

High Level Synthesis of Neural Network Chips

Meyer Elias Nigri

a thesis submitted for the degree of

Doctor of Philosophy in Computer Science

University of London

Department of Computer Science

University College London

February 1993

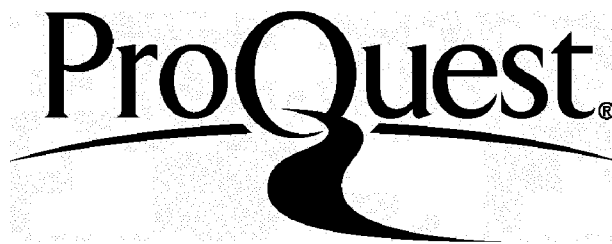
ProQuest Number: 10044300

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10044300

Published by ProQuest LLC(2016). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code.
Microform Edition © ProQuest LLC.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Abstract

This thesis investigates the development of a silicon compiler dedicated to generate Application-Specific Neural Network Chips (ASNNCs) from a high level C-based behavioural specification language. The aim is to fully integrate the silicon compiler with the ESPRIT II *Pygmalion* neural programming environment. The integration of these two tools permits the translation of a neural network application specified in *nC*, the *Pygmalion*'s C-based neural programming language, into either binary (for simulation) or silicon (for execution in hardware). Several applications benefit from this approach, in particular the ones that require real-time execution, for which a true neural computer is required.

This research comprises two major parts: extension of the *Pygmalion* neural programming environment, to support automatic generation of neural network chips from the *nC* specification language; and implementation of the high level synthesis part of the neural silicon compiler.

The extension of the neural programming environment has been developed to adapt the *nC* language to hardware constraints, and to provide the environment with a simulation tool to test in advance the performance of the neural chips. Firstly, new hardware-specific requisites have been incorporated to *nC*. However, special attention has been taken to avoid transforming *nC* into a hardware-oriented language, since the system assumes minimum (or even no) knowledge of VLSI design from the application developer. Secondly, a simulator for neural network hardware has been developed, which assesses how well the generated circuit will perform the neural computation. Lastly, a hardware library of neural network models associated with a target VLSI architecture has been built.

The development of the neural silicon compiler focuses on the high level synthesis part of the process. The goal of the silicon compiler is to take *nC* as the input language and automatically translate it into one or more identical integrated circuits, which are specified in VHDL (the IEEE standard hardware description language) at the register transfer level. The development of the high level synthesis comprises four major parts: firstly, compilation and software-like optimisations of *nC*; secondly, transformation of the compiled code into a graph-based internal representation, which has been designed to be the basis for the hardware synthesis; thirdly, further transformations and hardware-like optimisations on the internal representation; and finally, creation of the neural chip's data path and control unit that implement the behaviour specified in *nC*.

Special attention has been devoted to the creation of optimised hardware structures for the ASNNCs employing both phases of neural computing on-chip: recall and learning. This is achieved through the data path and control synthesis algorithms, which adopt a heuristic approach that targets the generated hardware structure of the neural chip in a specific VLSI architecture, namely the *Generic Neuron*.

The viability, concerning the effective use of silicon area versus speed, has been evaluated through the automatic generation of a VHDL description for the neural chip employing the Back Propagation neural network model. This description is compared with the one created manually by a hardware designer.

To my adorable wife,

Deborah

Acknowledgements

The completion of this work owes much to many people. Firstly, I would like to express my indebtedness to my supervisor Prof. Philip Treleaven who assisted, advised and gave me tremendous support throughout this research. His continuous encouragement made the conclusion of this thesis possible.

Secondly, I would like to thank the Brazilian institution CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) for the financial support.

Special thanks must go to Marley Vellasco for her precious friendship, her patience while reading the earlier draft of the thesis, and for providing invaluable discussions throughout this research.

I am also grateful to Dr. Peter Rounce and Annop Mangat for their patience and constructive criticism while reading the earlier draft of the thesis, and to Michael Recce for his extremely important incentive during the writing up phase of this thesis.

I am thankful to all my colleagues in the Department of Computer Science at UCL for their friendship, in particular to Cesare Alippi, Carlo de Oliveira, Marco Pacheco, Zé Luiz Ribeiro, and Paulo Rocha. I am also grateful for the support received from my colleagues at the *GALATEA* group.

I also thank my dear friends Pedro Vellasco and Iclea Ortiz for their friendship, concern and support throughout these years.

Most of all I wish to thank my wife, Debinha, for her inestimable support and motivation. Without her patience, confidence and love this work would have never been thinkable.

Last but not least, I want to thank my parents Esther and Elias and my sisters, Tuna and Sonia, for their immeasurable love and strength during all these years.

Table of Contents

Abstract	3
Acknowledgements	7
Chapter 1 — Introduction	17
1.1. Neural Computing.....	17
1.1.1. Artificial Neural Networks	18
1.1.2. Applications of Neural Networks.....	21
1.1.3. Neural Network Implementations	22
1.2. Silicon Compilation.....	24
1.3. Motivations and Thesis Goals.....	25
1.4. System Overview	27
1.5. Thesis Contributions	29
1.6. Thesis Organisation.....	30
Chapter 2 — Neural Computing Systems	33
2.1. Overview	33
2.2. Neural Programming Environments.....	34
2.3. Neurocomputers	37
2.3.1. Electronic Neural Chips.....	38
2.3.2. Optical Neurocomputers	43
2.4. Integrating Software and Hardware Tools	44
2.5. Summary.....	45
Chapter 3 — Silicon Compilers	47
3.1. Introduction	47
3.2. Characteristics of Silicon Compilers	49
3.2.1. Input Description	50
3.2.2. Internal Representation.....	52
3.2.3. Target Architecture	53
3.2.4. High Level Synthesis	54
3.2.5. Logic Synthesis	54
3.2.6. Layout Synthesis	55
3.3. Anatomy of High Level Synthesis.....	55
3.3.1. High Level Transformations	56
3.3.2. Data Path Synthesis.....	57

3.3.3. Control Synthesis	61
3.4. Taxonomy of Silicon Compilers.....	61
3.4.1. Olympus System.....	62
3.4.2. Yorktown System.....	63
3.4.3. HIS System	64
3.4.4. Cathedral System.....	64
3.4.5. Galatea System.....	65
3.4.6. UCL's Neural Silicon Compiler	66
3.5. Summary.....	67
Chapter 4 — <i>nC</i> Neural Network Specification Language	69
4.1. <i>Pygmalion</i> Programming Environment	69
4.2. <i>nC</i> Network Specification Language.....	72
4.2.1. Basic Concepts.....	72
4.2.2. <i>nC</i> Application Definition	77
4.2.3. <i>nC</i> Algorithm Definition	78
4.3. <i>nC</i> Extensions	79
4.3.1. <i>nC</i> Strengths and Weaknesses.....	79
4.3.2. Time and Area Constraints	81
4.3.3. Fixed-Point Precision Calculation	82
4.3.4. Realisation of the Activation Function	83
4.4. Summary.....	84
Chapter 5 — Simulation of Neural Networks Hardware	85
5.1. Overview.....	85
5.2. Back Propagation Computation.....	86
5.3. Hardware Constraints.....	87
5.3.1. Multiplication	87
5.3.2. Summation	88
5.3.3. Activation Function	89
5.3.4. Table's Indexing Mechanism.....	89
5.4. Back Propagation and Precision Requirements	90
5.4.1. Theoretical Analysis	90
5.5. Hardware Simulation.....	93
5.5.1. Hardware Library	93
5.5.2. User Configuration Parameters	94
5.5.3. Simulation Results	94
5.6. Summary.....	98
Chapter 6 — <i>NSC</i> and Target Architecture	99
6.1. Overview.....	99

6.2. High Level Synthesis	100
6.3. <i>Generic Neuron</i> Architecture	102
6.4. Processing Element Organisation.....	106
6.4.1. Communication Unit	106
6.4.2. Memory Unit.....	107
6.4.3. Execution Unit	108
6.5. VHDL Implementation of a Back Propagation Neural Chip.....	109
6.5.1. Communication Unit	111
6.5.2. Memory Unit.....	115
6.5.3. Execution Unit	117
6.6. Processing Element	119
6.7. Back Propagation Neural Chip	120
6.8. Summary.....	121
Chapter 7 — NSC Implementation	123
7.1. Overview	123
7.2. <i>nC</i> Compilation and Transformation Steps	125
7.2.1. <i>nC</i> Parsing	125
7.2.2. <i>nC</i> Data Structure Transformations	127
7.2.3. <i>nC</i> Syntax Tree Transformations	128
7.3. Intermediate Code Representation.....	131
7.4. <i>ICR</i> Transformations.....	133
7.4.1. Graph Partitioning	134
7.4.2. Constant Propagation and Storage Elimination	134
7.5. Target Architecture.....	136
7.6. Data Path Synthesis.....	138
7.6.1. Allocation of Storage Elements	140
7.6.2. Allocation of Operators	142
7.6.3. Allocation of Interconnections.....	143
7.6.4. Scheduling	144
7.7. Module Generation	146
7.8. Control Synthesis	148
7.9. VHDL	151
7.10. Neural Network Partition	152
7.11. Summary.....	152
Chapter 8 — From <i>nC</i> to VHDL Neural Chips	155
8.1. Overview	155
8.2. <i>nC</i> Description of a Back Propagation Network.....	155
8.2.1. Application Definition	155

8.2.2. Algorithm Definition.....	156
8.3. <i>nC</i> Compilation and Syntax Tree Transformations	159
8.4. Hardware Synthesis.....	160
8.4.1. Graph Generation and <i>ICR</i> Transformations	160
8.4.2. Data Path Synthesis	165
8.4.3. Control Synthesis	169
8.5. VHDL Description	170
8.6. Summary	170
Chapter 9 — Assessment	173
9.1. Software and Hardware Integration.....	173
9.2. <i>Pygmalion</i> Extensions	174
9.2.1. <i>nC</i> Extensions	175
9.2.2. Hardware-Specific Algorithm Library	176
9.2.3. Simulation of Neural Networks Hardware	177
9.2.4. Summary	177
9.3. Neural Silicon Compiler	178
9.3.1. <i>nC</i> High Level Transformations	179
9.3.2. Intermediate Code Representation	180
9.3.3. Target Architecture	180
9.3.4. Data Path Synthesis	181
9.3.5. Control Synthesis	182
9.3.6. VHDL Description of Neural Chips	182
Chapter 10 — Conclusions and Future Work	183
10.1. Summary	183
10.2. Research Contributions.....	184
10.3. Future Work.....	186
References	189
Appendix A — Published Work	203
Appendix B — <i>nC</i> Language Syntax and Example	205
Appendix C — Hardware Simulation Results	221
Appendix D — VHDL Description of Processing Elements	229
Appendix E — List of Abbreviations	247

List of Figures

Figure 1.1 — Multi-layer Neural Network	18
Figure 1.2 — (a) Artificial Neuron; (b) Sigmoidal Activation Function.....	20
Figure 2.1 — Current Design Cycle of Neural Network Applications	35
Figure 2.2 — Basic Analogue Artificial Neuron	39
Figure 2.3 — General Schemes for Optical Implementation of Neural Networks.....	44
Figure 2.4 — Proposal for Integrating Software and Hardware Tools	45
Figure 3.1 — Example of the Design Hierarchy in VHDL	50
Figure 3.2 — Examples of Control and Data Flow Graphs.....	53
Figure 3.3 — ASAP Schedule, ALAP Schedule, and Mobility.....	58
Figure 4.1 — <i>Pygmalion</i> Neural Programming Environment.....	70
Figure 4.2 — The <i>nC</i> Hierarchical Data Structure.....	73
Figure 4.3 — The <i>nC</i> system Data Structure.....	74
Figure 4.4 — The General RULE Data Structure	74
Figure 4.5 — Description of the Neuron's State Update rule level	75
Figure 4.6 — Description of the Cluster's State Update meta-rule level	75
Figure 4.7 — The TAGVAL and UNVAL Data Structures	76
Figure 4.8 — The config Structure	78
Figure 4.9 — The Extended RULE Structure.....	81
Figure 4.10 — The Extended neuron_type Structure	82
Figure 4.11 — The Extended TAGVAL Structure.....	83
Figure 5.1 — Methods of Employing Truncation After a Multiplication Operation.....	88
Figure 5.2 — The Effect of Overflows and Underflows.....	89
Figure 5.3 — The Lookup Table Indexing Mechanism	90
Figure 5.4 — Mechanism for Converting a Value into the Table's Index	90
Figure 5.5 — The Character Recognition Application	95
Figure 5.6 — The Effect of the Lookup Table.....	96
Figure 5.7 — Impact of Hardware Constraints During the Learning Phase	97
Figure 6.1 — The Hardware Development Extensions to the <i>Pygmalion</i> Environment.	100
Figure 6.2 — A Generic Neural Silicon Compiler.....	101
Figure 6.3 — The <i>Generic Neuron</i> Model.....	104
Figure 6.4 — Processing Elements Interconnection: (a) Single Bus; (b) Multi-busses...	105
Figure 6.5 — Processing Element's Internal Organisation	106

Figure 6.6 — Communication Unit's Internal Structure.....	107
Figure 6.7 — Memory Unit's Internal Organisation.....	108
Figure 6.8 — Execution Unit's Internal Structure.....	109
Figure 6.9 — VHDL Implementation of the PE's Communication Unit.....	111
Figure 6.10 — VHDL Implementation of the Communication Unit's Data Path	112
Figure 6.11 — Specification of the PE's my_address Register.....	113
Figure 6.12 — VHDL Implementation of the Communication Unit's Control Module..	114
Figure 6.13 — VHDL Implementation of the Processing Element's Memory Unit.....	115
Figure 6.14 — VHDL Implementation of the Memory Unit's Storage Module	116
Figure 6.15 — VHDL Implementation of the Memory Unit's Addressing Module.....	117
Figure 6.16 — VHDL Implementation of the Processing Element's Execution Unit	117
Figure 6.17 — VHDL Implementation of the Execution Unit's Data Path Module	118
Figure 6.18 — VHDL Implementation of the Execution Unit's Control Module.....	119
Figure 6.19 — VHDL Implementation of a Hidden Layer Processing Element	120
Figure 6.20 — VHDL Implementation of a Simple Back Propagation Chip	121
Figure 7.1 — <i>nC</i> Compilation at the Neuron Level.....	126
Figure 7.2 — Example of a <i>nC</i> Syntax Tree Structure for an Assignment Operation.....	127
Figure 7.3 — Transforming Memory References into Registers	130
Figure 7.4 — The <i>ICR</i> Syntax Definition.....	133
Figure 7.5 — Example of Graph Partitioning.....	134
Figure 7.6 — Graphical View of Variables Lifetime	135
Figure 7.7 — A Generic Multi-Port Template for Registers.....	146
Figure 7.8 — The Counter Module	147
Figure 7.9 — ALU Module for Implementing Typical Neural Network Computation ...	148
Figure 7.10 — A Finite State Machine Template in VHDL.....	150
Figure 8.1 — The config structure for the OCR Application.....	156
Figure 8.2 — State_Update Rule Definition.....	156
Figure 8.3 — Err_cal_hidden Rule Definition.....	157
Figure 8.4 — Err_cal_output Rule Definition	157
Figure 8.5 — Weight_Update (Forward) Rule Definition for the Output Layer	158
Figure 8.6 — Weight_Update (Backward) Rule Definition.....	158
Figure 8.7 — <i>ICR</i> Format for a Hidden Layer Neuron.....	161
Figure 8.8 — The CDFG for the Hidden Layer's Rules	162
Figure 8.9 — Lifetime Analysis for Variables in the Back Propagation Example	163
Figure 8.11 — Synthesised Data Path for the Execution and Memory Units	168
Figure 8.12 — Generated List of Actions for the FSM	169

List of Tables

Table 2.1 — Neural Network Programming Environments.....	36
Table 3.1 — The Hardware Design Hierarchy.....	47
Table 3.2 — Simple Taxonomy of High Level Synthesis Systems.....	61
Table 5.1 — Theoretical Results for the Sigmoid Function.....	92
Table 5.2 — 32-bit Floating-Point Simulation Results.....	98
Table 6.1 — Characteristics of Some Popular Neural Network Models.....	103
Table 6.2 — PE Functionality for Hidden and Output Layers.....	110
Table 7.1 — PE Functionality for Hidden and Output Layers.....	138
Table 8.1 — Transforming Generic and Extended Parameters.....	159
Table 8.2 — Activity List for Some Storage Elements	163
Table 8.3 — Allocation and Binding of Registers in the Data Path	166
Table C.1 — Data Precision Influence without Employing Lookup Table.....	222
Table C.2 — Data Precision Influence Employing Lookup Table	223
Table C.3 — Impact Caused by the Employment of the Lookup Table (12 bits data)....	224
Table C.4 — Impact Caused by the Employment of the Lookup Table (14 bits data)....	224
Table C.5 — Data Precision Influence Employing Lookup Table (16 bits Data)	224
Table C.6 — Data Precision Influence Employing Lookup Table (with μ term).....	225
Table C.7 — Simple vs. Complex Back Propagation Computation.....	226
Table C.8 — Effects of the Back Propagation Parameters.....	227
Table C.9 — Effects of the Back Propagation Parameters.....	227

Chapter 1

Introduction

This chapter presents a brief introduction to the neural computing and the silicon compilation areas. It then discusses the motivations and goals of this research, gives an overview of the neural silicon compiler system developed, and concludes with an outline of the thesis contributions and thesis organisation.

1.1. Neural Computing

Neural computing has developed over the last decade as an important paradigm to solve pattern processing problems, such as speech and image processing, that are particularly difficult for traditional computing [44, 76, 82]. Unlike conventional computing, which requires the development of a set of rules and algorithms for solving problems, neural computing attempts to solve these problems by presenting a set of inputs and training the system to output the appropriate responses [199].

Neural computing is the computation of artificial neural networks, which are modelled upon the knowledge of how the human brain is structured. Although the precise knowledge of how the human brain computes is still very limited, its anatomy and physiology have been identified in detail over the last decades [167]. Inspired by these studies, several artificial neural network models have been devised [39, 79, 82, 86, 99, 106, 122, 161, 202] and shown their ability to exhibit properties analogous to the brain: association, generalisation, parallel search, learning, and flexibility.

As it is well known, neural computing is an interdisciplinary subject, which has attracted researchers from different areas. As a result, a large number of books have been published providing comprehensive introductions to every aspect of the field: theory, software, hardware, and applications [12, 76, 78, 82, 122, 199]. However, for the purpose of completeness, the next sub-sections provide a brief review of this area. The discussion focuses on the hardware implementation aspects, since this is the main topic of study in this research.

1.1.1. Artificial Neural Networks

Artificial neural networks are massively parallel systems that combine a network of highly interconnected processing elements (PEs), or *artificial neurons*, that use mathematical algorithms, or *neural network models*, to process the information [44]. These PEs are simplified versions of the biological counterpart and carry out very simple calculations. Each interconnection between PEs have an associated weight value. The process of programming a neural network is achieved by a *learning algorithm* that adjusts the values of the weights in response to training patterns.

Artificial neural networks can be characterised by four key properties:

- network topology;
- recall procedure;
- learning procedure; and
- input values.

Network topology, the neurons' interconnection pattern, is perhaps the most distinguishing characteristic of neural models, and the most difficult issue when hardware implementation is sought. Typically, PEs are arranged into disjoint structures called *layers*. Each PE in a layer has usually the same activation function and learning rule. Early models, such as the Perceptron [122], are single layer *feed-forward* networks. In this topology each network input is connected to all PEs, and the information flows through the network from input to output, without feedback of outputs.

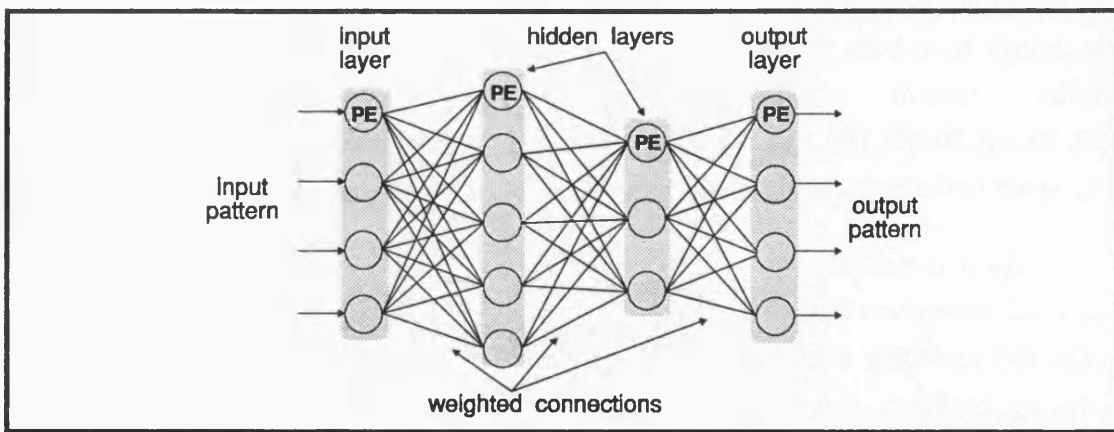


Figure 1.1 — Multi-layer Neural Network

More recent models have extended the concept of a single layer network into a structure of multiple feed-forward layers. Multi-layer networks are composed of one input layer, one output layer, and one or more internal (*hidden*) layers (see Figure 1.1). A

further extension includes the introduction of *backward* connections, which aims to improve the learning procedure by feeding backward *error* values between two consecutive layers [161].

An appropriate interconnection strategy for data communication among neurons is fundamental for an effective implementation of neural networks in hardware. In digital electronics technology, for instance, point-to-point interconnection among every neuron is clearly impossible due to limitation of the silicon area and number of pins. Thus, broadcast bus architectures are commonly used as the most flexible approach [180].

Recall procedure is realised by every PE in the network. Each PE is primitive, meaning that it can be an analogue neuron or a simple reduced instruction set (RISC) microprocessor. The recall procedure computes the neuron's state S . It is specified by the propagation rule net , which is typically the sum of weighted input values ($s.w$) modified by an offset θ that defines the neuron bias (see Figure 1.2a), followed by an activation function f :

$$net = \sum s.w - \theta \quad \text{and} \quad S = f(net)$$

The activation function may be implemented by: a simple linear function,

$$S = K.net$$

where K is a constant; a threshold function,

$$S = \begin{cases} 1 & \text{if } net > T \\ 0 & \text{otherwise} \end{cases}$$

where T is a constant threshold value; or a function that more accurately simulates the non-linear transfer characteristic of the biological neuron and permits more general network functions. The sigmoid function (see Figure 1.2b),

$$S = \frac{1}{1 + e^{-net}}$$

is one of the non-linear function most used by popular neural models. Another non-linear function, the hyperbolic tangent function,

$$S = \tanh(net)$$

is often used by biologists as a mathematical model of nerve-cell activation [199].

Hardware implementation of the propagation rule can be easily achieved by both analogue and digital circuits. However, the implementation of some complex activation functions, such as the sigmoid and hyperbolic tangent, is very expensive using digital technology. In those cases, it is common to overcome this difficulty using look-up tables to implement the desired function [130].

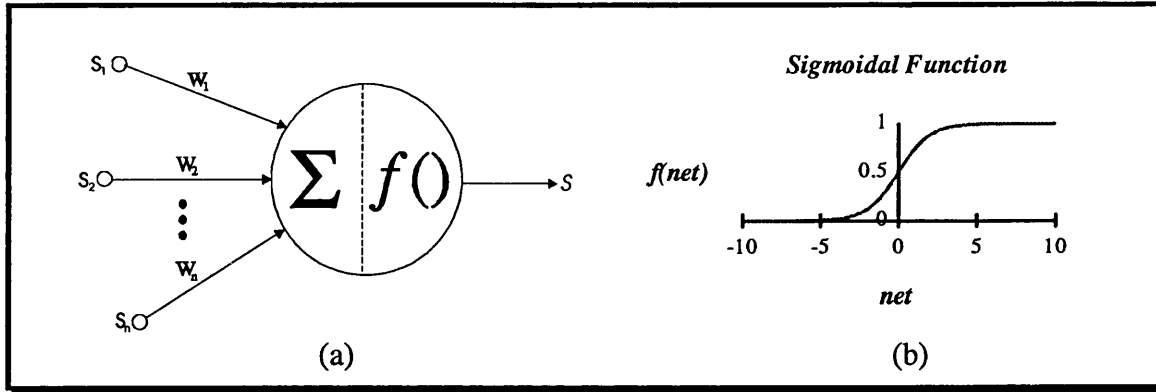


Figure 1.2 — (a) Artificial Neuron; (b) Sigmoidal Activation Function

Learning procedure is an iterative process, by which the network is presented several times with training patterns. It is either *supervised* or *unsupervised*. In supervised learning, input patterns are presented to the network together with the corresponding desired output pattern. The network's task is to adjust the connection weights to provide the correct input-to-output mapping. The procedure for adjusting weights is usually based upon the PE's error value, typically the difference between the expected and the computed output of each PE [199]. However, there are some applications where the desired input-output mapping is not available or known. In those cases, where input patterns are classified without any knowledge of the target output, unsupervised learning is more suitable. The network detects the patterns' regularities and classifies them into disjoint groups according to their similarity categories [99].

Learning is far more computationally intensive than the recall procedure, which suggests that hardware implementation is needed. Nevertheless, a large silicon area is generally used, thus limiting the number of possible artificial neurons integrated into a single chip. Learning can be easily implemented in digital circuits. However, it is very difficult to implement learning with analogue circuits, due to the complexity in conceiving programmable analogue chips. As discussed later, some specific applications demand real-time responses and on-line learning, which can only be achieved by specialised hardware.

Finally, **input values** for neural models are characterised by the adopted range, which can be either binary or continuous.

There is an extensive range of different neural network models — more than 100 neural network models have been reported [80, 82]. These models differ in specific features such as network topology, activation functions, and learning algorithms. These features are fundamental for hardware implementation, which is further discussed in this chapter. Before entering into this discussion, the next sub-section briefly points out some potential applications of neural networks, which may benefit from hardware implementation.

1.1.2. Applications of Neural Networks

Neural computing can be applied in a wide spectrum of fields, ranging from commercial to engineering applications [76]. It is important to identify these applications and investigate the benefits they can obtain from hardware implementations of neural networks.

The commercial world is a fertile area to apply neural networks to improve earnings and lower losses. Candidate applications involve financial analysis, marketing, and optimisation, which are usually simulated in conventional computers. Engineering applications include those problems that are intractable or extremely cumbersome by traditional computing methods [44, 82], and can, in general, benefit from hardware implementation. Such applications, found in industrial and military systems, encompass pattern matching, adaptive control, and signal analysis. In particular, neural computing can solve problems that require pattern recognition, pattern mapping, dealing with noisy data, pattern completion, associative look-ups, and systems that learn or adapt during use [79].

Initial results in applying neural networks in financial systems have shown their ability to handle non-linear data sets with parameter shifts [93]. This brings a good alternative to the area of financial forecasting to overcome the limited ability of modelling such data processes with current techniques. Other candidate applications include financial evaluation and analysis. Problems such as improving forecasting, credit evaluation, and securities trading have been tackled with relative success [42, 51, 201]. Examples include the application of neural networks to bond rating [51], which has been reported as performing better than mathematical modelling techniques like regression; and mortgage underwriting judgements [42], which has outperformed underwriters' judgements by employing a system composed of multiple neural networks, each of which performing part of the application evaluation's task. The nature of these problems are today adequately explored through simulations in conventional computers. It is generally expected that hardware implementations should not play an important role.

Speech generation and analysis are typical engineering problems that have been given much attention. A good example of speech generation is the NETtalk system [165], in which a Back Propagation neural network [161] has been trained to translate English text to speech. Speech-recognition tasks have also obtained good results [196, 197], which have encouraged several researchers to continue tackling the problem to obtain better performance [107].

Control systems are another branch of neural network applications [94] which have been investigated over the last decade in the industry as well as in military systems. Control problems cover a wide range of complexities, from simple systems, such as balancing an inverted pendulum [18], to complex systems such as autonomous control of a moving vehicle [111].

Applications of neural networks for military systems fall mainly into the category of automatic target recognition (ATR) [200]. ATR is a problem that involves extraction of critical information from complex and uncertain data, such as distinguishing tanks from trucks, jeeps, and other less important targets. A multiclass discrimination system is required, for which traditional techniques of signal processing, pattern recognition, and rule-based artificial intelligence have been unable to provide adequate solutions, as previous ATR systems have experienced high false-alarm rates [156]. Other applications in military systems include anti-missile homing, adaptive tracking, and radar signal categorisation [200].

Engineering applications generally demand hardware implementation to meet the required performance. Applications particularly found in control and military systems present two basic characteristics. Firstly, they are typically embedded systems. Secondly, they perform specific tasks which are critical in time, requiring fast response from the controlling devices. Clearly, these two requisites call for the construction of dedicated and application-specific hardware.

1.1.3. Neural Network Implementations

Implementations of artificial neural systems fall into three categories: *simulation* in conventional (sequential or parallel) computers, *emulation* in general-purpose neurocomputers, and *hardware execution* in special-purpose neurocomputers [180, 182]. Simulation is usually provided through software environments for programming neural network models and applications. A high degree of flexibility is achieved, since any neural model can be programmed and simulated, but low performance is obtained. General-purpose neurocomputers allow emulation of a range of neural network models, therefore obtaining high degree of flexibility while medium performance is generally

provided. Special-purpose neurocomputers implement a specific neural network model directly in hardware. High performance is obtained, although at the cost of reduced flexibility.

Neural programming environments span over a large variety of products, ranging from simple academic environments to more complex research and commercial products [183]. These systems have been mainly used to simulate neural networks in sequential computers, despite the fact that artificial neural networks present high degree of parallelism. The ESPRIT II *Pygmalion* project [19, 24] is an example of a complex research system, which will be described in the next chapter.

General-purpose neurocomputers have been mainly addressed by two approaches: accelerator boards, typically based on floating-point or standard state of the art digital signal processor, plugged onto workstations; and parallel processor arrays, formed by a large number of simple processing units, usually based on systolic array techniques specialised in matrix multiplication operations [153]. General-purpose neurocomputers are generally implemented using digital electronics technology. The ESPRIT II *Galatea* project [20], successor of the *Pygmalion* project, extends the neural programming environment to operate in conjunction with a general-purpose distributed neurocomputing system. The *Galatea* system will be briefly described in the next chapter.

Special-purpose neurocomputers vary enormously, depending upon the technology and technique adopted: electronics or optics, and digital or analogue. Many digital and analogue electronic neural chips have been successfully built [81, 82]. Optical neurocomputers have been proposed by some researchers [8, 52, 81, 103, 151, 203] since this technology promises to fully exploit neural network characteristics [28]. Chapter 2 discusses these issues in more depth.

Considering the potential applications of neural networks discussed in the previous sub-section, it is unlikely that commercial systems will require specialised hardware. Simulation in conventional computers should dominate in commercial applications. Accelerator boards could also be used to speed-up some time-consuming applications. Conversely, some engineering problems, such as embedded systems found in process control and military systems, require the construction of specialised neurocomputers.

The development of neurocomputers and neural programming environments is recognised as fundamental to the progression of neural computing. However, these research directions have been carried out independently, with no integration between software and hardware tools. Clearly, it is important to combine these two trends and develop a complete integrated system incorporating two routes towards neural networks'

realisation: simulation on conventional computers or emulation on general-purpose neurocomputers; and optimised hardware execution through special-purpose neurocomputers, using high performance application-specific neural chips. This research involves the integration of such a system and the development of a hardware design tool that automatically synthesises application-specific chips from a neural network programming language.

1.2. Silicon Compilation

The term *silicon compilation* is used to describe the concept for the whole process of implementing an integrated circuit from an abstract high level behavioural input description [59, 198]. Such a powerful design automation tool has been the ultimate goal of the microelectronics industry over the last decade [59]. It has enormous strengths which includes:

- considerable reduction on the development cycle of VLSI chips;
- allowing the designer to experiment with several different alternatives to obtain, for example, the optimal size/speed trade-off for a particular application;
- *correct-by-construction* feature, a very important issue, since the complexity of VLSI circuits has increased considerably, with single chips containing over a million transistors being commonly used nowadays; and
- easier management of the design cycle, whereas high level behavioural specifications are generally shorter than low level structural specifications, easier to write, understand, and modify, less error-prone, and faster to simulate.

Conversely, the use of design automation tools present one disadvantage, in particular at the highest level of the synthesis process: they cannot, in general, match the abilities of a skilled human designer, resulting in larger and slower hardware [35, 59, 162].

Silicon compilation can be divided in two distinct tasks: high level synthesis and low level synthesis. The former concerns with the identification of the input description, which is fully behavioural (also called functional), and the construction of a hardware structure that implements exactly the described functionality. The latter usually involves logic synthesis and mapping of the hardware structure onto a technology-specific description, which is then used to produce the actual integrated circuit.

Low level synthesis tools have been developed over the last decade with great success [61, 102, 139]. Several steps of the low level synthesis task are already automated and used through the CAD industry [2]. In contrast, high level synthesis tools are still in their infancy [187, 198]. In fact, the design of such a tool is a very difficult task when

optimal results are sought [162]. There is no single mapping from one behavioural description to the structure that implements it. Some assumptions are generally required, such as the target architecture of the synthesised circuit, which constrains the use of the silicon compiler to specific domains. Therefore, two classes of silicon compilers arise: special-purpose and general-purpose.

Some special-purpose silicon compilers have been developed with good results, in particular the ones that are dedicated to generate Digital Signal Processing (DSP) chips [108]. Conversely, although some interesting general-purpose silicon compilers have been developed [30, 120], none have yet been accepted as a standard tool by the CAD industry, because the quality of the synthesised circuits is still poor when compared to the circuits produced by experienced hardware designers. Chapter 3 examines some current developments on silicon compilers.

The development of a silicon compiler system directed towards neural computing field is seen as very attractive. This is motivated by analyses of the typical design cycle of neural network applications. Firstly, the application is defined and mapped onto a neural network, which is specified using a neural network programming language. This involves the specification of the neural model and the configuration of the neural network. Secondly, the network is trained using the simulation tool provided by the software environment. Finally, after the network has been tuned to the particular application, the user may wish to execute the trained network in hardware, which is particularly useful in industrial and military systems.

The design of a tool that synthesises integrated circuits for the execution of the neural network in hardware provides a fast route towards the development of specialised neurocomputers. Ideally, such a tool, a neural silicon compiler, takes the same neural network description used for programming and simulation purposes, and generates one or more identical integrated circuits. Furthermore, applications that require on-chip learning also have the neural network's learning algorithm *silicon compiled* into the specialised neural chips.

1.3. Motivations and Thesis Goals

Two basic tools are required for neural computing: a neural programming environment, in which any neural model can be simulated in conventional computers, or emulated in general-purpose neurocomputers; and a neural silicon compiler (NSC), which provides fast prototyping of specialised neural chips for hardware execution.

Neural network programming environments are powerful tools during the specification, simulation, and training of the neural network. Neural silicon compilers are important to integrate neural networks into one or several chips, and to implement (possibly trained) neural networks in specialised hardware. While programming environments are mature tools, neural silicon compilers are still rudimentary [132, 137, 189]. This thesis reports the development of a neural silicon compiler (*NSC*), and the approach adopted to fully integrate it into the *Pygmalion* neural network programming environment [24].

An important requirement for this integrated system is that a single specification language is used by both the programming environment and the *NSC*. Therefore, the designer uses the same language to specify, simulate, and train the network, as well as to generate neural chips that implement the specified neural network.

Another requirement pursued throughout this research is the minimum knowledge of VLSI design from the designer. This implies using a software-oriented specification language, rather than hardware-oriented, as the input for the *NSC*. Other developments of silicon compiler systems assume that the user is a hardware designer, who uses the silicon compiler as a replacement for conventional VLSI design tools [36, 121, 157]. This fact distinguishes the *NSC* developed in this thesis from those silicon compilers. Rather than letting the user provide hardware-specific information, a great degree of effort is put upon the design of the *NSC*. Moreover, optimised hardware structures are achieved by constraining the target architecture to an appropriate VLSI architecture, namely the *Generic Neuron*, which has been devised as part of another PhD thesis [190].

The integration between the *Pygmalion* system and the *NSC* is enforced through the development of a simulator for the hardware's target architecture. The simulator extends the software environment, allowing the user to experiment with several hardware constraints and to analyse their impact during the execution of the neural computation in hardware. While the *NSC* can provide the *correct-by-construction* feature for VLSI design, it cannot guarantee correct neural computation. The hardware simulator bridges this gap providing the *NSC* with the necessary information concerning the hardware constraints.

The development of the *NSC* is concentrated on the high level synthesis part of the silicon compiler. As mentioned earlier, low level synthesis tools have reached a stage of maturity, while high level synthesis tools still need some research. The aim of the *NSC* is to extract from *nC*, the C-based input language, all relevant information of the neural network, and to synthesise optimised hardware structures for neural chips. The *NSC* outputs the result as a structural representation at the register transfer level of the circuit in

VHDL, the IEEE standard hardware description language [6]. This approach relies on state of the art CAD tools, which are capable of implementing low level synthesis from VHDL.

Finally, it is paramount for an appropriate exploitation of neural networks' hardware that a trade-off between performance and flexibility be achieved. High performance is obtained through the synthesis of specific neural circuits dedicated to execute a particular neural model. Flexibility is provided at the design stage by the *Pygmalion* environment, which allows experimentation with several different neural models, and by the *Generic Neuron* VLSI target architecture [190], which is capable of implementing most of the neural models.

1.4. System Overview

This section outlines the major parts of this research, which comprises:

- introduction of hardware extensions to the *nC* language;
- design of a simulator for neural network hardware;
- design of the intermediate code representation (*ICR*) for neural network hardware;
- implementation of the *NSC*, comprising the data path and control synthesis tools; and
- translation of the synthesised circuits into a VHDL description.

The incorporation of *hardware extensions* in the *nC* specification language has considered the requisite of minimum (or even no) knowledge of VLSI design from the user. Typically, the user only needs to supply high level data, such as the weights' precision, activation function's realisation, and time constraints in any of the neural computing phases: recall and learning. This extended *nC* specification is the input for both the hardware simulator and the *NSC*.

The design of a *simulator for neural network hardware* has four main goals: to provide a tool to analyse the impact caused by hardware constraints on the design of digital neural chips; to provide a way to assess, in advance, the performance of the target architecture design; to provide the necessary hardware information for the *NSC*; and to permit the experimentation with different hardware configurations (such as precision requirements for weights and states, specially during the learning phase), which will ultimately result in smaller (i.e. VLSI area) and faster neural chips.

The design of an intermediate representation for neural networks' hardware serves as the starting point to the hardware synthesis tools. The *nC* language is initially compiled using conventional techniques of software compilation. The textual description is

translated into a parse tree representation, from which several optimisation steps are carried out. This representation is further translated into a graph-based internal representation, which is more suitable for expressing data dependency and control flow for hardware synthesis.

The specification of the *intermediate code representation (ICR)* separates the *NSC* system from the input description and the programming environment. Therefore, any other (neural network) specification language can be used, given that the translation to the *ICR* is provided. Similarly, different programming environments can be built upon this intermediate representation. This approach provides an open system, in which new developments can be integrated on top of the *NSC* system.

The *implementation of the NSC* centres on allocation and scheduling algorithms [35, 46, 114, 146], which are employed in conjunction with each other to create efficient data path hardware structures for the neural circuits. These algorithms are interdependent and the order of their execution has a considerable impact on the results. Generally, scheduling of operators and operations to control steps attempts to minimise the execution time of a particular task, while allocation of operators and variables to structural elements seeks the minimisation of hardware components, and consequently, VLSI area.

The approach adopted in this thesis comprises two modes: *default mode* and *user-driven mode*. The first approach tries to produce the smallest possible circuit. This is obtained by employing hardware allocation before scheduling [46]. In this case, a straightforward sequential schedule is adopted, since duplication of hardware components is not permitted. The second approach requires a more elaborated scheduling strategy to match the user-specified time constraint, such as the time required to accomplish a single feed-forward computation of the whole network. In this case, scheduling is employed through an *ASAP (As Soon As Possible)* strategy [146], which leads to a highly parallel implementation, since functional units are duplicated. In both cases, hardware allocation makes use of multi-port storage elements and multiple busses for interconnecting data and operators.

Control synthesis is performed after data path synthesis, and consists in constructing a finite state machine that implements a controller for the data path created [77, 152]. Two-phase clocking scheme is employed, which permits a certain degree of parallelism in executing neural algorithms.

After the neural circuit has been synthesised, its hardware structure is translated into a *VHDL description*. A library of technology-dependent VHDL modules (ALU, multi-port registers, flip-flops, RAMs, etc.) has been built during this work, which is

known and used by the *NSC*. These modules are parameterised and used by the module generator, another integral part of the *NSC*.

The module generator builds specific VHDL cells based upon the hardware allocation strategy carried out by the *NSC*. These cells are automatically generated and configured to be used according to several issues related to the synthesised data path structure. For example, multi-port registers are generated following the description on the number of busses connected onto the register, the definition of the data width, and the optional use of tri-state logic.

1.5. Thesis Contributions

This thesis presents an effective and consistent approach to support two of the specialised tools for appropriate utilisation of neural computing: a neural programming environment and a neural silicon compiler. The need for integrating these two tools has been fully investigated, producing a system that enables the user to use two different routes for neural networks written in the *Pygmalion*'s *nC* specification language: either binary, for simulation; or silicon, for hardware execution.

Based on this motivation, a neural silicon compiler has been developed, focusing on the high level synthesis part of the process. The *Pygmalion* environment has been extended to provide an integrated system for exploiting software and hardware in neural computing. The main contributions of this thesis are:

- extension of the *Pygmalion* neural network programming environment, consisting of hardware extensions to the *nC* language, and hardware extensions to the environment as a whole, which comprises a hardware library, the hardware simulator, and the *NSC*;
- development of a parameterised simulation tool for neural networks' hardware, which has generated a platform for investigating the effectiveness of implementing neural networks in digital hardware; the results obtained through these simulations are reported in chapter 5;
- development of a high level synthesis tool that takes a fully behavioural C-based program as input, identifies all the required information concerning the neural network's topology, neuron functionality, weights and states data, and creates a hardware structure for the neural chips;
- development of data path and control synthesis algorithms, based on heuristics that guide the generation of the neural circuits towards the *Generic Neuron* architecture;

- development of a cell library of VHDL components, dedicated to the implementation of the neural circuits, and development of a translator for the synthesised hardware structure into a VHDL description, at the register transfer level. This description is capable of being input into any commercial CAD system supporting VHDL and low level synthesis tools.

