A VLSI Architecture for Neural Network Chips

Marley Maria Bernardes Rebuzzi Vellasco

a thesis submitted for the degree of

Doctor of Philosophy in Computer Science

University of London

Department of Computer Science
University College London
Gower Street, London WC1E 6BT

December 1991
Abstract

This thesis reports the research for the development of a neural network VLSI design environment where a neural application defined in a high-level programming environment is automatically mapped into custom VLSI chips.

This work forms the basis of the UCL hardware investigations in the Esprit II Pygmalion project into VLSI architectures for neural network chips. The ultimate goal is to take a neural network application defined by Pygmalion's neural specification language, and automatically translate it to one or more CMOS integrated circuits.

The thesis is composed of four parts: the neural network specification language; the target architecture, formed by the generic neuron model and its corresponding VLSI architecture; a simulator for the architecture; and finally a prototype Back Propagation VLSI chip.

The neural network specification language at the centre of Pygmalion has been designed to achieve flexibility and portability to allow an easy translation from a neural network specification to either binary code, for simulation, or to silicon, for execution. The language, named $\text{nC}$, has been defined as a subset of C. The basic concepts and the issues concerning how to use $\text{nC}$ are fully examined in this thesis.

The target architecture is the critical issue for automatically translating a high-level specification of a neural network into application-specific chips. Consequently, the definition of a generic neuron model incorporating the main features of neural algorithms, and its associated VLSI architecture, form the main scope of this thesis. The architecture's communication strategy and the internal organisation of the processing element are thoroughly investigated.

The adequacy of the proposed architectural model is analysed using a simulator implemented in the C language. Simulation results of the Back Propagation execution are presented, verifying the effects of the hardware implementation on the neural network execution.

The viability in terms of layout design has been evaluated by designing and fabricating a hand-crafted prototype VLSI chip, performing the "Back Propagation" algorithm. A detailed examination of the layout results is provided, including a full description of the cell library designed.

This work is now being advanced in the Esprit II Galatea project where a silicon compiler is to be incorporated to obtain a complete and integrated route from a $\text{nC}$ neural model specification into VLSI neuro-chips. This incorporation will lead to a complete integrated programming environment for artificial neural networks.
To Pedro, my loving husband
Acknowledgements

I would like to acknowledge my indebtedness to my supervisor Prof. Philip Treleaven who assisted, advised and gave me tremendous support throughout this work. His guidance and continuous encouragement made the conclusion of this thesis possible.

I am also indebted to many other staff and colleagues in the Department of Computer Science at UCL, for their valuable discussions, encouragement and friendship. In particular, I would like to thank Marco Pacheco, Carlo de Oliveira and Meyer Nigri for their irreplaceable friendship and invaluable discussions through the whole process of the architecture specification and implementation. I am also very grateful to Michael Recce for his patience and real incentive during the writing up phase of this thesis.

I would also like to thank Dr. Peter Rounce and Dr. Apostolos Refenes for reading this report and giving invaluable comments.

Special thanks must go to all my friends, Debinha, Iclea, Cristina, Sergio, Marquinhos and many others, for their support, encouragement and friendship throughout these years.

I wish to thank the Brazilian institution CNPq and the Pontifícia Universidade Católica do Rio de Janeiro for sponsoring my postgraduate studies here at University College London. In addition, I want to thank the Esprit II Galatea project for financing the fabrication of the prototype integrated circuit.

Last but not least, I want to thank my loving parents, for their inestimable love, encouragement and confidence during all these years, without which this thesis would not have been thinkable.
# Table of Contents

**CHAPTER 1 - Introduction** ................................................................. 11  
1.1. Neural Computing ........................................................................ 11  
1.1.1. Artificial Neural Networks ...................................................... 13  
1.2. Motivations & Design Issues ....................................................... 17  
1.3. Thesis Goals ................................................................................ 19  
1.4. Thesis Contributions ................................................................... 20  
1.5. Thesis Organisation ................................................................. 21

**CHAPTER 2 - VLSI Architectures for Artificial Neural Networks** ..... 23  
2.1. Neurocomputers .......................................................................... 23  
2.1.1. Special-Purpose Neurocomputers ......................................... 24  
2.1.2. General-Purpose Neurocomputers ......................................... 33  
2.2. Weight-based Architectures ...................................................... 40  
2.2.1. Siemens Architecture .......................................................... 40  
2.2.2. Philips Architecture ............................................................ 43  
2.3. Neuron-based Architectures ....................................................... 46  
2.3.1. INPG Architecture ............................................................. 46  
2.3.2. UCL Architecture .............................................................. 49  
2.4. Summary ................................................................................... 52

**CHAPTER 3 - Pygmalion Neural Programming Environment** ........ 54  
3.1. Neural Programming Environments .......................................... 54  
3.2. Pygmalion Programming Environment ...................................... 57  
3.3. Neural Network Representations in Pygmalion ....................... 61  
3.3.1. N High Level Language ...................................................... 62  
3.3.2. nC Specification Language ................................................. 63  
3.4. Neural Network Execution Methods ....................................... 64  
3.4.1. Simulation .......................................................................... 64  
3.4.2. Silicon Execution ............................................................... 65  
3.5. Summary .................................................................................. 66

**CHAPTER 4 - nC Neural Network Specification Language** ............ 67  
4.1. nC Programming ....................................................................... 67
List of Figures

Figure 1.1: Multilayer Neural Network. ................................................................. 14
Figure 1.2: Internal structure of the generic neuron model. ............................. 15
Figure 1.3: Neural Network Threshold Functions. ............................................. 16
Figure 2.1: General View of a Special-Purpose Neurocomputer. .................... 25
Figure 2.2: Personnaz & Weinfeld's interconnection strategy. ......................... 30
Figure 2.3: The Fujitsu Basic Architecture. ....................................................... 30
Figure 2.4: System configuration for the NTT AAP-2 architecture. ................. 36
Figure 2.5: Adaptive Solution processor array system architecture. ............... 38
Figure 2.6: Hitachi's WSI neural network architecture. .................................... 39
Figure 2.7: Schema of the Siemens 4-neuron module with 16 multipliers. ....... 41
Figure 2.8: Data flow through a 4-neuron unit. ................................................ 42
Figure 2.9: Philips' basic circuit architecture. ................................................... 44
Figure 2.10: INPG's basic neural processor structure. ....................................... 47
Figure 2.11: A partial view of the 2-D array of neural processors. .................... 48
Figure 2.12: 4x2 array of neural processors. ..................................................... 48
Figure 2.13: Generic Neuron Architectural Framework. .................................. 50
Figure 3.1: A Typical General-Purpose Programming Environment. .............. 57
Figure 3.2: Structure of the Pygmalion Neural Programming Environment. .... 58
Figure 3.3: Programming a Neural Network Application. ............................... 61
Figure 4.1: Representation of the nC Programming method. .......................... 68
Figure 4.2: The Back Propagation state update of a neuron-level rule level. .... 71
Figure 4.3: The Back Propagation state update of a cluster-level meta-rule level. 71
Figure 4.4: The nC hierarchical structure system. ........................................... 72
Figure 4.5: The complete system structure definition. ..................................... 74
Figure 4.6: config general framework. .............................................................. 77
Figure 5.1: The structure of the generic neuron model. .................................... 85
Figure 5.2: The generic neuron's bus interconnection strategy. ....................... 90
Figure 5.3: The architecture's interconnection strategy. ................................... 92
Figure 5.4: The multi-bus interconnection strategy. .......................................... 93
Figure 5.5: The processing element's internal structure. .................................. 94
Figure 5.6: The processing element's internal organisation. ............................ 97
Figure 5.7: The memory unit's internal structure. ............................................ 98
Figure 5.8: The communication unit's internal structure. ..................................................... 102
Figure 5.9: Execution Unit's internal structure. ................................................................. 105
Figure 6.1: Results from the investigation of the look-up table effects. ............................ 117
Figure 6.2: Results from the network performance examination. ......................................... 121
Figure 7.1: VLSI Design Environment. .............................................................................. 126
Figure 7.2: The Interconnection Structure of Cells. .......................................................... 129
Figure 7.3: Schematic of the basic register. ....................................................................... 131
Figure 7.4: Carry chain circuit for the ALU. ..................................................................... 134
Figure 7.5: Organisation of the generic neuron Operative Part. .......................................... 137
Figure 7.6: Organisation of generic neuron Control Part. .................................................. 139
Figure 7.7: Clock system used in the generic neuron prototype. ....................................... 139
Figure 7.8: Floor Plan of the Back Propagation Prototype. ................................................. 141
Figure 7.9: Back Propagation Prototype Layout. ............................................................... 143
Figure 8.1: Packing density curves for the generic neuron architecture. .............................. 156
List of Tables

Table 3.1: Neural Network Programming Environments. ................................. 56
Table 5.1: The mapping of popular neural models into the *generic neuron* model. ...... 86
Table 5.2: Command lines in the control bus of a Back Propagation chip. ................. 101
Table 6.1: Minimum number of bits for the fractional part, from equation 6.1. ............. 119
Table 6.2: Parameters used for the network’s performance examination. .................... 120
Table 7.1: Features of the *Generic Neuron* Back Propagation Prototype. ................. 142
Table 8.1: Packing Density Investigation of the *generic neuron* architecture. .......... 156
Chapter 1

Introduction

This chapter presents the motivations and main goals of this thesis. A brief introduction to the neural computing area is initially provided for the purpose of completeness. Then, the motivations and goals of the thesis are discussed. The chapter concludes with an overview of the thesis contributions and thesis organisation.

1.1. Neural Computing

Neural computing has re-emerged in the last five years as an important programming paradigm that attempts to mimic the functionality of the human brain. This area has been developed to solve demanding pattern processing problems, like speech and image processing, which were intractable or extremely cumbersome when implemented using traditional computing methods.

By analogy with the human brain, artificial neural networks are massively parallel systems that rely on simple processors and dense arrangements of interconnections. These interconnection patterns can range from single-layer, feed-forward networks, like the early Perceptron model, to more complex topologies, formed by multi-layers with backward connections, as in the Back Propagation model, or even a vector connected to a two-dimensional grid of processing elements, as happens in the Self-Organising Map algorithm.

Despite this inherent high degree of parallelism, artificial neural networks have been mainly implemented as simulations on sequential machines. However, in the near future, these conventional machines are unlikely to be effective in providing the required computing capabilities to execute more challenging neural network applications. Highly parallel architectures, based on very primitive processors, have the potential to demonstrate the genuine computing potential of artificial neural networks. To achieve the high performance required by some applications, it is important that special neural network computers, called neurocomputers, be developed. These highly parallel, application-specific neurocomputers provide optimised execution of the neural algorithm to match the requirements of more demanding applications.
On the other hand, due to the continuous evolution of artificial neural networks, there is a great demand for the development of neural network programming environments, where neural algorithms can be developed, tested and improved in a suitable software environment. These neural network programming environments, which usually run on conventional computers, are tool-kits for the development and simulation of neural algorithms and applications. These software environments provide to the neural network user the essential tools to investigate and adjust the neural algorithm parameters to the particular application.

Many investigations are being carried out along these two research lines, namely neurocomputers and neural programming environments. In the neurocomputer area three basic approaches have been followed: accelerator boards, which are usually based on standard signal-processing chips and plugged onto workstations; parallel processor arrays, formed by a large number of simpler processing units; and specialised VLSI (very large scale integration) neuro-chips, dedicated to a specific neural network application. Neural programming environments also span over a large variety of products, ranging from simple academic environments, to more complex research and commercial products. However, although these two research directions are recognised as fundamental to the progression of the neural computing area, their final products have been developed independently, with no integration between the software and hardware tools. It is clear that it is important to combine these two trends and develop a complete and integrated environment incorporating two routes towards neural network execution: simulations on conventional machines; and optimised hardware execution via high performance application-specific chips. In such a complete and integrated environment, a neural network can be developed and tested in the simulation tool using various sequential and parallel conventional architectures. After the network has been analysed and appropriately configured, application-specific integrated circuits (ASICs) can be automatically produced, providing optimised computation of the specific neural model. This integrated programming environment permits the designer to match the required performance to accomplish the particular neural network application. The development of such complete and general programming environment was the main goal of the Esprit II Pygmalion project\textsuperscript{17,20}.

The main goal of this research has been to propose and develop the hardware route of the Pygmalion environment, providing a design framework to automatically generate VLSI neuro-chips from an application specified in the Pygmalion neural network programming environment. The idea is to construct a flexible and portable system where a neural
network application can be either simulated on a target machine, or automatically mapped into silicon. The design framework developed in this thesis is now being advanced in the *Esprit II Galatea* project where a silicon compiler will be incorporated to achieve a complete, integrated, neural network programming environment for the automatic generation of application-specific integrated circuits (ASICs).

The critical issue in the automatic generation of ASICs is the target VLSI architecture. This architecture should be sufficiently simple to achieve high performance, but general enough to execute the large diversity of existent neural models. Hence, it is necessary to devise a generic model that embodies the main features of the artificial neural networks. To conceive such a generic model, a comprehensive understanding of neural network models is necessary. So, although the basic concepts of artificial neural networks are widely discussed in the literature\(^\text{25,49,68,131}\), it is interesting to review the properties of artificial neural networks, before presenting the *generic neuron* architecture.

### 1.1.1. Artificial Neural Networks

An artificial neural network is a computing system that combines a network of highly interconnected processing elements (PEs) that use mathematical algorithms, or *neural network models*, to carry out the information processing. These networks have demonstrated their ability to deliver simple and powerful solutions in areas that for many years have challenged conventional computing approaches.

An artificial neural network is represented by *weighted* interconnections between artificial neurons, or PEs. These weights and the states of PEs represent the data of the network. A neural network that *learns* to recognise patterns, does so by adjusting the weights of the connections between the processing elements. A network *recalls* patterns based on information derived from associations already established between input and output patterns. Thus neural networks are inherently *adaptive*, conforming with the imprecise, ambiguous and faulty nature of real-world data.

Neural networks can be characterised by a few key properties:

- *network topology*
- *recall procedure*
- *training/learning procedure*
- *input values*
Network topology — The neuron interconnection pattern is perhaps the most distinguishing characteristic of a neural model. Usually, processing elements (PEs) are arranged into disjointed structures called layers. Early models such as the Perceptron consist of a single layer, in which each network input is connected to all PEs. The information flows through the network from input to output, without feedback of outputs. Models belonging to this class are called feed-forward. More recent models have extended the concept of a single layer network into a structure of multiple feed-forward layers (Figure 1.1). In multilayer networks, internal layers are hidden from external inputs and outputs, but receive weighted inputs. Models have also introduced the concept of backward connections. These networks improve the training and learning procedure by feeding backwards error values from a succeeding layer to a previous one.

Figure 1.1: Multilayer Neural Network.

To discuss the other neural network properties, a model of a generic artificial neuron is introduced. The generic neuron concept, developed in this research, incorporates the main variations of learning and recall procedures found in the different classes of neural network models. A more detailed description of the generic neuron features is presented in chapter 5.

Figure 1.2 depicts the overall structure of the generic neuron model. It comprises three specific functions (f1, f2 and f3), a table of weights (W) and sets of input (S,E) and output (s,e) signals. A neuron receives the input states S from other neurons, and forwards its own state output s. Similarly, a set of input errors E and an error output e
provide backward connections in the network. In the generic neuron model, $f_1$ stands for the activation function, $f_2$ for the weight-updating function, and $f_3$ for the error-calculation function.

Recall procedure — This procedure is specified by the activation function $f_1$, which comprises the propagation rule $g$ and the threshold function $T$:

$$\text{net} = g(S, W) \quad \text{and} \quad s = T(\text{net})$$

In its simplest form, the propagation rule calculates the weighted sum of the inputs, modified by an offset $\theta$ that defines the neuron bias:

$$\text{net} = \sum S \cdot W - \theta$$

The non-linear (threshold) function uses the net value (the propagation rule's result) to evaluate the actual neuron's output state. Common threshold functions (illustrated by Figure 1.3) include hard limiter, sigmoid, and pseudo-linear.

Training/learning procedure — This typically interactive process relies on the presentation of many training patterns. It is, therefore, more computationally intensive than the recall procedure. Learning procedures are either supervised or unsupervised. In supervised learning the system submits a training pair to the network, consisting of an input pattern and the target output. The network adjusts the weights based upon the PE's error value $e$ (usually the difference between the expected output and the computed
output of each PE), so that the difference diminishes with each cycle. Unsupervised learning procedures, on the other hand, classify input patterns without requiring information on target output\textsuperscript{64}. In such procedures, the network must detect the patterns' regularities, classifying them into disjoint groups according to their feature similarities.

Learning algorithms generally involve two additional functions in comparison to the recall phase: the $f_2$ and $f_3$ functions. The error calculation function $e = f_3(s, E, W)$ directs (in supervised learning) the updating of weights by determining how far the current result is from the target output. The function $\Delta W = f_2(S, e, W)$ actually updates the weights, using the resultant value from the evaluation of $f_3$ function. Most models employ as weight-updating function some variation of the Hebbian learning rule\textsuperscript{46}. This rule states that the weight between neurons should be strengthened proportionally to their activities:

$$\Delta W_{ij} = s_i \cdot s_j$$

where $W_{ij}$ is the weight associated with the connection between $PE_i$ and $PE_j$, $s_i$ is the $PE_i$ state value, and $s_j$ is the $PE_j$ state value. Other learning algorithms include the Hopfield model, Delta rule, competitive learning, the Boltzmann learning algorithm, and the Back Propagation algorithm\textsuperscript{68}.

Finally, the input values of neural network models can be characterised by the range they adopt - namely, binary or continuously valued input.

After this brief introduction to the neural computing area, the next sections present the main motivations and goals of this thesis.
1.2. Motivations & Design Issues

As mentioned before, artificial neural networks have been mainly implemented as simulations on sequential machines. More recently, the implementation of neural computers is being recognised as the way to achieve the real potential of artificial neural networks. Nevertheless, current hardware implementations lean either to the optimisation of the network performance, as happens in the case of special-purpose neurocomputers, or try to provide more flexibility for the execution of a large range of neural network models, as occurs with the general-purpose neurocomputers. Hence, it is desired to achieve a compromise between these two trends in order to provide high-performance application-specific neurocomputers and, at the same time, allow the user to cost-effectively execute different neural algorithms.

Designing a VLSI neurocomputer to optimise performance and flexibility requires exploiting the architectural properties of VLSI design. Some of the VLSI properties relevant to the design of a cost-effective neurocomputer are summarised below:

- Design complexity can reach critical proportions, which encourages the development of simple, regular and replicated structures.
- Much of the space on an integrated circuit is occupied by wires that carry control and data between functional blocks.
- Communication degrade performance, becoming progressively more expensive in silicon area and propagation time.

Consequently, to cope with these VLSI properties and with the intrinsic neural network features, the architectural framework proposed in this research exhibits the following architectural properties for the PEs' internal structure and their interconnection strategy:

**Flexibility**

The VLSI architectural framework, based on the generic neuron model, provides a flexible skeleton able to execute a wide range of neural network models. The processing element's functionality (including the learning capability) is defined by the user, which makes use of the high level specification language named nC. A silicon compiler receives this information and, based on the proposed architectural framework, produces customised ASICs with the required functionality. This approach offers the desired flexibility at the neural model specification phase, without affecting the final performance.
**High Performance & Parallelism**

The necessary high performance is achieved by using very primitive, parallel processing elements (PEs). The PEs are optimised for the computation of a specific neural network model and are automatically generated by the silicon compiler.

**Modularity**

To constrain communication and, consequently, reduce interconnection cost and communication delays, the *generic neuron* architecture is formed by replicative, self-contained processing elements, each comprising processor, communication and memory functions.

**Flexible & Regular Communication**

To reduce wiring problems between PEs and also to cope with the considerable number of different interconnection patterns existent among neural models, the architectural framework uses the bus interconnection strategy. The broadcast bus strategy is able to implement all complex topologies and handle all connections with minimum overhead in communication protocol and routing algorithms. This approach allows the interconnection of a large number of PEs, and the performance can be further increased with the utilisation of multiple busses.

**Expansibility**

The broadcast bus structure and the use of replicative processing elements permit an easy and fast expansion of the number of PEs, in order to comply with different applications.

**Design Scalability**

With the fast improvements in VLSI design technology, the architectural framework should provide for a scalable design as feature size decreases. The modularity of the layout design and the bus interconnection support this property, allowing the addition of more PEs within the same integrated circuit without affecting the total number of pins.

**Minimum Silicon Area**

Because artificial neural networks make use of a large number of PEs, optimising silicon area is an important issue for the development of a successful neurocomputer. The approach taken of producing application-specific integrated circuits considerably cuts down the area, as the final chip is tuned to a specific functionality. In addition, simple and regular structures were used to build the cell library, reducing wiring inside the chip.
In spite of the smaller area that can be achieved using analogue devices, this design method leads to circuits with high power consumption and low noise immunity. Moreover, certain neural network models require an accuracy that is not attainable using analogue systems. On the other hand, digital systems can be faster and their design techniques are more advanced. Furthermore, the learning phase is more naturally implemented using digital devices as it is difficult, using analogue technology, to devise a general learning mechanism which is suitable for a large variety of neural network algorithms. For all these reasons, digital design has been utilised throughout the architecture implementation.

1.3. Thesis Goals

Artificial neural networks are progressing to the point where, for some specific application such as speech recognition, sequential computers will no longer be adequate for their computation. It is believed that massively parallel computers, composed of very simple, replicative processing elements are the answer to reach the natural capability of artificial neural networks. Furthermore, neural network algorithms are continuously evolving, originating various neural programming environments that allow researchers to experiment with existent neural algorithms as well as develop new ones. There is, therefore, a need to integrate these two systems - neurocomputer and neural programming environment - to create an automatic route from the software environment to the manufacture of neuro-chips dedicated to the specified and tested application.

Hence, the ultimate goal of this research is to define a design framework for the automatic generation of dedicated neural network chips from the Pygmalion neural network programming environment. The research reported in this thesis comprises four steps:

- firstly, the definition of the Pygmalion’s neural network specification language, namely nC, used to describe and test the intended neural network application;
- secondly, the proposal of the generic neuron model and its associated VLSI architecture, which is used as the target architecture to be configured according to the application;
- thirdly, the architectural simulations executed to assess the suitability of the proposed architecture;
• and lastly, the implementation of a VLSI Back Propagation neuro-chip to assess the hardware complexity of the proposed architecture.

This work is now being extended as part of another PhD thesis\(^8\) with the incorporation of a silicon compiler. The silicon compiler will translate the *nC* neural network specification into dedicated neuro-chips, making use of the *generic neuron* architecture as the basic architectural framework.

### 1.4. Thesis Contributions

Based on the proposed aims and the outcomes of this research, the main contributions of this thesis are:

— the concise documentation of the *Pygmalion* and *Galatea* hardware research, provided through the reviews in chapters 2 and 3. Chapter 2 presents a significant discussion of the four different architectures under investigation in the *Galatea* project. Chapter 3 presents the basic features and motivations of the *Pygmalion* programming environment.

— for the *Pygmalion nC neural network specification language*, a major contribution to its design and particularly its specification, with a subsequent development of the language parser for syntax checking.

— the proposal of the *generic neuron model* - a general model for the artificial neuron - produced from the investigation of the main neural network algorithms. This *generic neuron* model combines into a single parameterised structure the common features extracted from these neural algorithms, providing a general framework to describe the neuron's functionality.

— a *VLSI architecture* specified, designed and implemented based on the *generic neuron* model. The *generic neuron* VLSI architecture encompasses the definition of the processing element’s internal structure and the associated interconnection strategy for the data communication.

— a *prototype VLSI chip*, completely designed and fabricated using CMOS 2\(\mu\)m technology. The prototype, based on the Back Propagation algorithm, demonstrates the viability of the proposed architecture, both in terms of the processing elements packing density and the overall performance.
Additionally, the work accomplished has generated several published papers\textsuperscript{20, 118, 119, 125} and has also contributed to other research projects\textsuperscript{89, 90}. The architectural framework developed in this research is currently being used as the target architecture for the UCL neural silicon compiler\textsuperscript{89}.

1.5. Thesis Organisation

The remainder of this thesis is organised as follow:

Chapter 2 provides a survey of VLSI architectures for artificial neural networks. It firstly presents an overview of neurocomputers and then describes some implementation examples, classifying them into special-purpose and general-purpose. This is followed by an examination of the four architectures under analysis in the \textit{Esprit II Galatea} project: the Siemens, Philips, INPG (Institut National Polytechnique de Grenoble), and, very briefly, the UCL architecture developed in this thesis.

Chapter 3 presents a description of the neural network programming environment developed in the \textit{Esprit II Pygmalion} project. It initiates with a short review of neural programming environments and their functionality. This is followed by an introduction to the \textit{Pygmalion} environment. Then, the two network representation levels available in \textit{Pygmalion}, the high level language, \textit{N}, and the specification language, \textit{nC}, are discussed. The chapter finalises with the analysis of the two execution methods: simulations on serial or parallel conventional computers, and execution in silicon as application-specific neuro-chips.

The \textit{Pygmalion}'s machine-independent neural network specification language, called \textit{nC}, is discussed in Chapter 4. The chapter commences with a general description of the \textit{nC} programming features and then introduces the basic concepts that make \textit{nC} a neural network dedicated language. It carries on with information on how to programme an algorithm and an application, the two basic elements to fully specify a neural network in \textit{nC}. The chapter terminates with an examination of the \textit{nC} parser, designed to check if the user specification conforms with the language syntax.

Chapter 5 gives a thorough description of the architectural framework developed in this research. It starts with an explanation of the \textit{generic neuron} model and how different neural algorithms can be mapped into the proposed model. It then follows with a complete analysis of the VLSI architecture, providing all the details in terms of the processing element structure and the communication strategy chosen. Finally, it presents the architectural parameters that can be used by the silicon compiler to generate different
neural network applications.

After the complete specification of the architectural framework, a mechanism to assess the suitability of the proposed model was necessary. So, chapter 6 examines the architectural simulations carried out in this research. Firstly it presents a short introduction to the simulator, describing the user interface provided and the data structure used. It then exhibits the simulation results based on the Back Propagation algorithm.

Chapter 7 investigates the implementation of the proposed architecture as VLSI chips. It firstly analyses the design approach taken and then provides a full description of the cell library used in the project. The chapter then concludes discussing the results of the Back Propagation layout design implemented using CMOS 2μm technology.

Chapter 8 is the assessment chapter. The four main elements of the research - the $nC$ language, the generic neuron model and the corresponding VLSI architecture, the simulator and the VLSI chip - are discussed, investigating their strengths and weaknesses.

Chapter 9 concludes this thesis by discussing the results of this work and describing the activities currently under way in the Galatea project, indicating the necessary future work to achieve the final goal of a complete programming environment for artificial neural networks.
Chapter 2

VLSI Architectures for Artificial Neural Networks

This chapter provides a review of VLSI architectures for artificial neural networks. Neurocomputers are first classified into special-purpose and general-purpose. Then the chapter examines the four architectures under analysis in the Esprit II Galatea project: the Siemens, Philips, INPG (Institut National Polytechnique de Grenoble), and finally the UCL architecture developed in this thesis.

2.1. Neurocomputers

As mentioned in Chapter 1, neural network models are evolving in such a way that conventional computers will not be capable of providing the necessary performance to execute more elaborate applications. Only highly parallel machines, with neural network dedicated hardware, namely neurocomputers, will achieve the required performance to execute applications such as speech recognition\(^{19,49,100}\). Therefore, many researchers are investigating different approaches for the development of high performance neurocomputers.

A neurocomputer is essentially a parallel array of interconnected processors that operate concurrently. Each processing element is primitive (that is, it can be an analogue neuron or a simple reduced instruction set - RISC - microprocessor) and may contain some local memory.

When considering the set of possible architectures\(^{124}\) for the basis of a neurocomputer, the important design issues are parallelism, performance, flexibility, and their relationship to the silicon area. These issues, which are directly influenced by the node complexity\(^{111}\), lead to radically different systems. Neurocomputer systems range from dedicated hardware as simple as traditional random access memory (RAM), to programmable processors that are analogous in complexity to conventional computers. Despite this variety of possible systems, neurocomputer architectures can be classified into two groups, according to their degree of flexibility and performance: special-purpose and general-purpose neurocomputers\(^{118,119}\). The special-purpose neurocomputers tend to favour speed in comparison to generality. They are specialised neural network hardware implementations which are dedicated to a specific neural network model, having poten-
tially high performance. Special-purpose neurocomputer designers have studied various different technologies, including electronic (digital and analogue), optical and electro-optical\textsuperscript{47,49}. \textit{General-purpose neurocomputers}, on the other hand, are generalised, programmable neural computers for emulating a range of neural network models. Consequently, they tend to be slower than the special-purpose machines, but allow efficient experimentation with different network models. This approach is generally implemented using digital technology.

A review of current implementations in both special-purpose and general-purpose neurocomputers is presented in the next sections.

2.1.1. Special-Purpose Neurocomputers

The approach for \textit{special-purpose neurocomputer} architectures is to directly implement a specific neural network model in hardware. Any neural network model could theoretically be chosen, although Kohonen\textsuperscript{64} and Hopfield\textsuperscript{53} associative memory models are typically favoured, because of their simplicity in terms of regular connectivity and processor's functionality.

Designers have successfully built several examples of very high performance, special-purpose neurocomputers. This review will concentrate exclusively into the existent analogue and digital implementations. Information on current designs employing other technologies, like optical and even electrochemical, can be found in Hecht-Nielsen's book\textsuperscript{49}.

\textit{Analogue Designs}

Analogue electronic implementations of neural networks typically employ physical structures that resemble the simplified mathematical model of the neuron’s state calculation:

\[ s_j = T \left( \sum_{i=1}^{N} S_i \times W_{ij} - \theta \right). \]

They usually utilise amplifiers to perform the neuron’s functions and resistors to substitute the synaptic weights. Because of this approach, analogue implementation are normally restricted, since only those mathematical functions found in the physical materials are available for use\textsuperscript{49}. In addition, they are susceptible to noise and temperature changes, providing low accuracy. These deficiencies restrict the range of possible neural models that can be implemented, since models like Back Propagation are not tolerant to
low accuracy\textsuperscript{19}. Nevertheless, analogue designs usually produce smaller circuits allowing therefore more processors per chip, which is an important issue in the development of neurocomputers.

The circuit of Figure 2.1a illustrates the simplest structure used in analogue implementations. It is a straightforward electronic implementation of the simplified electrical model of the biological neuron. Wires replace the input (dendrites) and output (axon) structures. Resistors model the synaptic connections between neurons, and the amplifier models the cell body by executing the threshold function. Inputs appear as current on the summing wire. Using these electronic neurons, a neural network can be implemented as a crossbar representation (Figure 2.1b). Outputs (vertical lines) A1-A4 are connected through resistors to the horizontal summing wires, which contain the weighted sum of the output signals of other processing elements (PEs). Inverted input lines are also provided to be used when both positive and negative values are required.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example-figure}
\caption{(a) Electronic Neuron \hspace{1cm} (b) Interconnected Processing Elements}
\end{figure}

\textbf{Figure 2.1: General View of a Special-Purpose Neurocomputer.}

The main problem of this configuration is that resistors occupy a large area on VLSI chips, making it impossible to implement networks with a large number of interconnections. Consequently, more recent circuits have replaced fixed resistors by \textit{synaptic circuits}. Besides making the integration of a large amount of synapses in one chip easier, synaptic circuits also support the implementation of programmable connections\textsuperscript{41}. Programmable here means that the synaptic weight can be set to a specific value, without having to redesign the integrated circuit. It does not imply, however, that the architecture is able to implement learning, as defined in the artificial neural models.
Many groups are working on analogue implementations of neural networks. Graf, Jackel and their colleagues at AT&T Bell Labs have designed a number of neural network chips. Their first implementation consisted of an analogue chip with a 22×22 matrix of fixed, microfabricated resistive synapses and off-chip amplifiers. The resistor matrix was fabricated on oxidised silicon substrates, occupying an area of 88×88μm². Another early design consisted of a fully interconnected array of 256 on-chip amplifiers. The connections between amplifiers were made through fixed resistors, using the same microfabrication process utilised in the previous chip. The chip was designed in 2.5 μm CMOS technology occupying an area of 5.7×5.7mm². More recent designs that provide a certain degree of programmability for the weight values include:

- a hybrid analogue/digital chip, comprising 54 neurons fully interconnected through a programmable resistor matrix. The synaptic circuit is programmed by the contents of two RAM cells, providing excitatory, inhibitory and disabled connections. Although this approach supports adjustable weight values, the use of RAM cells increases substantially the synaptic circuit size, losing, therefore, the main advantage of analogue designs. The chip was fabricated in 2.5 μm CMOS technology resulting in an area of 6.7×6.7mm².

- a fully analogue design of an adjustable connection matrix composed of 1104 connections. The matrix is arranged in an array of 46 inputs and 24 outputs. The chip, which does not include the amplifiers, was designed in 1.25 μm CMOS. The synaptic circuit is based on dynamic weight storage, where the analogue weight value is stored as a charge on a capacitor. Dynamic weight storage grants much smaller synaptic circuits but, owing to the intrinsic dynamic characteristic, problems like charge leaks and data corruption are likely to occur if long hold times are required.

Another group at Caltech, led by Mead, has also pioneered analogue neural chips. Their first prototype was an associative memory chip containing 22 amplifiers and a full interconnection matrix that consumed 462 programmable interconnection elements. Progressing from the resistor/amplifier approach, Mead's methodology now centres on MOSFET transistors operating in the subthreshold regime. This new scheme reduces considerably the power consumption as extremely small currents are flowing in the circuit. Mead's interests lie primarily in the implementation of processing functions such as of the retina and the problem of motion detection. Mead's more recent experiments with biologically inspired, analogue, VLSI systems are fully described in his book.
Verleysen and his group at Université Catholique de Louvain, Belgium\textsuperscript{127-129}, are also developing analogue, special-purpose neurocomputers. Their synaptic circuit design follows a similar approach to the programmable, analogue/digital circuit developed by Graf. They also utilise RAM cells to programme the synaptic value but the circuit was improved to overcome the problem of not attaining equal values for the excitatory and inhibitory currents. The problem is avoided by using the same type of transistor to sink and source current on the input line of the neural processor. This has been achieved by using two distinct lines to sum the currents: one for excitatory currents and another for inhibitory currents\textsuperscript{129}. The design of the neural processor had also to be modified in order to be adapted to two input current lines. To test the performance of this new scheme, they have designed a small chip containing 14 neurons and 196 synapses.

A different approach, that uses pulse-stream arithmetic, is used by Murray and his group at University of Edinburgh\textsuperscript{82-85}. The architectural structure is also based on a fully interconnected network arranged in a 2-dimensional array of synapses, but signals flowing through the network are actually a stream of pulses. Their main motivation for using pulse-stream form was its analogous relationship with biological neurons\textsuperscript{84}. In the pulse-stream design, the neuron behaves as a switched oscillator, where the oscillator’s firing rate is controlled by the level of accumulated neural activity. Therefore, the neuron’s output state is represented by the output pulse frequency and each synapse determines the proportion of the input stream that will pass to the neuron input. Their first implementation consisted of a 3 \( \mu \text{m} \) CMOS chip, comprising a 10\( \times \)10 synaptic matrix, and 10 off-chip neurons. The synaptic circuit was based on a RAM, to store the synaptic weight, and on \textit{chopping clocks} to form the pulse-stream input to the neuron. The basic component of the neuron was a voltage-controller oscillator (VCO) which is controlled by the charge on a capacitor. Due to the low level of integration caused by the digital weight memory, a new design for the synapse circuit is being implemented, deriving from the dynamic weight storage technology\textsuperscript{82,84}.

An Intel research group\textsuperscript{52} is working with \textit{floating gate}, non-volatile, memory technology. This technique, firstly proposed by Alspector\textsuperscript{14}, is founded on the use of an analogue charge on a floating gate of a transistor to represent the stored weight value. The architecture proposed by Intel, called Electrically Trainable Neural Network (ETANN), improves this basic idea by using differential operation in the synaptic circuit, affording full four quadrant multiply operation. The synaptic cell comprises two EEPROM (Electrically Erasable Programmable Read-Only Memory) cells in which the differential voltage represents the stored weight. Each synapse multiplies a signed analogue voltage by the
stored weight and generates a differential current proportional to the product. The differential currents are then summed, transferred through a sigmoid function and finally appear at the neuron output as an analogue voltage. To adjust the weight value, an external intelligent controller must calculate the voltage to be applied to each synapse in order to attain the required weight value. The weights are addressed and modified one at a time. This technique affords electronic programmability at the expense of demanding non-standard fabrication processes\textsuperscript{83}. The Intel group produced a chip with 64 neurons and 10240 floating gate synapses, where 2048 are used for setting bias for each neuron, 4096 used to fully interconnect the neurons and the remaining 4096 to connect the 64 neurons to 64 independent inputs.

Finally, some other groups working with analogue implementations include: Agranat, developing circuits using charge coupled devices (CCD)\textsuperscript{7}; Rossetto and his group at Institut National Polytechnique de Grenoble\textsuperscript{106}; and Goser and his team at University of Dortmund\textsuperscript{38}.

The above examples demonstrate that analogue circuits are suitable for implementing small-scale, specialised neural network applications that can be integrated into one single chip\textsuperscript{86,135}. This comes as an immediate consequence of the fully connectivity approach used in all analogue systems. The data communication is performed with no multiplexing of signals, using dedicated wires for each necessary value that must be transmitted from one neuron to another. This scheme results in faster circuits but also makes the cascading of various chips very complex. Additionally, due to the native analogue characteristics, it is very difficult to produce two integrated circuits with the same features. On the other hand, analogue technology can grant smaller circuits, although in order to achieve small and programmable synaptic circuits, non-standard processes are being used, such as the MNOS (metal nitride oxide silicon) technique for EEPROM-based circuits. Then, to overcome some of these intrinsic problems found with the analogue technology, many groups are inclined to follow the digital approach.

**Digital Designs**

Although analogue designs can result in smaller circuits, many researchers are developing digital special-purpose neurocomputers. The main advantages of digital over analogue designs are the better accuracy for weight values, the facility of using well understood fabrication techniques, and the possibility to easily implement the learning phase on-chip. Additionally, digital circuits are much less susceptible to instability due to changes in temperature. Therefore, many different approaches are being pursued for the
development of high performance, digital special-purpose neurocomputers.

One of the most well-known digital implementation is the WISARD machine, developed by Aleksander and his group.\(^9\)\(^-\)\(^11\). The WISARD architecture (Wilkie, Stoneham, Aleksander Recognition Device) is an adaptive pattern recognition system designed specifically for image processing. The system employs statistical pattern-classification techniques to achieve impressive recognition and judgement capabilities. The design of the WISARD system centres on arrays of RAM cells that operate as image discriminators. Each discriminator consists of \(K\) RAMs of \(N\) bits followed by a summation operator that counts the number of RAMs with output 1. Therefore, the input pattern represents the \(N \times K\) input image and the discriminator output indicates the similarity of an input pattern with each of the previously training set. So, when submitted to input images, the system produces a response that expresses a "measure of confidence". The system digi­tises the input image and stores it as a binary pattern in an image memory in which a single bit represents each pixel. Recognition occurs when discriminators are trained with examples of objects (frames captured by a camera) that associate with specific classes (according to the views of the object). The WISARD system has innumerable applications involving classification, such as: inspection and quality assurance, robot vision, identification and sorting.

