It was decided to store the weights together with the MIMD instructions, minimising the number of memory blocks, and pointers required.

The technique of combining data and instructions is not a new idea in neural processors - for example it is seen in the Phonetic Typewriter[11], where a TMS320 Digital Signal Processor is used. This
processor uses fast on-chip memory, which allows an immediate addressing mode to be used without any reduction in speed. This requires each weight to be defined as an immediate operand in the multiply instruction:

\[
\text{MPYK weight} \quad \#\text{Multiply Immediate}
\]

Such a technique enables both data and program instructions to share the same memory without the use of any additional control logic. However it does represent a considerable overhead in the memory requirement; this being due to each weight having to be defined as an opcode followed by a data value.

**First Proposed Solution**

The method first considered for this architecture used a similar scheme to this immediate operand addressing mode. However in this architecture there is no need to define the multiply opcode for each weight since this operation is performed under SIMD control. All that would be required in this case would be some means to distinguish between the following:

- Use this byte as a Weights Value
- Use this Byte as a MIMD Instruction

To distinguish between these two, a single flag bit is required, which would be added to each byte in the MIMD program store. From this flag bit each processor could decide whether to execute a byte as a MIMD instruction, or whether to switch to SIMD control, and use the byte as a Weights Value.

This method is shown in Diagram 5.1(a). Here the flag bit from the memory is used to select which instruction to execute. If the flag bit is high SIMD control is used, and if it is low the byte is used as a MIMD instruction.

This scheme allows the weights and MIMD instruction to co-exist in memory with a fairly low overhead (9 bits required to hold each byte of
Diagram 5.1 Methods Proposed for MIMD and SIMD Switching

data). This scheme also provides a neat solution to the selection between the two control strategies; this flag bit can be used directly to select between SIMD control or MIMD control.

This scheme was eventually discarded when it decided that external memory was going to be used for the first implementation of the architecture. The use of 9 bits to store each byte is not convenient, this was seen to be a major obstacle with this technique. What was then looked for was a more conventional method that could be used to distinguish between the weights and MIMD instructions.

Second Proposed Solution

The second method proposed the use of a special instruction at the beginning of a block of weights to switch between MIMD and SIMD control. This scheme allows conventional memories to be used, since the need for the additional flag bit has been removed.

One possible implementation using this scheme is shown in Diagram 5.1(b). In this arrangement the instruction multiplexer is controlled by a status register in the datapath: this status register acts as a zero flag for a counter register. This counter register is loaded with a value upon the start of a block of weights, resulting in the zero flag being reset, and a switch being made to SIMD control. This counter register is then decremented each time a weight value is read, resulting...
in the zero flag becoming set when the last weight value is read. This will then switch the control for that processor back to MIMD control.

This scheme seems to provide a fairly conventional method to controlling operations on blocks of data. The use of block processing instructions can seen in many processors. The Z80\(^2\) has a series of block processing instructions, which use general purpose registers to control how many times an instruction is to be repeated. An example being the CPDR (ComPare, Decrement and Repeat).

The use of a counter to control the number of iterations for an instruction presents some overheads. An additional register is required to hold the counter, and additional cycles are required to decrement the counter, and to test the zero flag. These extra cycles are seen as a major drawback in the operation of the scheme, so this scheme was rejected.

**Final Solution Used**

The solution finally proposed in shown in Diagram 5.1(c). Here the use of a special instruction at the start of a block of weights is used to select SIMD control. This instruction is decoded conventionally by the instruction decoder, which sends a signal to a finite state machine that actually controls the instruction multiplexer.

The end of a block of weights is not determined by a counter, but instead by a special instruction that is positioned at the end of each weights block. This instruction is identifiable from the weights since it represents a non-valid weights value, and is detected by special logic built into the instruction multiplexer.

This final method offers a very low overhead in time and hardware requirements. The addition of the two instructions at the start and end of each weights block offers low memory overheads for the majority of networks, and the logic required to detect these instructions is minimal.
5.3.1.3 Proposed Communication Scheme

Before considering the communications strategy, it is worth mentioning the various values that are to be transmitted between each processor. The most obvious of these are the neural inputs and outputs; these are transmitted between each virtual neuron, and form the bulk of the communications workload. A single-level communications scheme should be used for this, since this ensures maximum flexibility in the network mapping.

The other values to be transmitted are the partial sums for those virtual neurons that are spread across more than one processor. The performance of this operation is less critical, since this is an infrequent event.

Communications of Neural Inputs and Outputs

As previously discussed most of the processing is carried out under SIMD control, ensuring that each processor remains synchronised. This enables a very simple communications scheme to be used for the transmission of neural input and output values.

The communication scheme used is a shared memory, with a common bus. This communications scheme must be one of the simplest available, offering very low hardware overheads.

The arbitration for the common bus is performed by spreading out the SIMD instructions across the array. This has the effect of pipelining the instructions in the SIMD program through the array, thus ensuring that no two processors operate on the same instruction simultaneously.

Such a simple communications scheme does present a limit to the number of processors that can be used in a single array. In this case the limit is determined by the number of instructions in the SIMD program. However since this architecture is aimed at a small-scale device, this lack of expandability is not seen as a major disadvantage.
Communications of Partial Sums

To be able to implement a virtual neuron on more than one processor requires the ability to add the partial sums across these processors. To achieve this a bus is required that can transmit the partial sum from one processor to another. It will be seen that this bus is also used for the sigma function lookup table.

A complication in the partial sum communications is that it is performed under MIMD control, requiring some form specialised synchronisation. This synchronisation is performed at compile-time thus minimising the hardware requirements that would have been necessary for run-time synchronisation. The synchronisation scheme used for this is shown in more detail in Section 5.6.1.4.

5.3.1.4 Synchronisation of Processors

As in any array of processors, it is important to be able to synchronise processors in the array for certain events. In this architecture there are two main synchronisation events that need to be considered:

- Synchronise SIMD Control between Processors
- Synchronise Each Network Update

The first of these ensure that when a processor switches to SIMD control, it will only start to execute the SIMD program at the first instruction in the SIMD sequence. The second of these events is to enable the array to be synchronised at the start of each network update. This is important since input and output from the device must occur between each network update, requiring each processor to wait while this occurs.

Synchronise SIMD Control between Processors

To synchronise the execution of the SIMD control requires each processor to start the execution of the SIMD program on the first instruction in the sequence. To achieve this a synchronisation signal is broadcast with the first instruction of each SIMD program iteration, with each
processor only commencing SIMD control when this signal is present.

Synchronise Each Network Update

The synchronisation of each network update requires first an indication of when each processor has completed all its virtual neurons, and then an indication to when the next network update can commence.

To indicate the completion of each network update a signal is generated: this is controlled by an instruction in the MIMD program. This signal is daisy-chained between the processors to indicate when all processors have completed their virtual neurons. This signal can then be used to control the input and output from the device. To indicate when this input and output is complete, an externally generated signal is broadcast to each processor, this results in the next network update commencing.

5.4 Proposed Architecture

Now that the key points have been discussed it is possible to present the overall architecture. An outline of this architecture is shown in Diagram 5.2.

Processor Array

This array is composed of custom built processors that operate on a special instruction set optimised for neural models. These processors are each connected to four buses:

- **Sigma Bus**
  The Sigma Bus connects each processor to the Sigma Function lookup table. This bus is also used for transferring partial sum values between processors. This bus is multiplexed between a 32 bit partial sum, which is used as a lookup address, and an 8 bit data value.

- **SIMD Bus**
  The SIMD Bus is a bi-directional bus that is used to broadcast the SIMD program to the array, and to monitor the status of the
Diagram 5.2 Outline of Architecture

array.

- **Input Bus**
The Input Bus is shared between the processors, and is used to access the Input Table. This bus is multiplexed between a 16 bit address and an 8 bit data value.

- **Weight Bus**
The Weight Bus is owned individually by each processor, and is multiplexed between a 16 bit address and 8 bit data value.

The other features of this architecture are required to implement the global control, to provide storage for the input values and weights and the implementation of the sigma function. These blocks are described below:
• SIMD Control
The SIMD control operates by broadcasting the SIMD program to the array, this program updates a single synapse upon each iteration. The SIMD control is also used to synchronise the processor array, and external input and output devices.

• Weights Table
The Weights Tables hold both the weights and the MIMD instructions, with each processor having exclusive access to its own weights table.

• Input Table
The Input Table provides the communications between each virtual neuron and the input and output from the array.

• Modulo Address
The Modulo Address generation is performed under control from each processor, and is used in the implementation of input windows. This method of implementing an input window follows a similar technique to that used by the Motorola DSP56000. (See Chapter 3 section 3.5.2.1.)

• Sigma Function
The Sigma Function provides a lookup table with which to compute the sigma function. The use of a lookup table allows the sigma function to be altered for different network characteristics.

• Sigma Threshold
The Sigma Threshold is used to cut down on the size of the sigma lookup table required. This is required since simulations have shown that a 2K bytes lookup table is adequate for this type of application; this threshold function is used to compress large partial sums into this 2k bytes lookup table.

5.4.1 Architecture of Processing Element

The architecture of each processor follows that used by conventional processors incorporating RISC techniques[3]. These techniques minimise
the complexity of the processor by using a restricted instruction set, and by ensuring that all instructions are executed in a single cycle.

Diagram 5.3 gives an overview of the processor architecture. This processor incorporates a 16 bit datapath, instruction decoder and status register, and the specialised hardware to perform the selection between the two control types.

**Diagram 5.3 Overview of Processor Architecture**

- **Datapath**
  A sixteen bit datapath incorporating special features to implement byte and long word operations.

- **LOCAL FSM**
  This finite state machine controls the next instruction selection, and is used to switch and synchronise between the two control types.
• **Instruction Multiplexer**
  This multiplexer selects the next instruction to be executed, and is controlled by the LOCAL FSM.

• **'ESCAPE' Logic**
  This block is positioned on the weight bus, and is used to detect the instruction that indicates the end of each block of weights.

• **Delay Unit**
  This block is positioned on the SIMD control bus, and imposes a single cycle delay to the SIMD program. This delay has the effect of pipelining the SIMD program throughout the array, thus performing the SIMD arbitration.

5.5 **Programming Strategy**

The requirements that are imposed on the programming strategy have been introduced earlier in this chapter; the main requirement being the ability for both SIMD and MIMD control types to be incorporated in the design. When considering the programming strategies for these control types it was decided to use a single instruction set, which simplifies the programming, and allows both control types to be decoded on the same hardware. The instructions are:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD_WEIGHT</td>
<td>Load Weight Value and Increment Pointer</td>
</tr>
<tr>
<td>LD_INPUT</td>
<td>Load Input Value and Increment Pointer</td>
</tr>
<tr>
<td>LD_SIGMA</td>
<td>Perform Sigma Function on Partial Sum</td>
</tr>
<tr>
<td>MULTI</td>
<td>Multiply Iterate</td>
</tr>
<tr>
<td>MULTT</td>
<td>Multiply Terminate</td>
</tr>
<tr>
<td>RESET</td>
<td>Reset Processor</td>
</tr>
<tr>
<td>ADD_PAR_LSBW</td>
<td>Add Partial Sum LSWord</td>
</tr>
<tr>
<td>ADD_PAR_MSBW</td>
<td>Add Partial Sum MSWord</td>
</tr>
<tr>
<td>ACC_RESET</td>
<td>Reset Accumulator</td>
</tr>
<tr>
<td>LOCAL_OFF</td>
<td>Start of Weights</td>
</tr>
<tr>
<td>STOP</td>
<td>Stop at End of Network Iteration</td>
</tr>
<tr>
<td>BIAS_LSBW</td>
<td>Load Bias LSByte LSWord</td>
</tr>
<tr>
<td>BIAS_MSBW</td>
<td>Load Bias MSByte LSWord</td>
</tr>
<tr>
<td>BIAS_LMSW</td>
<td>Load Bias LSByte MSWord</td>
</tr>
<tr>
<td>BIAS_HMSW</td>
<td>Load Bias MSByte MSWord</td>
</tr>
<tr>
<td>MODULO_ON</td>
<td>Turn Modulo Address Generation On</td>
</tr>
<tr>
<td>MODULO_OFF</td>
<td>Turn Modulo Address Generation Off</td>
</tr>
<tr>
<td>LD_PAR_LSBW</td>
<td>Load Partial Sum LSWord from Sigma Bus</td>
</tr>
<tr>
<td>LD_PAR_MSBW</td>
<td>Load Partial Sum MSWord from Sigma Bus</td>
</tr>
<tr>
<td>INPUT_PTRL</td>
<td>Load Linear Input Pointer LSByte</td>
</tr>
<tr>
<td>INPUT_PTRH</td>
<td>Load Linear Input Pointer MSByte</td>
</tr>
<tr>
<td>ADD_MOD_STEP</td>
<td>Add Modulo Step to Modulo Input Pointer</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>Output Neuron Result</td>
</tr>
<tr>
<td>RETURN</td>
<td>Jump to Weight Table 0h0000</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
</tr>
</tbody>
</table>

**Instruction Set**

93
5.5.1 SIMD Program

The SIMD control is fixed, and is generated by dedicated hardware on-chip. This program consists of a sequence of instructions that performs a single synaptic update with each iteration, this being continuously broadcast to the array.

The SIMD program is given in Listing 5.1. This program consists of 14 instructions that will fetch an input value and its associated weight, increment the pointers, multiply the weight and input value and add the product to the partial sum.

```c
# SIMD Synaptic Update Program
# This Sequence of Instructions is Continuously Broadcast to the Array
# Each Iteration Updates a Single Synapse.
LD_WEIGHT       #Load Weight Value and Increment Weight Pointer
LD_INPUT        #Load Input Value and Increment Input Pointer
RESET_ACC       #Reset Accumulator
MULTI
MULTI
MULTI
MULTI
MULTI
MULTI
ADD_PAR_LSW     #Add Partial Sum LSWord
ADD_PAR_MSW     #Add Partial Sum MSWord
```

Listing 5.1 SIMD Synaptic Update Program

This program utilises 9 multiply cycles for each synaptic update. This is required since 8 bit data values are used, with the ninth cycle being required to ensure that the sign bit of the product does not become corrupt. (See Section 6.2.3.)

The distribution of the SIMD program across the array is performed by pipelining the instructions through the array as previously described. This operation can be shown diagrammatically in Table 5.3 where the execution of the SIMD program across four processors is shown.

The bold type in the table highlights the instructions that require arbitration, and the italics are used to indicate SIMD control.

The continuous broadcasting of the SIMD program allows the autonomous
updating for each virtual neuron. All that is required from each processor is the scheduling of each virtual neuron. These operations are performed by the MIMD program.

Table 5.3 Execution of SIMD Program Across an Array

<table>
<thead>
<tr>
<th>TIME</th>
<th>PROCESSOR 1</th>
<th>PROCESSOR 2</th>
<th>PROCESSOR 3</th>
<th>PROCESSOR 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME 0</td>
<td>LD_WEIGHT</td>
<td>ADD_PAR_MSW</td>
<td>ADD_PAR_LSW</td>
<td>MULTI</td>
</tr>
<tr>
<td>TIME 1</td>
<td>LD_INPUT</td>
<td>ADD_PAR_MSW</td>
<td>ADD_PAR_LSW</td>
<td>MULTI</td>
</tr>
<tr>
<td>TIME 2</td>
<td>RESET_ACC</td>
<td>ADD_PAR_MSW</td>
<td>ADD_PAR_LSW</td>
<td>LD_WEIGHT</td>
</tr>
<tr>
<td>TIME 3</td>
<td>MULTI</td>
<td>ADD_PAR_MSW</td>
<td>ADD_PAR_LSW</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 4</td>
<td>MULTI</td>
<td>RESET_ACC</td>
<td>MULTI</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 5</td>
<td>MULTI</td>
<td>MULTI</td>
<td>MULTI</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 6</td>
<td>MULTI</td>
<td>MULTI</td>
<td>MULTI</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 7</td>
<td>MULTI</td>
<td>MULTI</td>
<td>MULTI</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 8</td>
<td>MULTI</td>
<td>MULTI</td>
<td>MULTI</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 9</td>
<td>MULTI</td>
<td>MULTI</td>
<td>MULTI</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 10</td>
<td>MULTI</td>
<td>MULTI</td>
<td>MULTI</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 11</td>
<td>MULTI</td>
<td>MULTI</td>
<td>MULTI</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 12</td>
<td>ADD_PAR_LSW</td>
<td>MULTI</td>
<td>MULTI</td>
<td>ADD_PAR_LSW</td>
</tr>
<tr>
<td>TIME 13</td>
<td>ADD_PAR_MSW</td>
<td>MULTI</td>
<td>MULTI</td>
<td>ADD_PAR_LSW</td>
</tr>
<tr>
<td>TIME 14</td>
<td>LD_WEIGHT</td>
<td>ADD_PAR_MSW</td>
<td>ADD_PAR_LSW</td>
<td>MULTI</td>
</tr>
<tr>
<td>TIME 15</td>
<td>LD_INPUT</td>
<td>ADD_PAR_MSW</td>
<td>ADD_PAR_LSW</td>
<td>ADD_PAR_LSW</td>
</tr>
<tr>
<td>TIME 16</td>
<td>RESET_ACC</td>
<td>LD_INPUT</td>
<td>LD_WEIGHT</td>
<td>ADD_PAR_LSW</td>
</tr>
</tbody>
</table>

5.5.2 MIMD Program

The MIMD program is used to define each Virtual Neuron, and to implement a variety of other functions that may be required by the network. A few application examples are given at the end of the chapter to show the variety of techniques that can be incorporated into the MIMD program.

Additional to these functions the MIMD program must also synchronise and control each processor. The instructions used for these operations are given special consideration below:

• **LOCAL_OFF**

This instruction is used to indicate the start of a block of weights values. It has the effect of switching control from MIMD to SIMD for that individual processor. The following bytes in the weights table will then be treated as weights data for the next virtual neuron.
**ESCAPE**

This instruction is used to determine the end of each block of weights values, and results in a switch back to MIMD control. This instruction acts in a similar manner to a 'rogue value' at the end of a block of data. This rogue value is detected by special logic built into the weight's table bus controller. When this instruction is detected the processor stops executing the SIMD instruction stream, and uses the following bytes in the weights table as MIMD instructions.

**STOP**

This instruction is used to indicate that the processor has completed its virtual neurons for a network update. The processor will then wait until all the other processors have completed their virtual neurons, and the external input and output has been carried out, before commencing with the next network update.

The execution of each instruction in the MIMD program requires two cycles: a fetch and an execute cycle. Since both these cycles require access to the weights table it is not possible to overlap their execution, resulting in MIMD control not being as efficient as SIMD control. However since the time spent in MIMD control is small when compared to SIMD control, it was decided that this factor did not represent a serious drawback. It will be seen later in this chapter that the synchronisation required during MIMD control again impairs its efficiency, with NOP instructions being frequently inserted to achieve this. However it is felt that the minimisation of the hardware required in this synchronisation counteracts this reduced efficiency.

The following sections show how various programming techniques can be used, and give examples of various applications. After presenting these examples, three different implementations of the Tx Extraction Network are presented.
5.6 Application Notes and Examples

This section shows how this architecture can be used to implement a variety of networks. It starts by introducing a few techniques that can be used to implement various functions, and then shows the full implementations for some different applications. This includes fully connected networks, state feedback networks and conventional digital filters. (Both finite impulse response and infinite impulse response filters are considered.) Finally the full implementation of the Tx Extraction Network is considered. This implementation experiments with three different network mappings, showing how a network can be optimised for different criteria.

This section then concludes with a comparison between the performance of this architecture, and that of a typical SIMD array architecture. The three different Tx Extraction Network mappings are used in this comparison. These comparisons show the improved performance and flexibility of this architecture over a conventional SIMD array.

5.6.1 Programming Techniques

Before considering the full implementation of any network, a few programming techniques are presented. These techniques can be used in many ways to implement a wide variety of networks. The examples that follow develop these techniques further to demonstrate the flexibility that can be obtained from this architecture.

The techniques shown below are the definition of individual virtual neurons, the initialization and synchronisation between processors, the programming of an input window and the addition of the partial sums over several processors.

5.6.1.1 Definition of Virtual Neuron

The definition of each virtual neuron involves several functions. This includes the setting up of the input pointer, loading the neuron bias, updating each synapse, computing the sigma function and writing the neuron output back into the input table. This series of functions can
be represented as the algorithm below:

```
LOAD BIAS
LOAD INPUT POINTER
DO UNTIL ESCAPE
    UPDATE EACH SYNAPSE
ENDDO
SIGMA FUNCTION (partial_sum)
LOAD INPUT POINTER
WRITE NEURON OUTPUT
```

Listing 5.2 shows a virtual neuron with four synapses, this virtual neuron takes its inputs from Oh0100 - Oh0103 and the output is written to Oh0200.

<table>
<thead>
<tr>
<th>#</th>
<th>Virtual Neuron Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>Input Data Oh0100 - Oh0103</td>
</tr>
<tr>
<td>#</td>
<td>Output Oh0200</td>
</tr>
<tr>
<td>BIAS_LLSW</td>
<td>Oh53</td>
</tr>
<tr>
<td>BIAS_HLSW</td>
<td>Ohf7</td>
</tr>
<tr>
<td>BIAS_LMSW</td>
<td>Ohff</td>
</tr>
<tr>
<td>BIAS_HMSW</td>
<td>Ohf0</td>
</tr>
<tr>
<td>INPUT_PTR_L</td>
<td>Oh00</td>
</tr>
<tr>
<td>INPUT_PTR_H</td>
<td>Oh01</td>
</tr>
<tr>
<td>LOCAL_OFF</td>
<td>#End of Local Instructions</td>
</tr>
<tr>
<td>0h34</td>
<td>#Weight Values for Neuron</td>
</tr>
<tr>
<td>0h23</td>
<td></td>
</tr>
<tr>
<td>Ohf4</td>
<td></td>
</tr>
<tr>
<td>0h45</td>
<td></td>
</tr>
<tr>
<td>ESCAPE</td>
<td>#End of Weights</td>
</tr>
<tr>
<td>LD_SIGMA</td>
<td>#Calculate Neuron Output by Sigma Lookup Table</td>
</tr>
<tr>
<td>INPUT_PTR_L</td>
<td>Oh00</td>
</tr>
<tr>
<td>INPUT_PTR_H</td>
<td>Oh02</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>#Write Neuron Output into Input Table</td>
</tr>
</tbody>
</table>

Listing 5.2 Virtual Neuron Definition

The bus arbitration required in any section of MIMD code must be carefully implemented. Two instructions in this section of code require consideration:

**LD_SIGMA**

The **LD_SIGMA** instruction only has to be arbitrated with other processors running under MIMD control, since the **LD_SIGMA** instruction only occurs during MIMD control. To ensure that this instruction does not create any conflicts a simple convention is used:

The **LD_SIGMA** instruction should only be used in the instruction following the **ESCAPE** instruction.

This convention ensures that the **LD_SIGMA** arbitration follows the same scheme as used by the SIMD control. This arbitration scheme is shown in
Table 5.4. Here the start of MIMD control follows the pipelining of instructions across the array as seen under SIMD control, thus automatically pipelining the LD_SIGMA in a similar manner.

**OUTPUT**

The arbitration for the OUTPUT instruction has two cases that need consideration. These two cases depend upon whether all the processors are running under MIMD control, or whether some are under MIMD control and some are under SIMD control.

• *All Processors under MIMD Control*

When all the processors are running under MIMD control a similar convention to the LD_SIGMA arbitration can be used. This ensures that all OUTPUT instructions occur in the same position in each MIMD program. This scheme can be also seen in Table 5.4, where all the OUTPUT instructions are again distributed across the array, in a similar manner to the LD_SIGMA instructions. In this case the OUTPUT instruction is positioned four instructions after the ESCAPE instruction.

Table 5.4 Arbitration of LD_SIGMA Instructions

<table>
<thead>
<tr>
<th>TIME</th>
<th>PROCESSOR 1</th>
<th>PROCESSOR 2</th>
<th>PROCESSOR 3</th>
<th>PROCESSOR 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME 0</td>
<td>LD_WEIGHT</td>
<td>ADD_PAR_MSW</td>
<td>ADD_PAR_MSW</td>
<td>MULT_P</td>
</tr>
<tr>
<td>TIME 1</td>
<td>ESCAPE</td>
<td>LD_WEIGHT</td>
<td>ADD_PAR_MSW</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 2</td>
<td>FETCH</td>
<td>ESCAPE</td>
<td>LD_WEIGHT</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 3</td>
<td>LD_SIGMA</td>
<td>LD_SIGMA</td>
<td>LD_WEIGHT</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 4</td>
<td>FETCH</td>
<td>FETCH</td>
<td>ESCAPE</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 5</td>
<td>INPUT_PTR_L</td>
<td>INPUT_PTR_L</td>
<td>INPUT_PTR_L</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 6</td>
<td>FETCH</td>
<td>INPUT_PTR_H</td>
<td>INPUT_PTR_H</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 7</td>
<td>INPUT_PTR_H</td>
<td>FETCH</td>
<td>INPUT_PTR_H</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 8</td>
<td>FETCH</td>
<td>INPUT_PTR_H</td>
<td>INPUT_PTR_H</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 9</td>
<td>OUTPUT</td>
<td>FETCH</td>
<td>INPUT_PTR_H</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 10</td>
<td>.</td>
<td>OUTPUT</td>
<td>INPUT_PTR_H</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 11</td>
<td>.</td>
<td>.</td>
<td>INPUT_PTR_H</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 12</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 13</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

• *Some Processor Running under MIMD Control*

It is expected that some processors will operate under SIMD control, while others are under MIMD control. In this case, the
method of bus arbitration discussed above is no longer safe. The best method to use in this instance is to insert `NOP` instructions to delay the `OUTPUT` instruction. By calculating the number of `NOP` instructions that are required, the `OUTPUT` instruction can be executed during the same cycle that an `LD_INPUT` instruction would have been executed had the processor remained in SIMD control.

This operation is shown in Table 5.5. In this table Processor 2 and 4 are operating under MIMD control, while processor 1 and 3 are still under SIMD control. By inserting three `NOP` instructions before the `OUTPUT` instruction, processors 2 and 4 have become synchronised with the SIMD control, allowing the `OUTPUT` instruction to be executed without any conflicts.