Additionally, this research has generated three published papers [15, 132, 133], and has contributed to the ESPRIT II *Galatea* Neurocomputing System [20].

1.6. Thesis Organisation

The remainder of this thesis is organised in nine chapters, which range from a survey of neurocomputers and silicon compilers to a detailed description of the whole system and the implementations carried out during this research.

Chapter 2 examines existing neural computing systems. It firstly presents an overview of neurocomputers, followed by a description of some implementation examples using electronics and optical technology. It then examines neural programming environments, and concludes with a proposal for integration between hardware and software.

Chapter 3 presents a survey on silicon compilers. It starts by characterising silicon compilers through the identification of all phases of the process. This is followed by a taxonomy of silicon compilers, which are classified into general-purpose and special-purpose. It carries on with a review of some systems, and concludes with a brief description of the neural silicon compiler developed in this thesis.

Chapter 4 presents a description of the *nC* neural network specification language, which has been developed as part of the ESPRIT II *Pygmalion* project [24]. It initiates with a review of the *Pygmalion* programming environment and its functionality. After describing *nC*, this chapter concludes with a description of the proposed and developed hardware extensions for the language.

Chapter 5 reports the development of the simulation tool for neural networks' hardware. It begins with a general overview of the problems involved in the realisation of neural networks in digital hardware. It then discusses hardware constraints, which are specific to neural computing, analysing both recall and learning phases. The chapter concludes by presenting the simulation results, which form the basis for studying the implementation of neural networks in digital hardware.

Chapter 6 presents the neural silicon compiler framework proposed in this thesis. The two key components, the *nC* input description and the *Generic Neuron* target architecture, are reviewed from the silicon compiler viewpoint. A Back Propagation neural network prototype for this architecture is implemented and simulated in VHDL, along with the construction of the VHDL cell library.

Chapter 7 gives a thorough description of the neural silicon compiler developed in this research. It starts with the procedures adopted to compile *nC* into the internal representation (*ICR*), which has been defined during this work. All successive transformation steps are then fully detailed, including descriptions of the hardware synthesis algorithms.

Chapter 8 investigates the internal structure of the synthesised neural network circuit by comparing it with the prototype developed in chapter 6. It evaluates the *NSC* by running the synthesis of a Back Propagation neural chip, which serves as the basis for a discussion on the quality of the generated neural circuits.

Chapter 9 provides an assessment for the whole research developed in this thesis. The main elements are discussed, investigating their strengths and weaknesses. These include the integration of the *NSC* into the *Pygmalion* environment, the hardware extensions introduced into the *nC* language, the simulator developed for neural networks' hardware, the high level synthesis developed for the *NSC*, and finally the VHDL description of the neural chips, which is ultimately the output generated by the *NSC*.

Chapter 10 concludes this thesis by discussing the results achieved during this work, drawing some conclusions, and pointing out possible future work.

Chapter 2

Neural Computing Systems

This chapter provides a review of proposed schemes for exploring neural computing technology. These include the development of neural programming environments, the design of novel architectures for neurocomputers, and investigations of new technological alternatives. The chapter concludes with a discussion on how these schemes can be integrated into a single development system for both software and hardware exploitation of neurocomputing.

2.1. Overview

Implementations of artificial neural networks fall into three categories:

- *simulation* in conventional computers;
- *emulation* in general-purpose neurocomputers; and
- *execution* in special-purpose neurocomputers.

Most implementations of neural networks are based upon simulations in conventional computers, which are adequate for various current applications. However, for several real-time applications found in engineering systems, this approach does not provide the necessary performance. For these applications, the development of general-purpose or special-purpose neurocomputers — which are optimised to the computation of neural models — is required [180].

Simulations are generally built together with software environments, which provide tools for programming and simulating neural network models and applications, called *neural network programming environments*. They are powerful and flexible systems for experimenting on neural models and applications. However, since they are usually built upon conventional computers, the performance is generally poor.

General-purpose neurocomputers attempt to provide the same kind of flexibility offered by software simulations, but with an improved performance. They are generally attached to a neural network programming environment. In this case, the environment is

expanded, so that neural network algorithms can be simulated in the supported conventional machines or emulated in the massively parallel neurocomputer.

Special-purpose neurocomputers lack flexibility in executing different neural models, but are capable of yielding very high performance through optimised hardware resources, which are tuned to a specific neural algorithm. They are demanded in several important applications, such as the ones found in process control and military systems, which generally require (embedded) real-time execution.

The development of VLSI special-purpose neurocomputers requires a set of sophisticated hardware design tools to help the construction of specialised chips. In spite of these tools, the design of integrated circuits remains a laborious task. In addition, ASIC design tools are completely independent of neural programming environments. The incompatibility between software and hardware tools for neural networks raises the problem that a neural network application tested in the software environment has to be fully re-designed from scratch if the designer wishes to map the application onto hardware. Therefore, the construction of a set of integrated tools is demanded to automatically provide fast prototyping of application-specific neural network chips (ASNNCs) from a high level specification of neural network applications. This issue is investigated throughout this dissertation.

The subsequent sections review some existing schemes for implementing neural networks. A proposal for integrating programming environments with the automatic synthesis of special-purpose neural chips concludes this chapter.

2.2. Neural Programming Environments

The continuous developments in the neural computing area, where new algorithms and applications are commonly developed and explored, have fuelled the design of sophisticated software environments for programming neural networks. These environments range from commercial products, provided by several companies, to academic systems, generally available free from university research groups [154].

The implementations of software environments generally follow current trends towards the design cycle of a neural network application. Figure 2.1 illustrates a typical path, which usually starts with the definition of the neural network in a high level programming language. This includes the complete specification of the algorithm or the modification of parameterised models, which are supplied in a algorithm library. The high level programming language is then translated into an intermediate-level, machine independent, neural network specification language, that describes the complete neural

network and its configuration for the particular application. This portable specification can then be translated into a simulation file for execution in a specific machine. The execution of the simulation is controlled by the command language together with the graphic monitor.

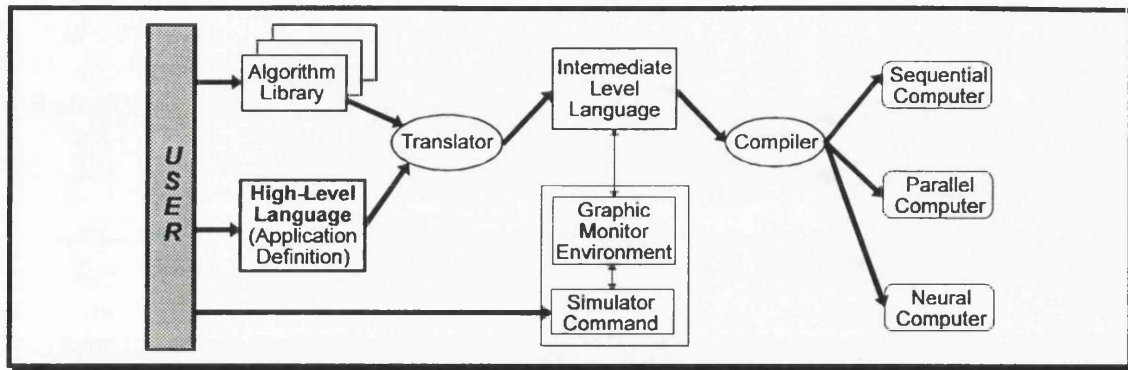


Figure 2.1 — Current Design Cycle of Neural Network Applications

According to the intended use and scope of the application, neural programming environments can be classified into three distinct categories [183]: *application-oriented*, *algorithm-oriented*, and *general programming* systems.

Application-oriented systems are designed for specific market domains, such as finance and transportation. Professionals in these fields have very little neural network expertise, but want to apply the technology without having to deal with the process of programming algorithms.

Algorithm-oriented systems provide an environment in which the user can experiment with a particular neural model, or several models, which are held in a library of parameterised algorithms.

General programming systems, provide a flexible tool for investigating a wide range of algorithms and applications. They can be further sub-divided into: *educational* systems, which provide, for the novice user, a simple tool that helps in learning neural network techniques; and *general-purpose* systems, which are the most sophisticated environments for programming any algorithm and application. General-purpose systems can still provide an *open* environment, where the user can modify any part of the system. In such an environment, the network can be mapped onto certain target computers, such as the parallel transputer-like architecture, thus taking advantage of the natural parallelism of neural networks.

Although the spectrum of neural programming environments have different design goals, they share several common features, which comprise:

- a *graphic interface*, with menus and a command language for configuring a neural network, controlling and monitoring the simulation;
- an *algorithm library*, holding a parameterised set of the most common neural network models;
- a *high level language*, for programming or customising an algorithm or application;
- an *intermediate-level network specification language*, which holds the (partially) trained network in a machine-independent representation; and
- *compilers*, for mapping the network representation to the available target machines.

Table 2.1 exemplifies some of the well-known neural programming environments. Further details on these and several other systems can be found in [183].

<i>System</i>	<i>Organisation</i>	<i>Category</i>	<i>Brief Description</i>
<i>Nestor Learning System</i>	Nestor	Application-oriented	Core of three major products: the Decision Learning System, the Nestor Development System, and the Nestor Character Learning System. Targeted to the Financial market [42].
<i>AMT</i>	BehavHeuristics		Adaptive decision making system targeted to airline transportation control.
<i>BrainMaker</i>	California Scientific Software	Algorithm-oriented	Complete environment targeted to the Back Propagation model.
<i>NeuroShell</i>	Ward Systems Group		Back Propagation shell. Designed to solve classification type expert systems problems.
<i>PDP</i>	University of California, San Diego	Educational Programming system	Simulation package that provides a wide range of exercises for experimenting with PDP models [159].
<i>Explorer</i>	Neuralware		Small application development. User can write C code for network's monitoring. On-line help facility, and <i>checkpoint</i> for saving and displaying the user specified networks.
<i>Genesis</i>	California Institute of Technology	General-purpose system	X-Window based research environment simulation kit. Includes graphical interface (Xodus), a script language interface, an interpreter, and other supporting facilities.
<i>Anzal/Axon</i>	HNC		Commercial product. Includes NetSet, a graphical interface; AXON, a language description; a library of mathematical subroutines; Interact, an interactive debugger; and a co-processor board [69, 82]
<i>ANSpec</i>	SAIC		Commercial product. Includes a graphical interface, an object-oriented language (ANSpec), an algorithm library, and a co-processor board (DELTA).
<i>Pygmalion</i>	ESPRIT II Project		Research environment. Includes graphical interface, high level and intermediate-level languages, algorithm library, and compilers.

Table 2.1 — Neural Network Programming Environments

2.3. Neurocomputers

The development of specialised hardware for neurocomputing attempts to achieve the required performance through the exploitation of the inherent parallelism in neural networks. A neurocomputer is essentially a parallel array of interconnected processors that operate concurrently. Therefore, it is important to devise an appropriate architecture for the neurocomputer that efficiently explores issues like *parallelism*, *performance*, and the relationship with the (*silicon*) *device area*.

Several different approaches have been addressed to tackle these issues. They can be divided according to the technology adopted in the construction of the neurocomputer (electronic, optical, or electro-optical devices), and the technique used for the circuits (analogue, digital, or hybrid).

Most of the research has been carried out in the electronics domain, since this is a well understood and mature technology [123]. Current fabrication technology permits the construction of structures less than 1 μm in size, allowing the implementation of several millions of transistors per chip. Thus, it is possible to build extremely complex electronic chips, or neural chips with many primitive processing elements (PEs). However, neural networks pose a difficult problem for electronics: the massive degree of interconnection. Current silicon technology allows the design of circuits only in a two-dimensional plane with few available layers. As a result, the required massive interconnectivity is bound to have a severe effect on the amount of VLSI area. Therefore, the interconnection scheme generally includes some form of multiplexing [25]. The multiplexed interconnection is shared by PEs in the network, thus limiting the total number of wires to a realistic size.

Optical computers can potentially circumvent the interconnection bottleneck, since beams of light can cross each other with no interference. Such capability is believed to be fundamental for building massively parallel neural computing systems [12, 82, 174, 199]. In addition, there are two other properties of optics that can make it outperform electronic processing: large time bandwidth, around a gigahertz for laser diodes, and large space bandwidth product, which provides the exploitation of the third dimension for data processing. However, this technology is still in its infancy and the results are still far from reaching any practical design. There are several technological issues that must be solved before appropriate exploitation of such system can be realised [170]. Nevertheless, attempts have been made to build optical neurocomputers, which are reviewed later in this chapter.

Some researchers are trying combine the best that each technology can offer by building electro-optical neurocomputers. Preliminary results indicate that this approach can

provide a good trade-off between electronics' high degree of integration and optics' high degree of interconnection [27, 53, 72, 73]. This suggests that the functionality of the PEs is implemented by digital electronic circuits, while the interconnection pattern is realised through optical devices.

Neural circuits are built based upon digital or analogue techniques in association with the technology used. In the electronics field, the greatest advantage shown by digital neurocomputers is flexibility. Programmable chips can be easily built, allowing the implementation of neural circuits with on-chip learning. The greatest advantage of analogue neurocomputers is size. Potentially, larger networks can be built onto a single chip, since simple operations such as multiplication and addition can be performed by only a few transistors. The digital counterpart requires several tens or hundreds of transistors to perform the same operations [82, 123, 149].

However, analogue implementations suffer from several problems; the most important ones being lack of thermal stability, limited accuracy, and high source of noise [123]. Along with the problems inherent to any analogue electronic implementation, there are some other specific problems that directly affect neurocomputers. Firstly, the implementation of memory (required to support the learning) remains a difficult task [49, 178]. Secondly, the interconnection among several processing elements is another difficult problem, particularly if it is required to connect multiple chips. Finally, it is not currently possible to mass produce analogue chips with predetermined weight values [67, 82, 83, 123, 149].

The next sub-sections review some existing implementations of electronic chips (analogue and digital) as well as some dedicated neural optical systems.

2.3.1. Electronic Neural Chips

The implementation of integrated circuits dedicated to neural networks is divided into three categories: analogue, digital, and hybrid.

Analogue Designs

Analogue hardware implementations of neural networks commonly use amplifiers, based on transistors, parasitic capacitors, and resistors, to realise physical structures that resemble the simplified mathematical model of the neuron's state calculation:

$$s_j = f\left(\sum_{i=1}^N S_i \times W_{ij} - \theta\right)$$

The basic principle of operation of an analogue electronic neuron is shown in Figure 2.2.a. The amplifier mimics the above equation, i.e., it performs the sum of products followed by the activation function f . Inputs and outputs are replaced by wires, while synapses are replaced by resistors. These electronic neurons can implement a neural network by using a crossbar scheme as shown in Figure 2.2.b. The main problem with this approach is that resistors occupy a large silicon area.

Several research groups have designed and implemented electronic neural chips based on this principle. At *AT&T Bell Laboratories* [63, 64, 65, 66, 87, 88, 90] Graf and his group have designed a set of analogue chips, which include: a 22 x 22 array of micro fabricated resistive synapses (amplifiers are implemented separately) [88]; a fully interconnected array of 256 amplifiers and fixed connection matrix of resistors¹; and finally, an analogue-digital (hybrid) chip, described later in this section.

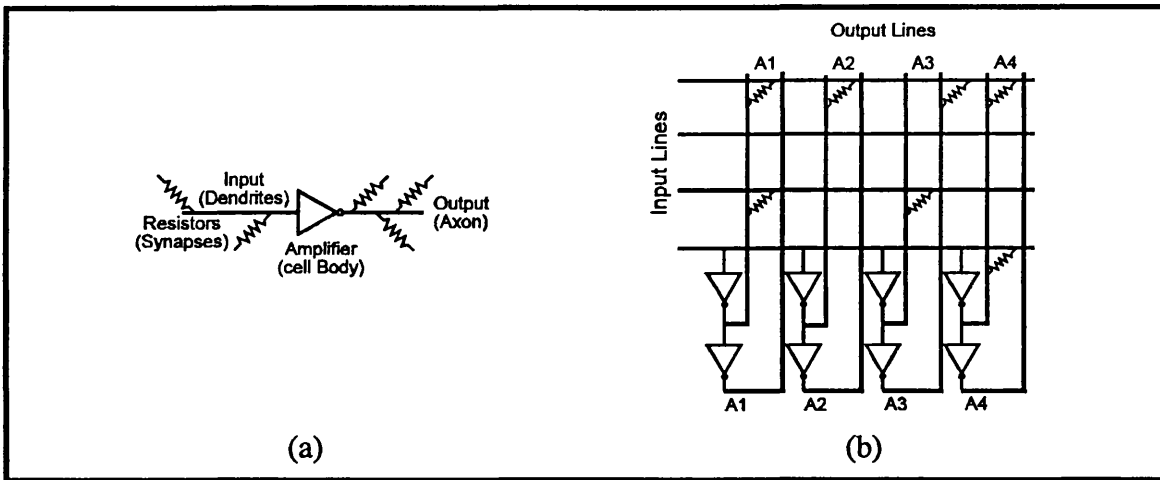


Figure 2.2 — Basic Analogue Artificial Neuron

At *Caltech*, a group led by Mead has designed an analogue neural chip based on the Hopfield model for associative memory [168]. The prototype contains 22 amplifiers and a fully interconnection matrix of 462 programmable interconnection elements. Mead's approach for implementing large arrays is also to limit the current consumption of the amplifiers, which is done by keeping most of the MOS devices in a sub-threshold regime of operation [116]. The synapses are implemented by 41 transistors, most of them operating in their ohmic (resistive) regime. Mead's interests lie primarily in the implementation of processing functions such as the retina and the problem of motion, which are reported in his book [116].

¹To reduce the size of the resistors, a special fabrication process has been developed, in which a finished CMOS chip is covered with amorphous silicon and the resistors are patterned by electron-beam lithography and reactive-ion-etching [66].

Intel Corporation has commercialised an *experimental* analogue chip, called the 80170NX Electrically Trainable Analogue Neural Network (ETANN) [1, 84]. The chip contains 64 neurons and 10,240 synapses (4,096 used to fully interconnect the 64 neurons, 4,096 to connect the 64 neurons to 64 independent inputs, and 2,048 used for setting bias for each neuron) using “floating gate” non-volatile memory technology. This analogue memory replaces the digital flip-flops with a considerable reduction in the synapse’s size. The synapse cell comprises a pair of EEPROM 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. These differential currents are summed and transferred through a sigmoid function to the neuron’s output. Although the chip can perform fast computation, low accuracy is obtained. Typical resolution of the analogue inputs and outputs ranges from 4 to 6 bits. Better accuracy is obtained if the *Bake Re-Training Method*² is used, in which the circuit is exposed to high temperature for some time to reduce the effect of charge movements in the nitride layer between the EEPROM cells [1]. Learning is done off-line by a conventional computer with weights set one at a time. Cascadability is possible, although limited, in which chips can be connected to others directly or through a bus to increase the number of neurons per layer or the number of layers per network. However, they cannot increase the number of synapses per neuron.

Digital Designs

Fully digital neural chips are based on an array of simple (digital) processing elements that can potentially be designed to perform any desired function, thus broadening the scope of application to several neural models.

At the *Institut National Polytechnique de Grenoble* (INPG), France, a fully digital neural chip has been designed employing a two-dimensional array of PEs [137, 138]. The novelty of this approach concerns the communication mechanism, which is done by an elaborate shifting scheme on the array, where data busses are connected through *soft switches*. The basic structure of the PE contains: a memory block for holding synaptic weights and algorithmic-dependent parameters; a data path dedicated to perform multiplication and summation; a controller that implements a finite state machine; and input and output registers, used to shift the neuron’s state value onto the bus. The PE is able to autonomously perform all steps of the recall and learning phases, and it can be customised to the required precision of weights and states through the *Galatea* silicon compiler (see section 3.4.5). The main disadvantage of this approach is the complexity of

²Due to the floating gate memory technology, the value of a weight changes in time. For long period of time, 4-bit resolution is the limit. However, if Bake Re-Training method is used, resolutions greater than 6 bits can be achieved [1]

the communication scheme. Cascadability is limited by the number of pins available per chip. However, it is claimed that this approach is tailored for future Wafer Scale Integration (WSI).

Neural Semiconductor Inc. has developed a different approach that uses stochastic functions of time to represent neuron activations and synaptic weights [179]. The activations are represented by firing frequencies, and the function of a synaptic weight is used to modify this frequency. This technique yields high density, since multiplying stochastic pulse trains require only a single *AND* gate, and the summation is realised by an *OR* operator. Two chips have been designed: the SU3232, containing 1,024 synaptic elements arranged as a 32 by 32 matrix; and the NU32, which implements 32 neurons, each including an 8-bit divide-by-*N* prescaler, a 16-bit integrator, and a stochastic pulse generator. The precision is controlled by the number of clock cycles (256), thus higher precision results in slower speed and vice-versa. The architecture is very flexible, being able to realise any network topology. However, on-chip learning is not possible; the chips are dedicated to the recall phase only.

At *Imperial College of Science and Technology* (UK), Aleksander and his group have developed a series of *WISARD* machines, which are adaptive pattern recognition machines, designed specifically for image processing [12, 13, 14]. The system is based on the principle that a simple memory device has neural-like properties [12]. The system centres on arrays of RAM cells that operate as *image discriminators*, in a way analogous to a hologram. Each discriminator consists of *K* RAMs of *N* bits followed by a summation operator that counts the number of RAMs in which the output value is '1'. The input images are digitised, so that each pixel is represented by a single bit. The discriminators are trained with example of objects (frames captured by a video camera). Therefore, the image (represented by *N*×*K* pixels) is fed into the discriminators, which indicate the similarities among the input pattern and each of the previously training set of patterns.

At *University College London*, research in neurocomputers have produced three distinct digital chips. The first is a general-purpose chips based on the RISC architecture [140, 180], in which PEs are interconnected through linear arrays in the form of rings to a host microcomputer. The PE unit is divided into communication unit and execution unit with local memory for program and data, providing a full MIMD machine. The second is also a general-purpose architecture, which combines MIMD and SIMD approaches into a single architecture [141]. This combination allows the implementation of *virtual neurons*, in which a single PE can embody the function of more than one neuron. The third is a special-purpose neural chip based upon the *Generic Neuron* architecture [140], described in chapter 6 of this thesis.

Several other research groups follow the digital design approach. Examples include the group led by Ramacher at *Siemens* [153] and the group led by Duraton at *Philips* [112]. Both designs are based upon systolic architectures providing efficient circuits to employ synaptic matrix multiplication [105].

Hybrid Designs

A hybrid neural chip combines analogue and digital techniques in an attempt to provide the best features of each. For instance, analogue circuits performing the propagation rule and the activation function may be combined with digital memories. This avoids the storage problems of analogue technology, and allows the implementation of on-chip learning [123].

Graf and his group have extended their analogue designs to provide learning for the chips [65]. They have designed a hybrid chip comprising 54 neurons (amplifiers) fully interconnected through a programmable resistor matrix of 2916 elements. Two RAM cells are used in the synaptic circuit, providing excitatory, inhibitory, and disable connections. The problem with this arrangement is that inhibition is six times stronger than excitation, which causes mismatching problems, thus limiting the precision.

At the *Catholic University of Louvain, Belgium*, Verleysen and his group have built a small Hopfield based analogue neural chip to test their approach [192, 193, 194]. The chip contains only 14 neurons and 196 synapses. The synapse circuit is similar to the one developed by Graf. The synapses are programmable and implemented by RAM cells, which were improved to overcome the problem of mismatching when inhibitory and excitatory currents are summed. Their solution consists in summing all excitatory currents on one line and all inhibitory currents on another. By employing identical transistors on each line, the mismatching problem disappears.

At the *University of Edinburgh*, Murray and his group have devised a different approach using pulse-stream arithmetic, which is inspired by its analogous relationship with biological neurons [124, 126]. In this scheme, neurons operate as switched oscillators, and the level of accumulated neural activity controls the oscillator's firing rate. Therefore, the neuron's output state is represented by a pulse frequency, and each synapse determines the proportion of the input stream that passes to the neuron's input. The architectural structure is based on a fully connected network, arranged in a two-dimensional array of synapses. The first prototype is a 3 μm CMOS chip, comprising a small network of 10 by 10 synaptic matrix and 10 off-chip neurons. The synapse circuit is based on a digital RAM, that stores the weight, and on *chopping clocks*, that form the pulse-stream input to the neuron. The basic component of the neuron is a

voltage-controller oscillator (VCO) that is controlled by the charge on a capacitor. This digital weight-storage memory occupies a large area, therefore the implementation of the subsequent prototype includes a fully analogue pulse-stream synapse [74, 125].

Murray's approach has received considerable support, being followed by a number of other research groups [74, 125].

2.3.2. Optical Neurocomputers

The most crucial issue in developing optical computers is to find suitable device technologies for the target application [82]. For fully optical neurocomputers, the research is centred on two basic issues: the devices that simulate the non-linear mapping performed by the neurons, and the interconnection.

Spatial Light Modulators (SLM) are perhaps the most important devices, being used as input-output devices, two-dimensional memory, and parallel operation device [203]. A SLM is basically an electrically controlled two-dimensional array of optical modulators. Several types of SLMs, built with different materials and principle of operation, are commercially available [28]. Most of them operate in a high contrast mode, producing a two-dimensional light modulation that is proportional to a thresholded version of an image, thereby simulating the action of neurons [151]. Although several other approaches have been investigated to simulate an array of neurons [174], the research in optoelectronics [89, 169] is probably the most important, in which two-dimensional arrays of detectors and light emitters (using light emitting diodes — LEDs) are integrated onto a single substrate. A proposal for such an optoelectronic neural system has been reported in [101].

Regarding interconnection, one of the most promising technologies for realising programmable optical interconnections (required for learning), with potential high density, relies on photorefractive crystals [151]. By recording the hologram on these crystals, programmable connections can be achieved.

Optical implementation of neural networks started to receive a great attention when Hopfield published his paper back in 1982 [86]. Since then, researchers started to devise optical techniques to implement the Hopfield Neural model.

Psaltis and Farhat [150] first proposed the arrangement shown in Figure 2.3, which behaves basically as associative memories. The row of transmitters (LEDs) represents N logic elements with binary states (LED on or off), the two-dimensional array represents the synaptic-weights, and the column of detectors (photo-diodes), together with the

feedback pattern, represent the array of thresholding devices. The feedback can be done electronically or optically. Another group, led by Athale, has developed a similar approach based on electro-optic components [23].

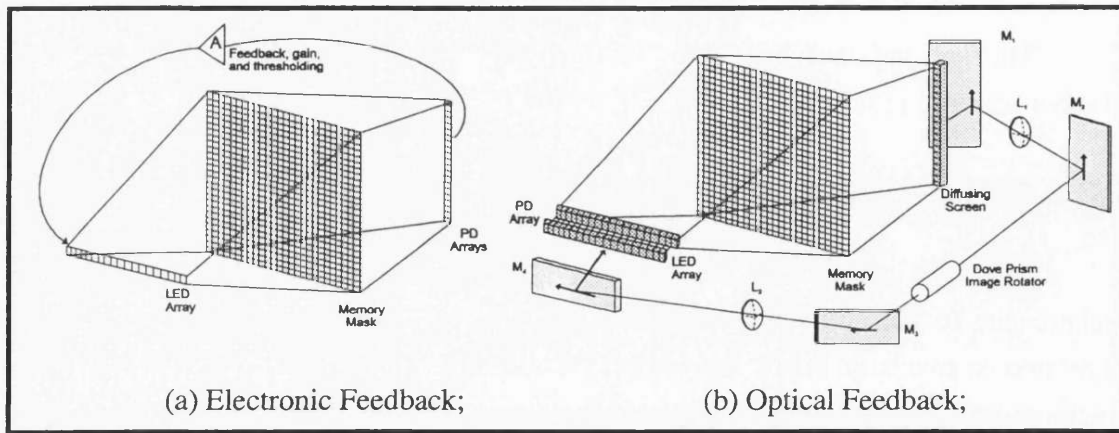


Figure 2.3 — General Schemes for Optical Implementation of Neural Networks

Psaltis and his group have reported several other experiences. A system based on a variation of the above scheme has been assembled for a network of 32 neurons [52]. A holographic associative memory arranged as a complex optical *loop* system for pattern recognition has been constructed [8]. The system comprises several lenses (L_i), mirrors (M_j), pinhole arrays, threshold devices (consisting of 10,000 elements simulating neurons), and a pair of planar holograms. A multilayer optical system performing an approximate implementation of the Back Propagation learning algorithm has also been built [195].

In Japan, two major companies are investigating optical neurocomputers: NTT and Mitsubishi. At NTT, an optical multi-layer Back Propagation network, with learning capability, has been constructed [98]. The modifiable synaptic weights were realised by using a photorefractive crystal and microchannel spatial light modulator. At Mitsubishi, an optical neural chip has been fabricated using the GaAs technology [135]. The device consists of an array of LEDs, an interconnection matrix, and an array of photo-diodes that were integrated into a hybrid-layered structure on a GaAs substrate. The fabricated device contains 32 neurons, and implements the Hopfield associative memory with three stored vectors.

2.4. Integrating Software and Hardware Tools

As seen before, techniques and technologies for implementing specialised hardware for neural networks are an intensive area of research. Although some areas, such as optical systems, are still in early stages of development, others, such as analogue and digital electronics, are very mature and can be applied immediately. For these systems, several

neural chips have been designed independently of neural programming environments, although a clear connection between them exists [68].

In this research, integration of software and hardware tools for neural computing is investigated in depth. An integrated system, able to automatically generate a specialised neural circuit tuned to execute optimally neural applications, is proposed. The core of this integration relies on the development of the hardware design tool, which in the case of electronic chips is called *silicon compiler*. In addition, a hardware simulator tool is also needed to simulate exactly the neural computation in hardware. This tool is required because hardware constraints can severely restrict the correct computation of neural algorithms.

By integrating these tools (programming/simulation environment and silicon compiler), the same neural network description used in the programming environment for simulation is also used by the silicon compiler for hardware implementation. Figure 2.4 depicts the general view of an integrated neural programming environment (see also Figure 2.1). The implementation of such an integrated system is discussed throughout this thesis.

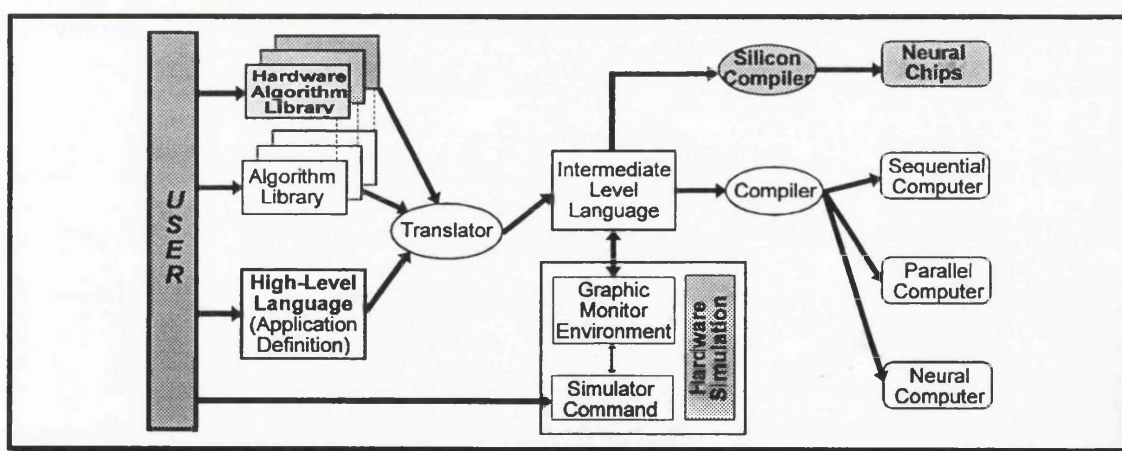


Figure 2.4 — Proposal for Integrating Software and Hardware Tools

2.5. Summary

This chapter has reviewed past, current, and future trends in developing systems to exploit neural computing technology. In the software domain, sophisticated neural programming environments have been developed. They provide an appropriate platform for experimenting with the continuous progress of neural models and applications. In the hardware domain, several researchers have devised new architectures and devices that can emulate the way artificial neurons compute. The range of options varies considerably, and

several different technologies have been tried. In the near future, electronics should continue to dominate the area, but in the medium to long term, optical devices should reach a point of maturity, in which practical optical neurocomputers might be constructed for several different applications.

Chapter 3

Silicon Compilers

This chapter presents the background for the high level synthesis system developed in this thesis. The entire silicon compilation process is first reviewed and its characteristics examined. Then, the anatomy of the high level synthesis process is investigated. The chapter concludes by presenting a simple taxonomy of silicon compilers, focusing the discussion on systems that employ high level synthesis.

3.1. Introduction

The design of digital VLSI integrated circuits is clearly a laborious task, involving several levels of specification: system, algorithmic, register transfer, logic, and circuit (Table 3.1). While system and algorithmic levels contain very little detail about the actual chip, the circuit level gives a detailed description of the elements that implement it. Along with the specification levels, designs can be described at three different domains: *behavioural*, *structural*, and *physical* [58].

In the *behavioural domain* the system is specified as a black box comprising a given set of inputs and outputs, and a set of functions describing the mapping from inputs to outputs. In the *structural domain*, the system is described by a set of interconnected components, while the *physical domain* describes exactly how these interconnected components should be built.

Level of Abstraction	Domain of Representation		
	Behavioural	Structural	Physical
System	Communicating Processes	Processors, Memories, Switches	Physical Partitions
Algorithmic	Input-Output	Memories, Ports, Processors	Clusters
Register Transfer or Architectural	Register Transfer	ALUs, Registers, MUXs, Busses	Floor Plans, Macro Cells
Logic	Boolean Equations	Gates, Flip-Flops, Cells	Standard Cells, Module Plans
Circuit	Network Equations	Transistors, Connections	Layout

Table 3.1 — The Hardware Design Hierarchy

The translation of a design's high level description to its final layout has been frequently defined as a silicon compiler process¹ [41, 43, 58, 171]. However, what is understood by high level description varies from system to system. Depending upon the level of abstraction contained in the input description, the complexity of the translation process differs considerably. This level of abstraction, among the several silicon compilers available today, ranges from parameterisable layout descriptions to procedural languages, from macro cell assemblers to user defined module generators, and from completely specified structural descriptions to behavioural specifications [59].

Today, the most accepted definition of silicon compilation comprises the whole process of an integrated system that implements a chip from an abstract behavioural input description at the algorithmic level [198]. Starting from this behavioural level, several transformation steps are required to synthesise the behaviour of the input description into a hardware structure.

High level descriptions starting at the system level (Table 3.1) require a further degree of transformation, which is higher than the one produced by current silicon compilers. Typically, it involves the configuration of a system comprising components such as processors, memories, and controllers, instead of designing those individual pieces. Thus, a system level synthesis tool comprises two main tasks: partitioning of processes into realisable sub-systems; and specification of interfaces between these processes. To date, none of the existing systems provide complete design aids at this level [187]. Nonetheless, a few research projects have started to tackle some of these issues [144, 177].

There are several reasons for producing tools that synthesise electronic circuits from an abstract description of a design's behaviour. The most important ones are outlined below:

- *Fast design cycle* — reduces the design time, and consequently the time to market, an important issue in today's competitive electronics industry;
- *Design space exploration* — permits the designer a high degree of flexibility, by experimenting several design possibilities, and avoids the tedious task involved in the low level tasks of VLSI design;
- *Correct-by-construction design* — eliminates the possibility of human errors during the design; and

¹Although the term *silicon compilation* was introduced in 1979 to describe the concept of assembling parameterised pieces of layout [92], its meaning has gained popularity and broadened dramatically during the 1980's.

- *Easy to specify designs* — permits wider utilisation of the system by non-expert users, and facilitates the design of very complex systems with little or no human intervention.

These strengths have a great impact when hardware implementation of neural networks is sought. Firstly, a fast route from a neural network specification to silicon is generally desired. Secondly, the details involved in designing integrated circuits should be avoided. Finally, there is a great need for implementing neural chips quickly and reliably, i.e., fast prototyping with high probability of obtaining working chips. Therefore, the investigation of a silicon compiler dedicated to produce neural chips represents an important tool for neurocomputing systems.

The next section analyses the characteristics of silicon compilers and identifies the current state of the art, giving motivations for the methodology adopted in this research.

3.2. Characteristics of Silicon Compilers

The design of silicon compilers can be divided in two main phases:

- *High level synthesis*, which generates a certain hardware structure for the specified behaviour; and
- *Low level synthesis*, which converts the structural description into a layout.

The term *high level synthesis*, in electronic design, concerns the identification of the high level input description, which is fully behavioural (also called functional), and the construction of a hardware structure that implements exactly the described functionality [35, 114, 187]. This hardware structure is usually a register transfer level (RTL) description, but can also include logic level descriptions. Since this process consists of translating behaviour into structure, it is frequently referred as *behavioural synthesis*. Because the synthesised structure implies a given architecture, this translation is sometimes called *architecture synthesis*.

High level synthesis tools are still under research development and their use by industry is very limited [114]. In fact, the design of such a tool is a very difficult task when optimised results are sought [162]. There is no single mapping from one behavioural description to the structure that implements it. Therefore, certain assumptions are generally required, such as the target architecture, which constrains the use of the silicon compiler to specific domains [109].

Low level synthesis is the actual construction of integrated circuits from RTL descriptions at the structural domain [38, 59, 108]. It involves logic synthesis and mapping of the hardware structure onto a technology-specific description, which is then used to

produce the final integrated circuit. Low level synthesis is also called *structural synthesis* or *silicon assembly*.

Low level synthesis tools have been developed over the last decade with great success [29, 31, 102, 139]. Several steps of the synthesis tasks are already automated and used throughout the CAD industry [32, 61].

High level synthesis and low level synthesis can be characterised by several aspects, such as the degree of abstraction of their input specification (see Table 3.1) and the complexity of their synthesis algorithms. The most important issues are reviewed in the next sub-sections.

3.2.1. Input Description

Input descriptions are extremely important for silicon compilers, since their level of abstraction has a great impact on the silicon compiler's complexity. Throughout this thesis, a high level description is assumed to be a behavioural description of the design at the algorithmic level. In addition, only systems that accept high level descriptions as input and generate the layout of an integrated circuit as output are considered silicon compilers.

As Table 3.1 suggests, input descriptions can be at the behavioural, structural, or physical domains. Based on this premise, the complexity of CAD systems varies considerably: behavioural specifications are likely to be the input of high level synthesis systems (and consequently silicon compilers), while structural descriptions are usually the input of logic systems, and physical descriptions are the input of layout synthesis systems. For purposes of clarification, Figure 3.1 shows a simple description of an adder at different levels of detail and domains of representation: algorithm level (ALG_Behaviour) and logic level (LOGIC_behaviour) both at the behavioural domain; and gate level (GATE_Structure) at the structural domain.

```

ENTITY adder IS
  PORT (X: IN BIT;
        Y: IN BIT;
        Z: OUT BIT;
        C: OUT BIT);
END adder;

ARCHITECTURE ALG_Behaviour OF adder IS
  BEGIN
    PROCESS (X, Y)
    BEGIN
      Z <= X + Y;
      IF (X = '1' AND Y = '1') THEN
        C <= '1';
      ELSE
        C <= '0';
      END IF;
    END PROCESS;
  END ALG_Behaviour;

  ARCHITECTURE LOGIC_Behaviour OF adder IS
    BEGIN
      PROCESS (X, Y)
      BEGIN
        Z <= X xor Y AFTER 5 ns;
        C <= X and Y AFTER 5 ns;
      END PROCESS;
    END LOGIC_Behaviour;

    ARCHITECTURE GATE_Structure OF adder IS
      COMPONENT and_gate
        PORT (
          in1 : IN BIT;
          in2 : IN BIT;
          out  : OUT BIT);
      END COMPONENT;

      COMPONENT xor_gate
        PORT (
          in1 : IN BIT;
          in2 : IN BIT;
          out  : OUT BIT);
      END COMPONENT;

      BEGIN
        add: and_gate PORT MAP (
          in1 => X,
          in2 => Y,
          out => Z);

        carry: xor_gate PORT MAP (
          in1 => X,
          in2 => Y,
          out => C);
      END GATE_Structure;
    END adder;
  
```

Figure 3.1 — Example of the Design Hierarchy in VHDL

Today, most of the existing hardware description languages (HDL) allow the design to be specified in several levels of abstraction. For example, VHDL², the IEEE standard HDL [6], can describe designs from system to gate level, at the behavioural and structural domains. EDIF³, the EIA standard interchange format [4], can describe designs at lower levels such as gate and circuit level, at the structural and physical domains. For this reason, VHDL is frequently the input for high level synthesis systems [37], and EDIF is frequently the input for low level synthesis systems, in particular, layout synthesis tools [166].

Today, there is a trend towards VHDL as the input language for high level synthesis, as well as logic synthesis [10, 37, 38, 47, 48, 128, 157, 173], in spite of its inherent difficulty for such task⁴. VHDL is generally considered difficult to synthesise because there is no clear guideline for synthesis input [95]. Examples of such difficulties include semantics such as '*a* \Leftarrow *b* after 10ns, *c* after 20ns;' and language constructs, such as *assert* statements and *access* types [37]. Although solutions have been proposed [37], they usually tend to restrict the language to a specific subset, and therefore defeat the idea of a standard language. Nevertheless, these problems are currently being addressed by the VHDL standard group, and it is expected that these issues will be resolved by the new definition [3].

Several other HDLs exist, such as UDL/I⁵, Verilog, and Ella [45], which, as well as VHDL are directed to hardware designers. Some research projects have adopted well-known software languages, such as C [5, 46], Pascal [148, 184], and ADA [62, 70], and extended them to include hardware constructs, therefore converting them into HDLs. Two examples include HardwareC [5] and Flamel [184]. Another approach is to use these software languages without transforming them into HDLs. In this case, no structural or control information are present in the user's specification. Therefore, hardware synthesis becomes a greater problem, since several hardware constraints are omitted, leaving the synthesis task with a wide range of options. Some specific application domains require this approach, such as the Cathedral-II [108] silicon compiler along with its language SILAGE, the *Galatea* system [20], and the *NSC* developed in this thesis, which adopts a C-like input language. For these systems, in particular the last two, the hardware details should be completely omitted from the user, as discussed in chapter 1.