Personnaz and Weinfeld\(^100\),\(^133\) have also designed a digital VLSI architecture, in this case specialised in the fully connected Hopfield neural model.\(^53\) The architecture was devised to maximally reduce the necessary silicon area per processor, sacrificing, to a certain extent, the communication speed. The network contains \(N\) identical neural processors, each one with full arithmetic capability for learning and updating, and for storing the \(N\) relevant synaptic values. Because the architecture implements the Hopfield algorithm, the state values are coded in one bit, considerably simplifying the communication and arithmetic calculations. The transmission of states among the processor is performed by a circular \(N \times 1\) bit shift register, as shown in Figure 2.2. Every neuron executes, simultaneously, a partial weighted sum at each elementary shift. After a complete revolution, each neuron, in parallel, reloads the shift register with the new state value. The neural processor (PE) is very simple, comprising: a basic ALU, composed of a parallel adder, complementers and a shifter; the synaptic memory containing 64 9-bit words; and a one bit shift register (SR) for communication. The neuron sequencing is determined by the command lines, common to the whole network, which are issued by a central clock and sequencing unit.
Another interesting architecture, which also relies on circular shift-registers for the intercommunication structure, has been developed at Fujitsu Laboratories\(^6^1\). The architecture, which executes the Back Propagation algorithm\(^5^8,1^0^7,1^3^0\), consists of multiple processing units (PUs)\(\dagger\), and trays to implement the cyclic shift register. Each tray is connected to its two neighbours and functions as a container and a router for the data being transmitted. The basic structure of the architecture is depicted in Figure 2.3.

\[\text{Artificial Neural Network} \quad \text{Architectural Implementation}\]

\[\text{Input layer} \quad \text{hidden layer} \quad \text{output layer}\]

\[\dagger\text{ The expression } \textit{processing unit} \text{ is used throughout this thesis to designate a physical processor that can implement a number of virtual neurons. } \textit{Processing element} \text{ denotes a physical processor that evaluates one single neuron.}\]
The example in Figure 2.3 depicts a neural network composed of four neurons in the input layer, three in the hidden layer, and two in the output layer. The architectural implementation has four trays, three PUs, and no physical layer structure. Neurons at the same row (belonging to different layers) are mapped to the same tray and simulated successively by the associated PU. Therefore, each processing unit actually emulates more than one neuron. Each PU has a floating point multiplier, an adder and some local memory for programme and weight values. A prototype, called Sandy/8, has been developed using a floating point digital signal processor as PU. The prototype consists of 256 PUs, each with 2K words internal RAM and an external RAM of 64K words. The tray circuit was implemented using gate-arrays. The prototype is connected to a host machine (Sun-3) and the estimated performance is of 135 million connection updates per second. Since the system uses a (programmable) digital signal processor, other algorithms besides Back Propagation can also be programmed. Therefore, the architecture might also be considered of general purpose.

A different architecture that is also dedicated to the Back Propagation neural model has been developed by a group at University of Ancona, Italy\textsuperscript{101}. The proposed architecture is composed of a set of elementary processing elements connected in a linear sequence, where every PE communicates only with its two nearest neighbours. The processing element is formed by three basic units: I/O, execution and control units. The I/O unit, that deals with the communication, contains 3 input and 3 output data busses, of which two are used in the forward mode (left to right direction in the array) and one is used in the backward mode (right to left direction). The execution unit comprises the RAM memory for weights and inputs, an ALU able to perform add and multiply operations, a look-up table for the threshold function, and some registers to configure the network. Finally, the control unit controls the whole execution inside the neural processor. This "snake" architecture, as it is called, is a SIMD (Single-Instruction-stream, Multiple-Data-stream) machine since every processor performs the same operations on different data. The system can be configured in terms of number and width of layers, and can be expanded to as many processors as necessary. A more recent work has been carried out by the same group where they have studied the feasibility of restricting the weight values to power-of-two or sums of powers-of-two, in order to reduce silicon area and multiplication time. They have performed some tests on a simple pattern recognition problem and have shown the feasibility of the approach\textsuperscript{71}.

Lastly, a novel approach that applies stochastic functions of time to represent the state and weight values has been presented by Tomlinson and his team at Neural Semiconduc-
tor. The architecture is dedicated to the implementation of the recall phase, executing in hardware only the state calculation. Hence, all weight values have to be evaluated by simulation and then loaded into the system. The architecture, however, is very flexible, being able to realise any arbitrary network topology. The architectural structure is based on the assumption that the states are represented by firing frequencies and the weight is a stochastic pulse train which modifies the state frequency. With this approach, the multiplication computation can be implemented by AND gates, significantly reducing the silicon area. Two chips were designed to test the architectural approach. One chip contains the synaptic values, composed of a 32×32 matrix, and a stochastic pulse generator. The second chip implements 32 neurons, each comprising an integrator and a stochastic pulse generator.

As can be seen from the above examples, many research groups are developing specialised digital hardware for high performance execution of artificial neural models. Different approaches, to match the desired neural application, are being used for the topological distribution and for the processor's configuration. Despite these dissimilarities, there is a common tendency among all the proposed architectural systems regarding the communication strategy. All of them use a localised communication plan in contrast to the fully connectivity found in analogue designs. Additionally, the processor's structure is limited to one arithmetic unit, instead of providing hardware circuitry to execute the neural algorithm with full parallelism, which is one of the recognised powerful features attained by analogue circuits. The decision to apparently sacrifice the system's performance is due to the necessity to save silicon area and to produce expandable systems, as it is impossible, with current technology, to integrate large networks in one chip. Although this approach causes a moderate reduction in the efficiency of the system, the loss is compensated by the simplicity of the processing element. As the processor's hardware is developed to execute a specific algorithm, and as very high performance cells can be obtained with state-of-the-art technology, the final performance is generally very good.

The main drawback of the analogue/digital special-purpose neurocomputers, however, is their lack of flexibility. They are dedicated to one or a limited set of neural algorithms; if a different application that requires a diverging model must be executed, a different architecture will have to be used. Therefore, these architectures are very useful when the neural application is well defined and no substantial variations are expected.
2.1.2. General-Purpose Neurocomputers

Artificial neural networks are still under great evolution, with many different algorithms being frequently developed. Yet, no neural model is applicable to all possible neural network applications. Consequently, a flexible architecture, with the ability to implement a wide range of neural models and still afford higher performance than simulations on conventional machines, is required. These architectures are called general-purpose neurocomputers.

General-purpose neurocomputers are programmable machines that can support a wide spectrum of neural network models, thus providing a framework analogous to traditional computers.

General-purpose neurocomputers can be subdivided into commercial coprocessors boards, and parallel processor arrays\textsuperscript{120}. Commercial coprocessors usually consist of a single floating-point or signal processing accelerator board, generally based on a powerful processor supplied with a large memory. These boards plug into the backplane of an IBM PC or interface to a SUN Workstation or a DEC VAX. Parallel processor arrays, on the other hand, are cellular arrays\textsuperscript{111} composed of a large number of primitive processing units connected in a regular, and usually restricted, topology.

Accelerator boards, such as the HNC (Hecht-Nielsen Corporation) ANZA\textsuperscript{48}, and the SAIC (Scientific Applications International) SIGMA-1\textsuperscript{4}, implement large networks of virtual neurons by means of an industry-standard signal processing chip or microprocessor (such as the MC68020 in the ANZA system), interconnected through a standard parallel broadcast bus such as the VMEbus. The physical processor is multiplexed across a large number of virtual processing elements and virtual interconnections, demanding large memories to represent them. Performance comparisons between these products involve capacity (the maximum size of neural network) and speed (the time to process a neural network). For instance, the ANZA Plus\textsuperscript{48} supports 1M virtual processing elements, with 1M interconnections. It can perform 1.500 connection updates per second during learning and 6,000 updates during recall.

These programmable boards have provided a reasonable improvement in the execution performance of neural networks over the usual simulations of neural models. Nevertheless, many neural applications demand faster machines with an efficient exploitation of the intrinsic neural network parallelism. This demand has led to the design of general-purpose neurocomputers based on parallel processor arrays, constituting a natural evolution from the coprocessors boards. Parallel processor arrays optimise neural processing
by distributing the network through a larger number of processors, increasing therefore the final performance. Many different approaches are being followed, spanning from replication of the single processor accelerator board, through transputer-based networks, to the more recent systems based on custom-designed neuro-processors.

The earliest commercially available neurocomputer, the TRW Mark III, and its improved version Mark IV, were based on replicative accelerator boards. Mark III is composed of a maximum of 15 boards, using a similar approach to the 68020-based ANZA board as the essential replicative unit. Another example of this approach is the Computation Network Environment (CONE) developed at IBM's Scientific Center in Palo Alto. CONE is based on the Network Emulation Processor (NEP), a cascadable board that acts as a coprocessor for a PC host. The NEP board is based on the 5 MIPS (million instructions per second) TMS320 fixed-point digital signal processor and up to 256 boards can be connected to the interprocessor communication bus (NEPBUS). The complete system is able to simulate 1M nodes with up to 4M connections and perform 30 to 50 network updates per second.

A similar approach has also been followed by the developers of the Netsim system. The Netsim system, developed at Cambridge, consists of a collection of neural network emulator cards (Netsim cards) arranged in 3-dimensional array structures, with a host acting as a system controller. Each Netsim card is an autonomous single board which is also based on an industry-standard microprocessor (80188). Nevertheless, two additional custom-designed chips were also developed: a solution engine and a communication processor. The solution engine operates as a back-end vector coprocessor for the local microprocessor, performing mathematical functions upon the contents of the synapse/input memory. The engine basically computes the sum of products between the input vector and the associated synapse vector present in its memory. The standard microprocessor then computes the nonlinear function to produce the final neuron value. The communication processor implements the communication protocol, interconnecting Netsim cards to their nearest neighbours. Communication is performed via message-passing using a 64-bit message register, where 16 bits represent the destination address and the other 48 bits are used for data. The addressing scheme allows messages to be transmitted to around 15 Netsim cards in each of the three dimensions. The host acts as the system controller for the neurocomputer by providing initial conditions and defining the neural network properties which include the threshold function, network size and topology, and the input and training data.
Other research institutions are applying arrays of transputers to execute neural network applications. One of the first systems based on transputers was developed at the US National Aeronautics and Space Administration's Johnson Space Center in Houston, Texas. The system is called Neural Network Environment Transputer System (NNETS) and comprises forty 32-bit transputers interconnected via four full-duplex serial links. Other examples of transputer networks for simulating neural network applications include: at the University of Palermo, Italy, where a double linked ring topology has been chosen to interconnect 4, 8, 12, or 16 transputers; at the University of Marburg, Germany, where the ANNE (Another Neural Network Emulator) system has been developed, which allows an arbitrary size for the transputer net without having to recompile existing programmes; at the University of Zurich, Switzerland, which uses a board (Levco Translink II card) of 4 transputers connected to a Macintosh computer to simulate neural nets; and finally at Clarkson University, USA, where a board containing four transputers (called Quadputer and implemented by MicroWay) is used to execute Back Propagation networks of various sizes, for image processing applications. These transputer-based systems are usually integrated to a programming environment tool, with a high level network specification language and graphical facilities. These programming tools help to overcome the common difficulties found in programming transputer systems and provide a good environment for the simulation analysis.

Replicative boards and array of transputers have considerably increased the processing power for executing neural networks. However, the degree of parallelism is still limited as many virtual neurons are mapped into a restricted number of physical processors (processing units). For real-time and more complex applications, a larger number of processing units is required in order to attain a higher degree of parallelism and much more dense systems. Hence, many research groups are developing their own custom designed neural chips, achieving more compact designs with a substantial number of physical processors.

These custom-designed chips follow two distinct approaches that can be classified as **weight-based architectures**, which are based on systolic array techniques, and **neuron-based architectures**, where the physical processors are based on the neuron's functionality.

Systolic array techniques have been frequently used in neural network architectures. One of these architectures, called AAP-2, has been developed at NTT LSI Laboratories. The AAP-2 is a massively parallel computer composed of four major components: an array of 256x256 processing units; an array control unit that
broadcasts commands to all processors; a data buffer memory; and an interface unit to communicate with the host computer. Each processing unit is essentially composed of two versatile data transfer units, a one-bit ALU, and 8K bits of local memory. In addition, to increase the communication efficiency, the architecture provides three different mechanisms to perform data transfer between processors: connections to the nearest neighbour; connection through a programmed data transfer path; and communication via message passing. The AAP-2 architecture is able, through a folding mechanism, to simulate a network with up to 2 million connections.

![Diagram of system configuration for the NTT AAP-2 architecture.](image)

Figure 2.4: System configuration for the NTT AAP-2 architecture.

Important experiments have also been carried out using the Warp machine, a programmable systolic array developed at Carnegie Mellon University. Each cell in the Warp machine has a 5 MFLOPS (million floating-point operations per second) adder chip, a 5 MFLOPS multiplier chip and a 10 MIPS (million instructions per second) integer ALU. Each cell is capable of very high bandwidth communication (80 Mbytes/sec) with its left and right neighbours in the array. The communication between adjacent cells can be performed in parallel through two independent channels. The neural network investigations were performed utilising the common Warp configuration of a 10-processor linear array. Simulation have been implemented using the NETtalk network configuration, running the Back Propagation algorithm. An approximate rate of 17 million connections per second has been achieved by inserting a small change in the Back Propagation model, in order to obtain a better data partitioning that takes advantage of the Warp architectural features.
Two additional examples of weight-based architectures are presented in more details in section 2.2, namely the Siemens$^{104,105}$ and the Philips architectures$^{27,28}$. These systolic array architectures offer very fast computation of basic neural functions, like the vector-matrix multiplication required for the evaluation of the weighted sum of input values. This is due to the fact that these systems focus on the synapse functionality. Consequently, the hardware is basically formed by a 2-dimensional array of multipliers, used for processing the weighted values. As a direct effect from this hardware configuration, these architectures are not only limited to neural network implementations, being able to execute other signal processing applications, as in the case of the Warp and its improved version, the iWarp machine$^{23}$. However, some drawbacks arise from using the matrix configuration. These systems are more suited to fully connected neural networks, achieving with these regular configurations their best performance. When non-regular network topologies are used, like sparsely connected networks, the resultant system efficiency decreases$^{132}$. The use of high speed memories is also an important issue in these architectures, in order to afford high rates of data transfer between the processor array and the memory bank. Moreover, due to the two-dimensional topology, these architectures are non-scalable, constraining the network size to the medium scale range. Indeed, the non-scalable nature of this topology prevents the construction of a cost-effective network with a very large number of weight-based processors. Additionally, it binds the total number of processors that can be integrated into one chip to the maximum number of pads available in the fabrication technology. $^{105}$ Due to these intrinsic limitations of the weight-based architectures, a different approach is being followed by many research groups, which are developing neuron-based architectures. Instead of focusing on the synapse functionality, these systems centre their attention into the neuron’s operations. Many architectures have been developed using this method. The neural RISC machine designed at UCL$^{98,118}$ is one example. The system architecture consists of linear arrays connected in the form of rings to a host microcomputer. The communication is executed via two bidirectional, point-to-point links, and a simple protocol is used for broadcasting and routing packets. The basic processing unit is divided into three components: a communication unit, an execution unit and the local memory for programme and data. The neurocomputer is configured by the host which downloads simple programmes into each processor. This code can differ for each processing unit, providing a full MIMD machine.
Another neuron-based general-purpose neurocomputer, named CNAPS - Connected Network of Adaptive Processors, has been developed by Adaptive Solution Inc\textsuperscript{44}. The CNAPS system consists of a number of simple, digital processing units, operating in a SIMD configuration. Communication is executed through broadcast busses that provide high-performance intercommunication. Each processor is actually connected to three global busses (see Figure 2.5): an input, an output and a command bus (INBUS, OUTBUS and PNCMD respectively). There is also an inter-processor bus between adjacent processing units used to effect the arbitration algorithm for the output bus and to allow inter-processor data transfer.

![Figure 2.5: Adaptive Solution processor array system architecture.](image)

Each processing unit is composed of a simple arithmetic processor with an internal memory. Arithmetic operations are executed using fixed-point representation and internal busses are coded in 16 bits. The neural processor is optimised for traditional neural network applications, but is general enough to implement classical digital signal processing and rule based processing\textsuperscript{44}. The network of processors is supervised by an external sequencer controller chip that specifies, via the command bus, the instruction that must be executed by all processors. Adaptive Solutions claims that for 1-bit weights, the maximum performance reaches 12.8 billion connections per second, without learning.

Lastly, Hitachi has developed a neuron-based neurocomputer using wafer scale integration (WSI) technology. There is a trend towards the use of this technology in digital implementation of neural network applications. Wafer scale integration is capable of realising higher density and higher speed hardware than ordinary VLSI technology\textsuperscript{73}. However, the problem of manufacturing defects, commonly generated when using this technology, has prevented its wide use. However, neural networks contain redundant neurons which, in some cases, can compensate for faulty ones. Therefore, neural networks should be a good candidate for WSI implementation.
The Hitachi’s architecture is also based on a time sharing digital bus, controlled by a selector that broadcasts an address to determine the sender processor. The neural processor is divided into two modules, a synapse and a cell body. The synapse module comprises 64 weight values, coded in 8 bits, and a multiplier. However, a tree structure of processors can be constructed to increase this number of weights per processing unit. The cell body accumulates the weighted inputs, issued by the synapse module, and when all input values have been computed, it broadcasts its output to other processors through the selector command. The sigmoid function is implemented using look-up table which is designed to have redundancy in the wafer. The general WSI architecture is shown in Figure 2.6.

![Hitachi's WSI neural network architecture](image)

**Figure 2.6:** Hitachi’s WSI neural network architecture.

The silicon wafer is five inches in diameter and has been implemented using 0.8 μm CMOS gate array technology. The wafer is composed of 49 neuron chips and each neuron chip contains 12 processors. One of the neuron chips is used as spare, granting a total of 576 active processors. This first prototype did not include the learning procedure but another wafer, with learning circuits, is under fabrication. Even though some manufacturing defects in a wafer can be resolved through the intrinsic robustness of neural networks, some basic components have to be made redundant, like the learning control.
and the sigmoid function tables. When a fault occurs in these circuits, incorrect learning results, causing improper operation of the network\textsuperscript{135}.

Neuron-based architectures are usually designed to be easily expandable to a massive number of processors and to be scalable when feature size decreases. These characteristics allow for more processors in one chip (wafer) without affecting the number of pins. These systems also solve the problem of requiring high speed memories by integrating the weight memory into the chip area. This approach increases the performance because all processors access their memories in parallel. However, the number of weight values that can be stored is bounded by the size of the integrated memory, unless a mechanism like the tree structure implemented by Hitachi\textsuperscript{135,136} is used. The number of weight values per processor is further reduced if virtual neurons are mapped into one physical processor. To overcome problems with memory size, these architectures generally try to reduce as much as possible the neural processor area.

The next sections present a more detailed description of the four general-purpose neurocomputers under investigation in the \textit{Esprit II Galatea} project. The neurocomputers developed at Siemens in Munich\textsuperscript{104,105} and at the Laboratoires d'Electronique Philips (LEP) in Paris\textsuperscript{27,28} are presented in section 2.2 as weight-based architectures. Section 2.3 presents the architecture devised at Institut National Polytechnique de Grenoble (INPG)\textsuperscript{94-96} and a brief introduction to the generic neuron architecture proposed in this thesis as neuron-based architectures.

2.2. Weight-based Architectures

Weight-based architectures, as already introduced in the last section, focus on the synapse functionality, usually providing a matrix of multipliers. They normally apply systolic array techniques and some schemes have already been proposed to map neural model functions into matrix multiplication operations\textsuperscript{66}. In this section, the two weight-based architectures under analysis in the \textit{Galatea} project are introduced: the Siemens and Philips general-purpose neurocomputers.

2.2.1. Siemens Architecture

The architecture developed by U. Ramacher and his group at Siemens, Munich\textsuperscript{104,105}, to emulate neural network models, is based on systolic arrays. The basic idea is to generate a large network by means of a systolic array of small nets.

40
The design of the Siemens architecture centres on cascadable modules (chips) formed by a matrix of multipliers. Each multiplier represents a synapse, where the multiplication of the weight value by the neuron’s state takes place, and each column corresponds to one neuron unit. Figure 2.7 shows the conceptual structure of a 4-neuron module with 16 synapses. This module is composed of 4 columns, each comprising 4 multipliers. Every multiplier receives, in parallel, a weight and an input value. At the bottom of each column there is an adder, that sums the 4 weighted inputs calculated by the multipliers, and an accumulator, which stores the partial weighted sum.

Figure 2.7: Schema of the Siemens 4-neuron module with 16 multipliers.

Using the above 4-neuron module and assuming that one specific layer of the network is composed of M neurons and each neuron receives N inputs, the time schedule for receiving inputs and weights is depicted in Figure 2.8. A group of 16 weights and 4 inputs is presented, in parallel, at every cycle. It takes K (K = N/4) cycles for the first set of 4 neurons to calculate their states. Then, the state computation for the next 4 neurons in the layer can be evaluated. After T = (M/4).K cycles, the whole layer has been calculated.
This approach results in an important constraint. The number of neurons that can be implemented in one chip is directly determined by the number of pads needed to transfer the weights. For instance, in the case of the 4-neuron unit of Figure 2.7, if weight values are coded in 8 bits, 128 pads are necessary (4 neurons x 4 weights x 8 bits). To overcome this problem, the Siemens group proposed a different mapping of the weighting algorithm that circumvents this bottleneck without decreasing the computational throughput\textsuperscript{105}. The solution is acceptable when sequences of patterns can be processed at once, instead of evaluating each pattern separately. For neural network algorithms that allow this optimisation, the architecture is modified in two ways:

- the parallel weight inputs of each neuron (column) is transformed into a serial one, interleaving the scalar product computation for several patterns; and

- the parallel data input is converted to serial, interleaving the scalar product computation for several neurons.
As a consequence of this new configuration, the number of accumulators at the bottom of each column increases to 16. These accumulators now store the intermediate state value for 4 patterns, of a block of 4 neurons. Using this proposed configuration and with the addition of some extra circuitry, this architecture can also be applied to execute the learning procedure\textsuperscript{105}.

The Siemens group believes that, with present VLSI technology, this 4-neuron module can be implemented in one chip, with a resulting processing speed of about $5 \times 10^2$ MC/sec (Million Connections per Second, where 1 connection = 16 bits) at a clock rate of 40 MHz. A complete neural-emulator board is constructed by cascading many of these neuro-chips along with: DRAM banks for weight and data; an arithmetic unit to execute the threshold function; an external controller to load/read data; and an interface circuitry for communication with the host. The final emulator board is part of a VMEbus system which provides an easy interface between the emulator’s controller and the host, as well as facilitates the integration of commercially available interface boards, necessary for some applications (e.g. video/audio). Siemens expects to attain a processing power of about $10^5$ MC/sec at board level.

The Siemens architecture offers very good performance by using the matrix of multipliers and very fast memories. This high degree of parallelism however, is directly related to the assumptions of having full connectivity and the possibility of processing multiple patterns at the same time. Although the system architecture is flexible and general enough to implement a wide range of neural models, the performance is somewhat reduced if the previous assumptions can not be applied.

\subsection{2.2.2. Philips Architecture}

The Philips architecture is another general-purpose neuro-chip, designed to be highly efficient in matrix operations. It is a fully digital CMOS VLSI architecture that allows various kinds of neural networks with learning algorithms to be implemented\textsuperscript{27,28}.

Figure 2.9 shows the structure of the basic module in the architecture. The core of the basic module is the synaptic matrix memory, implemented in RAM to enable the modification of weights during the learning phase. This matrix contains $N^2$ weight values, coded in 2's complement over 8 or 16 bits, to execute a layer composed of $N$ neurons, each receiving $N$ inputs. The neuron state is coded over 1 to 8 bits.
The circuit of Figure 2.9 performs the scalar product for the evaluation of the $j^{th}$ neuron's next state, according to the following equation:

$$s_j = \sum_{i=1}^{N} S_i \times W_{ij}.$$ 

The multiplication of the weight values ($W_{ij}$) by one bit of the input states ($S_i$), and the addition of these products, are executed in parallel. The multiplication step $S_i \times W_{ij}$ is serialised over the different bits of $S_i$, avoiding the implementation of parallel multipliers. Instead, they use AND gates to compute the 8 by 1 bit multiplication, reducing silicon area. The partial products are accumulated in a shift register. With this approach, despite the fact that the module is able to store weights for $N$ neurons, each neuron in the layer is treated sequentially. This means that the circuit is able to calculate (in parallel) the weighted sum of only one neuron each time.

The threshold function, which is the final step to calculate the neuron's state, is evaluated off-chip, either by a dedicated hardware or by a standard processor. After the threshold function is performed, the new state is written back into the Dual Neural State Register (see Figure 2.9).

Other features that the circuit provides include:

- a Dual Neural State Register, for synchronous and asynchronous mode. If synchronous mode is being used, the evaluated state of a neuron is stored in a register until all states have been calculated. If, on the other hand, asynchronous mode is applied, the computed state is written directly into the register used for calculation.
- an arithmetic unit to accumulate internal as well as external partial products, permitting an easy cascading of a large number of chips. This feature allows the composition of layers with a larger number of neurons than can actually be implemented in one chip.

- a column of latches to enable free access to the synaptic matrix while the rest of the circuit performs an operation. This property is used when the circuit is executing the learning phase.

A first version of this architecture was implemented using 1.6μm CMOS technology, containing 64×64 synaptic memory, represented in 8 bits. This first version did not include the learning routine²⁷.

A more recent work²⁸ has described how learning algorithms can be executed using this architecture. They have stated that their architecture is more appropriate to what has been called local learning rules, where the updating of a weight value involves only information about neurons directly connected by that specific weight. The local learning step can be generalised by: 

\[ W_{ij}(t+1) = W_{ij}(t) + S_i A_j, \]

where \( A_j \) is a scalar value that depends only on the output neuron \( j \). These learning algorithms are easily mapped into the described architecture.

However, the implementation of non-local learning rules is more complex as it requires information distributed throughout the whole network. Therefore, for non-local algorithms like Back Propagation, they have devised a simple arrangement that overcomes this problem. The learning algorithm is decomposed into local stages, namely scalar product and local learning steps, to allow the usage of the same architecture. This is accomplished by using the transposed synaptic matrix to calculate the error value of the neuron. Hence, to implement the Back Propagation algorithm, several chips are cascaded: some of them evaluate the state update, utilising the \( W_{ij} \) synaptic matrix, while others compute the error values, making use of \( W_{ji} \) matrix. According to Philips, a VLSI chip, with learning procedure, can be developed in 1.6μm CMOS technology comprising a 32×32 synaptic matrix, coded in 16 bits.

The architecture is cascaded by connecting the Dual Neural State Register and the accumulator according to the required topology. The control of the network is also executed by an external controller, such as a transputer, through common broadcast busses, dedicated to transmit address, control and data.
The Philips architecture overcomes the necessity to have very fast memories (as happens in the Siemens case) by integrating the synaptic memory on-chip. This approach also solves the limitation in the pad number for parallel feeding of weights into the matrix of multipliers. The disadvantage is in terms of silicon area, as RAM memory usually takes a large amount of the chip area, reducing the possible number of processors that can be integrated in one chip.

2.3. Neuron-based Architectures

Neuron-based architectures, rather than concentrating on the matrix multiplication operations, emulate the neuron’s functionality in each processor. They generally try to integrate all weight values into an internal memory and execute all the necessary functions to evaluate the recall and learning phases. This section describes the INPG architecture and presents a brief introduction of the generic neuron architecture developed in this thesis.

2.3.1. INPG Architecture

Saucier and her group at Institut National Polytechnique de Grenoble - INPG - have developed a distributed, synchronous, neuron-based architecture to execute a large variety of artificial neural models\(^{94-96}\). The main goal is to provide an implementation framework to generate application-specific VLSI chips. The basic processing element is able to autonomously perform all steps of recall and learning phases and it can be customised to the required data precision (for weights and states) through silicon compilation.

Figure 2.10 shows the basic structure of the neural processor. It comprises 6 modules:

- an input register storing input values;
- an output register storing the output state;
- a local memory storing the weight values;
- a datapath able to perform multiplications and additions/subtractions;
- a controller to supervise the recall and, if implemented, the learning phase;
and some identification registers (ID_R) to configure, among others, the number of bits encoding state and weights and the number of weights in the memory.

Figure 2.10: INPG's basic neural processor structure.

The identification registers and the weight memory are implemented as RAM when the learning phase is executed on-chip. If the neural processor is used just for the recall phase, ROM technology must be utilised in order to reduce silicon area. The input and output registers are connected to two external busses and the input register (which is actually a shift register) transfers data from one bus to the other.

The final architecture is formed by grouping many processors into a two-dimensional array. In this array, processing elements (PEs) are organised in rows between two busses. These busses are made up of bus segments (see Figure 2.11) which are linked by software programmable switches, or soft switches. These soft switches are programmed by the processing element, during execution, to be open or closed, according to the neural computing phase being performed. Soft switches can be also permanently closed if, for that particular bus segment, no configuration is requested.

Data transfer is performed by the input register which, in each cycle, shifts data from one bus segment to the other. The direction of the shifting is defined by one of the identification registers, determining if the data shifting should be implemented downwards or upwards. Figure 2.12 helps to clarify this communication strategy, where a 2x4 processing element array is used to implement a fully connected, 8-processor layer, executing the weighted sum\textsuperscript{95,96}. In the first computational step, bus segments are connected as illustrated in Figure 2.12a. In this configuration, the odd-numbering inputs are firstly presented, with PE\textsubscript{1} and PE\textsubscript{2} receiving input in\textsubscript{1}, PE\textsubscript{3} and PE\textsubscript{4} receiving input
Figure 2.11: A partial view of the 2-D array of neural processors.

In Figure 2.11, PE₅ and PE₆ receiving input in₅, and PE₇ and PE₈ receiving input in₇. In the second step, a circular shifting is executed, producing the bus data configuration shown in Figure 2.12b.

Figure 2.12: 4x2 array of neural processors.

It is clear, from these two computation steps, that in order to allow all processors to receive all inputs (odd and even inputs), 8 elementary shiftings are required. Therefore, in order to evaluate the weighted sum of a layer containing N neurons, N elementary shiftings are necessary.
The sigmoid function is carried out through a polynomial expression approximation, although a look-up table can also be generated by the silicon compiler\textsuperscript{95}.

A first prototype has been implemented using 2\textmu m CMOS technology. The resulting area for one neural processor, executing just the recall phase\textsuperscript{95}, is $15\text{mm}^2$, and the processor is expected to run up to 20 MHz\textsuperscript{96}.

Saucier’s group intend to use this architecture in a silicon compiler environment, where the architectural parameters are extracted from a high level language. Some of these parameters are:

- number of inputs the neural processor should receive;
- precision for state and weight;
- the position of the neural processor in the physical layer;
- the shifting direction of the input values;
- the number of necessary shifts to format the multiplication’s result.

With the above parameters and block generators to implement the controller, memory and datapath, the architecture can be tuned to the required algorithm.

The architecture developed at INPG is very flexible and can run without waiting for commands from a central controller. This autonomy has the price of increasing the complexity of the processor, since the soft switches have to be controlled by the processing elements. In addition, the shift register communication imposes limitations in the number of processors per chip due to restrictions in pad number. This is a direct effect from the non-scalable matrix topology chosen.

As a conclusion to this review of VLSI architectures for neural networks, the next section presents a brief introduction to the \textit{generic neuron} architecture developed in this research. The \textit{generic neuron} architecture is fully described in Chapter 5.

\subsection*{2.3.2. UCL Architecture}

The \textit{generic neuron} architecture has also opted for the \textit{neuron-based} approach into the development of high-performance VLSI architectures for neural networks. The option has been made based on the major properties associated with this method: \textit{expandability} and \textit{scalability}, allowing the construction of cost-effective, massively parallel neurocomputers.
The *generic neuron* architecture\(^{121,125}\) was derived from the *generic-neuron* model presented in Chapter 1. The *generic-neuron* model synthesises the main features of neural network models into one single structure. The aim is to provide a general architectural framework that combines the high performance of special purpose hardware with the flexibility offered by general-purpose neurocomputers. This trade-off is achieved by providing a very simple, replicative processing element with user-defined functionality. A programmer uses a high level, neural network specification language (the \(nC\) language described in Chapter 4) to specify the network topology, the algorithm functions, and some basic parameters. These parameters include state and weight resolution. A silicon compiler then generates the application-specific architecture on silicon, with or without the learning procedure.

The architecture relies on the bus interconnection strategy for data communication. Communication occurs through a common broadcast bus, and each processing element gains access to the bus sequentially via central controller commands (see Figure 2.13a). The bus topology provides the desired trade-off between speed and other design requirements such as flexibility, expandability and scalability. The broadcast bus can effectively accomplish any desired topology and allows interconnection of a large number of processors. Moreover, as feature size decreases, the interconnection regularity supports the addition of more processors within the same chip without affecting the total number of pins.

![a) Overview of the generic neuron architecture.](image1)

![b) Processing element internal structure.](image2)

*Figure 2.13: Generic Neuron Architectural Framework.*
The processing element structure is divided into two logic units: the weight unity, which accomplishes the synaptic function; and the neuron unity, which carries out the neuron functions. The weight unit updates the weight values (when learning is being executed), and the neuron unit performs the calculation of the processor’s new state and error values. As discussed in Chapter 1, these three functions - state and error calculations, and weight updating - are the basic functions required in most neural network algorithms. The two logical units can be physically separated, with two datapaths and their respective controllers, or can be implemented into one single datapath and controller. The choice depends on a compromise between performance and silicon area. The final configuration, defined by the user, is implemented through the silicon compiler.

The physical processor, illustrated in Figure 2.13b, basically consists of: a memory bank, for storing the synaptic weights as well as the state and error values received from other neural processors; a datapath to execute the necessary operations for learning and recall; a programmable logic array (PLA) to implement the control of the whole processor; and finally a ROM memory with the look-up table for the threshold function. The size of this look-up table is specified by the designer.

The processing element basically interfaces with two busses, an input bus for reading input values (states and errors), and an output bus for sending the processor’s output values. It also receives control commands from additional busses, which specify, for instance, the owner of the data bus.

In addition to the modules mentioned above, some circuitry is also necessary to perform functions such as: reading and writing of data; address comparison; and central controller command analysis. This is executed in a communication unit which is not shown in Figure 2.13b for simplicity.

When a neural model requires bidirectional flow of data (as occurs with the Back Propagation algorithm), both processors involved in a connection require the associated weight value to execute internal functions, such as state and error calculation. To avoid transmitting the updated weight between processing elements, both interconnected processors have a copy of the weight value. This approach of duplicating the weight value in both processors, although increases the memory inside the chip, allows the data communication to be performed in broadcast mode in both forward and backward phases.

A prototype VLSI chip has been implemented in 2μm CMOS technology. The chip performs the Back Propagation algorithm, including the learning procedures.
The main benefit of this architecture is the conjunction of two important design features: flexibility and simplicity. This combination was achieved by the use of the broadcast bus communication strategy and the *generic neuron* model. A more detailed analysis of the *generic neuron* architecture can be found in Chapter 5.

### 2.4. Summary

This chapter has presented a review of VLSI architectures for artificial neural networks. The presented architectures, or *neurocomputers*, have been classified into two categories: *special-purpose* and *general-purpose* neurocomputers.

The *special-purpose* section has examined the two distinct approaches for implementing the architectures, analogue and digital techniques. Analogue designs are adequate for implementing smaller, pre-trained, application-specific neural networks. This implies that analogue systems are more suitable for non learning architectures, where all weight values have been evaluated off-chip. Besides the recognised problems of analogue circuits, such as low accuracy and noise susceptibility, the use of dedicated wires for transmitting state values makes the cascading of chips very difficult. For this reason, analogue neurocomputers are usually based on single chip applications. To overcome some of the analogue technical problems, some digital special-purpose neurocomputers have been produced. With the advent of smaller and faster digital VLSI technologies, a more flexible design can be easily achieved with a minimum increase in hardware complexity. Therefore, the current tendency in digital neurocomputers is to move from special-purpose to general-purpose architectures.

The *general-purpose* section has analysed the evolution in the development of flexible neurocomputers, from the early coprocessor boards that could be plugged into a portable computer, to the more powerful array processors. There have been two basic techniques for implementing the array processors: the coarse-grain approach, where a small array of transputers (or standard digital signal processors) are applied to simulations of neural network models with lower degree of parallelism; and the fine-grain approach, which makes use of a large number of processors, generally constructed from customised neural chips.

The higher parallelism of the fine-grain neural network dedicated chips is believed to be the answer to achieve more compact and faster general-purpose neurocomputers. This tendency has been confirmed by recent projects developed at important companies like Hitachi, Siemens and Adaptive Solutions. In conclusion, to attain a very high degree of parallelism and yet obtain a compact system, VLSI customised integrated circuits are
necessary. A further improvement to increase the parallelism is to produce neural networks based on WSI technology, as in the Hitachi system.

The ultimate progression into the development of a general as well as high performance system is viewed as a complete neural network programming environment, which integrates a silicon compiler for producing application specific neuro-chips. This environment, whose hardware design framework is the aim of this thesis, provides a general tool for developing and testing neural algorithms and, in conjunction, supplies an easy and fast route to application specific hardware, resulting into very high performance dedicated neurocomputers.

Therefore, the next two chapters describe the *Pygmalion* neural programming environment and its associated *nC* neural network specification language, which is used for describing the neural application to be translated into hardware. Then in Chapter 5, the silicon compiler’s architectural framework, namely the *generic neuron* architecture, will be fully analysed.
Chapter 3

*Pygmalion* Neural Programming Environment

This chapter presents the Esprit II Pygmalion programming environment. It provides a short overview of neural programming environments and then discusses the Pygmalion system, describing its network representation languages (N and nC) and execution methods (simulation and silicon compilation).

3.1. Neural Programming Environments

Programming systems for neural networks - *Neural Network Programming Environments* - have been under great expansion in recent years. Due to the continuous developments in the neural computing area, new algorithms are being developed to cope with a wider range of neural applications. Consequently, to experiment with existent as well as emergent neural models, a flexible, interactive software environment is required.

Many commercial and research institutions are engaged in developing these increasingly sophisticated neural programming environments. These software environments range from commercial products, obtainable from diverse companies, to public-domain software, available free from university research groups. According to the intended user and scope, these neural systems provide different characteristics \(^{122}\). The business professional user depends upon specialised *black box* utilities, targeted at particular applications. On the other hand, a neural network expert requires an open environment, in which any part of the system can be modified.

Related to the scope, neural programming environments can be classified into three categories: *application-oriented*, *algorithm-oriented* and *general* systems \(^ {122}\). *Application-oriented systems* are designed for business professionals that have little or no knowledge of neural algorithms. They are normally used in specific market domains such as finance and transportation. *Algorithm-oriented systems* can be: either a software tool where the user can experiment with different configurations of one specific neural model; or a library of parameterised algorithms that can be easily accessed to execute one particular application. Lastly, *general programming systems* provide a flexible tool for investigating a wide range of algorithms and applications.
General programming environments can be further divided into two sub-classes:

- **Educational Systems** - that provide, for the novice user, an introductory tool to learn the neural network theory. These systems are usually limited to the implementation of small networks, based on common algorithms. The PDP\(^7\) and Explorer\(^10\) systems are two of many examples developed for educational purposes;

- **General-Purpose Systems** - which are more sophisticated environments for programming any algorithm or application. Typical examples include the HNC\(^{43,49}\) and ANNE\(^{22}\) systems.

Table 3.1 exemplifies some of the well-known neural programming environments, presenting a short description of their basic features. A more detailed survey of neural network programming environments can be found in Treleaven’s paper\(^{122}\). This survey provides various case studies of research as well as commercial products.

As can be observed from Table 3.1 descriptions, despite the great variety of available general-purpose systems, they share some common features. A typical general-purpose environment executes on a host computer and generates a specific network simulation file to be executed directly on the host or on a specific target machine. Monitoring of the network’s evolution is carried out from the host through a graphical monitor and its corresponding command language.