Table 5.5 Execution of SIMD Program Across an Array

<table>
<thead>
<tr>
<th>TIME</th>
<th>PROCESSOR 1</th>
<th>PROCESSOR 2</th>
<th>PROCESSOR 3</th>
<th>PROCESSOR 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME 0</td>
<td><code>LD_WEIGHT</code></td>
<td><code>ADD_PAR_MSB</code></td>
<td><code>ADD_PAR_LSB</code></td>
<td><code>MULTI</code></td>
</tr>
<tr>
<td>TIME 1</td>
<td><code>LD_INPUT</code></td>
<td><code>ADD_PAR_LSB</code></td>
<td><code>ADD_PAR_MSB</code></td>
<td><code>ADD_PAR_LSB</code></td>
</tr>
<tr>
<td>TIME 2</td>
<td><code>RESET_ACC</code></td>
<td><code>LD_WEIGHT</code></td>
<td><code>ADD_PAR_MSW</code></td>
<td><code>ADD_PAR_MSW</code></td>
</tr>
<tr>
<td>TIME 3</td>
<td><code>MULTI</code></td>
<td><code>ESCAPE</code></td>
<td><code>LD_WEIGHT</code></td>
<td><code>ADD_PAR_MSW</code></td>
</tr>
<tr>
<td>TIME 4</td>
<td><code>MULTI</code></td>
<td><code>FETCH</code></td>
<td><code>LD_INPUT</code></td>
<td><code>ADD_PAR_MSW</code></td>
</tr>
<tr>
<td>TIME 5</td>
<td><code>MULTI</code></td>
<td><code>LD_SIGMA</code></td>
<td><code>RESET_ACC</code></td>
<td><code>ESCAPE</code></td>
</tr>
<tr>
<td>TIME 6</td>
<td><code>MULTI</code></td>
<td><code>FETCH</code></td>
<td><code>MULTI</code></td>
<td><code>FETCH</code></td>
</tr>
<tr>
<td>TIME 7</td>
<td><code>MULTI</code></td>
<td><code>INPUT_PTR_L</code></td>
<td><code>MULTI</code></td>
<td><code>LD_SIGMA</code></td>
</tr>
<tr>
<td>TIME 8</td>
<td><code>MULTI</code></td>
<td><code>FETCH</code></td>
<td><code>MULTI</code></td>
<td><code>INPUT_PTR_L</code></td>
</tr>
<tr>
<td>TIME 9</td>
<td><code>MULTI</code></td>
<td><code>INPUT_PTR_H</code></td>
<td><code>MULTI</code></td>
<td><code>FETCH</code></td>
</tr>
<tr>
<td>TIME 10</td>
<td><code>MULTI</code></td>
<td><code>NOP</code></td>
<td><code>MULTI</code></td>
<td><code>INPUT_PTR_H</code></td>
</tr>
<tr>
<td>TIME 11</td>
<td><code>MULTI</code></td>
<td><code>FETCH</code></td>
<td><code>MULTI</code></td>
<td><code>FETCH</code></td>
</tr>
<tr>
<td>TIME 12</td>
<td><code>ADD_PAR_LSB</code></td>
<td><code>NOP</code></td>
<td><code>MULTI</code></td>
<td><code>NOP</code></td>
</tr>
<tr>
<td>TIME 13</td>
<td><code>ADD_PAR_MSB</code></td>
<td><code>FETCH</code></td>
<td><code>MULTI</code></td>
<td><code>FETCH</code></td>
</tr>
<tr>
<td>TIME 14</td>
<td><code>LD_WEIGHT</code></td>
<td><code>NOP</code></td>
<td><code>ADD_PAR_LSB</code></td>
<td><code>NOP</code></td>
</tr>
<tr>
<td>TIME 15</td>
<td><code>LD_INPUT</code></td>
<td><code>FETCH</code></td>
<td><code>ADD_PAR_MSB</code></td>
<td><code>FETCH</code></td>
</tr>
<tr>
<td>TIME 16</td>
<td><code>RESET_ACC</code></td>
<td><code>OUTPUT</code></td>
<td><code>LD_WEIGHT</code></td>
<td><code>NOP</code></td>
</tr>
<tr>
<td>TIME 17</td>
<td><code>MULTI</code></td>
<td><code>FETCH</code></td>
<td><code>LD_INPUT</code></td>
<td><code>FETCH</code></td>
</tr>
<tr>
<td>TIME 18</td>
<td><code>MULTI</code></td>
<td><code>BIAS_LLSB</code></td>
<td><code>RESET_ACC</code></td>
<td><code>OUTPUT</code></td>
</tr>
<tr>
<td>TIME 19</td>
<td><code>MULTI</code></td>
<td><code>FETCH</code></td>
<td><code>MULTI</code></td>
<td><code>FETCH</code></td>
</tr>
</tbody>
</table>

The techniques presented above for processor synchronisation may appear to be rather contorted. However it should be noted that it is not expected for this to be done manually. A compiler would be developed that can manage these synchronisation functions, thus freeing the network developer of this task.
5.6.1.2 Initialisation and Synchronisation

The processor initialization is performed by a reset, which loads zero into all the registers and pointers. This action requires that the MIMD program should start at a base address of 0h0000 with the code required to initialise the first virtual neuron.

After this first virtual neuron has been initialized each processor is synchronised before commencing with the first network update. This synchronisation is performed by the STOP instruction, and by the externally generated start signal.

Upon completion of a network update each processor must be reset ready for the next network update. To perform this the RETURN instruction is used. This loads 0h0000 into the weights table pointer, thus starting the initialization sequence again. By repeating this process for each network update ensures that each processor remains synchronised with each other, and with the external input and output devices.

An example of this initialization and synchronisation, is shown in the Listing 5.3.

```
# Virtual Neuron Definition with Synchronisation

BIAS_LLSB  0h53   #Load Bias into Partial Sum (0hfffff753)
BIAS_HLSB  0hf7
BIAS_LMSB  0hff
BIAS_HMSB  0hff
INPUT_PTR_L 0h00   #Set Input Pointer for Virtual Neuron (0h0000)
INPUT_PTR_H 0h00
STOP        0h34   #Wait for Synchronisation from SIMD Control
               #Weight Values for Neuron
ESCAPE      #End of Weights
LD_SIGMA    #Calculate Neuron Output by Sigma Lookup Table
INPUT_PTR_L 0h00   #Set Input Pointer for Neuron's Output (0h0300)
INPUT_PTR_H 0h03
OUTPUT      #Write Neuron Output into Input Table
RETURN      #Return to Start of MIMD Program
```

Listing 5.3 Initialization and Synchronisation
5.6.1.3 Implementation of Input Window

The use of an input window to represent the input layer of a signal processing network was presented in Chapter 3, with two different approaches given to its implementation. (The method used in the TMS320 required specialised memory, while the Motorola DSP56000 used a dedicated address generation unit.) In this architecture an approach similar to the Motorola DSP56000 is used, with the Modulo Address Generator performing the necessary functions.

To make use of this Modulo Address Generator some specialised programming techniques are required. Before describing these, the general concept of how the input window is to be implemented will be covered.

Diagram 5.3 Input Window Implementation

Diagram 5.3 shows how the input window for the Tx Extraction Network would be implemented. This input window uses forty one time slices, each composed of six frequency bands. These samples are labelled as below:
At the start of each network update the input pointer should be pointing at the sample representing \( t-40, f_6 \), since this is the first input value to be processed. As the \( 1^{st} \) hidden layer is processed the input pointer will increment through this input window, and upon completion will point to the memory location just after sample \( t_0, f_1 \).

Before the next network update can commence this input pointer has to be moved back within the input window. This requires an offset to be added to the input pointer, so that the pointer is set to the sample representing \( t-39, f_6 \). The reason for the pointer returning to this sample is that during the next network update this sample will represent \( t-40, f_6 \).

This action is performed by the \texttt{ADD\_MOD\_STEP} instruction. This instruction takes the offset, which should be loaded over the previous instructions, and adds it to the input pointer. The value used for the \texttt{mod\_step} should be calculated as below:

\[
((\text{No. Times Slices} - 1) \times (\text{No. Samples in each Time Slice}))
\]

The effect of this action is that the input window will slowly move through the input table as each network iteration occurs. The use of the modulo address generation unit limits this movement to a modulo sized buffer within the input table.

One last function is required to implement fully this input window. This is required since the modulo address generation is only necessary for the \( 1^{st} \) hidden layer, while the remaining layers require fixed locations for their input and outputs. To enable this a software switch is used to turn the modulo address generation on and off as required. This switch not only selects between modulo address generation and linear address generation, but also swaps between two different input table pointers registers. These two registers are termed:

\begin{center}
Linear Input Pointer Register
\end{center}
Modulo Input Pointer Register

The reason that two registers are required is so that the current position of the input window, which will be held in the modulo input pointer register, is not overwritten when the following layers are updated.

Listing 5.4 shows the implementation of the input window that is used in the Tx Extraction Network.

```
# TX Extraction Network Description
# Processor 1 Weight Table
# First Hidden Layer Neuron 1
# Input Data from Input Window
# Output 0h0100
BIAS_LLSB 0ba3  # Load Bias into Partial Sum (0h000000a3)
BIAS_HLSB 0h00
BIAS_LMBS 0h00
BIAS_HMSB 0h00
MODULO_ON 0h00
STOP 0h02
0h32
0hf3
ESCAPE  # End of Weights
LD_SIGMA  # Calculate Neuron Output by Sigma Lookup Table
MODULO_OFF  # Turn Modulo Address Generation Off for the Remaining Layers
INPUT_PTR_L 0h00  # Set Input Pointer for Neuron's Output (0h0100)
INPUT_PTR_H 0h01
OUTPUT  # Write Neuron's Output into Input Table
INPUT_PTR_L 0h10  # Load Modulo Step into Input Pointer (0hff10 or -240 decimal)
INPUT_PTR_H 0hff
ADD_MOD_STEP  # Move Modulo Input Pointer to Start of Input Window
```

Listing 5.4    Input Window Implementation

5.6.1.4 Communication of Partial Sums

The communications of partial sums between two processors require special consideration, primarily due to the synchronisation that is required to achieve this. To synchronise this transfer, the two processor must transmit and receive the data during the same clock cycles; NOP instructions are again used to achieve this.

The listings of two processors that transfer a partial sum between them is given in Listing 5.5. Here the first processor has a NOP instruction bringing it into synchronisation with the second processor.
The transfer of the partial sum relies upon the transmitting processor executing the `LD_SIGMA` instruction. This instruction puts the 32 bit partial sum onto the Sigma Bus, where it can be read by the receiving processor. To allow the partial sum to be added into the receiving processor's partial sum, requires that this instruction is repeated twice, with the receiving processor adding the LSWord during the first cycle, and the MSWord during the second cycle. Two different instructions are used to distinguish between these cycles on the receiving processor: `LD_PAR_LSW` and `LD_PAR_MSW`.

The execution sequence in Table 5.6 shows how four processors can be synchronised. In this table the partial sum from processor 3 is transferred to processor 1, and from processor 4 to processor 2.

```
#Processor to Read Partial Sum

0h32
0hf3
ESCAPE #End of Weights
NOP #Synchronise Processors
LD_PAR_LSW #Add Partial Sum LSWord
LD_PAR_MSW #Add Partial Sum MSWord

#Processor to Output Partial Sum

0h8A
0ha5
ESCAPE
LD_SIGMA #Output Partial Sum
LD_SIGMA #Output Partial Sum
```

Listing 5.5 Partial Sum Communications

There is one small difficulty in achieving this inter-processor synchronisation for partial sum communications. This is due to each MIMD instruction requiring two cycles for its execution. The effect of this is that each NOP instruction inserts a two cycle delay, resulting in the first processor in the array becoming synchronised with the third processor. In fact any odd numbered processor can become synchronised with any other odd numbered processor, but it is impossible to synchronise between an odd and an even processor. This limitation is only related to partial sum communications, and as demonstrated in the following examples, it has very little effect on the flexibility of the network mappings.
Table 5.6 Synchronisation of Processors

<table>
<thead>
<tr>
<th>TIME</th>
<th>PROCESSOR 1</th>
<th>PROCESSOR 2</th>
<th>PROCESSOR 3</th>
<th>PROCESSOR 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME 0</td>
<td>LD_WEIGHT</td>
<td>ADD_PAR_MSW</td>
<td>ADD_PAR_LSW</td>
<td>MULTT</td>
</tr>
<tr>
<td>TIME 1</td>
<td>ESCAPE</td>
<td>LD_WEIGHT</td>
<td>ADD_PAR_MSW</td>
<td>ADD_PAR_LSW</td>
</tr>
<tr>
<td>TIME 2</td>
<td>FETCH</td>
<td>ESCAPE</td>
<td>LD_WEIGHT</td>
<td>ADD_PAR_MSW</td>
</tr>
<tr>
<td>TIME 3</td>
<td>NOP</td>
<td>FETCH</td>
<td>ESCAPE</td>
<td>LD_WEIGHT</td>
</tr>
<tr>
<td>TIME 4</td>
<td>FETCH</td>
<td>NOP</td>
<td>FETCH</td>
<td>ESCAPE</td>
</tr>
<tr>
<td>TIME 5</td>
<td>LD_PAR_LSW</td>
<td>FETCH</td>
<td>LD_SIGMA</td>
<td>FETCH</td>
</tr>
<tr>
<td>TIME 6</td>
<td>FETCH</td>
<td>LD_PAR_LSW</td>
<td>FETCH</td>
<td>LD_SIGMA</td>
</tr>
<tr>
<td>TIME 7</td>
<td>LD_PAR_MSW</td>
<td>FETCH</td>
<td>LD_SIGMA</td>
<td>FETCH</td>
</tr>
<tr>
<td>TIME 8</td>
<td>.</td>
<td>LD_PAR_MSW</td>
<td>.</td>
<td>LD_SIGMA</td>
</tr>
<tr>
<td>TIME 9</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

5.6.2 Application Examples

This section will take several different applications, and show how they would be programmed on this architecture. The examples are chosen as they represent the far ends of the spectrum of the different models that can be implemented by this architecture.

The first of these examples is a fully connected Hopfield style network, and the second is a state feedback network. The remaining two examples show how this architecture can be used to implement conventional digital processing models. Two different filters are shown, finite impulse response (FIR) and infinite impulse response (IIR).

5.6.2.1 Fully Connected Network

Fully connected networks represent a different approach to the use of neural networks than those considered in Chapter 2 and Chapter 3. These networks can be used as associative memories, that is they can recall patterns from partial inputs. These networks are primarily used in visual, and other pattern recognition applications, for pre-processing.

They are constructed from a single layer of neurons, with each output being used to provide full feedback to the input. A diagram of a three neuron, fully connected network is shown in Diagram 5.4.

The operation of this type of network is usually asynchronous in nature,
analogue techniques are commonly used to implement these. However this architecture is able to implement these networks, although it does this in a synchronous manner. Listing 5.6 shows how the network in Diagram 5.4 would be implemented.

Listing 5.6 Fully Connected Network

```
# Fully Connected Network
# Definition for First Neuron
# Input Data 0b0000 and 0b0100 - 0b0102
# Output 0b0100
BIAS_LLSB 0b00 #Reset Partial Sum (0b00000000)
BIAS_HLSB 0b00
BIAS_LMSB 0b00
BIAS_HMSB 0b00
INPUT_PTR_L 0b00 #Set Input Pointer for Input Value (0b0000)
INPUT_PTR_H 0b00
STOP 0b34 #Wait for Synchronisation
ESCAPE
INPUT_PTR_L 0b00 #Set Input Pointer for Feedback Values (0b0100)
INPUT_PTR_H 0b01
LOCAL_OFF #End of MIMD Instructions
Oh85
Ohf4
Ohf6
ESCAPE #End of Weights
LD_SIGMA #Calculate Neuron Output by Sigma Lookup Table
INPUT_PTR_L 0b00 #Set Input Pointer for Neuron's Output (0b0100)
INPUT_PTR_H 0b01
OUTPUT #Write Neuron Output into Input Table
RETURN
```
A single neuron (N:1) is shown, this takes its input from Oh0000, and used the feedback values from Oh0100 - Oh0102. The output from this neuron is written to Oh0100.

5.6.2.2 State Feedback Network

The use of state feedback networks was presented in Chapter 3. This technique enables a few output units to be feedback to the input layer, so providing some means of holding the previous state of the network. Diagram 5.5 shows a state feedback network, with two output units providing feedback to the input layer. In this network a two cycle delay is imposed on the feedback.

**Diagram 5.5 State Feedback Network**

Listing 5.7 shows how this network would be implemented. This listing shows one of the neurons (N:1), and one of the delay units (D:1,1). The important technique used in this implementation is the delay unit. This is implemented by a single input virtual neuron, with a weight value of
1. The only special technique used here is that the partial sum is used as the output, with the sigma function not being performed. This will have the effect of passing the input to the output unchanged.

Listing 5.7  State Feedback Network

5.6.2.3 Finite Impulse Response Filter

A simple model of a finite impulse response filter can be shown diagrammatically in Diagram 5.6[41]. Here there are four time samples (t0, t-1..), each weighted by a constant (h0, h-1 ..). The filter output is represented by the sum of these products (Yt).

It is easy to see how such a filter can be implemented on this architecture. The filter operates in the same way as a neuron does, except that no sigma function is used on the output. Listing 5.8 shows
Diagram 5.6  Finite Impulse Response Filter

how this filter would be implemented.

Listing 5.8  Finite Impulse Response Filter

5.6.2.4  Infinite Impulse Response Filter

To provide an infinite impulse response filter requires the use of some feedback, this allows the filter to hold a representation of the previous inputs. This is provided by taking a function of the output, and providing this as an additional input. This is shown in Diagram
5.7.

The implementation of this type of filter is again possible on this architecture. The interesting point to note is how the input to the filter must swap between using modulo address generation, and using linear address generation. This is required as the input samples will require an input window, while the feedback value will require a fixed location. This implementation is shown in Listing 5.9.

```
# Infinite Impulse Response Filter
# Input from Input Window and OhOl00
# Output OhOl00
BIAS_LLSB 0h00   # Reset Partial Sum (0h00000000)
BIAS_HLSB 0h00
BIAS_LMSB 0h00
BIAS_HMSB 0h00
MODULO_ON 0h00   # Select Modulo Address for Input Samples
LOCAL_OFF 0h31   # Filter Constants
Oh8c
Oh8d
ESCAPE
INPUT_PTR_L 0hff  # Load Mod Step (0hffffff or -3 decimal)
INPUT_PTR_H 0hff
ADD_MOD_STEP 0h00 # Move back to Start of Input Window
MODULO_OFF 0h82   # Turn Modulo Off for Feedback Value
INPUT_PTR_L 0h00  # Set Pointer for Filter Output (0h0100)
INPUT_PTR_H 0h01
LOCAL_OFF 0h82
ESCAPE
INPUT_PTR_L 0h00  # Set Pointer for Filter Output (0h0100)
INPUT_PTR_H 0h01
OUTPUT 0h00
RETURN
```

Listing 5.9 Infinite Impulse Response Filter

5.6.3 Tx Extraction Implementation

This section will take the Tx Extraction Network that was presented in Chapter 3, and show how this architecture could be used to implement it. There are three different implementations given, this shows how the network mapping can be varied to achieve different objectives. These three examples are for silicon efficiency, reduced sample period and reduced response time.

This section uses the same technique that was used in Chapter 4 to represent the network mapping. It should be stressed that these measurements are only a first order approximation. The actual efficiency of the
Diagram 5.7  **Infinite Impulse Response Filter**

architecture must take into account the proportion of time spent in SIMD and MIMD control, this is dealt with in more detail in Section 6.10.1.

At the end of this section a comparison between each of these mappings can be made to the SIMD implementation shown in Chapter 4. These comparisons are then used to form some conclusions on the success of this architecture.

5.6.3.1 Silicon Efficiency

In this network mapping six processors are used in a non-pipelined manner. This mapping results in a trade-off between the slightly longer sample period (258 time units) and the reduced number of processors that are used (6 processors). However the response time is fairly low (258 time units) due to the use of a non-pipelined mapping. This mapping can be summarised in Table 5.7.
Table 5.7 Silicon Efficient

<table>
<thead>
<tr>
<th>Mapping</th>
<th>1st Hidden Layer</th>
<th>2nd Hidden Layer</th>
<th>Output Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration</td>
<td>246</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

5.6.3.2 Reduced Sample Period

In this arrangement 13 processors are used, these are arranged to optimise the sample period. This is achieved by scheduling each 1st hidden layer neuron across two processors (requiring 12 processors and 123 time units). The 2nd hidden layer and output layer can all be implemented by the 13th processor (requiring 42 time units). This arrangement requires a two stage pipeline for each network update, with the processing of the 1st Hidden layer being overlapped with that of the 2nd Hidden Layer and Output Layer. By using this arrangement the sample period is considerably reduced (123 time units) with the response time remaining fairly slow (246 time units). This mapping is summarised in Table 5.8.

Table 5.8 Reduced Sample Period

<table>
<thead>
<tr>
<th>Mapping</th>
<th>1st Hidden Layer</th>
<th>2nd Hidden Layer</th>
<th>Output Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration 1</td>
<td>12 PEs</td>
<td>. . . 1 PE . . .</td>
<td>6</td>
</tr>
<tr>
<td>Iteration 2</td>
<td>123</td>
<td>36 + 6 + 81 Idle</td>
<td></td>
</tr>
</tbody>
</table>

5.6.3.3 Reduced Response Time

In this arrangement 12 processors are used in a non-pipelined manner. Each neuron in the 1st and 2nd hidden layers will be scheduled across 2 processors, so halving the time for each neuron (123 time units). The output layer is then implemented by 6 processors, this reduces the time for this layer to 1 time unit. Because this mapping is implemented
sequentially the response time is considerably reduced (127 time units), as is the sample period (127 time units). Table 5.9 shows this mapping.

<table>
<thead>
<tr>
<th>Mapping</th>
<th>1st Hidden Layer</th>
<th>2nd Hidden Layer</th>
<th>Output Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration</td>
<td>123</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

5.6.4 Summary of Implementation Examples

The application examples given in this last section has shown a wide range of possibilities for this architecture. The ability for this architecture to implement different networks, and even convention signal processing models, has highlighted this flexibility.

The three examples used in the different mappings of the Tx Period Extraction Network were used to demonstrate the variations obtained using a flexible mapping of a network onto an array. These examples used a range of different mappings, to obtain the required real-time performance, or to minimise the hardware.

To be able to form a more solid conclusion, some comparisons are made between this architecture, and a conventional SIMD array as presented in Chapter 4. To form a comparison several measures are used. These measures were chosen as they show the performance, efficiency and cost of each implementation. The measured used were as below:

• No. Processors
  The number of processors required by an implementation is used as a measure of the silicon cost, and also can be used to represent the power consumption of each implementation.

• Sample Period
  This represents the pipeline iteration time. This period determines the speed that input can be accepted by the network.
• Response Time
This represents the time taken for a change at the input to affect the outputs. This measure is calculated by taking the product of the pipeline iteration time and the number of pipeline stages.

• Efficiency
This is a percentage of the number of active synaptic update time units to the total number of synaptic update time units taken by an implementation.

Table 5.9 summarises these measures for all the implementations shown. This summary also includes the SIMD implementation given in Chapter 4 so that it can be used for comparison.

Table 5.9 Summary of Solutions

<table>
<thead>
<tr>
<th></th>
<th>SIMD Array</th>
<th>Silicon Efficient</th>
<th>Reduced Sample</th>
<th>Reduced Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. Processors</td>
<td>13</td>
<td>6</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>Sample Period</td>
<td>246</td>
<td>258</td>
<td>123</td>
<td>127</td>
</tr>
<tr>
<td>Response Time</td>
<td>738</td>
<td>258</td>
<td>246</td>
<td>127</td>
</tr>
<tr>
<td>Efficiency</td>
<td>47.5%</td>
<td>98%</td>
<td>95%</td>
<td>99.6%</td>
</tr>
</tbody>
</table>

From these examples it is now possible to formulate some conclusions on how successful this architecture has been in achieving its objectives. Before doing this it is worth considering the beneficial properties from MIMD arrays that this architecture attempted to capture.

• High Flexibility
The examples shown above indicate the flexibility this architecture offers. The range of different networks, and even the ability to implement conventional digital signalling models, has highlighted this feature.

• Software Programmable
The fact that all the above implementations were defined by the MIMD programs shows that this architecture is fully software programmable.
• High Efficiency
The efficiency figures given in Table 5.9 show the considerably higher efficiency of this architecture over conventional SIMD array techniques. With figures of well over 90% shows this architecture has been able to make full use of its potential processing power.

To be able to state whether this architecture has achieved the remaining objectives requires an in depth look at an implementation using this architecture. The next chapter covers such an implementation, and gives the details of a CMOS device, and discusses future developments for this architecture.
Chapter 6

Detailed Description of Design

An architecture has been presented that attempts to achieve high performance in small-scale neural implementations. This architecture has been developed to combine beneficial properties from both MIMD and SIMD arrays.

A series of application examples have been presented to demonstrate the flexibility and the performance efficiency that this architecture can achieve. The conclusions from this are that the architecture presented does implement the desirable properties of MIMD architectures that it was attempting to achieve. The purpose of this chapter is to further illustrate the architecture in order to assess how successful it has been in capturing the desirable properties of SIMD arrays as outlined in Chapter 4.

6.1 Design of CMOS Implementation

The most important property that this architecture is attempting to capture from SIMD arrays is low complexity. This factor would ensure both a low silicon area and low power consumption: key requirements in any small-scale portable device.

To ensure that a low complexity is achieved, a full custom layout was decided upon. The tool used for this custom design was MAGIC. This VLSI layout tool comes with the University of California at Berkeley VLSI Design Tools, and was the only VLSI layout tool available at UCL at the time of development. (For a full description of the design environment, and the interaction between the different tools used, the reader should consult Section 6.10.)

The decision to use a full custom layout tool presented the daunting task of designing a full datapath and control logic. This chapter will discuss the design of these sections, and the steps involved in the development of a device using the presented architecture. This discussion will follow a similar order to the chronological events that
The first section to be developed was the datapath. This involved first finding suitable algorithms with which to implement the arithmetic functions, and then building the required functional units to implement these algorithms.