²VHDL stands for VHSIC Hardware Description Language; where VHSIC is the Very High Speed Integrated Circuits Program Office of the American Department of Defense.

³EDIF stands for Electronic Design Interchange Format.

⁴VHDL has been originally directed at meeting mixed level simulation requirements only.

⁵In Japan, VHDL has not been adopted. Instead, UDL/I, Unified Design Language for Integrated Circuits, has been standardised by Japanese industries [7] to be a logic synthesis oriented design language [95].

3.2.2. Internal Representation

The textual form of the input description is usually translated into an internal representation, which is more powerful for expressing the data and control flow, and more suitable for computer manipulation. Two types of representations are generally used, namely *parse trees* and *graphs*, upon which synthesis algorithms are applied. The vast majority of high level synthesis systems use graphs; examples include the Yorktown Intermediate Format (YIF) [30], Value Trace (VT) [113], Irvine's Behavioural Intermediate Format [50], and IBM's Sequential Synthesis In-core Model (SSIM) [33].

The role of an intermediate representation is to describe the design's data and control flow in a suitable form, in which algorithms can be applied to extract useful information. Several types of graph representation have been proposed [136]: precedence graph, control flow graph (CFG), data flow graph (DFG), control and data flow graph (CDFG), and Petri-nets.

Precedence graphs specify the dependence among operations through the precedence relations determined by the order they are executed. It is a digraph [40] composed of a set of vertices and a set of arcs. Each vertex contains one or more operations. An arc A from $V1$ to $V2$ indicates that $V1$ is an immediate predecessor of $V2$. The control flow and data flow information are obtained by reading the whole graph from vertex to vertex. The YIF is an example of a system using precedence graphs.

Data flow graphs (DFG) specify data dependencies. Its vertices represent the data (variables, constants, and signals) and operations, while its arcs represent communication paths that carry values between nodes, indicating the direction of the data flow (Figure 3.2a). Control flow graphs (CFG) do not imply data dependency; it only describes the sequence in which operations are executed. The nodes correspond to the operations, and the arcs link immediate predecessor-successor pairs, in which conditional branching is indicated by more than one successor (Figure 3.2b). Several design automation systems use a form of DFG and CFG formats. Some systems use both formats (CDFG), in which either separated graphs for data and control are used or a single graph unifying control and data flow information are employed (Figure 3.2c). The Value Trace is an example of such approach [113]. Figure 3.2 illustrates these three types of flow graph representation.

Finally, some systems use Petri nets, in which transitions represent actions (or operations) that are executed (or fired) whenever certain conditions are true, which are denoted by markings of places [148].

The choice of an intermediate format is usually based upon the type of the input description and, what is more important, upon the synthesis algorithms. For example, Petri-nets, precedence graphs, and control flow graphs are useful for the determination of control states, while data flow graphs are more suitable for the synthesis of data path structures.

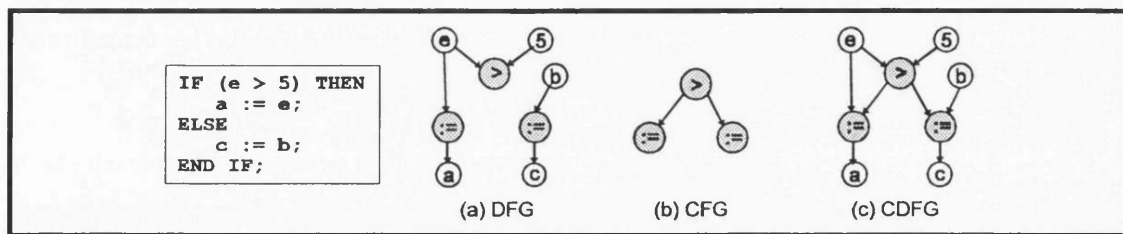


Figure 3.2 — Examples of Control and Data Flow Graphs

3.2.3. Target Architecture

The automatic synthesis of a hardware structure from an abstract high level description involves a search through a large design space, since there is no single mapping from one behaviour to the structure that implements it. Therefore, an exhaustive search for the optimal implementation is required. This is clearly a very difficult and extremely time consuming task. Several approaches have been proposed for this task [114] but with limited practical success [162] for general-purpose systems. Conversely, some special-purpose systems, in particular those for digital signal processing (DSP) applications, have reported good results [188]. This is mainly because the special-purpose approach restricts the design space by limiting the architectural model (or target architecture).

There are several possible architectures for each application field. Silicon compilers have been typically developed to design processors and DSP circuits. For these applications the target architecture is fixed by choosing, for example, the clocking scheme, the type of communication (synchronous or asynchronous), the type of controller (finite state machines, or microcode), and the type of data path (serial, parallel, pipelined, etc.).

The definition of a target architecture also permits the synthesis of optimised circuits, which can potentially match the same quality of a design produced manually. This is a very important issue, in particular, when neural chips are to be generated through silicon compilation. By choosing an appropriate target architecture, very dense circuits can be achieved. Consequently, several processing elements can be integrated into a single chip, permitting the construction of large neural networks in hardware.

3.2.4. High Level Synthesis

High level synthesis is the process of transforming an abstract behavioural specification of a digital system into an interconnected set of RTL components, such as ALUs, registers, memories, multiplexers, and busses. The task of finding a structure that implements the behaviour is usually achieved by satisfying a set of constraints and goals. High level synthesis is the front-end of the silicon compilation process, being crucial for producing efficient designs. Its output, the RTL specification, can be further transformed by logic and layout synthesis systems to produce the final chip.

After the input specification has been compiled into an intermediate representation, the core of transforming behaviour into structure is accomplished by two tasks: data path and control synthesis. Upon this phase, the following major tasks can be identified:

- *Hardware allocation*, also called resource allocation or simply allocation, involves the definition of execution units, storage units, and interconnection units. The allocation of these units is generally accomplished by an algorithm that takes into consideration time and area constraints.
- *Hardware assignment*, which binds operations to specific hardware units, such as ALUs, registers, and busses.
- *Scheduling*, which consists in assigning operations to the so-called control steps⁶. It determines the exact time slot each operation will be executed, as well as the composition and contents of the controller.
- *Hardware sequencing*, which generates a sequencer that uses control signals for commanding the actions in the data path.

The main tasks involved in high level synthesis are examined in section 3.3.

3.2.5. Logic Synthesis

Logic synthesis fits between the RTL specification and the netlist of gates' specification [31]. It takes a structural design, with an interconnected set of RTL components, and generates an optimised combinational logic, which is mapped onto a library of available cells in a specific technology. Logic synthesis is commonly used for the synthesis of control parts, since it applies only to the logic extracted from the RTL

⁶A control step is the fundamental sequencing unit in synchronous systems, corresponding to a clock cycle. It is equivalent to a state in a control finite state machine, or microprogram step in a microprogrammed controller.

specification. Therefore, memory blocks, ALUs, registers, etc., are simply set aside with their inputs and outputs being outputs and inputs, respectively, to combinational logic blocks. Two approaches are used: two-level (PLA-like) logic synthesis; and more recently multi-level logic, also called random logic, synthesis [31].

Two-level logic synthesis has dominated the arena in the early 80's [29, 31, 158]. It has been focused on the automatic generation of PLA structures [54, 118]. Since PLAs are among the most popular structures for implementing logic functions in two-level form [134], they are today vastly used in commercial microprocessors. The synthesis basically concerns with minimising the number of PLA product terms through two-level minimisation steps, aiming at faster and smaller circuits. Although efficient optimisation algorithms have been devised, two-level logic is not best suited to handling multiple blocks of combinational logic [31].

Multi-level logic synthesis started in the late 70's and became mature in the late 80's, when results achieved the same level of advancement as for two-level logic synthesis [29, 31]. Today, several commercial CAD companies successfully adopt the methods and algorithms developed during that period [32]. Multi-level logic is a general case that includes two-level logic. Its main goals are: (i) to minimise the area occupied by the logic equations (measured as a function of the number of gates and transistors), while simultaneously satisfying the timing constraints; and (ii) to maximise the testability of the synthesised logic [31].

3.2.6. Layout Synthesis

Layout synthesis tools are the most common [102, 139]. The problems corresponding to physical design are well understood, even though they remain difficult from a computational standpoint [102]. Layout synthesis takes the netlist of interconnected cells as input and creates the actual physical geometry (layout) of the design. Several design systems are commercially available [2, 117], and nowadays research is focusing more on the problems resulting from emerging design methodologies, for example, field programmable gate arrays (FPGAs) [162].

3.3. Anatomy of High Level Synthesis

The main steps involved in a high level synthesis system can be summarised as:

- *High level transformations* of the HDL into an internal representation, usually a CDFG, and further transformations (and optimisations) of the internal representation into a more suitable form towards hardware synthesis;

- *Data path synthesis*, comprising two major tasks: *allocation* and *scheduling*. The first assigns each operation to a piece of hardware. This involves the selection of the type and quantity of hardware modules from a library (called module generation) and the mapping of each operation to the selected hardware module. The second assigns each operation defined in the graph to a control step;
- *Control synthesis*, which generates a controller based on the scheduling performed previously.

3.3.1. High Level Transformations

The main goal of high level transformations is to optimise the behaviour of the design in such a way as to produce a more suitable specification for the synthesis of the hardware structure. Most of the transformations are software-like optimisations [11], such as constant propagation (where constant expressions are evaluated at compile time), dead code elimination (where pieces of code not reachable by the program flow are eliminated), and common sub-expression elimination (where sub-expressions are identified and their result stored for later reference). Other transformations are more specific to the hardware synthesis, and they include:

- Hardware-specific local optimisations, which transform some specific operations into faster and smaller structures. An example includes the division or multiplication of a number by another that is a power of 2. In this case, shifting operations produce faster results.
- *In-line expansion* of subroutines, which unlike software compilers increase the optimisation possibilities for hardware synthesis, since some of the control steps in the subroutine can be interleaved with the surrounding statements. Furthermore, the *call* and *return* instructions are eliminated.
- *Select combination*, which permits the combination of successive *if* or *case* statements into one *case* statement. This eliminates all control steps necessary to make sequential decision and implements them in a single step. An exception to this rule occurs when values in one branch of the selection statements are dependent upon values calculated by previous branches of the same selection.
- Transformation of some specific complex data structures into simpler ones for better hardware manipulation. An example includes the sum of products operation, present in the majority of neural models, in which two blocks of memory are the input data. In this case, it is very common to specify in the input language only one block in which

the data is put in sequence and a pointer is incremented each time a data is read. For hardware purposes, it is better to have two separated blocks of memory in which data can be read simultaneously. This is done in the high level synthesis developed in this work and is explained in chapter 7.

- Introduction of concurrent processes, which may be synthesised as concurrent hardware, leading to faster designs.
- Reducing the number of levels in the graphs, such as the height reduction algorithm called *level compression* in Flamel [184], which may lead to faster hardware.

The design of a digital circuit usually involves the definition of a data path structure and a control unit that commands data transfers. Data path structures are simply a network of registers, functional units, multiplexers and busses. The control unit can be either a microcode, PLA, or random logic. The synthesis of these two hardware structures is discussed in the next two sub-sections.

3.3.2. Data Path Synthesis

Data path synthesis consists of two basic tasks: scheduling and resource allocation (binding) [46, 176]. Scheduling involves the determination of which operations are executed in which control states. Resource allocation consists of: mapping all variables and operators specified in the input description into storage elements and functional units, respectively; defining the interconnection resources; and optimising all created resources.

Scheduling and allocation activities are interdependent tasks. According to the amount of hardware allocated, different scheduling possibilities arise. Conversely, for a given schedule strategy, different pieces of hardware would be necessary to match timing requirements. For example, two operations can be scheduled in the same control step only if they do not use common hardware resources; similarly, the decision about the number of functional units to be used depends on what operations are done in parallel, which is determined by the scheduling strategy. Therefore, the order in which these algorithms are executed is very important. Since allocation aims at minimising area and scheduling attempts to minimise time delay, a trade-off between the two algorithms must be achieved for each particular case, depending upon the main aspects to be optimised.

Scheduling Algorithms

The task of scheduling is to assign operations to specific clock cycles (in synchronous systems). The order in which specific operations are accomplished is obtained

from the input description. The goal is to optimise the number of clock cycles in order to maximise speed given certain limits on hardware resources and cycle time.

Scheduling algorithms must respect hardware constraints. For synchronous designs, every component can be used only once during a specific control step. Therefore, registers can be loaded only once, combinational logic may evaluate only once (no feedback), and busses may carry only one value.

The approaches developed so far include: *As-Soon-As-Possible (ASAP) scheduling*, *As-Late-As-Possible (ALAP)*, *list scheduling*, *force-directed scheduling*, and *path-based scheduling* [198].

The **ASAP scheduling** assigns operations to the earliest possible control step by sorting the data-flow graph topologically, according to its data dependencies using a depth-first search [164]. This means that an operation is performed as fast as its inputs are available. Conversely, **ALAP scheduling** assigns all operations to the latest possible control step. To illustrate how these two algorithms work, Figure 3.3 shows a CFG after ASAP and ALAP algorithms are employed for the given specification [198]. Note that the ALAP algorithm resulted in the possibility of merging operations 5 and 6 in a single execution unit, thus saving silicon area. The *mobility* is thus defined as the difference between the two approaches, meaning that each operation with a mobility greater than one (operations 3, 4, 9, and 10 in the example) can be scheduled in one of the control steps comprised by its mobility.

Examples of systems performing ASAP scheduling include Facet [185], DAA [100], Bridge [186], HARP [175], and Flamel [184]. ALAP algorithm is generally used in conjunction with ASAP to calculate the mobility of operations, so that more sophisticated algorithms can be applied.

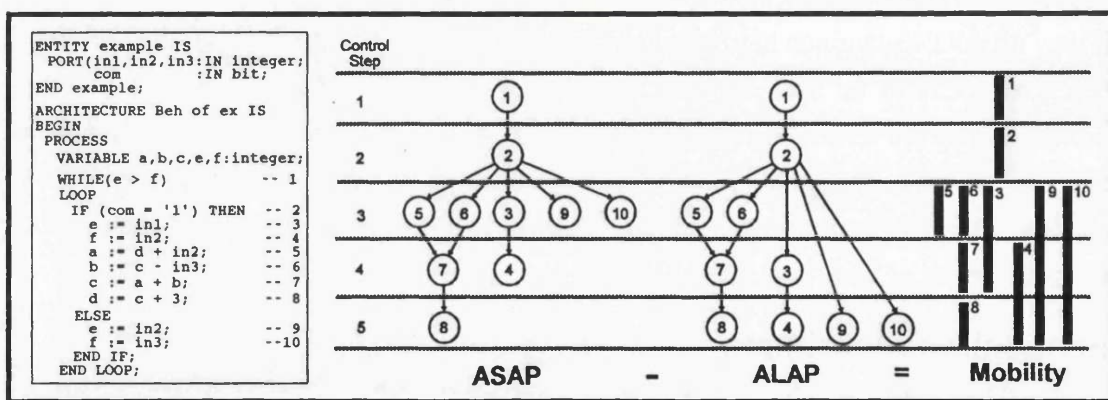


Figure 3.3 — ASAP Schedule, ALAP Schedule, and Mobility

In ASAP and ALAP algorithms, the decision of the next operation to be scheduled is made locally, which generally leads to sub-optimal designs. *List scheduling* overcomes this problem by using a more global criterion for selecting the next operation [198]. It creates a list of available operations to be scheduled, and sorts the list following some priority function, such as the length of the longest path. Thus, operations on longer paths are scheduled earlier than operations on shorter paths, since longer paths require more control steps. For each control step a list of operations is constructed, containing operators in which inputs are produced in earlier control steps and therefore do not violate any resource constraints. Then, the highest priority operator is placed into the current control step, the list is updated, and the process continues until no more operators can be placed into that control step. This process is repeated for each control step, until the entire design is scheduled. Several systems use list scheduling, such as MAHA [145], Slicer [142], and Cathedral-II [108]. They differ from each other depending upon the adopted priority function.

Forced-directed scheduling employs a different mechanism for deciding the next operation to be scheduled. It uses the so-called *force* on each operation, which measures the number of operations of the same type that can be scheduled on the same control step. By minimising the force, the algorithm tends to balance the use of functional units across the data flow graph, producing a schedule that minimises the number of resources needed to meet a given time constraint. To accomplish this, a *distribution graph* is constructed for each set of operations that could share a functional unit, which shows how loaded the control step is. The mobility of each operation is used to determine the distribution graph. Once all the forces have been calculated, the operation-control step pair with the least positive force is scheduled. The advantage of this algorithm is that it tends to balance the distribution graph, so that a minimum amount of hardware is used to achieve a given time constraint. HAL [147] adopts such strategy.

Lastly, *Path-based scheduling* focuses on the conditional branching of control flow graphs by independently scheduling each path (using one of the above algorithms) to allow sharing of hardware. YSC [30] uses a heuristic path-based scheduling, while HIS [36] uses an *AFAP* (*as fast as possible*) path-based scheduling, described later in section 3.4.3.

Allocation Algorithms

The main goal of allocation algorithms is to share hardware units, so that the whole design is optimised. This means that operations share functional units, variables are mapped onto common registers and memories, and data transfers share busses and

multiplexers. The problem of allocating hardware structures to each operation, variable, and communication path falls into three parts:

- *Generation of storage elements*, such as register, signal, up/down counter, RAM, ROM, and I/O ports);
- *Generation of functional operators*, such as adder, subtracter, logical *and*, logical *or*, comparator, etc.; and
- *Network generation*, which attempts to gather several source signals into one destination, which can be either a multiplexer or a uni-directional bus. If more than one destination is desired, bi-directional busses are then required.

There are basically three approaches for solving the allocation problem [198]: *heuristic*, *linear programming*, and *graph-based*.

Heuristic approaches usually select one element (variable or operation) at a time according to specific criterion. This may involve the use of a global cost function, which chooses the assignment that saves the most hardware, or the use of a data-flow order that results in the lowest cost. Examples of systems employing such technique include DAA [100], HAL [147] and Olympus [121].

Linear programming approaches formulate the problem from a mathematical standpoint and try to solve it using linear programming techniques. This was first proposed in 1983 [71], but due to lack of computational resources the method produced reasonable results only in small designs. Today, the technique has reappeared in some systems, solving several allocation problems [26, 60, 104, 143].

Finally, *graph-based approaches* formulate the problem as a *clique* partitioning problem. A clique is defined as a complete sub-graph, where each node is connected to every other by a direct arc. The problem is to find a minimum set of cliques in the entire graph that every node is in one and only one clique. Tseng and Siewiorek were the first to transport this problem to allocation [185]. In this case, an arc indicates that two connected nodes do not share a common control step, thus being candidates for sharing resources. By finding the minimal number of cliques, by analogy, the minimum number of physical operators is achieved. Bridge [186] and Facet [185] are both examples of systems employing such a technique.

3.3.3. Control Synthesis

The task of generating a controller for the synthesised data path is directly related to the scheduling strategy. Although some authors categorise scheduling as part of the control synthesis [198], rather than an integral part of the data path synthesis, scheduling and control synthesis can be distinguished from each other quite dramatically [152].

Control synthesis is the creation of a controller in the form of a finite state machine, which commands the sequence of the operations executed on the data path by providing control signals to the data path modules [59]. This is the last step in the synthesis process, and involves the definition of a controller architecture and the instantiation of the controller components' parameters. The finite state machine can be implemented in various ways, such as random logic, PLA, or microprogram (using a microcode ROM with program counter) [77, 97, 163].

3.4. Taxonomy of Silicon Compilers

According to the characteristics presented by silicon compilers employing high level synthesis, Table 3.2 presents a simple taxonomy for some well-known systems. This includes the characteristics presented in section 3.2 and high level synthesis algorithms examined in section 3.3. The remaining of this section reviews some of these silicon compiler systems and classifies them according to the aimed applications, input description, internal representation, target architecture, and type of synthesis algorithms employed.

High Level Synthesis System	Primary Applications	Input Description	Internal Representation	Target Architecture	Synthesis Algorithms
<i>Olympus</i>	General-Purpose	HardwareC	SIF	✕	Iterative graph-based scheduling
<i>Yorktown</i>	General-Purpose	V	YIF	✕	Heuristic path-based scheduling
<i>HIS</i>	General-Purpose	VHDL	SSIM	✕	AFAP path-based scheduling
<i>Cathedral-II</i>	DSP Chips	SILAGE	SILAGE primitives	Bit-parallel processors for DSP applications	List Scheduling, Bus Merging
<i>Galatea</i>	Neural Chips	N	Internal Tables	INPG	Folding of virtual neurons into PEs
<i>UCL's NSC</i>	Neural Chips	<i>nC</i>	<i>ICR</i>	<i>Generic Neuron</i>	<i>Default mode and User-driven mode</i>

Table 3.2 — Simple Taxonomy of High Level Synthesis Systems

3.4.1. *Olympus* System

The *Olympus* system [121] is a general-purpose integrated set of tools for synthesis of digital circuits, supporting behavioural, structural, and logic syntheses with timing constraints. The output of the design is given as a netlist of gates or compiled macrocells, with no support for placement and routing tools. For these non-supported tools, interface to standard physical design tools is provided.

The input to the *Olympus* system is *HardwareC* [5], an extension of the C programming language [96], with features to support hardware synthesis. At the behavioural level, the language includes: support for synchronisation and communication between different processes; modelling parallelism; and timing constraints' specification. At the structural level, the language permits the definition of memory modules, parameter classes (*in*, *out*, *inout*), and *architectural registers* that should be implemented as a register during data path synthesis.

High level synthesis is performed by *Hercules* and *Hebe* [121]. Initially, parse trees are used for the high level transformations. An internal representation, called *Sequencing Intermediate Form* (SIF), is used for scheduling and allocation. SIF is represented as a directed acyclic graph with operations being the vertices and the arcs representing the predecessor-successor relationships (which are subject to the data flow dependencies between operations).

The high level synthesis starts with *Hercules*. The input description is parsed and translated into an abstract syntax tree representation. The parse tree abstraction provides the underlying model for semantic analysis and optimisations similar to those used in optimising compilers, including variable and constant propagation, dead-code elimination, and data-flow analysis. In addition, *Hercules* includes a mechanism for hardware-oriented optimisations, called *reference stack* [119]. This mechanism provides information to structural synthesis, resulting in a minimised number of registers needed to implement the functionality of the input description. In addition, control steps needed to implement alignments to variables are eliminated.

Hebe takes the *SIF* representation generated by *Hercules* and implements the structural synthesis through allocation and scheduling algorithms. *Hebe's* strategy for data path synthesis is to first perform hardware allocation followed by scheduling. It binds operations to specific resources, and then performs scheduling that satisfies the timing constraints. This process is repeated for different possible binding alternatives.

Control synthesis is implemented as an interconnection of Moore type finite state machines, one for each state vertex of the control graph. Each FSM may be designed either as level-sensitive or as edge-sensitive registers. Control synthesis consists of two tasks. The first is the *sequencing control* that is responsible for preserving the sequencing behaviour from the SIF. The goal is to capture the behaviour of the structure in a minimal number of states and transitions between states. The second is the *constraint control* that deals with the constraints imposed by real hardware systems. The goal is to find an optimal cycle time based on both design and timing considerations.

In summary, *Olympus* is an open system, and therefore not restricted to specific applications. It supports only synchronous logic implementation, with no provisions for the synthesis of pipeline structures or multi-phase synchronous logic. Consequently, the quality of the generated circuit is generally poorer than the one created manually, occupying larger silicon area. Nonetheless, it is a flexible system, comprising several tools ranging from high level to low level synthesis, which are in continuous development.

3.4.2. Yorktown System

The YSC (Yorktown Silicon Compiler) [30] is also a general-purpose system. Its synthesis system starts from *V* descriptions, a Pascal-like language, which includes processes, asynchronous calls, and queues.

High level transformations convert *V* into the Yorktown Internal Format (YIF), a control and data flow graph, through a series of software-like compilations. In the data-flow graph, the vertices represent operations and variables, while the arcs represent data dependencies. In the control-flow graph, the nodes represent operations, and the arcs represent predecessor-successor relationships.

Scheduling is first performed by a control state assignment, which minimises the number of control states. The algorithm starts with a single control state. Additional states are added to permit modelling loops and procedure calls (called *module call*) [34]. States are also added according to timing constraints and data flow restrictions. States can be further split or folded to meet constraints.

Allocation is performed by initially assigning each operation to a combinational function unit, and each variable to a register, producing a block of combinational logic for each procedure. Then, for each block, a clique partitioning algorithm is used to accomplish two functions: to fold together functional units performing the same operation in different time steps or branch alternatives; and to fold together registers as guided by lifetime analysis.

3.4.3. *HIS* System

The *HIS* (High level IBM Synthesis) [36] is another general-purpose system, and has its roots in the *YSC*. It uses VHDL as the input language, but can also read the YIF. An intermediate representation, called SSIM (Sequential Synthesis In-core Model) is used, which represents control flow and data flow separately, although links between them are explicitly kept. SSIM is hierarchical in that it can represent several modules at once, each being synthesised separately.

Scheduling is first performed by an *As-Fast-As-Possible* (AFAP) algorithm on each possible path in the CFG. AFAP scheduling attempts to find all possible execution paths (defined by conditional branches) by searching the control-flow graph. Timing and resource constraints are expressed as intervals where the paths must be cut to separate the operations into states. Clique partitioning is then used on each path to find the minimum number of cuts that meet the constraints, which results in the minimum number of control steps for each path. This scheduling emphasises conditional branching rather than potential parallelism like in the list scheduling and force-directed scheduling.

Data path allocation is very similar to the one used in the *YSC*. It falls into the category of allocation algorithms based on clique covering. First, a complete initial data path is generated by allocating functional units and registers, defining interconnections (multiplexers) between these functional units and registers, and finally deriving the control signals for the allocated hardware. In a second step, the initial data path is optimised based on clique covering and colouring approaches, which consists in merging functional units, registers, and multiplexers.

Control synthesis is done immediately after scheduling, and consists in constructing a control FSM.

3.4.4. *Cathedral* System

The *Cathedral-II* [108] is a system for synthesising digital signal processing (DSP) integrated circuits. The synthesis process translates a behavioural, flow-graph algorithm description expressed in SILAGE, a Pascal-like language, into a dedicated multi-processor architecture.

SILAGE is a language specialised for high level description of signal processing algorithms. While the basic object in SILAGE is the signal, a vector where components are samples in the time domain from infinity to actual time, the basic operation is a functional application of the signals.

The synthesis process is strongly based upon the target architecture, which allows considerable pruning of the search space and permits the use of dedicated optimised techniques to exploit the architectural properties. The intermediate representation is simply a set of SILAGE primitives, which are the results of parsing SILAGE and performing syntax and semantic analysis that determine the data types of all signals.

Data path synthesis, called *Jack-the-Mapper*, first selects a number of executing units (EXUs) from a library according to the input specification. Then, dedicated bus connections between different EXUs within a single processor are allocated. Next, a partially rule-based and partially algorithmic translator assigns each operation to a particular type of EXU, assigns variables to register files, and generates a dedicated bus for each variable. Scheduling is done by list scheduling algorithm with priority to operations on the longer critical paths.

3.4.5. Galatea System

In the *Galatea* project, neural chips are automatically generated through a set of tools that comprise: the compilation of the neural network language *N* into the intermediate language *VML*; the application of a folding algorithm that attempts to merge *virtual neurons* onto physical neural processors [129, 190]; and the generation of a VHDL description of the neural chip. Although the *Galatea* system uses some ideas developed in the *NSC*, it is not qualified as a silicon compiler, since no compilation nor synthesis is done; it is best classified as a *high level module generator*. However, the *Galatea* tool is included in this discussion for purposes of comparison with the *NSC* system, described in the next sub-section.

The approach adopted in the *Galatea* project is limited in two basic issues: (i) it supports only the recall phase of neural algorithms, that is, on-chip learning is not supported by the generated circuits; (ii) it aims to generate optimal chips for a specific application — the OCR (Optical Character Recognition). The compilation of *N* into *VML* produces several tables, which specify the network's topology, connectivity, and size. A separate tool, called *type simulator* [190], provides hardware-specific parameters, which are obtained from a simulation of the neural network application.

An interesting investigation in the *Galatea* system concerns the mapping of virtual neurons (artificial neurons) into physical neurons (the PEs). This idea comes from the fact that large networks cannot be integrated into a reasonable number of chips. Therefore, in some applications, each PE should implement more than one neuron. The exact specification of which neurons each PE realises is performed by the folding computation [129]. This is done by a rule-based mechanism that examines the structure of

the neural network and applies special folding procedures. The disadvantage of this strategy is that parallelism is badly affected.

The *folded* network is then described by an intermediate format that fully specifies the physical processor by a data structure that defines the number of virtual neurons, the data precision, and the parameters specific to the neural computation. This format is submitted to the hardware generator tool, which maps the description onto the specific target architecture, the INPG's architecture described in the section 2.3.1.

The *Galatea* project is still under development and will end by March 1994. Although the adopted approach limits the number of neural network applications allowed, designs very close to manual ones are expected to be achieved.

3.4.6. UCL's Neural Silicon Compiler

The UCL's *NSC* synthesis system, developed in this work, is more ambitious than the *Galatea* system. The goal is to produce neural chips with on-chip learning capability, supporting the majority of the neural models known. The high level synthesis starts from the neural network language *nC* [191].

Several transformation steps are performed at the high level input. Because *nC* is not an HDL, the data structures are not suitable for immediate hardware synthesis. The transformations performed in *nC* include: transformations upon the data structures, which allow better manipulation by the hardware; and hardware-specific optimisations that attempt to eliminate extra nodes. Transformations are first carried out upon the parse trees, which are then translated into a graph-based intermediate format (a CDFG), namely intermediate code representation — *ICR*. Further transformations are employed, this time upon the *ICR*, before data path and control synthesis are applied.

Scheduling and allocation algorithms are simultaneously executed on the graph-based intermediate representation. This means that rather than employing first scheduling for the whole design followed by hardware allocation, or vice-versa, at each iteration of the algorithm, both mechanisms are employed. However, according to the guideline implicitly specified by the user, the approach focuses either on allocation or scheduling. If constraints, such as maximum time to execute certain tasks, are specified, scheduling is first executed, followed by the allocation for that particular scheduling. Conversely, if nothing has been defined at the *nC* level, then allocation takes place first, because the aim is to optimise primarily the VLSI area, so that several neurons can be integrated into the same chip. In this case, ASAP scheduling is used, since this algorithm maximises speed.

Then, ALAP can be next employed to check whether some resource sharing is possible without increasing the control steps.

After data path synthesis is accomplished, the control synthesis takes place by constructing a finite state machine, which implements the exact scheduling strategy created previously.

According to the hardware obtained by the synthesis steps previously described, a module generation phase follows. It generates multi-port registers (according to the necessity), memory modules (RAM and ROM), and functional units (selected from a library containing ALUs, multipliers, a parameterised set of multi-port registers, comparators, tri-state buffers, etc.). At this stage a VHDL description is generated completing the high level synthesis of neural chips.

The *NSC* has certain similarities with some of the systems described previously. It is directed to the generation of ASNNCs as in the *Galatea* system. It restricts the target architecture based upon a suitable choice for executing neural computation, in the same way that the *Cathedral-II* system defines a suitable architecture for DSP chips. Finally, it uses a C-based high level language for its input, as in the *Olympus* system.

The algorithms for hardware synthesis are similar to the ones described previously in this chapter. The major distinction between the *NSC* and other systems is that it is targeted to be used by non-expert in hardware design, which consequently qualifies the input language as primarily behavioural. The hardware synthesis is performed considering constraints imposed by the *Generic Neuron* architecture [190], and optionally by high level *hints* defined by the user, which consist of limiting determined neural function in time, or limiting the number of neural processing elements in each chip.

3.5. Summary

This chapter has presented a review of silicon compilers that employ high level synthesis. The characteristics of such systems have been examined, constituting the background for the *NSC* developed in chapter 7 of this thesis. Based upon these characteristics, a taxonomy of high level synthesis systems has been presented, and their approaches have been discussed.

Two types of silicon compilers have been identified: general-purpose and special-purpose. While general-purpose approaches can generate a circuit that realises any function described in the input language, special-purpose can only generate application-specific circuits. Usually, general-purpose silicon compilers try to synthesise a

circuit with no fixed target architecture; just a few issues are restricted, such as synchronous or asynchronous systems, pipeline or non-pipeline, and clocking scheme. Conversely, special-purpose approaches use a pre-defined target architecture, which is generally optimised for the particular application. For this reason, general-purpose systems tend to produce circuits that do not comply with the microelectronics industry standards in terms of silicon area, while special-purpose systems tend to yield reasonable circuits.

The next chapter presents an overview of the *Pygmalion* programming environment and its input language *nC*. The *Pygmalion* system is the adopted platform for the development of the *NSC*.

Chapter 4

nC Neural Network Specification Language

This chapter describes the nC neural network specification language, the core of the Esprit II Pygmalion neural programming environment. After introducing the Pygmalion environment, the nC language is fully described and discussed. The chapter concludes presenting the hardware specific extensions to nC, proposed and implemented in this work.

4.1. Pygmalion Programming Environment

Chapter 2 presented the requirements and current trends for developing neural network programming environments. The *Pygmalion* environment follows these requirements, providing a sophisticated and powerful tool for programming and simulating neural networks in different platforms.

The *Pygmalion* project [19, 21, 24, 181], sponsored by the European Community ESPRIT Programme, has been developed with three basic objectives: firstly, to develop a European *standard* general tool that includes the same facilities and functionality found in commercial systems; secondly, to provide an *open* environment, which can be easily extended and interfaced to other tools; and finally, to provide *portability* for neural network applications, so that trained and partially trained networks can be easily moved from machine to machine. To meet the first two objectives, the software environment is based on X-Windows, C, and C++, while portability is achieved by defining an intermediate level network specification as a subset of C.

The *Pygmalion* environment comprises five major parts, as shown in Figure 4.1:

- **Graphic Monitor** — the graphical software environment, based on X-Windows, for controlling the execution and monitoring the simulation of a neural network application;
- **Algorithm Library** — the parameterised library of common neural models, written in the high level language (*N*), that provides the user with a number of validated modules for constructing applications;

- **High Level Language *N*** — the object-oriented programming language that defines, along with the algorithm library, a neural network algorithm and application by describing the network topology and its dynamics;
- **Intermediate Level Language *nC*** — the machine-independent network specification language that represents the partially or fully trained neural network applications; and
- **Compilers** — utilised to translate the user-specified neural network application into the target *UNIX*-based workstations and parallel Transputer-based machines.

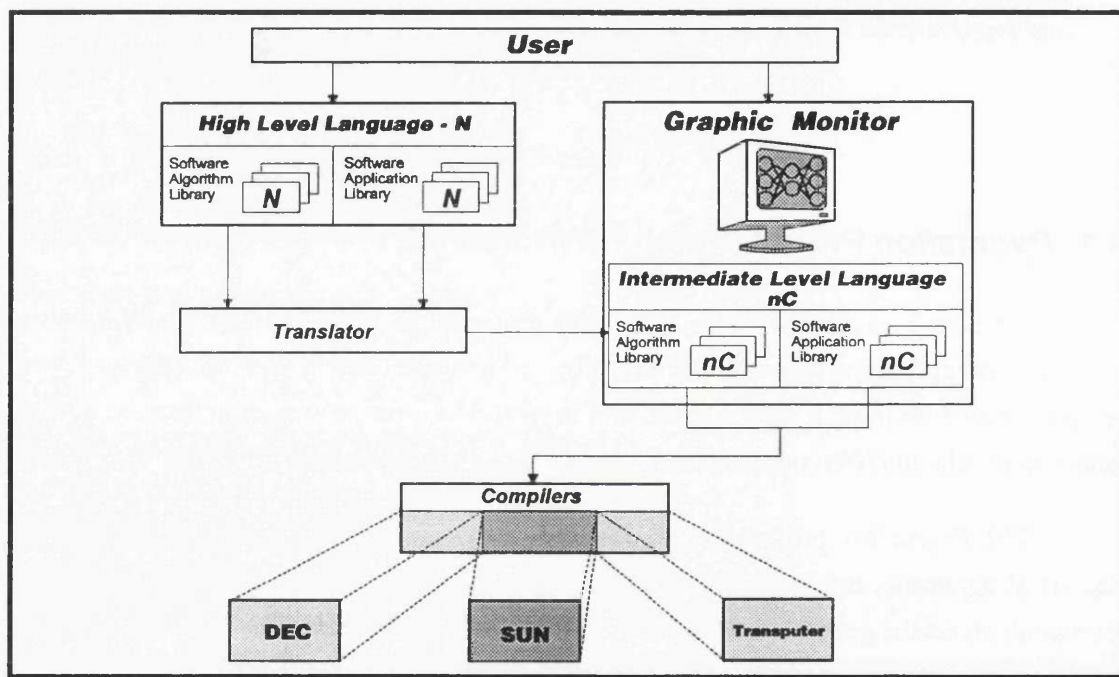


Figure 4.1 — *Pygmalion* Neural Programming Environment

The **Graphic Monitor** executes on a host computer and, through the *X-Windows* graphic tools and a command language, monitors the execution of the application on a target machine. Pull-down menus are provided to: select and change the I/O format; the network architecture, the network learning algorithm, the network training and execution; and to display neurons' states and synaptic weights.

The **Algorithm Library** contains the classical neural network models in a parameterised specification that can be configured for a particular user application. It includes popular algorithms such as Hopfield [86], Back Propagation [161], and Boltzmann Machine [9] where the interconnection geometry and their associated functions are already specified. The user configures the required application by utilising these

algorithms and selecting the number of processing elements, their initial state and weight values, learning rates, and time constants.

The *High Level Language N* has been designed to allow its usage by both naive and expert users of neural networks. Its syntax is a subset of C++ augmented with additional neural-oriented features. *N* follows the object-oriented methodology, which permits a neural network algorithm to be specified in two ways: by defining specific types having their own data and functionality, in analogy to a class in C++; and by assembling pre-defined types in a modular tree hierarchy [110].

The *nC intermediate level language* and the *Graphic Monitor* are closely related. The *nC* language, described in the next section, is a subset of C, in which only a limited set of statements is allowed and a very specific data structure is defined. Algorithms and applications are programmed using this restricted set of C statements and the pre-defined network-specific data structure, called **system**. This data structure represents a hierarchy of five basic levels: networks, layers, clusters, neurons, and synapses. The *Graphic Monitor* uses this structure to display a hierarchy of windows, each corresponding to a particular level in the **system** data structure.

Compilers can be easily used since both *N* and *nC* are based upon *standard* programming languages, C++ and C, respectively. In addition, due to the pragmatic approach adopted by the *Pygmalion* project, a neural network structure and algorithm written in *N* can be translated into an equivalent *nC* structure, thus generating the *nC* version of any *N* program. From *nC*, standard C compilers can be used to generate binary code for specific target machines.

To ensure uniformity and consistency, all major parts of the *Pygmalion* environment are centred on a common interface, which conforms with the following properties: (i) all components are based on the *nC* hierarchical structure **system**; (ii) all algorithms are parameterised; (iii) algorithms and applications are independently specified; and (iv) algorithms and applications are interfaced through common (*nC*) data structures, function names, and system variables.

After this brief introduction, the next section describes in detail the *Pygmalion*'s core component — the *nC* intermediate level language.

4.2. nC Network Specification Language

In the *Pygmalion* environment, the *nC* programming language acts as the intermediate-level, machine-independent representation for neural networks. It has been developed based on requirements that provide:

- a general framework, for easy definition of any neural network model with arbitrary functionality, topology, connectivity, and parameters;
- separate description of algorithms and applications; and
- machine independence, in which a neural network specified in *nC* can be executed in a variety of target machines.

Generality is obtained by defining the **system** hierarchical data structure. This structure contains all the necessary information to describe a particular neural network, including the network topology, the data of the system (neuron's status and synaptic weights), the functionality of each processing element (PE), the connectivity among PEs, and the controllability of the network activities.

The independence between algorithms and applications is obtained through the definition of two different levels of programming. Each algorithm is parameterised, so that it can be configured to implement different applications. Conversely, the application level contains information about the network configuration and the algorithm control. The separation of algorithm and application expertise is very important for the development of generic applications and algorithms in a concise way.

The machine independence feature is achieved by making *nC* based on the C language.

4.2.1. Basic Concepts

The *nC* language centres upon the definition of some basic concepts that are closely related to neural network algorithms [191]. These are: the **system** hierarchical data structure, that groups into one structure all neural network information; and the **rule** concept, that embodies the functionality and controllability of neural network models.

Hierarchical Data Structure — **system**

The core of the *nC* language is the **system** data structure. This generic structure defines the complete configuration of any neural network model, giving the explicit location of each synapse and neuron inside the whole network. This structure is fixed and

cannot be modified by users, since it represents the common interface by which the graphic monitor controls and accesses the network data.

The **system** data structure comprises five sub-levels, which provide the required generality to describe any topological configuration. Figure 4.2 shows the **system**'s structure, comprising a tree of networks that have layers, composed of clusters, which comprise neurons, which finally contain synapses. This arrangement fully describes the topology of the network.

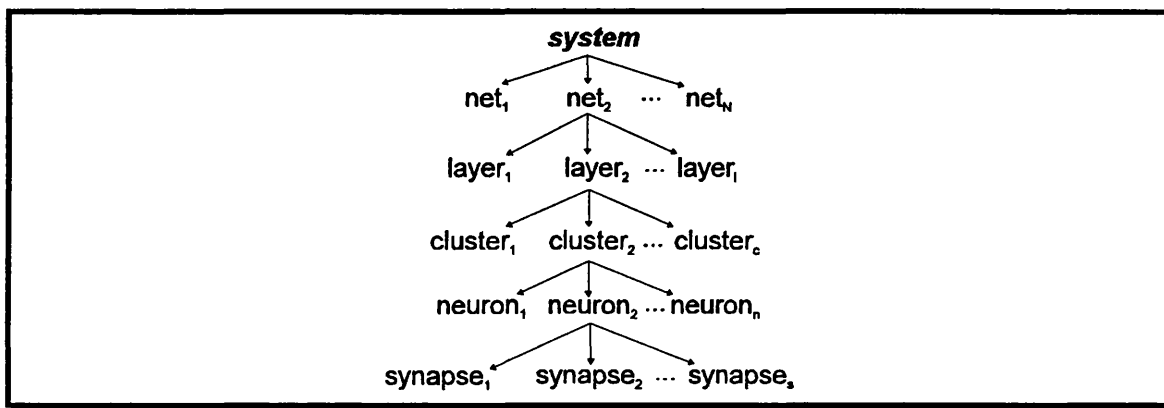


Figure 4.2 — The *nC* Hierarchical Data Structure

The **network sub-level** allows the development of heterogeneous neural models consisting, for instance, of a Hopfield and a Back Propagation network. The **layer sub-level** supports the usage of multilayer networks. The **cluster sub-level** permits the assembly of neurons into separated groups. This is motivated by the fact that some neural network models, such as the Competitive Learning [160] and the Self-Organising Map [99], assume a sub-division at the layer level. In these cases, neurons belonging to the same cluster are said to compete with each other, so that only the winner fires. Alternatively, neurons that exhibit similar functionality can be clustered, thus their behaviour is determined by the cluster level. The **neuron sub-level** and the **synapse sub-level** incorporate the *data* of the network, which are the neuron's output state and the synaptic weight, respectively.