A general-purpose programming environment, as illustrated by Figure 3.1, generally comprises: a *software environment* consisting of a graphic monitor and a command language for controlling the simulation; an *algorithm library* of the most common neural network models; a *high level language* for developing new neural networks; and an *intermediate-level network specification language* for holding the representation of a (partially) trained network.

The user defines the desired neural network application in the high level language, either by completely specifying the algorithm or, more likely, by modifying a parameterised model supplied in the algorithm library. This high level programme is translated into the intermediate-level network specification language. Then, this specification is compiled into a simulation file for execution on a specific machine. The execution of the simulation is controlled and monitored using the graphic monitor and its command language.

These sophisticated programming environments represent the current trend in systems for programming neural network applications. However, most of the commercial products
<table>
<thead>
<tr>
<th>Category</th>
<th>Organisation</th>
<th>System</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Educational Systems</td>
<td>UCSD</td>
<td>PDP</td>
<td>A simulation package(^2\textsuperscript{4}) that runs under MS DOS and UNIX operating systems. It provides a wide range of exercises for learning and practising with PDP models(^1\textsuperscript{07}).</td>
</tr>
<tr>
<td></td>
<td>NeuralWare</td>
<td>Explorer</td>
<td>Designed as a learning tool for small-application development. It provides on-line help and a check-point facility for saving and displaying the specified network(^1\textsuperscript{03}).</td>
</tr>
<tr>
<td>General Systems</td>
<td>IBM</td>
<td>CONE</td>
<td>A research environment comprising a graphical interface, high and intermediate network specification languages, and an interactive execution programme, running on an IBM PC. The network is executed on the NEP coprocessor board(^2\textsuperscript{4},4\textsuperscript{5}).</td>
</tr>
<tr>
<td></td>
<td>HNC</td>
<td>Anza/Axon</td>
<td>A commercial product specially designed for ANZA coprocessor boards. It is composed of a graphical user interface, the AXON description language and additional utilities to debug and explore the network parallelism(^4\textsuperscript{3},4\textsuperscript{9}).</td>
</tr>
<tr>
<td></td>
<td>SAIC</td>
<td>ANSpec</td>
<td>A commercial environment containing an algorithm library, an object-oriented programming language (ANSpec) and a graphical interface. It also includes the floating point accelerator DELTA board.(^4)</td>
</tr>
<tr>
<td></td>
<td>Oregon</td>
<td>ANNE</td>
<td>A simulation environment specifically designed for Intel's iPSC hyper-cube computer system(^2\textsuperscript{2}). It comprises network description and intermediate-level languages, and a mapper with a physical architecture descriptor (PAD) to partition the network amongst the iPSC processors. The PAD allows the ANNE system to be used with different architectures.</td>
</tr>
<tr>
<td></td>
<td>Adaptive Solutions</td>
<td>CNAPS/CodeNet</td>
<td>a commercial tool dedicated to the CNAPS neurocomputer(^5,4\textsuperscript{4}). It encompasses a programming language, a graphical user interface, an algorithm library and several related tools to debug and execute programmes(^5).</td>
</tr>
<tr>
<td></td>
<td>Lucid</td>
<td>Plexi</td>
<td>A commercial open environment where the user can modify any part of the system. It has a menu driven graphical interface running on Sun workstations(^1\textsuperscript{22}).</td>
</tr>
</tbody>
</table>

Table 3.1: Neural Network Programming Environments.

available in the market are often closed systems, making them relatively difficult to modify, enhance or interface with other software/hardware systems. One of the few exceptions is the Plexi open system\(^1\textsuperscript{22}\). In addition, current programming environments are usually tied to a specific target computers, such as a proprietary coprocessor board for a PC or SUN. Thus, once a neural network application is developed, it might be difficult to port/interface to another machine. The Esprit II Pygmalion Programming Environment has been developed to overcome these two limitations.
3.2. **Pygmalion Programming Environment**

The *Pygmalion* project\textsuperscript{16,17,20,123}, sponsored by the European Community Esprit Programme, has been developed to ensure the widest usage of the neural programming environment by making it as flexible and portable as possible.

The design philosophy of the *Pygmalion neural network programming environment* is threefold. Firstly, to provide a general environment for neural networks that includes the same facility and functionality encountered in commercial systems. Secondly, to produce an *open* programming environment - which can be easily extended and interfaced to other tools. For this reason the core of the environment is *X-windows*, *C* and *C++*, running on a colour workstation. Lastly, to provide *portable* neural network applications, so that trained and partially trained networks can be easily moved from one machine to another. As a consequence, the (partially) trained neural network applications are specified in a subset of *C* - essentially a *C* data structure.

The structure of the *Pygmalion* environment is composed of 5 major parts as shown in Figure 3.2:
- **Graphic Monitor** - The graphical software environment for controlling the execution and monitoring of a neural network simulation. It includes a command language to set up a simulation, monitor its execution, interactively change internal values, and save a trained network.

- **Algorithm Library** - The parameterised library of common neural networks, written in the high level language \((N)\), that provides users with a number of validated modules for constructing applications.

- **High Level Language \(N\)** - An object-oriented programming language that defines, in conjunction with the algorithm library, a neural network algorithm and application by describing the network topology and its dynamics.

- **Intermediate Level Language \(nC\)** - The intermediate-level, machine independent network specification language that represents the partially or fully trained neural network application.

- **Compilers** - Used to translate the user-specified neural network application into the target UNIX-based workstations and parallel Transputer-based machines.

![Figure 3.2: Structure of the Pygmalion Neural Programming Environment.](image-url)
The Graphic Monitor controls the simulation of a neural network application. It executes on a host computer and, through X window graphical tools and a command language, monitors the application evolution on a target machine.

As illustrated in Figure 3.2, the Graphic Monitor is directly related to the intermediate-level language nC. This language encompasses a hierarchical data structure, called system, comprising five basic levels: networks, layers, clusters, neurons and synapses. The Graphic Monitor reflects this hierarchical structure by providing a hierarchy of windows, each corresponding to one level in the data structure. Each item displayed in a window is associated with its equivalent nC specification.

The Graphic Monitor window environment provides pull-down menus to select and change: I/O format; network architecture; network learning algorithm; network execution; and display of neurons states and synaptic weights. It has two types of windows:

- **Top Window** - provides facilities for controlling the simulation and displaying status information as well as the network execution evolution;

- **Level Windows** - provides control facilities and displays status information (through graphical representation) for each hierarchical level.

The Algorithm Library contains the classical neural network algorithms in a parameterised specification, allowing them to be configured for a specific user application. It includes popular algorithms such as Hopfield, Back Propagation, and Boltzmann Machine. These neural algorithms specify the interconnection geometry and the basic functions associated with the neural model. In addition, users can select the number of processing elements, their initial state and weight values, learning rates, and time constants, to have the parameterised algorithm configured to the required application.

The other main parts of the Pygmalion programming environment - the high level language N, the intermediate level specification language nC, and the mapping of a neural application into target machines - are described in the next sections. Additionally, a full description of the nC syntax and basic features is provided in Chapter 4, since this language represents the neural network specification tool for the silicon compiler environment.

To ensure uniformity and consistency, all major parts of the Pygmalion environment interact through a common interface, which conforms with the following properties:
all modules are based on the \textit{nC} hierarchical structure system;

the algorithm library is composed of parameterised neural models;

algorithms and applications are independently defined;

algorithms and applications interface through common (\textit{nC}) data structures, function names and system variables.

The resulting uniform interface permits the development of generic applications and algorithms; one application may be executed by various algorithms, and one parameterised algorithm may be configured for many applications\textsuperscript{17}.

The environment provides four mechanisms for programming and controlling a neural network application execution:

- **Creation** - where the Graphic Monitor provides a window that allows the user to create a neural network system from scratch, using the high level language \textit{N}. This involves selecting a file from the algorithm/application library, or programming the algorithm/application in the textual language.

- **Modification** - where the Graphic Monitor allows the user to alter his neural network application during the simulation. Modifications are of two types. Changes to the network \textit{data} simply require modifications to the data stored in the simulation data structure, without affecting the \textit{N} and \textit{nC} specification. Changes to the \textit{topology}, \textit{functions} and \textit{control} of a network application, on the other hand, require the high level programme text to be modified (via a text editor) and a new \textit{nC} specification and simulation to be generated.

- **Execution** - where the Graphic Monitor provides the means to control the execution of a neural network system. This includes the control of the overall simulation as well as commands for accomplishing the input/output data transfer from files.

- **Monitoring** - where the Graphic Monitor allows the user to examine the status of an executing network. The two basic types of monitoring are: static - defining the current status, and dynamic - recording the history of the status.

The methodology of programming a neural network application using the \textit{Pygmalion} environment is illustrated by Figure 3.3. A neural network application is initially specified in the high-level language \textit{N} utilising modules from the \textit{Algorithm Library}. The resulting \textit{N} programme is compiled into the intermediate level \textit{nC} programme. At this stage the
neural network is a data structure that completely specifies the neural algorithm and the corresponding application. Subsequently, the \( nC \) specification is compiled into a binary file, for simulation onto a specific target machine.

![Diagram](image)

**Figure 3.3: Programming a Neural Network Application.**

After a simulation, two types of changes can be made to the application execution: changes to data values, such as system variables, which merely requires modification of the data stored in the simulation; and changes to the hierarchical structure, such as topology, which results in altering the \( N \) programme. In this case, the \( N \) programme should be re-compiled to generate a different \( nC \) specification. The Graphic Monitor can be used to dynamically modify the network, translating it to a particular target machine or saving the trained or partially trained network for later usage, possibly on a different target computer.

Having introduced the *Pygmalion* programming environment, next sections examine its two levels for representing neural networks - \( N \) and \( nC \) - and its two execution methods - simulation on a target machine or direct implementation on silicon.

### 3.3. Neural Network Representations in *Pygmalion*

There are two levels for representing a neural network in the *Pygmalion* programming environment: a high level language, called \( N \), that offers to the user a concise format for specifying the required neural network application; and an *intermediate level language*, named \( nC \), which is a machine-independent language used as the core of the *Pygmalion* environment.
3.3.1. \textit{N} High Level Language

The \textit{Pygmalion} high level language - \textit{N} - has been designed to allow its usage by both expert and naive users of neural networks. Users can define new neural algorithms or configure already operational algorithms to perform a certain application. The \textit{N} language has been developed as part of the \textit{Esprit II Pygmalion} project at Thomson-CSF, and the work is now being carried on at Mimetics, France\textsuperscript{70}.

The goal of the development team was to design a language that would allow a concise specification of neural network algorithms and applications. Consequently, the language follows the object-oriented approach, using a subset of C++\textsuperscript{116} as the basic syntax, with some additional neural-oriented features. The language is, therefore, based on autonomous components (\textit{objects}) that can be easily assembled to generate a neural network algorithm or application.

Due to the object-oriented methodology, the \textit{N} language permits a neural network algorithm to be specified in two ways: by defining specific types with their own data and functionality, in analogy to a class in C++; or by assembling already defined types in a modular tree hierarchy. These objects then communicate by sending execution messages through connection links between \textit{plugs}.

An \textit{object} in \textit{N} is basically composed of two main parts: \textit{memory} for data, and \textit{methods} to describe the execution procedure on the object's private data. The \textit{N} language also encompasses a library of predefined objects and algorithms that can be reused in new neural model specifications. Objects (types) already supplied in the language include: \textit{systemtype}, \textit{nettype}, \textit{layertype}, \textit{clustertype}, \textit{neurontype}, \textit{synapsetype} and \textit{somatype} for defining types of neural networks and their sub-structures. Therefore, a typical \textit{N} programme consists of a list of type definitions, allowing nesting of types i.e., one type to be defined from previously defined ones. A type definition has the following structure:

\begin{verbatim}
newtype xxx (parameter list)
parameter declarations
{
  plugin: list of variable declarations
  plugout: list of variable declarations
  above: list of shared variable declarations
  element: list of objects that will be defined under this object
  private: list of local variable definitions and local function definitions
  method: list of functions declarations and definitions
}
\end{verbatim}
The C++-like syntax of the N language permits that any programme can be easily translated into C or C++ programmes. Consequently, algorithms can be simulated on a sequential computer such as a PC or Sun workstation. The N programming language provides three types of facilities: to support the conversion of a N model into an abstract representation; to allow the re-use of previously defined algorithms; and finally, to supply tools to translate N applications into the intermediate-level \textit{nC} network specification.

3.3.2. \textit{nC} Specification Language

The core of the \textit{Pygmalion} environment is the \textit{nC} target language, developed at UCL. \textit{nC} is an intermediate level, machine-independent representation of neural networks. As a consequence of the machine independence feature, a neural network specified in \textit{nC} can be translated onto a variety of computers for simulation. Furthermore, a trained network can be stored in a library and mapped for execution in a different machine. Machine-independence is considered the major feature of the \textit{nC} language, and, to enhance this characteristic, the language syntax has been defined as a subset of C.

The \textit{nC} intermediate level representation divides the neural network information into four different domains:

- the network \textit{topology}, that describes the central hierarchical configuration as well as the connectivity of a neural network;
- the \textit{data} of the system, comprising the neuron's status and synaptic weights;
- the \textit{functions}, that defines the processing of the network; and
- the \textit{control} of the network activities.

These four domains are basically described by one special data structure: \textit{system}. The \textit{system} structure defines the \textit{topology} by giving the explicit location of each synapse and neuron inside the network. The \textit{system} is composed of networks that have layers, that are composed of clusters, that comprise neurons, that contain synapses. Alternatively, synapses can be allocated as a cluster sub-level to cope, for instance, with models that use shared weights. This hierarchy is very general, allowing the development of heterogeneous systems consisting, for instance, of a Hopfield and a Back Propagation network.

The \textit{system} structure also encloses the \textit{data} of the complete system. This comprises the neurons' output values and the synaptic weights. Lastly, \textit{functions} performed by each element in the system and the overall system \textit{control} are defined by the concept of rules.
Rules are incorporated in all levels of the *system* hierarchy and they specify the function that should be performed by a neuron, such as state calculation and weight update, as well as define the control of these functions. Consequently, the *system* data structure concentrates all the basic neural network informations: topology, data, function and control.

In addition, to cope with algorithm-dependent variables, a list of *user-parameters* is also provided. The following data structure framework illustrates, through the example of the layer level definition, how each level of the hierarchy is specified.

```c
typedef struct {
  int n_rules; /* # of rules */
  rule_type *rules; /* list of rules */
  int n_parameters; /* # of user parameters */
  para_type *parameters; /* list of user_parameter */
  (....) /* system variables (e.g. int clusters;) */
  (....) /* lower level elements (e.g. cluster_type *cluster;) */
  (....)_type; /* name of the level (e.g. layer) */
} (....)_type;
```

Finally, to overcome the inadequacy of the C syntax in dealing with explicit control of parallel execution, a new control operator - PAR - has been introduced in the *nc* syntax. This operator indicates to the *nc* compiler that the statements embraced by the PAR operator must be executed in parallel.

A detailed description of the *nc* neural network features is presented in Chapter 4.

### 3.4. Neural Network Execution Methods

The *Pygmalion* programming environment envisages two basic methods for executing neural network algorithms and applications. They can be either compiled and then simulated into different target machines, or directly translated into silicon for faster execution.

#### 3.4.1. Simulation

One of the main requirements in the development of the *Pygmalion* project was to attain a portable system. The use of such a system allows neural network algorithms and applications to be executed in a variety of different machines, including conventional and parallel architectures. Hence, a neural network that has been trained (or incompletely trained) in one particular machine, can be stored in the system and later recalled for re-evaluation in a different machine.

An additional benefit of a portable system is the possibility to evaluate a neural network in the architecture that best suits the application. Furthermore, it permits the implementation of heterogeneous networks with real parallelism, where each network is mapped into a different machine.
For this mapping of neural networks onto diverse machines to be effective, compilers that translate the \( nC \) machine-independent specification into the specific hardware must be available. The current version of the \textit{Pygmalion} programming environment supports mapping to sequential machines, such as SUN and DEC workstations, and to transputer-based systems where a better exploitation of the intrinsic neural network parallelism can be achieved\(^{21}\).

The compilation of neural networks onto transputer-based machines is the initial step in the investigation of the virtual-neurons-to-physical-processors mapping issue, in order to achieve the best performance. Extensions to this first approach are to be carried out in the \textit{Esprit II Galatea} project, where a general-purpose neurocomputer, composed of different boards, is to be implemented.

3.4.2. Silicon Execution

The automatic translation of application-specific integrated circuits (ASICs) from a high level network specification has also been examined in the \textit{Pygmalion} project. Two architectural frameworks have been devised, namely the INPG architecture\(^{94-96}\) presented in Chapter 2, and the \textit{generic neuron} architecture, outcome of this thesis, which is fully described in Chapter 5.

The integration of a silicon compiler is regarded as very advantageous\(^{122}\), since it grants many important features such as:

- suitability for non VLSI experts (access to custom VLSI for non VLSI experts);
- design at high level using behavioural neural network specification language;
- generation of high performance ASICs (optimised for speed and area);
- correct-by-construction design;
- greatly reduced time to manufacture custom neuro-chips.

Therefore, neural network dedicated ASICs can be produced in a very short time, providing high-performance neural chips with the highly attractive correct-by-construction feature.

The complete development and integration of the silicon compiler will be effected in the \textit{Galatea} programming environment, an extension of the \textit{Pygmalion} system.
3.5. Summary

This chapter has described the Pygmalion project, a programming environment funded by the European Community Esprit Programme. The project has involved many of the leading neural computing research groups from European industries, research institutes and universities. The chapter has also presented a short review of current programming environments available as research or commercial products.

The analysis of these programming environments has demonstrated the necessity to develop an open and portable system, where the user can easily change or expand the environment, as well as execute the neural network in many different machines. These were the two major requirements in the Pygmalion development. After examining the overall structure of the Pygmalion environment, the chapter has presented a brief description of the two neural network representation levels, available in the system, and the two execution procedures provided for the user.

Next chapter presents the description of the Pygmalion machine-independent language, nC, where the author of this thesis played a significant role in its specification and syntax checking (parser). The language has been designed as the core of the Pygmalion neural programming environment and will be used as the high-level interface between the user and an integrated silicon compiler. The implemented silicon compiler will automatically produce VLSI neuro-chips based on the generic neuron architectural framework.
Chapter 4

\textit{nC Neural Network Specification Language}

This chapter discusses the Pygmalion's machine-independent neural network specification language, called \textit{nC}. The chapter starts with a description of the \textit{nC} programming methodology and basic concepts. Then it examines how to programme an algorithm and an application, the two basic elements that fully specify a neural network in \textit{nC}. It also examines the \textit{nC} parser designed to check the language syntax.

4.1. \textit{nC} Programming

The \textit{nC} programming language developed in the \textit{Pygmalion} project\textsuperscript{16,17,20,123} acts as an intermediate level, machine-independent representation for artificial neural networks. It has been developed as the target language of the \textit{Pygmalion} programming environment to accommodate the diverse requirements of neural network programming\textsuperscript{122}.

Machine independence is considered the major feature of the \textit{Pygmalion nC} language\textsuperscript{17}. A neural network programme specified in \textit{nC} can be executed in a variety of target machines controlled by the \textit{Pygmalion}'s Graphic Monitor, or be automatically translated into application-specific neuro-chips, making use of an integrated silicon compiler. Therefore, to enhance the machine independence characteristic, the language has been defined as a subset of the ISO standard C language\textsuperscript{62}. The approach of using the C language for the hardware synthesis is also supported by another research group\textsuperscript{65,76}, which is investigating a C-based hardware description language for the high level synthesis of VLSI chips.

A \textit{nC} neural network programme comprises two main parts\textsuperscript{126}:

- \textit{Algorithm} - which includes the definition of a parameterised neural network algorithm that can be configured to implement different applications;

- \textit{Application} - which contains information about the network configuration and the algorithmic control.

The separation of algorithm and application expertise is very effective for the development of generic applications and generic algorithms. This approach allows a single appli-
cation to be used by different algorithms and a single parameterised algorithm to be experimented with various applications\textsuperscript{17}. Based on this division, the development of a complete $nC$ programme (application + algorithm) can be visualised as illustrated in Figure 4.1.

As shown above, a neural network execution (as a simulation or in silicon) is constructed from three basic units:

- **parameterised algorithm**, which is an application independent specification of a neural network model, where the programmer, making use of the hierarchical structure $\text{system}$ (described in section 4.2.2), implements basic system functions like weight update, learn and recall;

- **application configuration**, where the application programmer basically configures the chosen algorithm, supplying the correct values for the algorithm’s parameters, such as number of nets, layers and neurons in each layer;

- **application program**, where the application builder specifies the sequence of calls to the system functions. These functions control the algorithm’s initialisation, training, recall, saving etc.

Hence, the first basic element in the development of a complete neural network programme in $nC$, is the definition of a parameterised neural network algorithm (see Figure 4.1). A neural network algorithm, either extracted from the algorithm library or
developed by the algorithm user, is essentially composed of a set of parameterised functions based on a built-in, hierarchical data structure, called system. These functions allow a neural network algorithm to be coded independently of the topological configuration and the network parameters.

The next component in the specification of a complete neural network execution is the definition of the network configuration. This definition encompasses topological information and initialisation of parameter values necessary to transform the parameterised neural algorithm into a fixed neural network model. A fixed neural model is, therefore, the assembly of a parameterised algorithm with an application configuration. The resultant neural model is a "static" image of the parameterised algorithm, with fixed number of elements and with the correct initialisation for weights and states.

The last necessary element to produce a complete programme for execution is the specification of the application control, which indicates how the neural model should be executed. This last part is defined by the application programmer who specifies how and which algorithm functions should be executed to perform the desired application. This control can be executed with or without the Graphic Monitor command.

As mentioned in the previous chapter, the nC neural network information is divided into four different domains: topology, data, functions and control. These four domains are incorporated into the hierarchical data structure system and in the rule structure definition. These two basic concepts, system and rule, are the essential entities which define nC as a neural network dedicated language.

4.2. nC Basic Concepts

This section explains the fundamental concepts of the nC programming language. These are: the rule concept, that embodies the functionality and controllability of a neural network model; and the system data structure that groups into one structure all neural network information.

4.2.1. Rule Concept

The information about function and control in a nC neural network programme is defined by the concept of rules. Rules can be visualised in two levels: the functional level, or simply rule level; and the control level, or meta-rule level. These two levels are defined through the same data structure, namely RULE. The basic specification of the RULE structure is shown below.
As illustrated above, the `RULE` structure comprises, essentially, three fields:

- **name** - defines the name of the rule, which is used for display purposes in the Graphic Monitor.
- **class** - contains, among other fields, a pointer to a procedure that specifies the rule’s functionality/controllability. The `class_type` data structure is described later in this section.
- **para_list** - is an array that contains the procedure’s parameters list. The `caddr_t` type allows the parameters to be pointers to any type specifier (int, float etc), including user defined types, such as `RULE`.

The basic elements of the `RULE` structure are then a function pointer (accessed indirectly via the `class_type` structure) and a list of parameter pointers (see Figure 4.2). The function pointer (`fun_ptr`) points to a basic procedure that is applied to the parameter list (`p`) when the rule is executed. At the rule level, (where functionality is specified), the parameters are numeric values, upon which the function performs a specific calculation. On the other hand, as shown in Figure 4.3, at the meta-rule level (where controllability is defined) the parameters are pointers to other rules. In this case, the associated function determines how these rules should be executed. There are basically two ways of executing rules: either sequentially, using the `sexec` built-in function; or in parallel, utilising the `pexec` built-in function.

As exemplified by the `neuron.state_upd` rule in Figure 4.2, the parameter list at rule-level can be divided into two blocks, namely **Generic Parameters** and **Extended Parameters**. The former includes the fundamental numeric parameters that must be present in all rules of one specific class. In this example, the neuron’s output state (`state`) and the accumulator (`acc`) are the generic parameters for the `State_Update` function. The extended parameters, on the other hand, are related to the variable part (concerning the size of the list) of the rule, such as the input states (`S_i`) and their associated weights (`W_i`). Owing to this division, the `class_type` structure was created, in order to specify a group of rules that contain the same set of generic parameters as well as the same function body.
The class_type structure, shown below, is composed of: a function pointer (*fn) to access the function body; a string containing the function's name (*fn_name); the number of generic parameters (n_generic_parameters); and the number of extended parameters (n_extend_parameters).

```c
typedef struct {
    int (*fn)();
    char *fn_name;
    int n_generic_parameters;
    int n_extend_parameters;
    class_type;
} class_type;
```

The n_extend_parameters field in the class_type structure does not contain the total number of extended parameters, since this number varies according to the network configuration. In fact, it contains the number of elements in the replicated group. This idea can be clarified by analysing Figure 4.2. As shown in the parameter list, the extend-
ed parameters part comprises, besides the \textit{size} field, groups of two elements each: the state of an input neuron \((S_i)\) and the associated weight value \((W_i)\). This group is "extended" to the correct amount which, in this case, depends on the number of input neurons. The number of groups in the extended parameters part is stored in the \textit{size} field, which must always be the first one in the extended parameters list.

The algorithm programmer should declare and initialise all different rule classes required for the particular neural network model being implemented. More details about the rule concept and its related data structures can be found in the \textit{nC Manual}\textsuperscript{126}.

4.2.2. Generic Hierarchical Structure - \textit{system}

The key to an algorithm specification in \textit{nC} is the hierarchical data structure \textit{system}. This structure defines the complete configuration of a neural network model, giving the explicit location of each synapse and neuron inside the whole network. The \textit{system} structure is fixed for all neural network algorithms and no modification is permitted, since it also provides the common interface by which the Graphic Monitor controls and accesses the network data.

The \textit{system} hierarchical structure is composed of 5 sub-levels achieving the required generality to implement any topological configuration. It is composed of one or more \textit{networks} that have \textit{layers}, that are composed of \textit{clusters}, that comprise \textit{neurons}, that finally contain \textit{synapses}. Alternatively, synapses can be allocated at the cluster sub-level to cope, for instance, with applications that use shared weights. The hierarchical structure of the \textit{nC} language is shown in Figure 4.4.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{system_diagram.png}
\caption{The \textit{nC} hierarchical structure \textit{system}.}
\end{figure}
In addition to the usually supported layer and neuron sub-levels, the nC language has also incorporated the net sub-level, in order to allow the development of heterogeneous neural models consisting, for instance, of a Hopfield and a Back Propagation network. Furthermore, some models, such as the Competitive Learning, assume a sub-division at the layer level, where competing neurons are assembled into separated groups. Inside these groups, only the winner neuron fires. To be able to distinguish these groups from normal layers, the cluster sub-level has been included. The cluster sub-level can also be used to assemble neurons belonging to the same layer into groups that exhibit similar behaviour.

The system hierarchy embodies the 4 basic neural network domains (topology, data, function and control) in one single structure. The topology is expressed by the system’s five levels: net, layer, cluster, neuron and synapse. The data is incorporated into the neuron and synapse sub-levels, which contain the neuron’s output state and the synaptic weight. Functions and control are specified by a list of rules, included in each sub-level of the system structure. The five system sub-levels share a common framework composed of:

- a list of rules;
- a list of user-parameters;
- a list of system-parameters; and
- a list of lower level structures.

The list of rules defines the control and functionality associated to the hierarchy’s sub-levels. The list of user-parameters encloses the algorithm dependent variables, like the learning rate and tolerance parameters implemented at the net level of the Back Propagation algorithm. The list of system-parameters contains the algorithm independent parameters, associated with the specific system’s sub-level. Finally, each system’s sub-level holds the pointers to the lower-level structure’s elements like clusters inside layers. The complete system specification is shown in Figure 4.5.

By analysing the system structure declaration, it can be noticed that all sub-levels follow a common framework definition. User-parameters are declared as a list of para_type structures, which comprises two basic units: the parameter’s name (as a string) and the parameter’s value. The algorithm programmer can create any algorithm dependent parameter and the Pygmalion Graphic Monitor can still monitor the parameters by their associated names. The para_type structure definition can be found in the nC Manual.
As shown in Figure 4.5, all lists are declared as pointers instead of arrays\(^6\). This feature allows all lists to have unfixed size at the time of declaration. So, although the system structure declaration is fixed for all algorithms, the application builder can configure it to have any number of elements in the hierarchy’s sub-levels. This type of list, however, provides no memory allocation at declaration time. Therefore, all sub-levels of the system structure must have their necessary memory allocated at run time, according to the application configuration. This is accomplished by two functions, namely connect and build_rules, which are coded by the algorithm programmer.
The synapse_type, cluster_type, layer_type and system_type, as shown in Figure 4.5, include, in addition to rules and parameters lists, simply the list of lower level elements, like weight inside synapses and clusters inside layers. The cluster_type allow synapses and neurons as lower level elements. The algorithm programmer should choose where to allocate the synapses, either in the neuron level or in the cluster level, and programme the algorithm accordingly.

The neuron_type and net_type sub-level structures, on the other hand, have more complex structures. The neuron_type has six additional system parameters:

- **state[NO_STATES]** array - provides memory storage for the neuron’s output state and other relevant values, such as an accumulator for the weighted sum. The size of this array is algorithm dependent and the programmer should specify the NO_STATES value for each different algorithm. For the Back Propagation algorithm, for instance, the array is formed by the neuron’s output state, its error value, the target output value, and an accumulator for the weighted sum.

- the **route[4]** array - uniquely identifies a neuron inside the whole system structure. The four fields of this array define: number of the net that the neuron belongs; number of the layer inside the net; number of the cluster inside the layer, and finally the neuron’s index inside the cluster.

- **fanin** and **fanout** fields - define, for each particular neuron, the number of inputs and outputs the neuron contains.

- the **input_neuron** and **output_neuron** fields - are lists of pointers to neuron structures, providing the connectivity information.

The net_type structure has also some additional parameters: the **fanin** and **fanout** parameters, which hold the number of input and output neurons of that specific net, and three list of pointers - **input_port**, **output_port** and **target** - which contain the input, output and target pattern values, respectively.

The system hierarchy and the concept of rules provide the required generality in conjunction with efficient execution. Moreover, they produce a fully parameterised neural network algorithm, implying that it can be employed by many different applications.

The third basic concept incorporated into the nC language is the PAR operator, presented in the next section.
4.2.3. \textit{nC} Parallel Execution Operator - PAR

In addition to the common \textit{C} statements, a new operator, namely PAR, has been introduced into the \textit{nC} language, in order to have an explicit control statement for indicating parallel execution. The \textbf{PAR} control operator indicates that all instructions within the following open-and-close brackets should be executed in parallel. Furthermore, the \textbf{PAR} operator also serves as an operation modifier when applied to a \textit{for} statement. It converts an iterative \textit{for} loop into a replication operation. The two examples presented below help to clarify the use of this new operator.

\textit{Example 1:}

\begin{verbatim}
PAR
{
    statement1;
    statement2;
}
\end{verbatim}

\textit{Example 2:}

\begin{verbatim}
PAR for (i=0; i < config.layer[j].cluster.neurons; i++)
    statement;
\end{verbatim}

In example 1, the two statements are executed in parallel, while in example 2 the statement within the \textit{for} loop is replicated for all neurons (index \textit{i}) in the specific layer \textit{j}. This explicit control of execution provides the necessary information to guide the mapping of \textit{nC} programmes into different parallel machines and into silicon.

The following sections discusses the important issues on how to programme the two essential modules of a neural network specification in \textit{nC}: the parameterised algorithm and the application definition.

4.3. Programming an Application

To write an application the designer should specify two basic modules: the \textit{application configuration}, to obtain a fully configured and initialised neural model, and the \textit{application control} to direct the neural model execution. These two application parts, together with the parameterised algorithm, form a complete programme for execution.

4.3.1. Application Configuration Definition

The application configuration part contains the network configuration and some parameter definitions. These definitions can be implemented in two different ways. Firstly by creating a file called \texttt{config.h}, which must be compiled together with the neural algorithm every time one of the declared values needs to be changed. The second method is
provided to overcome this drawback of having to recompile the whole programme when just a small change in a parameter value is required. Hence, a file (currently called test) is generated every time a neural model is executed for the first time. This file is then read back when a new run of the neural model is desired. If, for instance, the learning rate should be modified, the user edits the test file and changes the setting of the learning rate parameter. A new configuration can then be tested, without requiring recompilation of the neural network model.

In addition to parameter values and network configuration, the test file can contain initialisation for weights and states of a neural model. This enables the user to run a partially trained neural model, utilising weight values that have been previously calculated.

To configure a parameterised algorithm for the first time however, the application builder should generate the config.h. This file basically comprises the config structure declaration and initialisation shown in Figure 4.6. An example of a config.h file can be found in Appendix 4.

**config_decla:**

```c
struct
  { int nets;
    struct { int layers;
      struct { int clusters;
        struct { int neurons;
          } cluster [CLUSTERS];
        } layer [LAYERS];
      } net [NETS];
    } config = { NETS,
      { /* number of layers in net 1 */,
        { /* number of clusters in layer 1 */,
          { /* number of neurons in cluster 1 */},
          { /* number of neurons in cluster 2 */},
              ...;
          { /* number of neurons in cluster CLUSTERS */}
        },
        ...;
        /* number of cluster in layer LAYERS */,
        { /* number of neurons in cluster 1 */},
        { /* number of neurons in cluster 2 */},
          ...;
        { /* number of neurons in cluster CLUSTERS */}
      },
      /* number of layers in net 2 */,
      
      ...;
    },
    /* number of layer in net NETS */,
    
    ...;
  };
```

Figure 4.6: config general framework.
This structure establishes the number of elements for each sub-level of the system structure, like number of layers inside each network and number of neurons inside each cluster, completely configuring the network topology.

The next step to fully define an application is the application control specification, described in the following section.

4.3.2. Application Control Definition

In the application control part, the programmer specifies the application dependent control structures, which determines the neural model execution. There are essentially two ways of defining the application control of a neural model:

- **Menu driven via Graphic Monitor** (GM), where the user selects, in the GM windows, the desired functions to be executed, like load, learn and recall.

- **main C routine**, where the user programmes the main routine to control the neural model. In this mode, the neural model simulation is executed independently from the Graphic Monitor.

The structure of the main routine is basically composed of function calls and loops/iterations statements to control the functions' execution. Functions are of two types: system functions, that deal mainly with input/output of data; and algorithm functions that are the routines defined in the parameterised neural network model. The system functions are algorithm independent, provided as part of the built-in library. Examples of system functions are: init_pattern, read_input, read_target, sys_load and sys_save. The first three handle the inputting of patterns, while the last two analyse the test file.

Fundamentally, any algorithm function defined in the neural network algorithm can be called in the application programme part. However, there are four compulsory algorithm functions that should be included in all application controls:

- **connect** - allocates memory for all sub-levels of the system structure and assigns the correct pointers to establish the network connectivity.

- **build_rules** - constructs the rules that each system sub-level should perform.

- **Learn** - learns the pattern that has been previously stored in the system structure through system functions.
• Recall - recalls one pattern, that was previously loaded in the system structure.

The connect and build_rules routines must be called before any other algorithm function. Apart from the constraint of having to call connect and build_rules to fully initialise the system data structure, the application programmer can use full C syntax to determine the application's execution.

4.4. Programming an Algorithm

A neural network algorithm programmed in nC should be specified in a parameterised way to allow it to be used by a wide range of different applications. Therefore, the algorithm programmer must conform with the following nC principles:

• make use of the general, built-in, hierarchical data structure system;
• programme all lower level functions, or rule functions, using the nC syntax allowed;
• code connect and build_rules procedures to allocate memory and assign the correct pointers for the system structure.

The system hierarchical structure should be included in all algorithms coded in nC, using the sysdef.h file that contains all built-in structures of the nC language (see Appendix 4).

The next essential step in the development of a new neural network algorithm is to code the necessary rule functions. To code a rule, the programmer should initially decide what are the necessary parameters and then use the allowed statements to code the rule function. More details about the syntax allowed to code the rule function bodies can be encountered in the nC Manual126. Appendix 4 contains the complete specification of the Back Propagation algorithm coded in nC.

The last basic element to programme an algorithm is the declaration of the connect and build_rules routines, discussed in the next section.

4.4.1. Coding connect and build-rules

The basic element in the developing of a new algorithm is the coding of two important procedures - connect and build_rules - that: allocate the correct amount of memory in the system structure; assign the correct pointers for the neurons connectivity; and extend all different rules. More specifically:
- **connect** - converts the general system structure declaration into a specific memory structure, using the set of constants provided by the config structure in the application part. It also assigns the correct pointers for the system connectivity.

- **build_rules** - uses the network topology set by the connect routine to construct all the necessary rules, at rule and meta-rule levels. It basically attributes the rule function pointer and assigns the correct pointers for the rule’s parameter list.

The **connect** and **build_rules** functions basically generate the whole system structure in the memory, with all pointers set to the correct address. These two functions expand the general symbolic structures into an explicit form. The names of these two functions are compulsory but the function bodies are algorithm dependent and must be coded correctly to generate the desired topology.

The **connect** procedure must: (1) allocate memory for the lower level structures, such as allocating the correct number of neurons for all clusters in the system; (2) must allocate memory for all the necessary rules and parameters in the system sub-levels; and finally, (3) it should establish the neuron’s connectivity. This is accomplished by setting the correct pointers to the input_neuron and output_neuron lists of each neuron in the system.

The **build_rules** routine essentially constructs the rules for all sub-levels of the system structure, assigning all the correct pointers. The algorithm programmer writes this routine using two built-in functions available in the built_in_fn.c file:

- **rule_init** - Initialise a rule, setting all the correct values for rule’s name, class, function pointer and generic parameters. Additionally, it allocates memory for the extended parameters.

- **rule_extend** - Sets the correct pointer for the rule’s extended parameters, using the memory previously allocated in the **rule_init** function.

### 4.5. nC Parser

As already stated, the nC language designed in the Pygmalion programming environment is based on a fixed hierarchical structure that embodies all the required information to describe a neural network. This hierarchical structure is the core of the programming environment, which the Graphic Monitor, the compilers to target machines and the silicon compiler depend upon. Therefore, even though the nC language is founded on the C syn-
tax, a mechanism to verify the \textit{nC} basic and fixed concepts is necessary. To perform this task, a \textit{nC parser} has been developed.

The \textit{nC parser} is a syntax checker for neural network algorithms and applications written in the \textit{Pygmalion nC} language. It reads an \textit{nC} programme and analyses it to find out whether its syntax conforms to the \textit{nC} language specification. If the user has not correctly defined the neural network according to the \textit{nC} directives, the programme notifies the error that must be corrected.

The parser is divided in three basic modules:

(i) \textit{config parser} - basically verifies the \textit{config} data structure specification, which has been defined by the application builder in the configuration file \textit{config.h};

(ii) \textit{sysdef parser} - validates the entire set of \textit{nC} special data structures used in a neural network algorithm specification. It essentially analyses the \textit{sysdef.h} file to check the system hierarchical structure and rule related data structures;

(iii) \textit{nC parser} - the main module that analyses the whole specification of the application and parameterised algorithm. It examines if the \textit{config.h} and \textit{sysdef.h} files have been included and calls their associated checker. In addition, it investigates the \textit{class_type} declarations, the rule function definitions, the existence of the \textit{connect} and \textit{build_rules}, and finally, if present, the \textbf{main} function syntax.

If any of the above modules recognises an error in the syntax format, the user is informed about the file and respective line in which the error has occurred, in order to take the appropriated measures. After the syntax checking succeeds, the user can be assured that the \textit{nC} programme can be correctly compiled and executed under the \textit{Pygmalion Graphic Monitor} control.

The \textit{nC parser} has been developed making use of the UNIX tools for lexical and grammar analysis, namely Lex\textsuperscript{67} and Yacc\textsuperscript{57}, respectively. Appendix 1 presents the \textit{nC} grammar used for the \textit{nC parser} implementation.