Once the datapath was developed, and the arithmetic algorithms had been fully tested on the datapath, the instruction decoding logic could be developed. Following from this the instruction selection hardware was built, and fully tested.

The design of an individual processor was then fully complete, allowing for the development of the SIMD control block to be carried out, and remaining bus control functions to be implemented. The final simulations were then performed; this took the form of a full gate level simulation of two processors, each with its own weights table and a shared input and sigma function look-up table.

6.2 Algorithms used for the Arithmetic Functions

Before considering the algorithms that were developed, the representation of the different data values had to be decided upon. In this design there are four basic data types:

- **Input Value and Weight Value**
  During the development of the Tx Extraction Network it was decided to use an 8 bit 2's complement integer to represent the weights and the input values. Simulations had been carried out on various data lengths, concluding that 8 bits provided a safe margin of accuracy in these models for the recognition phase.

- **Address Pointers**
  The size of the address pointers has a direct effect on the maximum size of network that can be implemented. The size was chosen to be 16 bits, this would provide an adequate address range for most networks, and provide a convenient size for the storage of the pointers.
**Synaptic Product**
The size of this value is dependent upon the length of both the input and weight values. In this implementation 8 bits are used to represent both these values, resulting in the synaptic product requiring 16 bits for its representation.

**Partial Sum**
The size of the partial sum is again fairly arbitrary. All that is required is that it should be large enough to ensure that overflows do not become too frequent. The size chosen for this value was 32 bits, this again represented a convenient length for its storage.

After considering the different data representations it was decided to use a 16 bit datapath, with special instructions being included to process the inputs and weights, and for the double length partial sum. The size of the datapath was chosen primarily to ensure that each pointer could be updated in a single clock cycle: this is important since pointer manipulations form a large role in the processor's operations.

The arithmetic operations to be performed on the datapath were then considered. These operations were broken-down into three groups, and considered separately.

6.2.1 Pointer Manipulations

One pointer manipulation is the increment of the relevant pointer during each memory fetch. This requires a conventional adder unit, with the ability to set one of the inputs to zero, and the carry_in high. The only other pointer manipulation function is the ADD_MOD_STEP instruction, which requires the use of a conventional adder with no additional facilities.

6.2.2 Partial Sum Addition

The partial sum additions require additional control and selection logic
on the datapath to cater for the different length operands: the synaptic product being 16 bits wide, while the partial sum is 32 bits wide.

As mentioned previously any operation that is carried out on the partial sum is performed over a double length instruction. Between these two instructions the carry generated during the first addition instruction must be carried into the second addition, and the sign of the synaptic product has also to be carried through to an operand in the second addition. The sign extension is needed to convert the 16 bit synaptic product into a 32 bit value. This requires either a 0 extension (0h0000) for a positive synaptic product, or -1 extension (0hffff) for a negative synaptic product. The algorithm used for this addition of the synaptic product to the partial sum can be described thus:

\[
\text{LSWord Partial Sum} := \text{LSWord Partial Sum} + \text{Synaptic Product} \\
\text{IF Sign of Synaptic Product is +ve} \\
\quad \text{MSWord Partial Sum} := \text{MSWord Partial Sum} + 0h0000 + \text{Carry} \\
\text{ELSE} \\
\quad \text{MSWord Partial Sum} := \text{MSWord Partial Sum} + 0hffff + \text{Carry}
\]

The other complication in this partial sum addition is the requirement for saturated arithmetic, this ensures that the sign does not alter if an overflow occurs. Saturated arithmetic operates during the second addition cycle, and requires special logic to be placed on the output of the addition unit. This logic performs the following algorithm:

\[
\text{If No Overflow} \\
\quad \text{MSWord Partial Sum} := \text{Output of Adder} \\
\text{If Overflow from +ve to -ve} \\
\quad \text{MSWord Partial Sum} := 0h7fff \\
\text{If Overflow from -ve to +ve} \\
\quad \text{MSWord Partial Sum} := 0h8000
\]

6.2.3 Multiplication Instructions

The multiplication algorithm requires special consideration since it represents the main function for each synaptic update. To ensure maximum throughput, the algorithm should not require any more cycles than the
minimum required for the loading of the operands and in the computing of the product. This factor plays heavily in the choice of the algorithm since it must cater for:

- Unsigned Multiplication (+ve x +ve)
- Mixed Multiplication (+ve x -ve and -ve x +ve)
- Signed Multiplication (-ve x -ve)

The algorithm chosen was taken from the Texas Instruments 74888/74890 Bit Slice Processor[1]. This algorithm can perform all the above multiplication types, with no time overheads in sign correction after the product has been computed.

The algorithm requires three registers for the multiplication, two of these being general purpose registers (acc and reg_b), and the third being a special shift register (reg_sh). Before a multiplication can start, the registers should be loaded as below:

- acc zero
- reg_b multiplicand
- reg_sh multiplier

The multiplicand holds the negative integer during mixed and signed multiplication, allowing it to be adjusted during the last multiplication iteration if required. This operation is determined by the 'Mode Bit' in the multiplication algorithm. To ensure that the operands are loaded into the correct registers the following algorithm is used:

1. Load Weight into reg_b and reg_sh
2. If Sign of Weight is -ve
   1. Load Input into reg_b
3. Else
   1. Load Input into reg_sh

This algorithm is implemented in hardware, thus reducing the number of instruction cycles required to load the operands.
Each multiplication requires N iterations to provide the 2N bit result (where N = size of operands). Upon completion the result is held in the registers as below:

- acc: MSByte of product
- reg_sh: LSByte of product
- reg_b: Unchanged

The multiplication operation can be expressed as the following recursion; this recursion repeats for J = 0 to N. Iterations J = 0 to N-1 are termed iterative cycles (MULTI), while iteration J = N is termed the terminating cycle (MULTT):

\[ P^\prime_{J+1} = 2^J(P^\prime_J + M(Multiplicand \times Multiplier[J])) \]

Where:
- \( P^\prime \) = Partial Product for Jth Iteration
- J = Iteration Number [Repeat for J = 0 to N]
- \([J]\) = Bit at position J
- 2 = Right Shift
- [Varies according to iteration type and sign of operands]
- M = Mode Bit
- [Varies according to iteration type]

• **Right Shift**

The right shift operates as a double length shift on the partial product. The type of shift mode used varies according to the signs of the operands, and whether it is an iterative cycle or a terminating cycle. A selection mechanism switches between an arithmetic shift, which shifts in the MSBit of the partial sum, and a logical shift which uses the carry out from the adder unit. The selection between these different shift modes is shown in Table 6.1.

• **Mode Bit**

The mode bit is used to perform the sign adjustment on the multiplicand, and is dependent only upon the type of multiplication iteration. During an iterative cycle this mode bit
is set to 1, allowing the multiplicand to be added to the partial product unaffected. During the terminating cycle this mode bit is set to -1, selecting the 2's complement of the multiplicand for the addition. This 2's complementation is performed by the negation of the multiplicand, and the setting of the carry in for the adder to high.

One last point to mention in this algorithm is the problem encountered during the simulations of large positive and negative integers. It was found that the sign of the final product became corrupt when multiplying large negative integers. To counter this problem the use of a sign guard bit was introduced; this requires the extension of both the multiplier and the multiplicand from 8 bits to 9 bits. This is performed by copying the MSBit from each operand into the sign guard bit. The multiplication is then performed on the extended 9 bit multiplier and multiplicand, resulting in a 18 bit product. The top two bits of this 18 bit product represent two sign guard bits, and can be truncated. The 16 bit product obtained after this truncation can then be used to represent the synaptic product.

6.3 Datapath Construction

The datapath was constructed using conventional bit-slice techniques, and was built to incorporate the functional units required to implement the algorithms developed above. Diagram 6.1 shows the organisation of the datapath and how the data flows through the various functional units.
Diagram 6.1  Datapath Schematic

The main functional sections of this datapath are:

*Bus A and Bus B*
Two sixteen bit wide buses connected to all the functional units, and the external bus controllers.

*Arithmetic Unit*
Full 16 bit adder, with input conditioning for each operand. The input selection can select between four different conditions, these are the current bus value, its inverse, $0h0000$ and $0hffff$.

*Saturated Arithmetic*
This provides saturated output control on the result from the arithmetic unit.

*Bus Shift and Shift Register*
These two functional units occupy only half the width of the datapath, and are used to provide a double length shift for the multiplications.

*Register Block and Accumulator*
These two blocks provide an accumulator, and six general purpose dual-ported registers.

A key point to note on this diagram is how the functional units that only occupy the lower half of the datapath operate. These functions are
used in the multiplications and consist of the bus shift, the shift register, and the 9 LSBits of the accumulator. Upon completion of each multiplication the 18 bit product is held with the 9 MSBits in the lower half of the accumulator, and the 9 LSBits in the shift register.

To output this 18 bit product as a 16 bit synaptic product requires the use of the vertical bus. This bus connects the lower 7 bits of the accumulator to the upper 7 bits of bus B. This bus operates simultaneously with the shift register outputting the lower half of the product to the 9 LSBits of bus B. This transfer has the effect truncating the 18 bit product to a 16 bit synaptic product, with the top two sign guard bits from the accumulator being ignored. This synaptic product can then be added into the lower word of the partial sum. The whole of the above operation is performed in a single clock cycle since this vertical bus performs all the necessary data truncations and transfers between the lower and upper halves of the datapath.

6.3.1 Timing Strategy

Each operation executed on the datapath is completed in a single clock cycle, after which all registers, and status registers, hold their contents in static latches. This clocking scheme allows the device to be single stepped during testing and debugging.

Diagram 6.2 shows the timing strategy used throughout the design. This strategy involves a conventional two phase non-overlapping clock, with an additional phase. This extra phase is required to allow for the hold time of the static latches, and is similar to $\phi_2$ but with a delay being added to the falling edge. Both $\phi_1$ and $\phi_2$ are externally generated, while $\phi_3$ is generated internally.

6.3.2 Arithmetic Unit

The arithmetic unit consists of a 16 bit parallel adder with conditioning on both inputs. The carry is propagated through the adder without any carry lookahead. This was decided to simplify the design, and because the processor would be memory bound in its operation due to there being no on-chip RAM.
Diagram 6.2 Clocking Strategy

The input selection to the arithmetic unit allows four possible conditions to be selected. These conditions are required by the arithmetic operations discussed in the previous sections. Table 6.2 shows these four conditions, and the signals that control them.

Table 6.2 Adder Input Selection

<table>
<thead>
<tr>
<th>02a &amp; 02b</th>
<th>Inva &amp; Invb</th>
<th>Input to Adder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>Low</td>
<td>Value on Bus</td>
</tr>
<tr>
<td>Low</td>
<td>High</td>
<td>Inverse of Bus</td>
</tr>
<tr>
<td>High</td>
<td>Low</td>
<td>0h0000</td>
</tr>
<tr>
<td>High</td>
<td>High</td>
<td>0hffff</td>
</tr>
</tbody>
</table>

The arithmetic unit and input selection is shown in Diagram 6.3. This diagram shows the logic used in the input selection, and how the clocking is performed. During $\phi_1$ the operands are clocked from left to right into the arithmetic unit, while the outputs are clocked from right to left during $\phi_3$.

6.3.3 Saturated Addition

To perform saturated addition, without any time overheads, relies upon specialised logic to be added to the output of the arithmetic unit. This logic is controlled by the states of the carry in and carry out from the MSBit in the arithmetic unit. An additional signal is also provided from
the instruction decoder to select whether saturated addition is required: saturated arithmetic is only used during the second cycle of partial sum additions.

The status of the carry in (msb_c_in) and carry out (msb_c_out) from the MSBit are used to select one of three possible outputs, these are shown in Table 6.3. The logic used to evaluate these conditions is designed to operate very quickly, this is required since its output must be evaluated and established before \( \Phi_3 \) becomes high.

Diagram 6.4 shows the layout of the saturated addition, and how it fits into the datapath. This saturated arithmetic control only operates during \( \Phi_3 \). This allows the operands to pass unaffected from left to right during \( \Phi_1 \), and to only be subject to control during \( \Phi_3 \).

Table 6.3 Saturated Addition Control

<table>
<thead>
<tr>
<th>msb_c_out</th>
<th>msb_c_in</th>
<th>Saturated Addition Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>Low</td>
<td>Sum from Arithmetic Unit</td>
</tr>
<tr>
<td>Low</td>
<td>High</td>
<td>Highest Positive</td>
</tr>
<tr>
<td>High</td>
<td>Low</td>
<td>Lowest Negative</td>
</tr>
<tr>
<td>High</td>
<td>High</td>
<td>Sum from Arithmetic Unit</td>
</tr>
</tbody>
</table>
6.3.4 Double Length Shift

The double length shift is provided by the bus shift and the shift register (reg_sh). These are both positioned to the left of the arithmetic unit in the datapath.

The input to the MSBit of the bus shift is determined by the multiplication algorithm, and is selected by the shift multiplexer, the control for which comes from the instruction decoder. Between these two shift units a link is provided, so that a double-length shift can occur, with the LSBit from the arithmetic unit being used as an input to the shift register (reg_sh).

Diagram 6.5 shows the double length shift, and how the data is clocked through it. During $1$ the operands for the arithmetic unit pass unaffected on Bus A and Bus B from left to right. Meanwhile the contents from the shift register (reg_sh) are output via Q bar, and shifted one
place to the right. The value shifted into the shift register (reg_sh) is latched during $\phi 1$ with the LSBit of the sum; this operation is possible since this value becomes stable long before the end of $\phi 1$.

Diagram 6.5 Double Length Shift

During $\phi 2$ the bus shift operates, resulting in the output from the arithmetic unit being shifted one place to the right.

6.3.5 Execution of a Single Multiply Iteration

To help clarify how the datapath works, and how the various units fit together, Diagram 6.6 and Diagram 6.7 show the data flow for a single multiply iteration. The bold lines in these diagrams show which signals are active during each part of the multiplication operation.

The breakdown of the operations carried out during the two clock phases is as below:
Operations during $\phi 1$
During $\phi 1$ the operands from the accumulator (acc) and from reg_b are output to the Bus A and Bus B respectively. Conditional summation then occurs, with the LSBit of the shift register (reg_sh) acting as the condition flag. Simultaneously with this operation the shift register (reg_sh) is shifted to the right, with the LSBit from the arithmetic unit being shifted into the MSBit.

Operations during $\phi 2$
During $\phi 2$ the result of the conditional summation is made available. This is shifted right, with the selected value from the shift multiplexer being used as the input to the MSBit. The resulting value on Bus A is then latched into the accumulator (acc). Simultaneously with this operation the now shifted value for the shift register (reg_sh) is also latched.

6.3.6 Simulation of Datapath

After the completion of the datapath layout, a full simulation was performed. This simulation was performed using an extracted netlist from the physical layout. This simulation allowed a very accurate analysis of the design to be obtained. Not only could the algorithms be tested, but different configurations in the datapath could be compared. Using this technique several different layout styles were tested, with the most efficient in terms of speed, and silicon area, being chosen.

To carry out this simulation required the generation of extensive test vectors, these were used to simulate the control signals that would eventually be generated by the instruction decoder. To produce these test vectors a microassembler was used, allowing small programs to be written.

These simulations were all performed using RNL. RNL is the switch level simulator available at UCL, and comes with the Northwest Laboratory for Integrated Systems at University of Washington Tool Set. (Section 6.10 discusses the model used by this simulator, and how it fits into the
Diagram 6.6  Multiplication Flow for φ1
Diagram 6.7  Multiplication Flow for $\Phi^2$
6.3.7  Datapath Performance

The simulations carried out on the datapath enabled an estimate to be made to its maximum clocking speed. This estimate considers the slowest period required for each clock phase, and the minimum non-overlap time.

**Minimum $\phi_1$ Period**

The minimum period for $\phi_1$ is determined by the time for an addition with a carry propagation through the entire length of the addition unit. The time required for this is: 33.9 ns

**Minimum $\phi_2$ Period**

The minimum period for $\phi_2$ is determined by the shift operation, followed by the write back of the results into the register block. The time required for this is: 4.8 ns

The non-overlap period between each phase must also be carefully calculated. The two non-overlap periods require different considerations.

**Non-overlap between $\phi_1$ and $\phi_2$**

The non-overlap period between $\phi_1$ high to $\phi_2$ high is determined by the operation of the saturated arithmetic control. The time required for this is: 4 ns

**Non-overlap between $\phi_2$ and $\phi_1$**

The non-overlap period between $\phi_2$ high to $\phi_1$ high is determined by the hold time of the static latches. This period is built into the delay between the falling edges of $\phi_2$ and $\phi_3$, and was extensively simulated to obtain a minimum period of: 4.5 ns.

The non-overlap period between $\phi_3$ high to $\phi_1$ high represents a safe margin during which no operations occur. To ensure that this safe margin is always present an additional 2.5 ns delay should be added to the non-overlap period between $\phi_2$ high to $\phi_1$ high. This gives a non-overlap period between $\phi_2$ high to $\phi_1$ high of: 7
By taking these minimum periods into consideration an estimate to the fastest clock period can be made: 50 ns. This will allow the datapath to operate at 20 MHz.

Further simulation using the instruction control, and decoding hardware, showed that the datapath represents the slowest section of the processor. Therefore the clock period, and frequency, determined above are the values that can be used for the complete processor.

The complete datapath operated as required, and had a final size of 2000μm x 1400μm using 2 micron CMOS. Plate I at the end of the thesis shows the layout of the datapath, and various functional units.

6.4 Generation of Instruction Decoder

The instruction decoder required some thought as to the type of control methodology to be used. There are two distinct approaches that can be taken in developing the control section[2], the choice of which can be critical in certain instances. These two approaches are as below:

*Moore Approach
This approach to the control relies upon generating a set of commands as determined by the flow of the algorithm. To implement conditional instructions requires complex branching mechanisms, and condition code evaluation.

*Mealy Approach
This approach generates a set of commands as determined by the flow of the algorithm, and a set of input conditions. This approach eliminates the need for conditional branching, and can reduce the number of cycles require to implement an algorithm. However these benefits come at the cost of increased complexity in the instruction decoder, as it not only has to decode the opcode, but also it must consider all the relevant conditions for each instruction.
The choice was made to use the Mealy approach, since this eliminates the need for complex sequencing, and conditional branching. This is crucial since the use of conditional branches is impossible for processors running under SIMD control.

Once this choice was made, the development of the instruction decoder could commence. This development made use of MPLA, which is the PLA generator that comes with the University of California at Berkeley VLSI Tools.

The use of MPLA requires two distinct steps, first a template file has to be created, and then a personality matrix for the PLA has to be generated. The purpose and execution of these two steps are considered below.

6.4.1 PLA Template

The PLA layout style is defined by a template, this is designed in MAGIC. This template is used to define the tiles that make up the complete PLA. Each tile represents a different section of the PLA layout, for example the tile labelled <bot_and> is used to define the layout for the bottom of each AND plane. This tile represents the input to the AND plane in the template style used.

Special customising of the PLA template was carried out to allow the use of clocked static registers on the PLA inputs, and the conditioning of the PLA outputs. By incorporating these features into the PLA template the layout size, and routing requirements for these functions were reduced.

The template used for the instruction decoder is shown in Plate IIa. The important features to note on this template are the input and output control. These tiles are shown on the bottom of the template and are labelled <bot-and>, <botl-or> and <botr-or>. The functions carried out by these tiles are:

<bot-and>

This tile represents the input to the PLA, and has been customised
to include a latched input, and a static storage element. This allows the instruction register, and condition statuses, to be latched into static registers during $\Phi_2$.

$<botl-or>$ and $<botr-or>$
These two tiles are functionally similar, and are used to select the output from the PLA. There are two signals used to select this output, providing either the OR plane output, or setting all the outputs to low.

The other tiles to note are the control for these sections. These tiles are labelled $<ll-and>$ and $<lr-or>$. 

$<ll-and>$
This tile is used to input and buffer the clock signal. The use of a large buffer stage in this tile removes the need for additional buffer stages to be inserted along the inputs to the PLA.

$<lr-or>$
This tile provides the output control for the PLA. The signals used to control this tile are derived from a finite state machine, and are used to select between executing the next instruction, or inserting a NOP.

Care has been taken in the development of these tiles to ensure that all the relevant control signals abut. This allows the PLA to be generated to an arbitrary size, with the control signals being stretched to the required length. Plate IIb shows a PLA constructed using this technique. (The PLA shown in this plate is not the Instruction Decoder.)

6.4.2 PLA Personality Matrix

The second requirement for the PLA generation is the personality matrix, this expresses the function that the PLA is required to implement. Various methods can be used to generate this personality matrix, in this case a special program was developed that could convert a textual description of the PLA into the required format. This program is
considered in more detail in Section 6.10

The description that was used to generate the personality matrix for the instruction decoder is given in Appendix A.3. This description includes every instruction in the instruction set, and considers all the related conditions for each instruction. The penalty for using the Mealy approach can be seen in the large number of terms needed to define some of these instructions. For example the MULTI instruction requires 8 different terms to define the operations for all the relevant conditions.

6.5 Development of Control Switching

The method that is used to control the switching between SIMD and MIMD control was presented in Chapter 5. This scheme uses special instructions in the MIMD program to determine control switching. These instructions are:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCAL_OFF</td>
<td>Select SIMD Control</td>
</tr>
<tr>
<td>ESCAPE</td>
<td>Select MIMD Control</td>
</tr>
<tr>
<td>STOP</td>
<td>Select SIMD Control upon Initiation of Next Network Update</td>
</tr>
</tbody>
</table>

The main point to consider in this discussion is how each of these instructions are decoded, and the effects of these methods. The reason for the different decoding techniques is due to the different control modes in which each will be executed. The two techniques used are considered below:

• LOCAL_OFF and STOP
  The LOCAL_OFF and STOP instructions are decoded conventionally by the instruction decoder. This is made possible since the processor will already be executing the MIMD instruction stream, so these instructions will automatically be loaded into the instruction decoder.

• ESCAPE
  To detect the ESCAPE instruction is more complex since this instruction is read as a weight's value, while the processor is
under SIMD control. To detect this combinational logic is added to the instruction multiplexer.

This operation is made possible by using an exclusive bit pattern for the **ESCAPE** instruction (0b10000000). This pattern was chosen since it represents the largest 2's complement negative integer, which is redundant in this design.

The effect of these two different decoding techniques is that each control type requires a different pipeline strategy. This is due to the SIMD control requiring an additional cycle in which to detect an **ESCAPE** instruction. A summary of the two different pipeline schemes is given below:

**MIMD Control**

The MIMD control requires a two stage pipeline, decode and execute. This pipeline arrangement is fairly conventional, and is shown in Table 6.4.

**SIMD Control**

For SIMD control a three stage pipeline is used. The requirement for this additional stage in the pipeline is to allow time for the **ESCAPE** instruction to be detected upon each **LD_WEIGHT** instruction. The other two stages follow the conventional decode and execute pipeline strategy. This three stage pipeline is shown in Table 6.5.
Table 6.5 SIMD Three Stage Pipeline

| Stage 1 |  |
|---------|  |
| Φ1      | Latch SIMD Instruction |
| Φ2      | Decode Weights Bus for ESCAPE |

| Stage 2 |  |
|---------|  |
| Φ1      | Select Next Instruction |
| Φ2      | Decode Instruction |

| Stage 3 |  |
|---------|  |
| Φ1      | Output Operands and Execute |
| Φ2      | Shift and Write Result |

To summarise the pipeline strategy for both SIMD and MIMD control, and to how the switching between them occurs, a PERT[2] chart is shown in

Diagram 6.8 Pert Chart Showing SIMD and MIMD Control
Diagram 6.8. This shows the pipelining for both MIMD and SIMD control, and how the transition between these two pipeline schemes occurs.

This diagram shows the SIMD control, represented by a dashed line, as three concentric loops. These loops represent the three stage pipeline used for SIMD control. The MIMD control, represented by the dotted line, only has two loops, representing the two stage MIMD pipeline. The transitions between the two pipeline schemes are represented by the two lines labelled 'ESCAPE', and 'LOCAL OFF or STOP'. The diagram shows how these lines are used to jump between the two pipeline schemes. The 'LOCAL_OFF or STOP' transition results in a change from the two stage pipeline to the three stage pipeline, while the 'ESCAPE' transition changes from the three stage to the two stage pipeline.

Both transitions require special control to synchronise, and to adjust between the different pipelining schemes. The problems found in implementing these transitions are discussed below, and the solutions finally used are presented.

6.5.1 Transition from MIMD to SIMD

This transition is performed by either the LOCAL_OFF or STOP instructions. These two instructions require synchronisation to occur before SIMD control can commence. The different requirements for these two instructions are as below:

- **LOCAL_OFF**
  This instruction is used to switch to SIMD control to execute the next virtual neuron. To synchronise this transition relies upon waiting for the synchronisation signal that is sent with the first instruction in the SIMD program. NOP instructions are inserted until this synchronisation signal arrives.

- **STOP**
  This instruction is used to synchronise the start of each network update, and requires the processor to wait until the start signal is generated by the external input and output hardware. NOP instructions are again inserted until this signal is set. Once
this signal is generated, the processor synchronises with SIMD control using the technique given above.

6.5.2 Transition from SIMD to MIMD

The transition between SIMD and MIMD control presents a problem that is very similar to the execution of conditional branch instructions in conventional processors. Such conditional branch instructions present problems when they are performed on a pipelined architecture[3]. This problem is highlighted below, and possible solutions to this are presented.

The problem that occurs in conditional operations on a pipeline is that the condition only becomes valid after the instruction that sets the condition has been executed. This results in the condition being set too late for the next instruction to use. This problem is outlined below:

<table>
<thead>
<tr>
<th>FETCH instr n</th>
<th>FETCH instr n+1</th>
<th>FETCH instr n+2</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXEC instr n-1</td>
<td>EXEC instr n*</td>
<td>EXEC instr n+1</td>
</tr>
</tbody>
</table>

* Condition for instr n valid here.

In this case instr n+1 cannot use the conditions set by instr n. The first instruction that can use these conditions is instr n+2.

Unless special attention is made, the instruction following the branch instruction (instr n+1) may be incorrectly executed. To ensure this does not occur one of several techniques can be used:

**Page-Mode Access**

Processors that use on-chip microcode can make use of a technique called page-mode access. An example of such a processor is the Motorola 68000.