All the **system** sub-levels share a common framework composed of: (i) a list of rules; (ii) a list of user-parameters; (iii) a list of system-parameters; and (iv) a list of lower level structures. The list of rules defines the control and the functionality associated with the hierarchy's sub-levels. The list of user-parameters embodies the algorithm dependent variables, as the *learning rate* and *tolerance* parameters specified at the network level of the algorithms. The list of system-parameters contains the algorithm independent parameters, associated with the specific **system**'s sub-level. Finally, the list of lower level

parameters is held by each **system**'s sub-level, such as lists of clusters inside layers. The complete definition of the **system** data structure is shown in Figure 4.3.

```
typedef struct {
    int n rules;
    rule type *rules;
    int n parameters;
    para type *parameters;
    int nets;
    net type **net;
} system type;

typedef struct {
    int n rules;
    rule type *rules;
    int n parameters;
    para type *parameters;
    int fanin;
    caddr_t fanout;
    caddr_t *input port;
    caddr_t *output port;
    int *target;
    int layers;
    layer type **layer;
} net type;

typedef struct {
    int n rules;
    rule type *rules;
    int n parameters;
    para type *parameters;
    int clusters;
    cluster type **cluster;
} layer type;

typedef struct {
    int n rules;
    rule type *rules;
    int n parameters;
    para type *parameters;
    int neurons;
    synapse type **synapse;
} cluster type;

struct NEURON {
    int n rules;
    rule type *rules;
    int n parameters;
    para type *parameters;
    TAGVAL state[NO_STATES];
    int route[5];
    int fanin;
    int fanout;
    struct NEURON **input neuron;
    struct NEURON **output neuron;
    int synapses;
    synapse type **synapse;
};

typedef struct NEURON neuron type;

typedef struct {
    int n rules;
    rule type *rules;
    int n parameters;
    para type *parameters;
    TAGVAL weight;
} synapse type;
```

Figure 4.3 — The *nC* **system** Data Structure

Rules

The information concerning *functionality* and *controllability* of a neural algorithm is also incorporated in the **system** structure through the concept of *rules*. There are two levels associated with rules: the functional level, or simply *rule level*, and the control level, or *meta-rule level*. Both levels are defined by the same data structure, as shown in Figure 4.4.

```
struct RULE {
    char tag;
    char *name;
    class_type *class;
    caddr_t para_list;
};

typedef struct RULE rule type;
```

Figure 4.4 — The General **RULE** Data Structure

The **rule** data structure comprises four fields: **tag**, **name**, **class**, and **para_list**. The **tag** field has a special meaning which will be discussed later. The **name** field defines simply the rule's name, which is used by the graphic monitor for display purposes. The **class** field, a pointer to the **class_type** data structure, holds information on the functionality (at the rule level) and controllability (at the meta-rule level), which are defined by specific functions. In addition, it specifies the number of parameters described in the **para_list** field, which contains a list of parameters that are manipulated by the respective functions.

At the rule level, where the functionality is specified, parameters in the **para_list** field are numeric values, upon which the function performs a specific calculation. As an example, the **state_update** rule for a particular neuron manipulates values, such as weights

and states, to compute the sum of products, and applies the activation function to obtain the neuron's output state. Figure 4.5 illustrates this example.

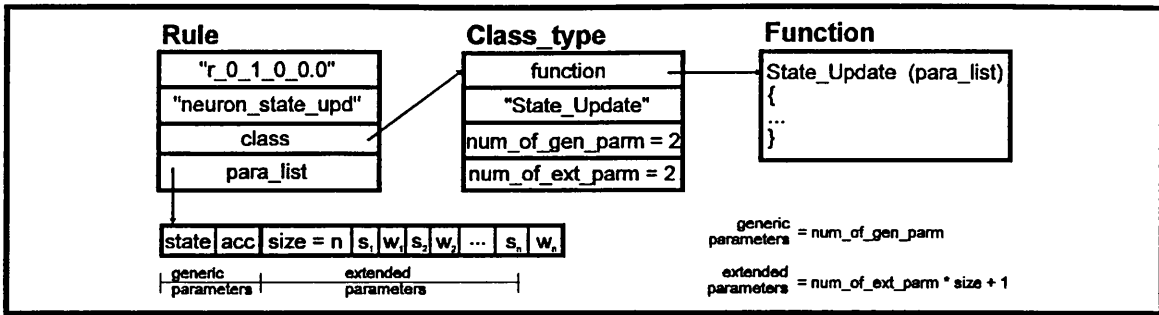


Figure 4.5 — Description of the Neuron's State Update *rule level*

At the *meta-rule level*, where controllability is specified, parameters in `para_list` are pointers to other rules (either at the *rule level* or at the *meta-rule level*). In this case, the associated function does not perform any specific calculation. Instead, it defines how the network execution is controlled, either sequentially, using `sexec` built-in function; or in parallel, using `pexec` built-in function. Figure 4.6 shows an example in which the `state_update` rule for a particular cluster defines the *rule-level* function `state_update` for each neuron in the cluster.

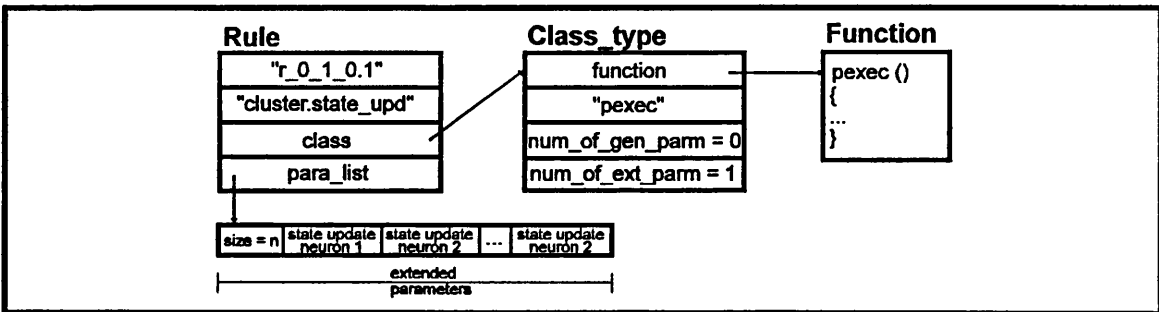


Figure 4.6 — Description of the Cluster's State Update *meta-rule level*

Tag Mechanism

It can be observed by Figures 4.5 and 4.6 that `para_list` has different contents depending upon the context. In the *rule level* case, the `para_list` field contains two main sections: *generic parameters* and *extended parameters*. The former include the fundamental numeric parameters that must be present in all rules of one specific *class*. In the example given in Figure 4.5, these parameters are the neuron's output (`state`) and the accumulator (`acc`). Extended parameters are related to the variable part of the rule, which in this example are the input states s_i and their associated weights w_i . In the *meta-rule level* case, the `para_list` field simply includes the extended parameters.

Due to the context sensitivity of the `para_list` field, it becomes difficult to determine what the contents of generic and extended parameter lists are during compilation¹. The tag field has been included in a specific data structure, namely **TAGVAL**, which defines the explicit type of the value, either float, integer, etc., through the **UNVAL** union definition (see Figure 4.7). Thus, the **TAGVAL** structure of each parameter provides a proper mechanism for tracing parameters. Similarly, the **RULE** structure includes a tag field that gives the exact level of the function rule (see Figure 4.4).

```

typedef struct {
    char    *tag;
    char    *type;
    UNVAL   value;
} TAGVAL;

typedef union {
    float    f;
    short    s;
    int      i;
    long     l;
    char     c;
    char     *cp;
    double   d;
} UNVAL;

```

Figure 4.7 — The **TAGVAL** and **UNVAL** Data Structures

A tag field is a symbolic name defined over three possible syntactic formats [131]:

1. neurons: "*n_x_y_z_w.v*"
2. synapse weights: "*s_x_y_z_w*"
3. parameter values: "*p_x_y_z.w*"
4. rules: "*r_x_y_z_w.v*"

The prefix *n* in the first string identifies a neuron state value, while *v* specifies the type of state. Similarly, the prefix *s* defines a synapse value, the prefix *p* a parameter value, and the prefix *r* defines the address of a rule function. The remaining of the string is variable and describes the hierarchical structure of the value being represented by the symbolic tag name. For instance, the symbolic name

"*n_0_2_3_6.0*"

is equivalent to the *nC* structure:

`sys→net[0]→layer[2]→cluster[3]→neuron[6]→state[0],`

which represents the output state value (determined by *v* = 0) of the sixth neuron in the third cluster of the second layer in the network 0. The purpose of '.' in parameter and rules

¹The tag field was originally envisaged to provide an unambiguous mechanism during the translation of *nC* into *nC_Code* [131], a special version of *nC* that includes explicit initialisation of all data structures, thus serving as a portable code for the target machines.

is to remove ambiguity, since parameters and rules may appear at all levels. For instance, “*r_1_2.3*” represents unambiguously the rule number 3 of the layer 2 in the network 1.

Parallel Execution Operator — PAR

The third basic concept incorporated into the *nC* language is the **PAR** operator, which indicates that all instructions within the following *open-and-close* brackets should be executed in parallel. Two basic uses of this operator are illustrated below:

Example 1: **PAR** {
 statement1;
 statement2;
 }

Example 2: **PAR** for (i=0; i<system->layer[l]->cluster[c]->neurons; i++)
 statement();

In these two examples, the compiler (for parallel machines) is instructed to map the statements in different processors for parallel execution.

As mentioned before, *nC* programs are divided in two sections: *application* and *algorithm*. The former is the specification of a particular application in which the user configures the neural network and specifies which model from the algorithm library should be used. The latter is the description of the functionality of a particular neural model, which is parameterised so that it can be used by any application. The next two sub-sections describe in more detail these two sections.

4.2.2. nC Application Definition

The application programmer can build a neural network application by specifying its configuration and producing the required code to control the execution of the specified algorithm, which is held in the library.

To specify the network configuration, a special data structure, namely **config**, must be used. An example of the config data structure is shown in Figure 4.8. Note that the complete network’s topology is specified. There is one network composed of three layers where each layer has one cluster. The first cluster has 96 neurons, the second has 24 neurons, and the third has 96 neurons.

Based upon this initial information, the **system** structure is constructed at run time in the very beginning of the *nC* execution. The **config** structure does not contain information about the network’s connectivity. The user must provide the functions that are

responsible for building the network's connection pattern, as described in the next sub-section.

The control section of the application program is defined in the main routine, which is basically composed of function calls and loops or iteration statements to control the function's execution. Functions are of two types: *system functions*, which deal mainly with input and output of data; and *algorithm functions*, which are the routines defined in the parameterised neural network algorithm library.

```

#define NETS          1
#define LAYERS        3
#define CLUSTERS      1

struct {
    int nets;
    struct { int layers;
            struct { int clusters;
                    struct { int neurons;
                            } cluster [CLUSTERS];
                } layer [LAYERS];
        } net [NETS];
    } config =
1, { 3,
    { {1, {96}},
      {1, {24}},
      {1, {96}} },
    /* One network of three layers */
    /* Layer 1 - One cluster of 96 neurons */
    /* Layer 2 - One cluster of 96 neurons */
    /* Layer 3 - One cluster of 96 neurons */
};

```

Figure 4.8 — The **config** Structure

Although any algorithm function defined in the library can be called in the application section, there are four compulsory algorithm functions that should be included in all applications: (i) **connect**, which allocates memory for all sub-levels of the **system** structure and assigns the correct pointers to establish the network connectivity; (ii) **build_rules**, which assigns pointers to the *rules* that each *system* sub-level should execute; (iii) **Learn**, that executes the learning of a specific pattern, previously stored in the **system** structure through the *system functions*; and (iv) **Recall**, that executes the recall phase for one pattern.

The application section can use the full C syntax to determine the application's execution. The only constraint is that **connect** and **build_rules** routines must be called before any algorithm function to fully initialise the **system** data structure.

4.2.3. nC Algorithm Definition

The algorithm programmer must conform to certain rules to code a particular algorithm in *nC*. Firstly, he or she must use only the restricted subset of C statements available to program all *rule functions*. Secondly, the built-in **system** structure must be used to describe the functionality of the algorithm in a parameterised way, thus allowing its usage by a wide range of different applications. Finally, the programmer must provide the code for **connect** and **build_rules** routines.

The coding of a *rule function* involves, initially, the decision of what parameters are necessary. Next the rule's functionality should be written according to the allowed *nC* statements. This should be performed through the manipulation of appropriate parameters.

The code for `connect` and `build_rules` routines is fundamental for the correct execution of a *nC* neural model. The `connect` procedure converts the general **system** structure declaration into a specific memory organisation, using the set of constants provided by the config data structure in the application section. It also assigns correct pointers to the *system* connectivity. The `build_rules` procedure uses the network topology set by the `connect` routine and constructs all necessary rules, at *rule-level* or *meta-rule level*. It basically attributes specific pointers to the rule function and its parameter list (see Figure 4.4). The names of these two procedures are compulsory, but their bodies are algorithm dependent, and must be coded correctly to generate the desired topology.

The complete specification of the *nC* language, together with a description of how to code each module (application and algorithm), can be found in the *nC* manual [191]. An example of a complete *nC* program can be seen in Appendix B of this thesis, which contains the description of a Back Propagation neural network.

4.3. *nC* Extensions

4.3.1. *nC* Strengths and Weaknesses

The *nC* language has proved to be an effective intermediate language for neural networks. In addition, it has also been used as the main programming language by users (bypassing the language *N*), despite its apparent lack of simplicity from the user's standpoint. In particular, the **system** hierarchical data structure is pivotal for fully describing neural networks in every aspect: topology, connectivity, functionality, controllability, and data. By searching through the **system** structure, the (software or hardware) compiler can build each element of the network description, and compile their functionality.

However, it must be noted that this language has been conceived precisely following a software-oriented approach, rather than a hardware-oriented methodology, as commonly used by HDLs. Neural networks coded in *nC* are meant to be simulated in sequential or parallel machines, and, when available, emulated in neurocomputers. Therefore, the *Pygmalion* environment provides only a software route from *nC* (and consequently, from *N*). There is no provision for hardware support, in which *nC* would be used to specify neural networks' hardware. To make a hardware route from *nC*, the language, as well as the environment as a whole, must be extended. While extensions to

the *Pygmalion* have been discussed in the previous chapter, this sub-section concentrates only on the necessary extensions to the *nC* language that are important for generating neural chips through silicon compilation.

The hardware extensions to the *nC* language followed the basic guideline: it should not transform *nC* into an HDL, since the user is assumed to be either an application or algorithm expert, and therefore has no knowledge of hardware. This design philosophy poses severe limitations on the capability of creating neural networks in hardware. However, it is believed that with an appropriate tool (the neural silicon compiler), efficient hardware implementations can be achieved even with minimal hardware information provided by the user. As discussed before, the success of this approach depends heavily on the design of the silicon compiler tool as well as its target architecture.

The approach developed in this work follows the above guideline closely. The user specifies high level commands that directly affect the hardware synthesis without giving any hardware-specific detail. These commands are fundamentally driven by the desired performance of the final application. These basically include time and area constraints. As an example, the user can specify a maximum time allowed to perform a particular neural *rule*. For real-time applications, it might be necessary to realise a complete recall or learning phase within a specific time. Conversely, the application might limit the number of chips due to space requirements. These would require a certain minimum number of PEs per chip.

Other specific hardware parameters, such as the precision for data and the mechanism for implementing the activation function, should also be specified by the user.

In summary, the most important strengths of *nC* are:

- *Complete information of neural networks* — the **system** structure contains all relevant data for describing neural networks;
- *Portability* — its structure provides a concise and efficient form to map any neural network onto several different machines, either based on sequential or parallel architectures; and
- *Algorithm and Application separation* — it is a convenient way to separate the algorithm from the application expertise, in which both can develop generic programs to be shared in several different circumstances.

The drawbacks of *nC* are:

- *Complexity* — its lack of simplicity, in particular for programming a neural algorithm, is mainly due to its memory management mechanism, which is performed by `connect` and `build_rules` routines;
- *No hardware support* — although very flexible, the language offers no hardware support, which is important for silicon compilation purposes; and
- *Hardware compilation* — there are several sections in the *nC* program that are not relevant to hardware. Therefore, the program should be *selectively* parsed and (*silicon*) compiled to efficiently generate hardware structures.

The lack of hardware support can be easily solved by extending *nC* with the focus on hardware compilation. Since the idea is to incorporate a hardware route to the *Pygmalion* environment, the fundamental conception of the language must not be changed. Furthermore, the extensions should be such that *nC* can still be used in the scope of the original software-oriented environment. The trade-off in extending *nC* (for silicon compilation purposes) without changing it considerably is obtained by adopting very simple solutions, which nevertheless provide the required support for the generation of neural chips.

4.3.2. Time and Area Constraints

As seen before, the **system** structure embodies a *rule level* concept that comprises the functionality and controllability of a neural network. A simple extension to this abstraction is the introduction of one further field to this structure that gives timing constraints. Therefore, when the neural silicon compiler synthesises each rule, its synthesis process is driven by the existence of such timing constraints. The extended rule structure definition is shown in Figure 4.9.

```
typedef struct {
    int max_time; /* Max time to perform rule in µsecs. */
} timing;

struct RULE {
    char *tag;
    char *name;
    class_type *class;
    caddr_t *para_list;
    timing *hw_time;
};

typedef struct RULE rule_type;
```

Figure 4.9 — The Extended RULE Structure

The **timing** structure determines the maximum time (in μs) that a specific neural rule should perform. This parameter should guide the hardware synthesis algorithms to generate an appropriate structure that meets this requirement. The introduction of timing constraints at the rule level is extremely convenient, since it is possible to specify the

precise timing information for each phase or sub-phase of a particular neural computation. This leads to several interesting possibilities, which should be resolved during silicon compilation time. For instance, the user may specify at the *macro level* that the recall phase should be performed by a given maximum time, or yet may specify at the *micro level* that one of its rules, for instance the Error_Calculation rule, should not exceed the specified maximum time.

Similarly, area constraints can also be specified by the user, so that a certain minimum number of PEs is implemented in each chip. Alternatively, the user may guide the network partition (see section 7.10) by determining the number of desired PEs in a chip. This is done for each type of PE (hidden layer, output layer, etc.). The *neuron_type* level has been chosen for incorporating this information (see Figure 4.10), since there is a direct mapping, regarding silicon compilation, between the neuron level and the PE that implements it.

```
typedef struct {
    int min_PEs; /* Minimum number of PEs in a chip */
} area;

struct NEURON {
    int n_rules;
    rule_type *rules;
    int n_parameters;
    para_type *parameters;
    TAGVAL state[NO_STATES];
    int route[5];
    int fanin;
    int fanout;
    struct NEURON **input_neuron;
    struct NEURON **output_neuron;
    int synapses;
    int synapse_type;
    area **synapse;
    area hw_area;
};
typedef struct NEURON neuron_type;
```

Figure 4.10 — The Extended *neuron_type* Structure

4.3.3. Fixed-Point Precision Calculation

Original *nC* programs available in the *Pygmalion* system have been written using floating-point precision for all data, despite the fact that the language definition supports multiple data types [191] through the definition of the **TAGVAL** and **UNVAL** structures (see Figure 4.7). Therefore, the first step towards the use of fixed-point precision (required by the target architecture) is to support integer data type for all data defined in algorithms and applications. This involved rewriting great part of the built-in functions, the algorithm library, and the application section.

The appropriate *nC* data structure for implementing such extension is the **TAGVAL** structure, as shown in the Figure 4.11. The **system** structure has thus been modified similarly as described for the **RULE** structure. A new field called *fixed_type* is introduced at the synapse and neuron level (to describe the data format for weight and state values, respectively) as well as at the parameter level.

It is important to observe that due to changes in the **system** data structure, the *Pygmalion*'s graphical monitor must be changed accordingly to allow monitoring hardware-specific parameters. These changes involve the definition of new *windows* in the process of monitoring internal variables of the neural algorithm. The simulator for neural networks' hardware takes immediate benefit from this extended monitor and is described in the following chapter.

```
typedef struct {
    int      int_part;      /* Specifies fixed-point data format */
    int      dec_part;
} fixed_pt;

typedef struct {
    char      *tag;          /* code_gen() and NSC */
    char      type;
    UNVAL     value;
    fixed_pt  hw_prec;      /* for fixed-point only */
} TAGVAL;
```

Figure 4.11 — The Extended TAGVAL Structure

4.3.4. Realisation of the Activation Function

As mentioned in chapter 1, certain activation functions are very expensive to implement in digital hardware. For those complex functions, a lookup table mechanism is usually employed. In this case, a new built-in function is defined, which is called `lookuptbl`. This function name is known by the compiler, which, instead of compiling it, executes its function definition to generate the table's contents. The table's size and its data width are specified by the user.

The user should write the activation function, regardless the employment of a lookup table mechanism. During simulation, the activation function is compiled and executed as any other function defined by the user. During hardware compilation however, if the user invokes the reserved function `lookuptbl`, no compilation takes place. Instead, a table is constructed with values obtained through the execution of the `lookuptbl` function.

Along with the lookup table mechanism, some neural algorithms work well with the implementation of a threshold activation function. This is clearly very simple to implement in digital hardware using a single comparator. To be able to generate such simple structure, another built-in function, called `threshold_fn`, is defined, which must also be provided by the user. The threshold value and the two extreme values are specified by three macro definitions, such as:

```
#define _THRESHOLD_VALUE    0.5
#define _THRESHOLD_ON      1
#define _THRESHOLD_OFF     0
```


This approach provides complete transparency between software simulation and hardware compilation. With new activation functions being incorporated into the *nC* language, the hardware compiler can be further expanded to handle the implementation of these new functions.

4.4. Summary

This chapter has briefly described the *Pygmalion* environment and in some detail its basic component — the *nC* intermediate level language. The language's strengths and weakness have been stressed, and extensions to incorporate hardware parameters for hardware generation through a silicon compiler have been proposed and implemented.

Extensions of the *nC* language followed a very simple approach, in which the hardware transparency from the user's viewpoint has been preserved. These include extending the **system** data structure at specific levels of the hierarchy. Hardware-specific high level parameters are introduced at the rule level to specify time/area constraints. The specification of the data format for the fixed-point calculation is provided at the neuron and synapse levels. Other extensions include the definition of activation functions, which can be implemented by lookup tables or very simple functions.

The effectiveness of these hardware extensions is central to the automatic synthesis of the hardware, being fully examined in chapter 7, where details of the *NSC* implementation are given. Before describing the *NSC*, the next chapter presents the implementation of the hardware simulator for neural networks developed in this work. The simulator is an integral part of the extensions carried out in the *Pygmalion* environment.

Chapter 5

Simulation of Neural Networks Hardware

This chapter presents the implementation of the neural networks' hardware simulator, which is fully integrated in the Pygmalion environment. An analysis of the implementation of neural networks in digital hardware is initially investigated, with emphasis on the learning phase. Then, the results of simulating a Back Propagation neural network under several hardware constraints are reported. These results serve as the basis for assessing the viability of implementing neural networks in digital hardware.

5.1. Overview

The great majority of software simulation tools for neural networks employ floating-point calculation with high precision, usually 32 and even 64 bits. In order to reduce the complexity involved in floating-point computation, both in terms of speed and area, digital hardware implementations tend to use fixed-point calculation with relatively low precision [83, 149]. This trade-off aims at minimising silicon area, thus allowing the integration of several PEs in the same chip.

The realisation of digital circuits to execute neural algorithms requires the adoption of several constraints. These restrictions are necessary to correctly perform the functionality of the system in a cost-effective way. Regarding the neurocomputing field, these hardware constraints introduce computational errors that lead to the degradation of the learning convergence and lack of accuracy in obtaining results. These errors can divert the trajectory of the learning process, generally increasing the number of cycles required to achieve the convergence. In extreme cases, they can completely prevent both learning and recall of some particular patterns.

Therefore, when digital hardware implementations are envisaged, it is important to investigate the effect of hardware constraints during the execution of neural algorithms. A flexible hardware simulation tool for assessing hardware performance of neural algorithms is required. This subject is little investigated in the literature [15, 130].

As seen before, most neural programming environments provide a powerful software simulation capability, but no hardware support. Therefore, to analyse those constraints, a parameterised neural network hardware simulator has been developed [130] and incorporated into the *Pygmalion* environment. This provides a smooth path towards the implementation of neural chips through silicon compilation. The simulator is able to mimic hardware realisations under a set of tuneable constraints.

The remaining sections of this chapter describe how this simulator works and, through its use, investigates the impact of digital implementations on the execution of neural networks. A Back Propagation network is used to carry out these studies because this model presents high complexity and generality embracing the majority of the neural models' characteristics.

5.2. Back Propagation Computation

Electronic hardware implementations of neural networks basically seek high speed operation with minimum use of silicon area. These requirements can be obtained through the employment of finite precision hardware. However, as mentioned before, this can lead to computational errors, which can prevent the correct operation of neural algorithms. Before analysing the errors caused by finite precision in digital hardware, the neural computation is reviewed, focusing on the Back Propagation network.

The basic computation performed by each neuron in artificial neural networks may be briefly summarised in the evaluation of the following expression:

$$o_i = y(Net(u_i)) = y\left(\sum_{j=1}^N W_{ij} o_j\right) \quad (1)$$

where o_i is the i -th neuron output, which is obtained by the mapping of its input $Net(u_i)$ through the activation function $y(\cdot)$, and W_{ij} is the weight associated with the connection between neurons i and j . The Hebbian learning rule [78] involves the updating of weights associated with inter-neuron connections, which according to the Back Propagation algorithm may be expressed as:

$$\Delta W_{ij} = \eta o_j \delta_i \quad (2)$$

The error δ_i assumes different expressions according to the layer involved in the learning phase. In particular, for the output layer, δ_i is simply expressed as the first expression of (3), while for hidden layers δ_i reduces to the second relation:

$$\delta_{i[output]} = (t_i - o_i)y'(x_i) \quad \delta_{i[hidden]} = y'(x_i) \sum_{k=1}^N W_{ki} \delta_k \quad (3)$$

where δ_k is the error generated in the subsequent layer, t_i is the target output value associated with the supervised learning, and y' is the derivative of the activation function. Further details can be found in [159].

The recall phase is dominated by Equation (1). Each neuron simply performs the sum of products operation, followed by the activation function mapping. Conversely, the learning phase is more computing intensive, where weights are updated according to Equation (2), which in turn depends on results processed by Equations (1) and (3). The updating of weights is repeated several times until the error reaches a value within a certain tolerance. Therefore, when the convergence is close to being achieved, very small values are obtained. In this case, digital implementations with low precision can prevent the required correctness.

The need for high precision when errors are being calculated is the reason for all software simulation tools to employ floating-point arithmetic. However, digital implementations use fixed-point arithmetic, and its influence in the neural algorithm should be analysed, along with other hardware constraints. The next section discusses these hardware constraints, which are usually required for finite precision (neural) computation.

5.3. Hardware Constraints

The employment of fixed-point arithmetic impacts on the hardware in several ways. By analysing Equations (1), (2) and (3), several sources of errors can be identified during the computation. These include the result of multiplication and summation operations, the implementation of the activation function through a lookup table, and the mechanism for addressing this table.

5.3.1. Multiplication

The multiplication of two binary values of n bits requires the result to be (correctly) represented by $2n$ bits. In digital hardware implementations, a register of $2n$ bits is normally provided. However, neural networks require the computation of a sum of products for each artificial neuron, as given by Equation (1). Therefore, truncation is generally employed to bring the result back to n bits. This alters the correctness of the learning algorithm, thus diverting the training process from the trajectory of the high computation.

Figure 5.1 illustrates two possible mechanisms for truncating a number back to its original precision. In this case, two 12-bit fixed-point numbers are multiplied, producing the result as a 24-bit number. The correct position of the decimal point, after the multiplication has taken place, is also illustrated in the Figure 5.1. The technique is a straightforward one, in which the most significant part of the result is extracted (Figure 5.1a). However, most of the precision, represented by the decimal part of the number is lost. The second technique (Figure 5.1b) avoids this problem by using a *window* that maintains the decimal point in the original position.

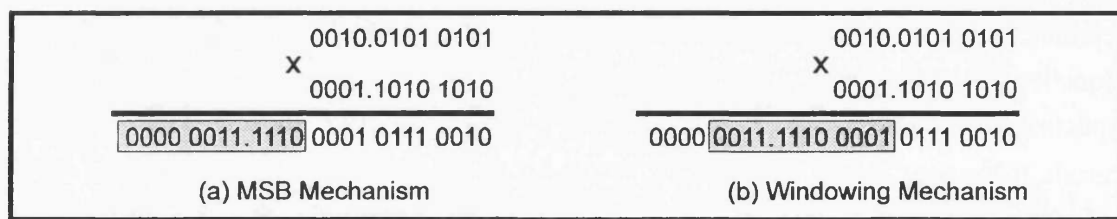


Figure 5.1 — Methods of Employing Truncation After a Multiplication Operation

The MSB mechanism guarantees that no overflow nor underflow conditions occur, at the expense of losing precision (in the above example, from 16 bits to 4 bits in the decimal part). The *windowing* mechanism provides an improved precision (8 bits in the example), but tends to cause overflows and underflows if large absolute operands are used.

Regarding VLSI design, the first approach (Figure 5.1a) is easily implemented in hardware. The second approach, however, may pose some problems in the interconnection of the data path, since signals flowing from one cell to another are usually at fixed positions. Nevertheless, this problem can be solved by directly employing the *windowing* mechanism in the hardware multiplication algorithm.

5.3.2. Summation

During the sum of two operands, overflow and underflow can occur, either because the operands have both large values, or due to the *windowing* mechanism employed by the multiplication operation. In either case, for a correct representation of results, numbers have to be saturated to the proper extreme values. Although these approximations may introduce further errors in the computation process, they can yield a positive effect, since the asymptotic behaviour of the activation function tends to saturate those values. Conversely, saturation may prevent the proper network evolution if they occur prematurely.

Figure 5.2 shows that for a sigmoid activation function, values beyond the shaded region should lead the output to the extreme values (0 or 1). Therefore, any increment in the *net* input value will not change significantly the output value $f(\text{net})$. As a consequence, it is expected that, whenever a solution exists, the learning phase will converge in fewer cycles than required when using greater precision.

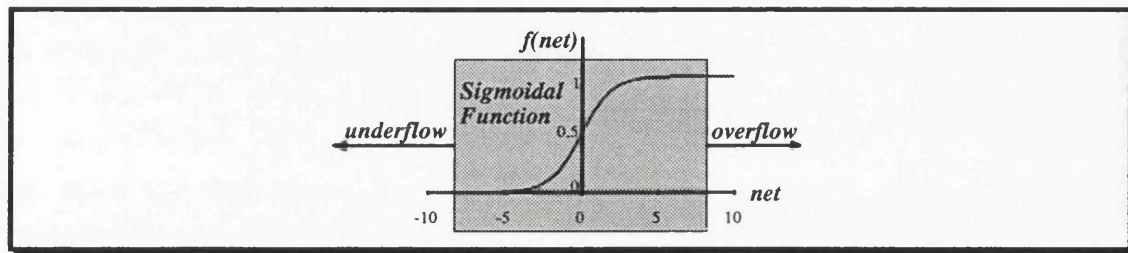


Figure 5.2 — The Effect of Overflows and Underflows

5.3.3. Activation Function

The non-linearity associated with activation functions represents one of the main bottlenecks in digital VLSI implementation of neural networks, since its realisation requires great overhead in time and silicon area. A common solution makes use of a lookup table [75] where pre-evaluated values are stored. Interpolation techniques can also be used to reduce silicon area as well as improving the activation function realisation [91]. Other solutions suggest architectures directly implementing approximating functions, as piecewise linear [127], or series expansion evaluation [16].

5.3.4. Table's Indexing Mechanism

When the sum of product's result is fed into a lookup table, other approximations arise. For instance, consider a 16 bit entry point. This would require 65,536 entries, which would be regarded as an unacceptable solution for a digital VLSI implementation. Thus, the table size imposes an upper bound on the number of bits. If the table has 256 entries, then only 8 bits out of 16 are taken as input. Again, overflow and underflow situations must be treated by saturating the activation value to the asymptotic values of the activation function. This situation is illustrated in Figure 5.3 for a 12-bit entry point and a lookup table of 256 elements, which forces taking only 8 out of the 12 bits. In this example, the indexing mechanism is employed by taking the most significant part of the value.

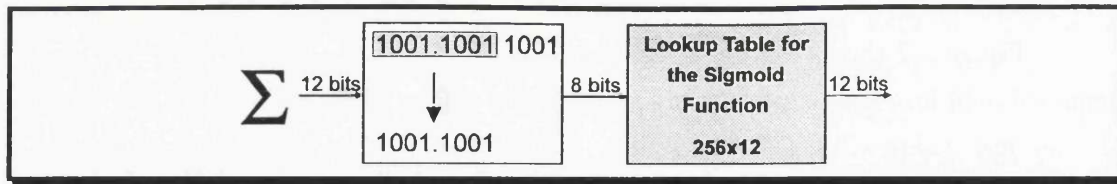


Figure 5.3 — The Lookup Table Indexing Mechanism

In the general case, the indexing mechanism should take into account the fixed-point representation. Figure 5.4 illustrates the strategy, in which saturation is employed when the number of bits in the integer part of the index is smaller than in the value. Note that if the *msb* field in the integer part of value is greater than 0, that means that the conversion mechanism has resulted in overflow (or underflow) due to fewer bits representing the integer part of the number. Therefore the saturation mechanism should result in the respective extreme value, which is dependent upon the way the table is built. For purposes of hardware simplicity and due to the two's complement representation, the first half of the table is filled in with positive indexes, while the second half is filled in with negative values. Thus, the maximum positive number is half the size of the table, while the maximum negative number is the half size of the table plus one.

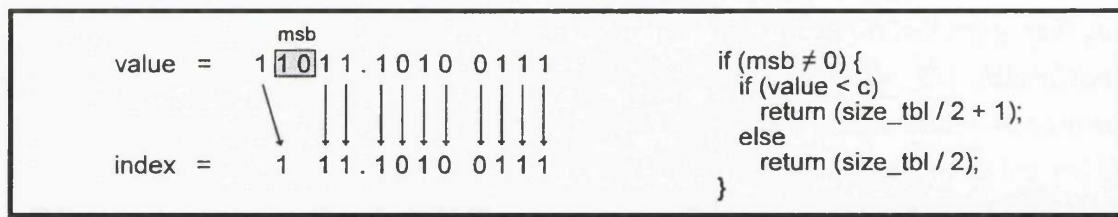


Figure 5.4 — Mechanism for Converting a Value into the Table's Index

5.4. Back Propagation and Precision Requirements

All the previously discussed hardware constraints are mandatory if efficient realisation is to be achieved. The effects of these constraints are analysed through simulations in section 5.5. However, preceding the simulation results, the next sub-section present two theoretical studies [17, 85] that lead to some guidelines on precision requirements for digital hardware realisation of Back Propagation neural networks.

5.4.1. Theoretical Analysis

Very little is actually known on the effect caused by finite precision arithmetic on neural computation. Nevertheless, two interesting works on theoretical analysis have provided a preliminary identification on the required precision for the Back Propagation

algorithm [17, 85]. This sub-section presents these studies, so that a comparison with the results obtained through hardware simulations and reported in section 5.5.3 can be made.

As seen in section 5.2, Equations (1), (2), and (3) define the Back Propagation computation process and represent the key issue in digital hardware implementation. Some of the previously discussed sources of error in a finite precision computation are analytically studied by Holt and Hwang in [85], and Alippi in [17]. These include: errors in the states and weights, in which sources are prior finite precision data manipulations; errors generated by operations using finite precision calculation, in particular, the multiplication operation and the non-linear operator $y(\cdot)$; and finally, errors generated by truncation techniques used in finite precision calculations.

Holt and Hwang [85] have developed an analytical analysis of the effect caused by these errors in all steps involved in the Back Propagation algorithm: recall phase (1), weight updating (2), and output and hidden layer error calculation (3). In their work, a statistical evaluation of the errors is considered. This evaluation is based upon the mean and variance analyses using truncation, jamming, and rounding techniques, and upon statistical properties of independent random variables, sum of independent random variables, and sum of products of independent random variables. The results obtained can be used to guide the determination of the necessary number of bits for weights in both phases of the neural computation: recall and learning.

For the learning phase, they have evaluated the effect caused by finite precision computation on the weight updating:

$$\rho \equiv \frac{E[\epsilon_{\Delta w}^2]}{E[\Delta w_{ij}^2]} \quad (4)$$

where ρ is defined as the ratio between the statistical expected square of the finite precision weight updating error and the full precision weight updating magnitude [85]. This ratio depends not only upon the number of bits assigned to the finite precision computation, but also upon the current stage of learning progress, which can be specified by the distribution of the difference between the desired and actual output.

Based on this result, it is reported that at least 16 bits of precision must be used for the learning phase to avoid having the training process diverting too much from the trajectory of the high precision computation. Conversely, for the recall phase, it is predicted that 8-bit (1 sign bit plus 3 bits to the left and 4 to the right of the decimal point) value for weights does not degrade significantly the performance, provided that the network is trained using high precision.

Alippi [15, 17] has elaborated a distinct approach to study finite precision computation in the Back Propagation algorithm. His approach provides a definite result on the minimum number of bits necessary to represent weights in digital implementation. The result gives bounds on the number of bits necessary to represent the integer and decimal parts of weight values. This study considers a large family of activation functions, given by Equation (5),

$$y = k + \frac{c}{1 + e^{Tx}} \quad c, T \neq 0 \quad (5)$$

including the sigmoid function as a subset of this family by choosing $k = 0$, $c = 1$, $T = -1$.

A general expression for the number of bits n required for the decimal part of weights is determined by Equation (6),

$$n \geq \log_2 \left[\frac{|c|}{\eta |T| \mu \varepsilon^2 (|c| - \varepsilon)} \right] \quad (6)$$

where, η is the learning rate and ε is the tolerance value. The number of bits m needed to represent the integer part of weights is set by:

$$m \leq \log_2 \bar{t} \frac{4\eta |Tc^2| \mu}{27} \quad (7)$$

where \bar{t} is the effective number of learning steps required to solve the application, and $\mu = \max(|k|, |k + c|)$. A formal proof of these results can be found in [17]. Table 5.1 shows the theoretical predictions given by Equation (6) for some values of ε and η .

ε	0.1	0.1	0.1	0.1	0.2	0.2	0.2	0.2
η	0.1	0.2	0.5	1	0.1	0.2	0.5	1
Number of Bits	10	9	8	7	9	7	6	5

Table 5.1 — Theoretical Results for the Sigmoid Function

Equation (6) gives the expression of the required number of bits needed to represent the decimal part of weight values. It can be seen that this relation is independent of the involved application. For greater values of ε , fewer bits are required, since less iterations are involved. Equally, for greater values of η , fewer bits are required, since less cycles are generally required to achieve the convergence. However, in this case, the probability of encountering a *local minima* is greatly increased [161]. When these situations occur, the convergence becomes impossible to be achieved.

Equation (7) gives a maximum number of bits required to represent the integer part of weights, beyond which no improvement in representing the value is obtained. The interesting aspect shown in Equation (7) is that results are independent of the specific problem or application involved. The chosen number of bits for the integer part of the weight values will support any application that converges in less than or about \bar{t} steps.

5.5. Hardware Simulation

The basic motivation for the development of a hardware simulation tool is the investigation of the target architecture's suitability for the realisation of cost-effective application-specific neural chips. This tool has been designed primarily to analyse how the adopted hardware constraints, previously discussed, affect the performance and correctness of neural networks' execution.

The hardware simulator extends the usage of the *Pygmalion* environment, so that users can test the performance of their application in neural networks' hardware, which comprises neural chips synthesised by the *NSC*. The simulator has been designed for target architectures that use fixed-point arithmetic and lookup tables for evaluating the activation function. The user may also evaluate the performance of the target architecture regarding the actual execution time — an important issue for real-time applications.

5.5.1. Hardware Library

The *Pygmalion*'s library of neural network algorithms has been expanded to incorporate a hardware library. It is functionally a copy of the software library, modified to perform calculations in the same way the target architecture's hardware does. This assumes that for each neural network model present in the software library, there will be a correspondent model in the hardware library, as many times as the number of target architectures supported by the *NSC*.

Most of the work at this stage involves: (i) re-coding all algorithms in the library for using integer data type, rather than floating-point type; (ii) re-coding all built-in functions to support integer as well as floating-point types and operations; (iii) implementation of the activation function either through a lookup table, a threshold function, or by a function that is coded for hardware execution.

Depending on the user's activity, either the software or the specific hardware library is used. If a hardware library is used, input and output operations during the hardware simulation are scaled from or to real value, respectively, to maintain compatibility with the *Pygmalion* environment. This approach permits total transparency

between software and hardware simulation of a particular application, that is, no modification in the user's application code is required. The only requirement for hardware simulations is that the user specifies hardware parameters, described in the following sub-section.

5.5.2. User Configuration Parameters

The simulator provides a flexible set of parameters. They allow the user to experiment with existing neural models in the *extended Pygmalion* environment, and to employ a wide range of hardware constraints, which include the definition of the following parameters:

- *Target Architecture* — uniquely specifies the modules to be used from the hardware library;
- *Data Precision* — specifies the number of bits used for the integer and decimal part of all data, such as weights, states, etc.;
- *Activation Function* — specifies how the activation function is implemented. The code for this function must be provided by the user. It is either digitised to produce the lookup table or *silicon* compiled during the hardware synthesis, according to the user's specification;
- *Table's Index Resolution* — determines the number of bits used for the integer and decimal parts of the index used to address a particular element in the table. The total number of bits is a function (\log_2) of the desired table's size;
- *Fixed-Point Mechanism* — determines the technique used to represent results after multiplication and summation operations are performed as described in section 5.3.