4.6. Summary

This chapter has introduced the basic concepts of the \textit{Pygmalion} neural network specification language, called \textit{nC}. This target language has been designed as part of the \textit{Esprit II Pygmalion} project to provide a concise and efficient form for executing neural network algorithms in different machines, either based on sequential or parallel architec-
tures. Additionally, the language should be adequate to the automatic translation of neural applications into ASIC neuro-chips, through a behavioural hardware synthesis tool.

The language, which is based on a subset of the C syntax, has been developed to conform with the following basic requisites:

- **neural net independence** - to permit a clear representation of a wide range of neural network applications, covering all stages of specification, training and usage;

- **target machine independence** - to allow a neural network specified in the language to be efficiently mapped onto a range of sequential and parallel computers, as well as automatically translated into application-specific VLSI chips;

- **algorithm/application separation** - to enable the development of generic applications and generic algorithms, where a certain application can be evaluated by different parameterised algorithms and a single algorithm can be configured to implement many applications.

To comply with the above requirements, the language has been defined as based on a hierarchical data structure that encompasses the whole neural network information in terms of data, connectivity, functions and control. This data structure, called **system**, together with the definition of the **rule** functions, allows a neural network application to be simulated in one of the available target machines. Additionally, with the integration of the silicon compiler, the neural application will be automatically compiled into silicon for faster execution.

The silicon compilation approach for implementing high performance ASICs requires, in addition to the neural network specification language, a target VLSI architecture. This architecture should provide a framework that includes the main features encountered in artificial neural networks, in order to be configured to implement the desired application. Such architectural framework, which is the main goal of this thesis, is described in the next chapter.
Chapter 5

Generic Neuron Architecture & Silicon Compilation

This chapter gives a description of the architectural framework developed in this research. It presents the generic neuron model, its associated VLSI architecture, and how different neural algorithms can be represented with the proposed model. It also describes the architectural parameters used by the silicon compiler to generate different neural network application-specific chips.

5.1. Design Approach

As stated in Chapter 1, the ultimate goal of this research is to achieve a behavioural VLSI design automation system, dedicated to artificial neural networks. The system is based on the Pygmalion environment and its associated neural network description language nC (described in the last two chapters), and also on the development of an integrated silicon compiler that will automatically generate specialised neural chips from the nC network specification language.

Current progress in the development of silicon compilers indicates that they usually tend to automate the lower levels (physical and gate) of the layout design, freeing the designer from having a detailed understanding of the VLSI design process. These silicon compilers are normally general purpose systems that can be applied to various different applications. Nonetheless, these systems still demand a considerable knowledge of electronics design, forcing the user to utilise a hardware description language to describe the application’s functionality. Moreover, due to their generality feature, these silicon compilers typically produce larger and lower performance circuits than a full custom design since, to achieve generality, they commonly make use of standard cell libraries to produce the final chip.

Therefore, to develop higher level synthesis tools that automatically generate compact and high performance application-specific integrated circuits (ASICs) from a high level description language (behavioural domain), it is crucial to specify a target VLSI architectural framework that encompasses the main features of the aimed application. The specification of such architectural framework for a silicon compiler environment allows the production of more optimised circuits in terms of silicon area, as well as makes the
utilisation of a behavioural silicon compiler feasible, since the architectural framework reduces the degree of variability.

To devise such architectural framework, a clear understanding of the basic features related to the specific application is required. In the case of neural network applications, the main properties of neural algorithms should be analysed in order to define an adequate target architecture. The architectural framework's design should provide: *generality* to implement the large variety of available neural algorithms; and *simplicity* to optimise the processing element's silicon area and performance. Consequently, prior to describing the VLSI target architecture conceived in this research, the following section presents the *generic neuron* model that embodies the main properties of neural networks. This is the basic model that led to the definition of the *generic neuron* VLSI architectural framework.

### 5.2. Generic Neuron Description

The continuous improvements in artificial neural networks have led to the development of a large variety of different neural algorithms, based on diverse topological arrangements and learning rules. The interconnection pattern of these models ranges from the simple one layer configuration, to the more complex multi-layer, forward/backward data communication scheme. Recall and learning procedures also span over an extensive spectrum, where a number of different functions, based on dissimilar mathematical models, specify how weights should be updated and states and errors calculated.

The *generic neuron* model has been devised to incorporate, into one single structure, these diverging aspects of neural network algorithms, both in terms of topology and functionality. The *generic neuron* model, introduced in Chapter 1 and reproduced in Figure 5.1 for the sake of clarity, provides four ports for external communication: 1) a set of state inputs $S$, where the specific neuron collects the state inputs received from the linked neurons; 2) a state output $s$, used to broadcast the neuron's output state to the appropriate neurons; and 3) a group of error inputs $E$ and 4) an error output $e$ to cope with models that require backward flow of data, such as the Back Propagation\textsuperscript{58,107,130} algorithm.

In terms of its internal functionality, the *generic neuron* model includes a table of weight values $W$ (associated with both groups of inputs - $S$, $E$ - for the forward and backward path, respectively) and three different functions: the activation function $f_1$, that evaluates the neuron's output state; the weight updating function $f_2$; and the error calculation function $f_3$.

84
The activation function $f_1$ is performed in two consecutive steps: the propagation rule, that defines how the state inputs are modified by the associated weight values; and the threshold function that makes use of the propagation rule's result to obtain the actual output state. Hence, the $f_1$ function depends on two basic parameters, $S$ and $W$:

$$s = f_1(S, W) \quad (5.1)$$

The weight updating function $f_2$, executed only during the learning phase, is typically a function of the weight values $W$, the state inputs $S$ and the neuron's output error $e$:

$$\Delta W = f_2(S, e, W) \quad (5.2)$$

The generic neuron's error value $e$, utilised by the weight updating function, is evaluated by the error calculation function $f_3$. This function usually depends on the input errors $E$, the neuron's output state $s$ and, in some cases, also on the weight values $W$:

$$e = f_3(s, E, W) \quad (5.3)$$

Most neural algorithms can be described by defining these three functions and by specifying the network's topology. Table 5.1 illustrates this idea by expressing, in accordance with the generic neuron's functions above, eight of the most popular neural algorithms. The next paragraphs presents a brief description of these algorithms and Table 5.1 demonstrates the mapping of these algorithms onto the generic neuron model by changing the configuration of its internal parameters.

As shown in Table 5.1, the spectrum of different neural network models is quite extensive. The network topology parameter can range from single layer, feed-forward models, such as Perceptron, through single layer with feedback connections, like the Hopfield algorithm, to more complex interconnection patterns, found in the multi-layer
## Neural Network Model

<table>
<thead>
<tr>
<th>Neural Network Model</th>
<th>Network Topology</th>
<th>range of input values</th>
<th>Recall/Learning Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>f1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Propagation Rule</td>
</tr>
<tr>
<td>Hopfield/Kohonen</td>
<td>single-layer with feedback</td>
<td>binary</td>
<td>net = \sum S.W</td>
</tr>
<tr>
<td>Perceptron</td>
<td>single-layer feed-forward</td>
<td>binary or continuous</td>
<td>net = \sum S.W</td>
</tr>
<tr>
<td>Widrow-Hoff (Delta Rule)</td>
<td>single-layer feed-forward</td>
<td>continuous</td>
<td>net = \sum S.W</td>
</tr>
<tr>
<td>Back Propagation</td>
<td>multi-layer bidirectional links</td>
<td>continuous</td>
<td>net = \sum S.W</td>
</tr>
<tr>
<td>Boltzmann Machine</td>
<td>multi-layer or randomly connected</td>
<td>binary</td>
<td>net = \sum S.W</td>
</tr>
<tr>
<td>Counter Propagation</td>
<td>multi-layer feed forward</td>
<td>binary</td>
<td>net = \sum S.W</td>
</tr>
<tr>
<td>Self-Organising Map</td>
<td>two-dimensional grid of outputs PE's</td>
<td>continuous</td>
<td>net = \sum S.W</td>
</tr>
<tr>
<td>Neocognitron</td>
<td>hierarchical multi-layer feed forward</td>
<td>continuous</td>
<td>net = ( \frac{(1+S_i \cdot W_e)}{(1+S_h \cdot W_h)} )</td>
</tr>
</tbody>
</table>

### Notation:
- \( w_{ij} \) = weight from PE; to PE;j
- \( s_i \) = state from PE;i
- \( e_j \) = output error of PE;j (\( e_o \) for output layer, \( e_h \) for hidden layer)
- \( t_j \) = target output of PE;j
- \( \eta \) = the learning rate, a constant between 0 and 1
- \( T'(net) \) = derivative of the threshold function
- N.A. = not applicable

### Table 5.1: The mapping of popular neural models into the generic neuron model.

**Bidirectional links of the Back Propagation** \(^{58,107,130}\) and two-dimensional grid of neurons of the Self-Organising Map \(^{64,68}\).

The activation function (f1), formed by the propagation rule and threshold function, also presents some variation among the current neural algorithms. The propagation rule basically calculates the weighted sum of the input states, although the Neocognitron model \(^{34,35}\) applies a more complex function of input states and weight values, which are classified into excitatory (\( S_e, W_e \)) and inhibitory (\( S_h, W_h \)) values. The threshold function is usually restricted to one of the three functions shown previously in Figure 1.3 (hard limiter, pseudo-linear and sigmoid families).

Apart from the network topology, however, the learning procedure is probably the most characteristic feature of a neural network model. In algorithms such as Hopfield/Kohonen's associative memories \(^{53,63,64}\), the learning algorithm does not in-
volve any error calculation (shown in Table 5.1 as N.A.), having the weight values set based on the Hebbian rule. This rule establishes that when the two nodes embracing a connection are active, the associated connection weight should be strengthened\textsuperscript{68,107}.

Nonetheless, the learning algorithm generally depends upon an error calculation (f3) and a subsequent weight updating function (f2). Error calculation functions can drastically vary among the neural algorithms. They can be as simple as the difference between a target value and the neuron’s output state (Perceptron\textsuperscript{79} and Delta Rule\textsuperscript{68} algorithms in Table 5.1), or involve more complex equations according to the mathematical modeling being used. In the Back Propagation algorithm, for instance, the error calculation also requires the derivative of the threshold function ($T'(net)$).

The Boltzmann Machine learning algorithm, on the other hand, uses the simulated annealing technique\textsuperscript{6,107} to modify the synaptic weights. With this technique, the weight values are adjusted by applying the Boltzmann distribution with a temperature parameter to determine the slope of the probability function. The learning algorithm is performed in two consecutive phases. The first phase employs supervised learning, where target patterns are fed into the output neurons and the network is stabilised using the current set of weights. An unsupervised phase is then utilised, where no target output is presented but the system again calculates its output states until equilibrium is reached. The weights are subsequently updated using statistical information obtained from the two previous phases (see Table 5.1): $<p_{ij}>$ the average probability that PEs $i$ and $j$ are active during the supervised phase; and $<p'_{ij}>$ the average probability that both PEs are active during the unsupervised phase. This process is continued until the average change in the weight values becomes zero.

The Counter Propagation algorithm\textsuperscript{130} (which employs competitive learning\textsuperscript{25} technique in one of its layers) also accomplishes the learning procedure in two phases. It is a multi-layer feed forward network, that employs two different learning rules for each of its layers. In phase one the weights from the input layer to the hidden layer ($\Delta w_{ij1}$) are updated using the Kohonen unsupervised algorithm, which forces only one neuron to be active in the hidden layer. The weights from the hidden layer to the output layer ($\Delta w_{ij2}$) are updated in a second phase using the supervised Grossberg algorithm (see equations in Table 5.1).

As can be seen from the above discussion, although the range of functions varies considerably from one neural algorithm to another, the generic neuron model provides the desired flexibility to describe a wide selection of neural models. The generic neuron
model supports the description of recall and learning procedures of various algorithms by merely tuning its internal parameters configuration. Based on this flexible model, the **generic neuron** architectural framework was devised, providing a simple and general target architecture for the integrated silicon compiler.

5.3. Architectural Framework

The **generic neuron** architectural framework was designed to incorporate all basic features encountered in current neural network algorithms, in order to implement a wide range of algorithms/applications. Therefore, the chosen architecture consolidates the basic concepts defined in the **generic neuron** model. The **generic neuron** architecture seeks to combine flexibility and high performance to provide a cost effective architectural design. To achieve this trade-off, some basic requirements have been imposed: the communication strategy should cope with any network topology; and the processing element’s structure should provide an optimised framework to attain high performance and minimum silicon area.

The following two sub-sections present the developed target architecture according to the adopted communication strategy and the proposed PE’s structure.

5.3.1. Communication Strategy

The interconnection strategy is regarded as one of the main issues in neurocomputer development, due to the neural networks’ inherent massive connectivity. In order to afford the implementation of current as well as novel neural algorithms, three main requisites were established for the interconnection strategy: flexibility, scalability and expandability.

The huge amount of connections inside a neural network and the considerable number of different interconnection patterns utilised by the neural algorithms demand an interconnection strategy that provides the necessary flexibility. The interconnection scheme must be able to implement complex topologies, adapting itself according to the required application. In addition, it should handle all the connections with minimum overhead in communication protocols and routing algorithms.

Design’s scalability is also a mandatory issue to reduce the cost per interconnection in large networks. The interconnection scheme should be regular to afford, without affecting the total number of pads, more PEs per chip when denser packing technologies are available.
Lastly, the number of recommended PEs in a neural network is application dependent, making *expandability* an important requirement. The user should be able to develop neural networks with a large number of processing elements, without prohibitively increasing the cost of the interconnection structure.

Based on the above requirements, an investigation of the possible interconnection schemes has been effected, trying to attain high performance communication without compromising its cost-effectiveness. Network topologies like the 2-dimensional mesh and the hypercube\textsuperscript{13,30} provide high connectivity per processing element but this advantage undermine two of the above basic requirements: scalability and expandability. Due to their higher dimensional topology, these interconnection strategies are not suitable for VLSI architectures with massive number of PEs, since the number of PEs per chip is limited by the maximum number of available pins in the integrated circuit. To overcome this limitation, these interconnection structures either restrict the connection width to narrow or even serial links, which considerably reduces their performance\textsuperscript{26}, or implement a different input mapping to achieve the same performance\textsuperscript{105}, which restricts the applicability of the design. For the same reason, these networks are not easily expandable off-chip, making impractical the construction of very large networks.

Dynamic topologies like the crossbar and the multi-stage networks\textsuperscript{13} have also been discarded due to the high cost of their interconnection network and the complexity of the switch design, which make them unsuitable for neural network applications. The crossbar topology is extremely expensive because of its high-level of connectivity, making its utilisation impossible in massively parallel systems. The multi-stage networks, on the other hand, have to either increase the complexity of the switch or expand the number of switch stages to avoid message blocking during simultaneous data transfers, which compromise the final network efficiency. Additionally, these topologies have also limited scalability (as the 2-dimensional mesh and the hypercube) in terms of VLSI design, since a separate connection must be provided per integrated PE to maintain the intrinsic performance associated with the interconnection scheme.

From this analysis, it was felt that the best compromise to provide the three mentioned requisites and to achieve a cost-effective interconnection scheme is the broadcast *bus* strategy. The system, shown in Figure 5.2, is composed of many processing elements connected through common broadcast busses to a central controller (a host machine). Each processing element gains access to the bus sequentially by means of central controller commands that define which PE can write data onto the bus.
The interface between central controller and processing elements is composed of three basic busses: data, address and control. The data bus, whose size (8, 16 bits, etc) can be configured by the silicon compiler to match the application builder's request, is used basically to transfer states and error values. During initialisation, the data bus is also utilised by the central controller to load into the processing element's internal memory all necessary data values, including initial random weights and internal parameters like learning rate.

The address bus, which can only be written by the central controller, specifies the owner of the data bus i.e., which processing element has the permission to place data (state or error value) into the data bus. All other PEs in the network examine the address bus to investigate if the data value on the data bus is relevant and should be read or ignored.

The address bus is divided in two fields which specify, respectively, the layer and the neuron inside the layer. The central controller should contain information about the total number of layers and the total number of neurons in each layer of the network. Through this information and with the benefit of two counters, the central controller defines, for each layer, the neuron which must send the output to the network. When all processors belonging to that particular layer have sent their output values, the layer counter is increased, the neuron counter is reset, and the counting is restarted.
The central controller makes use of a polling mechanism to decide if its internal counters can be increased. After placing a new value into the address bus, the central controller waits for a ready signal from the appropriate processing element. The address bus can only be modified to select a new PE after receiving this signal. Otherwise, it remains with the same value, until the related PE signs to the central controller that an output value has been deposited into the data bus. This ready signal is also used by other PEs in the network to verify if there is a valid data in the data bus. The employment of this polling mechanism permits the usage of a faster clocking period, improving the overall performance of the communication. Otherwise, the central controller clocking period would have to be set in order to accommodate the slowest operation (probably the multiplication) in one cycle. This would generate a large amount of idle periods when faster operations were being executed.

A timeout mechanism must also be implemented by the central controller in order to acquire a more robust system. The timeout technique helps to prevent an indefinite hold-up of the network while the central controller waits for a defective processing element to become ready. After the stipulated time has elapsed, the central controller should activate the ready line and send a null value to the data bus, avoiding other processing elements in the network from being affected by the defective processor.

Finally, the control bus contains all significant signals that direct the network's behaviour. It comprises commands from the central controller to processing elements and vice-versa, including signals such as reset, forward/backward phase definition, the PEs' ready signal, etc. The purpose of these signals will be better described later in this chapter (section 5.4.2).

The role of the central controller is therefore twofold: to perform initialisation of the whole network, providing initial weight values and some additional configuration parameters; and to manage the data communication in the network, by selecting the appropriate processing element to send its output value.

This bus strategy provides the vital requirements for the general-purpose hardware implementation of neural networks. It supports the necessary flexibility, being able to effectively realise any desired neural network topology. The bus interconnection provides low cost communication with the facility to access any processing element in the network with the same latency.
Its regular design also contributes to the \textit{scalability} property. As can be seen from Figure 5.3, with the constant improvements in VLSI technology, more processing elements can be packed onto the same integrated circuit, without affecting the total number of pins.

Due to the broadcast mode of communication, data is transferred between processing elements with no data routing. Each output value (state or error) is transmitted to all related processing elements in one single cycle, which considerably simplifies the communication circuitry. This is, in fact, the most efficient multiplexed interconnection for large fan-in and fanout architectures\textsuperscript{44}. Furthermore, this scheme greatly facilitates the wiring mechanism, avoiding many on-chip wiring problems encountered in other non-regular communication strategies that largely increase the necessary silicon area\textsuperscript{44}.

The main disadvantage of the bus topology, however, is that it achieves low cost, flexible and scalable communication scheme at the expense of providing lower expandability. Although theoretically the bus strategy can be expandable to any number of processing elements, in reality its size is limited by inherent difficulties, such as the propagation delay and the low bandwidth when many PEs are connected.

In order to reduce the degradation effect when large number of PEs must be interconnected, the \textit{generic neuron} architecture supports the implementation of multiple busses\textsuperscript{42}, where PEs that receive the same input (for instance, PEs of the same layer), are clustered onto the same data bus. This extended configuration is only possible due to the processing elements’ separated input and output data busses. This improved architectural framework is exemplified in Figure 5.4 with a multi-layered neural network.

In this new extended interconnection strategy, the architecture is formed by multiple busses according to the number of layers requested by the neural algorithm being imple-
mented. A more complex central controller is hence needed to correctly load and address all the present data busses. This new approach is quite useful in cases that allow multiple patterns to be treated simultaneously, where the data is pipelined through the layers.

To improve even further the bus performance, the processing element has been designed to support broadcast mode in both forward and backward data flow, reducing the traffic load on the bus. This mechanism will be explained in the next section.

5.3.2. Processing Element Overview

The processing element’s structure has been defined to interface with the chosen communication strategy and to reproduce the general properties of the generic neuron model. Its design observed the two main issues of the silicon compiler environment: simplicity, to attain minimum silicon area; and flexibility, to accommodate the diversity of neural algorithms, including the learning process.

To achieve these two main requirements, some fundamental features have been formulated, namely:

- **modularity** - in order to avoid paying the cost, time and pin count penalties caused by off-chip communication, the architectural specification has favoured the design of a complete processing element on chip. Therefore, the design of the processing element has resulted in a self-contained structure, composed of communication, execution, and memory units.

- **digital design** - this design approach is fundamental if generality is to be achieved, not only in terms of topology but also related to the processing element’s functionality concerning recall and learning phases. As discussed in Chapter 2, analogue
neurocomputers are more application specific, having their weight values calculated off-chip by a host machine, and then loaded into the neurocomputer using diverse techniques. The implementation of on-chip learning, utilising analogue devices, however, still remains a difficult issue. Therefore, a digital design has been chosen.

- **high performance** - to increase the final performance of the *generic neuron* architecture, the processing element final structure is tuned to the specific application defined by the silicon compiler's user. The self-contained structure also serves to increase the network performance since, by providing on chip memory, communication, and execution units, the load on the communication resources is substantially reduced\(^{26,42}\).

The resultant processing element structure is shown in Figure 5.5. It is divided in three basic units: *weight unit*, which accomplishes the synaptic function; *neuron unit* that carries out the neuron functionality; and *communication unit* that performs the reading and writing of input and output data.

![Figure 5.5: The processing element's internal structure.](image)

The *weight unit* essentially executes the updating of the synaptic weights, being active only during the learning phase. It comprises the control operations to implement the weight updating function \(f_2\), and a number of memory blocks to represent each connection. These memory blocks contain the three necessary parameters of equation (5.2): the weight value to be updated \((w_{ij})\); the input state \((s_i)\); and the output error \((e_j)\). To provide the required generality, the *generic neuron* model (Figure 5.1) encompasses two groups of inputs \((S\ and\ E)\) and, consequently, two sets of associated weight values. Therefore, the weight unit should provide two different types of memory blocks: *for-
ward blocks, related to the state-input weights, and backward blocks, associated with the error-input weights. The forward block parameters consist of the input state of the connection, the associated weight value, and the PE's error calculated in the neuron unit. Correspondently, the backward block comprises the error input of the connection, its associated weight, and the PE's output state evaluated in the neuron unit.

Backward memory blocks have been included, in spite of increasing the necessary size of the internal memory, in order to afford broadcast communication in the backward transmission phase. When the neural network's flow of data is bidirectional, both processing elements involved in the connection require the weight value to evaluate their internal functions. To refrain from having to transmit the updated weight between processing elements, the memory block associated with this particular connection is duplicated in both interconnected PEs. So, for each forward block in a specific PE, there is a replicated backward block in the other PE of the connection. This approach significantly improves the network performance, since data communication can be performed in broadcast mode both in forward as well as in backward stage. Other architectures that only integrate the forward weights associated to a particular PE\(^6\), have their performance reduced during the backward phase. In addition, as will be seen in Chapter 8, although the backward blocks increase the required internal memory, this is compensated by the simplicity of the PE's structure, resulting in a reasonable number of PEs per chip with state-of-the-art processing technology.

The neuron unit (see Figure 5.5) basically performs the calculation of the processing element's output state and error value. It accomplishes the two components of the activation function \( f_1 \) - propagation rule and threshold function - and the error calculation function \( f_3 \). According to equation (5.1) and the activation functions exemplified in Table 5.1, to calculate the propagation rule the neuron unit should receive the relevant input states and modify them by their associated weight values (S.W). In addition, as a second step in the state calculation \((s_j)\), a threshold function should be performed upon the result of the propagation rule. The look-up table technique has been chosen to implement such threshold function, where a ROM memory stores the digitised values of the function. This approach seems to be the tendency in neurocomputer design\(^{44,132,136}\), since the threshold function can be evaluated with one single memory access, improving the overall performance. In this case, the table size is specified by the silicon compiler's user, according to the precision requirements.

Similarly, in conformity with equation (5.3), for the error calculation function \( f_3 \) to evaluate the PE's error value \((e_j)\), the neuron unit requires its own output state \((s_j)\) and its
input errors, usually modified by their associated weights. Therefore, the neuron unit should also have access to these parameters to execute the error calculation function.

Finally, the communication unit controls the data communication of the processing element. It interfaces with the central controller, as well as with other processing elements, via two independent data busses, an input and an output bus. The availability of these two unidirectional busses (instead of one bidirectional bus) enables the system to be configured with one or multiple data busses, in order to match the performance requirements.

The final configuration of the generic neuron architectural framework, however, is highly dependent on the application builder, which trades off the performance and the silicon area required for the particular neural network. Each additional feature for the processing element's functionality should be analysed and considered against the increase in terms of the silicon area. Therefore, the user, via interactions with the silicon compiler, should decide which architectural configuration is best suited for the specific application.

Next section describes in more details the basic components of a processing element's implementation. Section 5.5 then presents some architectural parameters that can be configured by the silicon compiler to generate different application-specific neural chips.

5.4. Processing Element Framework Implementation

After the previous overview of the processing element basic requirements, this section gives a more detailed description of the architectural implementation of a processing element using, in some cases, the implemented Back Propagation prototype to exemplify its internal organisation.

Based on the essential requisites discussed in the previous section, the VLSI generic neuron architecture has been designed as a self-contained element, containing three basic modules: memory unit, communication unit and execution unit. The basic structure is depicted in Figure 5.6.

The memory unit contains the necessary memory blocks to store the algorithm dependent parameters (states, errors and weights values). The addressing mechanism to read/write data from/into the memory unit is implemented by an addressing module, which receives commands from either the communication or the execution unit. The communication unit implements all functions associated with transferring information to/from the central controller and other processing elements. It also executes the initialisation process, which
reads data from the bus and stores them into the on-chip memory space. The execution unit is the entity which actually executes the neural network functions. This module can be physically implemented as one single unit or, to improve internal parallelism, be composed of two separate units, allowing simultaneous calculation of independent portions of the neural network functions. The following sub-sections examine each of these three units separately.

5.4.1. Memory Unit

The basic purpose of the memory unit is to store all relevant data required to perform the neural network application. Data access is controlled both by the communication and execution units. The former accesses the memory unit to store all incoming data related to the processing element, while the latter accesses it to perform the appropriate mathematical operation upon its data values. Since it is desirable to allow the communication unit to store new input data independently from whatever operation the execution unit is performing, a mechanism to avoid conflict in the memory access must be provided. This has been achieved by using the two-phase clock mechanism described in Chapter 7. In this case, each of these two units should have a determined clock phase to completely read/write data from/to the memory unit, supporting totally separated memory accesses during the same clock cycle. In the Back Propagation prototype, for instance, the communication unit has access to the memory unit during $\Phi_1$, while the execution unit has access during $\Phi_2$. With this approach, the execution unit can start the necessary calculation without having to wait for all pertinent values to be collected. The communication unit can, for example, activate the weighted sum function immediately after receiving the first input state, allowing the execution unit to carry on with the calculation while the
communication unit continues collecting additional input states. The memory unit's internal organisation is depicted in Figure 5.7. It is composed of two basic modules: the *storage module*, where all the necessary data values are stored; and the *addressing module*, which provides the sequential addressing mechanism to access the memory space.

![Diagram of memory unit's internal structure](image)

**Figure 5.7: The memory unit's internal structure.**

**Storage Module**

This module contains the complete set of data values required to execute the neural network model. These values are related to the processing element's input connections and their associated synaptic weights. Although the final configuration is highly dependent on the neural algorithm/application being implemented, four basic blocks can be identified, according to the *generic neuron* model discussed in section 5.2: two blocks associated with the forward path, storing the input states (S) and their weight values (W_F); and two additional blocks for the backward flow of data, storing the input errors (E) and their related weights (W_B). These four memory blocks contain the necessary data information to accomplish the majority of neural models, as indicated by equations (5.1), (5.2) and (5.3). By organising different subsets of these four blocks, the processing element can be configured to the desired neural application. If, for instance, the application does not require the execution of the learning phase, or if the learning procedure does not involve back propagation of errors, the backward path memory blocks are excluded from the im-
plementation. If, however, the learning phase must be implemented with back propagation of errors, the complete set of memory blocks should be integrated into the processing element.

As can be seen from Figure 5.7, two data busses are provided for the storage module. The four memory blocks are arranged in order to afford maximum parallelism for the memory access. Since neural functions usually depend on calculations upon the input data set (states or errors) and their associated weight values, performance can be substantially improved if separate data busses to access both values in parallel are provided. Furthermore, owing to the fact that no simultaneous access is effected to both forward and backward blocks, they can share the same set of data busses, without compromising the overall performance. This approach also conforms with the execution unit design, which follows the common tendency of containing two main busses to perform most of the data manipulation.

Figure 5.7 also indicates the structure of the addressing mechanism. As shown in the diagram, all memory blocks share the same address bus, which appreciably simplifies the addressing circuitry. The control of which memory block should be activated at each particular point of time is defined by either the communication unit or the execution unit, according to their respective clock phases.

**Addressing Module**

This module provides a sequential pointer to address all memory blocks integrated into the processing element. It can be controlled by either the communication unit, to store input values into the internal memory, or by the execution unit, usually to access the internal data to carry out the required calculation. The addressing module’s basic components are illustrated in the left part of Figure 5.7.

The module consists of a counter and up to two comparators, depending on whether the storage module contains just forward, or both forward and backward memory blocks. The counter effects the sequential memory addressing, while the comparators provide the means to determine when all relevant input values have been analysed. Each comparator (for the forward and backward paths) embodies an associated register that contains the total number of inputs. Likewise other hardware related parameters, these registers can be initialised by two different methods: either by having fixed values, determined at (silicon) compilation time; or by receiving their appropriate values from the central controller, during the initialisation phase. The first method is utilised when a fully connected network is being implemented, where all processing elements of a particular layer re-
ceive the same number of input values. This approach provides a considerable reduction on the initialisation time and on the necessary silicon area, without affecting the requisite of producing identical chips. In the case where each processing element comprises different number of inputs, the second method should be implemented, at the expense of a longer initialisation phase.

Besides having the memory address generated by this addressing module, the memory unit’s address bus can also be driven directly from the external address bus, through the communication unit’s command. This mechanism is employed by the communication unit during data acquisition from other processing elements. Since the external address bus contains the sender’s identification, the communication unit can correctly associate the incoming data with its memory location. This independent addressing scheme permits the communication unit to store a new input value (during $\Phi_1$) using the external address bus, while the execution unit (during $\Phi_2$) recovers data from a different location, by making use of the addressing module’s counter. The communication and execution units are presented in the subsequent sections.

5.4.2. Communication Unit

The communication unit deals with all data movements between a specific processing element and the rest of the network. Therefore, all data directed to a processing element is firstly processed by this specific unit and then passed to the associated execution unit, accordingly.

The basic functions performed by the communication unit are:

- to initialise the required processing element’s parameters. This involves loading initial random weights, target output patterns (for supervised learning) and some architecture-specific parameters, such as identification of previous and next layers (when bidirectional flow of data is present) and the total number of inputs for the forward and backward paths.

- to read and store input data into the appropriate memory block (state or error inputs). This implies analysing both the control bus, to verify through the ready signal if there is a valid input on the data bus, and the address bus to confirm if the data value is addressed to the processing element. The communication unit should also indicate to the execution unit that a state or error calculation must be initialised.
• to write, when commanded by the central controller, the output value evaluated by the execution unit, which can be either a state or an error value. It should also set the *ready* signal to point out to other processing elements (as well as to the central controller) that a legal value has been written onto the data bus.

• to generate the memory access control and address signals, for data movement to/from the memory unit.

All these functions are executed by interfacing with the four basic off-chip broadcast busses defined in section 5.3.1: the two data busses - input and output - which can be externally implemented as a single bus; the address bus driven by the central controller; and the control bus containing the required command lines. The control bus can be divided into general and algorithm dependent command lines. Table 5.2 illustrates the functionality of the control lines by presenting the control bus of the Back Propagation prototype.

<table>
<thead>
<tr>
<th>Control Bus Command Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General Commands</strong></td>
</tr>
<tr>
<td>rst</td>
</tr>
<tr>
<td>ld</td>
</tr>
<tr>
<td>(\Phi_1, \Phi_2)</td>
</tr>
<tr>
<td>cs_rdy</td>
</tr>
<tr>
<td><strong>Algorithm Dependent Commands</strong></td>
</tr>
<tr>
<td>fw_bwb</td>
</tr>
<tr>
<td>l_rb</td>
</tr>
</tbody>
</table>

Table 5.2: Command lines in the control bus of a Back Propagation chip.

The internal configuration of the communication unit, as well as its external connectivity, is shown in Figure 5.8. It is basically composed of two modules: *datapath* and *control* modules.

**Datapath Module**

This module basically accomplishes the external address bus analysis, verifying the relevance of its contents during input and output phases. The functionality of the datapath module is illustrated in Figure 5.8. It is essentially composed of comparators and their associated registers to perform two basic tasks: determine when an output value
(state or error) can be broadcasted into the data bus; and verify when the data bus value must be stored into the processing element’s internal memory.

To determine when the PE is allowed to write into the off-chip data bus, the datapath module encloses: a register named my_address, which contains the processing element’s private number identification; and an associated comparator that detects if the value in the address bus matches the PE’s identification.

The my_address register must be correctly specified to avoid any data bus conflict between processing elements. The register, however, can not be initialised during the loading phase (as happens with other parameters), since the processing element is unable to detect, without its identification, if the data bus contains a pertinent value to be read. Furthermore, to conform with cost limitations in chip production, the application-specific neuro-chips should be made identical. This constraint also prevents the my_address register from being initialised by the silicon compiler before the actual chip fabrication, since it would force the production of a different mask for each PE in the system. Therefore, to overcome these problems, this register is initialised through hardwire process.

The use of the hardwire method implies that the register is initialised via external definition supplied at the integrated pads. However, due to pad number limitations, the hardwire initialisation must be multiplexed among the integrated PEs. This is achieved
by fixing at fabrication stage the my_address's lowest significant bits for each PE in the neuro-chip. External pads are then used only for the register's most significant bits, which are then fixed for each different chip in the network.

Regarding the second basic task of the communication unit's datapath - the analysis of the address bus to investigate if the data bus must be read - two approaches can be followed, according to the network's topology. Firstly, in fully connected systems, the communication unit only needs to analyse the address bus layer field, since all PEs of the previous layer (or next layer if back propagation of errors is involved) emit relevant data. This scheme considerably reduces the necessary circuitry for the input address investigation.

Secondly, in sparsely connected systems, two implementation methods can be executed, in this case according to the degree of connectivity. For a high degree of connectivity, with just few connections missing, the fully connected implementation should be applied. In this case, the communication unit circuitry is the same, with the addition of some zero entries in the weight memory to form a fully connected memory block. For a lower degree of connectivity, however, the communication unit's datapath should encompass more comparators, one for each valid input address. The decision of which approach should be taken depends on a trade-off between the silicon area occupied by communication's comparators and the increase in the size of the weight memory.

Additionally to the comparators/registers required to accomplish data movements between PEs, the communication unit's datapath also encompasses some tristate components to control the access to/from external busses (input/output and address busses) as well as internal memory busses.

**Control Module**

This module generates all command lines to effect data transmission between the specific processing element and other elements in the system. It is implemented through a programmable logic array (PLA), which considerably simplifies the silicon compiler task of generating the processing element's control. The PLA receives inputs from the associated datapath module, the external control bus, and the execution unit, and generates the control commands to perform the appropriate data movements. Its basic command lines include: access control signals for the memory blocks (i.e. chip select, read/write, etc.); signals to control the memory addressing module; and finally signals to command the execution unit, indicating which operation must be performed.
The control module operations can be classified into two groups: *initialisation* and *execution*. Initialisation commands are related to the loading mechanism, which initialises the weight values and all additional parameters concerning the processing element's complete configuration. Initial weight values are directly stored into the proper memory block, while configuration parameters are written into their specific registers. This complete set of initial values is provided by the central controller that indicates, on the address bus, the processing element being initialised, and on the data bus the value to be stored. The weight memory initialisation is assisted by the memory unit's addressing module that determines, in sequence, the specific memory entry to be initialised. Therefore, the initial weight values must also be supplied in sequence by the central controller.

Execution commands, on the other hand, are associated with the transferring of data during the execution of the neural model recall and learning phases. In this operational phase, the control module activates the appropriate command lines to select the pertinent memory blocks (forward or backward) and uses the external address bus to determine the specific address to store the input values. This is accomplished assuming the associated datapath module has detected a valid data in the external data bus.

In addition to controlling the memory access, the control module also signals to the execution unit the start of a data acquisition phase, triggering the appropriate mathematical operation such as state or error calculation. This allows both communication and execution units to perform their respective operations in parallel. The execution unit ensures that a signal is sent in return to indicate that the evaluation of an output value has been completed. With this feedback and the PE's selection mechanism implemented by the central controller, the communication unit's control module can decide when an output value can be broadcasted.

The control module also encompasses the selection mechanism to indicate to the central controller when the processing element is ready to write data into the data bus. As described in section 5.3.1, the central controller determines, through internal counters, the owner of the data bus. It then waits for the *ready* signal from the selected processing element to verify if an output value has been broadcasted. If however, the specified processing element has not concluded the appropriate calculation, the *ready* signal should remain inactive in order to express to the central controller, as well as to other processing elements, that no valid data has been written into the data bus. Since the *ready* signal is used to read and write data from/to the data bus, it is implemented as a bidirectional signal. The *ready* signal generation module (see Figure 5.8) is composed of two unidirectional tristates, activated in a mutually exclusive mode. They are usually configured to
the input mode, allowing the processing element to constantly verify when a valid data has been presented in the data bus. This selection module only converts to output mode when the execution unit has completed the evaluation of an output and the communication unit’s datapath has identified the processing element identification in the external address bus.

5.4.3. Execution Unit

The execution unit is responsible for the computation of the neural functions. It executes the three basic neural functions of the *generic neuron* model, in accordance with the high level description provided by the application designer. The physical implementation can be realised by one or two operative parts, depending on the necessary compromise between performance and silicon area. However, despite the fact that the final structure can be configured to match user’s demands, the execution unit has a fixed framework, presented in Figure 5.9.

![Figure 5.9: Execution Unit’s internal structure.](image-url)

The basic structure of the execution unit consists of: a *control module*, that receives control commands from the communication unit and directs the whole execution of the neural functions; and a *datapath module* that performs the mathematical operations. The design of these two modules follows two basic requirements: be easily configurable, to
accommodate the required neural functionality; and accomplish the mathematical operations in fixed-point, 2's complement representation, since it is general and easily implemented in hardware\textsuperscript{51}. The decision for fixed-point arithmetics has been supported by many simulation studies\textsuperscript{12,28,87} which have shown the applicability of this method. The two basic execution unit's modules are described below.

**Datapath Module**

This module contains all the necessary blocks to perform the mathematical operations associated with the neural algorithm being implemented. It is entirely controlled by its associated control module and, although it can be configured to match the desired application, some common features can be identified (see Figure 5.9). The datapath module is composed of two independent busses, that perform data transfer between all components of the datapath, and some basic modules that can be assembled according to the required capabilities. These basic modules are:

- **Arithmetic & Logic Unit (ALU)** - This module performs the basic mathematical operations associated with the neural network model. Two essential operations are required in all neural algorithms: additions and subtractions. Even the most common operation among neural algorithms - multiplication - can be easily implemented by addition and subtraction steps\textsuperscript{33}. Hence, the simplest ALU design possible should be able to support at least these two basic operations in 2's complement.

- **Accumulator/Shifter** - This module is required to store the ALU operation results, in order to be used in further calculations. Additionally, it provides means to perform the 1-bit step multiplication of the Booth's algorithm for two's complement numbers\textsuperscript{33}. If a multiplication of N by N bits is required, a 2N shift-register, with add operation for the most significant bits, should be available. This is the main purpose of this 2N module, which comprises two separately addressable units: the most significant unit is used for accumulation, and the least significant unit is also utilised as the multiplier operand of the multiplication. After N clock cycles, the multiplication result is present in the composition of these 2 modules.