The operation of this mode relies upon organising the microcode so that all possible conditional outcomes are held in the same row in memory. This then allows the condition to be evaluated as normal, and the row to be selected at the start of the next clock
phase. This scheme eliminates any latency in the conditional branching.

In the example below the row select for instr n+1 is made before the condition is established. Once the condition is established the column select and execution occurs together in the same clock cycle.

<table>
<thead>
<tr>
<th>ROW SEL instr n</th>
<th>ROW SEL instr n+1</th>
<th>ROW SEL instr n+2</th>
</tr>
</thead>
<tbody>
<tr>
<td>COL SEL instr n-1</td>
<td>COL SEL instr n</td>
<td>COL SEL instr n+1</td>
</tr>
<tr>
<td>EXEC instr n-1</td>
<td>EXEC instr n*</td>
<td>EXEC instr n+1</td>
</tr>
</tbody>
</table>

* Condition for instr n valid here.

**Insert a NOP**

By inserting a NOP after the conditional instruction, an extra cycle is made available during which the conditions can be correctly evaluated. This allows the conditional branch to occur without the accidental execution of the instruction following the conditional branch.

In the example below a NOP instruction is executed before instr n+1. This NOP ensures that all conditions are valid before instr n+1 is executed, thus enabling a conditional branch to occur correctly.

<table>
<thead>
<tr>
<th>FETCH instr n</th>
<th>FETCH nop</th>
<th>FETCH instr n+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXEC instr n-1</td>
<td>EXEC instr n*</td>
<td>EXEC nop</td>
</tr>
</tbody>
</table>

* Condition for instr n valid here.

The use of NOP instructions in this manner can be likened to introducing 'bubbles' into the pipeline scheme. These 'bubbles' can seriously impair the performance of a pipelined processor, and many techniques have been proposed to remove them. These techniques rely upon complex compiler algorithms to resolve any dependencies, hence minimising the cost of conditional branches.
To achieve this re-ordering of instructions is performed. Compilers for many RISC processors move instructions from either before an conditional branch, or from one of the target of the conditional branch, to the instruction just after the branch (instr n+1). The MIPS R3000 does this using a technique called delayed branches, while both the SUN SPARC and MIPS R4000 use a technique called annulling branches.

•Delayed Branch
By moving an instruction from before a branch to just after it results in a delayed branch. This instruction is always executed, with the branch occurring on its completion.

In this example the instruction at instr n+1 has been moved from before the conditional branch to just after it. This instruction will always be executed, allowing time for the conditions for instr n to be evaluated. In this example instr n+2 represents the first instruction from the target of the branch.

<table>
<thead>
<tr>
<th>FETCH instr n</th>
<th>FETCH instr n+1</th>
<th>FETCH instr n+2</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXEC instr n-1</td>
<td>EXEC instr n'</td>
<td>EXEC instr n+1</td>
</tr>
</tbody>
</table>

* Condition for instr n valid here.

•Annulling Branches
By shifting an instruction from one of the branch targets to just after the branch instruction, and conditionally executing it, results in an annulling branch.

In the example below instr n+1 is moved from one of the branch targets, and will only be executed if that target proves to be correct. If the branch target is incorrect the instruction is 'annulled' and a NOP is performed.
FETCH instr n  |  FETCH instr n+1  |  FETCH instr n+2
---|---|---
EXEC instr n-1  |  EXEC instr n  |  EXEC instr n+1 or NOP

Condition for instr n valid here.

**Branch Linking**

To improve on the likelihood of this instruction being executed correctly the SUN SPARC processor and the MIPS R4000 uses two specialised instructions:

Conditional Branch and Link Likely
Conditional Branch and Link Unlikely

These two conditional branch instructions state which of the two branch targets is most common, thus allowing the first instruction from the most frequent branch target to be selected.

**Selected Solution**

From these options the annulling branch technique was chosen. This was primarily because there is no convenient instruction that can be moved from before the branch to just after it - so a delayed branch was not possible - and because there is a low probability of the branch being taken.

This solution results in a zero overhead for the branch if control remains under SIMD mode, while a single cycle delay is imposed when a switch is made to MIMD control.

### 6.5.3 Implementation of Control Switching

The hardware that is used to control the execution of each control type, and to perform the switching between them, is implemented by a finite state machine. This finite state machine controls the instruction multiplexer and the instruction decoder, and provides all the signal required to synchronise and switch between the two different control
Diagram 6.9  SIMD and MIMD Control Selection in a Processor

types. The integration of this finite state machine, for a single processor, is shown in Diagram 6.9.

This finite state machine is shown in Diagram 6.10. This diagram shows the states that control the execution of both SIMD and MIMD control, and how the transitions between these occur. This finite state machine uses the Mealy approach, which means that the output from the finite state machine depends upon the current state, and upon the input conditions.

To represent this diagrammatically a standard format is used[3]. In this format each state is represented by a circle, and each transition by a line segment. The states are labelled inside each circle, with the transitions being labelled on each line segment. The transitions are labelled in a format that represents the input conditions that result in the transition, and the output signals that are set by the finite state machine. This format uses a slash '/' to separate the input conditions and the output signals.
A brief description of each state is given below:

• **GLOBAL**
  This state selects SIMD control, and the FSM will remain in this state until the **ESCAPE** instruction is detected.

• **FETCH** and **LOCAL**
  These two states control MIMD control, and the FSM will alternate between them. This alternation provides a means of interleaving each MIMD instruction with a **FETCH** instruction. The FSM will remain in these two states until the **LOCAL OFF** or **STOP** instruction is executed.

• **FINISH**
  This state represents that the processor has completed its virtual neurons. The finite state machine will remain in this state until the succeeding processors have also completed their virtual neurons.
neurons. When this occurs the FSM will move to the NEXT_IT state ready for the next network iteration.

**NEXT_IT**
This state represents that the processor is ready for the next network iteration. It will remain in this state until the start signal indicates that the next iteration can commence.

**WAIT**
This state is used to synchronise the start of SIMD control. A move to the GLOBAL state only occurs upon the sync signal going high.

### 6.6 Development of SIMD Control

This chapter has covered the complete operation of a single processor. The operation of the SIMD control, and the bus interfaces that connect to the array are now considered.

The functions required of the SIMD control block are:

- Supply SIMD instructions to the processor array
- Synchronise the array with external input and output

These functions are implemented by a finite state machine (GLOBAL). This finite state machine uses external signals and status signals from the processor array to synchronise the array with the external devices. The finite state machine also generates the SIMD program by a sequence of states, each of which sets the finite state machine output signals to the opcode of a particular instruction in the SIMD program.

Diagram 6.11 shows the configuration of this SIMD control block, and how the signals from the processor array and external devices are connected.

The finite state machine is shown in Diagram 6.12. This again uses the Mealy approach, so the same representational format of the finite state machine is used. The only slight variation is the simplification of the
Diagram 6.11  SIMD Control Block

output signals, these are shown as SIMD instructions, as opposed to the full opcode.

The states in this finite state machine are:

• **SET_UP**
This state is used to reset the processor array, which occurs when PAD_reset goes high. The SIMD control block then outputs a **RESET** instruction which loads zero into all registers in the processor array, and sets each processor to MIMD control.

• **WAIT**
After resetting each processor this state is used to wait until the processor array is ready to start with the first network update.
• **FINISH**
This state represents that the processor array has completed a network update. During this state, input and output from the device can be performed, this is initiated by PAD_finish going high. When the input and output is complete, indicated by setting PAD_start high, the SIMD control commences.

• **INST_1 - INST_14**
These states are used to broadcast the SIMD program to the processor array. The finite state machine automatically moves between these states, removing the need for an external program counter. At state INST_1 the finish_out signal is tested, this is used to signal that a network update has completed. If this signal is high then the finite state machine will move to the FINISH state.

Diagram 6.12 Global Finite State Machine
6.7 Implementation of Bus Logic Functions

The two logic functions that are required on the input and sigma buses are:

<table>
<thead>
<tr>
<th>Input Bus</th>
<th>Modulo Address Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sigma Bus</td>
<td>Sigma Threshold</td>
</tr>
</tbody>
</table>

These functions are performed by specialised logic positioned on the buses, and are shared between the processors using the same arbitration scheme as previously discussed.

6.7.1 Modulo Address Generation

The use of modulo address generation was discussed in Chapter 4. The ability to setup a modulo buffer within the input table was seen as crucial in the implementation of input windows for certain signal processing networks.

To perform modulo address generation on the input bus requires the top bits of the input address to be selectively set to zero. This has the effect of creating a modulo circular buffer within the input table, with its base address at 0h0000. The size of the modulo circular buffer is determined by the number of significant bits in the address (ie number of bits not set to zero).

To allow a variety of modulo circular buffer sizes there are six control pins that are used to configure the buffer size. These pins determine which address bits are to be set to zero during modulo address generation. Table 6.6 shows the pins used, and the relevant modulo buffer sizes for each setting.

The modulo address generation control is positioned on the input bus, and is shared between the processors on a chip. To control the type of address mode that is required a status signal is sent with the address
Table 6.6 Modulo Buffer Sizes

<table>
<thead>
<tr>
<th>mod_11</th>
<th>mod_10</th>
<th>mod_9</th>
<th>mod_8</th>
<th>mod_7</th>
<th>mod_6</th>
<th>Modulo Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4K Bytes</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2K Bytes</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1K Bytes</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>512 Bytes</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>256 Bytes</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>128 Bytes</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>64 Bytes</td>
</tr>
</tbody>
</table>

from each processor. This signal originates from a status register in each processor, and is controlled by the following two instructions:

MODULO_ON  #Turn Modulo Address Generation On
MODULO_OFF #Turn Modulo Address Generation Off

The logic used to control the address generation is shown in Diagram 6.13.
6.7.2 Sigma Table Bus

After all the inputs have been computed a sigma function is performed on the partial sum. This is performed by a lookup table, using the partial sum as the lookup address.

However the lookup function does not require all 32 bits of the partial sum for the lookup address. It is only the high gain region in the middle of the curve that requires to be evaluated. To minimise the size of the lookup table, only this high gain region need be included in the lookup table, with the neuron output saturating if the partial sum lies outside this range. Diagram 6.14 shows how the sigma function is represented, and where the saturating occurs.

![Diagram 6.14 Sigma Function Threshold Levels](image)

The logic that carries out this threshold function is positioned on the sigma bus, and is shared between the processors. This logic evaluates the partial sum, and according to whether it exceeds the threshold values, it will either saturate the sigma function, or use the value obtained from the lookup table.
The time overhead for this control is negligible since it overlaps the lookup table in its operation. The sigma threshold is evaluated during Φ1, during which time the lookup table is addressed. The output from the sigma threshold is selected during Φ2, which is either the lookup table data value, or 0h7f and 0h80 according to whether it saturates positive or negative. This overlap between the threshold function and the lookup table is required due to the size of the boolean functions that needs to be evaluated.

The derivations of the boolean functions that are used to detect these thresholds are given below:

Positive saturation:

\[(\text{sig}_{11} + \text{sig}_{12} + \text{sig}_{30}) \cdot \text{sig}_{31}\]

Which reduces to:

\[\frac{1}{1} (\text{sig}_{11} + \text{sig}_{12} + \text{sig}_{30}) + \text{sig}_{31}\]

Negative saturation:

\[(\text{sig}_{11} + \text{sig}_{12} + \text{sig}_{30}) \cdot \text{sig}_{31}\]

Which reduces to:

\[\frac{1}{1} (\text{sig}_{11} \cdot \text{sig}_{12} \cdot \text{sig}_{30}) \cdot \text{sig}_{31}\]

or:

\[\frac{1}{1} (\text{sig}_{11} \cdot \text{sig}_{12} \cdot \text{sig}_{30}) \cdot \text{sig}_{31}\]

The implementation of these functions is shown in Diagram 6.15. This diagram shows the logic used to detect the thresholds, and how the signals from this logic are used to select between 0h7f, 0h80 and the value obtained from the lookup table.
A further point to note from this diagram is how the lookup address is generated. This address requires 12 bits to access fully the 2 Kbyte lookup table. The bits used to make up this address are the lower 11 bits of the partial sum, and the MSBit of the partial sum. The MSBit is required since the sign of the partial sum must be incorporated in the lookup address.

A future development that could be used to reduce further the size of the sigma function lookup table, would be to use a sign reflection of the sigma function. This would require a 1K lookup table, which would hold the positive side of the sigma function only. For a negative partial sum the sigma function would have to be inverted, this would be carried out by the sigma threshold function, and be controlled by the sign of the partial sum.
6.8 Full Simulation of Processor Array

The last stage in the development of this device was the full simulation of two processors working in parallel. The layout for this simulation incorporated memory blocks for the input table, sigma lookup table, and local memory for each processor's weights table. These memory blocks were constructed out of PLAs, which would simulate the operations of ROM chips. To generate the data for the weights table MIMD programs were written, and assembled. These programs were then executed during the simulation.

Listing 6.1 shows the MIMD program that was executed on each processor. This program includes as many features as possible, and the data was carefully chosen to test the full range of the arithmetic operations.

The simulations were performed by RNL, which again enabled full simulation of the layout to be performed. The resulting output was then used to check the operation of the global control and the synchronisation between the processors.

These simulations showed the correct working of all sections. However due to the very large size of the output files, it is not possible to show the results of these simulations in this thesis.

6.9 Design Conclusion

A photomicrograph of the entire device is given in Plate III. This device incorporates a single processor, SIMD control block, and the modulo and sigma threshold functions. This first implementation only included a single processor, in order to minimise the silicon area and hence the cost of fabrication.

A special signal has been included in this first implementation to allow cascades of processor chips to be built on a PCB. This signal (lead) is used to interface with the sync signal, which is generated by the SIMD control block in the normal manner. The arrangement for such a cascade is shown in Diagram 6.16.
### Listing 6.1 MIMD Program Simulated on Processor Array

This lead signal is used as an input to the SIMD control, and will only allow the SIMD program to commence when this signal goes high. This signal is set high for the master processor, and is daisy-chaining with the sync signal for the succeeding processors. Such an arrangement will ensure that all the slave processors commence SIMD control in synchronisation with the master processor.
Diagram 6.16 Cascade of Processor Chips

The design statistics from this first 2 micron implementation are:

Total Size of Die:
\[ 4184\mu m \times 4354\mu m = -18.22 \text{mm}^2 \]

Size of a single processor:
\[ 2316\mu m \times 2461\mu m = -5.7 \text{mm}^2 \]

Size of a single processor with routing
\[ 2700\mu m \times 2461\mu m = -6.6 \text{mm}^2 \]

Size of array support blocks:
\[ -0.4 \text{mm}^2 \]

Number of transistors:
5300 per processor
6300 for whole device
6.10 Future Large-Scale Integration

After completing this initial implementation it is now possible to assess whether this architecture has achieved the objectives of developing a fully integrated neural device for small-scale applications. To assess this a feasibility study into building a fully integrated neural signal processing device is considered. This study estimates the size of device that would result in a number of different configurations using 0.8 micron CMOS as the target technology. This study is reasonable since 0.8 micron is now widely used in many commercial processes.

To perform a 1st order estimate of the silicon area when using 0.8 micron CMOS requires the use of a scaling factor $a$. This scaling factor is a dimensionless constant that can be applied to all features in a design. It should be noted that not all features scale in such a linear manner, however for the purposes of this estimate this fact is ignored. The use of this scaling factor, for the features that we are interested in, is considered below:

\[
\begin{align*}
\text{Length: } & \quad 2 \text{ micron} / a \rightarrow 0.8 \text{ micron} \\
\text{Area: } & \quad 2 \text{ micron} / a^2 \rightarrow 0.8 \text{ micron} \\
\text{Where: } & \quad \text{Scaling factor } a = 2.5 \\
\end{align*}
\]

From these scaling equations it is possible to scale the sizes of the various sections in the implementation given above:

- Size of a single processor: $1.06\text{mm}^2$
- Size of array support blocks: $0.064\text{mm}^2$

In order to fully integrate this device, RAM blocks and analogue and digital converters must be implemented. By using the Cadence RAM Generator[6], it is possible to calculate the size of various memory blocks, and by looking at ES2 Analogue Cell Library[7] the size of the analogue and digital converters can also be estimated.

From the estimated size of the RAM blocks the scaling equations can be used to calculate the area that they would occupy in 0.8 micron CMOS.
For the analogue cells these scaling equations are not appropriate. The reason for this is that all analogue cells are positioned in the pad frame, where they can be electrically isolated from the digital components in the core. Since these analogue cells are positioned directly as pad cells the scaling equations given above cannot be used, since pad dimensions do not scale in accordance with the other dimensions in any technology. This being due to the physical difficulties encountered in pad bonding, and the required electrical protection.

Before the area of these blocks can be calculated, it is necessary to stipulate the sizes of the various memory blocks that are to be used:

**Sigma Table: 1K Bytes**
This 1K sigma table makes use of the sigma reflection technique discussed in Section 6.7.2.

**Input Table: 1K Bytes**
This will allow a network to be approximately 4 times the size of the Tx Extraction Network to be implemented. Additional memory can be added, either internally or externally, if required.

**Weight Table: 1K Bytes**
This memory size is fairly arbitrary, and should be adjusted to take into account the connectivity of the types of network being implemented. (The greater the connectivity the larger the weights table should be.)

From these assumptions the sizes of the RAM blocks, and the analogue and digital converters, can be estimated, these values are specified as in a 0.8 micron technology:

- RAM Block (1K Bytes): 2.15mm²
- Analogue Pad Frame: 580μm wide

By using the above dimensions, the size of a fully integrated neural device can be estimated. Diagram 6.17 shows the total device size for
a number of different configurations. These configurations utilise different numbers of processors (1 - 14 processors), each using the same size memory blocks as specified above.

![Diagram 6.17 Graph of Die Sizes for Different Array Configurations]

From this analysis it is possible to see that a fully integrated device can be built, that incorporates 14 processors, requiring only 74mm².

To obtain an estimate the performance of such a device, the scaling factor \( \alpha \) can be used on the simulation results presented in Section 6.3.7.

To use this scaling factor \( \alpha \) on the clock period, the transient response \( \tau \) of a MOS transistor must be considered:

\[
\tau = \frac{\text{Length Transistor}}{\text{Electron Velocity}}
\]

By applying the scaling factor \( \alpha \) to the transistor length, we can calculate the scaling effect on the transient response \( \tau \).

\[
2 \text{ micron } \tau / \alpha \rightarrow 0.8 \text{ micron } \tau
\]
From the clock period derived under simulation it is possible to estimate the maximum performance this device can achieve in a 0.8 micron technology:

- **2 micron clock period:** 50 ns
- **0.8 micron clock period:** 20 ns

This clock period will allow the device to operate at 50 MHz, thus providing a maximum performance for a single processor of $3.5 \times 10^6$ connections per second, and an array of fourteen processors with $5 \times 10^7$ connections per second. The performance of a single processor, and that of a fourteen processor array, is compared with some other neural implementations in Diagram 6.18. This diagram is only interested in peak performances - and uses the same information that was presented in Chapter 4.

From this diagram it can be seen that the peak performance of this architecture compares favourably with the MIMD implementations, while saving considerably on the hardware costs. The performance of this architecture when compared to the SIMD implementations is not so impressive. However it should be remembered that the actual performance obtained from a generic SIMD array is considerably lower that the peak performance shown in this diagram. (The study in Chapter 4 showed that for the Tx Extraction Network the actual performance for a SIMD array obtained was only 47.5% of the peak performance.)

To study the performance of this architecture in more detail requires the breakdown of a typical network mapping - and an analysis of the proportion of SIMD instructions to MIMD instructions and NOPs. (See Appendix 7.)

This analysis takes the original network mappings for the Tx Extraction Network that were presented in Chapter 5, and carries out a more detailed analysis on their performances; these results are presented along with the initial figures in Table 6.7. The figures obtained from this analysis match fairly closely to those originally estimated in Chapter 5. This analysis shows that the average performance obtainable
Diagram 6.18  Peak Performance of Device

is within 93% - 96% of the maximum performance.

Table 6.7 Summary of Performance Efficiency

<table>
<thead>
<tr>
<th></th>
<th>SIMD Array</th>
<th>Silicon Efficient</th>
<th>Reduced Sample</th>
<th>Reduced Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Projected Efficiency</td>
<td>47.5 %</td>
<td>98 %</td>
<td>95 %</td>
<td>99.6 %</td>
</tr>
<tr>
<td>Actual Efficiency</td>
<td>&lt;=47.5 %</td>
<td>95.8 %</td>
<td>93.4 %</td>
<td>94.4 %</td>
</tr>
</tbody>
</table>

Diagram 6.19 shows this average performance, and plots a series of points that relate to the required performance for a number of neural network models. From this diagram it is possible see that this device
can used to implement a number of networks that have been developed for real-time signal processing tasks.

The performance required for the Tx Extraction Network can be matched by that of a single processor. Such a device would occupy less that 25 mm$^2$ providing a near ideal solution to the task presented in Chapter 3.

Diagram 6.19  Projected Performance of Device

---

1 TX Extraction Network [I S Howard, M A Huckvale. Speech Fundamental Period Estimation using a Trainable Pattern Classifier Speech 88 Edinburgh August 1988]


3 Isolated Work Recognition [I S Howard, M A Huckvale. Two-Level Recognition of Isolated Words using Neural Networks 1st IEE Int. Conf. on Artificial Neural Networks Oct 1989]

4 Electromyographic Signal Classification [C N Schizas, C S Pattichis, I S Schofield, P R Fawcett L T Middleton. Artificial Neural Network Algorithms in Classifying Electromyographic Signals IEE 1st Int. Conf. on Artificial Neural Networks Oct 1989]

6.11 Design Environment

The design environment used for this implementation was based on the University of California at Berkeley (UCB) VLSI Design Tools[8], and the Northwest Laboratory for Integrated Systems (LIS) at University of Washington Simulation Tools[9]. These tools were run on a SUN IPC, under SUN UNIX.

The target technology was ES2 (European Silicon Structures) 2 micron CMOS[10]. The format required for this is CIF (Calma Intermediate Format), requiring the internal MAGIC format to be converted to this before the shipping of the design.

Additional utilities were created to support these design tools. These utilities worked within the UCB environment, and allowed files to be created, and modified, to simplify repetitive functions. The two utilities created were ESP and NET. These were written using the UNIX CSHELL and AWK[11].

Diagram 6.20 shows the entire design environment, and how the various tools fit together. A discussion of these tools is given below:

- MAGIC
  MAGIC is the main tool that fits in the centre of this design environment. This tool is an interactive editor for VLSI layout, and allows full custom designs to be created. Various features are included to simplify this task for large designs; including cell hierarchies, and automatic routing algorithms.

  MAGIC uses its own internal format, with the facility of reading and writing to CIF.

- MEG
  MEG (Mealy Equation Generator) is a finite state machine compiler. It translates a high level language description of a finite state machine into several implementations and simulation formats. MEG uses the Mealy model for finite state machines, in which outputs are dependent upon inputs as well as the current state.
Diagram 6.20 Design Environment

Finite State Machine Descriptions

MEG
[Mealy FSM]

EQNTOTT
[Reduce terms]

MPLA
[Generate PLA Matrix]

ESP
[Generate PLA]

NET
[Create & Merge Netlist]

RNL
[Simulation]

MAGIC
[Cell Layout, Place & Route]

Generate PAD Frame

Insert Pads

Foundry

Pad_Frame_Request

Template

mag

Blank frame

esp

mag

Netlist, Top Level Cell

mag

mag

Net

esp

CDRC
[CIF DRC]
• EQNTOTT
EQNTOTT generates a truth table, suitable for PLA programming, from a set of Boolean equations that define the PLA outputs in terms of its inputs. EQNTOTT attempts to reduce the size of the truth table by merging minterms, it also maintains 'don't care' states to simplify further the size of the truth table output.

• MPLA
MPLA is a PLA generator for non-folded PLAs. Its takes its input as a truth table, in the same format as EQNTOTT output. This truth table defines where each transistor is placed in the AND and OR plane. A PLA template file defines the style of the PLA generated; this allows dynamic or static PLAs to be generated, with clocking on the inputs and outputs if required.

• ESP
ESP takes a textual description of a PLA and converts it into a truth table suitable for MPLA. No minimising is performed during this transformation, and 'don't care' states are not permitted.

This utility was written to simplify the generation of the instruction decoder. It allows the definition of each term to be written as a list of signals that should be generated for each input set. Each set of input conditions, and the output list, is then converted into a product term in the truth table format used as the input to MPLA.

Appendix A.5 gives the AWK listing for this utility.

• NET
NET takes the top level cell, and a Netlist to be merged, and creates a single Netlist file. NET works by combining all sub cell terminals, that have the same labels, into nets. Merging of the input netlist is then performed to create a single Netlist that is used to specify the connectivity for the routing algorithm. The Netlist file is read directly by MAGIC, which will route according to the defined nets in this file.
A three stage operation is required to generate a netlist from the top level cell:

Find all sub cell instances in the top level cell, and extract the sub cell name, and the instance name. These two names can then be passed on so that the terminals for each sub cell instance can be extracted. The output from this section is as below:

```
data_path.mag   data_path_0
```

For each sub cell instance, extract all terminals in that sub cell, and output these in a list. This output list should be in the hierarchical format used by MAGIC to label sub cell terminals. This format is as below:

```
data_path_0/msb_sum_b
    data_path_0/msb_c_out
```

From this complete list of sub cell terminals, a netlist can be generated by combining all terminals that have the same label. The merging of the input netlist is also performed during this operation. The output format of this file is in the format accepted by magic:

```
data_path_0/msb_sum_b
    shift_mux_0/msb_sum_b
```

```
data_path_0/msb_c_out
    decode_0/msb_c_out
```

This utility is given in Appendix A.4. This includes the three separate stages, these are held in individual programs called NET, LABELS and NETLIST. This collection of programs combines both UNIX shell scripts and AWK programs to manipulate the MAGIC files, and to perform the actual netlist generation.

- CDRC

CDRC is a design rule checker that works on CIF files. It checks the final CIF output against the ES2 design rules, and provides feedback
against latch-up rule violations.