5.5.3. Simulation Results

This section reports the most important results obtained through exhaustive simulations of a Back Propagation neural network. Further details can be found in [130] and in Appendix C. The classical character recognition application is used to test the hardware performance based on the assumptions discussed throughout this chapter. These simulations also serve to demonstrate the effectiveness of the modifications and extensions made in the *Pygmalion* environment and in the *nC* language. In particular, the flexibility provided through the independent specification of hardware constraints allowed the observation on individual effects caused by each of these constraints. The results are then compared with the theoretical analyses discussed in section 5.4.1.

Figure 5.5 shows the character recognition application used as a benchmark for the simulations. The Back Propagation neural network is configured with three layers of 96x24x96 artificial neurons, and is trained to recognise 10 numeric characters (0-9) defined over a matrix of 96 pixels (12x8). Such problem is a simplified version of the OCR (Optical Character Recognition) application, in which the network is trained to read corrupted characters (or hand-written versions) and complete them to produce the correct pattern.

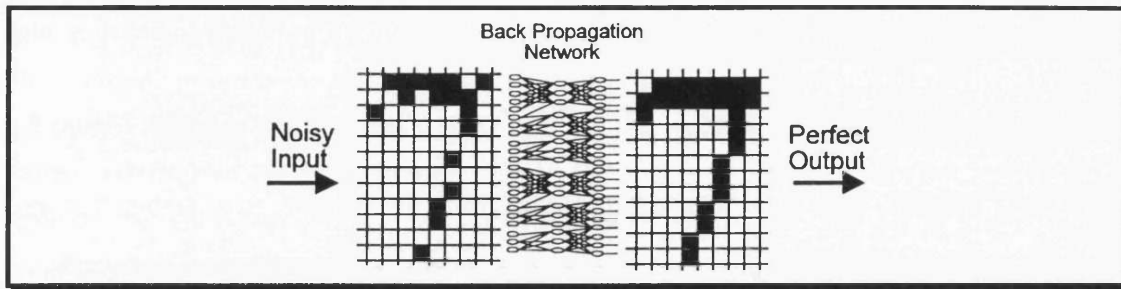


Figure 5.5 — The Character Recognition Application

Recall Phase

It has been frequently mentioned in the literature that fewer bits are required to correctly execute the recall phase of a neural network. This suggests that applications not requiring learning can be implemented in hardware using lower precision, and therefore using less VLSI area. This option is very important, and the neural networks' hardware simulator offers to the user the required flexibility to determine and tune the best configuration of hardware parameters.

It has been observed (see details in Appendix C) that at least 8 bits of data (4 in the integer part and 4 in the decimal part) are required to correctly retrieve the ten patterns without employing any *saturation* and *overflow/underflow* mechanism. The minimum number of 4 bits in the integer part guarantees that overflows and underflows do not occur, while 4 bits in the decimal part is the minimum precision needed to correctly retrieve all ten patterns. By reducing the number of bits in the integer part of data, overflows and underflows occur more frequently. When 2 bits are used, the network is no longer able to recognise all patterns. Appendix C gives details on the obtained results.

The influence of the lookup table is less effective than the precision of data. It has been found during these simulations that a lookup table made of only 32 entries is enough to run the network free of any error, while an 8-entry table has been reported to produce several overflows and underflows, but still able to correctly recognise all ten patterns.

Learning Phase

The learning phase in a Back Propagation neural network involves the recall phase followed by the processing of Equations (2) and (3). Since this phase deals with the updating of weights that is a function of the errors given in (3), very small values can be obtained when the convergence is close to be achieved. Therefore, it is expected that low precision can hinder the correct trajectory towards the convergence.

The influence of the lookup table is shown in Figure 5.6. It can be seen that by increasing the table size, the number of cycles required to get convergence is also increased, since the trajectory now takes smaller steps through the solution. A table with 256 entries has shown to be enough for the particular application. In addition, Figure 5.6 indicates that there is a minimum size for the lookup table beyond which no further improvement in the learning speed is achieved. It has also been observed that for an *ideal* sigmoid function (without using lookup table) the convergence is reached in 269 cycles.

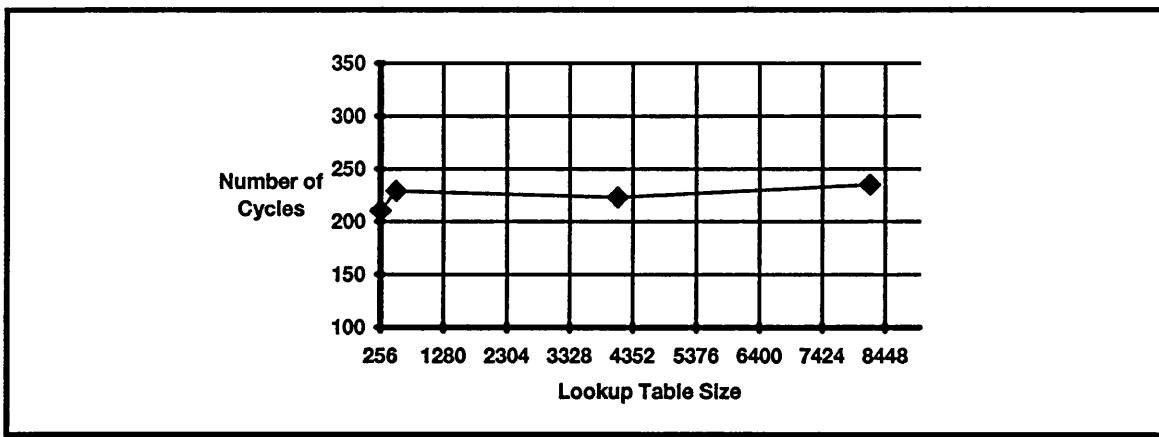


Figure 5.6 — The Effect of the Lookup Table

The reason why the convergence is slower when using high precision calculation of the threshold table is that it enforces a learning trajectory that uses very fine steps through the solution, which generally requires more iterations.

The effect of varying the decimal and integer part of the data is shown in Figure 5.7. By keeping fixed the integer part of the representation (6 bits in Figure 5.7a), and varying the decimal part, the simulations show the influence of ϵ and η . These results are congruent with results obtained in the theoretical analysis (section 5.4.1) and summarised in Table 5.1. Note that a minimum of 7 bits is required ($\epsilon = 0.2$ and $\eta = 0.5$) for the decimal part of weights, while for $\epsilon = 0.1$ and $\eta = 0.1$, a minimum of 10 bits is required.

Similarly, Figure 5.7.b shows the relationship between the learning phase and the integer part of weights (keeping fixed the decimal part at 13 bits, while the integer part is

varied from 3 to 6 bits). A minimum of 3 bits is required to represent the integer part. However, when few bits (such as 3 and 4) are used, overflows and underflows frequently occur, thus activating the saturation mechanism. Nevertheless, the network is still able to successfully learn the patterns, although it requires more cycles to converge. From 5 bits onwards, no saturation mechanism is required, and performance is improved (see details of these results in Appendix C). Again, these results are very close to the ones obtained in section 5.4.1.

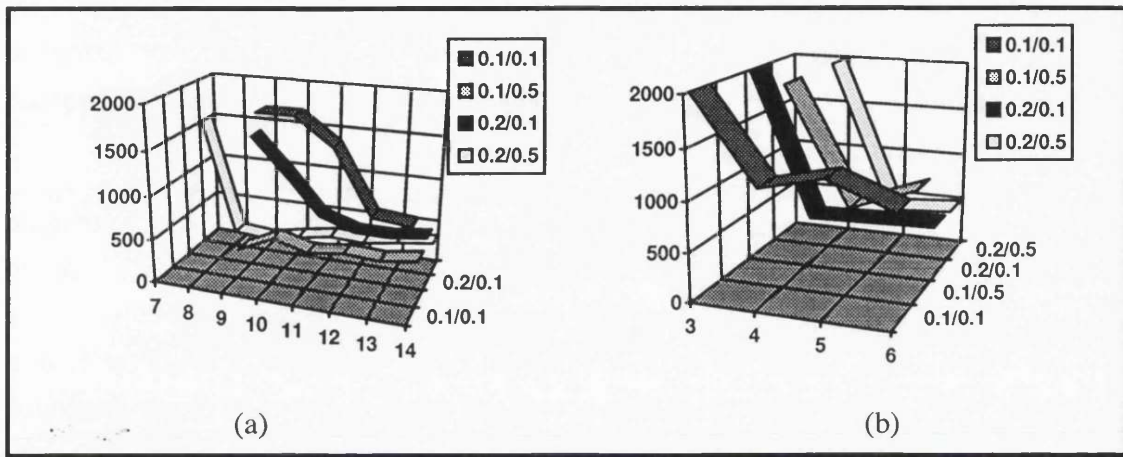


Figure 5.7 — Impact of Hardware Constraints During the Learning Phase

Some discrepancies between simulation results and the theoretical analysis discussed earlier are due to hardware constraints, which are not considered in the mathematical approach. The simulation shows that the implementation of the activation function through a lookup table plays an important role for the learning phase of the neural computation. Therefore, the table size and its indexing mechanism must be carefully designed.

Finally, it is interesting to compare hardware simulation with software simulation results for the same application, employing floating-point representation with no hardware constraints (Table 5.2). This comparison highlights the relevance of hardware constraints and shows the feasibility of using fixed-point arithmetic for the learning phase. In contrast to floating-point representation, experimental results show that with an appropriate choice of the number of bits, fixed-point notation reduces the required number of learning cycles. Furthermore, regarding the hardware implementation, fixed-point arithmetic leads to faster and smaller realisations.

ϵ	0.1	0.1	0.2	0.2
η	0.1	0.5	0.1	0.5
Cycles	963	>2000	512	>2000

Table 5.2 — 32-bit Floating-Point Simulation Results

5.6. Summary

The importance of the hardware simulator, integrated in the *Pygmalion* environment, is twofold:

- Hardware experimentation — it provides a tool for testing different hardware constraints for each particular application and algorithm, so that its final performance can be better predicted; and
- Hardware parameters tuning — it permits the adjustment of the ASNNs, so that the user can find an optimal hardware configuration in terms of area and speed. As an example, a design with on-chip learning should use higher number of bits for weights and states than a design that is limited to the recall phase only. In the latter case, a higher level of integration is obtained, affording a larger number of PEs per integrated circuit. Therefore, the NSC uses this information to yield the neural chip in an optimal way.

Although theoretical studies have shown the capability of expressing a relation between the algorithm (Back Propagation) and the required precision, the results are still very limited and dependent on the algorithm parameters. As an example, the introduction of a *momentum* term in the Back Propagation algorithm [161] invalidates the results obtained by Equations (6) and (7). Conversely, through simulations, the user has the flexibility to experiment any application or algorithm and find out what hardware parameters are suitable to solve the particular problem.

An exhaustive simulation of the OCR application under several hardware constraints has provided encouraging results in terms of hardware performance for the Back Propagation neural network. The results obtained are consistent with theoretical studies [17] as well as practical experiments [22]. The success verified through simulations is particularly important to the *Generic Neuron* architecture, because all issues investigated in this chapter are in fact implemented by this architecture.

The next chapter introduces the NSC framework and gives a detailed description of the *Generic Neuron* architecture.

Chapter 6

NSC and Target Architecture

This chapter presents the proposed framework for the design of the Neural Silicon Compiler (NSC). The compiler's key component, the Generic Neuron target architecture, is analysed in detail. Then, a Back Propagation neural network prototype for this architecture is written and simulated in VHDL. This includes developing a library of VHDL modules that are used by the NSC.

6.1. Overview

The main goal of this research is to propose and implement a hardware route to the existing neural network programming environments, in which ASNNs can be automatically synthesised to efficiently execute a particular neural network application. The *Pygmalion* system has been chosen as the platform for the development of this proposal.

As discussed before, the *Pygmalion* system has been originally conceived to program neural network algorithms, and to execute neural applications onto a variety of target machines, typically UNIX-based computers. The emulation of neural networks onto parallel machines and true neurocomputers is (at the time of writing) being developed by the *Galatea* project, which is the *Pygmalion*'s successor. The main goal of the *Galatea* project is the refinement of the *Pygmalion* environment and the construction of general-purpose neurocomputers [20]. A similar proposal has been developed as part of another PhD thesis [155], which centres on a flexible communication architecture that provides a distributed execution environment for neural network models.

Therefore, the main distinction between the above two proposals and the present research is that the first two works concentrate on the execution environment of neural algorithms, while this thesis focuses on the development environment, which has two routes: software and hardware. The first route is provided by the *Pygmalion* environment, while the second is accomplished through the hardware extensions introduced in this environment, as shown in Figure 6.1. The basic components of these extensions are:

- **Hardware Algorithm/Application Library** — the hardware counterpart of the software algorithm library; it holds a duplicate of all neural models, but it is specifically coded for the target architecture supported by the silicon compiler and the hardware simulator;
- **Hardware Simulation** — extends the simulation environment to assess the network behaviour during hardware execution; and
- **Silicon Compilation** — a hardware design tool that reads a *nC* program and synthesises a set of identical neural chips, which are then used to execute the entire neural network in hardware.

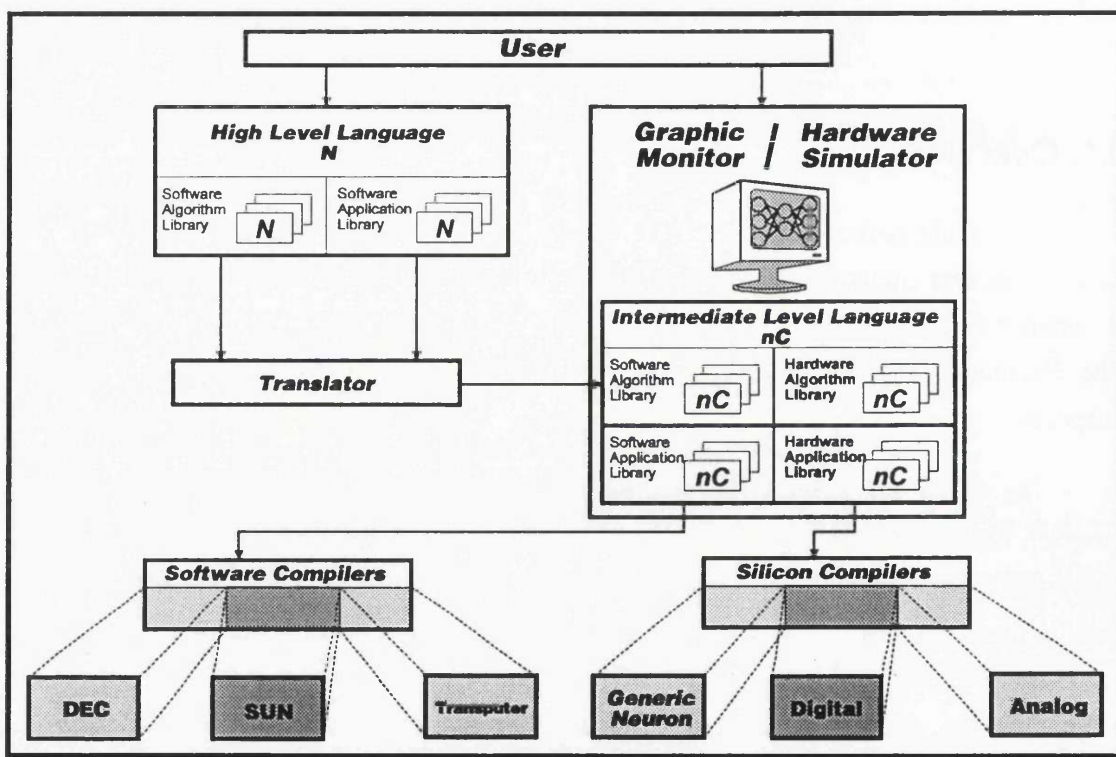


Figure 6.1 — The Hardware Development Extensions to the *Pygmalion* Environment

The hardware algorithm library and hardware simulation components have been described in the previous chapter. The general framework of the neural silicon compiler (high level synthesis) is discussed in the next section.

6.2. High Level Synthesis

The main goal of the NSC is to automatically synthesise neural chips with minimal knowledge of VLSI design from the user. Figure 6.2 shows the proposal for a *generic* silicon compiler for neural networks. The solid lines represent the implemented path of this

approach, while the dashed lines depict possible extensions. Central to this proposal is the *Intermediate Code Representation (ICR)*, an independent hardware description for neural networks. This representation incorporates information of the network's topology and interconnectivity, as well as the functionality of every neuron in the network. This functionality is described in terms of a graph, which contains data and control information.

The importance of the *ICR* focuses on the separation between the front-end compilation part and the back-end compilation (see Figure 6.2). This allows the construction of an open system: at the front-end, more than one specification language and programming environment can be used, while at the back-end, several target architectures and technologies can be synthesised. This approach is attractive for two basic reasons:

- There is no commitment with a particular programming environment — several neural programming environments exist today, and to make them compatible with the *NSC*, the only necessary action is to provide a compilation path from the adopted programming language to the intermediate representation;
- The system can be gradually extended to provide alternative technological routes, as soon as they become available; in particular, optical and opto-electronics technology look a promising alternative for neural computing.

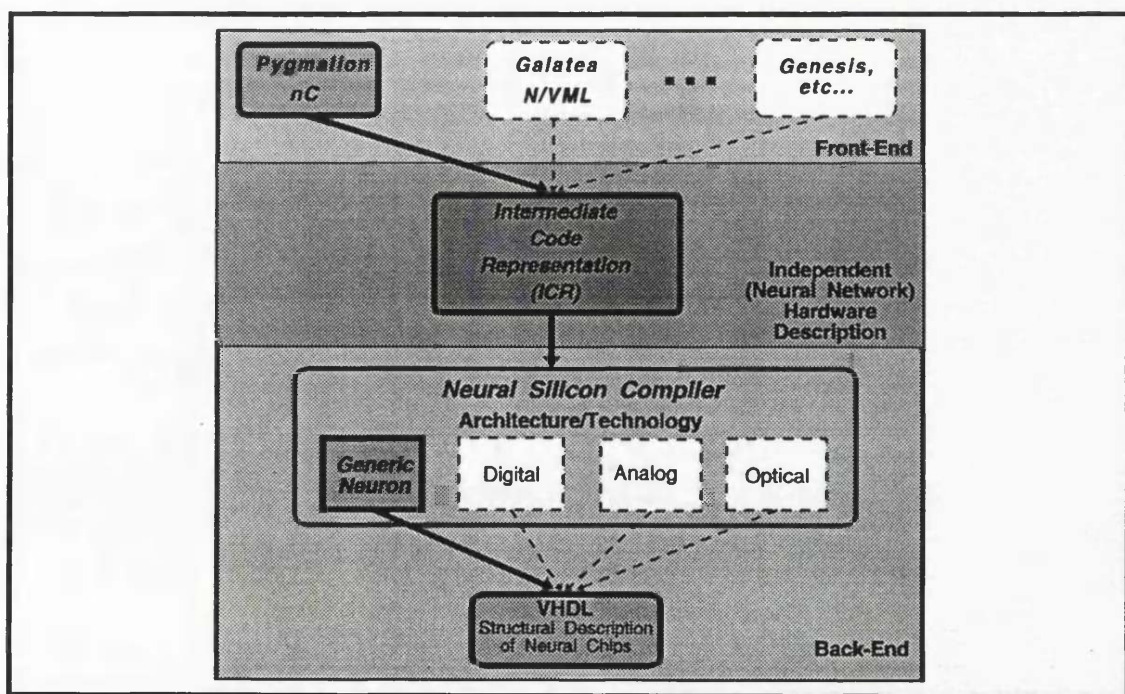


Figure 6.2 — A Generic Neural Silicon Compiler

By not committing the *NSC* with a particular programming environment, the use of a wide range of existing systems is permitted. Furthermore, there is no need for a complete re-writing of environments; instead, only the hardware extension is required, besides the compilation path from the high level language to the intermediate representation.

The support given to alternative technological routes provides a great degree of flexibility. Digital architectures are very flexible and potentially can implement on-chip learning. Analogue techniques are suitable only for the recall phase, but have a high degree of integration. Optical technology seems to be very promising, but still needs some research. Therefore, as technology progresses, the same system can be gradually expanded to support automatic synthesis of neural circuits at different technological alternatives. Following this direction, VHDL plays an important role, since it is a technology independent hardware description language. The generation of neural circuits at a specific technology is left to the low level synthesis tools, which are technology dependent.

The implementation of the entire system shown in Figure 6.2 is beyond the scope of this work. Instead, a single path is drawn: it starts from *nC* and the *Pygmalion* environment, passes through the *ICR*, and creates a VHDL structure of the neural circuit targeted at the *Generic Neuron* architecture. While the next chapter presents a detailed description of the *NSC*'s implementation, the remaining sections of this chapter are devoted to the description of the *Generic Neuron* architecture and the implementation of a Back Propagation neural network written in VHDL.

6.3. Generic Neuron Architecture

The *NSC* developed in this work synthesises circuits that are based upon the *Generic Neuron* architecture, developed as part of another PhD thesis [190]. The *Generic Neuron* architecture has been devised to encompass the main features of the existing neural models into a single building block. The architecture provides: **generality** to implement the wide range of available neural models; and **simplicity** to optimise the processing element's silicon area.

Generality is required by inspecting the spectrum of different neural network models, shown in Table 6.1. These models differ on several aspects: *network topology*, *recall phase*, and *learning phase*.

Network topology ranges from single layer, feed-forward models, such as the Perceptron [122], through single layer with feedback connections, as the Hopfield model [86], to more complex interconnection patterns, found in the multi-layer networks

with back propagation of errors [161], and two-dimensional grid of neurons of the Self-Organising Map [99].

Neural Network Model	Network Topology	Range of input values	Recall/Learning Phase			
			f_1 - Recall Phase		Learning Phase	
			Propagation Rule	Activation Function	f_2 - Weight Updating	f_3 - Error Calculation
Hopfield/Kohonen	single-layer with feedback	binary	$net = \sum s_i \cdot w$	hard limiter	$\Delta w_{ij} = s_i \cdot s_j$	\times
Perceptron	single-layer feed-forward	binary or continuous	$net = \sum s_i \cdot w$	hard limiter	$\Delta w_{ij} = \eta \cdot s_i \cdot \epsilon_j$	$e_j = t_j - s_j$
Widrow-Hoff (Delta Rule)	single-layer feed-forward	continuous	$net = \sum s_i \cdot w$	linear	$\Delta w_{ij} = \eta \cdot s_i \cdot \epsilon_j$	$e_j = t_j - s_j$
Back Propagation	multi-layer bidirectional links	continuous	$net = \sum s_i \cdot w$	sigmoid	$\Delta w_{ij} = \eta \cdot s_i \cdot \epsilon_j$	$e_n = T'(net) \cdot (t_j - s_j)$ $e_n = T'(net) \cdot \sum E \cdot W$
Boltzman Machine	multi-layer or randomly connected	binary	$net = \sum s_i \cdot w$	sigmoid	$\Delta w_{ij} = \eta \cdot \epsilon_j$	$e_j = \eta (\langle p_{ij} \rangle - \langle p_{ij} \rangle)$
Counter Propagation	multi-layer feed-forward	binary	$net = \sum s_i \cdot w$	hard limiter	$\Delta w_{ij} = -\eta \cdot e_{j1}$ $\Delta w_{ij} = -\eta \cdot e_{j2}$	$e_n = s_i - w_{in}$ $e_n = \eta_i \cdot t_j - \eta_i \cdot w_{in}$
Self Organising Map	2-dimensional grid of output PEs	continuous	$net = \sum s_i \cdot w$	sigmoid	$\Delta w_{ij} = \eta \cdot \epsilon_j$	$e_j = s_i - w_{ij}$
Neocognitron	hierarchical multi-layer feed-forward	continuous	$net = \frac{(1 + S_e \cdot W_e)}{(1 + S_h \cdot W_h)} - 1$	linear	$\Delta w_{ij} = \eta \cdot \epsilon_j$	\times

Table 6.1 — Characteristics of Some Popular Neural Network Models

The *recall phase* f_1 , formed by the *propagation rule* and *activation function*, presents some variations among these neural models. With the exception of the Neocognitron model [56, 57], the propagation rule basically calculates the weighted sum of the input states. The Neocognitron model applies a more complex function of input states and weight states, which are classified into *excitatory* (S_e, W_e) and *inhibitory* (S_h, W_h) values. The activation function is usually restricted to either a threshold function, pseudo-linear function, or sigmoid families function.

The *learning phase* (f_2, f_3) differs considerably among the models. In algorithms such as Hopfield/Kohonen's associative memories [86, 99], the learning phase does not involve any error calculation (shown in Table 6.1 as \times), updating the weight values based upon the Hebbian rule [106]. Other models generally depend upon an *error calculation*. They can be as simple as the difference between a target value and the neuron's output state (Perceptron [122] and Delta Rule [106]), or can involve more complex

computations, such as the Back Propagation model [161] in which the error calculation requires the derivative of the activation function ($T'(net)$).

Simplicity is necessary to integrate as many PEs into a single integrated circuit as the technology permits. The search for generality and simplicity has resulted in a structure as shown in Figure 6.3a. The *Generic Neuron* model performs the neural computation based on: the set of state inputs S (received from other neurons); the state output s (sent to other neurons); and the group of error inputs E and error output e , provided to cope with models with backward propagation of errors. Internally, this model comprises a memory block for holding the weight values (W), and three basic functional blocks for implementing the specific computation required by the majority of neural models: the recall function f_1 , the weight updating function f_2 , and the error calculation function f_3 (see Table 6.1).

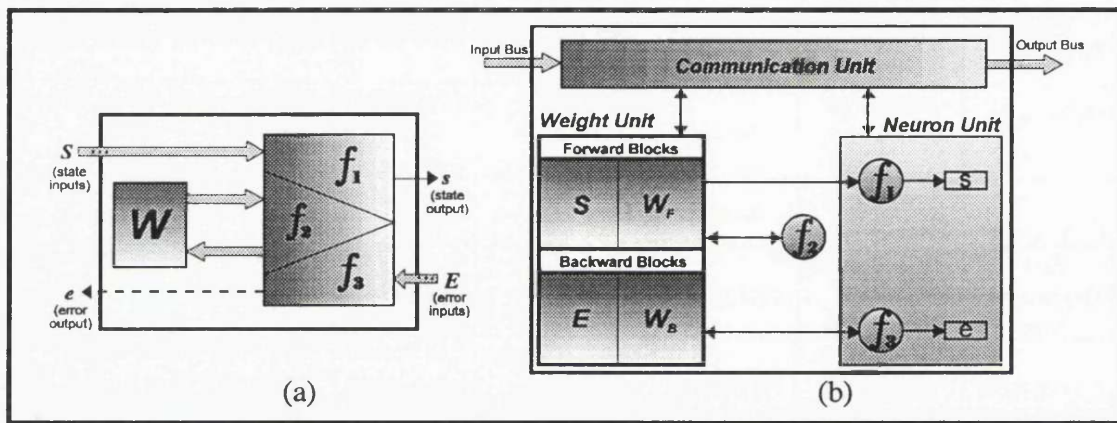


Figure 6.3 — The *Generic Neuron* Model

The simplicity of the *Generic Neuron* model is extremely suitable for silicon compilation, since neural models differ from each other essentially on the functions performed by f_1 , f_2 , and f_3 , along with the network's topology. The correspondent architectural framework of the *Generic Neuron* model is represented by the processing element (PE) shown in Figure 6.3b. The PE comprises three logical units, namely *communication*, *weight*, and *neuron* unit. The communication unit interfaces with the other two units and performs the reading and writing of input and output data. The weight unit executes the weight update during the learning phase, while the neuron unit calculates the neuron outputs (state and error).

The communication between PEs is accomplished through a broadcast bus (Figure 6.4a), which provides the system with important features such as *flexibility*, *expandability*, and *scalability* [190]. Firstly, flexibility is achieved since the bus interconnection can handle all possible complex topologies. Secondly, expandability on the

number of PEs is obtained by simply *plugging* new PEs onto the bus. Finally, scalability is attained since increasing the number of PEs integrated into the same chip does not affect its pin count. Moreover, to improve the bus performance, the system can be further expanded into several busses (Figure 6.4b), clustering together into the same bus PEs that receive the same data input.

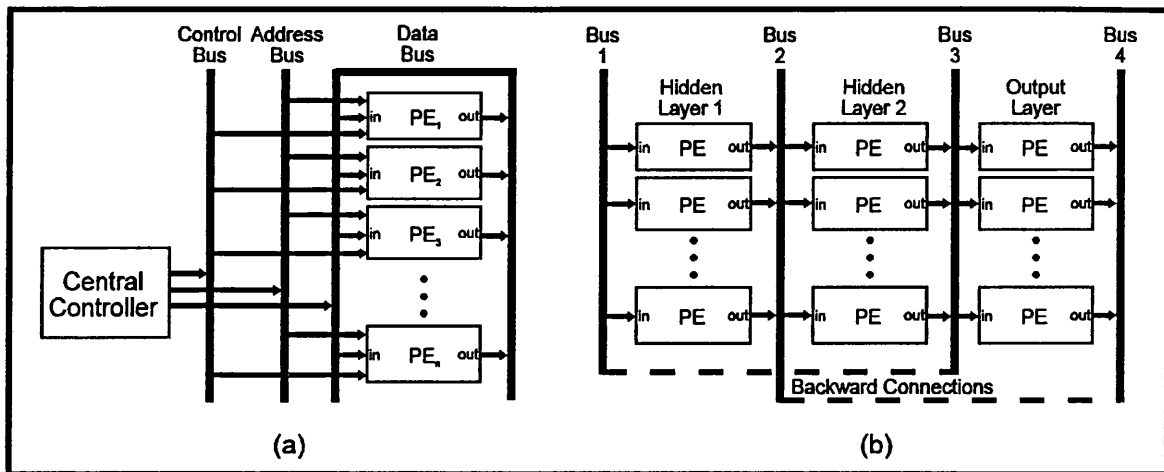


Figure 6.4 — Processing Elements Interconnection: (a) Single Bus; (b) Multi-busses

The PEs are commanded by the central controller through three busses (Figure 6.4a): *data*, *address*, and *control*. The *data bus* is used in two basic situations: (i) initialisation phase, during which the central controller loads into the PE's internal memory all necessary data values, including algorithm-dependent parameters and architecture-dependent parameters; and (ii) execution phase, when PEs send states and error values to other PEs in the network.

The *address bus* carries an identification of the PE that has exclusive access to the data bus. This arbitration is performed by the central controller in conjunction with the *control bus* according to a polling mechanism. Through this mechanism, the central controller waits for a ready signal from the PE, after which a new value is placed onto the address bus (the address of the PE in question). Therefore, the central controller can only modify the contents of the address bus after the PE has processed its output data (state or error value), and has activated the ready control signal.

The *control bus* contains all significant signals that direct the network's behaviour. It comprises signals from the central controller to each PE and vice-versa. These include signals such as reset, forward/backward phase definition, the PE's ready signal, etc.

6.4. Processing Element Organisation

Based upon the definition of the *Generic Neuron* model, the hardware implementation of the processing element (PE) can be organised into three units: *memory*, *communication*, and *execution*. Figure 6.5 shows the PE's internal structure and the following sub-sections examine each unit individually.

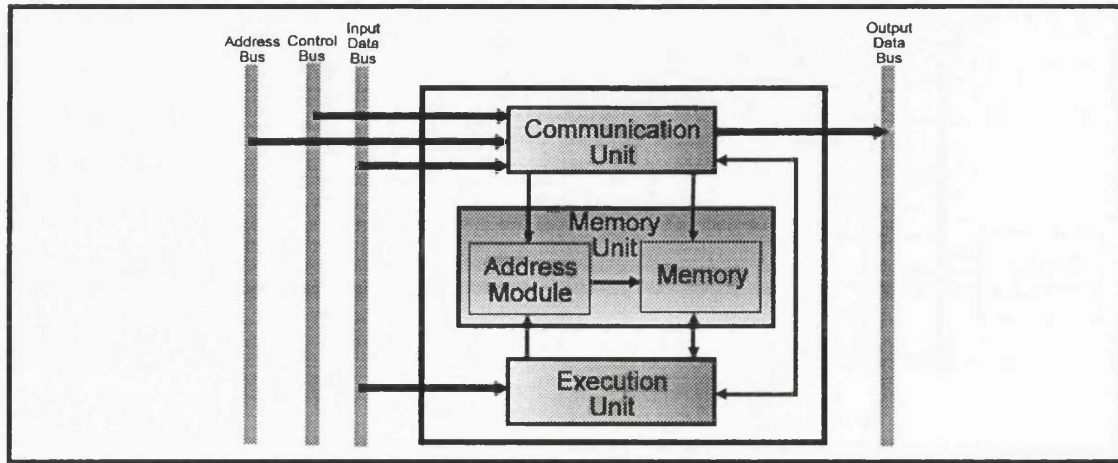


Figure 6.5 — Processing Element's Internal Organisation

6.4.1. Communication Unit

The purpose of the communication unit is to control the flow of data between a particular PE and the rest of the network. This unit is responsible for the following basic functions:

- initialise the PE's parameters, including initial weight values, target output patterns (for supervised learning), and some architecture-specific parameters;
- read and store input data into the appropriate memory block at specific instants, and issue control signals to the execution unit indicating when a state or error calculation should start; and
- transmit to the output data bus the calculated value from the execution unit, which can be either a state or an error value. In this case, the communication unit should also signal the rest of the network (and the central controller) that a legal value is being written onto the data bus.

The communication unit performs the above functions by interfacing with the off-chip broadcast busses, which are: the two data busses (input and output) that can be externally connected by a single bus (see Figure 6.4a); the address bus, and the control

bus, both driven by the central controller. To perform these functions, the communication unit is divided into two modules: *datapath* and *control*, as illustrated in Figure 6.6.

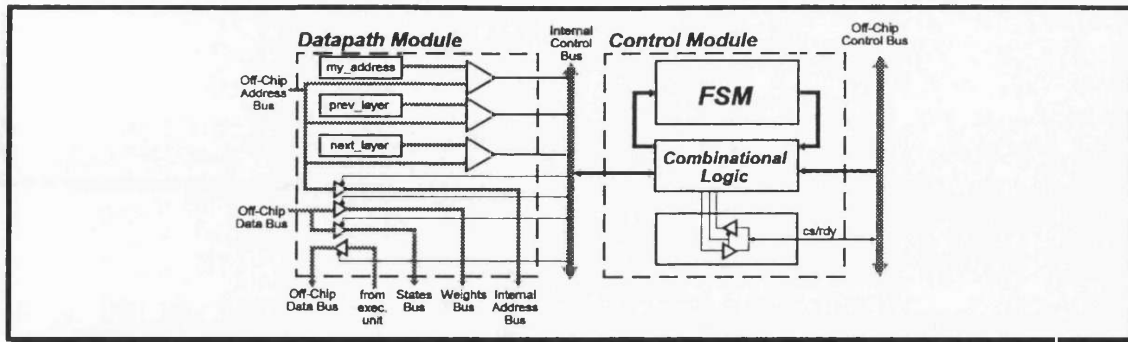


Figure 6.6 — Communication Unit's Internal Structure

The *datapath module* implements the PE's external bus arbitration. It is basically composed of comparators and registers, which determine when an output value (state or error) should be broadcasted onto the data bus, and verify when a valid data on the bus is addressed to the PE.

The *control module* generates all command lines to regulate the data transmission between the specific PE and the rest of the network. The control module is implemented by a FSM, which can be realised by a PLA, or random logic, according to the technique adopted by the low level synthesis tool used in conjunction with the NSC.

6.4.2. Memory Unit

The purpose of the memory unit is to store all relevant data required to execute neural network models. This data is accessed either externally or internally. External access is controlled by the communication unit to store all incoming data related to the PE. Internal access is controlled by the execution unit to perform the appropriate mathematical operations upon its stored data values.

Memory access can be done by the communication unit and the execution unit simultaneously. This is generally the case when the PE starts reading state values from other PEs, and as soon as at least one data set is obtained, the execution unit can start manipulating further data to perform the sum of products ($\sum s.w$). To achieve this parallelism, two-phase clock mechanism is employed.

The memory unit's internal structure is depicted in Figure 6.7. It comprises two major modules: the *storage module* and the *addressing module*.

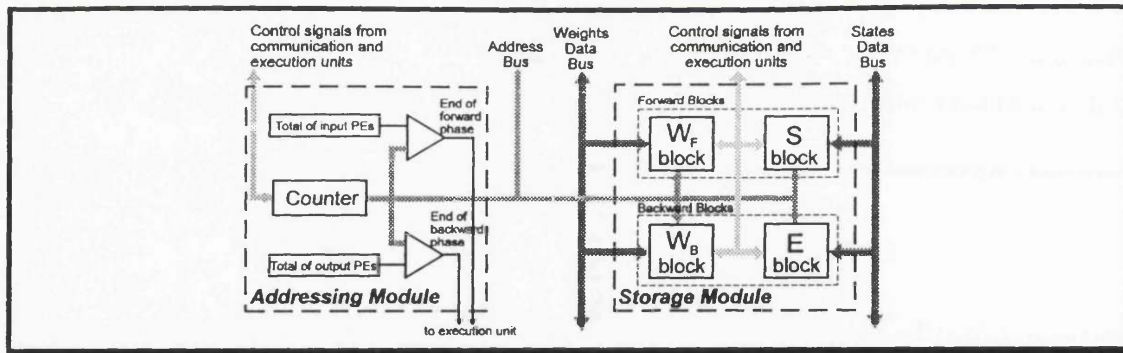


Figure 6.7 — Memory Unit's Internal Organisation

The *storage module* holds the basic data values required by the vast majority of neural models: input states and their associated synaptic weights. Although this module depends upon the neural algorithm and application being implemented, the *Generic Neuron* model identifies four basic blocks: a forward block, which stores the input states (S) and their weight values (W_F); and a backward block, which stores the backward flow of data, the input errors (E) and their related weights (W_B). Therefore, there are up to four blocks of memory, which contain the necessary data information to accomplish the recall and learning phase of the majority of neural models (see Table 6.1). Applications that do not require the learning phase, or back propagation of errors, can have the backward block excluded from the implementation of the storage module by the NSC.

The *addressing module* provides the mechanism to access all memory blocks integrated into the PE, according to the adopted neural algorithm. It consists of a counter, for controlling the sequential memory addressing, and up to two comparators, to determine when all relevant input values have been processed. One comparator is associated with the forward phase, and embodies a register that contains the total number of inputs. A second comparator, present only when learning with back propagation of errors is defined by the neural model, holds the number of backward inputs needed to implement the backward phase of such a neural model.

6.4.3. Execution Unit

The execution unit deals with the actual computation of the neural functions. It is responsible for executing all the three basic neural functions — f_1 , f_2 , and f_3 — of the *Generic Neuron* model (see Figure 6.3), according to the high level description provided by the application designer. The basic framework of this unit is depicted in Figure 6.8, which consists of two major modules: *datapath module* and *control module*.

The *datapath module* contains all the necessary blocks to perform the mathematical operations required by neural models. This module is ultimately configured

by the *NSC*, although some basic units can be identified. This includes: an *ALU*, which executes the basic operations such as addition and subtraction; an *Accumulator/Shifter module*, required to store intermediate results to perform the multiplication operation in conjunction with the *ALU*; some *registers*, to hold the algorithm-dependent parameters, such as the output state (s_j) and the output error (e_j); a *lookup table ROM*, responsible for implementing the activation function; a *multiplicand register*, a special register used to implement the multiplication algorithm; and finally, some *auxiliary registers*, required to store some intermediate results, which are used to process the neural function.

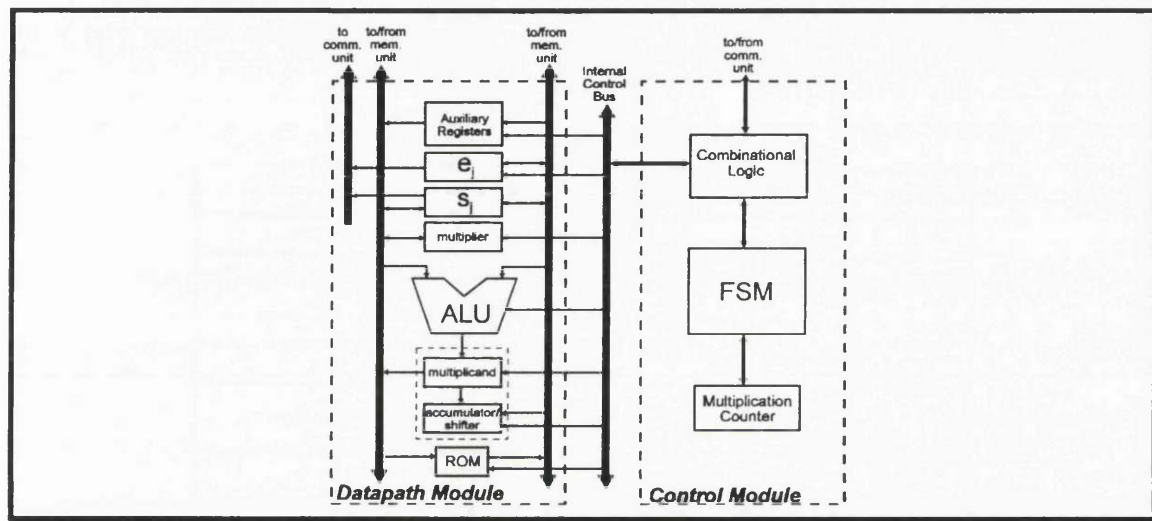


Figure 6.8 — Execution Unit's Internal Structure

The *control module* provides the necessary commands to execute all operations that are specific to the neural network algorithm. These include the propagation function followed by the activation function, the error calculation, and the weight update function. These commands are sent to the execution unit's data path, which effectively performs the computation. In addition, this module implements the multiplication operation, according to the Booth's algorithm [55].