- **State & Error Registers** - These two registers are used to store the processing element's output values: the state output ($s_j$) and, if the learning algorithm is being implemented, the error output ($e_j$). These values are stored into internal registers because besides being broadcast to the external data bus, they are also utilised during the learning process to update the processing element's synaptic weights.
• **Look-up Table ROM** - This module is fundamental for the state output calculation. As defined in section 5.2, the activation function $f$, which calculates the processing element's output state, depends on two simpler functions: the propagation rule and the threshold function. The propagation rule is easily calculated by ALU operations. Nevertheless, owing to the non-linear characteristic of the threshold function, it is simpler and faster to have it evaluated through a look-up table. Consequently, in the *generic neuron* architecture the threshold function is digitised, according to the user specified size, and then stored in a ROM memory. The implementation through look-up table allows the threshold function calculation to be fulfilled in one single cycle, instead of many additional cycles if sequences of mathematical operations have to be performed. In addition, as it will be discussed in Chapter 7, the area occupied by the ROM memory is very small, occupying just 6% of the total PE’s area in the Back Propagation prototype. In the case where the neural network algorithm requests simpler threshold functions, such as the hard limiter shown in Figure 1.3, the look-up table can be substituted by a simple comparator, considerably reducing the necessary silicon area, without compromising the processing element's performance.

• **Multiplicand Register** - A special register is required to store the multiplicand operand of a multiplication operation. Although the usage of a dedicated register to store this particular operand increases the necessary silicon area, this apparent waste is compensated by the substantial simplification of the multiplication control.

• **Auxiliary registers** - Finally, a provision for some auxiliary registers is also necessary in case intermediate values need to be stored during the neural function processing. The total number of auxiliary registers is algorithm dependent, being configured by the silicon compiler.

As can be seen by the basic building blocks described above, the execution unit’s data-path has been designed to afford ease configurability to comply with the neural application’s requisites. Special attention has been devoted to provide all the necessary facilities to perform the 1-bit multiplication step in one clock cycle, since this particular operation is one of the most common in neural network models.

**Control Module**

This module produces all the necessary control commands to direct the execution unit’s operations. These operations are related to the execution of the three basic neural functions formulated in the *generic neuron* model: the activation function, error calculation
and weight updating function. The control module commands all data transfers between the datapath's blocks, as well as generates control lines to supervise the ALU operation and to perform the multiplication step. It also provides control to access the memory unit but, in order to avoid conflict with the communication unit's memory access, this module ought to activate the memory lines on the complementary phase of the communication unit's control module.

The control module, as shown in Figure 5.9, also relies on a programmable logic array (PLA) module to generate the appropriate command lines. The use of a PLA to effectuate the execution control greatly facilitates the matching of the processing element design to the neural application since, apart from some minor changes in the datapath, the PLA is the basic module which requires modification.

This control module also contains a special counter dedicated to the multiplication operation. It is used to count the necessary number of multiplication steps, in order to indicate to the PLA control when a multiplication operation has been completed. The counter's design, in accordance with the Booth's algorithm for multiplication, is totally dependent on the required number of bits for the multiplier register, which is defined by the state and weight precisions supplied by the application designer.

The approach to the execution unit's design followed the idea of providing maximum flexibility for the implementation of a neural network model. This aim has been achieved by using easily generated PLA blocks and by providing simple building blocks that can be assembled to form a datapath module tuned to the desired application.

5.5. Silicon Compiler Parameters

The basic aim of this architectural design has been to provide a general framework that could be easily configured to implement a wide range of neural applications. This configurable architecture can be tuned according to the user's definition, and then automatically generated by the silicon compiler under development. The design had to achieve maximum degree of flexibility without either compromising the neural network's performance or making the usage of a behavioural silicon compiler impossible, by providing an extremely open architecture with too many variables to be configured. Therefore, the generic neuron target architecture provides some tunable parameters but restricts the basic framework to enable automatic production with a silicon compiler. A preliminary analysis of the necessary parameters that should be translated by the silicon compiler is summarised below:
• **the three basic generic neuron neural function** \((f_1, f_2 \text{ and } f_3)\) - define the processing element’s functionality, specifying the recall procedure and, if requested by the user, the learning phase operations. These parameters also determine if there is bidirectional flow of data and if error values must be internally stored.

• **network topology** - describes the entire network connectivity, specifying for each processing element the number of inputs and, consequently, the necessary memory size to be integrated.

• **full or sparse connectivity** - defines the internal circuitry for validating input data and accessing internal memory. In case of dense connectivity with just a few connections missing, the weight memory should be implemented as fully connected with some additional zero synaptic weights. On the other hand, if the network is extremely sparse, the communication unit should be implemented with more comparators to legitimate the input data.

• **weights and states precision** - specifies the required precision for the internal data representation. This definition also determines the number of bits coded for the off-chip data bus.

• **number of bits for address bus** - indicates the coding of the address bus, specifying the number of bits for the layer and neuron-inside-the-layer fields. This parameter is totally dependent on the network configuration, where the application designer defines the total number of layers and the correspondent number of neurons.

• **single or multi-bus configuration** - defines, in conformity with the user’s request, if the final implementation is composed of one single bus or multiple broadcast busses. The user chooses the final configuration according to the application and the trade-off between improved performance and central controller complexity.

• **threshold function look-up table** - determines which threshold function should be implemented and with which accuracy i.e., how many points must be used to digitise the mathematical function. This directly affects the look-up table size that must be generated.

Some extra parameters, such as learning rate and total number of state and error inputs, can also be generated by the silicon compiler if it does not cause differences in the processing elements implementation. Otherwise, these parameters have to be assigned by the central controller, during the initialisation phase.
5.6. Summary

This chapter has presented the basic concepts of the *generic neuron* model and its associated architecture. The architecture has been designed to serve as the basic target framework for a silicon compiler, which will automatically generate application-specific integrated circuits from the high level neural specification language nC.

The basic requirements established for the *generic neuron* target architecture were to afford maximum *flexibility* to implement a wide range of neural network models, without jeopardising the system's *performance* or increasing the necessary *silicon area*. To trade-off all these basic requisites, the architectural framework has been based on a modular, self-contained processing element, that communicates with a central controller and other processing elements in the system through common broadcast busses. The bus interconnection strategy provides the basic requisites for general purpose neurocomputer's design: flexibility, expandability and scalability. This approach has been confirmed by some currently available products\(^44,136\) as one of the most efficient for neural network architectures.

The processing element's basic structure, on the other hand, has been designed to optimise the necessary silicon area without compromising the final performance. Hence, the processing element is a self-contained module, comprising three basic units: memory unit, communication unit and execution unit. The provision of mechanisms to execute the learning process has been regarded as a fundamental requirement in the PE's design, since some applications require a network size that makes the weight values determination through simulation extremely cumbersome\(^44,104\).

A prototype VLSI implementation of a neuro-chip dedicated to the Back Propagation algorithm has been designed to analyse the suitability of this proposed architectural framework. The VLSI implementation issues, as well as the cell library designed, are presented in Chapter 7. However, before describing the VLSI design, the simulation results carried out to verify the adequacy of the architectural framework to neural networks are presented in the next chapter.
Chapter 6

Architecture Simulation

This chapter examines the architectural simulations carried out to evaluate the generic neuron architecture. It first presents a short introduction to the software simulator developed, describing the user interface provided and the data structure used. It then presents the simulation results obtained from the execution of the Back Propagation neural model.

6.1. Simulator Framework

The architecture simulations carried out in this research had as the primary goal the investigation of the generic neuron suitability for the development of cost-effective application-specific neural chips. Because the realisation of a complete and accurate simulator would be very time consuming and therefore beyond the scope of this thesis, the simulator has been designed based on a simple model. The model has been specifically designed for the analysis of how the adopted hardware specifications affect the performance and correctness of neural networks' execution.

According to the generic neuron architectural definition presented in the previous chapter, the basic contrasts between the hardware execution and the normal simulation of neural models are: the use of fixed-point arithmetic; and the threshold function evaluation via the look-up table approach. Consequently, the basic issue in the simulation studies has been to investigate the influence of these two hardware related parameters on the execution of neural network models. Additionally, some tests have been performed to evaluate the performance of the generic neuron bus strategy when a large number of PEs are interconnected.

The influence of the fixed-point arithmetic on the computation of neural algorithms is regarded as an important issue, since most of the simulations and theoretical predictions for neural network algorithms assume floating-point execution. On the other hand, as discussed in Chapter 5, although the evaluation of the threshold function through look-up table has been the common tendency in the hardware implementation of neural networks\textsuperscript{44, 132, 136}, its implementation involves various approximations, whose effects have to be examined. These approximations are related to the specification of the table.
size and the precision of the table’s entry value (number of fractional bits). If, for instance, the propagation rule’s result (the table’s entry value) is coded in 16 bits, a table of $2^{16}$ values would be required, which is an unacceptable size for the look-up table in terms of silicon area. To achieve a reasonable size for the look-up table, a subset of the total number of bits i.e., a frame, should be used. The choice of the frame’s total number of bits, as well as its precision, is expected to greatly influence the neural computation. In order to explore these variables, the software simulator has been developed.

To allow the computation of diverse neural network algorithms and applications, the software simulator has been designed as a flexible system, composed of adjustable modules that can be configured in accordance with the user specifications. Therefore, an extensive configuration module has been produced, which allows an easy definition of the network’s topology and functionality, as well as of important parameters such as data precision and threshold function digitisation. In addition, mechanisms to display the network’s performance and evolution was also regarded as an important issue. Hence, an appropriate user interface has been incorporated into the simulator. The simulator’s final structure, written in the C programming language, comprises 6 modules:

- **Configuration Module (confnet)** - provides a framework, analogous to the generic neuron model, where the user can define all parameters related to the processing element’s configuration and functionality. This module also contains the complete specification of the network topology, determining the size of the network and its connectivity.

- **Central Controller Module (main)** - simulates the central controller commands, determining the overall network operation. It performs the initialisation of all relevant parameters (including pattern loading) and controls the processing elements communication, collecting statistics about the execution.

- **Processing Element Module (control_pe)** - executes all the processing elements’ neural functions, i.e state calculation, error calculation and weight updating. Moreover, it also carries out, through the command of the central controller module, the initialisation of the PEs’ internal parameters.

- **User Interface Module (interface)** - encompasses all the necessary procedures to allow the user to interact with the software simulator. The module provides information about the computation evolution, as well as means to select a specific pattern to be recalled among the set previously learned.
• **Structures Definition Module (defsim)** - incorporates the required declarations of the processing element’s data structures. These data structures follow the internal structure of the *generic neuron* model.

• **Network Construction Module (environment)** - contains all the accessory functions to completely configure the neural network application. Among these functions are: the memory allocation for the whole system and the random initialisation for the weight values.

By observing the above simulator structure, it can be seen that it mimics the basic central controller/processing element organisation, implementing the specified communication protocol existent between these two entities. To avoid an unnecessary increase in the simulator’s complexity, the software has been designed following the single data bus communication strategy, instead of the multiple busses approach. This approach reduces the development time without compromising the defined simulation purposes.

The software simulator is clock driven, based on a global clock. At each clock cycle, the simulator scans the whole system and performs all the necessary operations in the processing elements composing the application. These operations include internal register transfers, read/write data from/into the data bus, and ALU operations. After the execution of each operation, the system timer is advanced by the number of cycles associated with that particular operation. The system’s clock period (in mega hertz) is specified by the user, who defines it according to the available integration technology.

The simulation analyses have been accomplished with the computation of the Back Propagation neural network algorithm. This model has been chosen for its particular requirements in terms of data precision and data communication, since it depends upon bidirectional flow of data and on small increments for the weight update.

Before presenting the simulation results, the following sections briefly describe the essential modules of the software simulator: the configuration module, that provides the user with a friendly interface to specify the tunable neural network parameters; and the basic data structures, that provides the processing element’s declaration according to the *generic neuron* model.
6.1.1. User Configuration Parameters

The software simulator provides an extensive configuration module that permits to perform many experimentations with a wide range of neural network models, comprising different topological organisations and arithmetic operations. The main configurable parameters supplied to the user are:

- **network topology** - contains the specification of the total number of layers and the number of neurons available in each layer. These values are used to configure the neural network system by allocating the required memory space for each processing element present in the network.

- **patterns specification** - where the total number of patterns and their respective values are defined. The simulator accesses the pattern specification structure and learns each pattern in sequence.

- **resolution definition** - contains the specification of the total number of bits for the state and weight values, as well as the determination of how many of these bits are used for the fractional part.

- **threshold table definition** - determines the threshold function requirements in terms of number of bits used for the look-up table entry value (which defines the table size), and the table’s input value precision (number of fractional bits).

- **PE’s functionality** - where the definition of the processing element’s functionality is determined. It follows the generic neuron model proposed in section 5.2, comprising the three basic neural functions (f1, f2 and f3) with their respective parameters. These three functions should be coded for each new neural algorithm to be simulated. Although the simulations carried out in this thesis are based on the Back Propagation algorithm, two other models have been coded (Hopfield and Boltzmann Machine) in order to verify the generality of the configuration module.

Other important parameters include the system’s clock frequency (specified in mega hertz) and the saturation values for the threshold look-up table.

6.1.2. Data Structure Organisation

To execute the neural model functions specified in the configuration module, the software simulator makes use of a global, hierarchical data structure, that defines the entire neural network algorithm. The data structure defined in the software simulator
complies with the generic neuron model proposed in Chapter 5, containing the same basic data components discussed in section 5.2.

An overview of the data structure organisation is shown below. The simulation system is based on an array of pe data structures (see below), which are composed of a neuron unit and a weight unit (structure types nu and wu respectively), as in the generic neuron model. Correspondingly, these units encompass their appropriate data values, such as the state and error registers in the neuron unit (state and error), and the forward and backward blocks (*wstable and *wetable) in the weight unit's case. The size of the pe array, as well as the number of forward and backward blocks allocated to each processor, is application dependent, being specified by the user. Consequently, the simulator uses the data structures definitions, collectively with the topology specification provided by the user at the configuration module, to arrange the necessary number of PEs in the system and to allocate the correct amount of internal memory for each PE in the network.

typedef struct tag_ws /* Forward Block Definition */
{  
   int weight;
   int state;
}ws;

typedef struct tag_we /* Backward Block Definition */
{  
   int weight;  
   int error;
}we;

typedef struct tag_wu /* Weight Unit Definition */
{  
   int error;
   ws *wstable;
   int state;
   we *wetable;
}wu;

typedef struct tag_nu /* Neuron Unit Definition */
{  
   int state;
   int error;
   long sw;
   long ew;
}nu;

typedef struct tag_pe /* Processing Element Structure Definition */
{  
   wu w_unit;
   nu n_unit;
}pe;

Having briefly described the overall simulation framework, the following sections present the simulation results acquired from the execution of different tests with the Back Propagation model.
6.2. Back Propagation Simulation Results

This section discusses the most important results from the simulation studies of the Back Propagation neural algorithm. The simulation analyses have focused on three basic aspects of the hardware implementation: the investigation of the influence of the threshold function digitisation; the effectiveness of the fixed-point arithmetic approach; and the performance of the broadcast bus interconnection scheme. To explore these hardware constraints and to be able to compare the results, the algorithm dependent parameters have been kept fixed throughout the simulation experiments. These parameters are:

- learning rate ($\eta$) 0.3
- tolerance ($\epsilon$) 0.1
- clock cycle 10 MHz

The following sub-sections present the investigation results for each of these three hardware implementation aspects.

6.2.1. Threshold Function Investigations

Following the look-up table approach for implementing the neural model’s threshold function, various simulation experiments, with different table sizes, have been performed.

The look-up table is generated according to the threshold function’s configuration parameters provided at the simulator’s configuration module. These parameters correspond to the specification of the table’s entry value format, which is defined to correctly extract the required subset of bits from the propagation rule’s result. There are two basic parameters: the total number of bits used for the table’s entry value; and the required number of fractionary bits. The former parameter determines the number of entries in the look-up table, while the latter specifies the step in the threshold function digitising as well as the table’s maximum and minimum values. All experimentations have been accomplished with 4 bits in the integer part, since it affords a reasonable output interval that should be sufficient for most neural network applications. Six different sizes for the threshold table have been used: 64, 128, 256, 512, 1024 and 2048. Figure 6.1 displays the simulation results accomplished with these table sizes, using a 64-neuron network distributed in three layers with 24, 16, and 24 neurons in the input, hidden and output layers, respectively. The network is presented with 6 patterns, with the results collected after all patterns have been successfully learned.
As can be seen from Figure 6.1, the use of the look-up table approach does not influence the execution of the learning process, still allowing the network to correctly learn all patterns. Additionally, Figure 6.1 indicates that there is a minimum size for the look-up table beyond which no further improvement in the learning speed is achieved. In fact, when the look-up table approach is compared to the normal execution of the threshold function, the former usually accomplishes the learning procedure in less number of cycles. This is due to the smaller step size of the real implementation of the threshold function, which forces the network to reach the convergence in a larger number of cycles.

In this particular configuration, using the same set of patterns, the network reaches the convergence only after 115 cycles when full execution of the threshold function is implemented externally.

It must be pointed out however that, although the six different table sizes tested in the simulations have successfully performed the learning algorithm (four cases in less than 100 iterations), this parameter is application-dependent. This implies that for a different neural application, with different values for $\eta$ and $\varepsilon$, a look-up table comprising 256 values might not be enough to correctly learn the required patterns in a reasonable number of cycles. Therefore, a hardware emulation is always recommended prior to the final neuro-chip fabrication, in order to establish the minimum table size necessary for the particular application.
6.2.2. Fixed-Point Arithmetic Analysis

Some simulation studies have also been performed to verify the network’s behaviour under the fixed-point arithmetic operation. Since most neural network simulations assume floating point arithmetic, some experiments have been executed in order to investigate the effects of the fixed-point operation on the learning process.

All experiments assumed an equal fixed-point configuration for states, errors and weights, encoded with 4 bits for the integer part and a variable number of bits for the fractional part. The experimentations have been accomplished with the same parameters used in the look-up table investigation ($\eta=0.3$ and $\epsilon=0.1$), as well as the same set of patterns. Tests with 8, 10, 12, 14 and 16 bits for the fractionary part have been performed and simulation results demonstrated that a minimum of 12 bits (for the fractionary part) were required to correctly learn the set of input patterns.

The simulation results have also indicated that after successfully executing the neural algorithm with a certain precision, no further improvement is achieved in the learning procedure with the usage of higher precision. In fact, the simulations have shown that, when more bits were being used for the data precision, more cycles were necessary to correctly learn the 6 patterns. Similarly to the threshold function implementation, this is due to the smaller steps taken towards convergence, which are originated from the higher precision being used.

These results are validated by the hardware emulations and theoretical analysis developed by Alippi and Nigri$^{12}$. They have shown, through exhaustive emulations using the Back Propagation algorithm, that fixed-point notation actually reduces the required number of learning cycles when an appropriate precision is being used. They made use of the character recognition application, and demonstrated that at least 16 bits (12 bits fractionary part) are required for data precision in most learning applications. They have also proved from theoretical analyses that, for the Back Propagation algorithm using the sigmoid function, the number of bits required for the fractionary part is determined by the following equation:

$$n_{\text{sigmoid}} \geq \log_2 \left( \frac{1}{\eta \epsilon^2 (1-\epsilon)} \right)$$  \hspace{1cm} (6.1)

where $n_{\text{sigmoid}}$ is the number of bit required for the fractionary part when the sigmoid function is being used; $\eta$ is the learning rate; and $\epsilon$ is the tolerance value. Table 6.1 illustrates some values obtained from equation 6.1, for different values of $\eta$ and $\epsilon$. As
shown in this table, for the above configuration of $\eta$ and $\epsilon$, a minimum of 9 bits is required to correctly implement the learning algorithm. However, due to additional hardware simplifications which were not taken into account in the theoretical studies, the final precision is expected to require at least two bits more. Therefore, a minimum of 11 bits is achieved for the fractionary part when using the above configuration.

<table>
<thead>
<tr>
<th>Tolerance ($\epsilon$)</th>
<th>Learning Rate ($\eta$)</th>
<th>0.1</th>
<th>0.2</th>
<th>0.3</th>
<th>0.4</th>
<th>0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>11 bits</td>
<td>10 bits</td>
<td>9 bits</td>
<td>9 bits</td>
<td>8 bits</td>
<td>7 bits</td>
</tr>
<tr>
<td>0.05</td>
<td>13 bits</td>
<td>11 bits</td>
<td>11 bits</td>
<td>11 bits</td>
<td>10 bits</td>
<td>9 bits</td>
</tr>
<tr>
<td>0.02</td>
<td>15 bits</td>
<td>14 bits</td>
<td>14 bits</td>
<td>13 bits</td>
<td>13 bits</td>
<td>12 bits</td>
</tr>
<tr>
<td>0.01</td>
<td>17 bits</td>
<td>16 bits</td>
<td>16 bits</td>
<td>15 bits</td>
<td>15 bits</td>
<td>14 bits</td>
</tr>
<tr>
<td>0.001</td>
<td>24 bits</td>
<td>23 bits</td>
<td>22 bits</td>
<td>22 bits</td>
<td>21 bits</td>
<td>20 bits</td>
</tr>
</tbody>
</table>

Table 6.1: Minimum number of bits for the fractionary part, obtained from equation 6.1.

These simulations and theoretical studies have been accomplished assuming that the execution of the learning procedure is implemented on chip. For applications that require only the recall phase, fewer bits are necessary to correctly perform the neural algorithm. In this case a higher level of integration is attained, affording a larger number of PEs per integrated circuit.

6.2.3. Network Performance Examination

This section presents the performance simulations executed to assess the efficiency of the generic neuron interconnection strategy. The main purpose of the network performance simulations has been to analyse the network's behaviour according to the number of processing elements connected into the broadcast bus. To perform this analysis, many different network sizes have been tested, using the time to perform a complete cycle (forward and backward) to compare the different configurations. The cycle computation time includes the neural functions processing time, state and error values propagation through the network, and also the overhead to load the input and output patterns into all input and output PEs, respectively. The different network configurations were based on a three-layer Back Propagation network, with full connectivity between layers.

To evaluate the network performance, some additional assumptions have been made related to the final hardware implementation. These suppositions are summarised in Table 6.2.

With the above hardware dependent parameters, the transference of an input/output pattern from the central controller into each appropriate PE was assumed to be executed in
Hardware Implementation of the Processing Element

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>data precision</td>
<td>16 bits</td>
</tr>
<tr>
<td>look-up table size</td>
<td>256 values</td>
</tr>
<tr>
<td>system clock cycle</td>
<td>10 MHz</td>
</tr>
<tr>
<td>communication timing</td>
<td>a single cycle for each data transfer</td>
</tr>
<tr>
<td>ALU operations (sum/subtract)</td>
<td>a single cycle</td>
</tr>
</tbody>
</table>

Table 6.2: Parameters used for the network's performance examination.

one cycle. Additionally, the data communication of state/error values between PEs is also accomplished in one single cycle. The multiply operation is executed in 16 cycles, based on the 16-bit precision used for states and synaptic weights and the 1-cycle ALU operations. All the above suppositions conform with the implemented VLSI Back Propagation prototype, described in Chapter 7.

Figure 6.2 displays the simulation results performed with 6 different network sizes (64, 128, 256, 512, 1024 and 2048 neurons). Figure 6.2a presents the actual time taken to complete the computation of one forward/backward cycle. Figure 6.2b, on the other hand, exhibits, for the same set of network configurations, the evaluation of number of connections treated per second.

The above simulation results have shown that the usage of a broadcast bus with sequential access by the PEs does not create a major bottleneck in the network performance, granting minimum computational overhead when compared to the neural model execution. Indeed, since the generic neuron interconnection scheme does not require any communication protocol and allows broadcast communication in both directions of the data transmission, the main computational power is dominated by the actual execution of the neural functions, which, in this case, is performed in parallel. Consequently, as shown by Figure 6.2b, the number of connections per second increases with an expansion of the number of PEs in the neural network application. The computational power of the above examples is further increased if it is considered that the connections between the hidden and output layers are duplicated in the PEs of both layers. If these backward connections are taken into account, the number of connections per second is even higher.

Some simulations have also been carried out using the Sejnowski and Rosenberg’s NETtalk network configuration as a benchmark. The NETtalk network learns to transform written English text into a phonetic representation. It consists of three fully connected layers composed of 203, 60 and 26 units, respectively, which results in a total
a) Analysis of the time to compute a cycle in various different network configurations.

b) Analysis of the total number of connection evaluated per second.

Figure 6.2: Results from the network performance examination.

of 13740 connections. With this configuration and using the same hardware parameters assumed before, the network performs one complete cycle in 0.79910 ms, which is equivalent to approximately 17.2 million connections per second.

It must be pointed out, however, that although the bus strategy does not present a bottleneck in terms of execution performance, it exhibits some limitations due to the propagation delay of data traversing the bus when a large number of processing elements is
necessary. In this case, the multi-bus configuration should be favoured, in order to afford more PEs in the neural application without having to decrease the global system clock cycle.

6.3. Summary

This chapter has examined the simulation results obtained from the execution of the Back Propagation algorithm in the *generic neuron* architecture. The simulation studies were instrumental in the investigation of the hardware parameters' effects on the neural network execution. Experiments have been performed with different configurations for the look-up table and data precision, which are the two main hardware-related simplifications when compared to neural model simulations. Additionally, the software simulator has also provided a means to analyse the effects of the broadcast interconnection strategy on the network performance.

These simulation studies have demonstrated that the communication scheme of the *generic neuron* architecture is not the major limitation in the neural network’s performance. The main bottleneck in the application’s execution is the number of cycles necessary to perform a multiplication operation. Simulations have also indicated that the use of look-up table and fixed-point arithmetic is viable, providing successful neural network implementations with a reasonable table's size and data precision.

The minimum number of bits for the data resolution and the minimum size to implement the look-up table are, however, application dependent parameters. They need to be tuned to the desired application in order to achieve the best trade-off between speed and silicon area. Consequently, a hardware emulator should be available to the user in a general silicon compiler environment, providing a useful integrated tool to experiment with different parameter values until the best configuration is achieved.

Some complementary studies of the hardware execution effects have therefore been carried out as part of the silicon compiler environment\textsuperscript{87}. A hardware emulator has been built and integrated into the *Pygmalion* programming environment to provide the user with a general tool to analyse, prior to the silicon compilation process, the best *generic neuron* configuration suited to the application.

The following chapter describes the Back Propagation neural chip prototype designed to verify the viability concerning the VLSI implementation of the *generic neuron* architecture.
Chapter 7

VLSI Architectural Implementation

Following the simulation results presented in Chapter 6, this chapter investigates a VLSI implementation of the proposed generic neuron architecture. It starts by presenting the design approach, taking into consideration the available VLSI design tools for the project. Next, it reports on the design of a modular cell library to implement specialised operative parts of the neural chip. Finally, it examines design considerations and results of the Back Propagation layout prototype, which was designed using CMOS 2µm technology.

7.1. VLSI Design Approach

The role of a good VLSI design is to reduce its complexity and to assure the designer a final product consistent with its architectural and functional specifications. Before embarking on the physical implementation, one must consider a structured approach that trades the expected end results with the available tools and design methodologies for a particular chip. Overall, the relevant attributes of a VLSI design can be summarised as follows:

- silicon area
- performance: speed and power
- implementation time
- testability

The first two attributes are particularly influenced by the silicon processing technology, while the last two attributes depend mostly on the design approach, design environment and on the designer’s expertise.

A number of design methodologies have been investigated, with an estimation of their impact on the design attributes. For silicon area, the primary objective was to pack multiple processing elements onto the silicon die, in order to evaluate future implementations with denser technologies. To meet this requirement, the cell library has been designed seeking maximum optimisation of the layout in order to provide minimum area for the chip’s datapath. The performance constraints were also rigorously observed in the conception of these cells. The cell library (described in section 7.2) was
implemented in collaboration with Carlo de Oliveira and Marco Pacheco, whose theses also involved VLSI design\textsuperscript{93, 99}.

Given our limited experience with VLSI design and the timetable constraints of the thesis, we sought to combine the available tools to take advantage of the automating procedures and leave a good deal of handwork out of the design. This approach offered the best prospects for dealing with a large design in a short time. By the time the implementation was started, the VLSI design tools available were: the Berkeley tool set, in particular Magic\textsuperscript{110}, and the CAD system Solo 2000 from European Silicon Structures\textsuperscript{1}. Magic is an interactive editor for creating and modifying VLSI circuit layouts\textsuperscript{97}. As a symbolic editor (it has knowledge of design rules), Magic provides better design resources than a simple layout editor. The disadvantage of using Magic is that it does not incorporate a complete design, verification, and test system, to attend all phases of an integrated circuit design. For instance, Magic offers limited support for automatic routing (it only deals with local routing) and does not work with schematics. On the other hand, SOLO provides a complete and integrated tool set, with an unified data structure, that includes: a schematic graphics editor, module generators for parameterised RAM, ROM and PLA, automatic placement and (global) routing facilities, and standard cells libraries for 1.5\textmu m and 2.0\textmu m CMOS technologies\textsuperscript{2}.

The option for SOLO seemed obvious, except that the version of SOLO available in the Department did not include the layout graphics editor. Consequently, there were two alternatives for implementing the processor's datapath: to use Magic and integrate the layout into the SOLO environment; or to use the standard cells supplied by SOLO. From the viewpoint of simplicity and design automation, the alternative of using standard cells, combined with the schematic editor, was certainly the most attractive one. However, silicon area would have increased significantly.

To evaluate this alternative properly, a toy microprocessor was designed using standard cells all over the design. The microprocessor was proportionally sized in order to provide a more precise idea of the effect standard cells would produce in the VLSI designs. The toy microprocessor was a 16-bit RISC containing: 2 instructions (add\&write and sub\&write); a set of 8 (16-bit) registers; 512 words of RAM; and 64 words of ROM.

The results obtained from this chip were extremely discouraging. The device measured 7.9\times 5.2\textmm, with 42\% of its area occupied by standard cells. Considering that the complexity of our actual designs were at least three times greater than that of the toy microprocessor, the use of standard cells for the datapath was out of question. We
therefore opted for the alternative which would result in the smallest size of die and could provide adequate basis for assessing performance and chip area of implementations with modern technologies. The cells were then designed using Magic and later exported to the SOLO environment.

Before starting the layout, an extensive analysis of the Magic's technology file was carried out to determine how manageable would be to convert it from the available 2\mu m to the 1.5\mu m technology. The analysis showed that the intended modification would demand a laborious work of reprogramming and testing Magic, which was considered outside the scope of the thesis. For this reason the chip was implemented in 2\mu m CMOS.

In order to achieve a highly integrated chip layout and to shorten the layout development time, the automated design system has been applied in all design levels. The flowchart diagram in Figure 7.1 describes the design environment used for the chip implementation. The chip layout consisted of a six-stage design process:

1) **Cells Design:** This involved the manual layout of bit cells and the automatic generation of the \( n \)-bit slices for each type of cell in the cell library. The slices were designed with Magic and their layouts exported to the SOLO environment by means of a software tool that converts formats (Calma to SDA). In the SOLO environment, three representations were created for each of these slices: a *symbol* for manipulating the slices in the design of the cell-blocks; a *schematic*, describing the slices' logic behaviour for functional simulation; and an *abstract* representation with pin and cell boundary information for placement and routing.

2) **Cell-Blocks Design:** Cell-Blocks are functional units that combine the logic features of a number of slices. They implement registers, ALU and other functional elements, present in the datapath. Cell-Blocks were designed in three phases: firstly, they were laid out in Magic by abutment to check for DRC (design rules check) errors and for the need of blank-jumper cells between slices; secondly, schematics were produced in SOLO for netlisting and simulation purposes (see some examples in Appendix 3); finally, symbols were automatically created for later use in the chip integration.

3) **Datapath Design:** In this stage, cell-blocks were combined to implement datapath segments. Their design followed the same steps executed for the cell-blocks and also included the generation of a textual placement of the cells. This text records geometric characteristics related to cells abutment, rotation, and overlapping.
4) **Macroblocks Design:** Macroblocks are RAM, ROM and PLA modules created automatically through the SOLO module generators. The generation of RAM and ROM blocks and their representations is straightforward. For the PLAs, the process also involved truth-table generation (Eqntott) and Boolean minimisation (Espresso), before the final module generation.

5) **Random-logic Design:** These are parts of the circuit that were designed with standard cells, using automated design programmes for schematic editing and capturing.

6) **Chip Integration:** Producing the final chip layout involved many steps, all of them supported by the automated design tools. In the first step all components were integrated into a single chip schematic (Schematic Editor) for generating netlist and
Cells and blocks were placed using a semi-automated method for controlling terminals position and direction, in order to reduce the interconnection area. Routing was performed by SOLO's advance routing programme that supports automatic channel generation and control functions to specify routing objectives. In this process, layouts were checked for DRC errors and the design process repeated from the beginning. The final optimised and correct layout was then converted to Calma and sent for fabrication.

7.2. Datapath Cell Library

The cell library contains a total of 117 different cells, organised into three main classes: modular register building blocks, functional elements, and interface & switching elements. The design of these cells followed a well-defined design philosophy to attain a balance between the minimum silicon area and high performance requirements.

7.2.1. Design Philosophy

As the complexity of VLSI chips increases, the need for a common, shared cell library becomes essential. Sharing cell library helps to prevent duplicate development efforts and promotes exchange of ideas for new cells architectures. In addition, it contributes to the achievement of rapid prototyping and allows designers to explore alternative architecture designs by combining different cell structures with varying design constraints, such as area, speed, fan-out and power consumption.

Designing a cell library involves choosing a number of features and standards for the library. These include: cell selection; performance attributes; topological, electrical and temporal characteristics; drive and interface capability; and a standard format for documenting the cells (Appendix 2 presents some documentation examples). The chosen design philosophy emphasises modularity and flexibility while attempting to trade-off performance and silicon area. Therefore, the designed cell library conforms with a consistent set of design rules that provide a near-optimum process for synthesising different functions. In this process, cells concatenate linearly to form multiple-bit slices which are arranged in stacks to customise registers and functional units to the design purpose.

The first design decision for the library is the choice of the logic clock regime. From the three classical choices, dynamic, pseudo-static and static, the pseudo-static was considered the most suitable. Dynamic circuits are smaller but unstable; static circuits
are more reliable but yield complex realisations for functional implementations; the pseudo-static is a trade-off. In a pseudo-static clock regime, the logic state of a register is temporarily stored in the gate capacitance of its transistor. The continuity of the clock guarantees the restoring of the logic levels. Compared to the static logic, this scheme is disadvantageous, since the clock can not be stopped during chip tests for examining the machine state, or for reducing power consumption in stand-by operation. The main advantage of the pseudo-static logic is that it provides a simple and small design for a master-slave register, which consists of a pair of inverters separated by pass transistors (static logic requires twice as much)\textsuperscript{92}.

The second important design choice regards the functional coupling of cells, meaning how transistors are arranged to synthesise a specific function. The coupling used is the "ratioed" logic\textsuperscript{80,117}. With this logic, the coupling to the register's core can be made through pass transistors, thus reducing the number of physical control lines. In addition, some simple functions can be implemented by using NMOS switching logic, which is highly area efficient. Another consequence of this technique is the use of limited voltage-swing bus which is discussed later in this section.

The combination of pseudo-static and ratioed logics is the key to achieve an effective modularity. The pseudo-static output stage provides the robustness of a dual rail drive that eliminates electrical hazards in module interfacing. The ratioed logic enables a wired-or input interface that reduces the interdependence between modules and simplifies logic circuitry.

The cells in the library are designed to interconnect with each other by abutment or overlapping, with coincidence of terminals. This requires a topological and geometrical standardisation of control, data and supply lines, as well as a fixed pitch in one direction. The implications for inter-cell routing were studied to create guidelines, yielding a balanced relation of area economy, layout complexity and aspect ratio.

The cell frame is structured in an orthogonal grid of rails and tracks, with bits concatenating alongside tracks and registers stacking alongside rails (Figure 7.2a). Tracks carry control lines within standard geometric patterns for each cell family. This allows the division of any functional block into specialised fields of any bit count. The rails determine the standard pitch of the cell library so that cells can stack without the need of terminal routing. Studies with circuit sketches resulted in the establishment of nine rails in a 64 lambda cell pitch (Figure 7.2a): 5 data rails (A, B, X, Y and Z), 2 supply rails (GND and VDD), and two connection rails (C). The supply rails are large-
current metal strips for powering the cells along the datapath. The five data rails cover the intercommunication needs of a datapath segment, while the connection rails are reserved for internal circuit routing and contacts between layers.

The concatenation of bit cells is made by overlapping adjacent supply rails alongside the tracks. To match the corresponding supply polarity, the cell must be flipped around the rail axis (Figure 7.2b). This reduces the total block area and alleviates the overhead caused by the large supply rails. Moreover, this strategy reduces the chances of well-substrate's conflicts and their electrical problems.

Three additional standards complete the set of design rules: the transistor banding, the transistor size and the bus voltage-swing. Transistor banding organises the transistor locations alongside the connection rails, reducing routing and supply lengths. The transistor banding also contributed to the standardisation of transistor width at 4 lambdas. This small size for transistors leads to faster circuits by limiting the load on control drivers.

Figure 7.2: The Interconnection Structure of Cells.
The voltage swing of signals that move data between resources in the datapath, affects the bus delay in proportion to the bus capacitance\(^3\). Bus precharging is a technique which reduces this problem and it is most useful in designs where the bus is idle every other phase or cycle due to the organisation of the processor\(^5\). For designs such as the generic neuron, in which the processing element uses the bus on every clock phase, the limited voltage-swing bus\(^3\) is a more suitable approach. The use of this technique (granted by the NMOS pass transistors used as register access gates) required careful design and tests, and contributed to minimise the machine cycle time.

Next sections describe the basic features of the three main classes of cells: *modular register building blocks*, *functional elements* and *interface and switching elements*.

### 7.2.2. Modular Register Building Blocks

The main requirement in the storage cells design was to attain a flexible set of modules that could be assembled according to the desired functionality. To meet this requirement, a set of modular building blocks has been designed, which can be assembled to create registers with the necessary number of ports and reset/preset features.

The register building blocks were designed using the "ratioed" logic coupling technique for minimum area consumption\(^1\). With this approach, the register terminals can be accessed through pass-transistors, considerably reducing the necessary number of control lines. Actually, the use of "ratioed" logic halved the number of control lines, since only one line is required for each pass-transistor, instead of the two in the transmission gate. Moreover, "ratioed" logic grants shorter machine cycles due to its intrinsic limited voltage for the level "one", caused by the pass-transistors\(^3\). The fundamental limitation in the "ratioed" logic technique is the asymmetry in the rise and fall timings\(^8\).

The register building blocks can be classified into four categories: *basic register*, *bus read and write*, *preset and reset* and *register-to-register connections* cells.

**Register Cells** - The basic register design was centered on the *semistatic* (or pseudo-static) approach, where the charge of the dynamic memory element is refreshed every cycle\(^8\). The circuit shown in Figure 7.3 implements a bit storage with two inverters and two transmission gates. The transmission gates TG1 and TG2 control the transmission and restore phases, respectively; the inverters provide complementary contents. For this circuit, the input data in X is transmitted to the output terminal Y during Φ1, and it is restored back into the input stage during Φ2. With this scheme, it is possible to read and write a register in the same phase, by using a two-phase nonoverlapping clock.
Input data as well as reset and preset commands are fed into the X terminal via pass transistors, making use of one control line per input (Figure 7.3). Likewise, the register's output is read into the datapath busses (or into another register) through pass transistors. Care must be taken in order to prevent TG2 from being closed simultaneously with an input pass transistor, to avoid data conflict between an input data and the restored value.