•RNL
RNL is a timing logic simulator for digital MOS circuits. It is an event driven simulator that uses a simple RC (resistive capacitance) model of the circuit to estimate node transition times and to estimate the effects of charge sharing.

•PAD Frame Request
The ULVC (London University VLSI Consortium) PAD generation facility was used to create the PAD frame. This option was required since the ES2 PAD library is not compatible with MAGIC. The library cells make use of a smaller grid size than MAGIC can use, making it impossible to correctly read the PAD library into MAGIC.

The ULVC PAD generation facility requires a PAD Frame Request File to be generated, which defines the position of each PAD and the library cell that is required. A blank PAD frame is then generated, which shows the size and position of each PAD, and its terminals. Routing to this blank PAD frame can then be carried out with the real PAD cells being inserted after the design has been output to CIF.

The PAD frame request file that was generated for this design is given in Appendix A.6.
Assessment of Architecture

This thesis has presented an architecture aimed at implementing neural networks and simple digital signal processing models. The work presented has outlined the requirements for such an architecture, and then developed it through a series of steps.

The resulting architecture can be seen to be very different to those that were discussed in Chapter 4. The requirements for this architecture has meant that a departure from the more conventional architectures has had to be made. Limitless expandability has not been the goal, but instead a compact, and highly efficient architecture.

This chapter leads on from the development of this architecture, and discusses where it fits into the general area of parallel processing.

7.1 Classification of Parallel Machines

When considering any parallel processor an attempt is normally made to classify it so that comparisons can be made to other similar machines. The most common classification scheme used for parallel processors is due to Flynn[11] who in 1966 introduced a classification based on the numbers of instruction streams and data streams that a machine possesses. Table 7.1 shows the Flynn Classification scheme.

Table 7.1 Flynn Classification Scheme

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Single Stream</th>
<th>Multiple Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Stream</td>
<td>SISD</td>
<td>SIMD</td>
</tr>
<tr>
<td>Multiple Stream</td>
<td>MISD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>

This classification has met with some criticism over the years due to its very broad nature, and due to the lack of examples of MISD machines.
The lack of examples of MISD machines can be attributed to the problems of potentially large variation in execution time across processors. This could prove to be inefficient since the speed of the data flow must be determined by the slowest processor. Despite this potential problem there has been an example of a MISD machine. This was the IBM Harvest attachment to the 7030 Stretch computer. The use of this attachment was in cryptography, where different algorithms could be applied to the same data stream in parallel.

Since this classification scheme several others have tried to use more criteria to introduce finer divisions in the classes. Most have concentrated on sub-dividing the parallel machines further. A high-level taxonomy of parallel machines is shown in Diagram 7.1. This taxonomy sub-divides SIMD and MIMD classes according to the connectivity and control style of each machine. (For further information on these classes of parallel machines the reader is referred to R. Duncan’s survey of parallel computer architectures[2].)

Diagram 7.1 High-level Taxonomy of Parallel Computer Architectures

In this classification scheme the architecture presented in this thesis would fit into the MIMD/SIMD class. There have been several machines that have been proposed, or built, that also fit into this class. These examples all take slightly different approaches to their control
mechanisms, and as a result have been further classified by other authors using extensions to Flynn's classification.

MSIMD Multiple Single Instruction Multiple Data  
SPMD Single Program Multiple Data  
MIMD/SIMD Partitionable SIMD/MIMD  
MIMSIMD Multiple Instruction for Multiple SIMD

Snyder[3] has extended Flynn's classification scheme in a different manner. The key point he introduced is that both the instruction stream, or thread, and the data stream are composed of addresses and values. His classification scheme uses the numbers of each of these to define the machine:

\[ I \times a \times v_a \times v \times a' \times v' \]

Where:
- \( a \) is the number of instruction threads
- \( v \) is the number of instructions
- \( a' \) is the number of operand addresses
- \( v' \) is the number of operands

Using this classification scheme a simplification is made to the actual values of \( a, v, a', \) and \( v' \). These are called class designators, and are specified as follows:

- \( s \) is for single value
- \( c \) is for some small constant
- \( m \) is from 1 to an arbitrarily large finite number

Thus a number of machines can be classified:

<table>
<thead>
<tr>
<th>Class Designations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IssDss</td>
<td>Von Neumann Machine (SISD)</td>
</tr>
<tr>
<td>IssDsm</td>
<td>SIMD machine with no addressability (systolic arrays)</td>
</tr>
<tr>
<td>IssDmm</td>
<td>SIMD machine with individual addressability</td>
</tr>
<tr>
<td>IssDcc</td>
<td>VLIW (Very Large Instruction Word) machines</td>
</tr>
<tr>
<td>IssDcc</td>
<td>MIMD machine using shared memory, or bus structure</td>
</tr>
<tr>
<td>IssDmm</td>
<td>MIMD array using an expandable communication scheme</td>
</tr>
</tbody>
</table>
To classify the architecture presented in this thesis using Snyder's classification scheme requires a number of different terms to include all the possible modes:

\[ I_{ssDcc} \] All processors operating under SIMD control
\[ I_{ccDcc} \] All processors operating under MIMD control
\[ I_{cc'Dcc} \] Some processors operating under MIMD control

Where:

\( c \) is the number of processors in array
\( c' \) is the number of processors operating under MIMD control + 1

The above classification schemes have been useful in defining the architecture that this thesis presented. It not only enabled a more formal description of its control type, but it also enabled similar machines to be identified. The following section describes some of these examples, and discusses the possible directions that this type of architecture may lead.

### 7.2 Other MIMD/SIMD Machines

A series of examples of MIMD/SIMD machines is given, which briefly outlines the methodology for the machine and the control mechanism that is used.

#### 7.2.1 Non-Von 4 - Columbia University

Non-Von 4[4] was developed at Columbia University during 1984. Its architecture is an extension from the preceding Non-Von machines, incorporating both MIMD and MSIMD (Multiple Single Instruction Multiple Data) control types.

The key motivation for this is to reduce the communications by the partitioning of different sections to work independently in a divide-and-conquer methodology. This allows different sub-tasks to operate remotely from the central processor, thus freeing the main control section, and reducing the top level communication workload. This partitioning is achieved by using a binary tree structure, with a
specified number of nodes supplying SIMD instructions to their sub-trees. This arrangement requires a master/slave relation, with each processor that can supply SIMD instructions being termed a LPE (Large Processing Element), and the processors in each sub-tree being termed SPEs (Small Processing Elements).

7.2.2 DADO - Columbia University

DADO\[5,6\] was again developed at Columbia University, at much the same time. DADO is a parallel machine developed to implement expert systems\(^1\). In this architecture there are several interesting features that merit some discussion. These features allow the array of processors to operate under SIMD or MIMD control, with each processor having the ability to switch other processors into or out of SIMD or MIMD control. A key point to note in this architecture is that there is no master/slave relation between groups of processors. Instead each processor is identical, thus increasing the flexibility in partitioning an algorithm on the machine since no account now has to be made to a fixed processor master/slave relation.

The basic operation in performing expert systems, relies upon the processors matching assertions with sets of productions. When a match is found, that processor is then able to set a number of relevant processors into SIMD control. This then allows each processor to update their assertions according to the actions defined by the matched production.

The interesting feature in this architecture is how the SIMD control is performed. This control is not performed in the conventional manner of broadcasting sets of instructions, but instead by a form of remote procedure call.

---

\(^1\) An expert System consists of a set of rules or production, and a database of assertions. Each production consists of a pattern element, called the left hand side (LHS) and a set of actions called the right hand side (RHS). The system operates by searching for a match of a LHS pattern with an assertion in the database. If a match is found the database of assertions is updated according to the actions specified by the RHS.
Each processor has within its memory a copy of the program required to update the assertion database. The relevant procedures within this program memory are initiated by this remote procedure call. This feature considerably reduces the communications overheads for the SIMD control, and allows increased flexibility by providing a means to customise the code in individual processors, and the implementation of conditional operations within the SIMD program.

7.2.3 XIMD - Carnegie Mellon University

XIMD[7] (Variable Instruction Stream Processor) is a more recent architecture that was presented in 1991. This architecture takes many ideas from VLIW Processors (Very Large Instruction Word)², and extends these to avoid some of the disadvantages found in implementing conditional operations.

XIMD takes the original VLIW model, and extends it by adding control mechanisms to each functional unit. This effectively allows each functional unit to operate under MIMD control. Special broadcasting of condition codes between functional units is provided so that each functional unit can operate on condition codes generated by other functional units. This not only allows more efficient execution of conditional operations, but it allows synchronisation between functional units to occur.

The broadcast of a busy/done flag is used in this synchronisation, enabling each functional unit to monitor the status of the remaining functional units. This allows each functional unit to operate independently on different sets of data, or to implement entirely different sections of code, and then to synchronise with each other for common portions of code. This can be used to simplify data transfers,

² VLIW Processors enable fine grain parallelism to be extracted from a wide range of applications. The operation involves the use of multiple function units. (eg. An address generation unit, floating point co-processor, memory control and integer processing unit.) The control program is horizontally coded in a single instruction stream, with each function units using a different fields; hence the requirement for the Very Long Instruction Word. The parallelism obtained is determined by the compiler, which resolves any conflicts, and maximises the overlap between the function units.
and to ensure that data dependencies are kept consistent. For example, one functional unit can be delayed until another functional unit's busy/done flag indicates that some section of data is now valid.

The use of this model enables all parallelism to be determined at compile-time, but still allow the flexible operation of each functional unit so that conditional operations, and uncertain events, can be efficiently handled. The compiler either resolves all these uncertainties and dependencies at compile time, or it generates code to resolve it at run-time, incorporating the synchronisation operations discussed above. In this model the compiler can be said to represent the main control unit for the XIMD processor, thus reducing the hardware requirements for the processor. Such an architecture is hoped to exhibit increased flexibility, and performance, with minimal extra hardware complexity.

7.3 MIMD/SIMD Machines in the Future

The examples above have indicated the varied use and reasons for using MIMD/SIMD control. In this thesis the architecture was developed so that software mapping of a neural network onto a processor array could be combined with a simple communications scheme. This architecture fits into the development of the general MIMD/SIMD machines discussed above, with the development of the XIMD architecture using some similar principles. It allows the independent operation for certain sections of code, followed by the synchronisation of functional units so that data dependencies can be resolved, and common input output routines can be implemented. This architecture also stresses the use of compile-time parallelism, which is again used in the XIMD architecture. This feature can be used to remove much of the control hardware since each processor's operation is determined at compile-time, and each processor operates deterministically.

It should also be noted that the goal for all of the above architectures was to improve efficiency, and to minimise hardware complexity. Despite the additional complexity in the control sections by having to select between various control modes, these architectures have seen an overall reduction in hardware due to the much simplified synchronisation and
communications control. These architectural features have recently been highlighted in two survey papers, both indicating the increased use of these more complex control mechanisms in the future.

T G Lewis[8], in a recent paper, highlighting the benefits of SPMD control (Single Program Multiple Data). In this model the data is partitioned across the processors, with each processor having its own copy of the program to be implemented. Each processor then runs under MIMD control using its own copy of the program. Upon completion the results from each processor is output and new data is distributed. Although this architecture requires separate program stores and control mechanisms for each processor, it does allow a reduced complexity from MIMD architectures due to the SIMD-like lock-step sequencing during the common parts of the program. This allows simplified synchronisation during input and output, while still offering more versatile control during the computations. The paper predicted that this type of parallel architecture would feature strongly in the 1990s due to the simplified hardware, and increased flexibility of the control strategy over SIMD arrays.

A study into massively parallel computers for artificial neural networks[9], has also pointed to the possibilities in developing machines with more elaborate control mechanisms. This report finishes in its conclusion with a goal for the future research:

The real challenge for computer architects in connection with the neural network area in the future lies in the implementation of Artificial Neural Systems (ANS), ie systems composed of a large number of cooperating modules of neural networks. Each of the modules should be allowed to implement a different network structure, and the modules must be able to interact in different ways at high speed. This implies that heterogeneous systems composed of homogeneous processing arrays must be developed, and that special attention must be paid to the problem of interaction between modules and between peripheral modules and the environment. The role of MIMD architecture in neural processing probably lies in this area, actually meaning MIMSIMD (Multiple Instruction Streams for Multiple SIMD arrays) architectures will be seen.
7.4 Postscript

Since writing this chapter the author has discovered another excellent example of a MIMD/SIMD machine; the Connection Machine CM-5[10]. This version of the connection machine departs from the original architectural model that resulted in the first Connection Machines. The original models of the Connection Machine were developed by Daniel Hillis and his company Thinking Machines, and are parallel arrays composed of thousands of single bit processors connected in a hypercube topology. In these first versions the control was purely SIMD, and a typical machine was composed of up to 64,000 processors.

The CM-5 is also a parallel machine, but with smaller numbers of processors (32 - 8,000), each composed of four accelerator chips and a RISC processor. This increased complexity allows each processor to operate under either MIMD control or SIMD control. According to Daniel Hillis the SIMD/MIMD duality of these machines will open up new classes of problems that earlier SIMD versions of the Connection Machine did not handle very well. This architecture is also hoped to be easier to program than conventional MIMD machines, due to the ability of synchronising between processors for common sections of code. Daniel Hillis is confident that this general trend for SIMD/MIMD parallel machines will be fairly settled for much of the next decade.
Chapter 8

Conclusion

This chapter outlines the work that has been covered in this thesis, and highlights the main research contributions that have been made.

8.1 Summary of Thesis

This thesis reviewed the foundations of neural networks, and identified some recent applications to which they have been applied. This preliminary study identified the requirements for a neural implementation to execute real-time neural signal processing models. A critical study of the current state in neural implementations showed that these requirements are not met in any single architecture; and an hypothesis was proposed for such an architecture.

This hypothesis drove the development of the architecture presented in this thesis. This architecture is unique in the area of neural implementations since it combines both SIMD and MIMD control modes. This makes the architecture very compact, whilst not compromising on the flexibility or programmability.

A silicon chip was developed using this architecture, and has been described fully in this thesis. This chip incorporates a single processor, allowing estimates to be made as to the performance, and silicon cost of a multi-processor device.

Chapter 7 presented a broader analysis of this architecture by looking at some similar parallel machines, and making some conclusions as to the likely course for the development of such machines.

8.2 Research Contribution

This section outlines the main contributions this thesis has made to the areas of neural architectures and parallel machines.
• Critical study of the current state in neural implementations
A critical study into the current state of neural architectures was able to identify some of the limitations imposed by these architectures. An analysis of the requirements for a portable real-time device showed that none of these architectures possessed the requisite criteria for the application area: high flexibility, soft-programmability and high efficiency.

• Development of an hypothesis out of which a new architecture could be developed
The isolation of beneficial features from current neural architectures allowed an hypothesis to be formulated. This stated that if the beneficial properties from MIMD and SIMD arrays could be combined then a suitable architecture may be developed.

• Development of a software methodology combining SIMD and MIMD control
A software methodology was developed allowing SIMD and MIMD control to be combined in a single architecture. This methodology utilises a single instruction that allows switching to occur between MIMD and SIMD instruction sources. The switching is controlled by the individual processors without the need for a master/slave arrangement or central controller.

• Development of an architecture to support this software methodology
An architecture was developed that could support this software methodology. This architecture enables each processor to switch between MIMD and SIMD instruction sources by under software control. This allows the architecture to exploit the flexibility provided by MIMD control with the low complexity of SIMD control and communication techniques.

• Design of a silicon chip incorporating this architecture
A silicon chip was designed using this architecture. This chip incorporates some novel techniques in the control mechanisms and pipelining strategies. This chip also allowed performance and
silicon costs to be estimated for future implementations incorporating multiple processor on a single chip.

• The identification of other similar work
The author has taken some time at the end of this thesis to present some recent developments in similar architectures. This review outlines the goals for a variety of architectures, and briefly looks at the control strategies used in implementing them. From this study the author was able to present some conclusions showing the increased interest in this class of MIMD/SIMD hybrid architecture.

8.3 Literature Published

The literature published relating to this thesis includes two technical articles from the TIMES newspaper, and a Conference Paper. This paper was presented to The 2nd International Conference on Microelectronics for Neural Network, held in Munich Germany, on October 16-18, 1991. All these articles are included in at the end of this thesis.

The author also hopes to get a further Journal paper published upon conclusion of this thesis. It is hoped that this paper will be complete upon submission, and will be included in the Appendix.
P.1 Confocal Light Microscope

The photomicrographs in the following pages were all taken using a confocal light microscope. The use of a confocal microscope allows very fine cross sectional images to be created. This is useful in many areas as it allows cross sectional images to be made inside a sample without the necessity for sample being cleaved. In medical applications it allows the internal operations of a groups of cells to be observed without having to damage the sample. It is interesting to note that the first recorded use of a confocal microscope was due to Marvin Minskey, who in 1957, used this technique for studying neurons in the brain. This technique is now widely used in the electronics area since it provides very good imaging of integrated circuits. By incorporating a digitiser, and appropriate software, 3-d models can be created, allowing a process engineer an unprecedented view of a section of circuit.

P.2 Operation of a Confocal Microscope

Diagram P.1 shows a simplified diagram of the confocal microscope principal. The microscope is represented as a single lens. Light from the light source is focused within the sample by the objective lens, and
the reflected back from the sample returns via the same path. Only light returning from the focused-on plane can return via the lens and the aperture, with the light from planes above or below the plane of focus being blanked off by the opaque part of the aperture. To actually create the image requires this point of light to be scanned over the sample, this is achieved by either moving the sample, scanning with the objective lens or by moving the aperture.

The high magnification plates were produced by the use of an oil immersion between the sample and the objective lens. This oil helps to direct the light, creating a brighter image, and reducing unwanted reflections from the sample.

Plates:

<table>
<thead>
<tr>
<th>Plate</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plate Ia</td>
<td>Datapath Layout (MAGIC Layout)</td>
<td>183</td>
</tr>
<tr>
<td>Plate Ib</td>
<td>Adder Layout</td>
<td></td>
</tr>
<tr>
<td>i)</td>
<td>MAGIC Layout</td>
<td>184</td>
</tr>
<tr>
<td>ii)</td>
<td>Photomicrograph</td>
<td>184</td>
</tr>
<tr>
<td>Plate Ic</td>
<td>B Input Selection (XOR Function)</td>
<td></td>
</tr>
<tr>
<td>i)</td>
<td>MAGIC Layout</td>
<td>185</td>
</tr>
<tr>
<td>ii)</td>
<td>Photomicrograph (Metal 2)</td>
<td>185</td>
</tr>
<tr>
<td>iii)</td>
<td>Photomicrograph (Poly &amp; Diff)</td>
<td>186</td>
</tr>
<tr>
<td>iv)</td>
<td>Photomicrograph (Metal 1)</td>
<td>186</td>
</tr>
<tr>
<td>Plate IIa</td>
<td>Template for Decoder PLA (MAGIC Layout)</td>
<td>187</td>
</tr>
<tr>
<td>Plate IIb</td>
<td>PLA Layout (MAGIC LAYOUT)</td>
<td>188</td>
</tr>
<tr>
<td>Plate III</td>
<td>Photomicrograph of Single Processor</td>
<td>189</td>
</tr>
</tbody>
</table>
Plate Ia  Datapath Layout (MAGIC Layout)
Plate Ib (i)  Adder Layout (MAGIC Layout)

Plate Ib (ii)  Adder Layout (Photomicrograph)
Plate Ic (i) B Input Selection (MAGIC Layout)

Plate Ic (ii) B Input Selection (Photomicrograph - Metal 2)
Plate Ic (iii)  B Input Selection (Photomicrograph - Poly & Diff)

Plate Ic (iv)  B Input Selection (Photomicrograph - Metal 1)
Plate IIa Template of Decoder PLA (MAGIC Layout)
Plate IIb  PLA Layout (MAGIC Layout)
Local Finite State Machine

This appendix shows the source code for the Local Finite State Machine. This code is written in MEG format; MEG is the Mealy Finite State Machine Generator that comes with the University of California at Berkeley VLSI Design Tools.

This finite state machine controls the source of the next instruction for each processor, and is responsible for MIMD/SIMD control switching and processor synchronisation.

The full state diagram for the Local Finite State Machine can be seen in Diagram 6.10.
LOCAL FINITE STATE MACHINE

Finite State Machine to Control Next Instruction Source

---

INPUT SIGNAL LIST

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Function</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>Start MLP Iterations</td>
<td>Global FSM</td>
</tr>
<tr>
<td>stop</td>
<td>Stop MLP Iterations</td>
<td>Decode PLA</td>
</tr>
<tr>
<td>sync_in (inv)</td>
<td>Synchronise Signal</td>
<td>Global FSM</td>
</tr>
<tr>
<td>local_off</td>
<td>Switch to Global Control</td>
<td>Decode PLA</td>
</tr>
<tr>
<td>local_pla_on</td>
<td>Switch to Local Control</td>
<td>Decode PLA</td>
</tr>
<tr>
<td>local_wgh_on</td>
<td>Switch to Local Control</td>
<td>instr_mux</td>
</tr>
<tr>
<td>finish_in</td>
<td>Following PEs Finished</td>
<td>Following PE</td>
</tr>
</tbody>
</table>

INPUTS : start stop sync_in local_off local_pla_on local_wgh_on \ finish_in;

---

OUTPUT SIGNAL LIST

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Function</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>sel_0</td>
<td>Select Instruction Fetch</td>
<td>instr_mux</td>
</tr>
<tr>
<td>sel_1</td>
<td>Select Local Instruction</td>
<td>instr_mux</td>
</tr>
<tr>
<td>sel_2</td>
<td>Select Global Instruction</td>
<td>instr_mux</td>
</tr>
<tr>
<td>instr</td>
<td>Output Data Path Control</td>
<td>decode</td>
</tr>
<tr>
<td>nop</td>
<td>Output NOP to Data Path</td>
<td>decode</td>
</tr>
<tr>
<td>finish_out</td>
<td>PE Finished Iteration</td>
<td>Preceeding PE</td>
</tr>
</tbody>
</table>

OUTPUTS : sel_0 sel_1 sel_2 instr nop finish_out;

---

FINITE STATE MACHINE DEFINITION

---

FSM CLOCK SIGNALS

- Input latched on Clock_2
- Output Available on Clock_1
- RESET on Clock_1 (Inhibits FSM Output)
--WAIT

--Operation: Synchronise PE on RESET or Start of Global Control
Remain in State until Synchronisation signal

WAIT : IF NOT sync_in THEN GLOBAL (sel_2 nop)

-- Select Next Global Instruction
-- Execute a NOP

ELSE WAIT (nop);

-- Continue to wait for ~sync_in, execute a NOP

--GLOBAL

--Operation: PE under Global Control
-- Remain in GLOBAL until Local Control is Selected

GLOBAL : CASE (local_pla_on local_wgh_on)
1 0 => FETCH (nop sel_0);
0 0 => FETCH (nop sel_0);
1 1 => FETCH (nop sel_0);

-- Select Local Instruction Fetch
-- Execute a NOP

ENDCASE => LOOP (sel_2 instr);

-- Select Next Global Instruction
-- Execute present Global Instruction

--LOCAL

--Operation: PE Under Local Control
-- Alternate between FETCH and LOCAL
-- Two method of turning Local Control Off
--    stop    End of Network Iteration
--    local_off    End of Local Code

LOCAL : CASE (stop local_off)
1 0 => FINISH (nop);

-- Finish Network Iteration
-- Execute a NOP

0 1 => WAIT (nop);

-- Local Control Off
-- Execute a NOP

ENDCASE => FETCH (sel_0 instr);

-- Select FETCH for next Local Instruction
-- Execute Present Local Instruction

--FETCH
--Operation: PE Under Local Control
-- Alternate between LOCAL and FETCH
-- Two methods of turning Local Control Off
-- stop End of Network Iteration
-- local_off End of Local Code

FETCH:
CASE (stop local_off)
1 0 → FINISH (nop);

-- Finish Network Iteration
-- Execute a NOP

0 1 → WAIT (nop);

-- Local Control Off
-- Execute a NOP

ENDCASE → LOCAL (sel_1 instr);

-- Select Next Local Instruction
-- Execute FETCH Instruction

--FINISH

--Operation: Pause at end of Network Iteration
-- Remain until ready for next Network Iteration

FINISH:
CASE (finish_in start)
1 0 → NEXT_IT (nop finish_out);

-- Ready for next Network Iteration
-- (Start signal reset and Following PEs finished)
-- Execute a NOP and Indicate that PE is ready

ENDCASE → FINISH (nop);

-- Wait for conditions - execute a NOP

--NEXT_IT

--Operation: PE ready for next Network Iteration
-- Remain until Start of Network Iteration
-- If condition change return to FINISH

NEXT_IT:
CASE (finish_in start)
1 0 → NEXT_IT (nop finish_out);

-- Still ready for next Network Iteration
-- Execute a NOP and Indicate PE is ready

0 1 → FINISH (nop);

-- Following PE no longer ready
-- Execute a NOP

1 1 → WAIT (nop);
-- Start of next Network Iteration
-- Execute a NOP

ENDCASE => NEXT_IT (nop);

-- Still ready for next Network Iteration
-- Execute a NOP
Global Finite State Machine

This appendix shows the source code for the Global Finite State Machine. This code is written in MEG format; MEG is the Mealy Finite State Machine Generator that comes with the University of California at Berkeley VLSI Design Tools.

This finite state machine controls the processor array, external input/output devices and generates the SIMD program.