6.5. VHDL Implementation of a Back Propagation Neural Chip

In this section, the design of a Back Propagation neural chip is coded in VHDL. This design is manually written in accordance to the *Generic Neuron* architecture and its PE's organisation, described above. The purposes of this implementation include:

- implementation of the basic PE corresponding to the *Generic Neuron* architecture;
- simulation and assessment of the effectiveness of the *Generic Neuron* model in realising the neural computation;

- assessment of the VHDL's suitability to specify neural chips;
- identification of the level of description at which the VHDL specification should be generated by the NSC; and
- building of an appropriate set of optimised VHDL library cells, which will be used by the NSC.

Clearly, these tasks represent the basis towards the automatic synthesis of neural chips. In addition, they also serve as a benchmark for assessing the quality of these chips, an issue that is analysed in chapter 8. The Back Propagation model has been chosen as the benchmark since it embodies enough complexity, in particular during the learning phase, to test the capabilities of the *Generic Neuron* architecture.


		
Rule/Layer	Hidden Layer	Output Layer
Propagation Rule	$s_i = \sum_k s_k \cdot w_{ki}$	$s_j = \sum_i s_i \cdot w_{ij}$
Error Calculation	$e_i = s_i(1-s_i) \cdot \sum_j e_j \cdot w'_{ij}$	$e_j = (t_j - s_j) \cdot s_j(1-s_j)$
Weight Update (forward)	$w'_{ki} = w_{ki} + \eta \cdot s_k \cdot e_i$	$w'_{ij} = w_{ij} + \eta \cdot s_i \cdot e_j$
Weight Update (backward)	$w'_{ij} = w_{ij} + \eta \cdot s_i \cdot e_j$	\times

Table 6.2 — PE Functionality for Hidden and Output Layers

Table 6.2 shows the basic computation required by each PE in the hidden and output layers of a Back Propagation neural network. It can be noted that, during the learning phase, PEs in the hidden layer perform different computations from PEs in the output layer. The basic difference relies on: (i) the error calculation; and (ii) on the extra weight update (backward) calculation performed by the PEs in the hidden layer. The difference between the error calculation function for neuron in the hidden and output layers is defined by the Back Propagation algorithm [161]. However, the calculation of backward weights is implementation-specific, being required for performance reasons. This is because the error calculation needs the weights from the connections between the PE in question and the PEs in the succeeding layer. Therefore, rather than broadcasting those weights, each PE in the hidden layer performs exactly the same weight update computation of the PEs in the next layer [190].

6.5.1. Communication Unit

As discussed in section 6.4.1, the communication unit is responsible for controlling the flow of data among PEs. The VHDL implementation of this unit is shown in Figure 6.9. It comprises the definition of the **comm_unit** entity, which specifies the input-output relation of signals flowing among the communication unit, the other two units, the other PEs in the network, and the central controller. This entity is described by a structural architecture comprising two components, namely **cu_datapath** and **cu_control**.

```

ENTITY comm_unit IS
  PORT(
    -- input-output pin specification (see Appendix D for full details)
  );
END comm_unit;

ARCHITECTURE structure OF comm_unit IS
  COMPONENT cu_datapath
    PORT(
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  COMPONENT cu_control
    PORT(
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  SIGNAL a,b,c,d,e,f,g: BIT;
  FOR b2: cu_control USE ENTITY WORK.cu_control(structure);
  FOR b1: cu_datapath USE ENTITY WORK.cu_datapath(behaviour);
BEGIN
  -- Components' instantiation and interconnection (see Appendix D for full details)
END structure;

```

Figure 6.9 — VHDL Implementation of the PE's Communication Unit

Data Path Module

This module performs two simple functions: (i) by using comparators, registers, and tri-state ports, the data path module analyses if the data circulating in the input bus is assumed to be read by its PE; if so, the appropriate internal busses are activated to write the incoming data into the PE's internal memory; and (ii) determines when the PE is allowed to broadcast its state or error value into the off-chip data bus, which is accomplished by comparing the incoming address bus data value with the **my_address** register (see Figure 6.6). Figure 6.10 shows the VHDL implementation of the communication unit's data path entity, called **cu_datapath**. This entity is implemented by a behavioural description of its architecture, which comprises the execution of seven concurrent processes. The first three processes (**test_my_add**, **test_prev_lay**, **test_next_lay**) can be realised through simple comparators at the structural domain, while the four remaining processes (**drive_addr**, **drive_out_bus**, **drive_weights_bus**, and **drive_states_bus**) describe the function of a tri-state buffer.

The three comparators analyse the external address bus and activate appropriate signals to the control module when certain conditions are detected. For instance, the process **test_my_address** monitors the address bus and activates the signal **ls_my_add**

when the central controller has loaded its address onto the external address bus, indicating that it is time to broadcast the results.

```

ENTITY cu_datapath IS
    PORT(
        -- input-output pin specification (see Appendix D for full details)
    );
END cu_datapath;

ARCHITECTURE behaviour OF cu_datapath IS
    CONSTANT my_address: mvl_vector(16 DOWNTO 1) := "0001000000000000";
    CONSTANT prev_layer: mvl_vector(4 DOWNTO 1) := "0000";
    CONSTANT next_layer: mvl_vector(4 DOWNTO 1) := "0010";
BEGIN
    test_my_add: PROCESS (add_bus)
    BEGIN
        IF (add_bus = my_address) THEN
            is_my_add <= '1';
        ELSE
            is_my_add <= '0';
        END IF;
    END PROCESS test_my_add;

    test_prev_layer: PROCESS (add_bus)
    BEGIN
        IF (add_bus(16 DOWNTO 13) = prev_layer) THEN
            prev_layer <= '1';
        ELSE
            prev_layer <= '0';
        END IF;
    END PROCESS test_prev_layer;

    test_next_layer: PROCESS (add_bus)
    BEGIN
        IF (add_bus(16 DOWNTO 13) = next_layer) THEN
            next_layer <= '1';
        ELSE
            next_layer <= '0';
        END IF;
    END PROCESS test_next_layer;

    drive_addr: PROCESS (en_mem_add, add_bus)
    BEGIN
        IF (en_mem_add = '1') THEN
            mem_add_bus <= add_bus;
        ELSE
            mem_add_bus <= "ZZZZZZZZZZZZZZZZ";
        END IF;
    END PROCESS drive_addr;

    drive_out_bus: PROCESS (en_outbus, z_bus)
    BEGIN
        IF (en_outbus = '1') THEN
            out_bus <= z_bus;
        ELSE
            out_bus <= "ZZZZZZZZZZZZZZZZ";
        END IF;
    END PROCESS drive_out_bus;

    drive_weights_bus: PROCESS (en_weightbus, in_bus)
    BEGIN
        IF (en_weightbus = '1') THEN
            weights_bus <= in_bus;
        ELSE
            weights_bus <= "ZZZZZZZZZZZZZZZZ";
        END IF;
    END PROCESS drive_weights_bus;

    drive_states_bus: PROCESS (en_staerrbus, in_bus)
    BEGIN
        IF (en_staerrbus = '1') THEN
            states_bus <= in_bus;
        ELSE
            states_bus <= "ZZZZZZZZZZZZZZZZ";
        END IF;
    END PROCESS drive_states_bus;
END behaviour;

```

Figure 6.10 — VHDL Implementation of the Communication Unit's Data Path

The tri-state buffers are commanded by the control module. They simply arbitrate the use of the busses to implement the communication protocol among PEs and to avoid bus conflicts. For example, when an external address corresponds to the memory location of data in the memory unit, then the communication unit's control module activates the `en_mem_add` signal. In this case, the process `drive_addr` inputs the external address bus into the PE. Otherwise, the internal bus `mem_add_bus` remains in tri-state logic.

Finally, the implementation of the `my_address` register requires some important considerations. Firstly, its content cannot be specified at *silicon compilation* time, since this would jeopardise the yield of identical chips. Secondly, it cannot be fully specified during initialisation time (as happens with other parameters), because the PEs must have *a priori* unique identification to avoid conflicts on the data bus. Therefore, this register

employs a *hardwired* technique, combined with configurable and fixed fields, as shown in Figure 6.11.

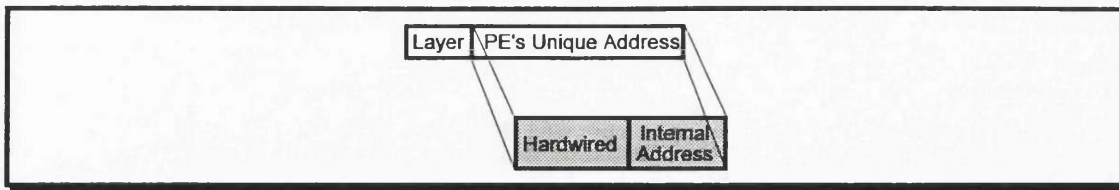


Figure 6.11 — Specification of the PE's **my_address** Register

The address of any PE in the network is composed of two fields: Layer and PE's Unique Address. The former identifies PEs in a particular layer, while the latter identifies PEs uniquely in the entire network, which comprises PEs inside a single chip as well as PEs among several identical chips. The PE's Unique Address field is further sub-divided in two fields: Hardwired and Internal Address. The former distinguishes PEs in different chips, while the latter distinguishes PEs inside a chip. The number of bits represented in the Hardwired field defines the maximum number of chips. Similarly, the number of bits represented by the Internal Address field defines the maximum number of PEs integrated in the same chip.

In this implementation, the **my_address** register is defined by a 16-bit register, where four bits are used by the Layer field, thus allowing a maximum of 16 layers, 8 bits are defined for the hardwired logic, thus allowing a maximum of 256 chips, and 4 bits are reserved for the Internal Address field, thus allowing a maximum of 16 PEs in each chip.

The implementation of the Back Propagation model requires two extra 4-bit registers: **prev_layer** and **next_layer**. In a fully connected network, data sent from PEs in preceding layers are to be read by all PEs in the next layer. Similarly, errors sent from PEs in succeeding layer are to be read by all PEs in the previous layer. The Layer field of **my_address** register, the registers **prev_layer**, and **next_layer** are all specified during the initialisation phase.

Control Module

The VHDL implementation of the control module is shown in Figure 6.12 and comprises the definition of the **cu_control** entity. This entity is described by a structural architecture composed of two components, namely **cu_fsm** and **cu_combinational**. The former is the communication unit's FSM, which receives inputs from its data path module, from the execution unit, and from external control signals. It generates output control signals that are driven to the memory unit (to control data movements to memory blocks),

to the execution unit (to control operations to be performed by this unit), and to the other PEs, either inside or outside the same chip.

```

ENTITY cu_control IS
  PORT(
    -- input-output pin specification (see Appendix D for full details)
  );
END cu_control;

ARCHITECTURE structure OF cu_control IS
  COMPONENT cu_fsm
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  COMPONENT cu_combinational
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  SIGNAL a, b, c: BIT;
BEGIN
  fsm_block: cu_fsm PORT MAP (
    -- input-output pin specification (see Appendix D for full details)
  );
  comb_logic: cu_combinational PORT MAP (
    -- input-output pin specification (see Appendix D for full details)
  );
END structure;

```

Figure 6.12 — VHDL Implementation of the Communication Unit's Control Module

The operations performed by this module can be classified in two groups: *initialisation* and *execution*. During initialisation time, which is signalled by the central controller, the memory counter (see section 6.5.2) is initially reset. Then the PE should read the forward weights and store them into the appropriate memory block. The central controller must send initial weight values sequentially, according to the addressing mechanism employed by the memory unit. Some additional parameters, such as learning rate, number of inputs, etc., are also sent by the central controller during the initialisation phase. This phase depends upon the neural algorithm, and its implementation should be adjusted by the NSC for each specific case. However, this adjustment only affects the implementation of the communication unit's control module. Therefore, the NSC is bound to synthesise different FSMs for each distinct implementation, while the core of the data path module remains unchanged.

During execution time, the control unit's data path is continuously monitoring the external address and data busses to decide when the data is assumed to be read by the PE. In this case, pertinent control lines are activated to instruct the writing of data into the appropriate memory block (forward or backward), in which the address is obtained by the external address bus. Furthermore, the control module signals to the execution unit the start of a data acquisition phase. After receiving this signal, the execution unit starts computing the appropriate neural computation (e.g., state or error calculation) in parallel with the communication unit. After finishing the computation, the execution unit sends a signal back to the communication unit, which waits for a specific moment to broadcast the value onto the output data bus. This moment is determined by the central controller through the polling mechanism mentioned earlier in this chapter.

The control module indicates to the central controller when the PE is ready to write data onto the data bus, which is done through the bi-directional **ready** signal. After the PE has calculated its output state the **ready** signal is activated. The central controller and the other PEs in the network also monitor this signal to decide whether the data being broadcasted into the bus is to be read by them. After the central controller has received this signal, it passes the control of the data bus to another PE.

6.5.2. Memory Unit

As mentioned before, the two-phase clock mechanism allows the memory to be accessed in parallel by both communication and execution units. In the Back Propagation prototype, during ϕ_1 the communication unit has access to the memory, while the execution unit accesses the memory during ϕ_2 . Therefore, as soon as the communication unit starts storing input states (sent by PEs in the previous layer), the execution unit can immediately initiate the necessary calculation, without having to wait for the remaining input values to be collected. The communication unit carries on storing input states, while the execution unit is calculating the propagation rule ($\Sigma s.w$).

```

ENTITY mem_unit IS
  PORT (
    -- input-output pin specification (see Appendix D for full details)
  );
END mem_unit;
ARCHITECTURE structure OF mem_unit IS
  COMPONENT storage_module
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  COMPONENT addressing_module
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  -- Definitions of internal signals (see Appendix D for full details)
BEGIN
  -- Components instantiation and interconnection (see Appendix D for full details)
END structure;

```

Figure 6.13 — VHDL Implementation of the Processing Element's Memory Unit

The VHDL implementation of this unit, shown partially in Figure 6.13, comprises the definition of the **mem_unit** entity. This entity comprises a structural representation of its architecture built by two components: the **storage_module** and the **addressing_module**.

Storage module

The four memory blocks necessary to implement the Back Propagation learning, have their control signals coming from the execution unit and communication unit. Two independent (internal) data busses are available to afford maximum parallelism: one for the weights (forward and backward) and another for the neuron states and errors. These busses are then connected to the execution unit, as described later. This is shown in Figure

6.14, through the port definition inside the **storage_module** entity. Note that states and errors share the same bus (state_bus), while forward and backward weights share the other bus (weight_bus), without compromising the overall performance. This is possible because the network is either executing the forward path (recall phase) or the backward path (learning phase).

The **storage_module** entity is defined by a structural architecture comprising the component ram, which is instantiated four times for each memory block. Weights and states are stored in pairs (in the same physical address). Therefore, a single address is able to load the required operands simultaneously onto the two busses. This simplifies enormously the addressing circuitry. In addition, the control of which memory block should be activated (weights forward or weights backward at one bus, and states or errors at the other bus) is defined either by the communication unit or the execution unit, according to their respective clock phases.

```

ENTITY storage_module IS
  PORT ( mem_add_bus      : IN wire_vector(ram_addr_lines DOWNT0 1);
        weight_bus       : INOUT wire_vector(data_bus_lines DOWNT0 1);
        states_bus       : INOUT wire_vector(data_bus_lines DOWNT0 1);
        cs_wgt_frww      : IN BIT;
        rd_wr_wgt_frww   : IN BIT;
        cs_wgt_bkw       : IN BIT;
        rd_wr_wgt_bkw    : IN BIT;
        cs_sta_frww      : IN BIT;
        rd_wr_sta_frww   : IN BIT;
        cs_sta_bkw       : IN BIT;
        rd_wr_sta_bkw    : IN BIT;
  );
END storage_module;

ARCHITECTURE structure OF storage_module IS
  COMPONENT ram
    PORT ( addr      : IN wire_vector(ram_addr_lines DOWNT0 1);
          data       : INOUT wire_vector(data_bus_lines DOWNT0 1);
          rd_wr      : IN BIT;
          cs         : IN BIT;
    );
  END COMPONENT;

  BEGIN
    weights_forward: ram PORT MAP(addr => mem_add_bus, data => weight_bus, rd_wr => rd_wr_wgt_frww, cs => cs_wgt_frww);
    weights_backward: ram PORT MAP(addr => mem_add_bus, data => weight_bus, rd_wr => rd_wr_wgt_bkw, cs => cs_wgt_bkw);
    states_forward: ram PORT MAP(addr => mem_add_bus, data => states_bus, rd_wr => rd_wr_sta_frww, cs => cs_sta_frww);
    states_backward: ram PORT MAP(addr => mem_add_bus, data => states_bus, rd_wr => rd_wr_sta_bkw, cs => cs_sta_bkw);
  END structure;

```

Figure 6.14 — VHDL Implementation of the Memory Unit's Storage Module

Addressing Module

Given the simplicity in which memory blocks are organised, the addressing module consists of a sequential pointer (counter) for each memory block: forward and backward. A comparator is given for each block, which compares the counter's value with an internal parameter containing the total number of inputs. The value for this parameter is defined during initialisation time.

Figure 6.15 shows partially the VHDL implementation of this module, which is defined by the **addressing_module** entity. This entity describes a structural architecture encompassing the following components: comparator, counter1, tri_state_ram_addr, and or2gate. The description of these components is given in Appendix D.

```

ENTITY addressing_module IS
  PORT (
  );
END addressing_module;

ARCHITECTURE structure OF addressing_module IS
  COMPONENT comparator
    PORT (
      -- input-output pin specification
    );
  END COMPONENT;
  COMPONENT counter1
    PORT (
      -- input-output pin specification
    );
  END COMPONENT;
  COMPONENT tri_state_ram_addr
    PORT (
      -- input-output pin specification
    );
  END COMPONENT;
  COMPONENT or2gate
    PORT (
      -- input-output pin specification
    );
  END COMPONENT;

  -- Internal signals and interconnection (see Appendix D for full details)
BEGIN
  -- Components' instantiation and interconnection (see Appendix D for full details)
END structure;

```

Figure 6.15 — VHDL Implementation of the Memory Unit's Addressing Module

The addressing module circuitry is connected to a tri-state port, `tri_state_ram_addr`, since the memory blocks can also be accessed externally through the external address bus, which is controlled by the communication unit. Whenever a PE broadcasts a state onto the data bus, the sender's unique identification is loaded onto the address bus. Thereby, the communication unit can directly associate the incoming data with its correct memory location. This simplifies considerably the mechanism for loading data into the memory, and permits parallel access of the memory by the communication unit, using the external address bus (during ϕ_1), and by the execution unit, using the internal addressing circuitry (during ϕ_2).

6.5.3. Execution Unit

The VHDL implementation of the execution unit is shown in Figure 6.16, and comprises the definition of the `exec_unit` entity. The structure of this entity includes an architecture composed of two components: `eu_datapath` and `eu_control`.

```

ENTITY exec_unit IS
  PORT (
    -- input-output pin specification (see Appendix D for full details)
  );
END exec_unit;

ARCHITECTURE structure OF exec_unit IS
  COMPONENT eu_datapath
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  COMPONENT eu_control
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;

  -- Internal signals specification (see Appendix D for full details)
BEGIN
  b1: eu_datapath PORT MAP (
    -- input-output pin specification (see Appendix D for full details)
  );
  b2: eu_control PORT MAP (
    -- input-output pin specification (see Appendix D for full details)
  );
  eu_reset_ptr <= reset_aux1;
END structure;

```

Figure 6.16 — VHDL Implementation of the Processing Element's Execution Unit

Data Path Module

The VHDL implementation of this module describes a structural architecture composed of several components, as shown in Figure 6.17. These components comprise: multi-port register, an ALU, a shifter, a ROM, tri-state buffers, and gates.

```

ENTITY eu_datapath IS
  PORT (
    -- input-output pin specification (see Appendix D for full details)
  );
END eu_datapath;

ARCHITECTURE structure OF eu_datapath IS
  COMPONENT tri_state_data_bus
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  COMPONENT tri_state_ram_addr
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  COMPONENT tri_state_rom_addr
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  COMPONENT reg1x1
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  COMPONENT reg1x2
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  COMPONENT reg1x3
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  COMPONENT reg0x1
    PORT (
      a : OUT wire_vector (data_bus_lines DOWNT0 1));
  END COMPONENT;
  COMPONENT reg2x1
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  COMPONENT reg2x1mr
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  COMPONENT reg3x3
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  COMPONENT alu
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  COMPONENT shift_n
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  COMPONENT rom
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  COMPONENT one_16
    PORT (
      one_16 : OUT wire_vector(16 DOWNT0 1));
  END COMPONENT;
  COMPONENT gnd_16
    PORT (
      gnd_16 : OUT wire_vector(16 DOWNT0 1));
  END COMPONENT;
  COMPONENT latch_db
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  COMPONENT or2gate
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  -- Internal signals specification (see Appendix D for full details)
BEGIN
  -- Instantiation and interconnection (see Appendix D for full details)
END structure;

```

Figure 6.17 — VHDL Implementation of the Execution Unit's Data Path Module

The basic component of the data path module is the ALU, since it is responsible for operations like multiplication, addition, and subtraction, which are commonly found in neural algorithms. Associated with the ALU, several registers are used to hold intermediate results. All registers are multi-port, since this optimises considerably the

design of the data path. Regarding silicon compilation, the NSC should analyse the data flow of the *nC* program and define the number of ports each register should have.

Control Module

The VHDL implementation of this module describes a structural architecture composed of two components: **eu_fsm** and **eu_combinational**. A counter, necessary to implement the Booth's multiplication algorithm is incorporated into the **eu_combinational** component. Like the communication unit's control module, the FSM is also implemented by a behavioural architecture, while the combinational logic of the module is described in the structural domain.

```

ENTITY eu_control IS
  PORT (
    -- input-output pin specification (see Appendix D for full details)
  );
END eu_control;

ARCHITECTURE structure OF eu_control IS
  COMPONENT eu_fsm
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  COMPONENT eu_combinational
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;
  -- Internal signal definitions (see Appendix D for full details)
BEGIN
  fsm_block: eu_fsm PORT MAP (
    -- input-output pin specification (see Appendix D for full details)
  );
  comb_block: eu_combinational PORT MAP (
    -- input-output pin specification (see Appendix D for full details)
  );
  reset_acc_mptr <= int_reset_mptr;
END structure;

```

Figure 6.18 — VHDL Implementation of the Execution Unit's Control Module

6.6. Processing Element

Considering that all three units have been designed, the construction of a single PE is accomplished by instantiating the appropriate units into a single entity, which for a neuron in the hidden layer is called **GN_hidden** (see Figure 6.19).

The VHDL **GN_hidden** entity is described by a structural architecture comprising three components, namely **comm_unit**, **mem_unit**, and **exec_unit**, which are described before.

```

ENTITY GN_hidden IS
  PORT (
    in_bus      : IN  mvl_vector(data_bus_lines DOWNT0 1);
    out_bus     : OUT  mvl_vector(data_bus_lines DOWNT0 1);
    add_bus     : IN  mvl_vector(ram_addr_lines DOWNT0 1);
    reset       : IN  BIT;
    load        : IN  BIT; -- 0=Initialisation; 1=Execution
    frw_bkw     : IN  BIT;
    cs_Fdy      : INOUT wire;
    lrb_rcl     : IN  BIT;
    phi1        : IN  BIT;
    phi2        : IN  BIT;
  );
END GN_hidden;

ARCHITECTURE structure OF GN_hidden IS
  COMPONENT comm_unit
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;

  COMPONENT mem_unit
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;

  COMPONENT exec_unit
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;

  SIGNAL a_bus, b_bus : wire_vector(data_bus_lines DOWNT0 1);
  -- Internal signals specification (see Appendix D for full details)

  SIGNAL end_frw_ph, end_bkw_ph : BIT;
BEGIN
  cu_hidden: comm_unit PORT MAP (
    -- Instantiation and interconnection (see Appendix D for full details)
  );

  mu_hidden: mem_unit PORT MAP (
    -- Instantiation and interconnection (see Appendix D for full details)
  );

  eu_hidden: exec_unit PORT MAP (
    -- Instantiation and interconnection (see Appendix D for full details)
  );
END structure;

```

Figure 6.19 — VHDL Implementation of a Hidden Layer Processing Element

6.7. Back Propagation Neural Chip

Finalising the design of the Back Propagation neural chip, Figure 6.20 presents the design comprising 3 PEs from the hidden layer and 5 PEs from the output layer.

Although the preceding sections have described only a hidden layer PE, clearly the design of an output layer PE is very similar. The description of the output layer PE would represent a repetition of the description above. Therefore its complete design will be omitted. It is assumed that a VHDL entity called **GN_output** has been defined.

The VHDL implementation of the Back Propagation neural chip comprises description of a structural architecture composed of two components: **GN_hidden** and **GN_output**. The instances of these two components are then created and interconnected among them forming a network of PEs connected through busses, according to the definition of the *Generic Neuron* architecture.

```

ENTITY BP_neural_chip IS
  PORT (
    in_bus      : IN   mvl_vector(data_bus_lines DOWNT0 1);
    out_bus     : OUT  mvl_vector(data_bus_lines DOWNT0 1);
    add_bus     : IN   mvl_vector(ram_addr_lines DOWNT0 1);
    reset       : IN   BIT;
    load        : IN   BIT; -- 0=Initialisation; 1=Execution
    frw_bkw     : IN   BIT;
    cs_fdy      : INOUT wire;
    lra_rcl     : IN   BIT;
    phi1        : IN   BIT;
    phi2        : IN   BIT;
  );
END BP_neural_chip;

ARCHITECTURE structure OF BP_neural_chip IS
  COMPONENT GN_hidden
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;

  COMPONENT GN_output
    PORT (
      -- input-output pin specification (see Appendix D for full details)
    );
  END COMPONENT;

  SIGNAL a_bus, b_bus : wire_vector(data_bus_lines DOWNT0 1);
  -- Internal signals specification (see Appendix D for full details)

  SIGNAL end_frw_ph, end_bkw_ph : BIT;
BEGIN
  gn_hidden1: GN_hidden PORT MAP (
    -- Instantiation and interconnection (see Appendix D for full details)
  );

  gn_hidden2: GN_hidden PORT MAP (
    -- Instantiation and interconnection (see Appendix D for full details)
  );

  gn_hidden3: GN_hidden PORT MAP (
    -- Instantiation and interconnection (see Appendix D for full details)
  );

  gn_output1: GN_output PORT MAP (
    -- Instantiation and interconnection (see Appendix D for full details)
  );

  gn_output2: GN_output PORT MAP (
    -- Instantiation and interconnection (see Appendix D for full details)
  );

  gn_output3: GN_output PORT MAP (
    -- Instantiation and interconnection (see Appendix D for full details)
  );

  gn_output4: GN_output PORT MAP (
    -- Instantiation and interconnection (see Appendix D for full details)
  );

  gn_output5: GN_output PORT MAP (
    -- Instantiation and interconnection (see Appendix D for full details)
  );

END structure;

```

Figure 6.20 — VHDL Implementation of a Simple Back Propagation Chip

6.8. Summary

This chapter has proposed a general framework for the NSC, built upon existing neural network programming environments. This proposal maintains these environments unchanged, and adds a hardware route aiming the automatic synthesis of ASNNs. The hardware route is made technology independent by using VHDL. Therefore, potential new technologies or different techniques can be later incorporated into the system.

A prototype of the NSC's target architecture, the *Generic Neuron*, has been coded in VHDL for the Back Propagation network. The design has been developed at the RTL representation, following a mixed approach of behavioural and structural domains. In particular, FSMs, memories, and counters are implemented in the behavioural domain, while the remainder of the design is implemented in the structural domain. This is compatible with the state of the art low level CAD tools, which will ultimately generate the mask layout of the neural chips.

Most of the hardware structures designed are parameterised, so that they can be used as part of the VHDL library, which will be utilised by the *NSC*.

This mixed approach is greatly favoured by the richness offered by VHDL, permitting the description of a hierarchical design, in which several architectures for the same entity can be defined, each with a different degree of detail. This also permits the simulation of the design, thus providing a way to test the entire approach.

Chapter 7

NSC Implementation

*This chapter describes in detail the implementation of the NSC. The necessary translation steps are described, starting from a high level of abstraction, represented in *nC*, and going to the actual hardware structure. This approach focuses on the high level synthesis part of the silicon compilation process. The synthesised hardware structure is specified in VHDL, the IEEE standard hardware description language, from which commercial low level synthesis tools can be applied to generate the final layout.*

7.1. Overview

The design of modern software compilers is carried out according to standard techniques [11], which are adopted by well-known compilers [172]. These techniques are generally organised in two logical activities:

- *front-end* — composed of lexical and syntactic analysis, the creation of a symbol table, semantic analysis, and the generation of an intermediate code. These phases primarily depend upon the source language and are largely independent of the target code.
- *back-end* — includes code optimisation and code generation. These phases are generally independent of the source language; they basically depend upon the target code and the intermediate language.

Equally, the design of the NSC can also be divided in two main activities: *software analysis* of the *nC* language followed by the generation of an intermediate representation; and *hardware synthesis* of ASNNCs starting from this intermediate representation.

A clear correspondence can be made between the design of software compilers and the design of the NSC. The software analysis task of the NSC corresponds to the software compiler's front-end, since both are implemented using basically the same techniques. Similarly, the NSC's hardware synthesis task corresponds to the software compilers' back-end. In this case, however, the techniques employed are very distinct. Hardware synthesis involves the *creation* of structures that implement a particular function. Such a

task has no similar in software compiler's techniques. This has several implications, which are analysed throughout this chapter.

The *NSC*'s front-end is divided into the following tasks:

- *nC* compilation and transformation steps — involves parsing the *nC* rules, constructing syntax trees for the *nC* language, and applying transformations upon the *nC* data structures and the syntax trees;
- construction of the intermediate code representation (*ICR*) — a graph-based structure incorporating control and data flow information (a CDFG);

The *NSC*'s back-end is hardware-oriented, being implemented by the tasks below:

- transformations upon the *ICR* — includes partitioning the *ICR* graph structure into simpler sub-graphs, and applying optimisations that carry out constant propagation and storage elimination;
- definition of a target architecture — involves choosing a suitable architecture (the *Generic Neuron* [190]) and making assumptions about its internal organisation, clocking scheme, etc., which are paramount to drive the hardware synthesis algorithms;
- hardware synthesis — comprises the synthesis of the processing element's (PE) data path structure (performed by hardware allocation of storage elements, operators, and interconnections, and definition of the scheduling) and synthesis of the PE's control structure;
- module generation — builds an instance of a particular functional unit or storage element according to the hardware structure synthesised before; and
- generation of a VHDL description — completes the design by providing a complete specification of the neural chip, according to the modules required to implement a particular PE, and according to the neural network partitioning strategy for mapping specific neurons onto PEs.

The *NSC*'s front-end and back-end, briefly introduced above, are thoroughly described in the following sections.

7.2. *nC* Compilation and Transformation Steps

The compilation of the *nC* program consists of two basic tasks: parsing its input, which creates a syntax tree representation entirely conforming with the *nC* syntax; and transformation of the *nC* syntax tree into simpler structures.

The primary goal in transforming *nC* at early stages of the compilation process is to analyse the tree structure and eliminate some unnecessary tree nodes, which are in principle independent of its structure, but can introduce large overheads after the hardware has been synthesised. These superfluous tree nodes are created either from the user's specification or from the language's structure itself. In either case, these redundancies result in more complex control units (since more clock cycles are required), resulting in larger and slower circuits.

Some of the transformations performed are very similar to the ones performed by optimising compilers [11]. Some are very specific to the *nC* language definition.

7.2.1. *nC* Parsing

The first task for creating a hardware structure of a particular neural network is to analyse the *nC* program and extract every relevant piece of information regarding the network's interconnection topology and neurons' functionality. As discussed in section 4.3, a *nC* program consists of several components, in which some of them are not possible to synthesise into hardware structures. Although these components are required for simulation purposes, they are not important for hardware synthesis. Therefore, the analysis of what part of the *nC* language is relevant or not for hardware synthesis is made at the early stages of the parsing phase.

By analysing the *nC* application control definition (see section 4.2.2), it is seen that one of the first actions is to initialise the system data structure through the `connect` and `build_rules` routines. Due to the complexity involved in these functions, a compilation step would lead to unnecessary and extremely inefficient hardware structures. Therefore, it is required to envisage an effective mechanism that selectively compiles only the important characteristics of a neural network described in *nC*.

Two alternative mechanisms which, applied at run time, are provided: generation of the *nC_code* [131], and direct compilation. The first involves the transformation of the initialised **system** data structure into a *nC*-based program, called *nC_code*. This program is a special version of *nC*, in which the information about the network is held as explicit flat data structures. Essentially, it is an image of the memory in the form of a *nC* program, thus

eliminating the need for the two initialisation routines, i.e., `connect` and `build_rules`. In this case, the actual compilation starts from `nC_code` rather than `nC`. The second mechanism simply reads the **system** data structure held in memory and compiles the necessary `nC` rules. Both mechanisms have been implemented and are seen as conceptually equivalents. Therefore, throughout this chapter, the discussions of the compiler will make no distinction upon which of these two mechanisms are employed.

Before `nC` is parsed, several preliminary steps are undertaken. These basically involve the initialisation of several internal data structures to conform with the `nC` syntax. Basically, the `nC` syntax tree is constructed and initialised with default values which correspond to all pre-defined `nC` statements (the subset of C), built-in functions (such as `dp`, `lookuptbl`, `thr_tbl`, etc.), and data types (including the **system** data structure).

Parsing is done selectively, based on the analysis of the **system** data structure. According to the rules defined by the user at the network, layer, cluster, and neuron levels, the specific functions are parsed to extract data and control flow. Neurons with identical functionality — typically neurons that are grouped into the same cluster or layer — are compiled only once and instantiated several times. However, rules defined at the network level are not compiled, since they are usually directed to the control of the entire network. An example includes the function that calculates the tolerance of a particular model to decide when the algorithm should stop. This function is not performed by the PE. Instead, the central controller is responsible for commanding the entire network of PEs (see chapter 6).

Figure 7.1 shows a simplified version of the algorithm for the selective parsing of `nC` rules. The processing of generic and extended parameters consists in performing local transformations upon these data, and is described in the next sub-section. If rules are not defined at the neuron level, then the same procedure is performed at the cluster level. In this case, rules defined for a particular cluster are applied for every neuron in this cluster.

```

foreach neuron do
  foreach RULE do
    process (generic_parameters)
    process (extended_parameters)
    compile RULE
    Incr (neuron_rules)
    If (neuron_rules = 0) do
      (repeat procedure for cluster level and layer level)
    end
  end
end
end

```

Figure 7.1 — `nC` Compilation at the Neuron Level

Data type analysis is also performed during the parsing phase. Each generated *nC* tree node has an associated code specifying its type, which can be a data type, a variable, an expression, or a statement. For each of these elements, the tree node holds specific information of the type. For example, a data type node for strings includes a pointer to the data stream and its length, while a statement node for the *if* clause contains links to other tree nodes corresponding to the *condition* branch, the *then* branch, and to the *else* branch. Figure 7.2 illustrates the syntax tree structure for a simple assignment statement of a generic parameter in *nC*. Note that the tree represents directly the syntactic structure of the language. Even for a simple statement such as an assignment, the complexity involved in the manipulation of the *nC* data structures is evident. This poses a severe penalty for a direct mapping of *nC* variables onto hardware structures. Therefore, a simplification mechanism becomes extremely important.

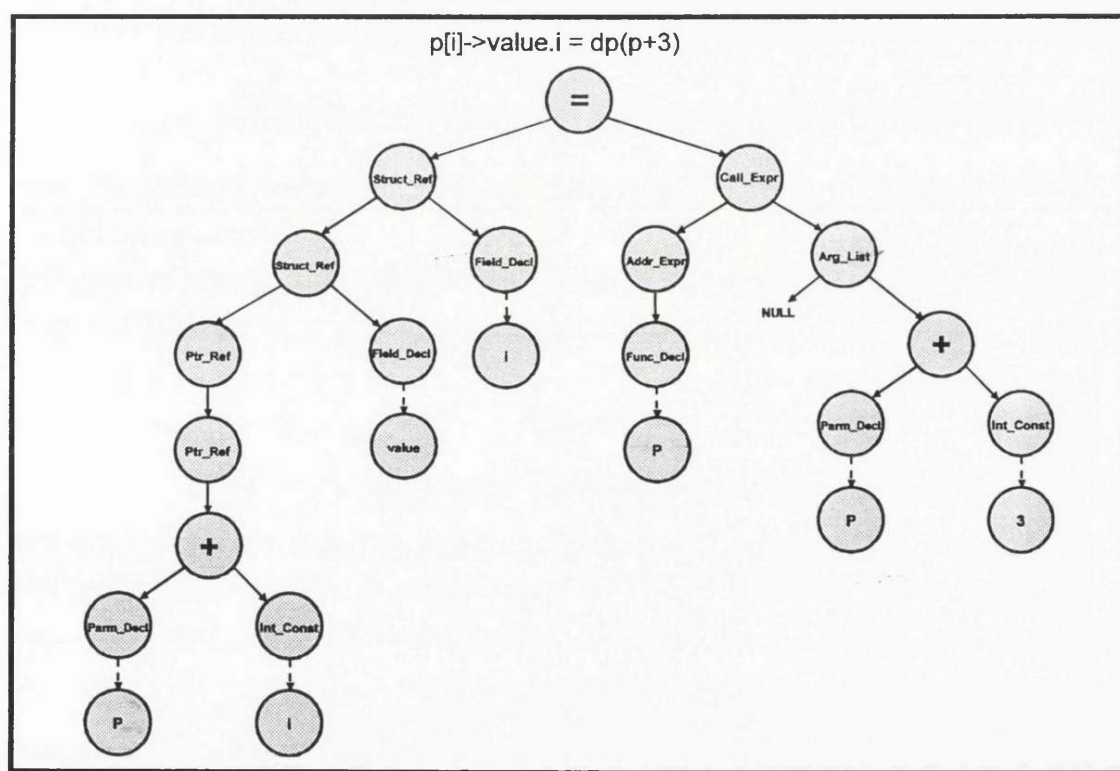


Figure 7.2 — Example of a *nC* Syntax Tree Structure for an Assignment Operation

7.2.2. *nC* Data Structure Transformations

The inherent complexity of the *nC* tree syntax, suggested by Figure 7.2, is resulted from the *nC* data structure definition. In particular, the **RULE** data structure along with its *para_list* structure requires the definition of every *nC* rule (or function) following a strict premise: they must manipulate data defined over a linear array structure (see section 4.2.1). Therefore, in order to avoid every data or parameter in *nC* being

represented by an array (which would lead to a single memory structure in hardware), the following transformations rules are performed upon the `para_list` structure:

- Any generic parameter, associated with either the neurons' states (such as the output, error, accumulation, and target states), synapses, or with any model-dependent parameter (such as learning rate), is mapped onto a *variable* of sub-type *register type*. During data path synthesis, these parameters are generally bound to register structures.
- The first item in the extended parameter field refers to its size, and therefore contains a constant value for each specific rule. If the constant is a non-zero value, then it is mapped onto a *constant* type, which can be bound, during data path synthesis, to a signal, counter, or register structure. However, if the constant is zero, it means that the rule in question does not require extended parameters, and the size parameter will be automatically eliminated during data path synthesis.
- The rest of the extended parameters are mapped onto a *variable* type, which are generally bound to memory structures.

It is important to note that the implementation of these transformations are only possible due to the *tag* mechanism incorporated into the **RULE** data structure (see Figure 4.4). Equally important is that these transformations permit a closer relation between the *nC* data and data path structures defined during the development of the VHDL neural chip, described in the previous chapter.

7.2.3. *nC* Syntax Tree Transformations

Following the transformations performed upon the *nC* data structures, there are several possible local transformations at the syntax tree level that not only optimise the behavioural level described in *nC*, but also simplify the synthesis of a hardware structure. In both cases, the result is a more compact circuit synthesised from the specified behaviour described in *nC*.

Nodes defined in the syntax tree are first checked against the use of some special cases. For instance, if the user specifies that the activation function is accomplished through a lookup table, then the correspondent variable is mapped onto a ROM structure, in which values are obtained by the user's activation function definition. Similarly, a built-in function call is automatically transformed into its appropriate code, and may or may not be compiled. For instance, a call to the function `lookuptbl` is not compiled, but simply transformed into an array reference. Conversely, a call to the built-in function `dp` results in compilation of this function.

The transformations consist in visiting every node and applying a set of heuristic rules that try to simplify the mapping of *nC* into the hardware structure that implements it. The set of heuristics are defined below:

- If a RAM type is added to a constant (which results from a pointer reference, indexed by a constant in *nC*, such as `p[1]`), then the whole forest is simply transformed in a register node as shown in Figure 7.3.a. According to the constant and to the tag defined by this parameter, an appropriate name (which has been defined previously) is assigned to the register.
- If a ROM type is added to a register (which results from a pointer reference, indexed by a variable in *nC*, such as `lookupbl(p[1])`), then the whole forest is simply transformed in the ROM node (used as operand) indexed by the register, as shown in Figure 7.3.b. In this case, the register is simply used to address the ROM memory, and the add operation is eliminated.
- If a RAM type operation is added to a register (which results from an array indexed by a variable in *nC*, such as `p[i]`), then the whole forest is simply transformed in the RAM node indexed by the register, similar to the previous transformation rule and as shown in Figure 7.3.d.
- Another transformation strategy looks for the *nC* construct such as `p[i+1]`, which is parsed as a partial tree as shown in Figure 7.3.c and replaced by a simple RAM node addressed by the variable `i`. This is an immediate consequence of the way extended parameters in *nC* are arranged, that is, as pairs or group of data (i.e., `num_of_ext_parm = 2`; see for example Figure 4.5). The exact RAM name is obtained again through the tag mechanism employed in all *nC* data structure.
- A more complex transformation strategy looks for the *nC* statement `for` and checks whether a control variable has been defined. If so, it checks what kind of statement has been defined and carries out an analysis on the whole loop, so that a *for* statement like:

```
for (i = 3; i < (2 * size) + 3; i += 2)
```

is transformed in an equivalent loop such as:

```
for (i = 0; i < size; i += 1)
```

which is necessary to comply with the two previous transformation steps. In other words, the original loop performs calculation upon a `para_list` structure (see Figure 4.5), in which the head of the list is the fourth element (`i = 3`) and data are

processed two-by-two ($i += 2$). Since this structure has been transformed by the above mechanisms, this loop must be adjusted accordingly, resulting in the second loop above.