This basic register has been developed using two different layout designs. The first layout provides access to the input/output terminals from both top and bottom cell boundaries. This feature permits direct data communication through the X and Y terminals, avoiding the unnecessary use of the common datapath busses. However, this cell is not transparent to bus Z, supplying terminals to busses A and B only. The second layout retains transparency to all three datapath busses A, B and Z but requires a larger area. Besides increasing the cell height, this configuration looses the access to X and Y terminals from one of the cell's boundary (X and Y terminals are available at the top, while their complements are provided at the bottom of the cell). Although this feature can be viewed as a limitation, it is actually quite useful in cases where the previous and current cell's output values are required simultaneously, or when a master-slave configuration is desired.

**Busses Read and Write Cells** - These cells comprise a single pass transistor which connects the X and Y terminals to one specific bus. This simplicity in the cell design has been achieved by selecting the "ratioed" logic coupling mechanism. These cells provide read and write facilities to all existing busses, namely A, B and Z. Any combination of these busses can be used, allowing the basic register to have up to 3 bus ports.

Figure 7.3: Schematic of the basic register.
**Preset and Reset Cells** - These are simple cells that provide the facility to force a specific value into the register. They are also implemented through pass transistors plugged into the register’s X terminal.

**Register-to-Register Connections** - To increase the register’s connectivity, two additional ports have been provided in the form of register-to-register connection cells. These cells contain one pass transistor that basically connects the X terminal of one register to the Y terminal of another register. The coupling can be made at the top and bottom boundaries of the register cell. Therefore, by coupling the appropriate building blocks, the basic register can communicate through up to 5 ports.

The above single-bit modular register cells can be arranged into one datapath width module or can be split in fields with isolated bus accesses and reset/preset configurations. Apart from having up to 5 ports and reset/preset attributes, the basic register can perform extra functions with the addition of other functional elements such as counters, comparators and shifters.

### 7.2.3. Functional Elements

Functional elements are special cells designed essentially to provide extra functionality to the basic register. They include comparators, counters, shifters, and a general-purpose arithmetic and logic unit (ALU). All these functional cells can also be organised into fields of any length or even be mixed, in order to provide the necessary functionality to the datapath.

**Comparators** - The cell library provides a set of four different comparators to conform with the required number of ports and functionality. Their design follows the same approach used in the basic register: some comparators offer transparency for the two basic datapath busses (A and B), while others also include the bus Z terminal, resulting in a higher cell. The library also includes dedicated comparators that test registers contents, either to zero or all one values.

Comparators in the cell library adopt a single design strategy. Their common structure is based on the traditional exclusive-or (XOR) design, implemented with AND/OR gates. Smaller circuits could have been achieved by using the transmission gate XOR logic\(^\text{134}\). However, switch level simulators such as Silos\(^2\), present some problems with this design, hence the decision not to use it.
**Counters** - The counter cells supplied in the cell library were designed to easily integrate with the basic register. The register’s output is used as the counter’s input; after evaluating the new value, the counter directs the result to the register’s input. Three different configurations are supplied: an UP counter, a DOWN counter and a combination of both - an UP/DOWN counter.

The method adopted for the counters’ design is based on an array of pass transistors, controlled by the register’s output. The counting is accomplished through a dual carry chain (the carry signal and its complement), similar to the one used in the ALU (see subsection Arithmetic & Logic Unit).

**Shifter** - The shifter functional element has also been specially designed to interface directly with the basic register. It is based on the barrel shifter approach and provides left and right shifting operations. Due to the register’s "ratioed" logic design, the basic shifter has been implemented using simple pass transistors to reduce the silicon area. To afford special facilities for arithmetic shifts as well as for shifting a concatenated register, specialised shift cells have also been designed.

**Arithmetic & Logic Unit** - For the ALU the important features are: the type of functions it can perform, the data width, and the operating speed. To support multiple VLSI projects, a general-purpose ALU has been designed, with a configurable width and an optimised carry chain. The structure of the ALU resembles that presented by Mead and Conway, i.e., it contains: three logic functional blocks for generating P (propagate), K (kill) and R (result) signals; and the carry chain circuit. These functional blocks provide inputs for the carry chain and for the ALU output stage. Twelve control wires (four for each function generator) are used to select among the various logic functions of three variables (A, B and Carry).

The carry chain is typically the most speed-critical component in an ALU, since it must take the carry information across the width of the ALU. In the standard implementation of the Manchester carry chain, a single carry chain (usually, taking the inverted carry signal) runs through a series of pass transistors with a precharge circuit at each stage. A pair of cascading inverters is inserted at regular intervals to overcome the quadratic delay originated by the series of pass transistors. A variation of the Manchester original design has been developed by adopting a dual carry chain with asynchronous charge. The circuit for one stage of the modified Manchester carry chain is shown in Figure 7.4. The circuit activates the asynchronous charge by "generating" a carry. Thus, when P=K=0 (corresponding to the inputs A=B=1), carry lines are charged properly.
The dashed lines represent how the carry chain is buffered: at restoring stages, an inverter is introduced in each chain and their outputs are interchanged between the two chains.

Figure 7.4: Carry chain circuit for the ALU.

The techniques used in this design serve a twofold purpose: they speed up the ALU operation and provide a design compatible with the proposed strategy for machine timing. The gain in performance is obtained by the use of a single inverter rather than two at each restoring stage, and by adopting asynchronous operation for the carry chain (instead of using a whole phase for precharging and the following phase for calculations, the ALU combines both operations concurrently in a full "shorter" cycle). This design uses more area than the single precharged chain, but it is justified since it compensates the lack of look-ahead circuitry.

An additional characteristic of the ALU allows writing the result of the ALU operation without consuming resources of the datapath. The ALU contains an internal path that takes its outputs back to its inputs location, so that it can be loaded into one of the feeding registers.
7.2.4. Interface and Switching Elements

Interface elements are cells designed to improve flexibility and compatibility among operative blocks. They include: bus drivers, data-alignment converters, and padding cells. Switching elements are simple arrays of pass transistors for interconnecting bus lines.

**Bus Drivers** - These are more complex interface cells, designed to match the electrical characteristics between internal and external busses. The bus drivers amplify the limited-driving signals of the register drivers and simultaneously convert them into full CMOS range. A bus driver has two stages: the first senses the ratioed levels; the second restores the signal level and supplies a fan-out sufficient to drive three standard cells. These drivers are designed in two types—unidirectional and bidirectional—and incorporate controls lines to put their outputs into high impedance.

**Data-alignment Converters** - These cells provide extra flexibility in the datapath design, by exchanging the topological assignment of busses for specialised connection. These converters transpose bus lines across bit boundaries, diverting the bus route. They are used as barrel shifters, that shift bus lines in steps of $n$ bits, aligning bytes packed in multi-field words.

**Padding Cells** - Padding cells are interface units, used to separate modules with incompatible borders. In the cell library, modules were designed to optimise the area used for each function. However, modules are conservative in respect to the margin used to avoid border conflicts. The avoidance of all possible conflicts would excessively penalise the silicon area. Therefore, a variety of padding cells have been designed. These cells interface contiguous modules both horizontally and vertically, interrupting the appropriate lines.

**Switching Elements** - These elements provide a simple gating to connect or isolate two busses under demand. No signal amplification is performed and the switching elements add their own load to the busses burden. These elements can substitute for full tri-state buffers, in cases where minimum area is decisive. Switching elements are also applied to provide pseudo-static storage in combinational circuits. For instance, they are positioned at the input of combinational PLAs to incorporate dynamic retention on the input lines. These elements do not comply with the topological standards of the other cells but, instead, they assume the characteristics of the external elements they connect.
7.3. Chip Organisation

Making use of the above cell library and the VLSI design tools available, a prototype chip, specialised in the Back Propagation neural functions, has been implemented. The chip implementation followed the *generic neuron*’s basic hardware specifications described in Chapter 5, with the particularities of the chosen neural algorithm configuring the final architecture.

In terms of the VLSI design, the chip organisation can be separated into two units: the *operative* part, that embodies the necessary blocks to perform the neural functions; and the *control* part, that comprises the control blocks to command the neural model execution in the operative part.

7.3.1. Operative Part Organisation

The processing element’s operative part determines the internal hardware implementation and the topological structure of the modules defined at the functional level. There are several criteria available to the machine designer for the organisation of the operative part\(^{15}\). Here, the design of the operative part is based on the principle of information locality to increase performance, comprising on-chip memory and registers with multiple accesses.

The *generic neuron* operative part is organised in three units: a three-segment datapath, the look-up table for the threshold function, and the local memory for storing input data and synaptic weights (Figure 7.5).

The local memory, which covers almost 36% of the chip area, is divided into four blocks that stores respectively: input states (\(S_F\)) and their associated weight values (\(W_F\)); as well as backward errors (\(E_B\)) with their related synaptic weights (\(W_B\)). All memory blocks share the same address bus, while the data busses are separated into two groups, one for the input values (states and errors) and another for the synaptic weights (forward and backward). The look-up table is implemented by a ROM memory with 128 words, occupying merely 6% of the silicon area. Both memories, RAM and ROM, were automatically generated using the SOLO module generators.

The datapath, on the other hand, was produced using the cell library designed with Magic, occupying approximately 9% of the chip area. The whole datapath is 16-bits wide, and has a structure which consists of three perfect rectangles, with busses in metal-1 and control lines in metal-2. This rectangular datapath is assembled from a large
number of small cells on a multiple bus structure. The bus system is interrupted by sets of tri-state drivers which allow data transfer or independent processing by the operative sub-parts. These tri-state drivers are made sufficiently large to drive the capacitive loads of internal and external busses lines, as well as to convert the signal of the limited voltage-swing busses to a level accepted outside, by the standard cells and memory blocks.

Functionally, the datapath is composed of three operative sub-parts (Figure 7.5):

- Neural Execution Processing
- Communication Processing
- Memory Address Processing
The neural execution processing unit includes all essential modules defined in Chapter 5, such as: the ALU; accumulator/shifter (ACC); multiplicand register (MPX); state and error registers (\(s_j\) and \(e_j\)); and two auxiliary registers (\(Aux1\) and \(Aux2\)) used to store intermediate results of the Back Propagation execution. The communication processing unit contains the three necessary comparators that verify the processing element's own address (\(my_add\)) and the layer addresses from which the processing element should receive inputs (\(prev_ly\) and \(next_ly\)). Finally, the memory addressing processing unit encompasses a counter to sequentially access the memory blocks, as well as its associated comparators to determine when the whole memory block has been searched, either in the forward (\(comp_fw\)) or backward (\(comp_bw\)) calculation.

7.3.2. Control Part Organisation

The control part of a sequential machine commands the operative part by activating its control lines at the right time according to the system timing. The control part of the generic neuron prototype takes less than 4% of the chip area and it is implemented with PLAs. Each PLA synthesises a "nondeterministic" finite state machine\(^{31}\), in the sense that a machine can be, simultaneously, in more than one state. The PLA implementation of nondeterministic finite automata\(^{114}\) provides a considerable reduction of silicon area in contrast with conventional implementations\(^{91}\).

The organisation of the control part, shown in Figure 7.6, is extremely simple and includes two control units:

- Neural Execution Control Unit
- Communication Control Unit

The neural execution unit controls the operation of all functional components of the neural execution processing in the operative part. It includes the necessary control commands to execute the three functions of the generic neuron model: state calculation, error calculation and weight updating. The neural execution control unit is composed of a single PLA and an auxiliary counter (\(mult_ptr\)) to perform the sequencing of the 16-cycle multiplication operation. The PLA generates 29 different outputs to direct memory accesses, ALU operation and register data transfer. As input, the PLA takes 7 input signals (apart from the 28 feedback machine states) from four different sources: the off-chip control bus, the memory address processing unit, the communication control unit, and the multiplier counter.
The *communication control unit* regulates the data transfer from/to the off-chip data bus to/from the internal memory blocks. It comprises a single *PLA* that provides 8 outputs and receives 9 inputs from either the external control bus, the neural execution’s PLA, or from the communication processing operative sub-part.

The system timing is based on two-phase nonoverlapping clock signals ($\Phi_1$ and $\Phi_2$), generated directly by an external oscillator (Figure 7.7). The two phases are assigned to the control units in a way that avoids conflict in the memory access. The control commands issued by the communication unit’s PLA are released during $\Phi_1$, while the execution unit’s signals are liberated during $\Phi_2$.

This control scheme permits both control units to work in parallel and share internal resources, without the necessity to implement complex self-synchronised control to prevent resource contention, which would require far more silicon area. Instead, with the approach taken, silicon area is saved, allowing the integration of multiple processing elements in the same chip.
7.4. Implementation Considerations and Results

The major trend in the design effort has been directed to two implementation considerations: the optimised use of the silicon area to improve parallelism of the integrated processing elements; and a short implementation time, compatible with the project schedule. The composition of the VLSI design tools used, allowed us to complete the chip design and to partially meet these consideration points. A number of difficulties have been experienced which were caused by not having access to a complete and integrated development system. It took longer than expected to produce the final layout and some planned tasks could not be executed. For instance, a consistent functional specification has been produced for all cells and blocks in the layout design, expecting to be used later to run a functional test. However, the lack of functional models for nMOS and pMOS transistors in the available version of the SOLO system, added to the limited time, prevented such test from being performed. Additionally, the design tools available provided insufficient support for layout measurements and therefore delivered poor statistics about performance and transistor count. Nevertheless, the automated place and route system played an important role for floor planning and area reduction.

The final floor plan organisation of the **generic neuron** neuro-chip was influenced by the area limitation of 100mm² imposed by the academic multi-project service for chip fabrication. It also took into consideration pin access and terminal proximity for size and routing optimisation. Figure 7.8 shows the schematic floor plan used for the Back Propagation prototype. It comprises two complete processing elements that are symmetrically disposed at the top and bottom of the die.

The three-part datapath is placed between the look-up table **ROM** memory on the left, and the four-block data memory (**RAM**) on the right. The control parts of both processing elements are aligned at the center of the chip, spanning control signals to the top and bottom PEs. The PLAs correspond to the neural execution control unit (**NU_PLA_i**) and the communication control unit (**CU_PLA_i**) of the processing element **i**. This row of PLAs also comprises the multiplier counters (**m_i**) that indicate when a complete multiplication has been effected. The areas between the PLA row and the datapaths contain standard cells that implement random-logic parts, such as drives for the registers and the necessary memory control signals generation. The relative placement of the blocks attempted to minimise the distance between connecting terminals, thus reducing routing lengths and propagation delays.
The prototype chip was implemented using 2μm CMOS process technology with double-metal interconnection layers. The features of this generic neuron prototype are summarised in Table 7.1. The chip contains about 60K transistors (the precise number of transistors could not be evaluated), 433 standard cells, and 179 customised cells, in a 7.5mm×10.1mm die area. The total number of 68 pins includes 48 I/O data pins, the others representing control and supply/clock pads. The layout photograph of the generic neuron prototype chip is shown in Figure 7.9.

In the experimental evaluations for operation speed, the programme SPICE was used to estimate the performance of possible critical paths in the processing element’s circuit. One possible critical path is the long carry chain of the arithmetic logic. Its total delay corresponds to the delay of 4 cascaded inverters plus the delay generated by the series of 12 pass transistors. Another suspect path is associated with the register transfer operation, which involves delays of the internal busses, the tri-state drivers and busses outside the datapath.

We observed the ALU operation for 16-bit data and the register transfer operation in the worst case conditions. The simulation performed with SPICE showed that arithmetic operations (add and sub) can be carried out with a 10MHz clock frequency. For the
<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Processors</td>
<td>2</td>
</tr>
<tr>
<td>Data Length</td>
<td>16-b</td>
</tr>
<tr>
<td>Arithmetic Unit</td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td>16-b</td>
</tr>
<tr>
<td>Data RAM</td>
<td>256×16-b (per processor)</td>
</tr>
<tr>
<td>Look-up table ROM</td>
<td>128×16-b (per processor)</td>
</tr>
<tr>
<td>Control PLAs</td>
<td>2 (per processor)</td>
</tr>
<tr>
<td>Machine Cycle</td>
<td>100ns (estimated)</td>
</tr>
<tr>
<td>Package</td>
<td>68 pin CLCC</td>
</tr>
<tr>
<td>Number of standard gate cells</td>
<td>433</td>
</tr>
<tr>
<td>Number of full-custom cells</td>
<td>179</td>
</tr>
<tr>
<td>Device technology</td>
<td>2μm CMOS</td>
</tr>
<tr>
<td>Die Size</td>
<td>7.5×10.1mm (76mm²)</td>
</tr>
</tbody>
</table>

Table 7.1: Features of the *Generic Neuron* Back Propagation Prototype.

register transfer operation, a pair of registers with full access to two busses were assembled together and tested. The busses were loaded with 0.1pf to simulate the complete bus load. The test revealed that the pseudo-static registers can operate at frequencies over 20MHz, not establishing therefore a critical path.

### 7.5. Summary

This chapter has presented the VLSI design approach taken to implement the Back Propagation prototype of the *generic neuron* architecture. It has described the design philosophy of the cell library produced, giving the main features of the basic cell blocks.

With this full custom cell library, a prototype chip was designed based on a 16-bit processing element with 256 words of data memory and a look-up table comprising 128 values. The final chip has been produced using CMOS 2μm technology and it is expected to run at a minimum 10 MHz clock frequency.

Due to unpredictable problems emerged from the non availability of a complete VLSI design tool, in addition to the restricted time available for the layout design, the final prototype is limited to two processing elements. However, with a more optimised placement and smaller CMOS fabrication technologies, a much higher number of processing elements will be able to be packed into a single integrated circuit. A prediction of the maximum number of processing elements that can be packed into a single chip, for different processing technologies and different data memory configurations, is provided in the next chapter.
Figure 7.9: Back Propagation Prototype Layout.
Chapter 8

Assessment

This chapter presents an assessment of the work accomplished in this thesis. It covers the four basic investigation aspects - the nC language, the generic neuron model and corresponding VLSI architecture, the simulator, and the VLSI chip implemented - necessary to achieve the thesis main goal: to define a target architectural framework for the automatic generation of application-specific neural chips.

8.1. Targets Review

As stated in Chapter 1, the neural computing research area requires two specialised tools for executing artificial neural models: a flexible software tool that permits experimentation with different aspects of neural networks, providing a framework to execute existent as well as new algorithms; and a massively parallel application-specific neurocomputer, which properly explores the intrinsic parallelism of neural networks, granting the necessary high performance for the neural application execution. Various software simulators as well as ASIC chips for neural networks have been designed, with many commercial products already available. However, these products have been designed independently, with no integration of the software and hardware tools. With this incompatibility between these two tools, a neural network application tested in the software environment has to be fully designed from scratch if a high performance VLSI neuro-chip is required for the computation of the particular application. Therefore, a neural network programming environment that integrates these two tools, with the capacity of automatically generating ASIC chips from a high level specification of the neural network application, is demanded. With this complete programming environment a neural network application can be described and tested using the software tool and, after all optimal parameters have been found, an ASIC can be automatically produced without requiring the user to have any knowledge of VLSI design.

The main goal of this thesis was hence to define a design framework that would allow the automatic generation of application-specific integrated circuits from a high-level description of a neural network application. The central point to achieve this aim is the design of a general target VLSI architecture that encompasses the main features of neural network
execution. In addition, a high level specification language is also required to describe the neural application to be generated, and to dissociate the user from the details of the VLSI design. The link between the high level language and the ASIC chips is a neural network silicon compiler, whose implementation is the subject of another PhD thesis in the Department.

Therefore, the research developed in this thesis has focused on the definition of the VLSI target architecture, named the \textit{generic neuron}, and on the specification of the neural network description language, called \textit{nC}. To verify the suitability of the proposed architecture for the neural network execution, and its viability in terms of VLSI design, a simulator and a VLSI prototype have been developed. The following sections provide an evaluation of these components according to the established goals.

8.2. \textit{nC} Neural Network Specification Language

Following the basic proposal of this research, two essential requirements were imposed on the high level neural network specification language \textit{nC}: to easily and concisely describe a wide range of neural networks; and to allow the translation of a trained, or un-trained neural network application into dedicated integrated circuits. To attain these requirements, some design goals were established:

- \textit{Neural Network Independence} - the language should be flexible to properly specify a vast number of neural network algorithms and applications, in both learning and recall modes.

- \textit{Target Machine Independence} - it should permit a neural network specification to be efficiently mapped onto a range of sequential and parallel conventional computers, as well as neurocomputers.

- \textit{Application/Algorithm Expertise Distinction} - it should enable the development of generic applications and generic algorithms, i.e., a single application should be executed by many different algorithms, and a parameterised algorithm should be configurable for many application.

- \textit{Application to ASICs translation} - the language should also be suitable for the automatic generation of VLSI chips based on the \textit{generic neuron} target architecture. Although the language is not expected to provide hardware description, it should encompass the necessary neural network information to allow a smooth compilation process into an adequate, intermediate, hardware-dedicated language.
To achieve these basic requisites, the \textit{nC} neural network specification language was designed based on the \textit{C} syntax and relying on a hierarchical data structure, named \texttt{system}, that incorporates all relevant neural network informations. The \texttt{system} data structure encloses into one hierarchical organisation the network's connectivity, data (states, errors and weights), functionality and control. By changing some parameters in the data structure, various different parameterised neural algorithms (application-independent algorithms) can be generated, which are then finally configured by the user's application definition.

The above strategy has proved to be quite efficient for the central language of a general software programming environment dedicated to neural network execution. This is demonstrated by the success of the \textit{Pygmalion} programming environment and the variety of neural models provided in the environment's algorithm library, such as Back Propagation, Hopfield, Self-Organising Map, and Boltzmann Machine.

The main weakness of the \textit{nC} language concerning the simulation of neural network models is its inability to deal with shared weights, which is a common feature in image processing applications. The shared weights facility is very advantageous for saving the memory space of conventional machines, since a group of neurons makes use of the same subset of synaptic weights. However, this apparent drawback for the software simulation tool is in fact very beneficial for the parallel hardware execution, where the weight values are likely to be individually implemented to reduce traffic between processing elements. Indeed, the idea of each entity of the \texttt{system} hierarchical structure containing all the pertinent data (including all weights associated with its inputs), is entirely in conformity with the \textit{generic neuron} model, greatly facilitating the silicon compilation process.

Another useful feature for the silicon compilation process is the \textit{nC rule} concept. The \texttt{rule} data structure embodies the required information about functionality and controllability for each entity in the \texttt{system} hierarchical structure. The \texttt{rules} associated with the neuron level are actually the ones defined in the \textit{generic neuron} model (state calculation, error calculation and weight updating), considerably simplifying the hardware synthesis operation.

The connectivity information is also adequately extracted from the \texttt{system} hierarchy, where all connections are explicitly declared. An extra desirable feature, which is not provided in the \textit{nC} language, is an explicit declaration of fully connectivity between layers. The usage of this parameter greatly improves the hardware implementation, being therefore quite beneficial to have it clearly specified in the language. The provision of
this extra parameter is now being incorporated into an extension of the \textit{nC} language, which is being implemented for the silicon compiler's design.

Some additional hardware-related parameters should also be appended to the language, in order to afford tunable hardware variables such as data precision, threshold function precision and look-up table size. These parameters are also being added to the language as a result of the silicon compiler project under development in the department.

An \textit{nC} language compiler is now being generated to translate a neural network application into a hardware-like, intermediate level representation (ICR - Intermediate Code Representation). This representation will then be compiled into the VHDL IEEE standard hardware description language, completing the high level part of the synthesis procedure, which will produce ASIC chips based on the \textit{generic neuron} target architecture.

\section*{8.3. \textit{Generic Neuron} Model & VLSI Architecture}

This section assesses the suitability of the \textit{generic neuron} architecture as the target framework for a neural network silicon compiler. The criteria used for the architecture's evaluation consider the four main design goals presented in Chapter 1:

- \textit{flexibility} - both in terms of the processing element's functionality, where it should be able to perform various different learning algorithms, as well as in terms of the interconnection strategy, that should be able to represent the wide spectrum of network topologies used by neural models.

- \textit{high performance & parallelism} - the architecture should conform with the intrinsic parallelism of artificial neural networks, granting the necessary high performance to execute the neural applications.

- \textit{minimum silicon area} - due to the large number of processing elements required by a neural application, the silicon area necessary to implement a PE should be optimised in order to grant a large number of PEs per chip.

- \textit{design scalability} - the architectural framework should adopt a modular and regular design which, in view of the improvements in VLSI processing technology, could grant more PEs per integrated circuit without affecting the number of pins.

Using the above design goals, the following subsections evaluate the two components of the \textit{generic neuron} architecture: the interconnection scheme and the processing element's design.
8.3.1. Interconnection Scheme

To achieve the above objectives, the \textit{generic neuron} architecture has adopted the broadcast bus strategy as the network data communication medium. This choice provides the required \textit{flexibility} to implement a vast range of different neural models, since the bus interconnection scheme can virtually realise any complex topology. Additionally, due to the regularity and simplicity of the network topology, this strategy also contributes to minimise the required silicon area as well as to achieve a \textit{scalable} design, which is an essential property in massively parallel VLSI architectures. Indeed, the number of PEs produced in one chip can be unconditionally increased, with no effect on the pin count of the integrated circuit.

Another important benefit of the interconnection broadcast mode is its ability to transfer a processing element’s output to all its destinations in one single cycle. This intrinsic feature is quite useful for neural network applications, enhancing the resultant network’s performance. Because of these fundamental features provided by the broadcast bus, some recent hardware implementations of neural networks have also opted for this specific communication strategy\textsuperscript{44,136}.

In the \textit{generic neuron} architecture, the broadcast mode is extended even further, being also applied to the backward data distribution that occurs in algorithms such as Back Propagation. In such algorithms, the error values of the PEs in the output layer must be transmitted to the connected PEs of the hidden layer, in order to allow them to compute their own error values. The error value must be multiplied by the connection weight, likewise the state calculation computation on the forward phase. In current neural hardware designs\textsuperscript{60}, a separate \textit{weight}×\textit{error} value is transmitted for each existent connection of a particular output PE, thus reducing substantially the performance of the learning procedure. The \textit{generic neuron} architecture avoids this problem by treating backward connections similarly to the forward connections, storing in each PE of the hidden layer, the backward synaptic weights as well. This approach increases the demand on the internal memory required to store data but the increase in silicon area (which occurs only for the hidden layer PEs) is compensated by the considerable gain in performance during the learning process execution.

An inconvenience of the \textit{generic neuron} architecture is its awkward way of dealing with neural algorithms based on growing and pruning of the network\textsuperscript{113}, where the total number of PEs is not established a priori. These algorithms are basically used to determine the best network configuration (in terms of number of PEs) for the particular appli-
cation. Although their implementation is not impossible in the generic neuron architecture, the network should be implemented with the maximum number of PEs expected, resulting in a waste of silicon area. In this case, the learning procedure should be halted every time the network’s size is changed, in order to allow the central controller to reconfigure the appropriate PEs, either enabling or disabling them accordingly. Regarding the applicability of the generic neuron architecture however, this is not actually a major drawback since the silicon compiler will only produce ASIC chips based on previously optimised neural networks with respect to their sizes, in order to provide the user with the best network configuration for the defined application.

The major limiting factor of the generic neuron interconnection strategy, though, is the delay existent between an output sent from a PE at one end and its reception at a PE on the other end of the bus. This propagation delay imposes a limitation on the total number of chips, consequently restricting the maximum number of PEs that can be connected to the bus segment without forcing an increase on the global clock cycle. In the generic neuron’s case, the total delay is basically determined by the inherent propagation delay of the bus segment, since no arbitration or protocol is required to perform the data communication.

The maximum number of chips allowed per bus segment can be estimated using the results obtained from the Back Propagation prototype produced. Considering the fanout of the pads used for the output bus and the input capacitance of the input bus pads utilised, a total of 46 integrated circuits can be connected to the broadcast bus. Assuming that at least 10 PEs can be packed into a chip with current fabrication technologies (see Table 8.1), a minimum of 500 processors can communicate through the same broadcast bus. If this number is not sufficient for the target application, external drivers will have to be included in the broadcast bus. An alternative approach is to use multiple busses, dividing the network according to the number of layers and then assigning each layer to a different bus segment. This method increases the complexity of the central controller, but it allows the implementation of the neural network application without increasing the global clock cycle. Indeed, as stated in Chapter 5, the multi-bus strategy actually increases the final performance by using pipelining of inputs when multiple patterns can be treated simultaneously. The total number of PEs permitted per bus segment, however, depends on the number of PEs integrated into one chip, which varies according to the algorithm being used and to the fabrication process technology.
8.3.2. Processing Element Design

Following the evaluation of the *generic neuron*’s interconnection strategy, this subsection assesses the efficiency of the processing element’s design for the computation of neural network applications.

The design of the processing element’s internal framework followed the same general requirements specified earlier in this chapter as fundamental to the *generic neuron* architecture: *flexibility*, *high performance* and *minimum silicon area*. Some additional PE-specific requisites were also considered for the final design, such as *modularity* to reduce the interconnection cost on and off chip, and *accuracy* to correctly represent the neural network data information.

To attain maximum flexibility for the PE’s internal structure, a thorough analysis of the main neural network algorithms was effected, investigating their similarities and singularities. This study resulted in the specification of the *generic neuron* model, which incorporates into one single unit all the essential features extracted from the examined neural algorithms. This model is composed of configuration parameters that can be modified to allow the definition of a wide range of different neural network algorithms. This generality has been indicated by Table 5.1 where various algorithms, with distinct characteristics, have been described by modifying the *generic neuron* model’s internal parameters.

Based on this generic model of the artificial neuron, the internal structure of the processing element was then defined, following the main requisites established for the architectural design. The first decision related to the PE’s design concerns its implementation technology. Due to the generality requirement imposed on the PE’s internal structure, digital design was favoured over analogue technology. As discussed in Chapter 2, analogue design is not capable of providing the desired flexibility to execute various different models, including full computation of learning algorithms. Moreover, digital implementation also supports the accuracy and noise immunity requested by many learning algorithms, which reinforced the elimination of analogue design. Therefore, digital technology has been utilised throughout the PE’s internal design.

A second design methodology was related to the PE’s internal organisation. In order to produce a modular design, the PE has been organised as a self-contained entity, incorporating three fundamental modules: the *execution unit*, the *communication unit* and the *memory unit* for storing the appropriate data to perform the neural functions. This approach, in addition to reducing the interconnection cost, also increases the system’s final performance since off-chip data communication is restricted to the transmission of the
relevant PE's output values. This idea has been stretched to also integrate the backward synaptic weights present in error back propagation algorithms (as discussed in the previous section), improving even further the network's performance in comparison with existent hardware implementations.

The modularity attribute has also been applied to the PE's internal units design, facilitating the automatic generation of the final ASIC chip. In this case, each internal module is composed of a control unit, based on PLAs, and an execution unit, based on a modular datapath. Therefore, the final PE's design is accomplished by generating the adequate PLA code to control the computation, and by assembling the correct modules to build the appropriate datapath. By using this approach, the silicon compilation process can rely on basic full custom blocks that are assembled according to the requirements imposed by the application, greatly simplifying the synthesis process.

To improve the system's performance, the control part of the communication and execution units have been conceived to allow independent computation of their correspondent tasks. This property has been achieved by using different clock phases (from the two-phase non-overlapping clock scheme) to control each unit. Consequently, both units can share internal resources, such as the internal memory, without having to provide extra circuitry to prevent resource contention.

The performance attribute has also been considered in the design of the internal units' datapath module. To afford the multiple bus configuration, the communication unit has been designed to interface with external entities (either the central controller or other PEs) through two independent unidirectional data busses, in opposition to one bidirectional data bus. In this case, when multiple busses are demanded by the neural application, the communication unit's data busses are connected to two different external busses, allowing an increase in performance by using pipeline process. On the other hand, to increase the performance of the execution unit's datapath module, the design has also focused on trying to reduce the number of cycles to perform a multiplication, which is the basic arithmetic operation present in neural network algorithms. Different implementations of the multiplication algorithm, with different degrees of parallelism, have been examined, verifying their effect on the silicon area. The result of this investigation showed that the best trade-off between performance and silicon area (making use of the cell library implemented) was the Booth's two's complement add and shift algorithm. However, the final decision can be left to the silicon compiler's user, who can decide, according to the application's requirements, which is the best compromise between the two competing requisites.
A compromise between area and performance has also been considered for the threshold function computation. In order to afford fast computation and, at the same time, not compromise the necessary silicon area, the threshold function has been accomplished via look-up table, implemented with ROM memory. Since ROM memory occupies less than 6% of PE's core area (see Chapter 7), the final silicon area is not jeopardised by this strategy, allowing reasonable table sizes to be implemented on-chip. Additionally, the threshold function execution can be performed in a single cycle, the necessary time to have access to the ROM memory. The final table size, as well as its precision, is specified by the user, in accordance with application demands.

One of the basic weaknesses in the PE's implementation structure is related to the input address analysis that verifies the relevance of the data presented on the external bus. The analysis mechanism is quite suitable for applications based on fully connected networks (or networks with few connections missing), requiring only a single comparison to verify the layer field of the external address bus. Nevertheless, when the network is sparsely connected, the communication unit requires a sequential search in an internal table to decide if the data should be read into the local memory, which ends up degrading the system's performance.

However, the main limitation of the PE's architecture is in terms of the capacity to integrate the required RAM memory for storing weights and input values. This limitation is the price that a self-contained processing element has to pay in favour of considerably improving the final neurocomputer performance. The final capacity of the processing element is application dependent, varying according to the specified neural algorithm provided by the user.

8.4. Architecture Simulator

This section assesses the simulation studies performed to evaluate the generic neuron architecture. The evaluation has been carried out in line with the main simulation objective, i.e., to investigate the influence of the two basic hardware-related parameters: the use of fixed-point data representation, and the implementation of the threshold function via look-up table.

To verify the effects of the hardware-related parameters on the execution of neural network applications, a simple software simulator was built. The simulator final version is composed of six basic modules that can be adapted similarly to the generic neuron model. Although based on a simple model, the simulator is quite general, being able to
simulate various different neural algorithms by defining the appropriate code for the three basic neural functions, together with the specification of some architecture-specific parameters. The simulator also incorporates mechanisms to estimate the network’s performance, enabling the analysis of the system’s behaviour according to the number of processing elements connected to the broadcast bus.

The final software simulator proved to be very effective in providing information about the effects on the hardware execution of neural network algorithms. The simulation results were extracted from the Back Propagation execution, because of the strict requirements it imposes on data communication and on data precision. The results obtained have successfully demonstrated that the hardware implementation does not affect the neural application execution, provided that the relevant parameters are appropriately set. This means that if the network is able to correctly perform the computation with a certain data precision, no further improvement is achieved by increasing the data accuracy. These simulation results have been substantiated by an independent hardware emulation in conjunction with some theoretical analyses executed by Alippi and Nigri. In their investigation, they have established a relationship between the number of fractionary bits necessary to represent the synaptic weights and some neural network application parameters, such as learning rate and tolerance.

The results have also shown the importance of utilising a hardware emulation tool in order to determine, prior to the chip fabrication, the best network configuration for the defined neural application. This architectural emulator is very important in the development of a VLSI chip that incorporates the best compromise between performance and silicon area for a particular neural application. Therefore, with such a tool, the silicon compiler’s user is able to experiment with different hardware parameters, until the best configuration is attained.

The basic limitation of the implemented simulator is related to the moderate precision it provides for performance measurements. This limited accuracy results from the simplifications imposed by the timetable constraints of the thesis, and by the relative small size of the simulated examples, inflicted by memory and speed limitations of the computational system available at the time of the implementation. However, despite the simplified simulation model, the measurements have indicated the appropriateness of the generic neuron hardware approach towards neural network application execution.
8.5. VLSI Prototype Implementation

This last section investigates the viability of the generic neuron architecture in terms of VLSI layout implementation. In this context, the primary goal was to analyse the hardware complexity of the target architecture, in order to determine the actual degree of parallelism that can be achieved with practical implementations. To perform this investigation, a prototype chip of the generic neuron architecture was implemented, using the Back Propagation algorithm as the target neural model.

The prototype chip implementation has concentrated on achieving two fundamental requirements: high performance and minimum silicon area. Consequently, several design methodologies have been considered, with a careful analysis of their impact on the established design attributes. An examination of the available design tools have also been accomplished, culminating in the design of a full custom datapath cell library, specifically designed for the PE's operative part implementation. The datapath cell library's design focused on providing flexibility and modularity, while attempting to attain a practical balance between performance and silicon area.

The resulting cell library is general, comprising a number of functional modules that can be appropriately assembled to provide the required functionality. The implemented cell modules can be stacked by abutment, substantially reducing the silicon waste in routing signals between cells. The generality and modularity characteristics of the cell modules provide an adequate cell library for the silicon compilation process, where different application-specific datapath modules can be generated without compromising the silicon area.

The design of the PE's control part was guided by the flexibility requisite in order to facilitate the silicon compilation process. The control part is based on PLA modules that are automatically generated by the SOLO system. With this approach, the PE's control units can be easily redesigned to conform with the desired neural application by simply redefining the PLA equations, reducing the complexity of the control part synthesis.

However, due to the limited time and resources available for the VLSI implementation, some auxiliary circuitry had to be realised using standard cells provided in SOLO, which caused a significant impact on the final core area. Despite this loss in silicon area and the diverse problems experienced with the unavailability of an integrated VLSI tool, the results obtained from the prototype VLSI implementation are quite promising. With the outdated 2μm CMOS processing technology used, the produced Back Propagation prototype integrates two self-contained processing elements in a 7.5×10.1mm die size,
comprising 128 input connections (512 bytes of RAM) per PE and operating at a minimum 10MHz machine clock.

This first prototype implementation has been quite important in providing feedback about the constraints of the hardware implementation and in verifying the critical modules in terms of silicon area. Nevertheless, to better assess the hardware complexity of the target architecture, aspects such as the use of standard cells and the limitation to maximum 100 mm\(^2\) core size must be ignored. By excluding their effects on the final layout, it is possible to analyse the real parallelism obtainable with a full custom design, examining the packing capabilities of the \textit{generic neuron} architecture when more advanced integration technologies are available.

To accomplish this evaluation some assumptions have been made, in order to correctly estimate the packing density of the \textit{generic neuron} architecture. These are:

- The influence of the standard cells on the core area has been neutralised by reducing the prototype core area by 15\%. This figure has been extracted from the experiences acquired by the development of the toy microprocessor explained in Chapter 7, which was fully designed with standard cells;

- The core area limitation has been expanded to 150 mm\(^2\) to comply with current technology;

- The model for scaling the core area has been founded on the first-order MOS scaling theory\textsuperscript{134}, which establishes that, for a $1/\alpha$ dimension scaling factor, the circuit area scales by $1/\alpha^2$. Although this theory is somewhat simplistic for the CMOS technology, it provides a reasonable approximation of the total number of PEs that can be integrated into a single chip;

- The area utilised by the I/O pads has been considered constant for all core area examinations, since the pad area is usually determined by the minimum size to which a bond wire can be attached\textsuperscript{134}. In addition, owing to \textit{generic neuron} scalability property, the total number of pads remains unchanged by an increase on the total number of integrated processing elements per chip.

Table 8.1 summarises the results of the \textit{generic neuron} packing density investigation using five different CMOS fabrication technologies, ranging from 2\(\mu\)m to 0.8\(\mu\)m. For each processing technology, seven different PE configurations (in terms of the number of input connections provided) have been experimented, establishing for each memory
configuration, the total number of PEs that can be produced per chip. Figure 8.1 shows the packing density evolution curves for the same set of CMOS technologies presented in Table 8.1.

<table>
<thead>
<tr>
<th>Number of Processing Elements Integrated per Chip</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PE Internal Configuration</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>64 16-bit connections (256 bytes)</td>
</tr>
<tr>
<td>128 16-bit connections (512 bytes)</td>
</tr>
<tr>
<td>256 16-bit connections (1K bytes)</td>
</tr>
<tr>
<td>512 16-bit connections (2K bytes)</td>
</tr>
<tr>
<td>1024 16-bit connections (4K bytes)</td>
</tr>
<tr>
<td>2048 16-bit connections (8K bytes)</td>
</tr>
<tr>
<td>4096 16-bit connections (16K bytes)</td>
</tr>
</tbody>
</table>

Table 8.1: Packing Density Investigation of the *generic neuron* architecture.