The full state diagram for the Global Finite State Machine can be seen in Diagram 6.12.
GLOBAL FINITE STATE MACHINE

Finite State Machine to Provide GLOBAL Instruction Stream

INPUT SIGNAL LIST

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Function</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>finish_out</td>
<td>PE array Finished</td>
<td>First PE</td>
</tr>
<tr>
<td>PAD_start</td>
<td>Start Global Program</td>
<td>Input PAD</td>
</tr>
<tr>
<td>PAD_lead</td>
<td>Lead Chip in Array</td>
<td>Input PAD</td>
</tr>
</tbody>
</table>

INPUTS : finish_out PAD_start PAD_lead;

OUTPUT SIGNAL LIST

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Function</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAD_finish</td>
<td>PE array finished</td>
<td>Output PAD</td>
</tr>
<tr>
<td>global_in_4</td>
<td>Global Instr bit_4</td>
<td>First PE</td>
</tr>
<tr>
<td>global_in_3</td>
<td>Global Instr bit_3</td>
<td>First PE</td>
</tr>
<tr>
<td>global_in_2</td>
<td>Global Instr bit_2</td>
<td>First PE</td>
</tr>
<tr>
<td>global_in_1</td>
<td>Global Instr bit_1</td>
<td>First PE</td>
</tr>
<tr>
<td>global_in_0</td>
<td>Global Instr bit_0</td>
<td>First PE</td>
</tr>
<tr>
<td>sync_in</td>
<td>Synchronisation Signal</td>
<td>First PE</td>
</tr>
</tbody>
</table>

OUTPUTS : PAD_finish global_in_4 global_in_3 global_in_2 global_in_1 / global_in_0 sync_in;

FINITE STATE MACHINE DEFINITION

FSM CLOCK SIGNALS

Input latched on Clock_2
Output Available in Clock_1
RESET on Clock_1 (Inhibits FSM Output)
--SET_UP

--Operation:  RESET State
Reset PE array, the WAIT

SET_UP : GOTO WAIT (global_in_2 global_in_0 PAD_finish);
-- Send RESET instruction to array (00101)

--WAIT

--Operation:  Wait for array to be Ready for next Network Iteration
WAIT:  IF finish_out THEN FINISH (sync_in)
-- PE array ready for next Network Iteration
ELSE WAIT (sync_in PAD_finish);
-- Wait until PE array is ready for next Network Iteration

--INST_1  (11111)
--Operation  Start Next Synaptic Update (LD_WEIGHT)
Check if PE array has finished Network Iteration

INST_1 : CASE (finish_out PAD_lead)
  1 0 => FINISH (sync_in PAD_finish);
-- PE array has finished Network Iteration
  1 1 => FINISH (sync_in PAD_finish);
-- PE array has finished Network Iteration
  0 1 => INST_2 (global_in_4 global_in_3 global_in_2 global_in_1
   global_in_0 PAD_finish);
-- LD_WEIGHT Instruction (11111)
  ENDCASE => INST_1 (sync_in PAD_finish);
-- Wait for PAD_lead before starting synaptic update

--LD_INPUT  (00001)
INST_2 : GOTO INST_3 (global_in_0 sync_in PAD_finish);

--RESET_ACC  (01000)
INST_3 : GOTO INST_4 (global_in_3 sync_in PAD_finish);

--MULTI  (00011)
INST_4  : GOTO INST_5 (global_in_1 global_in_0 sync_in PAD_finish);
-MULTI (00011)

INST_5  : GOTO INST_6 (global_in_1 global_in_0 sync_in PAD_finish);
-MULTI (00011)

INST_6  : GOTO INST_7 (global_in_1 global_in_0 sync_in PAD_finish);
--MULTI (00011)

INST_7  : GOTO INST_8 (global_in_1 global_in_0 sync_in PAD_finish);
--MULTI (00011)

INST_8  : GOTO INST_9 (global_in_1 global_in_0 sync_in PAD_finish);
--MULTI (00011)

INST_9  : GOTO INST_10 (global_in_1 global_in_0 sync_in PAD_finish);
--MULTI (00011)

INST_10 : GOTO INST_11 (global_in_1 global_in_0 sync_in PAD_finish);
--MULTI (00011)

INST_11 : GOTO INST_12 (global_in_1 global_in_0 sync_in PAD_finish);
--MULTI (00100)

INST_12 : GOTO INST_13 (global_in_2 sync_in PAD_finish);
--ADD_PAR_LSW (00110)

INST_13 : GOTO INST_14 (global_in_2 global_in_1 sync_in PAD_finish);
--ADD_PAR_MSW (00111)

INST_14 : GOTO INST_1 (global_in_2 global_in_1 global_in_0 sync_in PAD_finish);
--FINISH

--Operation: Wait for Start Signal before next Synaptic Update

FINISH :IF PAD_start THEN INST_1 (sync_in PAD_finish)

-- Start Global Instruction Stream

    ELSE FINISH (sync_in) ;

-- Wait for External Start Signal
Appendix 3

Instruction Decoder

This appendix shows the source code for the Instruction Decoder. This code is written in a format that ESP can translate to .esp format. ESP was written so that .esp files can be expressed textually - and converted automatically - to reduce the number of errors that kept on recurring.

The Instruction Decoder is implemented in the Mealy style, this allows conditional operations to be executed without the need for complex branching mechanisms. Each instruction is included with all the relevant conditions; with different output signals being expressed for each set of conditions. An example is the MULTI instruction; this instruction requires eight different output expressions so that all the possible input conditions can be implemented.
PLA FORMAT INSTRUCTIONS

Number of Inputs
.i 11

Number of Outputs
.o 57

Number of Product Terms
.p 42

INPUT SIGNAL LIST

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Function</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>instr_0-4</td>
<td>Op-code</td>
<td>instr_mux</td>
</tr>
<tr>
<td>sh_lsb_b</td>
<td>LSBit Shift Reg (reg_sh)</td>
<td>data_path</td>
</tr>
<tr>
<td>msb_c_out_b</td>
<td>Carry Out</td>
<td>data_path</td>
</tr>
<tr>
<td>wgh_sgn</td>
<td>Sign of Weight</td>
<td>status</td>
</tr>
<tr>
<td>inp_sgn</td>
<td>Sign of Input Value</td>
<td>status</td>
</tr>
<tr>
<td>modulo</td>
<td>Modulo Flag</td>
<td>status</td>
</tr>
<tr>
<td>op_sgn</td>
<td>Sign of B Operand</td>
<td>data_path</td>
</tr>
</tbody>
</table>

.ilb inst_4 inst_3 inst_2 inst_1 inst_0 sh_lsb_b msb_c_out_b wgh_sgn \ inp_sgn modulo op_sgn

OUTPUT SIGNAL LIST

<table>
<thead>
<tr>
<th>Signals Name</th>
<th>Function</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>local_off</td>
<td>Turn Local Control Off</td>
<td>local</td>
</tr>
<tr>
<td>local_pla_on</td>
<td>Turn Local Control On</td>
<td>local</td>
</tr>
<tr>
<td>stop</td>
<td>End of Network Iteration</td>
<td>local</td>
</tr>
<tr>
<td>wgh_l_in</td>
<td>wgh_bus &lt;0-7&gt; =&gt; BUS_B &lt;0-7&gt;</td>
<td>data_path</td>
</tr>
<tr>
<td>wgh_h_in</td>
<td>wgh_bus &lt;0-7&gt; =&gt; BUS_B &lt;8-15&gt;</td>
<td>data_path</td>
</tr>
<tr>
<td>wgh_r</td>
<td>BUS_A &lt;0-15&gt; =&gt; wgh_bus &lt;0-15&gt;</td>
<td>data_path</td>
</tr>
<tr>
<td>Signals Name</td>
<td>Function</td>
<td>Destination</td>
</tr>
<tr>
<td>-------------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>PAD_wgh_oe, PAD_wgh_rw</td>
<td>wgh_bus_arb</td>
<td></td>
</tr>
<tr>
<td>PAD_inp_oe</td>
<td>inp_bus_arb</td>
<td></td>
</tr>
<tr>
<td>inp_out</td>
<td>inp</td>
<td></td>
</tr>
<tr>
<td>BUS_A &lt;0-15&gt; =&gt; inp_bus &lt;0-15&gt;</td>
<td>inp_bus_rw</td>
<td></td>
</tr>
<tr>
<td>PAD_inp_rw</td>
<td>inp_bus_arb</td>
<td></td>
</tr>
<tr>
<td>BUS_A &lt;0-15&gt; =&gt; sig_bus &lt;0-15&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>PAD_sig_rw</td>
<td>sig_bus_arb</td>
<td></td>
</tr>
<tr>
<td>BUS_B &lt;16-31&gt; =&gt; sig_bus &lt;16-31&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>reg_g &lt;0-15&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>BUS_B &lt;0-15&gt; =&gt; reg_g &lt;8-15&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>BUS_B &lt;0-7&gt; =&gt; reg_g &lt;0-7&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>reg_g &lt;0-15&gt; =&gt; BUS_A &lt;0-15&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>reg_g &lt;0-15&gt; =&gt; BUS_B &lt;0-15&gt;</td>
<td>data_path</td>
<td></td>
</tr>
</tbody>
</table>

** Registers reg_g, reg_f, reg_e, reg_d and reg_c as above **

** Register Designated Functions as Below : -

- reg_g: Modulo Input Pointer
- reg_f: Linear Input Pointer & Mod Step
- reg_e: Weight Pointer
- reg_d: MSWord Partial Sum
- reg_c: LSWord Partial Sum & Output
- reg_b: Multiplicand (9 bits)
- reg_a: Accumulator
- reg_sh: Multiplier (9 bits)

<table>
<thead>
<tr>
<th>Signals Name</th>
<th>Function</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUS_B &lt;0-7&gt; =&gt; reg_b &lt;0-8&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>(reg_sh &lt;8&gt; = reg_sh &lt;7&gt;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BUS_B &lt;0-8&gt; =&gt; reg_b &lt;0-8&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>reg_b &lt;0-8&gt; =&gt; BUS_B &lt;0-8&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>BUS_A &lt;8-15&gt; =&gt; reg_a &lt;8-15&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>BUS_A &lt;0-7&gt; =&gt; reg_a &lt;0-7&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>reg_a &lt;0-15&gt; =&gt; BUS_A &lt;0-15&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>reg_a &lt;0-6&gt; =&gt; BUS_B &lt;9-15&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>BUS_B &lt;0-7&gt; =&gt; reg_sh &lt;0-8&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>(reg_sh &lt;8&gt; = reg_sh &lt;7&gt;)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BUS_B &lt;0-8&gt; =&gt; reg_sh &lt;0-8&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>reg_sh &lt;0-8&gt; =&gt; BUS_B &lt;0-8&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>Double Length Shift</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>BUS_A &lt;0-15&gt; =&gt; adder_bus_a &lt;0-15&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>BUS_B &lt;0-15&gt; =&gt; adder_bus_b &lt;0-15&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>adder_sum &lt;0-15&gt; =&gt; BUS_C &lt;0-15&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>zero =&gt; adder_bus_a &lt;0-15&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>INV BUS_A =&gt; adder_bus_a &lt;0-15&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>zero =&gt; adder_bus_b &lt;0-15&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>INV BUS_B =&gt; adder_bus_b &lt;0-15&gt;</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>Enable Saturated Arithmetic</td>
<td>over</td>
<td></td>
</tr>
<tr>
<td>Enable Adder Output</td>
<td>over</td>
<td></td>
</tr>
<tr>
<td>Carry In Value</td>
<td>data_path</td>
<td></td>
</tr>
<tr>
<td>Double Shift Input Select : -</td>
<td>shift_a_mux</td>
<td></td>
</tr>
</tbody>
</table>

0 sum 9_b
1 c_out 9_b
<table>
<thead>
<tr>
<th>Signals Name</th>
<th>Function</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod_t+</td>
<td>Switch Modulo Status</td>
<td>Status Register</td>
</tr>
<tr>
<td>modLatch</td>
<td>Modulo Switch Latch</td>
<td>Status Register</td>
</tr>
</tbody>
</table>

```
.olb local_off local_pla_on stop wgh_l_in wgh_h_in wgh_r inp_oe \ 
inp_r par sigma g_w_a g_w_b_h g_w_b_l g_r_a g_r_b f_w_a f_w_b_h \ 
f_w_b_l f_r_a f_r_b e_w_a e_w_b_h e_w_b_l e_r_a e_r_b d_w_a \ 
d_w_b_h d_w_b_l d_r_a d_r_b c_w_a c_w_b_h c_w_b_l c_r_a c_r_b b_w_l \ 
b_w_b_r a_w_h a_w_l a_r a_r_ms sh_w_l sh_w sh_r mult tga 02a inva \ 
02b invb sat add_op carry shift_a_sel mod_t+ modLatch
```

### INSTRUCTION DECODE LOGIC

**LD_WEIGHT** Load Weight into reg_sh and reg_b

Increment Weight Pointer (reg_e)

**Opcode** 11111

**Conditions**

- **Clk1:**
  - Weight Pointer (reg_e) <0-15> -> BUS_A <0-15>
  - BUS_A <0-15> -> wgh_bus <0-15>
  - BUS_A <0-15> -> adder_bus_a <0-15>
  - ZERO -> adder_bus_b <0-15>
  - HIGH -> carry

- **Clk2:**
  - adder_sum <0-15> -> BUS_A <0-15>
  - BUS_A <0-15> -> Weight Pointer (reg_e) <0-15>
  - wgh_bus <0-7> -> BUS_B <0-7>
  - BUS_B <0-7> -> Multiplier (reg_sh) <0-8>
  - BUS_B <0-7> -> Multiplicand (reg_b) <0-8>

**LD_INPUT** Load Input into sh_reg OR reg_b

Increment Input Pointer

Input Pointer is dependent upon MODULO State:
- reg_f Modulo Off
- reg_g Modulo On

Destination Register dependent upon Weight Sign:
- reg_sh Weight Sign +Ve
- reg_b Weight Sign -Ve
Opcode 00001

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000-0</td>
<td>Modulo Off</td>
</tr>
<tr>
<td>0000-1</td>
<td>Modulo On</td>
</tr>
<tr>
<td>0000-0</td>
<td>Weight +Ve</td>
</tr>
<tr>
<td>0000-1</td>
<td>Weight -Ve</td>
</tr>
</tbody>
</table>

Clk1: If Modulo On
      Modulo Pointer (reg_g) <0-15> => BUS_A <0-15>

      If Modulo Off
      Linear Pointer (reg_f) <0-15> => BUS_A <0-15>

      BUS_A <0-15> => inp_bus <0-15>
      BUS_A <0-15> => adder_bus_a <0-15>
      ZERO => adder_bus_b <0-15>
      HIGH => carry

Clk2: adder_sum <0-15> => BUS_A <0-15>

      If Modulo On
      BUS_A <0-15> => Modulo Pointer (reg_g) <0-15>

      If Modulo Off
      BUS_A <0-15> => Linear Pointer (reg_f) <0-15>

      inp_bus <0-7> => BUS_B <0-7>

      If Weight +Ve
      BUS_B <0-7> => Multiplier (reg_sh) <0-8>

      If Weight -Ve
      BUS_B <0-7> => Multiplicand (reg_b) <0-8>

LDSIGMA Perform Sigma Function Lookup on Partial Sum

Opcode 00010

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000-0</td>
<td></td>
</tr>
<tr>
<td>0000-1</td>
<td></td>
</tr>
<tr>
<td>0000-0</td>
<td></td>
</tr>
<tr>
<td>0000-1</td>
<td></td>
</tr>
</tbody>
</table>

Clk1: Partial Sum LSByte (reg_c) <0-15> => BUS_A <0-15>

      Partial Sum MSByte (reg_b) <0-15> => BUS_B <0-15>

      BUS_A <0-15> => sig_bus <0-15>
      BUS_B <0-15> => sig_bus <16-31>

Clk2: sig_bus <0-7> => BUS_B <0-7>

      BUS_B <0-7> => Output (reg_c) <0-7>

! 00001--00-0- f_r_a tga 02b carry f_w_a add_op inp_oe inp_r sh_w_1
! 00001--00-1- g_r_a tga 02b carry g_w_a add_op inp_oe inp_r sh_w_1
! 00001--01-0- f_r_a tga 02b carry f_w_a add_op inp_oe inp_r b_w_1
! 00001--01-1- g_r_a tga 02b carry g_w_a add_op inp_oe inp_r b_w_1

! 00010------ c_r_a d_r_b c_w_b_l sigma

204
MULTI Multiply Iterate Instruction

Opcode 00011
Conditions --00-- Unsigned Multiplication
--01-- Mixed Multiplication
--11-- Mixed Multiplication
0------ Signed Multiplication
1------ Perform Conditional Addition

Clk1:
Multiplicand (reg_b) <0-8> => BUS_B <0-8>
Accumulator (reg_a) <0-8> => BUS_A <0-8>
Multiplier (reg_sh) <0-8> => Shift_Bus <0-8>
If No Addition (sh_lsb_b = 1)
Zero => adder_bus_b <0-15>
If Addition (sh_lsb_b = 0)
BUS_B <0-15> => adder_bus_b <0-15>
LOW => Carry

Clk2:
adder_sum <0-15> => Shift_BUS_A <0-15>
Shift Right BUS_A <l-8> => BUS_A <0-7>
If Unsigned Mult (wgh_sgn = 0 && inp_sgn = 0)
c_out_8 => BUS_A <8>
If Mixed Mult ((wgh_sgn = 0 && inp_sgn = 1) ||
(wgh_sgn = 1 && inp_sgn = 0))
c_out_8 => BUS_A <8>
If Signed Mult (wgh_sgn = 1 && inp_sgn = 1)
Adder_sum <8> => BUS_A <8>
BUS_A <0-8> => Accumulator (reg_a) <0-8>
Shift Right Shift_Bus <l-8> => Multiplier (reg_sh) <0-7>
Adder_sum <0> => Multiplier (reg_sh) <8>

! 000110-00-- a_r b_r tga mult a_w_h a_w_l add_op shift_a_sel
! 000111-00-- a_r b_r tga mult 02b a_w_h a_w_l add_op shift_a_sel
! 000110-10-- a_r b_r tga mult a_w_h a_w_l add_op shift_a_sel
! 000111-10-- a_r b_r tga mult 02b a_w_h a_w_l add_op shift_a_sel
! 000110-01-- a_r b_r tga mult a_w_h a_w_l add_op shift_a_sel
! 000111-01-- a_r b_r tga mult 02b a_w_h a_w_l add_op shift_a_sel
! 000110-11-- a_r b_r tga mult a_w_h a_w_l add_op
! 000111-11-- a_r b_r tga mult 02b a_w_h a_w_l add_op
MULTT  Multiply Terminate Instruction

Opcode  00100
Conditions --00--  Unsigned Multiplication
--10--  Mixed Multiplication
--01--  Mixed Multiplication
--11--  Signed Multiplication
0-----  Perform Conditional Addition
1-----  No Conditional Addition

Clk1:  Multiplicand (reg_b) <0-8> => BUS_B <0-8>
Accumulator (reg_a) <0-8> => BUS_A <0-8>
Multiplier (reg_sh) <0-8> => Shift_Bus <0-8>

If No Addition (sh_lsb_b = 1)
  Zero  => adder_bus_b <0-15>
  LOW  => carry
  (Unsigned Only)

If Addition (sh_lsb_b = 0)
  INV BUS_B <0-15> => adder_bus_b <0-15>
  HIGH  => carry
  (Mixed & Signed)

Clk2:  adder_sum <0-15> => Shift_BUS_A <0-15>
Shift Right Shift_BUS_A <l-8> => BUS_A <0-7>

If Unsigned Mul (wgh_sgn = 0 && inp_sgn = 0)
  c_out_8  => BUS_A <8>

If Mixed Mul (wgh_sgn = 0 && inp_sgn = 1 || wgh_sgn = 1 &&
  inp_sgn = 0)
  Adder_sum <8>  => BUS_A <8>

If Signed Mul (wgh_sgn = 1 && inp_sgn = 1)
  Adder_sum <8>  => BUS_A <8>

BUS_A <0-8>  => Accumulator (reg_a) <0-8>
Shift Right Shift_Bus <l-8>  => Multiplier (reg_sh) <0-7>
Adder_sum <0>  => Multiplier (reg_sh) <8>

! 001001-00-- a_r b_r tga mult 02b a_w_h a_w_l add_op shift_a_sel
! 001000-10-- a_r b_r tga mult invb carry a_w_h a_w_l add_op
! 001000-01-- a_r b_r tga mult invb carry a_w_h a_w_l add_op
! 001000-11-- a_r b_r tga mult invb carry a_w_h a_w_l add_op

RESET  Reset Processing Element
Load Zero Into Registers, Turn Modulo On
Turn Local Control On

Opcode  00101
Conditions ------

Clk1:  Zero  => adder_bus_a
Zero  => adder_bus_b
LOW  => Carry
Modulo On
Local On
Clk2: adder_sum \Rightarrow BUS_A
BUS_A \Rightarrow \text{All Registers}

! 00101-\ldots 02a 02b \text{tga a_w_h a_w_l c_w_a d_w_a e_w_a f_w_a g_w_a} \\
add_op \text{ mod_t+ \ mod_latch \ local_pla_on}

ADD\_PAR\_LSW \quad \text{Add Partial Sum LSWord}

Opcode 00110
Conditions---------

Clk1: Product LSWord (reg_sh) <0-8> \Rightarrow BUS_B <0-8>
Product MSWord (reg_a) <0-6> \Rightarrow BUS_B <9-15>
Partial Sum LSWord (reg_c) <0-15> \Rightarrow BUS_A <0-15>
BUS_A <0-15> \Rightarrow \text{adder_bus_a} <0-15>
BUS_B <0-15> \Rightarrow \text{adder_bus_b} <0-15>
LOW \Rightarrow \text{carry}

Clk2: adder_sum <0-15> \Rightarrow BUS_A <0-15>
BUS_A <0-15> \Rightarrow \text{Partial Sum LSWord (reg_c)} <0-15>

! 00110-\ldots \text{sh_r a_r_ms c_r_a tga c_w_a add_op}

ADD\_PAR\_MSW \quad \text{Add Partial Sum MSWord}

Opcode 00111
Conditions-1-0 \quad \text{No Carry, Positive}
-1-1 \quad \text{No Carry, Negative}
-0-0 \quad \text{Carry, Positive}
-0-1 \quad \text{Carry, Negative}

Clk1: Partial Sum MSWord (reg_d) <0-15> \Rightarrow BUS_B <0-15>
Zero \Rightarrow \text{adder_bus_a} <0-15>
BUS_B <0-15> \Rightarrow \text{adder_bus_b} <0-15>

If Carry Out is Low (msb_c_out_b = 1)
LOW \Rightarrow \text{Carry}

If Carry Out is High (msb_c_out_b = 0)
HIGH \Rightarrow \text{Carry}

If Product is +Ve (op_sgn = 0)
ZERO \Rightarrow BUS_A <0-15>

If Product is -Ve (op_sgn = 1)
ZERO \Rightarrow BUS_A <0-15>
INV BUS_A \Rightarrow \text{adder_bus_b} <0-15>

Enable Saturated Arithmetic Output

Clk2: adder_sum <0-15> \Rightarrow BUS_A <0-15>
BUS_A <0-15> \Rightarrow \text{Partial Sum MSWord (reg_d)} <0-15>
ACC_RESET Reset Accumulator

Opcode 01000
Conditions ------

Clk1: Zero => adder_bus_a <0-15>
   Zero => adder_bus_b <0-15>
   LOW => Carry

Clk2: adder_sum <0-15> => BUS_A <0-15>
   BUS_A <0-15> => Accumulator (reg_a) <0-15>

LOCAL_OFF Turn Local Control Off

Opcode 01001
Conditions ------

Clk1: Turn Local Control Off

Clk2: No Operations

STOP Stop at end of Network

Opcode 01010
Conditions ------

Clk1: Turn Local Control Off
   Stop Executing Global Control

Clk2: No Operations

BIAS_LLSB Load Bias LSByte into reg_c (1)
Increment Weight Pointer

Opcode 01011
Conditions ------

Clk1: Weight Pointer (reg_e) <0-15> => BUS_A <0-15>
   BUS_A <0-15> => wgh_bus <0-15>
   BUS_A <0-15> => adder_bus_a <0-15>
   ZERO => adder_bus_b <0-15>
   HIGH => carry
Clk2: adder_sum <0-15> => BUS_A <0-15>
BUS_A <0-15> => Weight Pointer (reg_e) <0-15>
wgh_bus <0-7> => BUS_B <0-7>
BUS_B <0-7> => Partial Sum LSWord (reg_c) <0-7>

! 01011------ e_r_a tga 02b carry e_w_a add_op wgh_r wgh_l_in c_w_b l

BIAS_HLSB Load Bias MSByte into reg_d (h)
Increment Weight Pointer

Opcode 01100
Conditions------

Clk1: Weight Pointer (reg_e) <0-15> => BUS_A <0-15>
BUS_A <0-15> => wgh_bus <0-15>
BUS_A <0-15> => adder_bus_a <0-15>
ZERO => adder_bus_b <0-15>
HIGH => carry

Clk2: adder_sum <0-15> => BUS_A <0-15>
BUS_A <0-15> => Weight Pointer (reg_e) <0-15>
wgh_bus <0-7> => BUS_B <8-15>
BUS_B <8-15> => Partial Sum LSWord (reg_c) <8-15>

! 01100------ e_r_a tga 02b carry e_w_a add_op wgh_r wgh_h_in c_w_b_h

BIAS_LMSB Load Bias LSByte into reg_c (l)
Increment Weight Pointer

Opcode 01101
Conditions------

Clk1: Weight Pointer (reg_e) <0-15> => BUS_A <0-15>
BUS_A <0-15> => wgh_bus <0-15>
BUS_A <0-15> => adder_bus_a <0-15>
ZERO => adder_bus_b <0-15>
HIGH => carry

Clk2: adder_sum <0-15> => BUS_A <0-15>
BUS_A <0-15> => Weight Pointer (reg_e) <0-15>
wgh_bus <0-7> => BUS_B <0-7>
BUS_B <0-7> => Partial Sum LSWord (reg_d) <0-7>

! 01101------ e_r_a tga 02b carry e_w_a add_op wgh_r wgh_l_in d_w_b_l

BIAS_HMSB Load Bias MSByte into reg_d
Increment Weight Pointer

Opcode 01110
Conditions------
Clk1: Weight Pointer (reg_e) <0-15> => BUS_A <0-15>
    BUS_A <0-15> => wgh_bus <0-15>
    BUS_A <0-15> => adder_bus_a <0-15>
    ZERO => adder_bus_b <0-15>
    HIGH => carry

Clk2: adder_sum <0-15> => BUS_A <0-15>
    BUS_A <0-15> => Weight Pointer (reg_e) <0-15>
    wgh_bus <0-7> => BUS_B <8-15>
    BUS_B <8-15> => Partial Sum MSWord (reg_d) <8-15>