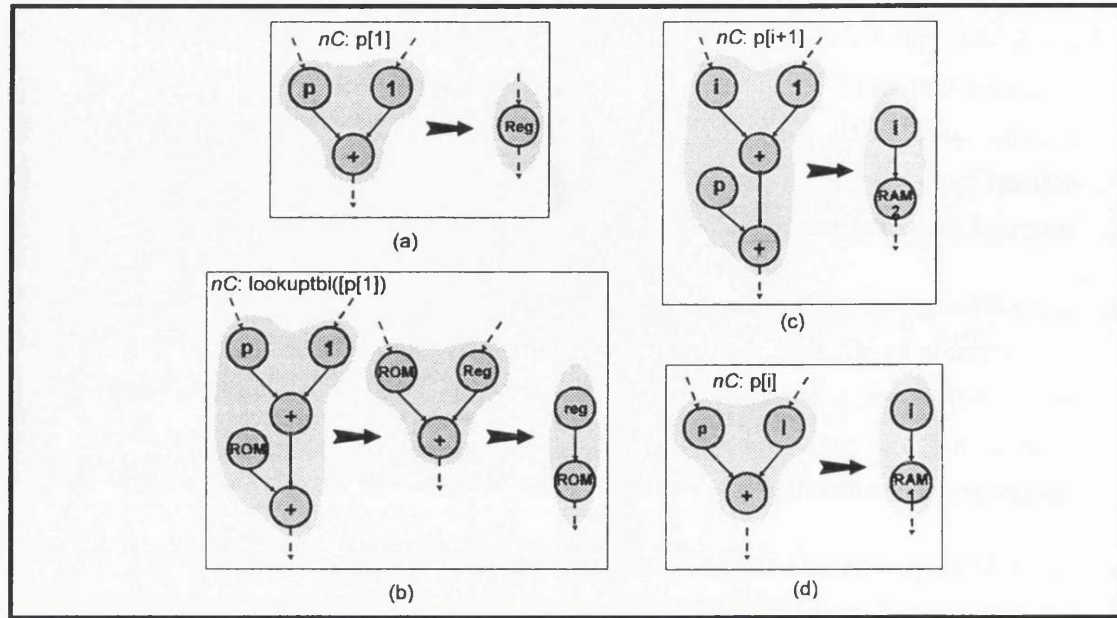


Figure 7.3 — Transforming Memory References into Registers

The success of the above high level transformations in every nC program can be guaranteed due to the rigid way (which is defined by the **system** data structure) that the algorithm (and application) designer must follow to correctly implement a particular neural network application in nC .

Along with the above optimisations, during the parsing of the syntax tree level some software-like optimisations are performed, such as, *constant folding*, and *global transformations*.

Constant folding is done at compile time by making local and global transformations. Local transformations include the replacement of constant expressions by its evaluated result. For example, the statement $x := 5 * 2$ is replaced by the value 10, while statements like $y + 0$ and $y * 1$ are simply replaced by y . Global transformations involve the analysis of the data flow. In the above examples, every time x appears in the program, its reference is replaced by the pre-calculated value 10, as long as x is not assigned elsewhere between the first and the last assignment of x .

Finally, in-line subroutine expansion is performed, which comprises replacement of function calls by their code. This transformation creates more opportunities for hardware

optimisations, since operations and storage elements can be merged, which also permits the exploitation of parallelism.

In summary, the first part of the *NSC*'s front-end is devoted to the development of several transformation rules, which are closely related to the intrinsic *nC* syntax and semantic definition. Without all these transformations described above, the hardware synthesis tools would tend to create a complex hardware structure, particularly because of the way *nC rules* are defined and data is manipulated (i.e., as array elements).

The second part of the *NSC*'s front-end involves the transformation of the *nC* syntax tree into a graph-based structure.

7.3. Intermediate Code Representation

The intermediate code representation (*ICR*), which is constructed from the *nC* syntax tree, constitutes the instrument for the hardware synthesis steps that follow. It is essentially a control and data flow graph (CDFG), represented by a directed graph with predecessor-successor information. The *ICR* graph is defined as:

$$ICR = (V, E)$$

where $V = \{v_1, v_2, \dots, v_n\}$ is the set of vertices (or nodes), and $A = \{a_1, a_2, \dots, a_m\}$ is a set of directed arcs between two nodes.

A vertex $v = \{\text{Type}, \text{Inps}, \text{Out}, \text{Attr}\}$ consists of:

- a **Type** field, which can represent either an operation or a storage element. Operations include add, subtract, multiply, assign, compare, increment, etc. Storage elements can be either a register, RAM, ROM, or counter;
- a set of input arcs **Inps**, which can be none, one or two arcs, according to the **Type** field in question;
- an output arc **Out**, which can be none or a single arc; and
- an attribute field **Attr**, which includes several sub fields according to the node's **Type** field. Examples include the number of bits in a storage element, number of clock cycles to accomplish a determined operation, etc.

An arc $a = \{\text{Src}, \text{Dst}\}$ comprises:

- a single source node **Src**; and
- a single destination node **Dst**.

The intermediate code representation (*ICR*), in its textual form, is a low level format for the specification of neural networks aimed to be mapped into hardware. In its simplest form, the *ICR* describes the network at the neuron level of the *nC system* data structure, since, at the hardware level, this is the only useful information concerning the implementation of processing elements. The format also provides an optional part that describes the network with layers, clusters, neurons, and synapses, similar to the *nC system* data structure. Since this data structure provides every possible information about the neural network, it may also be used to synthesise the functionality of the central controller. This can lead to a possible extension of the *NSC*, in which a system level synthesis can be designed (see chapter 3).

The syntax of the *ICR* is formally presented in Figure 7.4. It is a very simple format consisting of three basic parts:

- Explicit specification of the network's topology and the neurons' connectivity — The network's topology specification is optional (not shown in Figure 7.4), while the neuron's connectivity is compulsory. In this case, each neuron defines the neurons it is linked from, and the neurons it is linked to. Note that for fully connected networks, a special keyword **FullyConnected** is reserved to specify that neurons in a certain layer are fully connected to neurons in the previous and in the following layer;
- specification of each neuron's internal variables and constants that are required to implement the neural algorithm; and
- description of the neuron's computation, which is given by a control and data flow graph, defining the implementation of each neuron's rule.

In the *ICR* definition shown in Figure 7.4, constants and variables are first defined. The neurons' functionality is next defined by a CDFG including information on predecessors and successors.

The aim of the *ICR* is to serve as a bridge between the neural network high level languages (*HLLs*) and the hardware synthesis. The *HLL* is translated to the *ICR* through front-end compilation techniques, as described before with the *nC* language. The hardware synthesis tool reads the *ICR* format for a particular neural network and applies the required algorithms to create a hardware structure. Therefore, this intermediate format separates completely the front-end processing from the back-end, the hardware compiler. As a consequence, the *NSC* approach is not committed to any particular neural programming environment nor language. Its use can be widened to several neural network

systems, as long as the front end processing is able to generate correctly the *ICR* specification.

```

NeuralChipDefinition
  NeuronTypeDescr
  NeuralChipDefinition NeuronTypeDescr

NeuronTypeDescr
  NetTopologyOpt Connectivity DataPart CodePartOpt

DataPart
  NeuronNo NeuronDataOpt ConnectivityOpt

CodePartOpt
  RuleName TimeOpt CodeInstructions
  CodePart RuleName TimeOpt CodeInstructions

TimeOpt
  TIME INT_VALUE

CodeInstructions
  Node Operation NodeInputs NodeOutputs Predecessors Successors
  CodeInstructions Node Operation NodeInputs NodeOutputs Predecessors Successors

NeuronNo
  NEURON Name INSTANCES INT_VALUE

NeuronDataOpt
  DataDefinitions
  NeuronData DataDefinitions

DataDefinitions
  CONST Name INT_VALUE LEFT_BRK INT_VALUE DOTDOT INT_VALUE RIGHT_BRK
  REG Name IntValueOpt LEFT_BRK INT_VALUE DOTDOT INT_VALUE RIGHT_BRK
  COUNTER Name INT_VALUE LEFT_BRK INT_VALUE DOTDOT INT_VALUE RIGHT_BRK
  RAM Name INT_VALUE LEFT_BRK INT_VALUE DOTDOT INT_VALUE RIGHT_BRK ContentsOpt
  ROM Name INT_VALUE LEFT_BRK INT_VALUE DOTDOT INT_VALUE RIGHT_BRK ContentsOpt

IntValueOpt
  INT_VALUE

ContentsOpt
  LEFT_BRK ListOfIntValues RIGHT_BRK

Connectivity
  FULLYCONNECTED
  ListOfInputs ListOfOutputs

ListOfInputs
  INPUTS ListOfNeurons

ListOfOutputs
  OUTPUT ListOfNeurons

ListOfNeurons
  EXTERNAL INT_VALUE
  ListOfNumbers

ListOfNumbers
  LEFT_BRK ListOfIntValues RIGHT_BRK
  LEFT_BRK INT_VALUE TO INT_VALUE RIGHT_BRK

ListOfIntValues
  INT_VALUE
  ListOfIntValues INT_VALUE

RuleName
  RULE Name

Node
  NODE INT_VALUE

Operation
  OPERATION Operator

Operator
  ASSIGN
  INCR
  DECR
  ADD
  LT
  SUBTRACT
  MULTIPLY
  ISZERO

NodeInputs
  INPUTS MixData
  INPUTS MixData MixData

MixData
  INT_VALUE
  Name
  Name LEFT_BRK Name RIGHT_BRK

NodeOutputs
  OUTPUT MixData

Predecessors
  PREDECESSORS ListOfIntValues

Successors
  SUCCESSORS ListOfIntValues

Name
  IDENT

```

Figure 7.4 — The *ICR* Syntax Definition

7.4. *ICR* Transformations

Before the actual synthesis of the hardware is undertaken, further transformations simplify even more the behaviour of the input description. The two basic optimisations

employed are: graph partitioning and elimination of storage elements after constant propagation is accomplished.

7.4.1. Graph Partitioning

The original *ICR* graph, generated from the *nC* syntax tree, is partitioned into several sub-graphs according to the control information specified. The partitioning of a graph into several simple sub-graphs permits an easier manipulation of the graph, by explicitly describing its data and control flow. However, extra nodes are created to hold temporary variables. Nevertheless, the creation of the extra nodes does not pose a problem, because they can later be merged with others during the hardware synthesis. For example, a temporary variable, being mapped onto a register may be merged with another register, resulting in a multi-port structure, which minimises busses' resources and maximises parallelism.

The partitioning is carried out through analyses of all nodes in the entire graph. Operation nodes are isolated representing a unary or binary operation. Whenever an operation node outputs its result to another operation node in the original graph, another node is created to hold the intermediate result. Figure 7.5 illustrates the partitioning process.

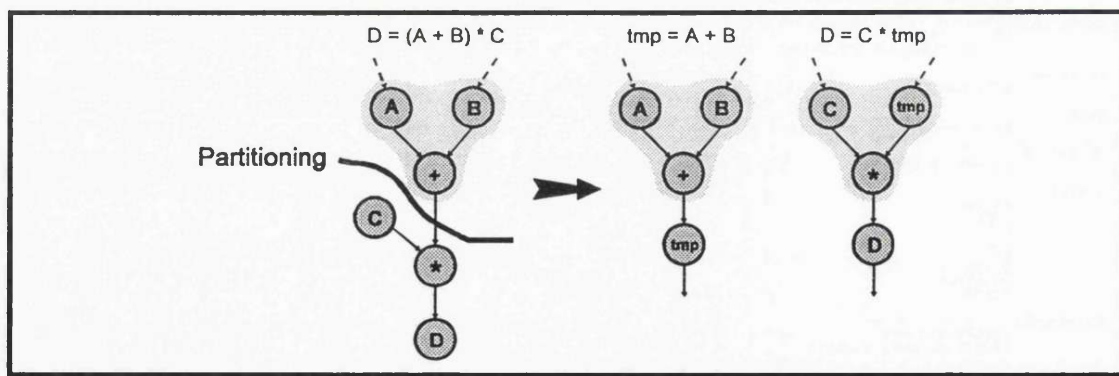


Figure 7.5 — Example of Graph Partitioning

7.4.2. Constant Propagation and Storage Elimination

Initially, all storage elements in the graph are visited to perform their lifetime analysis. On the basis of this analysis, an initial optimisation is accomplished, consisting of merging storage elements with disjoint lifetimes. The lifetime information is kept in memory as an *activity list*. This list is used throughout the synthesis process for every storage element in the design. This list is constructed for every variable and each element

in the list corresponds to the variables' activity during a specific control step. This activity is defined by five possible values :

Activity List = {*dead*, *idle*, *rd*, *wr*, *rdwr*, *wrrd*}

where each element represents the status of a storage element in each particular control state. The five possibilities are: **dead**, where no activity is performed in the variable; **idle**, in which the variable is idle but still alive, meaning that its data previously written is needed in a future state; and **rd**, **wr**, **rdwr**, or **wrrd**, meaning that the storage element is being read, written, read followed by a write operation, or written followed by a read operation in the same state, respectively.

Figure 7.6 gives a graphical interpretation for the registers' lifetime and its correspondence with the activity list. For example, it can be seen that the variable **_tmp_1** is **dead** during the control states 1, 2, 3, and 4; **_tmp_0** is **idle** in cycles 2, 3, and 4; **neu_state** is being read (**rd**) in control states 1 and 6; **_tmp_acc** has a **rdwr** operation in cycle 3; and finally, **_tmp_0** and **neu_error** are being written in states 1 and 6, respectively.

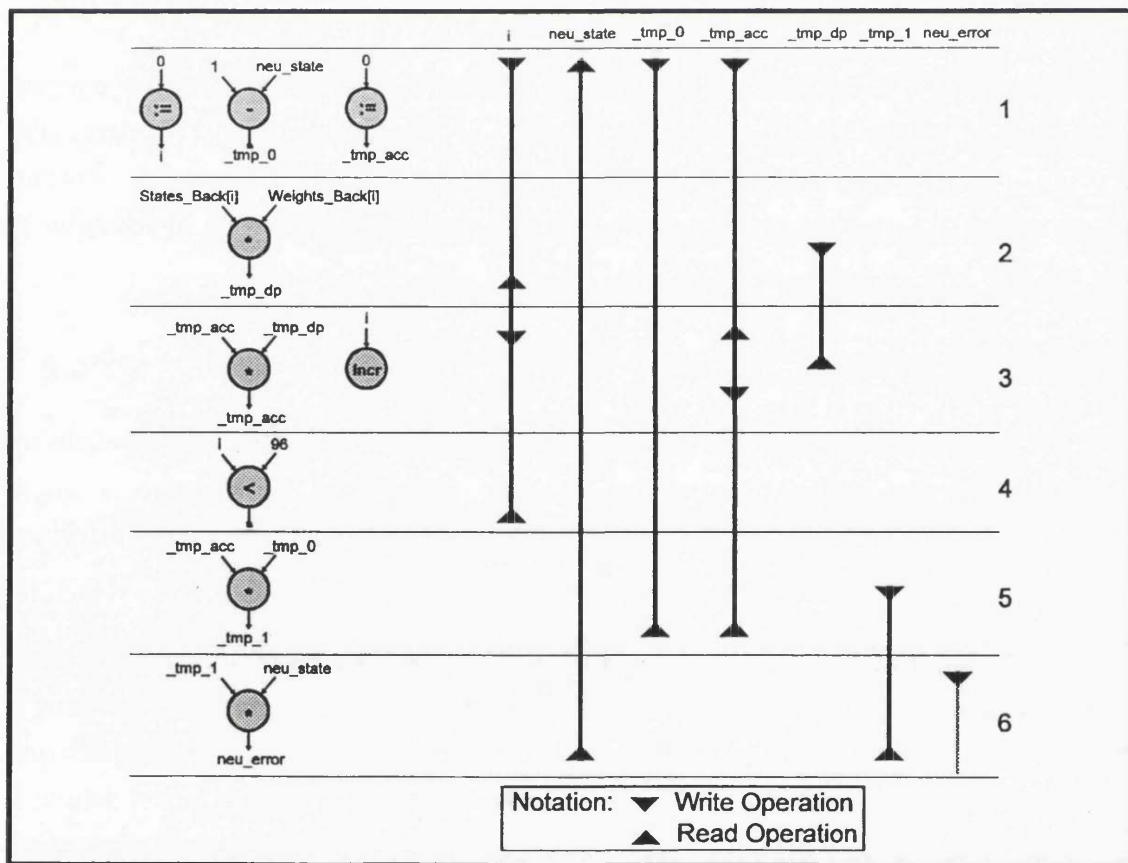


Figure 7.6 — Graphical View of Variables Lifetime

The resulted data flow graph is analysed, so that some useful optimisations can be performed. If a constant value is assigned to a storage element, then its lifetime is determined in the *ICR*. If this storage element is never assigned again, then the constant can be propagated to every instance of the storage element. As a consequence, the storage element can be eliminated.

The merging of storage elements of the same type can be achieved through the information provided by the activity list. In Figure 7.6, it is easy to see that registers `_tmp_dp` and `_tmp_1` can be merged. The algorithm for automatically merging variables is defined as follows:

```

For each variable Do
    ⌘ Read its entire activity list (every row in Figure 7.6)
    Loop
        ⌘ Get next variable
        ⌘ Read its entire activity list
        ⌘ Find any intersection between the two activity lists
        If intersection = Null Then
            ⌘ Merge
    End Loop
End For

```

Each time a variable is merged, its activation list is updated and the merged variable is eliminated. This allows the algorithm to carry on the process with the remaining variables from the same iteration where the merging has occurred. This algorithm turns out to be very similar to the well-known bubble sort algorithm for sorting elements in a list [164].

7.5. Target Architecture

One of the most important goals of this dissertation is to automatically generate an **efficient** hardware structure for the neural chips. To achieve this objective, a target architecture is adopted to restrict the design space and guarantee good results by providing guidelines to the hardware synthesis steps.

The adopted target architecture is based upon the *Generic Neuron* model [190], explained in section 6.3. While this model has proved to be flexible enough for realising a large class of neural models, its VLSI architecture has shown to be very efficient in terms of speed and silicon area. The VLSI architecture of the *Generic Neuron* model defines a rigid structure for the data path and control module of the three basic internal units: memory, execution, and communication unit. The automatic generation of these structures is the ultimate goal of the *NSC*.

According to the experience acquired during the development of the VHDL Back Propagation neural chip (see chapter 6), a few assumptions are made regarding the target architecture and the synthesis algorithms. These assumptions concern basically the interconnection mechanism, the clocking scheme, and the type of storage elements.

The interconnection structure of the data path is based upon multiple busses, which are allocated on demand according to the bus allocation algorithm explained in section 7.6.3. The use of multiple busses allows simultaneous data movements across the busses. In addition, *two-phase non-overlapping* clocking scheme is adopted, thus permitting the execution of operations in parallel, which saves several clock cycles and allows the design of a smaller controller. Finally, storage elements are multi-port structures, allowing them to be connected onto virtually any bus. These three characteristics exploit extremely well the type of computation usually employed in neural algorithms. The *Generic Neuron* architecture plays an important role towards the synthesis of the three PE's units.

The memory unit consists basically of RAM blocks, which are arranged by pairs (one block for the states, and other for weights). There may be one or two blocks of RAM, depending upon the necessity of having backward connections. For each block, one RAM is connected to one main bus (*Bus A*), while the other is connected to another main bus (*Bus B*). The addressing mechanism comprises the definition of a counter, to address the memories sequentially, and a set of register and comparator to determine the end of memory access. There is one such a set for each block of RAM. The number of RAM blocks is determined through synthesis, described in the next section.

The execution unit comprises a data path module and a control module. Both modules are fully synthesised by the NSC. This varies significantly from application to application. Depending upon the user's specification, a predominantly sequential data path is created, or a highly parallel one is generated. The synthesis of the control module consists in creating a FSM, in which sequence of operations varies from algorithm to algorithm.

Finally, the communication unit embodies dedicated structures, which provided the PE with abilities for controlling the other two units and for controlling the activities of the PE across the entire network. The input-output pins are pre-defined, having variations from one application to other concerning only with the length of busses. The data path depends upon the network's topology and interconnectivity (specified by the DataPart of the *ICR* — see Figure 7.4). Tri-state buffers, for handling data movements, and one register, for holding the PE's address, are always present. This last register must have its

lower significant part pre-set by the NSC, in order to uniquely possess a valid address. Each PE in the same chip must have a different address, as explained in section


		
Rule/Layer	Hidden Layer	Output Layer
Propagation Rule	$s_i = \sum_k s_k \cdot w_{ki}$	$s_j = \sum_i s_i \cdot w_{ij}$
Error Calculation	$e_i = s_i(1-s_i) \cdot \sum_j e_j \cdot w'_{ij}$	$e_j = (t_j - s_j) \cdot s_j(1-s_j)$
Weight Update (forward)	$w'_{ki} = w_{ki} + \eta \cdot s_k \cdot e_i$	$w'_{ij} = w_{ij} + \eta \cdot s_i \cdot e_j$
Weight Update (backward)	$w'_{ij} = w_{ij} + \eta \cdot s_i \cdot e_j$	\times

Table 7.1 — PE Functionality for Hidden and Output Layers

Table 7.2 shows the basic computation required by each PE in the hidden and output layers of a Back Propagation neural network. It can be noted that, during the learning phase, PEs in the hidden layer perform different computations from PEs in the output layer. The basic difference relies on: (i) the error calculation; and (ii) on the extra weight update (backward) calculation performed by the PEs in the hidden layer. The first is according to the Back Propagation algorithm [161]. However, the second is implementation-specific, and it is required for performance reasons. This is because the error calculation needs the weights from the connections between the PE in question and the PEs in the succeeding layer. Therefore, rather than broadcasting those weights, each PE in the hidden layer performs exactly the same weight update computation of the PEs in the next layer [190].

6.5.1. For fully connected multilayer networks, two extra registers are required: one to identify data coming from the previous layer; and other to identify data coming from the succeeding layer. For sparsely connected networks, however, and with very low degree of interconnectivity¹, more registers and comparators may be necessary to identify a valid data in the bus. The control module varies very little, depending basically upon the need for the learning phase or not, and upon the initialisation phase.

7.6. Data Path Synthesis

The synthesis of the hardware structure starts by reading the partitioned *ICR* graph and allocating storage elements, functional units, and busses to the three PE's units. The synthesis of the memory and execution unit is performed directly from the functionality of

¹Sparsely connected networks with high degree of interconnectivity may be considered as fully connected networks, in which some synapses have a null weight.

the *ICR*'s CDFG specification (CodePart in Figure 7.4), since they are directly related to the computation of the neural algorithms. However, the synthesis of the communication unit cannot be synthesised from the *ICR*'s CDFG. Instead, its synthesis is driven by the *ICR*'s DataPart, since this unit deals with the network's connectivity, and from the resulted memory and execution units synthesised before, since the communication unit commands operations on the other two units.

Two main busses are initially provided, namely *Bus A* and *Bus B*. The operations in the graph are then read and all three hardware structures are allocated in the same iteration. In addition, control data is stored in memory to feed the control synthesis with enough information of the sequencing and signal activations necessary to create the FSM.

During data path synthesis, the entire graph is visited to perform allocation and scheduling. Since the graph implicitly contains a schedule which is derived from the initial description in *nC*, a straightforward sequential scheduling is initially accomplished. The data path synthesis algorithm is basically as presented below:

```

For every operation type node in the graph Do
    ✎ allocate functional unit (if necessary)
    ✎ allocate storage elements
    ✎ allocate extra busses (if necessary);
    ✎ schedule the operation to the current control state
    ✎ Increment control state
End For

```

It must be noted that at this stage the number of storage elements has been already optimised by the previous transformation's steps. All temporary variables have been eliminated, except those that are essential to the correct computation. The number of functional units is minimum, that is, only one unit for each different operator. Although this sequential approach does not exploit parallelism as other scheduling algorithms, such as ASAP, it generates a highly optimised data path in terms of silicon area.

Furthermore, the straightforward sequential scheduling can be further improved by providing two modes of operation: *default mode* and *user-driven mode*. The *default-mode* is employed when nothing has been specified by the user concerning time and area constraints, or when the constraints given are exclusively related to the area. The *user-driven* mode is performed to meet a timing constraint specified by the user that could not be met by the *default mode*.

In greater detail, the general algorithm for data path synthesis includes: allocation of storage elements, functional units, and busses; and scheduling of operations to control

steps. The allocation and scheduling algorithms are performed simultaneously, as shown below:

```

State := 0
For each operation in the graph Do
    If operation = Assignment Then
        If input = Constant and output = Counter Then
            ⌘ Allocate a Counter;
            ⌘ Create Reset Logic;
            ⌘ Decrement (State);
        Else If input = Constant and output = register Then
            ⌘ Create input port;
            ⌘ Connect lines to GND and VCC;
            ⌘ Decrement (State);
        End If
    Else If operation = Binary operation Then
        ⌘ Allocate storage for inputs;
        ⌘ Allocate storage for output;
        ⌘ Allocate functional unit;
        ⌘ Create input port (output), and
        ⌘ Connect port into functional unit's output;
        ⌘ Create output ports (inputs);
        ⌘ Connect inputs into the two main busses;
    End If
    Increment (State);
End For

```

The allocation of hardware units is described in the following sub-sections. The algorithm above also considers the allocation of consecutive operation nodes in a single control state — denoted by the ‘Decrement (State)’ operation, meaning that the next operation will be performed in the same cycle. Thus, the algorithm is able to perform two or more assignments in the same control state, since two main busses are initially allocated, as described in sub-section 7.6.3. If more busses are created after the preliminary synthesis, then a mechanism to merge operations in consecutive control states is employed.

7.6.1. Allocation of Storage Elements

In the graph, a storage type node is allocated as a storage element in hardware, which can be either a register, a counter, a RAM, or a ROM. Each of the storage elements carries important information during the synthesis of the data path. The same information is also used to generate the controller.

An important feature of the storage elements in the NSC is the multi-port characteristic, which can be employed for registers, RAM, ROM, and counters in the NSC. The multi-port structure presents the advantage of reducing the number of cycles. This is possible because storage are made multi-purpose, thus eliminating the need for loading

operands into dedicated registers. By making storage elements connected to as many busses as necessary, virtually every element can be the input or output of any operation. In addition, a storage element is allowed to be read and written simultaneously in the same cycle. This is usually required for operations that have a particular storage element as input and output at the same time, which is resulted from operations like $a := a + b$. Although more busses are generally used, this approach tends to require a smaller number of storage elements.

A *register type* holds the following fields:

- number of input ports;
- which bus each input port is connected to;
- number of output ports;
- which bus each output port is connected to;
- a boolean value indicating whether or not tri-state logic is required; and
- number of bits for the register.

The number of ports (both input and output) and the bus connection information indicate that a multi-port register is required. Consequently, the number of control signals (necessary to access data to and from the busses) is determined. For each port, a boolean value indicates the need for a tri-state logic. If the port is connected to a bus, then tri-state logic is required. If the port is connected directly to a single hardware component (storage element or functional unit), then tri-state logic is not required, and the control signal for this port is eliminated.

A *memory type* (RAM or ROM), requires the following fields:

- number of input ports;
- which bus each input port is connected to;
- number of output ports;
- which bus each output port is connected to;
- memory size;
- data width;
- data bus; and
- address bus.

The first four fields are required for the same reason described above for registers. The memory size determines the number of lines for addressing the memory data. Data bus and address busses are explicitly specified. The control signals for ROMs include the *chip*

select and *read* signals, which must be provided by the controller in a correct way. RAMs include the same control signals, except that *read/write* control logic is required.

For a *counter type*, the fundamental fields include:

- number of input ports;
- which bus each input port is connected to;
- number of output ports;
- which bus each output port is connected to;
- a boolean value that indicates the need for a *load pin*;
- a boolean value that indicates the need for a *reset pin*;
- input bus; and
- output bus.

The load pin is required only if there is an assignment to a *counter type* node. However if the assignment refers to a constant 0, then a *reset pin* is used instead, indicating that the counter must have logic for resetting every internal flip-flop. If the *load pin* is present, then the specification of an input bus is provided. The need for an output bus is also optional, and is only specified if the intermediate results from the counter are required. This is often the case when a counter is used as an index for memory references. Conversely, a counter used in a single loop does not require intermediate values, and therefore output bus is not needed. Instead, an *output pin* is provided, which indicates end of counting.

A constant type defines the following necessary fields:

- output bus;
- number of bits for the constant.

A constant may be bound to a bus or directly to a specific storage or functional element. In the first case tri-state logic is needed. In both cases, the bit range specification must be provided to avoid aliasing problems.

7.6.2. Allocation of Operators

The NSC assumes a small set of functional units in its library. This set is chosen based upon the VHDL implementation described in the previous chapter. Operators in this library include VHDL templates for ALUs, comparators and shifters. These templates are used by the module generator (see section 7.7), which actually builds the necessary functional units.

Certain operations require the allocation of functional units, such as the operator '+'. However, operations like ':=', an assignment, do not require any allocation of operators. Instead it requires the allocation of busses. Yet, operations like '<', a comparison, may or may not require allocation. In its simplest form, a comparison operation is implemented by the control module. For example, an operation 'c < 96', where 'c' is a counter not connected to an output bus, does not require any functional unit nor bus allocation. The control module tests directly the counter's output pin for a *true* boolean value. Conversely, if the counter has an output bus, a comparator may be implemented instead, which involves the allocation of a comparator unit, or an ALU.

Arithmetic and logic operations are all performed by a single ALU, which is also capable of performing multiplication (in conjunction with a shifter and an accumulator). This assumption is based upon the premise that a simple add function implemented by a general-purpose ALU is more cost-effective, for VLSI area, than the implementation of random logic [115]. Nevertheless, other types of ALU can be included in the library to support, for example parallel multiplication, which could be used for critical real-time applications.

7.6.3. Allocation of Interconnections

As mentioned before, interconnection between the elements in the data path is accomplished by busses. A *bus type* node comprises the following fields:

- storage elements or functional units that write data onto the bus; and
- storage elements or functional units that read data from the bus.

The allocation of busses follows the approach usually taken by hardware designers. For the *Generic Neuron* architecture, two main busses are initially allocated.

During the functional unit allocation, which takes two operands and generates an output, such as an *add* operation, a local bus is created for one of its inputs. In this case, one operand loads the input data onto one of the main busses, the other loads the input data onto the operator's local bus, and the result is loaded in the second main bus by the functional unit. Thus, during data path synthesis, the following heuristic is used:

If operation has a storage element as one of its inputs and output *Then*

- ☛ Link input onto the local bus created for operator and the output onto one of the two main busses; Link the other input to the other main bus;
- ☛ Operator reads inputs from one main bus and its local bus and writes the result onto the other main bus;

End If

The heuristic above indicates that the storage element in question is read and written in the same cycle, and therefore it must be transformed into a multi-port structure. Note that during the allocation of the functional unit, a new bus is created, which is local to one of its inputs. Therefore, the output port of the storage element is linked to the local bus, while the input port is linked to one of the main busses. This rule must be accomplished to respect one of the basic rules in a synchronous digital system: only one source can load a data value onto the bus at a specific time.

After the allocation of all three hardware units (storage, operation, and busses) is finished, more busses can be created to meet some timing constraints. This approach differs considerably from the one adopted in the Cathedral-II system [152], in which busses are allocated on demand for each operation, which results in a structure with several redundant busses. A final phase of bus merging is thus accomplished in the Cathedral-II to eliminate the redundant ones. In the NSC, two main busses are the minimum configuration. This focuses on the VLSI area aspect of the design. However, if necessary, extra busses can be created, either to allocate functional units or to use fewer control cycles.

The main difference between the Cathedral-II and the NSC is that the former creates as many busses as required, thus exploring parallelism, and then applies a bus merging algorithm to eliminate the redundant busses. Conversely, the NSC works in a way similar to the one used by hardware designers. A minimum number of busses is initially used, and during data path synthesis more busses are allocated as required. A bus merging algorithm is not required because the optimal number is obtained by construction, which trades VLSI area against parallelism. This approach guarantees better use of VLSI area, and yet employing some degree of parallelism, and therefore allows more PEs in a single chip. Parallelism is already inherent in the neural network, in which PEs for each layer function in parallel. Better parallelism can be obtained through the ASAP scheduling algorithm. This may be required to meet some timing constraints not met before (see next sub-section), then full parallelism is explored, but the silicon area is sacrificed.

7.6.4. Scheduling

As described before, a straightforward sequential scheduling is initially performed. Then this scheduling is improved to provide some degree of parallelism, which is accomplished by merging operations with no bus activity into the same state. In addition, operations can also be merged if any idle bus is detected during a particular state.

Scheduling involves reading the graph and assigning each operation to one control state. Note that this algorithm is developed having in mind a FSM that implements the sequence of operations. Therefore, control states are the basic unit, instead of control steps. Since each control state can be performed in one or more control steps, the timing information is still available during the synthesis. For example, a multiplication employing the Booth's algorithm is executed in a number of control steps.

Some operations take *zero* number of control steps to be executed, or can be done in parallel with any other operation. This is the case of resetting a counter and loading a constant value onto a register. These operations can be performed in parallel because of the multi-port structure of the storage elements. During scheduling these possibilities are explored, which minimises considerably the FSM, thus resulting in a faster implementation of the neural computation by the PEs.

The algorithm operates as presented in section 7.6. It provides the minimum possible implementation with respect to hardware resources. Duplication of functional units is not allowed, which guarantees very compact designs, but slower FSMs, since more cycles are required to perform the computation. This approach focuses on the maximum possible number of PEs per chip. However, if the users specifies timing constraints in nC , which is not met by sequential scheduling, then an ASAP algorithm is employed.

The ASAP scheduling algorithm in the NSC starts from the sequential schedule done before, and modifies the graph in such a way that a certain function is performed in the fastest possible way allowed by the algorithm. The ASAP is performed as presented below:

```

For each control state Do
    ☞ Get Op1 = operation in control state;
    ☞ Increment (state);
    ☞ Get Op2 = operation in control state;
    If output (Op1) = inputs (Op2) Then
        ☞ next control state;
        ☞ Continue;
    Else If output (Op1) = output (Op2) Then
        ☞ next control state;
        ☞ Continue;
    Else
        ☞ Merge operations in the same control step;
    End If
End For

```

The above algorithm implements the ASAP strategy by reading the entire *ICR* graph from top to bottom. If the successive operations (Op1 and Op2) do not violate any of the rules above, which are: (i) writing to same storage element; and (ii) Op2 does not need the output of Op1 to compute, then these two operations can be merged into a single state, and therefore can execute in parallel. However, after the ASAP scheduling is performed, the allocation of new busses is necessary. Therefore, the allocation procedures are executed once more to provide the required resources to explore the parallelism enforced by the ASAP strategy.

7.7. Module Generation

After the data path has been synthesised, every necessary module is explicitly specified in the graph as a generic, parameterised cell structure. For example, a register can have one or more ports, a counter can have a load pin or not, etc. Thus, this information is used by the module generator to create a hardware structure for each of the required cells.

This task is dependent upon the technology and style used for the final chip design. A library of VHDL modules is thus created for each cell in the graph. The automatic generation is based on standard templates for each module, which are parameterised regarding several issues. Figure 7.7 shows the structure of a generic register, which is created and configured automatically for each allocated register, after the number of input and output ports have been calculated during data path synthesis.

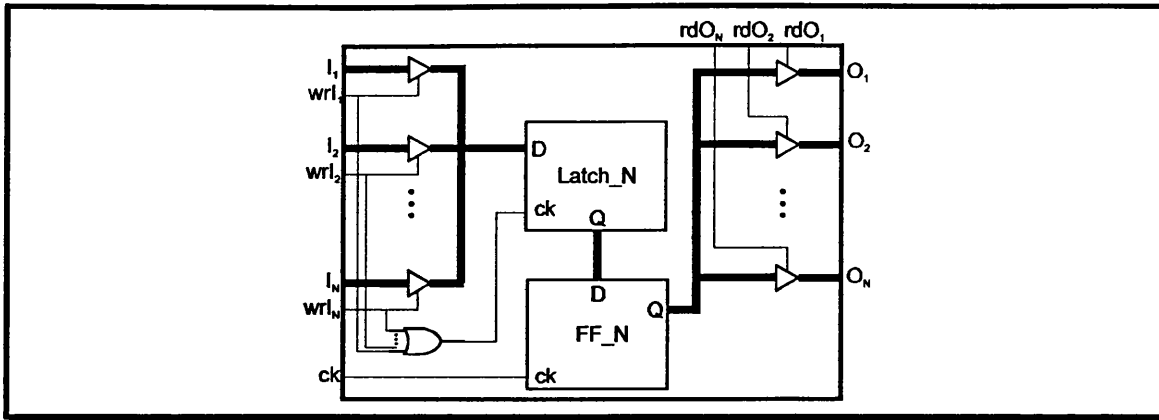


Figure 7.7 — A Generic Multi-Port Template for Registers

The procedure that generates multi-port registers requires six arguments that specify: (i) the number of input and (ii) output ports; (iii) the need or not for the reset function (which writes zero in every flip-flop of the register); (iv) the need for tri-state logic on each output port; (v) the number of data lines; and finally (vi) the name of the module.

Similarly, a procedure to generate RAM and ROM modules requires five arguments: (i) the number of input and (ii) output ports; (iii) the number of address lines; (iv) the number of data lines; and (v) the name of the module. If more than one port is required for a particular memory block, then the same structure for generating multiple ports in a register is employed here. Finally, a counter module (see Figure 7.8) is generated by taking five parameters that describe: (i) the need for a reset logic; (ii) the need for a preset logic; (iii) the number of input and (iv) output data lines; and (v) the name of the module.

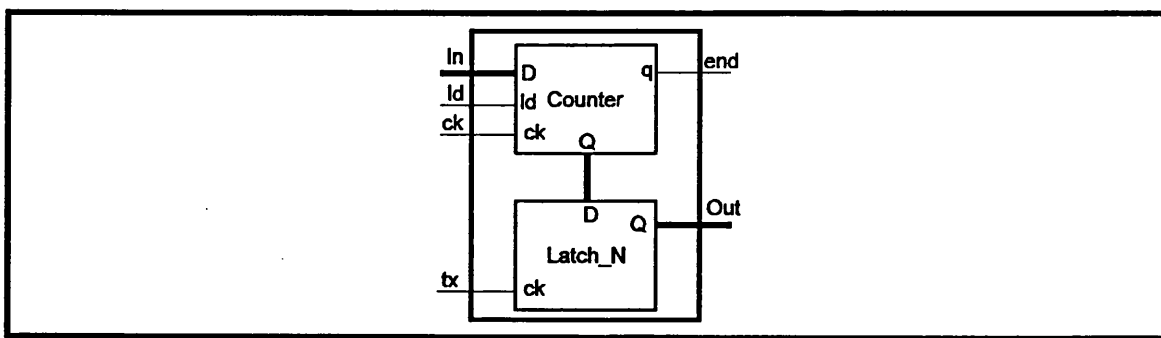


Figure 7.8 — The Counter Module

An ALU module has a rigid structure and besides the number of data lines for its inputs and output, and the name of the module, the only information required concerns the need or not of a multiply function. If a multiply operation is specified, then a shifter and a register are added to the ALU module to perform the Booth's algorithm for two's

complement numbers [55]. Figure 7.9 shows the ALU structure, including the multiplication operation. Note that the multiplier register (MR) is generated from the same template used for registers, but added with two output lines (lsb and lsb₋₁), which are required by the Booth's algorithm. If the multiply operation is not required, the unit is restricted to the add-subtract module.

The ALU performs an add or subtract operation by taking its inputs from *Bus A* and *Bus B*, and producing an output onto *Bus C*. The multiplication is a special operation, in which one input is the register MR (which must be loaded *a priori*), the other input comes from any storage element connected to *Bus A*, and the result goes to any register connected to *Bus B*, which must be set to zero before the operation begins.

In summary, a module is constructed from the graph specification after data path synthesis has been performed. Since possible merging of storage elements and functional units is done during synthesis, one module per element is generated. For example, if two memories are defined, thus two memory modules, with different names, are generated. Each generated module is saved as a VHDL file, which is named with the same name used for the module.

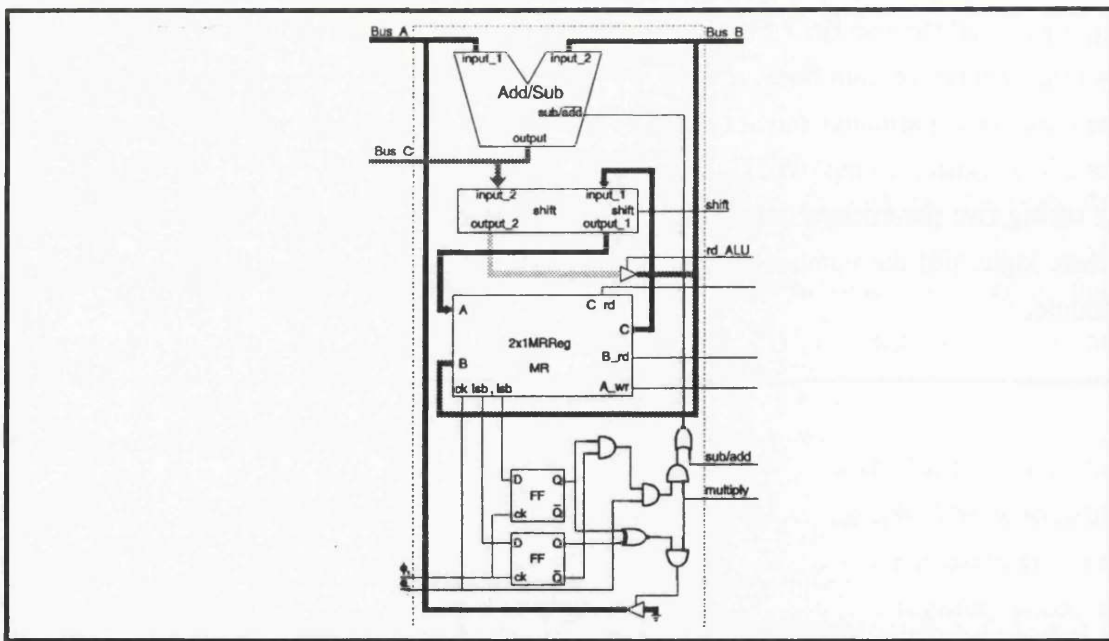


Figure 7.9 — ALU Module for Implementing Typical Neural Network Computation

7.8. Control Synthesis

As described before, during scheduling a list containing all signals that should be activated are explicitly kept updated during the data path synthesis process. This list

comprises every control state generated by the scheduling algorithm, each containing explicit information regarding control signals.