As can be seen from these results, the *generic neuron* architecture yields good packing density in most of the fabrication technologies, except for the 2μm that affords multiple PEs per chip only with 1K or less number of connections. It must be noted however, that the above figures have been originated considering the RAM cells generated by the SOLO RAM generator. If state-of-the-art RAM cells had been utilised, the number of PEs per chip would have increased substantially. This can be confirmed by considering that, using the SOLO RAM generator, the local memory occupies from 31% (64 connec-
tions) to 92% (4096 connections) of the total PE core area. Additionally, these evaluations have been carried out using the Back Propagation prototype example, with learning procedure on-chip. For simpler models or recall phase only applications, the maximum number of integrated PEs can be increased even further, since the Back Propagation learning algorithm is quite demanding in terms of silicon area.

The main weakness of the VLSI implementation is related to its final computational performance. This deficiency is mainly due to the non-optimisation, in terms of performance, of the customised cells. Owing to the VLSI implementation constraints associated with the development time and the silicon area available, no fast look-ahead carry chain has been provided, limiting the final clock frequency to 10MHz.
Chapter 9

Conclusions & Future Work

This last chapter presents some general conclusions of the work accomplished in this thesis. It starts by summarising the results and main contributions of the research and then outlines the prospects for future work, describing the activities under way in the Esprit II Galatea project.

9.1. Summary

The primary aim of this research has been the investigation of a general architectural framework for the automatic generation of high performance, application-specific neural chips. The motivation for the design of such VLSI target architecture has been the ultimate goal of producing a complete and integrated neural programming environment that provides the user with two complementary tools: a software tool to learn as well as to develop new neural network algorithms and applications; and a hardware tool to automatically generate ASIC chips from a high level specification of the user’s neural application.

This thesis has concentrated on the development of the hardware tool, defining the generic neuron target VLSI architecture, which is the basic framework for generating ASIC neuro-chips, and the $nC$ neural network specification language, which is used to define the required application. The link between these two entities is provided by a neural network silicon compiler, that translates the neural network application specified in $nC$ into dedicated VLSI neural chips, based on the generic neuron architectural framework. The design of such silicon compiler is currently under development as another PhD research in the Department.

The $nC$ language has been specified as part of the Esprit II Pygmalion project which involved the development of a neural software environment for simulating neural network algorithms and applications. As the central module of the Pygmalion environment, the $nC$ network specification language has been designed following basic requisites, such as generality and machine independence. The language has demonstrated its ability to define and execute a large range of neural algorithms/applications, supporting many of the widely known models in the associated algorithm library. Its suitability as the input
A specification language for the automatic production of ASICs is now being indicated by the current progress on the translation of the nC language into an intermediate, hardware-related representation. This translation is the first step in the development of a high level synthesis tool for generating ASIC chips.

The main element of the neural network hardware tool, however, is the development of the target architecture. The design of the generic neuron target architecture followed essential requirements like flexibility, high performance, and minimum silicon area, in order to provide a general and optimised architectural framework for the silicon compilation process. Although some of these attributes are somewhat conflicting, the resultant generic neuron architecture has been designed to achieve a balanced compromise among them.

The regularity of the generic neuron bus interconnection scheme grants the necessary generality to implement the diversity of neural network topologies, and the capability to be easily expandable to connect the appropriate number of processing elements. Due to the utilisation of broadcast mode in both directions of data communication and the absence of a complex arbitration protocol for bus accessing, this interconnection strategy also provides the required simplicity of implementation, without undermining the final performance.

To properly define the processing element’s internal structure, an analysis of neural network algorithms has been carried out, extracting the basic features required for neural network execution. This study resulted in the specification of the generic neuron model, which has provided the basis for the processing element’s framework design. The final processing element skeleton is based on a self-contained modular entity, that is easily configurable to perform the required neural network application. Its regularity and wholeness characteristics assist the hardware synthesis process and substantially reduce the off-chip communication.

The generic neuron target architecture has demonstrated the possibility to produce a high performance, massively parallel, application-specific neurocomputer, without affecting the fundamental generality feature, necessary to allow the execution of various different neural applications. The high performance feature is achieved by producing dedicated VLSI chips for the execution of the desired neural application, granting compact processing elements related to the necessary silicon area. On the other hand, due to the modularity of the architecture’s design, the final processing element can be easily re-modeled to execute a different application, providing the required flexibility.
The effectiveness of the \textit{generic neuron} architecture results from efforts put into the architectural design simulation and VLSI implementation. The software simulations helped to investigate and confirm the small influence of the hardware implementation on the execution of neural models, assuming that the appropriate hardware parameters have been correctly set. The VLSI implementation of the Back Propagation prototype chip has proved the viability of the architectural framework in terms of layout design. Moreover, the VLSI prototype has provided a good estimative of the architecture's packing capabilities, indicating that a reasonable number of PEs can be integrated into a single chip when state-of-the-art fabrication processing is used.

This proposed design framework for the automatic generation of application-specific integrated circuits provides the user with a dedicated neurocomputer to efficiently and optimally execute the desired neural network application. In addition, with the integration of this hardware tool into a general neural network programming environment, the user can experiment with a variety of different algorithms and parameters, until the best neural network configuration is found for the particular application. This is an unquestionably important feature, since the neural computing area is still under constantly evolution, with no optimal algorithm suitable for all applications.

\textbf{9.2. Research Contributions}

In pursuing the research described in this thesis, a prime consideration has been to investigate the possibility to define a design framework that would allow an easy, fast and reliable process to generate high performance, neural network application-specific integrated circuits. It is believed that such design framework has been proved feasible. In this respect, this thesis has provided a simple, and at the same time, flexible architectural framework that can be easily configured to execute a particular neural application, with or without the learning procedure. In addition, a high level language has also been proposed, in order to afford high level specification of the neural application, freeing the user from the details of VLSI hardware design. In summary, it is felt that the basic research contributions of this thesis are:

\textbf{Generic Neuron Model} - A model for the specification of the artificial neuron's functionality. It incorporates into a single entity the main properties encountered in neural models, supporting the necessary flexibility to describe a wide range of neural network algorithms. In spite of its generality feature, the model is quite simple and its correspondent VLSI architecture yields a reasonable number of processing elements per chip.
**Generic Neuron Target Architecture** - A definition of the PE's general structure and the associated interconnection scheme to allow the production of massively parallel application-specific neurocomputers. The PE’s internal structure combines processing, interconnection and storage capabilities in a self-contained design, which can be ultimately configured by the user’s application definition. The interconnection strategy, on the other hand, provides the necessary generality to implement complex topologies and the required expansibility to match the application requirements in terms of number of PEs.

**VLSI Implementation** - A demonstrative prototype, specialised in the execution of the Back Propagation algorithm, used to assess the architecture’s packing capabilities in future implementations with denser technologies. The chip’s internal structure has been developed by exploiting the intrinsic properties related to the research areas of VLSI design, neural computing and parallel architectures, trying to combine high performance and minimum silicon area.

**Modular Datapath CMOS Library** - A full custom library for the design of registers and the processing element’s operative part. The library includes a large number of basic cells that follow a standard design for minimum silicon area and high performance. The datapath library is very modular and general, facilitating the silicon compilation process of the internal operative parts.

Finally, a high level specification language, namely nC, has also been proposed for the definition of neural network algorithms and applications. This task has been accomplished in conjunction with the Pygmalion group at UCL, in order to produce a flexible language used as the core of a neural network software programming environment. This language provides a concise form to describe algorithms and applications, either for simulation on a conventional machine or for direct translation into dedicated ASIC chips.

**9.3. Future Work**

As seen from the description of the work accomplished in this research, the generic neuron architecture and the nC specification language form the basis of a hardware development tool which is part of a complete neural network programming environment. Regarding the development of this hardware tool and its integration into a complete programming environment, there are still a number of tasks to be carried out. A short list of
the most interesting tasks is provided below:

- the development of a dedicated neural network silicon compiler based on the *generic neuron* architectural framework. The implementation of this silicon compiler will conclude the basic configuration of the hardware development tool, providing the link between the high level neural network specification language and the final application-specific neural chips.

- the implementation of a complete hardware emulator for the *generic neuron* architecture, incorporating feedback of the execution performance, as well as of the effects of the hardware-related parameters on the network execution. This hardware emulator will allow the user to experiment with the PE's internal parameters until the application requirements are matched in terms of accuracy and performance.

- integration into a neural network software environment. This integrated programming environment will encompass the hardware and software tools, providing the user with a complete and general environment. In such environment, neural applications can be simulated and optimally configured using a conventional (sequential or parallel) or a general-purpose neurocomputer, and then be automatically translated into high performance optimised ASIC chips, when required.

- the development of a neural network application to demonstrate the capabilities of the complete neural programming environment. The application will allow the investigation of the software and hardware tools' efficiency, testing the effectiveness of the software tool in defining and configuring the network, as well as analysing the results from the VLSI chips produced via the silicon compilation process.

The research work listed above constitutes a major part of the *Esprit II Galatea* project. This is a 3-year project (started in January 1991) and it involves a significant amount of research into the development of a complete neural network programming system. The system is composed of many modules combining:

- a sophisticated graphical programming environment, where users can create their own customised, application-specific graphical interfaces (in addition to the available general graphical monitor);

- an heterogeneous general-purpose neurocomputer, comprising primarily neural network boards provided by Siemens*105* and Philips*28*,
— scheduler/mapper modules, to supervise and distribute the neural network execution over the available general-purpose neural boards;

— a neural network silicon compiler, that will automatically translate a neural network application into ASICs, using the user’s high level description;

— several neural applications, such as optical character recognition and orange grading systems, to consolidate the project and verify its applicability.

The silicon compiler to be developed in the *Galatea* project is dedicated to recall mode only, being divided into two complementary parts: a high level synthesis module, that compiles the high level specification into a hardware description; and a low level synthesis module that actually generates the final VLSI layout from the hardware description language. Although the silicon compilation process is restricted to recall phase, its implementation will allow the investigation of virtual neuron mapping, required when the application imposes some restrictions on the maximum number of physical processors, forcing more than one neuron to be mapped into the same physical processor. The *Galatea* silicon compiler will also exploit the possibility to generate a general high level synthesis module, that will be able to utilise distinct architectural frameworks. This will allow the user to verify the best target architecture for his/her specific neural application prior to the actual chip fabrication. Currently, the *Galatea* project incorporates two target architectures, the *generic neuron* and the INPG⁹⁶ architectural frameworks. Studies will be carried out to evaluate both architectures and verify their capabilities and deficiencies.

A different approach to the development of a silicon compiler for neural networks is being developed at UCL as part of another PhD thesis⁸⁹. In this case, the focus is exclusively on the high level synthesis aspects of the compilation process, providing full implementation of both neural network phases i.e., recall and learning procedures. This silicon compiler will make use of the *nC* neural network specification language and will translate the application description into a VHDL (IEEE standard hardware description language)³ description of the *generic neuron* architecture, configured for the user’s application. This implementation will offer the possibility to accomplish a more concrete assessment of the suitability of the *nC* and the *generic neuron* architecture for the automatic production of high performance dedicated VLSI chips.
References

1. ES2-European Silicon Structures, "SOLO 2000," Product Information, ES2, Germ­
ering, West Germany, 1987.

2. SDA Systems, Inc., "Design Framework; Automatic Place and Route; Design
Analysis (Vol. 1 and 2); Design Entry; SDA Language; Physical Design and


7. Agranat, A.J., Neugebauer, C.F., and Yariv, A., "A CCD Based Neural Network In­
tegrated Circuit with 64K Analogue Programmable Synapses," Int. Joint Conf. on
Neural Networks - IJCNN 90, vol. II, pp. 551-555, San Diego, California, June 17-
21, 1990.

8. Aiken, S.W., Koch, M.W., and Roberts, M.W., "A Parallel Neural Network Simu­
lator," Int. Joint Conf. on Neural Networks - IJCNN 90, vol. II, pp. 611-616, San
Diego, California, June 17-21, 1990.

9. Aleksander, I., Thomas, W.V., and Bowden, P.A., "WISARD - A Radical Step For­

10. Aleksander, I., "Adaptive Pattern Recognition Systems and Boltzmann Machines: A
Rapprochement," Pattern Recognition Letters, vol. 6, no. 2, pp. 113-120, July
1987.


12. Alippi, C. and Nigri, M.E., "Hardware Requirements for Digital VLSI Implementa­
tion of Neural Networks," Int. Joint Conf. on Neural Networks (to appear), Singa­


86. Myers, D.J. and Brebner, G.E., “The Implementation of Hardware Neural Networks,” The First IEE Int. Conf. on Artificial Neural Networks, pp. 57-61, October 16-18, 1989.


113. Sietsma, J. and Dow, R.J.F., "Neural Net Pruning - Why and How," *Proc. IEEE Int. Conf. on Neural Networks*, vol. I, pp. 325-333, San Diego, California, July 24-


This appendix presents the syntax of the Pygmalion nC language.

program:
  algorithm_part
  algorithm_part main_definition

algorithm_part:
  include_list hash_def_list_opt alg_decla_list mandatory_functions external_decla_list_opt

main_definition:
  main (identifier_list_opt ) decla_list_opt { decla_list_opt stat_list } hash_endif_opt

include_list:
  C_include_list alg_include_list
  alg_include_list C_include_list

hash_def_list:
  define_def
  hash_def_list define_def

alg_decla_list:
  decla_list_opt class_type_list

mandatory_functions:
  rule_function_list connect_def build_rules_def

rule_function_list:
  rule_function_def
  rule_function_list rule_function_def

external_decla_list:
  external_declaration
  external_decla_list external_declaration

external_declaration:
  add_function_def
  declaration
  hash_declarations

C_include_list:
  #include <filename.h>
  C_include_list #include <filename.h>

alg_include_list:
  mandatory_includes
  mandatory_includes other_includes

mandatory_includes:
  #include "pygmalion.h" #include "sysdef.h" #include "config.h" #include "pgmrc.h"

other_includes:
  #include "filename.h"
  other_includes #include "filename.h"

define_def:
  #define identifier constant
hash_declarations:
  #if constant expression
  #ifdef identifier
  #ifndef identifier
  #else
  #undef identifier
  #endif
define_def

decla_list:
  declaration
decla_list declaration

class_type_list:
  class_type decla
class_type_list class_type decla

rule_function_def:
  rule_func_name ( identifier_list ) decla_list restricted_comp_statement
one_of_r_types rule_func_name ( identifier_list ) decla_list restricted_comp_statement

connect_def:
  one_of_types connect ( identifier_list ) decla_list compound_statement

build_rules_def:
  one_of_types build_rules ( identifier_list ) decla_list compound_statement

add_function_def:
  fun_declarator decla_list compound_statement
deca_specifiers fun_declarator decla_list compound_statement

fun_declarator:
  direct_fun_declarator
  pointer direct_fun_declarator

direct_fun_declarator:
  lower_case_identifier
  direct_fun_declarator ( identifier_list )

class_type decla:
  class_type lower_case_identifier_class = { class_decla_field } ;

class_decla_field:
  pygmalion_fun_name , string , int_constant , int_constant

pygmalion_fun_name:
  rule_func_name
  built_in_fn_name

declaration:
  deca_specifiers ;
decla_specifiers init_declarator_list ;

deca_specifiers:
  storage_class_spec deca_specifiers_opt
type_specifier deca_specifiers_opt
type_qualifier deca_specifiers_opt

init_declarator_list:
  init_declarator
  init_declarator_list , init_declarator

init_declarator:
  declarator
  declarator = initialiser

type_specifier:
  r_typeSpecifier
  one_of_add_types
  one_of_add_structs
  one_of_patterns
  struct_union_spec
  enum_specifier

r_typeSpecifier:
  one_of_r_structs
  one_of_r_types
struct_union_spec:
    struct_union ( struct_decla_list )
    struct_union_identifier ( struct_decla_list )
    struct_union_identifier

struct_decla_list:
    struct_declaration
    struct_decla_list struct_declaration

struct_declaration:
    spec_qual_list st_declarator_list;

spec_qual_list:
    type_specifier
    type_specifier spec_qual_list
    type_qualifier
    type_qualifier spec_qual_list

st_declarator_list:
    struct_declarator
    st_declarator_list , struct_declarator

struct_declarator:
    declarator :
        constant_expression
    declarator :
        constant_expression

direct_declarator:
    ( declarator )
    direct_declarator [ ]
    direct_declarator [ constant_expression ]
    direct_declarator ( identifier_list_opt )

pointer:
    *
    * type_qual_list
    * pointer
    * type_qual_list pointer

type_qual_list:
    type_qualifier
    type_qual_list type_qualifier

identifier_list:
    identifier
    identifier_list , identifier

initialiser:
    assign_expression
    { initialiser_list } { initialiser_list , }

initialiser_list:
    initialiser
    initialiser_list , initialiser

compound_statement:
    { decla_list_opt }
    { decla_list_opt stat_list }

176
stat_list:
  statement
  stat_list  statement

statement:
  labeled_statement
  expression_statement
  compound_statement
  selection_statement
  iteration_statement
  jump_statement
  control_statement

labeled_statement:
  identifier : statement
  case constant_expression : statement
  default : statement

expression_statement:
  
  expression ;

selection_statement:
  if ( expression ) statement
  if ( expression ) statement else statement
  switch ( expression ) statement

iteration_statement:
  while ( expression ) statement
  do statement while ( expression ) ;
  for_loop

for_loop:
  for ( expression_opt ) ; expression_opt ; expression_opt ) statement

jump_statement:
  goto identifier ;
  continue ;
  break ;
  return expression_opt ;

control_statement:
  PAR { stat_list }
  PAR for_loop

expression:
  assign_expression
  expression , assign_expression

constant_expression:
  conditional_expression

assign_expression:
  conditional_expression
  assign_operator assign_expression

conditional_expression:
  logic_or_expression
  logic_or_expression ? expression : conditional_expression

logic_or_expression:
  logic_and_expression
  logic_or_expression || logic_and_expression

logic_and_expression:
  incl_or_expression
  logic_and_expression && incl_or_expression

incl_or_expression:
  excl_or_expression
  logic_and_expression | incl_or_expression

excl_or_expression:
  excl_or_expression
  excl_or_expression | excl_or_expression

and_expression:
  incl_or_expression
  excl_or_expression & incl_or_expression

equality_expression
  and_expression
  and_expression & equality_expression
equality_expression:
  relational_expression
  equality_expression == relational_expression
  equality_expression != relational_expression

relational_expression:
  shift_expression
  relational_expression < shift_expression
  relational_expression > shift_expression
  relational_expression <= shift_expression
  relational_expression >= shift_expression

shift_expression:
  additive_expression
  shift_expression <<= additive_expression
  shift_expression >>= additive_expression

additive_expression:
  multiplicative_expression
  additive_expression + multiplicative_expression
  additive_expression - multiplicative_expression

multiplicative_expression:
  cast_expression
  multiplicative_expression * cast_expression
  multiplicative_expression / cast_expression
  multiplicative_expression % cast_expression

cast_expression:
  unary_expression
  ( type_name ) cast_expression

unary_expression:
  postfix_expression
  ++ unary_expression
  -- unary_expression
  unary_operator cast_expression
  sizeof unary_expression
  sizeof ( type_name )

postfix_expression:
  primary_expression
  postfix_expression [ expression ]
  postfix_expression [ arg_expr_list ]
  postfix_expression ()
  postfix_expression .identifier
  postfix_expression ->identifier
  postfix_expression ++
  postfix_expression --

arg_expr_list:
  assign_expression
  arg_expr_list , assign_expression

primary_expression:
  constant
  ( expression )

type_name:
  spec_qual_list
  spec_qual_list abstract_decl

abstract_decl:
  pointer
  pointer direct_abstr_decl
  direct_abstr_decl

direct_abstr_decl:
  ( abstract_decl )
  direct_abstr_decl [ constant_expression ]
  direct_abstr_decl [ ]
  [ constant_expression ]
  [ ]

restricted_comp_statement:
  { r_decla_listopt }
restricted_stat_list:
  restricted_statement
  restricted_stat_list restricted_statement

restricted_statement:
  r_expression_statement
  r_selection_statement
  r_iteration_statement
  r_jump_statement
  r_control_statement

r_expression_statement:
  ;
  r_assign_expression

r_selection_statement:
  if (r_cond_expression) r_body_statement
  if (r_cond_expression) r_body_statement else r_body_statement

r_iteration_statement:
  r_for_loop

r_for_loop:
  for (r_assign_expression, r_cond_expression, r_assign_expression) r_body_statement

r_body_statement:
  restricted_stat_list
  restricted_statement

r_jump_statement:
  continue;
  break;
  return r_expression;

r_control_statement:
  PAR { restricted_stat_list }
  PAR r_for_loop

r_decl_list:
  r_declarator
  r_decl_list r_declarator

r_declarator:
  r_type_specifier r_init_declarator

r_init_declarator:
  r_declarator
  r_declarator = r_initializer

r_direct_declarator:
  r_declarator
  r_direct_declarator
  r_direct_declarator
  r_direct_declarator

r_initializer:
  r_additive_expression
  r_initializer_list
  r_initializer_list

r_initializer_list:
  r_initializer
  r_initializer_list, r_initializer

r_expression:
  r_assign_expression

r_cond_expression:
  r_logic_or_expression
r_assign_expression:
  r_logic_or_expression
  r_unary_expression assign_operator r_logic_or_expression

r_logic_or_expression:
  r_logic_and_expression
  r_logic_or_expression r_logic_or_expression

r_logic_and_expression:
  r_incl_or_expression
  r_logic_and_expression && r_logic_and_expression

r_incl_or_expression:
  r_excl_or_expression
  r_incl_or_expression | r_excl_or_expression

r_excl_or_expression:
  r_and_expression
  r_excl_or_expression ^ r_and_expression

r_and_expression:
  r_equality_expression
  r_and_expression & r_equality_expression

r_equality_expression:
  r_relational_expression
  r_equality_expression == r_relational_expression
  r_equality_expression != r_relational_expression

r_relational_expression:
  r_shift_expression
  r_relational_expression < r_shift_expression
  r_relational_expression > r_shift_expression
  r_relational_expression <= r_shift_expression
  r_relational_expression >= r_shift_expression

r_shift_expression:
  r_additive_expression
  r_shift_expression << r_additive_expression
  r_shift_expression >> r_additive_expression

r_additive_expression:
  r_multipl_expression
  r_additive_expression + r_multipl_expression
  r_additive_expression - r_multipl_expression

r_multipl_expression:
  r_cast_expression
  r_multipl_expression * r_cast_expression
  r_multipl_expression / r_cast_expression
  r_multipl_expression % r_cast_expression

r_cast_expression:
  r_unary_expression
  ( r_type_name ) r_cast_expression

r_unary_expression:
  r_postfix_expression
  ++ r_unary_expression
  -- r_unary_expression
  unary_operator r_cast_expression

r_postfix_expression:
  r_primary_expression
  r_postfix_expression [ r_additive_expression ]
  r_postfix_expression ( arg_expr_list )
  r_postfix_expression ( )
  r_postfix_expression identifier
  r_postfix_expression identifier
  r_postfix_expression +
  r_postfix_expression --

r_primary_expression:
  constant
  ( r_expression )

r_type_name:
  r_spec_qual_list
  r_spec_qual_list r_abstract_decl

180
r_spec_qual_list:
  type_specifier
type_specifier r_spec_qual_list

r_abstract_decl:
  r_pointer
  r_pointer r_dir_abstr_decl
  r_dir_abstr_decl

r_pointer:
  *
double_star
double_star:
  **

r_dir_abstr_decl:
  ( )
  ( r_abstract_decl )
  r_dir_abstr_decl [ ]
  r_dir_abstr_decl [ r_constant_expression ]
  []
  [ r_constant_expression ]
  r_dir_abstr_decl ( )

r_constant_expression:
  r_additive_expression

rule_fun_name:
  upper_case_identifier

built_in_fn_name: one of
  pexec sexec pexec_r sexec_r pexec_c sexec_c

storage_class_spec: one of
  auto register static extern

one_of_add_structs: one of
  tagval_type unval_type

one_of_r_structs: one of
  synapse_type neuron_type cluster_type layer_type net_type system_type rule_type para_type
exec_type

one_of_types:
  one_of_r_types
  one_of_add_types

one_of_add_types: one of
  void short long double signed unsigned

one_of_r_types: one of
  char int float caddr_t

one_of_patterns: one of
  any_elem pat_elem int_elem

type_qualifier: one of
  const volatile

struct_union: one of
  struct union

constant:
  int_constant
  float_constant
  hex_constant
  char_constant

unary_operator: one of
  & * + - ~ !

assign_operator: one of
  = += -= *= /= %= <<= >>= &= ^= |=

181
This appendix presents technical information on the principal cells of the datapath cell library.

**Logical Functions:**

<table>
<thead>
<tr>
<th>Cell Names:</th>
<th>Basic register</th>
</tr>
</thead>
</table>

**Description:**

Basic register controlled by phiIL, phi1H, phi2L and phi2H.

**Author:**

Marco Pacheco

**Last Updating:**

27/06/90

**Last Upd. done by:**

Marco Pacheco

**Signals:**

<table>
<thead>
<tr>
<th>Up/Layer</th>
<th>Down/Layer</th>
<th>Right/Layer</th>
<th>Left/Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>Y1</td>
<td>Xpolyc</td>
<td>B1</td>
</tr>
<tr>
<td>A1</td>
<td>Y1</td>
<td>Xpolyc</td>
<td>B1</td>
</tr>
<tr>
<td>phiH2</td>
<td>phi2H2</td>
<td>phi2L2</td>
<td>phi1L2</td>
</tr>
<tr>
<td>phiH2</td>
<td>phi2H2</td>
<td>phi2L2</td>
<td>phi1L2</td>
</tr>
</tbody>
</table>

**Input Signals:**

X

**Output Signals:**

Y

**Control Signals:**

phi1H phi2H phi1L phi2L

**Location:**

`marley/chip/mag/cells`

**Size:**

h = 60  w = 68

**Comments:**

It is transparent to buses A and B only.

**Variations:**

- reglr.mag is composed of two reg.mag cells, which are mirror images of each other.

- reg16.mag is composed of an array of 8 reglr.mag cells, concatenated along supply rails.

- reg4_4_4.mag is composed of (from left to right): 2 cells reglr.mag, 2 cells nullreg.mag, 2 cells reglr.mag, 2 cells nullreg.mag and 2 cells reglr.mag.

- reg7_7.mag is composed of (left to right): 3 cells reglr.mag, one cell reg.mag, 2 cells nullreg.mag, one cell reg.mag (sideways) and 3 cells reglr.mag.
### Logical Functions: Reset Register

| Description:                | a pass-transistor (n-type) that connects GND to the X input of a register, whenever rst signal is active. |
| Author:                     | Marco Pacheco |
| Last Updating:              | 27/06/90 |
| Last Upd. done by:          | Marco Pacheco |
| Signals:                    | Up/Layer: A1, Y1, X1, Z1, B1  
                            | Down/Layer: A1, Y1, X1, Z1, B1  
                            | Right/Layer: rst2  
                            | Left/Layer: rst2 |
| Input Signals:              | GND  
                            | X |
| Output Signals:             |  
| Control Signals:            | rst |
| Location:                   | marley/chip/mag/cells  
                            | h = 11  
                            | w = 68 |
| Size:                       |  
| Comments:                   | The cell is transparent to buses A, B and Z. |
| Variations:                 | - resetxlr.mag is composed of two resetx.mag cells, which are mirror images of each other.  
                            | - resetx16.mag is composed of an array of 8 resetxlr.mag cells, concatenated along supply rails.  
                            | - resetx4_4_4.mag is composed of (left to right): 2 cells resetxlr.mag, 2 cells nullbus.mag, 2 cells resetxlr.mag, 2 cells nullbus.mag and 2 cells resetxlr.mag. |

### Logical Functions: Preset Register

| Description:                | a pass-transistor (n-type) that connects Vdd to the X input of a register, whenever pr signal is active. |
| Author:                     | Marco Pacheco |
| Last Updating:              | 27/06/90 |
| Last Upd. done by:          | Marco Pacheco |
| Signals:                    | Up/Layer: A1, Y1, X1, Z1, B1  
                            | Down/Layer: A1, Y1, X1, Z1, B1  
                            | Right/Layer: pr2  
                            | Left/Layer: pr2 |
| Input Signals:              | Vdd  
                            | X |
| Output Signals:             |  
| Control Signals:            | pr |
| Location:                   | marley/chip/mag/cells  
                            | h = 11  
                            | w = 68 |
| Size:                       |  
| Comments:                   | This cell is transparent to buses A, B and Z. |
| Variations:                 | - presetxlr.mag is composed of two presetx.mag cells, which are mirror images of each other.  
                            | - presetx16.mag is composed of an array of 8 presetxlr.mag cells, concatenated along supply rails. |
### Logical Functions: Read register into bus A

**Cell Names:**
- busArd.mag, busArdlr.mag, busArdl6.mag

**Description:**
Reads from the output Y of a register and writes into bus A, whenever rdA signal is active.

**Author:**
Marco Pacheco

**Last Updating:**
27/06/90

**Last Upd. done by:**
Marco Pacheco

**Signals:**
- **Up/Layer:** A1, Y1, X1, Z1, B1
- **Down/Layer:** A1, Y1, X1, Z1, B1
- **Right/Layer:** rdA2
- **Left/Layer:** rdA2

**Input Signals:**
- Y

**Output Signals:**
- A
- rdA

**Control Signals:**

**Location:**
- marley/chip/mag/cells
- h = 11, w = 68

**Comments:**
The cell is transparent to buses A, B and Z.

**Variations:**
- busArdlr.mag is composed of two busArd.mag cells, which are mirror images of each other.
- busArdl6.mag is composed of an array of 8 busArdlr.mag cells, concatenated along supply rails.

### Logical Functions: Write into register from bus A

**Cell Names:**
- busAwl.mag, busAwlrl.mag, busAwl16.mag

**Description:**
Writes into the input X of a register what is in bus A, whenever wtA signal is active.

**Author:**
Marco Pacheco

**Last Updating:**
27/06/90

**Last Upd. done by:**
Marco Pacheco

**Signals:**
- **Up/Layer:** A1, Y1, X1, Z1, B1
- **Down/Layer:** A1, Y1, X1, Z1, B1
- **Right/Layer:** wtA2
- **Left/Layer:** wtA2

**Input Signals:**
- A
- X
- wtA

**Output Signals:**

**Control Signals:**

**Location:**
- marley/chip/mag/cells
- h = 11, w = 68

**Comments:**
This cell is transparent to buses A, B and Z.

**Variations:**
- busAwlrl.mag is composed of two busAwl.mag cells, which are mirror images of each other.
- busAwl16.mag is composed of an array of 8 busAwlrl.mag cells, concatenated along supply rails.
### Logical Functions: Write register from top register

<table>
<thead>
<tr>
<th>Cell Names:</th>
<th>topreg2reg.mag, topreg2reglr.mag, topreg2reg16.mag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>writes into the input X of a register what is in the output Y of another register that is at the top of the first one, whenever wtR signal is active.</td>
</tr>
<tr>
<td>Author:</td>
<td>Marco Pacheco</td>
</tr>
<tr>
<td>Last Updating:</td>
<td>27/06/90</td>
</tr>
<tr>
<td>Last Upd. done by:</td>
<td>Marco Pacheco</td>
</tr>
<tr>
<td>Signals:</td>
<td></td>
</tr>
<tr>
<td>Up/Layer</td>
<td>A1</td>
</tr>
<tr>
<td>Down/Layer</td>
<td>A1</td>
</tr>
<tr>
<td>Right/Layer</td>
<td>wtR2</td>
</tr>
<tr>
<td>Left/Layer</td>
<td>wtR2</td>
</tr>
<tr>
<td>Input Signals:</td>
<td>Y</td>
</tr>
<tr>
<td>Output Signals:</td>
<td>X</td>
</tr>
<tr>
<td>Control Signals:</td>
<td>wtR</td>
</tr>
<tr>
<td>Location:</td>
<td>marley/chip/mag/cells</td>
</tr>
<tr>
<td>Size:</td>
<td>h = 13  w = 68</td>
</tr>
<tr>
<td>Comments:</td>
<td>This cell is transparent to buses A, B and Z.</td>
</tr>
<tr>
<td>Variations:</td>
<td>- topreg2reglr.mag is composed of two topreg2reg.mag cells, which are mirror images of each other.</td>
</tr>
<tr>
<td></td>
<td>- topreg2reg16.mag is composed of an array of 8 topreg2reglr.mag cells, concatenated along supply rails.</td>
</tr>
</tbody>
</table>

### Logical Functions: Write register from bottom register

<table>
<thead>
<tr>
<th>Cell Names:</th>
<th>botreg2reg.mag, botreg2reglr.mag, botreg2reg16.mag, botreg2reg7_7.mag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>writes into the input X of a register what is in the output Y of another register that is at the bottom of the first one, whenever wtR signal is active.</td>
</tr>
<tr>
<td>Author:</td>
<td>Marco Pacheco</td>
</tr>
<tr>
<td>Last Updating:</td>
<td>27/06/90</td>
</tr>
<tr>
<td>Last Upd. done by:</td>
<td>Marco Pacheco</td>
</tr>
<tr>
<td>Signals:</td>
<td></td>
</tr>
<tr>
<td>Up/Layer</td>
<td>A1</td>
</tr>
<tr>
<td>Down/Layer</td>
<td>A1</td>
</tr>
<tr>
<td>Right/Layer</td>
<td>wtR2</td>
</tr>
<tr>
<td>Left/Layer</td>
<td>wtR2</td>
</tr>
<tr>
<td>Input Signals:</td>
<td>Y</td>
</tr>
<tr>
<td>Output Signals:</td>
<td>X</td>
</tr>
<tr>
<td>Control Signals:</td>
<td>wtR</td>
</tr>
<tr>
<td>Location:</td>
<td>marley/chip/mag/cells</td>
</tr>
<tr>
<td>Size:</td>
<td>h = 13  w = 68</td>
</tr>
<tr>
<td>Comments:</td>
<td>This cell is transparent to buses A, B and Z.</td>
</tr>
<tr>
<td>Variations:</td>
<td>- botreg2reglr.mag is composed of two botreg2reg.mag cells, which are mirror images of each other.</td>
</tr>
<tr>
<td></td>
<td>- botreg2reg16.mag is composed of an array of 8 botreg2reglr.mag cells, concatenated along supply rails.</td>
</tr>
<tr>
<td></td>
<td>- botreg2reg7_7.mag is composed of (left to right): 3 cells botreg2reglr.mag, one cell botreg2reg.mag, 2 cell nullreg2reg.mag, one cell botreg2reg.mag (sideways) and 3 cells botreg2reglr.mag.</td>
</tr>
<tr>
<td>Logical Functions:</td>
<td>2-Input Comparator</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td><strong>Cell Names:</strong></td>
<td>comp.mag, compu.mag, complr.mag, comp16.mag, comp4_4_4.mag</td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td>an exclusive-or that compares two inputs (Y1 and Y2) and puts the result in the compout output.</td>
</tr>
<tr>
<td><strong>Author:</strong></td>
<td>Marco Pacheco</td>
</tr>
<tr>
<td><strong>Last Updating:</strong></td>
<td>04/07/90</td>
</tr>
<tr>
<td><strong>Last Upd. done by:</strong></td>
<td>Marley Vellasco (compu.mag)</td>
</tr>
<tr>
<td><strong>Signals:</strong></td>
<td></td>
</tr>
<tr>
<td>Up/Layer</td>
<td>A1  Y2i  Y1i  B1</td>
</tr>
<tr>
<td>Down/Layer</td>
<td>A1  Y1i  Y2i  B1</td>
</tr>
<tr>
<td>Right/Layer</td>
<td>compout2</td>
</tr>
<tr>
<td>Left/Layer</td>
<td></td>
</tr>
<tr>
<td><strong>Input Signals:</strong></td>
<td>Y1  Y2</td>
</tr>
<tr>
<td><strong>Output Signals:</strong></td>
<td>compout</td>
</tr>
<tr>
<td><strong>Control Signals:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Location:</strong></td>
<td>&quot;marley/chip/mag/cells</td>
</tr>
<tr>
<td><strong>Size:</strong></td>
<td>h = 64  w = 68</td>
</tr>
<tr>
<td><strong>Comments:</strong></td>
<td>- The compout output of comp.mag cell is of &quot;wired-or&quot; type, which means that the output has only the pull-down n-type transistor. When designing a comparator of n bits, the output of n-1 comp.mag cells are wired together, with only the n-th cell containing the pull-up p-type transistor. The n-th cell should be compu.mag.</td>
</tr>
<tr>
<td></td>
<td>- This cell is transparent to buses A and B only.</td>
</tr>
<tr>
<td><strong>Variations:</strong></td>
<td>- compu.mag is exactly the same as comp.mag with an additional pull-up p-type transistor connected to the compout output. (h = 64, w = 66).</td>
</tr>
<tr>
<td></td>
<td>- complr.mag is composed of two comp.mag cells, which are mirror images of each other.</td>
</tr>
<tr>
<td></td>
<td>- comp16.mag is composed of an array of 7 complr.mag cells, concatenated along supply rails, and, for the two most significant bits, one comp.mag (MSB-1) and one compu.mag (MSB), also concatenated along supply rails.</td>
</tr>
<tr>
<td></td>
<td>- comp4_4_4.mag is formed by three groups composed of (left to right): 1 cell compu.mag, 1 cell comp.mag (sideways), 1 cell complr.mag. These groups are separated from each other by 2 cells nullcomp.mag.</td>
</tr>
<tr>
<td><strong>Logical Functions:</strong></td>
<td><strong>UP Counter and its variations</strong></td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td><strong>Cell Names:</strong></td>
<td>upl.mag, uplbuf.mag, upr.mag, uprbuf.mag, up4bits.mag, up16.mag, up4_4_4.mag, up7_7.mag, up16p.mag</td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td>Gets the Y output of a register as the input, counts up, and writes the result in the X output, whenever the wtX signal is active.</td>
</tr>
<tr>
<td><strong>Author:</strong></td>
<td>Marco Pacheco</td>
</tr>
<tr>
<td><strong>Last Updating:</strong></td>
<td>13/09/90</td>
</tr>
<tr>
<td><strong>Last Upd. done by:</strong></td>
<td>Marco Pacheco</td>
</tr>
</tbody>
</table>
| **Signals:**           | A1 Y polyc B1  
Down/Layer            | A1 Y1 X1 B1 |
Right/Layer            | cinL2 cinH2 wtX2 |
Left/Layer             | coutL2 coutH2 wtX2 |
| **Input Signals:**     | Y cinL cinH |
| **Output Signals:**    | X |
| **Control Signals:**   | wtX |
| **Location:**          | marley/chip/mag/cells |
| **Size:**              | h = 88  w = 68 |
| **Comments:**          | It is transparent to buses A and B only. |
| **Variations:**        | - upr.mag is basically the same as upl.mag cell. The difference is in the layout of the carry-in and carry-out signals. This has been done in order to still have them in the correct side of the cell (despite the sideways) that is, cin coming in from the right and cout going out from the left. |
|                        | - uplbuf.mag and uprbuf.mag are the same as upl.mag and upr.mag, with 2 additional inverters to bufferise the coutH and coutL signals. |
|                        | - up4bits.mag is composed of (left to right): 1 cell uplbuf.mag, 1 cell upr.mag, 1 cell upl.mag and 1 cell upr.mag. |
|                        | - up16.mag is composed of an array of 4 up4bits.mag cells, concatenated along supply rails. |
|                        | - up16p.mag is similar to up16.mag but with the cinL and cinH of the LSB connected to GND and VDD, respectively. |
|                        | - up4_4_4.mag is formed of 3 blocks, concatenated along supply rails, and composed of (from left to right): 1 cell upl.mag, 1 cell uprbuf.mag, 1 cell upl.mag, 1 cell upr.mag (with one cell ctcinplugr.mag superposed) and 2 cells nullcounter.mag (the third block does not contain the null cells). |
|                        | - up7_7.mag is composed of (left-to-right): 1 cell up1.lmag, 1 cell upr.mag, 1 cell upl.mag, 1 cell uprbuf.mag, 1 cell upl.mag, 1 cell upr.mag, 1 cell upl.mag (with one cell ctcinplugl.mag superposed), 2 cells nullcounter.mag, 1 cell upr.mag, 1 cell upl.mag, 1 cell upr.mag and 1 cell up4bits.mag (with 1 cell ctcinplugr.mag superposed). |