! 01110------ e_r_a tga 02b carry e_w_a add_op wgh_r wgh_h_in d_w_b_h

MODULO ON Modulo Input Address Generation On

Opcode 10000
Conditions------

Clk1: Turn Modulo On
Clk2: No Operations

! 10000------ mod_t+ mod_latch

MODULO Off Modulo Input Address Generation Off

Opcode 10001
Conditions------

Clk1: Turn Modulo Off
Clk2: No Operations

! 10001------ mod_latch

LDPAR_LSW Load Partial Sum LSWord

Opcode 10010
Conditions------

Clk1: Partial Sum LSWord (reg_c) <0-15> => BUS_A <0-15>
    sig_bus <0-15> => BUS_B <0-15>
    BUS_A <0-15> => adder_bus_a <0-15>
    BUS_B <0-15> => adder_bus_b <0-15>
    LOW => Carry

Clk2: adder_sum <0-15> => BUS_A <0-15>
    BUS_A <0-15> => Partial Sum LSWord (reg_c) <0-15>

! 10010------ par c_r_a tga add_op c_w_a
LDPAR_MSW Load Partial Sura MSWord

Opcode 10011
Conditions-0----  No Carry
             -1----  Carry

Clk1: Partial Sum MSWord (reg_d) <0-15> => BUS_A <0-15>
sig_bus <0-15> => BUS_B <0-15>
BUS_A <0-15> => adder_bus_a <0-15>
BUS_B <0-15> => adder_bus_b <0-15>

If Carry Out is Low (msb_c_out_b = 1)
   LOW => Carry
If Carry Out is High (msb_c_out_b = 0)
   HIGH => Carry

Enable Saturated Arithmetic

Clk2: adder_sum <0-15> => BUS_A <0-15>
BUS_A <0-15> => Partial Sum MSWord (reg_d) <0-15>

! 10011-0---- par_d_r_a_tga_add_op_sat_d_w_a
! 10011-1---- par_d_r_a_tga_add_op_carry_sat_d_w_a

INPUT_PTRL Load Linear Pointer (LSByte) into reg_f
Increment Weight Pointer

Opcode 10100
Conditions------

Clk1: Weight Pointer (reg_e) <0-15> => BUS_A <0-15>
BUS_A <0-15> => wgh_bus <0-15>
BUS_A <0-15> => adder_bus_a <0-15>
ZERO => adder_bus_b <0-15>
HIGH => carry

Clk2: adder_sum <0-15> => BUS_A <0-15>
BUS_A <0-15> => Weight Pointer (reg_e) <0-15>
wgh_bus <0-7> => BUS_B <0-7>
BUS_B <0-7> => Linear Input Pointer (reg_f) <0-7>

! 10100------ e_r_a_tga_02b_carry_e_w_a_add_op_wgh_r_wgh_l_in_f_w_b_l

INPUT_PTRH Load Linear Pointer (MSByte) into reg_f
Increment Weight Pointer

Opcode 10101
Conditions------
Clk1: Weight Pointer (reg_e) <0-15> => BUS_A <0-15>
     BUS_A <0-15> => wgh_bus <0-15>
     BUS_A <0-15> => adder_bus_a <0-15>
     ZERO => adder_bus_b <0-15>
     HIGH => carry

Clk2: adder_sum <0-15> => BUS_A <0-15>
     BUS_A <0-15> => Weight Pointer (reg_e) <0-15>
     wgh_bus <0-7> => BUS_B <8-15>
     BUS_B <8-15> => Linear Input Pointer (reg_f) <8-15>

! 10101------ e_r_a tga 02b carry e_w_a add_op wgh_r wgh_h_in f_w_b_h

ADD_MOD_STEP Add Modulo Step (reg_f) to Modulo Input Pointer

Opcode 10110
Conditions------

Clk1: Linear Input Pointer (reg_f) <0-15> => BUS_B <0-15>
     Modulo Input Pointer (reg_g) <0-15> => BUS_A <0-15>
     BUS_A <0-15> => adder_bus_a <0-15>
     BUS_B <0-15> => adder_bus_b <0-15>
     LOW => carry

Clk2: adder_sum <0-15> => BUS_A <0-15>
     BUS_A <0-15> => Modulo Input Pointer (reg_g) <0-15>

! 10110------ f_r_a g_r_b tga add_op g_w_a

OUTPUT Output Neuron Result to Input Valve Table

Opcode 10111
Conditions------

Clk1: Linear Input Pointer (reg_f) <0-15> => BUS_A <0-15>
     BUS_A <0-15> => Input Bus <0-15>
     Output (reg_c) <0-8> => BUS_B <0-8>
     Zero => adder_bus_a <0-15>
     BUS_B <0-15> => adder_bus_b <0-15>
     LOW => carry

Clk2: adder_sum <0-15> => BUS_A <0-15>
     BUS_A <0-15> => Input Bus <0-15>

! 10111------ f_r_a c_r_b inp_oe 02a tga add_op

RETURN Jump to Weight Zero

Opcode 11000
Conditions------

Clk1: Zero => adder_bus_a <0-15>
     Zero => adder_bus_b <0-15>
     LOW => Carry

Clk2: adder_sum <0-15> => BUS_A <0-15>
     BUS_A <0-15> => Weight Pointer (reg_e) <0-15>

---------------------------

212
NOP No Operation

Opcode 00000
Conditions-----

Clk1: No Operations
Clk2: No Operations

! 00000------
Appendix 4

Netlist Generation

This appendix shows the three program scripts for the Netlist Generation. These three programs are:

• **NET** *(CSHELL & AWK Script)*
  NET extracts all sub-cell instances from the top level cell, and outputs these in a list in the following format:

  gawk -f labels data_path.mag v=data_path_0

  This list of sub-cell instances is then converted to an executable file and executed.

• **LABELS** *(AWK Script)*
  LABELS takes a sub_cell name and instance number, and generates a list of terminals in MAGIC hierarchical format:

  data_path_0/msb_sum_b

  This list of terminals is then used by NETLIST to generate a single Netlist.

• **NETLIST** *(AWK Script)*
  NETLIST takes a list of terminals, and a Netlist File that is to be merged, and creates a single Netlist File. The algorithm works by matching all terminals with the same label:

  ie. the following terminals will be made into a Net

  clock_0/clk1
  latch_0/clk1
  instr_mux_0/clk1

  This netlist is then merged with the secondary Netlist to generate a single Netlist that can be used by the MAGIC Router.
# cshell and AWK Script to Generate Netlist from *.mag files

# Use by net ARGV[1] ARGV[2]

# ARGV[1] Top level cell to be Routed
# ARGV[2] Netlist to merge with created Netlist

# ARGV[2] to be in standard netlist format, without the first line:
# Program operates by extracting all sub_cell names from ARGV[1].mag
# All terminal names are then extracted from subcells.
# Finally all subcell terminals are matched to generate a Netlist File

# PROGRAM START

echo " Netlist Generation"

# Use gawk to extract all sub_cells from ARGV[1].mag
# The magic command 'use' indicates a subcell call and the instance number:

# use data_path data_path_0

gawk '/use/ {print "gawk -f labels " $2 ".mag v=" $3 }' $1.mag > sub_cells

# This creates a file containing all subcell calls.
# The format of this file is as:

# gawk -f labels data_path.mag v=data_path_0

# This then allow each subcell .mag file named to be processed.
# The v argument is used to pass the instance number for the call

# First convert file into an executable file

chmod +x sub_cells

# Execute file, catingating all outputs to ARGV[1].labels

sub_cells > $1.labels

# Finally create Netlist File from all terminals extracted

gawk -f netlist $1.labels $2.net > $1.net
# AWK Script to Extract Terminal Names from .mag File

# Use by gawk -f labels ARGV[1] v=ARGV[2]
# Program called by cshell script net
# ARGV[1] Magic file to use as source
# ARGV[2] Instance name of subcell call:
#ie data_path_0
# Program operates by scanning magic file for terminals.
# GND and Vdd terminals are ignored.
# Output in hierachical form for terminal:
#ie data_path_0/msb_sum_b

# PROGRAM START

# First split ARGV[2] to obtain Instance name
BEGIN {  split (ARGV[2], cell, "="))
# Scan magic file for all labels:
#ie rlabel polycontact -1234 118 -1230 118 5 msb_sum_b
# Ignore GND, Vdd and blank labels
# Print any terminals found - in hierachical format
$1 ~/label/ && $8 !~/GND/ && $8 !~/Vdd/ && $8 ~/./ {print cell[2] "/" $8 }
AWK Script to Generate Netlists


Program called by cshell script ‘net’

ARGV[1] List of labels in Top Level sub_cells
ARGV[2] Netlist to merge with Generated Netlist

ARGV[1] should be in the following format:

clock_0/clock1
clock_0/clock2
clock_0/clk1

AWK Script ‘labels’ can be used to extract terminals from sub_cells

ARGV[2] must be in following format:

clock_0/clock1
decode_0/reset_c
decode_0/latch_c

This net will be merged with net clock_0/clock1

Program operates by matching all terminals with the same last name
ie. the following terminals will be made into a Net

clock_0/clk1
latch_0/clk1
instr_mux_0/clk1

After generating a net, any nets in argv2 will be merged if necessary

PROGRAM START

Print Format Line at Start of Netlist File
Magic uses / as hierchay separator in terminal names

BEGIN { print "Netlist File"
    print ""
    FS = "\""
    OFS = "\"
}
Start by reading both argv1 & argv2 into different arrays

# 2 dimensional arrays required to split terminal names:
# array[ ,1] = Cell instance
# array[ ,2] = Label Name

END {

# Read argv1 into array1

while ( getline x <(ARGV[1]) == 1 ) {
  split (x, node)
  array1[++i,2] = node[2]
  array1[i,1] = node[1]
}

# Read argv2 into array2

while ( getline x <(ARGV[2]) == 1 ) {
  split (x, node)
  array2[++k,2] = node[2]
  array2[k,1] = node[1]
}

# set imax and kmax to size of arrays
imax = ++i
kmax = ++k

# Loop to create Nets from each terminal in array1
for (i = 0; i < imax; i++) {
  # If next terminal has not been set to " " then use it
  if ( array1[i,1] != " "){
    flag = 0
    # Search rest of array1 for matching labels
    # If match found add terminal to net and set to " 
    for (j = (i+1); j != imax; j++){
      if (( array1[i,2] == array1[j,2])
          && (array1[i,2] != "")){
        print array1[j,1], array1[j,2]
        array1[j,1] = " 
        flag = 1
      }
    }
  }
}
# If a net was formed add original terminal to net and merge net with any matching net in array2

if ( flag == 1){
    print array1 [i,1], array1 [i,2]
    for (k = 0; k < kmax; k++)(
        # If a net is found to be merged
        # catinate the two net
        if ( array1 [i,2] == array2 [k,2] ){
            array2 [k,1] = ""
            # Print each terminal in net
            while ( length (array2 [k,2] ) > 1){
                print array2 [k,1], array2[k,2]
                array2 [k,1] = ""
            }  
        }
    }  
    # End each Net with a blank line
    print " "
}
}

# At end of Netlist Generation print all un-merged Net from array2
for ( k=0;  k<  kmax; k++){
    # Search for Nets in array2
    if ( array2 [k,1] != ""){
        print array2 [k,1], array2 [k,2]  
    }
    # End each Net with a blank line
    if ( array2 [k,2] == " " ){
        print ""
    }
}
}
Appendix 5

ESP Program Listing

This program listing in AWK is for the ESP utility. This program converts a textual description of a PLA into the format that MPLA uses. The algorithm uses a simple textual replacements to generate the personality matrix.
AWK Script to Generate Personality Matrix from Decode

Use by gawk -f esp ARGV[1] > file_name.esp

Program called from cshell script make_pla in directory ~/magic/decode

ARGV[1] to contain text description of PLA

PLA Format Instructions to be in Standard Format, these are passed
to output file unchanged.

Each Product term to be in format as below:

! <input_pattern> <output_signal_list>

where

<input_pattern> is in standard format ie. 00010-----

<output_signal_list> Signals to be set high on input pattern

Program works by checking each term in the .olb list (PLA Outputs)
and if this is matched in output_signal_list print a 1, else 0.

Finally a check is made that all output_signals were recognised.

PROGRAM START

No spaces in Personality Matrix between records

BEGIN {
    ORS = ""
}

Pass all PLA Format Instruction unchanged

\./ { print $0 "\n"}

Read all Output Signals into node[]

\./olb/ { split ($0, node) }

Read Product Term into instr[] and Output as Personality Matrix

\!/ { split ($0, instr) }

Print Input Pattern Unchanged

print $2 " "

For each PLA Output Signal check if in output_signal_list

------------------------
count = 0
for (c = 2; node [c] != ""; c++ ){
    flag = 0
    for (cc = 3; instr [cc] != ""; cc++) {
        # Check if Output Signal is in output_signal_list
        if (node [c] == instr [cc]) {
            # If in output_signal_list set to 1 and inc count
            flag = 1
            ++ count
        }
    }
    # Output status of Signal (1 if in output_signal_list
    # 0 if NOT in output_signal_list)
    print flag
}

# Finally check that the number of PLA Output Signals set to high
# equals length of output_signal_list
if (count != (NF-2)) {
    print \n NF " " count "Node not recognised"
    print \n
}

# Print .end at end of Personality Matrix
END {
    print ".end"
}
Appendix 6

Pad Frame Request File

This file is used by the pad generation facility to determine the pad frame layout. Each pin is defined as a pad library cell - or by NOBOND to indicate a pin that is not connected.

This file was supplied to the University of London VLSI Consortium, who then created a blank pad frame so that routing could be performed. Upon completion this blank frame was replaced by a real pad frame and the chip core was inserted. This last function was done outside MAGIC using the GIF format for both the pad frame and the chip core.
FRAME REQUEST FILE - 84 PIN PACK -

FRAME REQUEST FOR RICHARD PALMER, 22th JANUARY 1991

Neural Processing Chip

Package Size: 84 Pins

Pins Used: 78 Pins:-

11 Power (6 vccs, 5 vsss)
68 Signal

Pad cells used:

ioslp Bi-Directional Tristate
ips8g Non inverting Input
opslu Inverting Output
opslw Tristate Output
vss GROUND
vcc 5 Volts
NOBOND Not Connected

Pan Frame: 19 Pins (Top & Bottom) x 20 Pins (Sides)

Internal Size: 4300um x 3230um

Chip Size: 4184um x 4354um

Chip Area 18.22 mm2

BEGIN

Fabricator ES2
Technology 2um

Route CMP

Package 84
Pads 78

CoreX 2970 in microns
CoreY 3200 in microns
### RHS from bottom to top

<table>
<thead>
<tr>
<th>NOBOND</th>
<th>pin 75</th>
<th>NC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ioslp</td>
<td>pin 76</td>
<td>wgh_2</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 77</td>
<td>wgh_3</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 78</td>
<td>wgh_4</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 79</td>
<td>wgh_5</td>
</tr>
<tr>
<td>vccs</td>
<td>pin 80</td>
<td>5 Volts</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 81</td>
<td>wgh_6</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 82</td>
<td>wgh_7</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 83</td>
<td>wgh_8</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 84</td>
<td>wgh_9</td>
</tr>
<tr>
<td>vss</td>
<td>pin 1</td>
<td>GROUND</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 2</td>
<td>wgh_10</td>
</tr>
<tr>
<td>vccs</td>
<td>pin 3</td>
<td>5 Volts</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 4</td>
<td>wgh_11</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 5</td>
<td>wgh_12</td>
</tr>
<tr>
<td>vccs</td>
<td>pin 6</td>
<td>5 Volts</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 7</td>
<td>wgh_13</td>
</tr>
<tr>
<td>vss</td>
<td>pin 8</td>
<td>GROUND</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 9</td>
<td>wgh_14</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 10</td>
<td>wgh_15</td>
</tr>
<tr>
<td>vccs</td>
<td>pin 11</td>
<td>5 Volts</td>
</tr>
</tbody>
</table>

### TOP from right to left

<table>
<thead>
<tr>
<th>NOBOND</th>
<th>pin 12</th>
<th>NC</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOBOND</td>
<td>pin 13</td>
<td>NC</td>
</tr>
<tr>
<td>ips8g</td>
<td>pin 14</td>
<td>lead</td>
</tr>
<tr>
<td>opslu</td>
<td>pin 15</td>
<td>finish</td>
</tr>
<tr>
<td>ips8g</td>
<td>pin 16</td>
<td>start</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 17</td>
<td>inp_15</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 18</td>
<td>inp_14</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 19</td>
<td>inp_13</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 20</td>
<td>inp_12</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 21</td>
<td>inp_11</td>
</tr>
<tr>
<td>ips8g</td>
<td>pin 22</td>
<td>mod_11</td>
</tr>
<tr>
<td>vss</td>
<td>pin 23</td>
<td>GROUND</td>
</tr>
<tr>
<td>vccs</td>
<td>pin 24</td>
<td>5 Volts</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 25</td>
<td>inp_10</td>
</tr>
<tr>
<td>ips8g</td>
<td>pin 26</td>
<td>mod_10</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 27</td>
<td>inp_9</td>
</tr>
<tr>
<td>ips8g</td>
<td>pin 28</td>
<td>mod_9</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 29</td>
<td>inp_8</td>
</tr>
<tr>
<td>ips8g</td>
<td>pin 30</td>
<td>mod_8</td>
</tr>
<tr>
<td>vss</td>
<td>pin 31</td>
<td>GROUND</td>
</tr>
<tr>
<td>ioslp</td>
<td>pin 32</td>
<td>inp_7</td>
</tr>
<tr>
<td>LHS from top to bottom</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nobond</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ips8g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ioslp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vccs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ioslp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ioslp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ioslp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>opsiu</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ips8g</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ioslp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ioslp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ioslp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vsss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ioslp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ioslp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ioslp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BOTTOM from left to right</th>
</tr>
</thead>
<tbody>
<tr>
<td>nobond</td>
</tr>
<tr>
<td>nobond</td>
</tr>
<tr>
<td>ioslp</td>
</tr>
<tr>
<td>ioslp</td>
</tr>
<tr>
<td>ioslp</td>
</tr>
<tr>
<td>vsss</td>
</tr>
<tr>
<td>ioslp</td>
</tr>
<tr>
<td>ioslp</td>
</tr>
<tr>
<td>ops1w</td>
</tr>
<tr>
<td>ops1w</td>
</tr>
<tr>
<td>ops1w</td>
</tr>
<tr>
<td>ops1w</td>
</tr>
<tr>
<td>ops1w</td>
</tr>
<tr>
<td>END</td>
</tr>
</tbody>
</table>
Appendix 7

Performance Analysis

This appendix carries out a detailed analysis of the performance of this architecture. This analysis considers three typical network mappings and calculates the proportions of time spent under SIMD control, MIMD control and in an idle state. From this analysis it is possible to estimate the average performance obtainable from this architecture.

This analysis is performed by breaking down the execution of each virtual neuron in a network mapping. This calculation uses units of 14 cycles (ie the number of equivalent SIMD program cycles) to simplify the arithmetic. Thus each virtual neuron is composed of:

1 Unit MIMD Control:  BIAS_LLSW
                        BIAS_HLSW
                        BIAS_LMSW
                        BIAS_HMSW
                        INPUT_PTR_L
                        INPUT_PTR_H
                        LOCAL_OFF

SIMD Control: Number of units dependent upon network mapping

1 Unit MIMD Control:  ESCAPE
                        LD_SIGMA
                        INPUT_PTR_L
                        INPUT_PTR_H
                        OUTPUT

The number of cycles spent in each control type, and the number of NOP instructions, are then calculated for the entire network. This enables the proportion of time spent in either type of control to be obtained, thus allowing the percentage of peak performance to be calculated for each network mapping.

The three network mapping used for this analysis are the same as those presented in Section 5.6.3
In this network mapping six processors are used in a non-pipelined manner. PE 1 implements a 1st Hidden Layer neuron, a 2nd Hidden Layer neuron and the Output Layer neuron. PEs 2-6 implement only a 1st Hidden Layer neuron and a 2nd Hidden Layer neuron.

### Network Mapping PE 1

<table>
<thead>
<tr>
<th></th>
<th>1st Hidden Layer</th>
<th>2nd Hidden Layer</th>
<th>Output Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping</td>
<td>246</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

### Network Mapping PEs 2-6

<table>
<thead>
<tr>
<th></th>
<th>1st Hidden Layer</th>
<th>2nd Hidden Layer</th>
<th>Output Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping</td>
<td>246</td>
<td>6</td>
<td>Idle</td>
</tr>
</tbody>
</table>

### Breakdown of Network Mapping

#### PE 1

- 1: MIMD
- 246: SIMD
- 1: MIMD
- 6: SIMD
- 1: MIMD
- 1: MIMD
- 6: SIMD
- 1: MIMD
- 1: MIMD
- 6: SIMD
- 1: MIMD

#### PEs 2-6

- 1: MIMD
- 246: SIMD
- 1: MIMD
- 6: SIMD
- 1: MIMD
- 1: NOP
- 6: NOP
- 1: NOP

### Total:

- 258 + 5(252) SIMD = 1518 = 95.8%
- 6 + 5(4) MIMD = 26 => 1.6%
- 0 + 5(8) NOP = 40 => 2.5%

**Percentage of Peak Performance:** 95.8%
Reduced Sample Time

In this arrangement 13 processors are used, these are arranged to optimise the sample period. PEs 1-12 implement a 1st Hidden Layer neuron. PE 13 implements all 6 2nd Hidden Layer neurons and the Output Layer neuron.

In this network mapping addition MIMD cycles are required for the partial sum accumulation for PEs 1-12. This action requires 12 additional cycles (each transfer requires two cycles for the LSWord and MSWord).

Network Mapping PEs 1-12

<table>
<thead>
<tr>
<th>Mapping</th>
<th>1st Hidden Layer</th>
<th>2nd Hidden Layer</th>
<th>Output Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Network Mapping PE 13

<table>
<thead>
<tr>
<th>Mapping</th>
<th>1st Hidden Layer</th>
<th>2nd Hidden Layer</th>
<th>Output Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>36</td>
<td>+</td>
<td>6</td>
</tr>
</tbody>
</table>

Breakdown of Network Mapping

<table>
<thead>
<tr>
<th>PEs 1-12</th>
<th>PE 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>123</td>
<td>36</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PEs 1-12</th>
<th>PE 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>80</td>
</tr>
</tbody>
</table>

Total:

<table>
<thead>
<tr>
<th>PEs 1-12</th>
<th>= 1518</th>
<th>=92.6%</th>
</tr>
</thead>
<tbody>
<tr>
<td>12(123) + 42</td>
<td>SIMD</td>
<td>1518</td>
</tr>
<tr>
<td>12(3) + 4</td>
<td>MIMD</td>
<td>40</td>
</tr>
<tr>
<td>0 + 80</td>
<td>NOP</td>
<td>80</td>
</tr>
</tbody>
</table>

Percentage of Peak Performance: 93.4%
Reduced Response Time

In this arrangement 12 processors are used in a non-pipelined manner. Even numbered PEs implement $\frac{1}{2}$ a 1st Hidden Layer neuron, $\frac{1}{2}$ a 2nd Hidden Layer neuron and $\frac{1}{6}$ of the Output Layer neuron. Odd numbered PEs implement $\frac{1}{2}$ a 1st Hidden Layer neuron and $\frac{1}{2}$ a 2nd Hidden Layer neuron.

In this network mapping addition MIMD cycles are required for the partial sum accumulation for even numbered PEs. This action requires 20 additional cycles (each transfer requires two cycles for the LSWord and MSWord).

Network Mapping Even Numbered PEs

<table>
<thead>
<tr>
<th></th>
<th>1st Hidden Layer</th>
<th>2nd Hidden Layer</th>
<th>Output Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping</td>
<td>123</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Network Mapping Odd Numbered PEs

<table>
<thead>
<tr>
<th></th>
<th>1st Hidden Layer</th>
<th>2nd Hidden Layer</th>
<th>Output Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping</td>
<td>123</td>
<td>3</td>
<td>Idle</td>
</tr>
</tbody>
</table>

Breakdown of Network Mapping

<table>
<thead>
<tr>
<th>PE Even Nos.</th>
<th>PE Odd Nos.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>123 SIMD</td>
<td>123 SIMD</td>
</tr>
<tr>
<td>1 MIMD</td>
<td>1 MIMD</td>
</tr>
<tr>
<td>1 MIMD</td>
<td>1 MIMD</td>
</tr>
<tr>
<td>1 MIMD</td>
<td>1 MIMD</td>
</tr>
<tr>
<td>1 SIMD</td>
<td>1 SIMD</td>
</tr>
<tr>
<td>1 SIMD</td>
<td>1 SIMD</td>
</tr>
<tr>
<td>1 MIMD</td>
<td>1 MIMD</td>
</tr>
<tr>
<td>1 SIMD</td>
<td>1 SIMD</td>
</tr>
<tr>
<td>2 MIMD</td>
<td>2 MIMD</td>
</tr>
</tbody>
</table>

Total:

- $6(127) + 6(126)$ SIMD = 1518 = 94.4%
- $6(7) + 6(4)$ MIMD = 66 = 4.1%
- $0 + 6(4)$ NOP = 24 = 1.5%

Percentage of Peak Performance: 94.4%
Published Literature

This appendix includes all the literature published that relates to this thesis. This includes two science articles from the TIMES Newspaper, and a Conference Proceedings.

R. Matthews *More research cash for computers that imitate brain*
TIMES Newspaper

P Write *NHS ear implant offer to totally deaf* TIMES Newspaper

More research cash for computers that imitate brain

By Robert Matthews, Technology Correspondent

The Government is about to announce substantial funding for research into computer science, including work on designing so-called "neural" computers which mimic the workings of the human mind.

The move, which emerged in London yesterday at an international conference on neurocomputing, is being seen as Britain's attempt to catch the United States and Japan in the technology likely to dominate the coming decades.

Neurocomputers are designed to carry out tasks, such as recognizing faces and exercising judgement in financial matters, which humans still do better than the fastest conventional computer.

That is because neurocomputers can solve many small problems at the same time, while conventional computers have to deal with one problem at a time.

The funding is to come from two separate sources. First, the Medical Research Council, the Science and Engineering Research Council and the Economic and Social Research Council are joining in a five-year, £12 million programme of research into how computers can help to unravel the mysteries of learning and of the brain.