187
### Logical Functions: SHIFT left/right

<table>
<thead>
<tr>
<th><strong>Cell Names:</strong></th>
<th>shift.mag, shiftr.mag, shiftlr.mag, shift16.mag</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong></td>
<td>Gets the Y output signal of a register and puts either in the X input of the register adjacent to the right, if SR is active, or in the X input of the register adjacent to the left, if SL is active.</td>
</tr>
<tr>
<td><strong>Author:</strong></td>
<td>Marco Pacheco</td>
</tr>
<tr>
<td><strong>Last Updating:</strong></td>
<td>27/06/90</td>
</tr>
<tr>
<td><strong>Last Upd. done by:</strong></td>
<td>Marco Pacheco</td>
</tr>
<tr>
<td><strong>Signals:</strong></td>
<td></td>
</tr>
<tr>
<td>Up/Layer</td>
<td>A1 Y1 X1 B1</td>
</tr>
<tr>
<td>Down/Layer</td>
<td>A1 Y1 X1 B1</td>
</tr>
<tr>
<td>Right/Layer</td>
<td>SR2 SL2</td>
</tr>
<tr>
<td>Left/Layer</td>
<td>SR2 SL2</td>
</tr>
<tr>
<td><strong>Input Signals:</strong></td>
<td>Y</td>
</tr>
<tr>
<td><strong>Output Signals:</strong></td>
<td>X</td>
</tr>
<tr>
<td><strong>Control Signals:</strong></td>
<td>SR SL</td>
</tr>
<tr>
<td><strong>Location:</strong></td>
<td>marley/chip/mag/cells</td>
</tr>
<tr>
<td><strong>Size:</strong></td>
<td>h = 22 w = 68</td>
</tr>
<tr>
<td><strong>Comments:</strong></td>
<td>It is transparent to buses A and B only.</td>
</tr>
<tr>
<td><strong>Variations:</strong></td>
<td>- shift16.mag is composed of an array of 8 shiftlr.mag cells, concatenated along supply rails.</td>
</tr>
</tbody>
</table>

### Logical Functions: 32 bits shift left/right

<table>
<thead>
<tr>
<th><strong>Cell Names:</strong></th>
<th>shift2x16.mag, ccshiftup16.mag, ccshiftdw16.mag</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong></td>
<td>A 32-bit shift register composed of 2 16-bit shift registers.</td>
</tr>
<tr>
<td><strong>Author:</strong></td>
<td>Marco Pacheco</td>
</tr>
<tr>
<td><strong>Last Updating:</strong></td>
<td>13/09/90</td>
</tr>
<tr>
<td><strong>Last Upd. done by:</strong></td>
<td>Marco Pacheco</td>
</tr>
<tr>
<td><strong>Signals:</strong></td>
<td></td>
</tr>
<tr>
<td>Up/Layer</td>
<td>A1 Y1 X1 B1</td>
</tr>
<tr>
<td>Down/Layer</td>
<td>A1 Y1 X1 B1</td>
</tr>
<tr>
<td>Right/Layer</td>
<td>SR2 SL2</td>
</tr>
<tr>
<td>Left/Layer</td>
<td>SR2 SL2</td>
</tr>
<tr>
<td><strong>Input Signals:</strong></td>
<td>Y</td>
</tr>
<tr>
<td><strong>Output Signals:</strong></td>
<td>X</td>
</tr>
<tr>
<td><strong>Control Signals:</strong></td>
<td>SR SL</td>
</tr>
<tr>
<td><strong>Location:</strong></td>
<td>marley/chip/mag/cells</td>
</tr>
<tr>
<td><strong>Size:</strong></td>
<td>h = see variations w = see variations</td>
</tr>
<tr>
<td><strong>Comments:</strong></td>
<td>- These cells are transparent to buses A and B only.</td>
</tr>
<tr>
<td><strong>Variations:</strong></td>
<td>- shift2x16.mag is the 32-bit shift register, composed of two separate 16-bit shift registers. It is composed of (top to bottom): 1 cell ccshiftup16.mag, 1 cell concat16.mag and 1 cell ccshiftdw16.mag.</td>
</tr>
<tr>
<td></td>
<td>- ccshiftup16.mag (w = 958, h = 22) is composed of (left to right): 1 cell shiftxce.mag, 1 cell shiftr.mag, 6 cells shiftlr.mag, 1 cell shift.mag and 1 cell shiftzero.mag. This cell is used as the top 16-bit shift register to form a 32-bit shift register.</td>
</tr>
<tr>
<td></td>
<td>- ccshiftdw16.mag (w = 953, h = 22) is composed of (left to right): 1 cell shiftself.mag, 1 cell shiftr.mag, 6 cells shiftlr.mag, 1 cell shift.mag and 1 cell shifttech.mag. This cell is used as the bottom 16-bit shift register to form a 32-bit shift register.</td>
</tr>
</tbody>
</table>
**Logical Functions:**  
**Cell Names:** nalu.mag, nalur.mag, nalubuf.mag, nalu4bits.mag, nalu16.mag, nalu16p.mag  
**Description:** Arithmetic and Logic Unit, controlled by 12 bits: K0-K3 (Kill carry), P0-P3 (Propagate carry) and R0-R3 (Execution code).  
**Author:** Marco Pacheco  
**Last Updating:** 13/09/90  
**Last Upd. done by:** Marco Pacheco  

**Signals:**  
<table>
<thead>
<tr>
<th>Up/Layer</th>
<th>Down/Layer</th>
<th>Right/Layer</th>
<th>Left/Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>alub</td>
<td>aluoutb</td>
<td>ai</td>
</tr>
<tr>
<td>A1</td>
<td>coutH</td>
<td>aluout</td>
<td>aluai</td>
</tr>
<tr>
<td>K3-K0ndiff</td>
<td>P3-P0ndiff</td>
<td>cinLpoly</td>
<td>cinHi poly</td>
</tr>
<tr>
<td>K3-K0ndiff</td>
<td>P3-P0ndiff</td>
<td>coutLpoly</td>
<td>coutHi poly</td>
</tr>
</tbody>
</table>

**Input Signals:** cinL, cinH, alua, alub  
**Output Signals:** coutL, coutH, aluout, aluoutb  
**Control Signals:** K0-K3, P0-P3, R0-R3  

**Location:** marley/chip/mag/cells  
**Size:** h = 452, w = 68  
**Comments:** - These cells are transparent to buses A and B only.  
**Variations:**  
- **nalur.mag** is basically the same as **nalu.mag** cell. The difference is in the layout of the carry-in and carry-out signals. This has been done in order to still have them in the correct side of the cell (despite the sideways movement), that is, CIN coming in from the right and COUT going out from the left.  
- **nalubuf.mag** is the **nalu.mag** with 2 additional inverters for cinL and cinH signals, in order to recover the signals to continue the transmission through the cell.  
- **nalu4bits** is composed of (left to right): 1 cell **nalubuf.mag**, 1 cell **nalur.mag**, 1 cell **nalu.mag** and 1 cell **nalur.mag**.  
- **nalu16.mag** is composed of an array of 4 **nalu4bits** cells, concatenated along supply rails.  
- **nalu16p.mag** is the same as **nalu16.mag** with the additional plugs - aluplugl.mag and aluplugr - to transform ndiff and poly layers to metal 1.
Logical Functions: Uni-directional tri-state buffer

Cell Names:

Description:
Uni-directional tri-state buffer, with enout and enoutb as control lines.

Author:
Marley Vellasco
Last Updating:
12/09/90

Signals:

<table>
<thead>
<tr>
<th>Up/Layer</th>
<th>Down/Layer</th>
<th>Right/Layer</th>
<th>Left/Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>Y1</td>
<td>X1</td>
<td>Z1</td>
</tr>
<tr>
<td>A1</td>
<td>Y1</td>
<td>X1</td>
<td>Z1</td>
</tr>
<tr>
<td>dpout</td>
<td>dpoutb</td>
<td>soloinb</td>
<td>soloin</td>
</tr>
<tr>
<td>dpout</td>
<td>dpoutb</td>
<td>soloinb</td>
<td>soloin</td>
</tr>
</tbody>
</table>

Input Signals: see comments
Output Signals: see comments
Control Signals: dpout dpoutb soloin soloinb

Location: marley/chip/mag/cells
Size: h = 62 w = 68

Comments:
- The input and output signals depend on which bus (A, B, Y or Z) is being bufferised.
- Each cell is transparent to all buses, except to the one which is being bufferised.

Variations:
- uni3st.mag is the basic cell that is used in all variations, namely uni3stA.mag, uni3stB.mag, uni3stZ.mag and uni3stY.mag. This cell does NOT contain any bus.
- uni3stlr.mag buffers the bus ? and is transparent to all other buses.
- uni3stlr.mag is composed of two uni3stlr.mag cells, which are mirror images of each other.
- uni3st16.mag is composed of an array of 8 uni3stlr.mag cells, concatenated along supply rails.

---

Logical Functions: Bi-directional tri-state buffer

Cell Names:

Description:
Bi-directional tri-state buffer, with different control lines for each direction.

Author:
Marley Vellasco
Last Updating:
13/08/90

Signals:

<table>
<thead>
<tr>
<th>Up/Layer</th>
<th>Down/Layer</th>
<th>Right/Layer</th>
<th>Left/Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>Y1</td>
<td>X1</td>
<td>Z1</td>
</tr>
<tr>
<td>A1</td>
<td>Y1</td>
<td>X1</td>
<td>Z1</td>
</tr>
<tr>
<td>dpout2</td>
<td>dpoutb2</td>
<td>soloin2</td>
<td>soloin</td>
</tr>
<tr>
<td>dpout2</td>
<td>dpoutb2</td>
<td>soloin2</td>
<td>soloin</td>
</tr>
</tbody>
</table>

Input Signals: see comments
Output Signals: see comments
Control Signals: soloin soloinb

Location: marley/chip/mag/cells
Size: h = 131 w = 68

Comments:
- The input and output signals depend on which bus (A, B or Z) is being bufferised.
- Each cell is transparent to all buses, except to the one which is being bufferised.

Variations:
- bi3st.mag is the basic cell that is used in all variations, namely bi3stA.mag, bi3stB.mag and bi3stZ.mag. This cell does NOT contain any bus.
- bi3stlr.mag buffers the bus ? and is transparent to all other buses.
- bi3stlr.mag is composed of two bi3stlr.mag cells, which are mirror images of each other.
- bi3st16.mag is composed of an array of 8 bi3stlr.mag cells, concatenated along supply rails.
Appendix 3

Back Propagation Prototype Schematics

This appendix illustrates the structure of the Back Propagation prototype chip by presenting some sample diagrams.
ALU_and_Accumulator
Appendix 4

Back Propagation Algorithm coded in $nC$

This appendix presents the Back Propagation algorithm coded in the $nC$ neural network specification language. It comprises the three basic files to develop a complete neural network algorithm/application: the $bp.c$ file that contains the parameterised description of the Back Propagation algorithm; the $bpconfig.h$ which describes the network configuration for the specific application; and finally the $sysdef.h$ that contains all the built-in $nC$ structures, including the hierarchical system structure.
/* Input vector;  * to receive input pattern */
float *input_vector;

/* Target vector;  * to receive target pattern */
float *target_vector;

int input_pattern_control = INPUT_PATTERN_CONTROL;
int target_pattern_control = INPUT_PATTERN_CONTROL;

/* Definitions of the back-propagation rule classes etc. */
/* */
/* */
/* Low level functions for the back-propagation model */
int State_update();
int Error_output();
int Error_hidden();
int Weight_update();
int Tolerance();

/* Specify maximum counts of rules and parameters at all levels */
int xc [ 6 ] [ ] = { 
  0, 0, /* system */
  5, 6, /* net */
  2, 1, /* layer */
  3, 0, /* cluster */
  3, 0, /* neuron */
  0, 0 /* synapse */
};

/* Names for parameters */
char *name [ 6 ] [ ] = { 
  ["score", "tolerance", "learn_rate", "measure", "input_control", "target_control"],
  ["neurons", "w", "w", "w", "w", "w"],
  ["w", "w", "w", "w", "w", "w"],
  ["w", "w", "w", "w", "w", "w"],
  ["w", "w", "w", "w", "w", "w"],
  ["w", "w", "w", "w", "w", "w"],
};

/* External functions and variables */

/* Declarations from ppymium.h */

extern char *rc [ ];

/* Declarations from pattern.c */

extern int init_patterns();
extern int count_patterns();
extern int read_pattern();
extern pat_elem *load_patterns();
extern pat_elem *indextop();
extern void add_to_pattern_list();
extern int read_input();
extern int read_target();
extern int read_xfile();
/* weight_update_class */
int Weight_update (p)
TAGVAL **p; /* error, *learn_rate, *SIZE, *weight, *state, ... */
{ int i, size;
  size = * (int *) p[2];
  PAR for (i = 3; i < size * size + 3; i++) {
    p[i] = value.f = (p[1] = value.f) * (p[0] = value.f) * (p[i + 1] = value.f);
  }
  return 0;
}
/* tolerance_class */
int Tolerance (p)
TAGVAL **p; /* *tolerance, *net.score, *net.measure, *SIZE, ... */
{ int size, control;
  float tol, *score;
  tol = (*p++) = value.f;
  score = *(*p++) = value.f;
  if (control = MERK) {  /* score = max.err_cal(p); if *score < tol { return (TERM); } */
  }
  else if (control = HAM) {  /* score = ham.dis.cal(p) / (float) size; if *score < tol { return (TERM); } */
    size = (*int *) p;
    score = *(*p++) = value.f;
    if *score < tol { return (TERM); }
  }
  else if (control = EUCL) {  /* score = eucl.dis.cal(p) / (float) size; if *score < tol { return (TERM); } */
    size = (*int *) p;
    score = *(*p++) = value.f;
    if *score < tol { return (TERM); }
  }
  else if (control = ANGL) {  /* error("Illegal tolerance test control [id]"), control; return (MOTOR); */
    error("Illegal tolerance test control [id]"), control;
    return (OK);
  }
  /* back propagation connect */
int connect (conf)
int conf[];
{ int cn, i, j, l, m, p, x, y, index;
  register int in_size;
  caddr_t *free_pointer1, *free_pointer2;
  rule_type *rp;
  para_type *sp;
  synapse_type *sp;
  sys = sys_alloc (conf); /* allocates memory for networks and neurons */
  /* allocate memory for the rules, meta rules and parameters */
  for (x = SYS: x < SYN: x++ ) {  /* y = ac (x) [0]; if (y ) */
    s = sizeof (rule_type);
    /* basic size to allocate */
    switch (x) {
      case SYS:
        sys->n_rules = y;
        sys->rules = (rule_type *) malloc (s * y);
        break;
      case NET:
        for (cn = 0; cn < sys->nets; cn++) {  /* y = sys->net[cn]->rules = y; */
          sys->net[cn]->rules = (rule_type *) malloc (s * y);
        }
        break;
      case LAY:
        for (cn = 0; cn < sys->nets; cn++) {  /* y = sys->net[cn]->rules = y; */
          rp = (rule_type *) malloc (s * y * (sys->net[cn]->layers - 1));
          for (i = 1; i < sys->net[cn]->layers; i++, rp = rp + y) {
            sys->net[cn]->layer[i] = (rule_type *) malloc (s * y);
            sys->net[cn]->layer[i]->rules = rp;
          }
        }
        break;
      case CLU:
        for (cn = 0; cn < sys->nets; cn++) {  /* y = sys->net[cn]->clusters = y; */
          rp = (rule_type *) malloc (s * y * (sys->net[cn]->layers - 1));
          for (j = 0; j < sys->net[cn]->layers; j++, rp = rp + y) {
            sys->net[cn]->layer[i] = (rule_type *) malloc (s * y);
            sys->net[cn]->layer[i]->clusters = j;
          }
        }
        break;
      case NED:
        for (cn = 0; cn < sys->nets; cn++) {  /* y = sys->net[cn]->clusters = y; */
          rp = (rule_type *) malloc (s * y * (sys->net[cn]->layers - 1));
          for (k = 0; k < sys->net[cn]->layers; k++, rp = rp+ y) {
            sys->net[cn]->layer[i] = (rule_type *) malloc (s * y);
            sys->net[cn]->layer[i]->clusters = j;
          }
        }
        break;
      default:
        break;
    }
  }
}
sys->net(cn)->fanin = lo_size;
sys->net(cn)->input_port = (caddr_t *)malloc(sizeof(caddr_t), lo_size);

input_vector = (float *) malloc(sizeof(float), sys->net(cn)->fanin);
target_vector = (float *) malloc(sizeof(float), sys->net(cn)->fanout);

/** build up the I-O pipe pointers for each layer ***/
free_pointer1 = sys->net(cn)->output_port;
free_pointer2 = sys->net(cn)->target;
for (i=0; i<sys->net(cn)->layers; i++) {
  for (j=0; j<sys->net(cn)->layer[i]->clusters; j++) {
    free_pointer1 = sys->net(cn)->layer[i]->clusters[j] - neurons; ++
    (caddr_t) sys->net(cn)->layer[i]->clusters[j] = state[STATE].value.f;
    free_pointer2 = sys->net(cn)->layer[i]->clusters[j] - neurons; ++
    (caddr_t) sys->net(cn)->layer[i]->clusters[j] = state[STATE].value.f;
  }
}

/** set up the number of neurons in each layer ***/
for (i=0; i<sys->net(cn)->layers; i++) {
  lo_size = 0;
  for (j=0; j<sys->net(cn)->layer[i]->clusters; j++) {
    lo_size += sys->net(cn)->layer[i]->clusters[j] - neurons;
  }
  sys->net(cn)->layer[i] = parameters; LAY_P_neurons.parameter.value.f += (float) lo_size;
}

/** allocate synapses for the neurons ***/
for (i=0; i<sys->net(cn)->layers; i++) {
  if (i != I_LAYER) {
    s = (int) sys->net(cn)->layer[i-1] = parameters; LAY_P_neurons.parameter.value.f;
  }
  for (j=0; j<sys->net(cn)->layer[i]->clusters; j++) {
    for (k=0; k<sys->net(cn)->layer[i]->clusters[j] - neurons; k++) {
      sys->net(cn)->layer[i]->clusters[j]->neuron[k] = synapse = 0;
      sys->net(cn)->layer[i]->clusters[j]->neuron[k] = synapse =
      (synapse_type *) malloc(sizeof(synapse_type), s);
      for (l=0; l<s; l++) {
        sys->net(cn)->layer[i]->clusters[j]->neuron[k] = synapse[l] = sp;
      }
    }
  }
}

/** write route[] data */
for (i=0; i<sys->net(cn)->layers; i++) {
  for (j=0; j<sys->net(cn)->layer[i-1] = clusters; j++) {
    for (k=0; k<sys->net(cn)->layer[i] = clusters[j] - neurons; k++) {
      sys->net(cn)->layer[i] = clusters[j] = neuron[k] = route[0] = cn;
    }
  }
}

/* end for cn ... */
return(OK);

}
/* build_rule.c */
/* */
/* */

int build_rules()
{
    int cn, i, j, k;

    if (sysNets == netNets && (!LAYER < 0))
        return (FAIL);

    for (cn = 0; cn < sysNets; cn++)
        for (i = 0; i < sysNet[cn].layers; i++)
            for (j = 0; j < sysNet[cn].layers[i].clusters; j++)
                for (k = 0; k < sysNet[cn].layers[i].clusters[j].neurons; k++)
                {
                    rule_init {
                        "neuron.state_upd",
                        sysNet[cn].layer[i].cluster[j].neuron[k].rules[NEUR_N_STATE_UPD],
                        sysNet[cn].layer[i].cluster[j].neuron[k].state_upd_class,
                        sysNet[cn].layer[i].cluster[j].neuron[k].fanin,
                        sysNet[cn].layer[i].cluster[j].neuron[k].state[NEUR_N_STATE],
                        sysNet[cn].layer[i].cluster[j].neuron[k].state[N_ACC],
                        EOP);
                    }

                    if (i == 0)
                    {
                        rule_init {
                            "neuron.err.cal",
                            sysNet[cn].layer[i].cluster[j].neuron[k].rules[NEUR_N_ERROR_CAL],
                            err_cal_output_class,
                            0,
                            sysNet[cn].layer[i].cluster[j].neuron[k].state[NEUR_N_ERR],
                            sysNet[cn].layer[i].cluster[j].neuron[k].state[NEUR_N_STATE],
                            EOP);
                        }
                    } else {
                        rule_init {
                            "neuron.err.cal",
                            sysNet[cn].layer[i].cluster[j].neuron[k].rules[NEUR_N_ERROR_CAL],
                            err_cal_hidden_class,
                            sysNet[cn].layer[i].cluster[j].neurons,
                            sysNet[cn].layer[i].cluster[j].neuron[k].state[NEUR_N_ERR],
                            sysNet[cn].layer[i].cluster[j].neuron[k].state[NEUR_N_STATE],
                            EOP);
                        }
                    }

                    rule_init {
                        "neuron.weight_upd",
                        sysNet[cn].layer[i].cluster[j].neuron[k].rules[NEUR_N_WEIGHT_UPD],
                        weight_update_class,
                        sysNet[cn].layer[i].cluster[j].neurons,
                        sysNet[cn].layer[i].cluster[j].neuron[k].state[NEUR_N_ERR],
                        sysNet[cn].layer[i].cluster[j].neuron[k].state[NEUR_N_STATE],
                        EOP);
                    }
                }

                /* end for k ... */
                /* end for j ... */
                /* end for i ... */
            }
        }
    }

    /* extending rules -- to fill in the parameter pointers */
    for (i = 0; i < LAYER; i++)
    {
        for (j = 0; j < sysNet[i].layers; j++)
            for (k = 0; k < sysNet[i].layers[j].clusters; k++)
            {
                rule_extend {
                    "sysNet[i].layer[j].cluster[k].rules[NEUR_N_STATE_UPD],
                    sysNet[i].layer[j].cluster[k].state_upd_class,
                    sysNet[i].layer[j].cluster[k].fanin,
                    sysNet[i].layer[j].cluster[k].state[NEUR_N_STATE],
                    sysNet[i].layer[j].cluster[k].state[N_ACC],
                    EOP);
                }
            }
    }

    /* end for j ... */
    /* end for i ... */
}

/* Initialize the rules at the cluster level */
for (i = 0; i < sysNet[cn].layers; i++)
    for (j = 0; j < sysNet[cn].layers[i].clusters; j++)
    {
        rule_init {
            "cluster.state_upd",
            sysNet[cn].layer[i].cluster[j].rules[CLU_N_STATE_UPD],
            state_upd_meta_class,
            sysNet[cn].layer[i].cluster[j].neurons,
            EOP);
        }

        rule_init {
            "cluster.err.cal",
            sysNet[cn].layer[i].cluster[j].rules[CLU_N_ERROR_CAL],
            err_cal_meta_class,
            sysNet[cn].layer[i].cluster[j].neurons,
            EOP);
        }
    }

    /* end for j ... */
    /* end for i ... */
rule_init {  
    "cluster_weight_upd",
    $sys->net(config)[layer[i]]->cluster[j]->_rules[ CLU_R_weight_upd ],
    $weight_update_meta_class,
    $sys->net(config)[layer[i]]->cluster[j]->_rules[ NEU_R_weight_upd ],
    EOP );
}

/*** Extending rules at the cluster level ***/
for (l=1: $sys->net(config)->layers: l++) {
for (j=0: $sys->net(config)->layers: j++) {
for (k=0: $sys->net(config)->clusters: k++) {

    rule_extend {
        $sys->net(config)[layer[i]]->cluster[j]->_rules[ CLU_R_state_upd ],
        k,
        $sys->net(config)[layer[i]]->cluster[j]->_rules[ NEU_R_state_upd ],
        EOP );
    }

    rule_extend {
        $sys->net(config)[layer[i]]->cluster[j]->_rules[ CLU_R_err_cal ],
        k,
        $sys->net(config)[layer[i]]->cluster[j]->_rules[ NEU_R_err_cal ],
        EOP );
    }

    rule_extend {
        $sys->net(config)[layer[i]]->cluster[j]->_rules[ CLU_R_weight_upd ],
        k,
        $sys->net(config)[layer[i]]->cluster[j]->_rules[ NEU_R_weight_upd ],
        EOP );
    }
}
}

/*** Initialize the rules at the net level ***/
/*** Initialize the recall rule ***/
rule_init {  
    "layer.state_upd",
    $sys->net(config)[layer[i]]->cluster[j]->_rules[ LAY_R_state_upd ],
    $state_upd_meta_class,
    $sys->net(config)[layer[i]]->clusters,
    EOP );

    rule_init {  
    "layer.err_cal",
    $sys->net(config)[layer[i]]->rules[ LAY_R_err_cal ],
    $err_cal_meta_class,
    $sys->net(config)[layer[i]->_clusters,
    EOP );

    rule_init {  
    "layer.weight_upd",
    $sys->net(config)[layer[i]->_rules[ LAY_R_weight_upd ],
    $weight_update_meta_class,
    $sys->net(config)[layer[i]->_clusters,
    EOP );
}

    for (l=1: $sys->net(config)->layers: l++) {
    for (j=0: $sys->net(config)->layers: j++) {
        rule_extend {
            $sys->net(config)[layer[i]->_rules[ LAY_R_state_upd ],
            j,
            $sys->net(config)[layer[i]->_clusters[j]->_rules[ CLU_R_state_upd ],
            EOP );
        }

        rule_extend {
            $sys->net(config)[layer[i]->_rules[ LAY_R_err_cal ],
            j,
            $sys->net(config)[layer[i]->_clusters[j]->_rules[ CLU_R_err_cal ],
            EOP );
        }

        rule_extend {
            $sys->net(config)[layer[i]->_rules[ LAY_R_weight_upd ],
            j,
            $sys->net(config)[layer[i]->_clusters[j]->_rules[ CLU_R_weight_upd ],
            EOP );
        }
}

    for (l=1: $sys->net(config)->layers: l++) {
    for (j=0: $sys->net(config)->layers: j++) {
        rule_init {  
        "net_recall",
        /* recall() takes layers-1 steps to */
        $sys->net(config)->rules[ NET_R_recall ], /* update the states of all layers */
        $recallMeta_class,
        $sys->net(config)->layers-1,
        EOP );

        rule_init {  
        "net.tol.test",
        /* control switch stored NET_T_measure */
        $sys->net(config)->rules[ NET_T_tol_test ], /* result placed in NET_T_score */
        $tol_test_class,
        $sys->net(config)->fanout,
        $sys->net(config)->parameters[ NET_T_tolerance ]._param,
        $sys->net(config)->parameters[ NET_T_score ]._param,
        $sys->net(config)->parameters[ NET_T_measure ]._param,
        EOP );

        rule_init {  
        "net.weight_upd",
        $sys->net(config)->rules[ NET_R_weight_upd ],
        $weight_update_meta_class,
        $sys->net(config)->layers - 1,
        /* except the input layer */
        EOP );

    } /* initialize the weight update meta_rule at net level */
}
/** Initialize the learn rule ***/
rule_init {
    "net.learn",
    <sys->net(cn)->rules[ NET_R_learn ],
    <learn_meta_class,
    2 * <sys->net(cn)->layers,
    EOP );
}

/** Initialize the step_learn rule ***/
rule_init {
    "net.step.learn",
    <sys->net(cn)->rules[ NET_R_step_learn ],
    <step_learn_meta_class,
    2 * <sys->net(cn)->layers,
    EOP );
}

/** Extend the recall rule ***/
for (i=1; i<sys->net(cn)->layers; i++) {
    rule_extend {
        <sys->net(cn)->rules[ NET_R_recall ],
        i-1,
        <sys->net(cn)->layer[i]->rules[ LAY_R_state_upd ],
        EOP );
    }

/** Extend the tolerance rule ***/
for (i=0, k=0; i<sys->net(cn)->layer[O_LAYER]->clusters; i++) {
    for (j=0; j<sys->net(cn)->layer[O_LAYER]->cluster[i]->neurons; j++) {
        rule_extend {
            <sys->net(cn)->rules[ NET_R_tol_test ],
            k++,
            <sys->net(cn)->layer[O_LAYER]->cluster[i]->neuron[j]->state[ N_TARGET ],
            <sys->net(cn)->layer[O_LAYER]->cluster[i]->neuron[j]->state[ N_STATE ],
            EOP );
    }
}

/** Extend the weight update rule at the net level */
for (i=1; i<sys->net(cn)->layers; i++) {
    rule_extend {
        <sys->net(cn)->rules[ NET_R_weight_upd ],
        i-1,
        <sys->net(cn)->layer[i]->rules[ LAY_R_weight_upd ],
        EOP );
    }

/** Extend the rule learn ***/
k=0:
rule_extend {
    <sys->net(cn)->rules[ NET_R_learn ],
    k++,
    <sys->net(cn)->rules[ NET_R_recall ], /* recall to evaluate the network output */
    EOP );
}

rule_extend {
    <sys->net(cn)->rules[ NET_R_learn ],
    k++,
    <sys->net(cn)->rules[ NET_R_tol_test ], /* calculate the maximum error of the output unit */
    EOP );
}

for (i=sys->net(cn)->layers-1; i>0; i--)
    rule_extend {
        <sys->net(cn)->rules[ NET_R_learn ],
        k++,
        <sys->net(cn)->layer[i]->rules[ LAY_R_err_cal ], /* calculate the errors */
        EOP );
    }

rule_extend {
    <sys->net(cn)->rules[ NET_R_learn ],
    k++,
    <sys->net(cn)->rules[ NET_R_weight_upd ],
    EOP );
}

/** Extend the rule step_learn ***/
 /*-----------------------------------------------*/
/** Extend the rule learn ***/
k=0:
rule_extend {
    <sys->net(cn)->rules[ NET_R_step_learn ],
    k++,
    <sys->net(cn)->rules[ NET_R_recall ], /* recall to evaluate the network output */
    EOP );
}

rule_extend {
    <sys->net(cn)->rules[ NET_R_step_learn ],
    k++,
    <sys->net(cn)->rules[ NET_R_tol_test ], /* calculate the maximum error of the output unit */
    EOP );
}

for (i=sys->net(cn)->layers-1; i>0; i--)
    rule_extend {
        <sys->net(cn)->rules[ NET_R_step_learn ],
        k++,
        <sys->net(cn)->layer[i]->rules[ LAY_R_err_cal ], /* calculate the errors */
        EOP );
    }

rule_extend {
    <sys->net(cn)->rules[ NET_R_step_learn ],
    k++,
    <sys->net(cn)->rules[ NET_R_weight_upd ],
    EOP );
}
/* end for cn ... */
return { OK );
}
ifndef OMIT_MAIN
main (argc, argv)
int argc;
char *argv[];
{

int 1, somefail, cycle;
int wh[4];
float result;
patt installing *p:
int load_file:

random(0x654790c); /* Initialize srand */

/* Construct the network */
if ( rc_read() )
{
printf ("Problem with .pgmrc file\n"):
exit (1);
}

printf ("local_user \%s\", rc [ RC_local_user ]);
printf ("local_host \%s\", rc [ RC_local_host ]);

if ( argc == 1 )
{
if (connect(config)) /* use internal configuration defaults */
exit(FAIL);
if (build_rules())
exit(FAIL);
}

else {
strcpy( system_filename, argv(1) );
/* load system file - if configuration not specified, */
connect() and build_rules() must be called here */
if ( ! (load_file = sys_load( system_filename )) & LO_CONFIG ) { /*
if (connect(config)) /* use internal configuration defaults */
exit(FAIL);
if (build_rules())
exit(FAIL);
}

/* Set pattern controls to loaded parameters. This is */
/* necessary because for this algorithm, pattern types other */
/* than BINARY may be specified - and the default may be */
/* altered by editing the system file */

input_pattern_control = (int)sys-net[c_net]-parameters[NET_P_input_control].parameter.value.f;
target_pattern_control = (int)sys-net[c_net]-parameters[NET_P_target_control].parameter.value.f;
/* Initialize patterns specified on command line */

if ( argc > 2 )
{
  i = init_patterns ( argv [ 2 ]);
  printf ("init_patterns returns \%d from \%s\", i, argv [ 2 ]);
}

if ( pattern_list == NULL ) /* new system - randomize the weights */
rand_weight();

while ( TRUE ) |

  cycle++;

  for ( i = 0; i < pattern_count; i++ )
  {
    cycle = 0;

    current_pattern = indextop ( i ); /* to gain access to pattern details */
    read_input ( i );
    read_target ( i );

    result = learn();
    if (current_pattern->input_pattern_type == PATTERN_TYPE_FONT )
      printf ("pattern [ %s ]\n\n", current_pattern->input_file,
        (char)current_pattern->input_index, result );
    else
      printf ("pattern [ %s ]\n\n", current_pattern->input_file,
        current_pattern->input_index, result );
  }

  somefail = FALSE ;
  for ( i = 0; i < pattern_count; i++ )
  {
    current_pattern = indextop ( i );
    read_input ( i );
    read_target ( i );

    result = recall (i);
    current_pattern->score = result;

    if (current_pattern->input_pattern_type == PATTERN_TYPE_FONT )
      printf ("cycle [ %d ] pattern [ %s ]\n\n", cycle, current_pattern->input_file,
        (char)current_pattern->input_index, result );
    else
      printf ("cycle [ %d ] pattern [ %s ]\n\n", cycle, current_pattern->input_file,
        current_pattern->input_index, result );
  }

  if ( result > sys->net[c_net]-parameters[NET_P_tolerance].parameter.value.f )
    somefail = TRUE ;

  if ( somefail == FALSE )
    break;

  printf ("Now least one pattern fails tolerance test - re-learning\n");
  sys-save ( system_filename ); /* may interrupt - last cycle is saved */

  if ( argc > 3 )
    /* may specify second test set in argv[3] */
  
  printf ("init_patterns returns \%d from \%s\", i, argv [ 3 ]);

  for ( i = 0; i < pattern_count; i++ )
    current_pattern = indextop ( i );
  
read_input ( i );
read_target ( i ); // only necessary to get pattern width for shownet */
recall();
shownet();
}

if (system_filename[0] == '\0') {
    sys_save ( "test" );
} else {
    sys_save ( system_filename );
}
rc_write();

#endif
/* ---------------------------------------------------------------------- */
/* --------------------------------------------------------------------- */

#include <stdio.h>

/* Defaults for back propagation algorithm */

/* parameter values */

#define TOLERANCE 0.25
#define LEARN_RATE 0.30
#define MEASURE MERR /* Options : MERR, HAM, EUCL, ANGL */
#define INPUT_PATTERN_CONTROL BINARY /* Options : BINARY, REAL */
#define TARGET_PATTERN_CONTROL BINARY /* Options : BINARY, REAL, SCALED_SIGMOID */
#define INPUT_SCALE 1.0
#define TARGET_SCALE 1.0
#define MOMENTUM 0.5

/* topology */

#define NETS 1
#define LAYERS 3
#define CLUSTERS 1

struct config_tag {
  int nets;
  struct {
    int layers;
    struct {
      int clusters;
      struct {
        int neurons;
      } cluster [CLUSTERS];
    } layer [LAYERS];
  } net [NETS];
} system_config =
{ 1,
  { 3,
    { 1,
      { 8*12 }
    },
    { 1,
      { 24 }
    },
    { 1,
      { 8*12 }
    }
  }
};

/* --------------------------------------------------------------------- */
/* ------------------------------- */

sysdef.h

/* ------------------------------- */

#include <stdio.h>
#include <sys/types.h>

/* Control key definition */
#define PAR
#define SEQ

/* Constant definitions */
#define NO_STATES 4
#define BOP NULL
#define CHILD 0
#define NULLCP (char *) NULL
#define CONFIRM_OK 0
#define SYNT_ERR 1
#define QUERY 2
#define SYNTAXY 3
#define EXIT 60
#define FAIL -1

/* Exit control for rules */
#define TERM -1

/* neuron.state indices */
#define N_STATE 0
#define N_ERR 1
#define N_TARGET 2
#define N_ACC 3
#define N_WEIGHT 4
#define N_RULE 5
#define N_PARA 6
#define N_NOMES 7

/* control values for network tolerance test */
#define MERR 0
#define HAM 1
#define EUCL 2
#define ANGL 3

/* control values for neighbourhood calculation */
#define EUCLIDEAN 0
#define CARTESIAN 1
#define CITY_BLOCK 2

/* Pattern value to clamp to input, target vectors */
#define BINARY 0
#define PLUSMINUS 1
#define REAL 2

#define SCALED_SIGMOID 3
#define NORMALIS 4

/* activation function switch */
#define STEP 0
#define SIGMOID 1
#define PATTERN_TYPE_FONT 0
#define PATTERN_TYPE_X 1
#define PATTERN_TYPE_FLOAT 2
#define PATTERN_TYPE_INT 3

/* system load return flags */
#define LD NOTHING 0
#define LD_CONFIG 1<0
#define LD_PARAMETERS 1<1
#define LD_STATES 1<2
#define LD_WEIGHTS 1<3
#define LD_PATTERNS 1<4
#define LD_CYCLES 1<5

/* Macro function definitions */
#define NO() (YES())
#define clearscren() putchar(CLEAR)
#define SYS 0
#define MST 1
#define LAY 2
#define CLU 3
#define NEU 4
#define SW 5

/* Rule structures */
typedef struct {
    int (*fn)(); /* pointer to function returning an integer */
    char *fn_name; /* the name of the function in the code - future use */
    int n generic parameters;
    int n extend parameters;
} class_type;

#define null_class_type (class_type *) 0

/* Rule structures */
struct RULE {
    char *tag; /* code gen() - future use */
    char *name; /* for display purposes */
    class type *class;
    caddr_t *para list;
} ;

typedef struct RULE rule type;
#define null_rule type (rule type *) 0
typedef struct
{
    int n_rules;
    rule_type *rules;
    int n_parameters;
    para_type *parameters;
    int nets;
    net_type **net;
} system_type;

#define null_system_type (system_type *) 0

/*
   various structures for patterns etc.
*/
struct elem_header{
    struct elem_header* next_ptr;
};
typedef struct elem_header any_elem;

struct pat_chain {
    struct pat_chain *next_ptr;
    int input_pattern_type;
    char *input_file;
    int input_index; /* index to pattern */
    int input_width;
    int input_height;
    int target_pattern_type;
    char *target_file;
    int target_index; /* index to pattern */
    int target_width;
    int target_height;
    float score;
};
typedef struct pat_chain pat_elem;

struct P_CHAIN {
    struct P_CHAIN *next_ptr;
    int index; /* pattern number */
    int count; /* number of values */
    long offset; /* file offset */
};
typedef struct P_CHAIN p_chain;

#define null_p_chain (p_chain *) 0

struct FILE_CHAIN {
    struct FILE_CHAIN *next_ptr;
    char *filename;
    int count; /* number of patterns in file */
    p_chain *pchain; /* start of p_chain list */
};
typedef struct FILE_CHAIN file_chain;

#define null_file_chain (file_chain *) 0

struct int_chain {
    struct int_chain *next_ptr;
    int val;
};
typedef struct int_chain int_elem;

}*/