Neurocomputers will be built to try to understand how children acquire language. Other research will look at how neurocomputers can directly help humans.

Scientists at University College London are working on a revolutionary hearing aid which incorporates a simple, tiny neurocomputing microchip capable of recognizing speech and making it more intelligible.

Dr Megan Davies, of the Medical Research Council, which is co-ordinating the programme, said yesterday: "All the research councils see this as an arena which is of considerable academic interest and an obvious area for exploitation".

Universities and polytechnics have a month to apply for the first round of funds.

Money is also to be provided by the Department of Trade and Industry and the Science and Engineering Research Council for a research programme on the construction of neurocomputers.

One project which has already benefited from department funding is a machine, built at Imperial College London, which can memorize different shapes and faces and pick them out from others in about 1/25th of a second. An early version is in commercial production.

The Department of Trade and Industry believes that British scientists can develop many more such machines able to compete with those from the US and Japan.

The conference was told that after years in the academic doldrums, neurocomputing is becoming a commercially viable technology in many fields, from diagnosing heart problems from cardiac measurements to predicting currency fluctuations. One American company is using a neurocomputer to make trading decisions on the New York Stock Exchange.

Neurocomputing Ltd, a small Winchester-based company, last night announced that it had become the first British company to import such systems for distribution in the UK. A number of insurance companies are using the system.
NHS ear implant offer to totally deaf

By Pearce Wright
Science Editor

Bionic ear implants that will provide a special type of hearing aid for totally deaf people are to become available through the National Health Service.

The Department of Health said yesterday it was allocating £1 million a year over the next three years to establish six centres to provide "cochlear" implants.

Mr Roger Freeman, Under-Secretary of State at the Department of Health, said proposals for implant programmes would be invited from health authorities. About 5,000 profoundly deaf people in Britain could benefit.

The Royal National Institute for the Deaf (RNID), which has funded most of the UK development of implant surgery at University College hospital, London, Addenbrookes Hospital, Cambridge, Guy's hospital and the London hospital said it was "delighted". Miss Hilary Pearce, of the RNID, said: "It does not restore hearing to the levels at which you and I hear but it helps to restore the sense of sound."

Cochlear implants are intended for the totally deaf who can hear nothing or get no benefit from even the most powerful hearing aids.

Normally, sound waves are transmitted across the ear drum to the middle ear. A sensitive structure of canals in the inner ear, called the cochlea because of its spiral shell-like shape, is stimulated by the vibrations and transmits signals directly to the auditory nerve which goes to the brain, producing the sensation of sound.

In a bionic implant the cochlea is stimulated artificially or by-passed by a fine wire electrode in the ear that picks up signals from an external receiver. The implant will help people whose deafness is caused by damage to the cochlea from such causes as meningitis, head injury and Menière's disease.

Implants will help with lip reading; improve control of voice volume and intonation; restore self-confidence and relieve the sense of isolation; improve tinnitus - ringing in the ears.

Later this year the Department of Health intends to start a series of pilot projects into methods of improving hearing aid services by speeding up referral from general practitioners to NHS hearing aid centres.

Ear implants cost between £500 and £10,000, depending on their complexity. Recipients all said that the sounds differ widely from those they remember before they were deafened, and are somewhat electronic - like a badly tuned radio.
HIGH PERFORMANCE DIGITAL NEURAL NETWORK IMPLEMENTATION FOR SMALL-SCALE PORTABLE APPLICATIONS

Richard Palmer
Department of Computer Science, University College London, UK

ABSTRACT
This paper presents a digital architecture that achieves a high performance in neural network implementations. Minimal size and power consumption have been achieved by combining the versatility obtained by a MIMD array, with the simple control and communications made possible by a SIMD array. The benefits of this arrangement are presented, as well as how such an array can be controlled by both MIMD and SIMD control strategies.

INTRODUCTION
This paper briefly outlines the available neural processing devices, and looks at their suitability for small-scale applications, where small size and portability are important features. After considering the available solutions an architecture is proposed that is better suited for these applications.

Typical application areas where such a device may be used are process control systems, medical diagnosis equipment and many forms of signal processing applications. An example would be in mobile communications: echo reduction and data compression are two such tasks that a neural digital signal processor could perform. For these applications a neural device is required that has the following features:

- Work independently from a host processor
- Be very compact
- Provide real-time performance
- Low power consumption
- Programmable for different Network Topologies

This paper takes a broad look at some of the techniques used in high performance neural devices. Particular emphasis is put on the flexibility, performance and achieving real-time constraints. A strategy is then presented that solves the limitations seen in many current neural devices. Finally the details of the design used to implement this strategy is given; both the programming and the hardware of this device are considered.

DIGITAL NEURAL ARCHITECTURES
There are two main classes of digital neural architecture available, these can be classified according to their control types:

MIMD ARRAYS
These implementations make use of parallel multipliers and pipelined architectures to obtain a very high performance. Due to their high performance it is possible to time multiplex the execution of a network across an array of processors. The ability to do this can considerably reduce the number of processors required for an implementation. By careful calculations the number of processors can be estimated to achieve the required run time performance for any network. In MIMD arrays the speed up obtained from the use of multiple processors can be close to linear with the number of processors used. This is provided that care is taken to ensure that the communications strategy used does not become overloaded.
The technique used in mapping neurons onto an arbitrary size array I have termed 'Virtual Neurons' since each neuron is not represented by a physical processor, but instead by a logical mapping onto the array. It is this factor that gives MIMD arrays their versatility, and their ability to implement almost any network topology. The communications carried out in MIMD arrays involves message passing [1]. This requires the packaging of each value to be transmitted into a packet, these packets contain the source address, destination address and the actual value. This packet is then transmitted from the source onto the communications bus. The technique used in actually transmitting this packet of data varies, and will depend upon the connectivity between the processors. Shared buses and multi dimensional array are the two most common forms of connectivity.

MIMD arrays offer good flexibility, and the use of software mapping allows many different network topologies to be mapped onto an array. However the requirement for each processor to have it's own control, memory and communications hardware makes MIMD arrays expensive in silicon area.

SIMD ARRAYS

SIMD arrays use very large numbers of processors to produce a fine granularity array. This approach is similar to biological systems, with very small individual processing elements working in parallel. This class of neural implementations can provide very compact, and efficient devices. Network mapping is defined by dedicated bus structures, which removes the requirement for external arbitration or global control. Each processor can work autonomously with a minimal control strategy thus exploiting the distributed nature of neural networks, and their local interconnections. This considerably reduces the device complexity, and can bring down the size of an implementation to a single chip. The following array topologies have been proposed for SIMD neural arrays.

Shift Arrays [2]
The processors are arranged in a linear array, with the output from one layer being shifted along, and used as input to the following layer.

Broadcast Bus [3]
The processors are arranged in layers with the outputs from one layer being connected to all inputs in the following layer via a broadcast bus. The output from a processor is then broadcast to all the inputs of the next layer simultaneously.

Single and Multi Dimension arrays [4]
Communications are performed by message passing via processor-to-processor links. Such a strategy requires the incorporation of routing hardware within each processor.

Diagram 1 shows an outline of how each of these communication methods would be used to implement a network of 4, 3 and 4 units in the 1st hidden, 2nd hidden and output layers respectively.

Diagram 1: Communication strategies
Two good studies have been carried out on the implementation of neural networks on SIMD arrays [5,6]. These studies show a small variation in performance between different communication strategies, with some strategies being better suited to certain network topologies than others. I will not recover the work done in these papers, but instead I will generalise all SIMD implementations into a single group. This generalisation is valid since it uses the best theoretical performances of these SIMD arrays.

When used to implement multilayer networks a SIMD Array will operate in a manner similar to a pipeline. This is because each layer of processors uses the previous layer’s output for its input. This arrangement will exhibit all the properties of a pipelined processor with regard to throughput (pipeline iteration time) and latency (time from input affecting output).

Using this assumption it is now possible to isolate various problems that SIMD arrays present, and to predict their performance in implementing specific networks.

FLEXIBILITY

The first problem found in SIMD arrays is due to the inflexibility of these implementations. The requirement for the bus structure in each implementation to match the network topology results in a customised device having to be built for each network. This factor seriously limits the flexibility of these implementations, and means that the soft programming of a network onto a device is not possible.

PERFORMANCE

As with any pipeline, the time required between each stage in the pipeline is determined by the slowest stage. This can cause major problems in some network topologies.

If the number of units in the different layers varies, then the time taken to compute the layer with largest number of inputs will determine the pipeline iteration time. Idle cycles will have to be inserted into the shorter layers to keep them in synchronisation. This problem is particularly acute in signal processing applications, where the input layer tends to be extremely wide. This width is required to capture time and frequency components of the signal being processed.

The method that I have used in the analysis of different architectures, involves looking at the number of cycles required to implement each neuron - and the number of idle cycles that a particular architecture will impose.

To do this I shall use a typical speech processing network as an example. This network is used to extract the Fundamental Period (Tx) from a speech signal, and is used help profoundly deaf people lip read [8]. The network is shown in Diagram 2, the main features of this network can be summarised as below:

- 246 Input Units
- 6 Neurons in 1st Hidden Layer
- 6 Neurons in 2nd Hidden Layer
- 1 Neuron in Output Layer

This network structure is required to present the full spectrum of speech, and to allow some time context to be present in the input signal. Forty one consecutive time samples, composed of six frequency bands, are used to compose the input layer.

Table 1 shows how such a SIMD array would be used to implement the Tx Network. This table indicates how many processors would be used for each layer in the network, and how many cycles each processor would require to implement each neuron. From this table it is possible to compute the average efficiency of the processor array:

<table>
<thead>
<tr>
<th>Layer</th>
<th>Synapses</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Hidden Layer</td>
<td>246 Synapses</td>
<td>100%</td>
</tr>
<tr>
<td>2nd Hidden Layer</td>
<td>6 Synapses (6/246)*100</td>
<td>2.4%</td>
</tr>
<tr>
<td>Output Layer</td>
<td>6 Synapses (6/246)*100</td>
<td>2.4%</td>
</tr>
</tbody>
</table>
Table 1  SIMD performance

<table>
<thead>
<tr>
<th>Mapping</th>
<th>1st Hidden layer</th>
<th>2nd Hidden layer</th>
<th>Output layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration 1</td>
<td>246</td>
<td>6 + 240 idle</td>
<td>6 + 240 idle</td>
</tr>
<tr>
<td>Iteration 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iteration 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The total efficiency can be calculated by taking the sum of the efficiency of each neuron, and finding the mean of this value.

1st Hidden Layer: 6 Neurons \((6 \times 100) = 600\)
2nd Hidden Layer: 6 Neurons \((6 \times 2.4) = 14.4\)
Output Layer: 1 Neuron \((1 \times 2.4) = 2.4\)

Average for Whole Network: 13 Neurons \((616.8/13)\)

= 47.5%

This calculation shows the low efficiency obtained from an array of SIMD neural processors operating in a pipelined manner. The mean utilisation is 47.5% and more importantly, is that 7 out of the 13 processors are only operating at 2.4% utilisation.

Diagram 2: Fundamental Period Extraction Network

REAL-TIME CONSTRAINTS

Achieving the required time performance can also present problems for such an implementation [7]. Since the speed of operation is determined by the pipeline iteration time and the number of layers, there are limited methods that can be used to adjust the performance of a network implementation. In any real-time system there are two primary constraints that have to be met:

Sample Period
This determines the speed that data is input to a system. The sample period can be critical in many digital signal processing applications. If it is not obtained then frequency folding will occur in the input frequency spectrum, causing great inaccuracies in the models being implemented.
Response Time
This represents the time that occurs between a change on the input affecting the output. This can often be critical, for example in speech processing, the delay must be short to avoid any mismatch between audio and visual signals.

It is important to achieve both these constraints as closely as possible. If the device is too slow then problems such as those indicated above will occur. If it is too fast then the device must be more complex than necessary, representing wasted silicon area and power consumption. Therefore it is important to be able to adjust the real-time performance of an implementation to match that which is required.

SUMMARY OF DIGITAL NEURAL ARCHITECTURES
From the assessments carried out in this section, it has been possible to highlight the relevant advantages, and disadvantages of MIMD and SIMD arrays. A summary of the properties of each of these is given below:

-- MIMD arrays rely upon a software mapping of a network across an array, thus giving MIMD arrays considerable flexibility and the ability to change a network mapping by simply altering the software. Other benefits include the ability to distribute evenly a network’s workload across the processors, thus ensuring maximum utilisation of all processors. However these benefits come at a cost in considerable increased complexity. The requirement for each processor to have its own control, local memory and communications support all add to the size, cost and power consumption of an implementation.

-- SIMD arrays represent the opposite of MIMD arrays. They offer much simpler methods in implementing neural networks. The devices surveyed in this paper make use of hard wiring to define a network topology, this eliminates the need for complex control and communications. However this simplification is at the cost of reduced flexibility, and the lack of soft programmability.

By considering the beneficial features of each of these types of architecture a hypothesis can be formed. This hypothesis states:

If an architecture could be developed that can combine the properties exhibited by both MIMD and SIMD arrays, then a device could be built that exhibits all the following properties: High Flexibility, Low Complexity, Soft Programmable and Low Power Consumption, High Performance, Low Cost of Implementation.

PROPOSED SOLUTION
The architecture that I am presenting is to enable a cost effective, small-scale portable neural implementation to be built. This requires a combination of the approaches used in both MIMD and SIMD arrays. By combining both MIMD and SIMD control it will be possible to implement 'Virtual Neurons' on a device with minimal control and communications hardware.

ARCHITECTURE OF DEVICE
The architecture of this device is outlined in Diagram 3. The key points in this diagram are discussed below:

Processor Array
The main processing is performed across an array of custom built processors. These operate on a special instruction set that has been optimised for neural models. Each processor in this array has the ability to operate under both MIMD and SIMD control.

SIMD Control
The SIMD instructions for the array are generated by the SIMD Control Block. This operates by broadcasting instructions to the array. This is done so that each processor imposes a single cycle delay in the SIMD instruction stream. This has the effect of separating the SIMD instructions across
the array, thus ensuring each processor is operating on a different instruction. This scheme is used to arbitrate the shared buses, since it ensures that no two processors can be executing the same instruction at the same time.

**MIMD Control**
Because each processor will have a different MIMD program, the source for the MIMD control must be held locally. This local memory block holds both the MIMD program and the Weights. The idea to combine both weights and data in the same memory block was to simplify the hardware requirements. Kohonen used a similar technique in the Phonetic Typewriter. He developed this using a TMS320 digital signal processor, with all the weights being held as an immediate operand:

```
MPYK 0346    #Multiply Immediate
```

The drawback to using this technique was the requirement for the opcode to be repeated for each weight. Such a scheme would represent an unnecessary waste of memory space in a portable device. The technique developed in this design was to use two special instructions to indicate the start and the end of a block of weights in the memory. Such a scheme would allow program instructions and data to co-exist in the same memory with a very low overhead.

**Input Table**
The communications for the array is provided by a shared memory block, this is called the Input Table. This memory block also allows the input and output from the device. A synchronisation scheme is used to allow external devices to read and write from this memory block when the processors are not using it.

**Sigma Function**
The last feature to note is the use of a lookup table to implement the sigma function. This lookup table is again shared between the processor to minimise on hardware.

**PROGRAMMING STRATEGY**
The programming strategies used in this architecture has two main requirements:

SIMD Program
MIMD Program
The instruction set used is the same for each program type. These instructions are specially designed for neural computations, and for pointer manipulations. The most commonly used instructions from this instruction set are given below:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD_INP_PTR 0hxx</td>
<td>Load Input Pointer</td>
</tr>
<tr>
<td>LD_BIAS 0hxx</td>
<td>Load Neuron Bias</td>
</tr>
<tr>
<td>LD_WEIGHT</td>
<td>Load Weight Value</td>
</tr>
<tr>
<td>LD_INPUT</td>
<td>Load Input Value</td>
</tr>
<tr>
<td>ACC_RESET</td>
<td>Reset Accumulator</td>
</tr>
<tr>
<td>MULTI</td>
<td>Multiply Iterate</td>
</tr>
<tr>
<td>MULTT</td>
<td>Multiply Terminate</td>
</tr>
<tr>
<td>ADD_PAR_LSB</td>
<td>Add Partial Sum LSWord</td>
</tr>
<tr>
<td>ADD_PAR_MSB</td>
<td>Add Partial Sum MSWord</td>
</tr>
<tr>
<td>LD_SIGMA</td>
<td>Perform Sigma Function on Partial</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>Output Neuron Result Sum</td>
</tr>
<tr>
<td>LOCAL_OFF</td>
<td>Start of Weights</td>
</tr>
<tr>
<td>ESCAPE</td>
<td>End of Weights</td>
</tr>
<tr>
<td>STOP</td>
<td>Stop at End of Network Iteration</td>
</tr>
<tr>
<td>RETURN</td>
<td>Jump to Weight Table 0h0000</td>
</tr>
<tr>
<td>RESET</td>
<td>Reset Processor</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
</tr>
</tbody>
</table>

The programming method used in each of type of control is very different. The SIMD program is hardwired, and is not user modifiable, while the MIMD program is fully flexible and is totally user defined.

**SIMD PROGRAM**

The SIMD control is fixed - and is generated by dedicated hardware on-chip. The SIMD control consists of a sequence of instructions that performs a single synaptic update with each iteration. This program is continuously broadcast to the array - which will then update each virtual neuron under this SIMD control. The SIMD program consists of 14 instructions that will fetch an input value and its associated weight, multiply them and add the product to the partial sum.

The hardwiring of the SIMD program into the SIMD control block is done since the algorithm used for each synaptic update remains fixed. All that will alter between different networks are the virtual neuron mappings and the weights themselves.

**MIMD PROGRAM**

By the continuous broadcasting of the SIMD program allows each processor to update automatically a number of synapses. The MIMD program is required to define which input values to use, and to provide the weights. The sequence of operations required for each virtual neuron can be summarised as below:

- Load Input Pointer for Neuron's Input
- Load Neuron's Bias
- Update each Synapse (Performed under SIMD Control)
- Compute Sigma Function
- Load Input Pointer for Neuron's Output
- Write Neuron's Output into Input Table

Additional to this sequence of instructions are instructions that define the start and end for the weights of each virtual neuron, and to indicate the completion of each network update. These instructions require special considerations since they have a great effect on the operations of each processor.

**LOCAL OFF**

This instruction is used to indicate the start of a block of weights values. It has the effect of switching control from MIMD to SIMD for that individual processor. The following bytes in the
weights table will then be treated as weights data for the next virtual neuron. This virtual neuron will then be updated under SIMD control, until an ESCAPE instruction is reached.

ESCAPE
This instruction is used to determine the end of each block of weights values for a virtual neuron. This instruction acts in a similar manner to a 'rogue value' at the end of a block of data. This rogue value is detected by special logic built into the weight's table bus controller. When this instruction is detected the processor stops executing the SIMD instruction stream, and uses the following bytes in the weights table as MIMD instructions.

STOP
This instruction is used to indicate to the processor that a network update is complete. The processor will then send a signal to the SIMD control - which will synchronise the input and output from the network. Each processor will then wait for a signal from the SIMD control before initiating the next network update.

IMPLEMENTATION EXAMPLES

Now that the proposed design has been introduced it is possible to carry out some analysis on it's possible performance. The method used here will be the same as that used for the SIMD array carried out above.

Three examples are given, these indicate the possibilities that exist in optimising the network mapping for various performance criteria. The three criteria used in these examples are silicon efficiency, reduced sample period and reduced response time.

Tables 2 - 3 show these examples, and explains how the network mapping were performed.

<table>
<thead>
<tr>
<th>Mapping</th>
<th>1st Hidden Layer</th>
<th>2nd Hidden Layer</th>
<th>Output Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration</td>
<td>246</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>5 PEs Idle</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2 Virtual neuron -- Silicon effect

<table>
<thead>
<tr>
<th>Mapping</th>
<th>1st Hidden Layer</th>
<th>2nd Hidden Layer</th>
<th>Output Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration 1</td>
<td>123</td>
<td>36</td>
<td>6 + 81 Idle</td>
</tr>
<tr>
<td>Iteration 2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3 Virtual neuron -- Reduced sample period

Table 4 summarises the key features from each example. This allows comparisons to be made between the different implementations.

The figures in Table 5 are calculated as below:

No. Processors is the number of processors required by that example.

Sample Period is the pipeline iteration time. This period is what determines the speed that input can be accepted by the network.

Response Time is calculated by the product of the pipeline iteration time and the number of pipeline stages. This represents the time taken for a change at the input to affect the outputs.
<table>
<thead>
<tr>
<th>Mapping</th>
<th>1st Hidden Layer</th>
<th>2nd Hidden Layer</th>
<th>Output Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration</td>
<td>123</td>
<td>3</td>
<td>6 PEs Idle</td>
</tr>
</tbody>
</table>

Table 4 Virtual neuron -- Reduced response time

<table>
<thead>
<tr>
<th></th>
<th>Table 1</th>
<th>Table 2</th>
<th>Table 3</th>
<th>Table 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. Processors</td>
<td>13</td>
<td>6</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>Sample Period</td>
<td>246</td>
<td>258</td>
<td>123</td>
<td>127</td>
</tr>
<tr>
<td>Response Time</td>
<td>738</td>
<td>258</td>
<td>246</td>
<td>127</td>
</tr>
<tr>
<td>Idle Time</td>
<td>1680</td>
<td>30</td>
<td>81</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 5 Summary of solutions

Idle Time is a measure of the efficiency with which an implementation can map the network. This value is a sum of the idle cycles for an individual network update.

This table indicates the advantages that can be achieved by Virtual Neurons. It has shown the improved performance that can be obtained by eliminating as many idle cycles as possible, and the flexibility in being able to map a network onto an array in a variety of ways.

CONCLUSION

At present there are many possible areas where neural networks can be put to use. However, what is stopping their widespread use in many consumer products is the lack of small and cheap hardware on which to implement these networks.

This paper tackles this problem, and presents a widely versatile neural implementation that can carry out the recognition phase on a very small device. Parallelism is used with a combination of SIMD and MIMD control strategies, this allows Virtual Neurons to be mapped on the available processors in a variety of ways. This flexibility allows a wide range of different network topologies to be mapped on such an array, and ensures that real-time constraints that a system may present, are met.

The implementation of this design was carried out with each processor requiring only 5300 transistors (not including PADS or RAM). A single processor occupies only 5.5 microns^2 in 2 micron CMOS. This device shows the potential of future implementations that can provide multiple processors and RAM all on a single chip.

ACKNOWLEDGMENTS

The author would like to thank Dr Peter Rounce for his help in this work, and to SERC who has funded this research.

REFERENCES


-- 215 --
3. P. Treleaven M. Velasco, "Neural Networks on Silicon", 3rd IFIP Workshop on WSI 1989
5. Y. Kashai Y. Be'ery, "A Comparison Between Digital Neural Network Architectures", Dept. Electronic Communications, Control and Computer Systems, Tel-Aviv Univ., Israel
Chapter 1 Introduction

[2] SAIC SAIC Sales Literature Science Applications International Corporation
[3] P M Shea Operational Experience with a Neural Network in the Detection of Explosives in Checked Airline Luggage Int. Joint Conf. on Neural Networks 1990

Other Sources
[12] JESSI Study Microelectronics for Artificial Neural Networks

Chapter 2 Neural Networks


Other Sources


[9] W. M. Becker *The World of the Cell* Chap. 21


Chapter 3 Application Areas

Speech Pattern Stimulation in The Profoundly Deaf: A Comparison Of
The SiVo Speech Pattern Aid and Conventional Hearing Aids Speech 88
Edinburgh 22/26 August 1988. 7th FASE Symposium

The Profoundly And Totally Deaf IEE Int. Conf. Speech Input/Output.
London 24-26 March 1986


[5] F. Fallside Connectionist Models in Speech Recognition, A Brief
Summary and Bibliography Cambridge Univ. Report CUED/F-INFENG/TR.18
April 1988


Speech Recognition Computer Speech and Language Journal 1 pp3-27
1986

Speech Recognition Proc. IEEE Int. Conf. on Neural Networks San

Perception: The TRACE Model Parallel Distributed Processing Vol 2
Chapter 15 1986

[10] A J Robinson F Fallside The Utility Driven Dynamic Error Propagation

Satisfaction Networks To Acoustic Phonetic Attribute Determination
Proc Institute of Acoustics Vol.8 pp 323-330 1986

[12] I. Howard and M. Huckvale Training Feature Detectors for use in
Automatic Recognition Speech 88 Edinburgh 22/26 August 1988. 7th
FASE Symposium

Satisfaction Networks To Acoustic Phonetic Attribute Determination
Proc Institute of Acoustics Vol.8 1986 pp 323-330

a Trained Intermediate Representation Speech 88 Edinburgh 22/26
August 1988. 7th FASE Symposium

using a Trainable Pattern Classifier Speech 88 Edinburgh 22/26 August 1988. 7th FASE Symposium


Chapter 4 Architectural Considerations


[5] M. Holler S. Tam H. Castro R. Benson An Electrically Trainable Artificial Neural Network (ETANN) with 10240 "Floating Gate" Synapses IEEE Int. Conf. on Neural Networks 1989


Richard Palmer Phd. Thesis

Oxford 20-22 July 1988


[15] A. F. Murray Silicon Implementations of Neural Networks First IEE Conf. on Artificial Neural Networks Oct. 1989


[17] T. Kohonen New Analog Associative Memories Third Int. Conf. on AI August 1973

[18] H. P. Graf P. Vegvar A CMOS Associative Memory Chip Based on Neural Networks IEEE Int. Conf. Solid-State Circuits 1987

[19] W. Wike D. Van den Bout T. Miller The VLSI Implementation of STONN Int. Joint Conf. on Neural Networks 1990


[26] D. J. Myers G. E. Brebner The Implementation of Hardware Neural Net Systems First IEE Int. Conf. on Artificial Neural Networks London 1989

[27] Y. Kashai Y. Be'ery A Comparison Between Digital Neural Network
Chapter 5  Proposed Architecture


Other Sources

Chapter 6  Detail Description of Design

Chapter 7 Conclusion

[4] D E Shaw SIMD and MSIMD Variants of the Non-Von Supercomputer COMPCON 84 February 27 - March 1 1984

Other Sources