Therefore, the control synthesis task starts analysing this list and then builds a FSM that controls the allocated modules. This is already performed in conformity with the VHDL syntax.

An important issue during the synthesis of the *Generic Neuron's* PEs is the synchronisation among the three internal units. As explained in the previous chapter, the memory unit is accessed by both the communication and execution unit. To achieve full parallelism, the communication unit uses one phase of the two-phase clock cycle (ϕ_1) while the execution unit uses the other phase (ϕ_2). The memory and execution unit need to have some internal parameters initialised, such as the memory blocks and the algorithm-dependent variables, like the *learning rate* for the Back Propagation network. This is controlled by the *initialisation phase* of the communication unit's FSM, since this unit is responsible for the interface with other PEs as well as with the central controller.

The parameters that should be initialised are not explicitly defined in the *ICR*, nor in *nC*. Nevertheless, this information is **implicitly** specified in the graph and can be easily obtained by searching the activity list of all variables. The following proposition defines what variables need to be externally (to its own unit) initialised, and when this should happen:

```

For each variable Do
    If variable's activity list = dead state Then
        ⌞ Do nothing;
        ⌞ continue;
    End If
    If variable's activity list = rd state Then
        ⌞ Inform control synthesis the need for initialisation;
    Else
        ⌞ Variable does not need initialisation
    End If
End For

```

The above algorithm employs a search in the activity list for every variable. If the first element of a variable begins with a *read* operation, then this variable must receive a value. The choice of when the scheduling takes place must not exceed the control state by which the *read* operation occurs. This proposition can be seen graphically by looking for the symbol '▲' as the first occurrence for any variable. For instance, in Figure 7.6, it is shown that *neu_state* needs to be initialised externally.

Similarly, PEs must send some values to other PEs, and again this is not explicitly specified in the graph. A similar proposition, however, can be made, which is shown below:

```

For each variable Do
    If last element in the variable's activity list = wr state Then
        Inform control synthesis the need for sending out;
    End If
End For

```

This can also be interpreted graphically (see Figure 7.6) by looking for the last cycle in each neural rule represented in the graph. If the last symbol for a particular variable is '▼' before it crosses the boundary of the graph, then a value has been written and not read yet, which means that this data must be sent out to the other PEs in the network. In Figure 7.6, neu_error register is written in cycle 6 and crosses the boundary without a further read operation.

The above propositions are very general and they reflect faithfully the way a *nC* neural network program works. It must be noted that if the user writes a function that defines an internal variable and then reads its contents in an expression evaluation, without initialising it before, then the *NSC* will tend to incorrectly initialise this variable externally. However, to read a value that has never been written before is a serious programming error. Conversely, if the user defines a variable and initialises it, but never uses it throughout the program, then the *NSC* would tend to incorrectly send this data outside the PE. In fact, this situation can never present itself because during the *nC* compilation, redundant variables are automatically eliminated. The only variables that survive are the ones originated from the generic and extended parameter lists. Since these lists are initialised *a priori* by the main part of the *nC* program, the two propositions are by construction correct.

The only exception is when these circumstances occur inside a loop or a conditional loop path of the graph. The algorithms are developed to cope with these conditions by keeping updated the data flow and, in particular, the control flow graph.

The task of the control synthesis is therefore to generate a FSM, written in VHDL, according to the scheduling strategy described before. For every control state, the control synthesis collects the list of actions to be realised. The information regarding the *next state* is obtained through the *ICR* graph, in particular from the part that describes the control flow. Figure 7.10 shows the template for a finite state machine, in VHDL, which basically consists of a *case* clause with several possible states which define the list of actions to be

performed, during a particular phase of the clock (ϕ_2). The test for jumping to the next state, according to the logic, is performed in the other phase of the clock (ϕ_1).

```

ARCHITECTURE behaviour of FiniteStateMachine IS
    SIGNAL current_state =: CHARACTER := 'A';
    SIGNAL next_state =: CHARACTER := 'A';
BEGIN
    Synch: PROCESS
    BEGIN
        WAIT UNTIL phi2'event AND phi2 = '1';
        current_state <= next_state;
    END PROCESS Synch;

    FiniteStateMachine: PROCESS
    VARIABLE ...
    PROCEDURE ...
    BEGIN
        END ...;
    BEGIN
        WAIT UNTIL phi1'event AND phi1 = '1';
        CASE current_state IS
            WHEN 'A' =>
                IF (cond = '1') THEN
                    -- List of actions
                    next_state <= 'C';
                ELSIF ( .. ) THEN
                    -- List of actions
                    next_state <= 'D';
                ...
            ELSE
                -- List of actions
                next_state <= 'A';
            END IF;
            WAIT UNTIL phi2 = '1';
            --List of Actions;
        WHEN 'B' =>
            ...
        END CASE;
    END PROCESS FiniteStateMachine;
END behaviour;

```

Figure 7.10 — A Finite State Machine Template in VHDL

7.9. VHDL

The generation of VHDL code for the synthesised circuit is a straightforward operation, which uses the designs developed in Chapter 6 as templates for most of the modules used by the NSC. The data path is fully composed of VHDL structural *components* and behavioural descriptions. Examples of structural descriptions include registers, gates, busses, and ALU. Behavioural descriptions include flip-flops, memories, and FSMs (see Figure 7.10).

Therefore, the generated VHDL code is described in a mixed mode. For some parts of the design a structural representation is generated; for others a behavioural one is produced. Those that are in the structural level do not require any further synthesis step. Instead, a technology mapping can be directly employed. These include primitive components such as logic gates, flip-flops, and some macro cells such as the ALU, which is composed of those primitive structural cells. For behavioural level descriptions, a synthesis step may or may not be required. For example, the description of the RAM and ROM does not require synthesis. However the description of the FSM needs to be fed into

a FSM compiler. Such a tool can be easily found in several commercial and educational tools.

It is important to note that this approach tries to meet a compromise on what is available and what is required. The data path synthesis performed by the *NSC* is crucial for the compactness of the circuits. Conversely, the control synthesis, which consists in generating a controller for a given scheduling strategy, has been done by several other projects. Therefore, it is assumed that the VHDL description of the controller can be fed into one of these available tools and integrated into the *NSC* easily. If these tools do not support VHDL, then it is necessary to design a translation between VHDL and the tool's native input description.

The produced VHDL code in the *NSC* is guaranteed to work as shown through simulations (using the Cadence software). It is not however guaranteed that the *NSC*'s VHDL output can be further fully synthesised with no modifications by lower level synthesis tools. This final test has not been done due to lack of appropriate tools in the Department. Nevertheless, the design philosophy undertaken throughout this work conforms with the current state of CAD tools, which implies in a smooth path towards the integration of the *NSC* and these low level tools. Finally, since the Cadence software does not fully implement the VHDL language, some modifications may still be necessary to make the output description compatible with the VHDL low level synthesis tool.

7.10. Neural Network Partition

The synthesis process developed in the *NSC* has concentrated on the design of the processing elements. To implement a complete neural chip, these PEs must be connected according to the bus strategy presented in chapter 6.

The problem of partitioning the neural network into several neural chips is posed by the difficulty of integrating large neural networks into a single chip. A mapping strategy must be envisaged without compromising the generation of identical chips. As mentioned before, the Back Propagation network has different neurons between hidden and output layers.

The adopted strategy integrates hidden neurons and output neurons into the same chip according to the proportion of neurons in each layer. For the optical character recognition (OCR) example (see chapter 5), in which 24 neurons are present in the hidden layer, and 96 in the output layer, there are 4 output neurons for each hidden neuron. Therefore, according to the space occupied by PEs, the partitioning strategy would result

in 24 chips of 1 hidden neuron plus 4 output neurons, or 12 chips of 2 hidden neurons plus 8 output neurons, and so on.

Note that this partition can only be applied if a single bus is used to connect PEs. If a multi-bus strategy is adopted, then depending upon the partition strategy, more pins may be required to provide the correct connection of PEs onto the appropriate busses. This is because in a multi-layer network PEs compute differently according to the layer they belong to. In this case, a multi-bus connection means that PEs in a single chips must be connected to different busses. However, this does not mean that scalability is affected. If a particular neural chip has several PEs of only two layer, then two busses are provided, regardless the number of PEs in the chip.

The final VHDL chip description comprises instantiations of several PEs. Each PE is, in turn, composed of instantiations of the three basic units: communication, memory, and execution (See appendix C for details).

7.11. Summary

In this chapter, a thorough description of the development of the *NSC* is given. The approach is based on the high level synthesis of the silicon compilation process. Much of the effort is concentrated on the high level transformations of the *nC* language, which ensures synthesis of optimised hardware structures.

Some important results obtained through the techniques developed during the design of the *NSC* include:

- development of several optimisations at different level of the *NSC*, which has produced a synthesis tool that creates very compact hardware structures; and
- development of the *activity list*, which serves as the basis for the data path synthesis; in addition, several interesting algorithms are developed upon this technique, including verification of the correctness for the synthesised hardware;
- design of two modes for hardware synthesis to create structures that meet the user's specification: *default mode* and *user-driven mode*;
- development of an automatic mechanism for discovering which variables need to be initialised;
- development of an automatic strategy to determine which variable should have its result sent to other PEs in the network;

- development of a module generation tool, which automatically creates a VHDL hardware module based upon parameterised templates.

Chapter 8

From *nC* to VHDL Neural Chips

This chapter presents the complete cycle of generating VHDL neural chips from a behavioural description of the neural network in nC. A Back Propagation neural network is used to show all steps toward the automatic synthesis of hardware, according to the algorithms developed in the previous chapter. In addition, the synthesised design is compared with the one manually developed in chapter 6.

8.1. Overview

The primary aim of this chapter is to show the entire process of compiling a *nC* program and synthesising neural chips that implement the functionality specified by the user. The design starts with a *nC* description of the Back Propagation neural network implementing the same OCR application used by the simulations developed in chapter 5. Then, all steps involved in the *NSC*'s high level synthesis are stressed. The final design of the VHDL neural chip is compared with the one manually developed in chapter 6.

8.2. *nC* Description of a Back Propagation Network

As described in chapter 4, a *nC* program involves the specification of two sections: application and algorithm.

8.2.1. Application Definition

The application definition of the OCR problem involves the specification of the **config** data structure, precisely as shown in Figure 8.1, and the specification of the function **main**.

The **config** structure specifies only the network's topology, which defines a network of three layers composed of one cluster in each layer; each cluster comprises 96, 24, and 96 neurons, respectively.

The application control, implemented by the **main** function must be written according to the chosen algorithm and following a sequence of function calls that control

the flow of the entire application. Typically, this is performed by a loop that calls the function **learn** for every input pattern followed by the function **recall**. Furthermore, **connect** and **build_rules** must be called in advance, so that the **system** data structure is correctly configured.

```

#define NETS          1
#define LAYERS        3
#define CLUSTERS      1

struct {
    int nets;
    struct { int layers;
        struct { int clusters;
            struct { int neurons;
                } cluster [CLUSTERS];
            } layer [LAYERS];
        } net [NETS];
    } config =
    { 1, { 3,
        { { 1, { 96 } },
          { 1, { 24 } },
          { 1, { 96 } },
        }
    }
};
/* One network of three layers */
/* Layer 1 - One cluster of 96 neurons */
/* Layer 2 - One cluster of 96 neurons */
/* Layer 3 - One cluster of 96 neurons */

```

Figure 8.1 — The **config** structure for the OCR Application

8.2.2. Algorithm Definition

The algorithm definition consists of initially deciding what functions are required and what parameters these functions need to manipulate. According to the Back Propagation algorithm (see Table 6.2), there are four basic functions to be defined: **State_Update** and **Weight_Update**, which are standard for neurons in the hidden and output layers, **Err_Cal_Hidden** for neurons in the hidden layer, and finally **Err_Cal_Output**, for neurons in the output layer. In addition, the connectivity of the network must be specified through the **connect** function, and the functionality of each element in the network must be defined through the function **build_rules**, which defines the exact contents of the important **RULE** data structure.

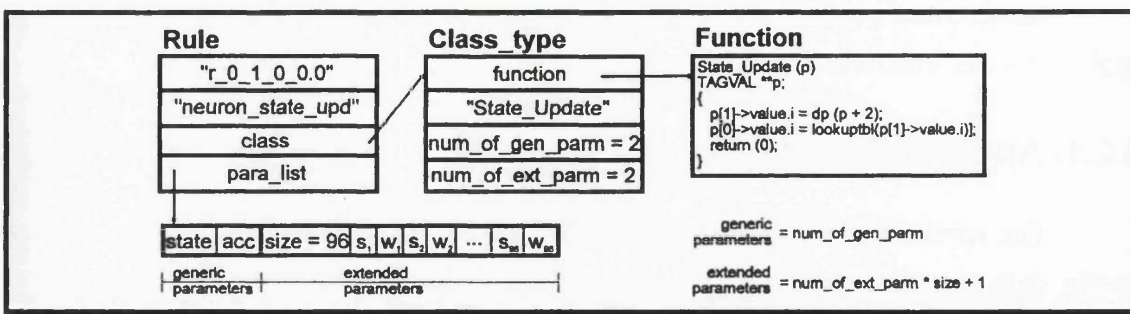
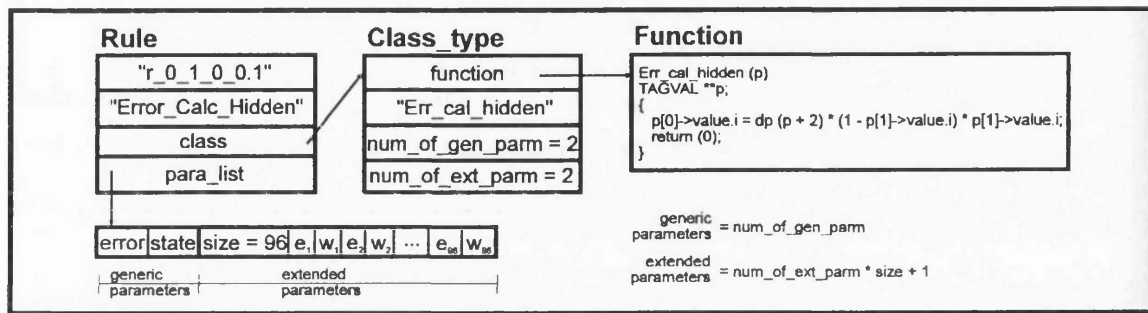


Figure 8.2 — **State_Update** Rule Definition

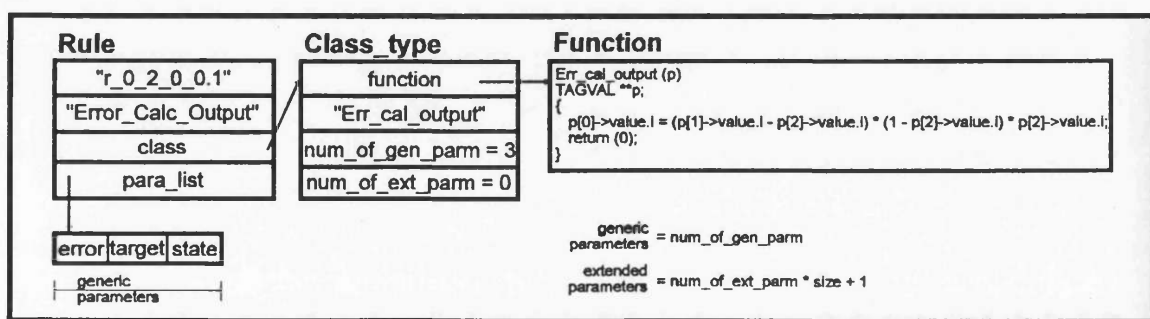
Figure 8.2 exhibits the complete **RULE** data structure for the **State_Update** function. This structure defines a *nC* rule named as "neuron_state_upd" (for the purposes of graphic monitoring) and identified as rule 0 of the neuron 0, in the network 0, layer 1 (hidden layer), cluster 0. This is represented by the tag field "r_0_1_0_0.0" (see

section 4.2.1). This rule corresponds to a class which is specified by the `State_Update` function implementing its computation based upon the list of parameters (`para_list`). This list comprises two parts: generic parameters and extended parameters. The generic parameters' part is composed of two values (`num_of_gen_parm = 2`): the neuron's state and its accumulator value (required to compute the sum of products performed by the built-in function `dp`). The extended parameters' part comprises a constant value (`size = 96`), followed by 96 pairs of values (since `num_of_ext_parm = 2`) composed of states and weights.

Figure 8.3 — `Err_cal_hidden` Rule Definition

The next rule to be programmed is `Err_cal_hidden`, which is responsible for computing the error values of the neurons in the hidden layer during the learning phase. Figure 8.3 presents the code for the function, as well as the complete configuration of the **RULE** data structure. The parameters manipulated by this function are the neuron's state and its associated error, plus the pair of backward connection weights and error states coming from the subsequent layer.

Figure 8.4 shows the equivalent error calculation function for the neurons in the output layer (`Err_cal_output` rule). In this case, there are no backward states and weights to manipulate. Instead, this rule computes the error based upon the difference between the target value and the neuron's output state. Consequently, the list of parameters comprises only the generic parameters.

Figure 8.4 — `Err_cal_output` Rule Definition

The last function required to perform the Back Propagation algorithm is the **Weight_Update** rule. Figure 8.5 shows its code and respective **RULE** structure for a neuron in the output layer. The same function applies to neurons in the hidden layer, differing only on the number of paired elements in the list of extended parameters.

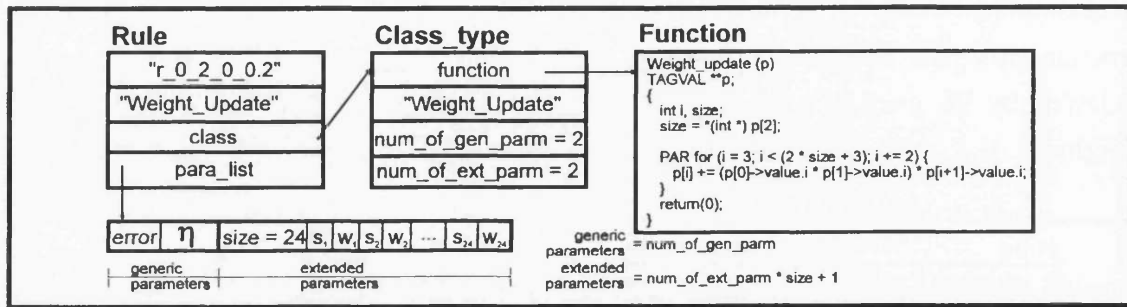


Figure 8.5 — **Weight_Update** (Forward) Rule Definition for the Output Layer

As explained before, the Back Propagation algorithm involves bi-directional connections among neurons between two consecutive layers. To avoid having to broadcast the weighted values of the backward connections, the user should supply an extra rule that duplicates the weight updating rule of a particular layer in its preceding layer. For example, neurons in the hidden layer should implement the same **Weight_update** rule realised in the output layer. Thus, backward connections in the hidden layer are identical to forward connections in the output layer. This redundant, but necessary step for performance reasons, is shown in Figure 8.6.

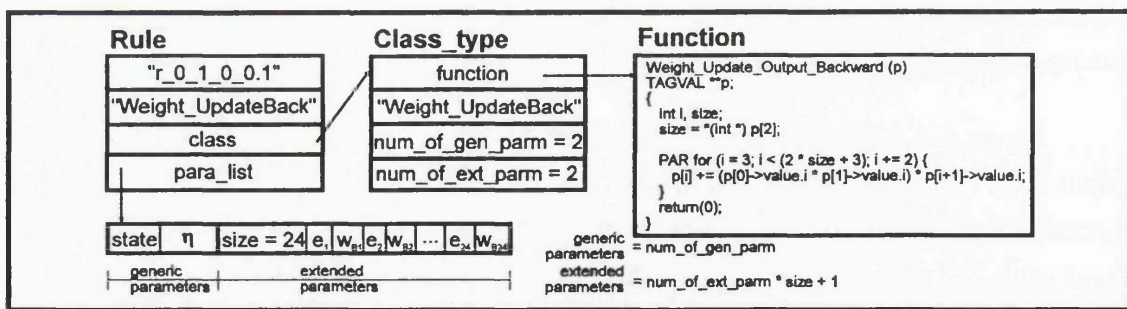


Figure 8.6 — **Weight_Update** (Backward) Rule Definition

Note that the tag field in the **RULE** data structure of the rule definitions above defines the order each function is executed. Thus, for neurons in the hidden layer, the execution order is **State_Update**, **Weight_Update_Backward**, **Err_Cal_Hidden**, and **Weight_Update**.

8.3. *nC* Compilation and Syntax Tree Transformations

As explained in the previous chapter, the **system** data structure is read either directly from the memory, at run time, or from the *nC_code* program, which is generated during simulation. In either case, the data structure provides all the information required for the high level synthesis.

Each time a certain rule is compiled, its **RULE** data structure is analysed to identify the elements of the generic and extended parameters. On the basis of these parameters, Table 8.1 shows all transformation steps realised upon the five rules specified for the Back Propagation network. These transformations correspond to the ones developed in sections 7.2.2 and 7.2.3. They also include the transformation of the `lookuptbl` function into an array, which is further transformed into a ROM structure, and the transformations performed in the loop statements (not shown in the Table), since the generic and extended parameters have been modified.

<i>Rule</i>	<i>nC Parameter</i>	<i>New Internal Name</i>	<i>Type</i>	<i>Hardware</i>
State_Update (Hidden and Output Layer)	p[0]->value.i	neu_state	Register	Register
	p[1]->value.i	neu_acc	Register	Register
	p[2]->value.i	size_State_Update	Constant	Signal
	p[i]->value.i	States_Forward[i]	Variable	RAM
	p[i+1]->value.i	Weights_Forward[i]	Variable	RAM
	lookuptbl (...)	lookup_tbl [neu_acc]	Variable	ROM
Err_cal_hidden (Hidden Layer)	p[0]->value.i	neu_error	Register	Register
	p[1]->value.i	neu_state	Register	Register
	p[2]->value.i	size_Err_cal_hidden	Constant	Signal
	p[i]->value.i	States_Backward[i]	Variable	RAM
	p[i+1]->value.i	Weights_Backward[i]	Variable	RAM
Err_cal_output (Output Layer)	p[0]->value.i	neu_error	Register	Register
	p[1]->value.i	neu_target	Register	Register
	p[2]->value.i	neu_state	Register	Register
Weight_Update Forward (Hidden and Output Layer)	p[0]->value.i	neu_error	Register	Register
	p[1]->value.i	learn_rate_of_net_0	Register	Register
	p[2]->value.i	size_Weight_Update	Constant	Signal
	p[i]->value.i	States_Forward[i]	Variable	RAM
	p[i+1]->value.i	Weights_Forward[i]	Variable	RAM
Weight_Update Backward (Hidden Layer)	p[0]->value.i	neu_state	Register	Register
	p[1]->value.i	learn_rate_of_net_0	Register	Register
	p[2]->value.i	size_Weight_UpdateBack	Constant	Signal
	p[i]->value.i	States_Backward[i]	Variable	RAM
	p[i+1]->value.i	Weights_Backward[i]	Variable	RAM

Table 8.1 — Transforming Generic and Extended Parameters

Also shown in Table 8.1 is the mapping of internal variables and constants into hardware. The *new* names created for the generic and extended parameters are used in the data definition part of the *ICR* format.

In-line expansion is performed during the conversion of the syntax tree into graph. This is done by visiting every tree node and replacing every call node by its correspondent code. The function **dp(p)** is expanded in **State_Update** and **Err_cal_hidden** functions. As described before, **dp(p)** is a built-in function that operates on the extended parameter list. Again, since this list has been modified, the code of the function must be adjusted accordingly.

8.4. Hardware Synthesis

8.4.1. Graph Generation and *ICR* Transformations

The next step is the generation of the *ICR* for the application in question. The *ICR* format consists of a data section, which represents the storage elements required to implement the algorithm, and a code section, which is a textual format for the control and data flow, thus representing the neuron's functionality. Figure 8.7 shows the *ICR* format for a PE belonging to the hidden layer.

Hardware synthesis actually starts from the *ICR* format. After the tree structure has been converted into the *ICR*, an internal CDFG is created, which is partitioned into several simple data flow graphs comprising only one operation node, which has up to two input nodes and one output node. This is equivalent to decomposing assignment statements in a programming language to form statements that have only two operands on the right side of the statement.

The partitioning of the entire graph into simple expressions, generally results in the proliferation of temporary variables for holding intermediate results (see Figure 7.5). Although this might seem to complicate the design, in fact, it does not, because temporary variables are eliminated during data path synthesis. Those that remain are compulsory, representing intermediate values for storage, which are necessary for the computation.

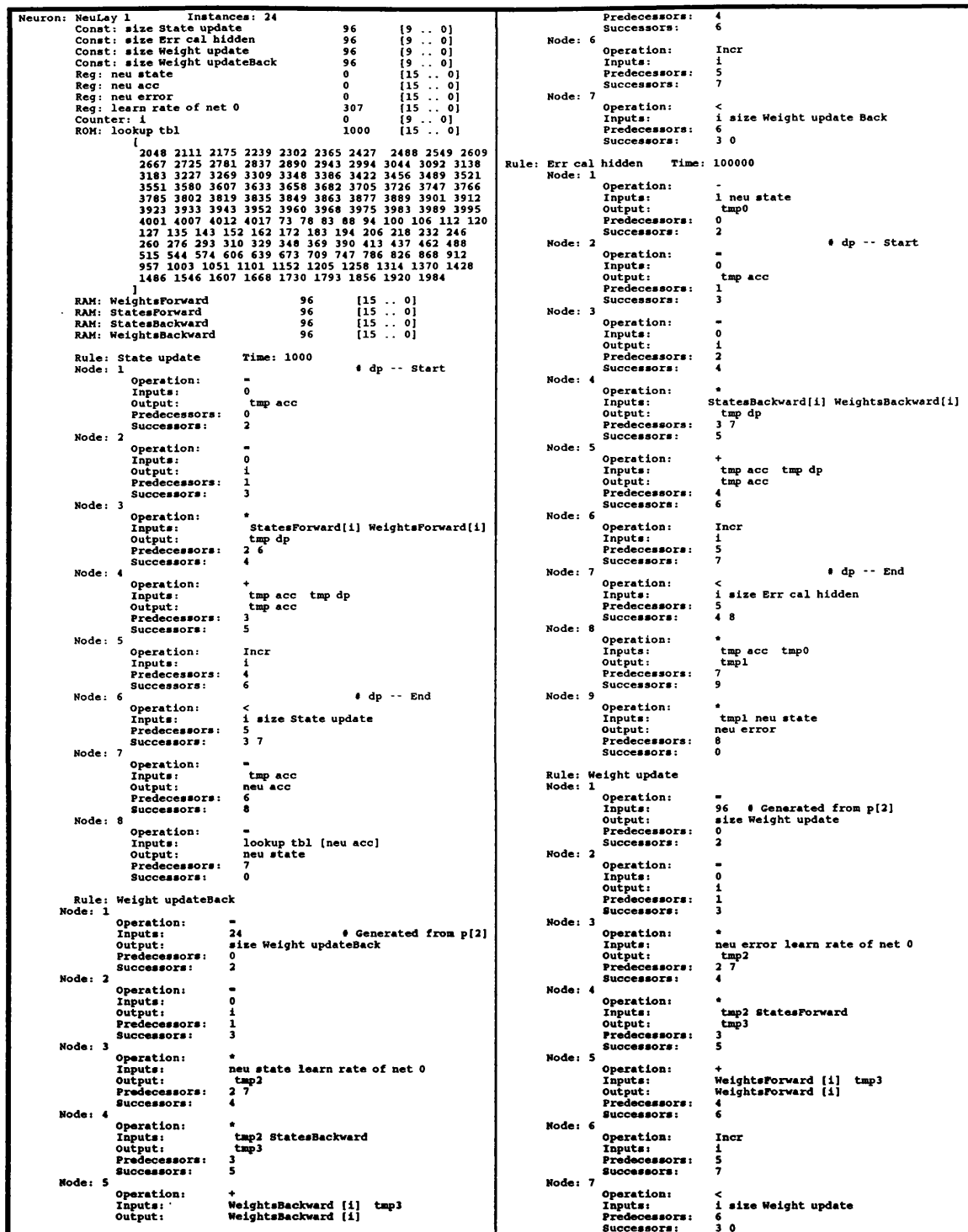


Figure 8.7 — ICR Format for a Hidden Layer Neuron

Figure 8.8 shows graphically the CDFG for each of the four rules used in the Back Propagation network for neurons in the hidden layer¹. It is upon this graph structure that the hardware synthesis takes place. However, before the hardware synthesis is employed, some local and global transformations on the graph are performed.

¹In fact, this CDFG is resulted after some transformations have been performed. Note that it is the initial sequential graph, improved by implementing some simple operations in parallel.

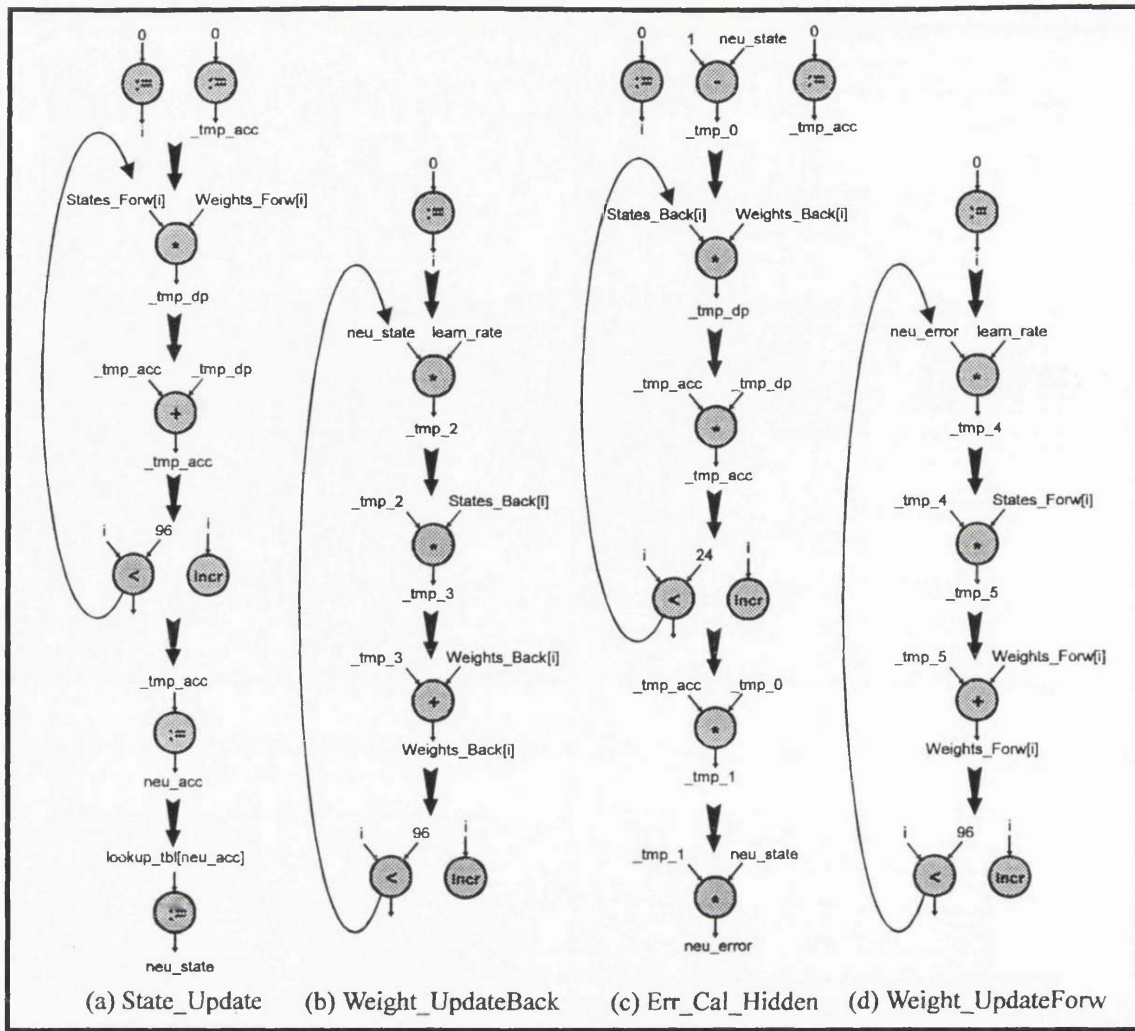


Figure 8.8 — The CDFG for the Hidden Layer's Rules

After the graph is partitioned, an *activity list* for each variable is created, so that lifetime analysis can be performed. Figure 8.9 presents a graphical representation for the activity list.

This graphical view is computed internally by the synthesis algorithms through the activation list, which is an adequate representation used throughout the allocation and scheduling algorithms. The activation list for some of the storage elements in Figure 8.9 is computed as shown in Table 8.2.

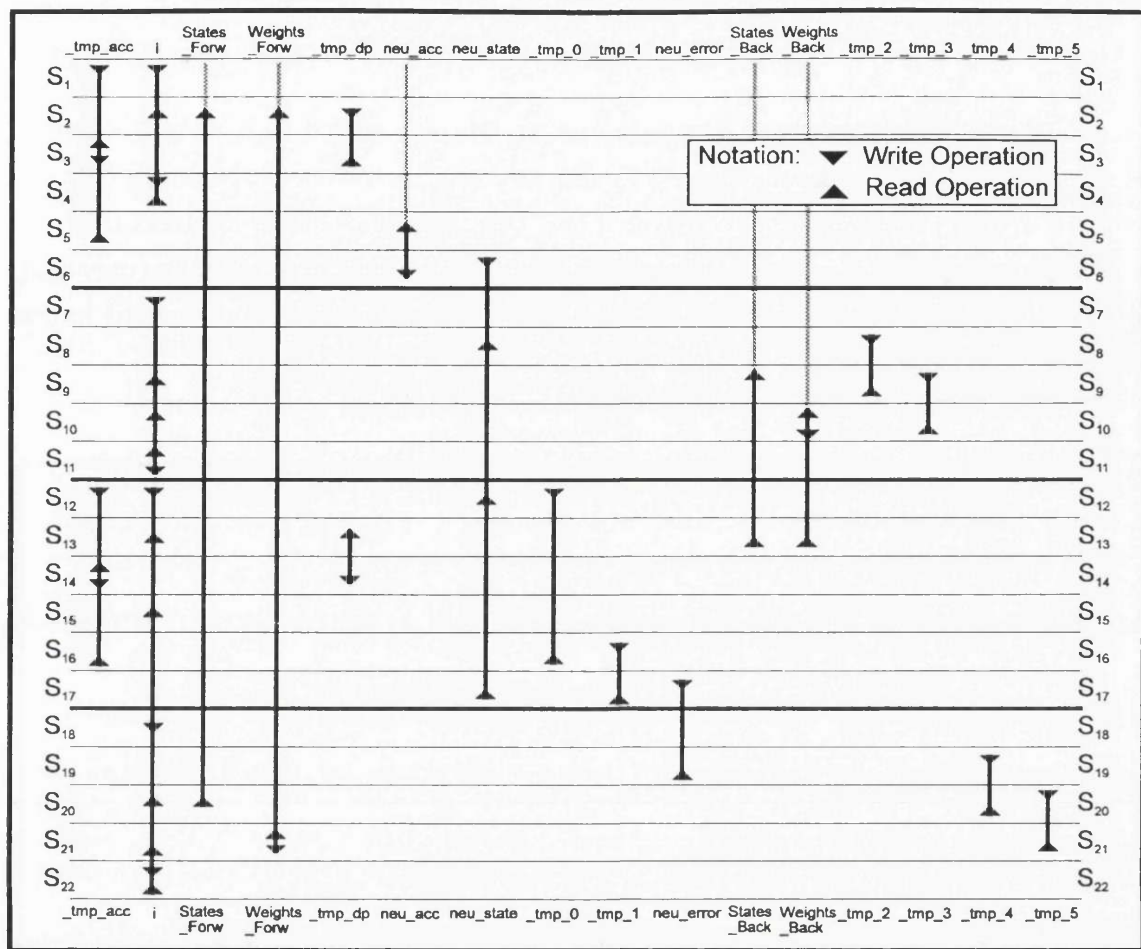


Figure 8.9 — Lifetime Analysis for Variables in the Back Propagation Example

States	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
_tmp_acc	2	5	3	5	1	0	0	0	0	0	0	2	5	3	5	1	0	0	0	0	0	0
i	2	1	5	4	0	0	2	5	1	1	4	2	1	5	1	0	5	2	5	1	1	4
States_Forw	5	1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	1	0	0
neu_state	0	0	0	0	0	2	5	1	5	5	5	1	5	5	5	5	1	0	0	0	0	0
Obs: dead = 0, rd = 1, wr = 2, rdwr = 3, wrrd = 4, idle = 5.																						

Table 8.2 — Activity List for Some Storage Elements

Based upon the activation list, several optimisations' steps are possible. For example, by employing the algorithm described in section 7.4.2, it can be seen that the following registers can be merged:

- _tmp_acc, neu_error, _tmp_2 neu_error.
- _tmp_dp, neu_acc, _tmp_1, tmp_3, and _tmp_4;
- neu_state and _tmp_5.

Note that `_tmp_5` could not be merged with `neu_error` because both registers are used inside the same loop (see Figure 8.8.d). Therefore, `neu_state` is later merged with `_tmp_3`.

In addition, the registers `Size_state_update`, `Size_err_cal` and `Size_Weight_UpdateB`, declared in the *ICR*'s *DataPart* shown in Figure 8.7, have been eliminated. This is because they are written just once with a constant value. Therefore, a simple replacement of these registers by the correspondent constant value each time they are read gives an equivalent effect. Figure 8.10 shows the result obtained after the transformations performed in the graph presented in Figure 8.8.

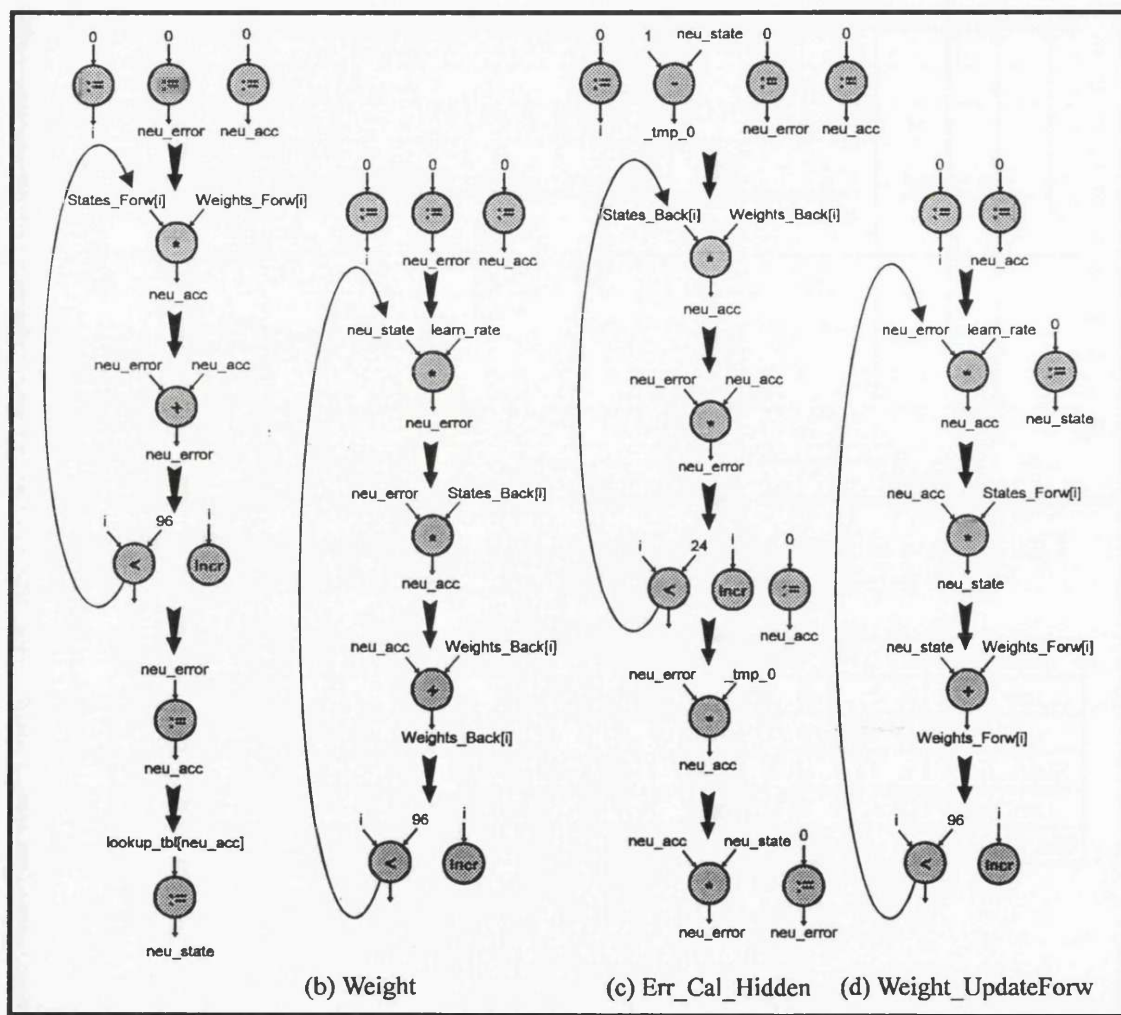


Figure 8.10 — The Transformed CDFG for the Hidden Layer's Rules

As a result, only four registers (`neu_acc`, `neu_error`, `neu_state`, and `_tmp_0`) are necessary to implement the whole Back Propagation computation, namely the learning and recall phases. This corresponds to fewer registers than used in the manual prototype developed in chapter 6, in which 7 registers were used. This saving in the number of

registers is a result of the approach taken by the synthesis algorithms described in the previous chapter.

8.4.2. Data Path Synthesis

The synthesis of the hardware structure starts by reading the internal graph (obtained from the transformation steps performed upon the *ICR*) and allocating storage elements, functional units, and busses, as explained in section 7.7.6. Two main busses are initially provided, namely *Bus A* and *Bus B*. The operations in the graph are then read and all three hardware structures are allocated in the same iteration. In addition, control information is stored in memory to feed the control synthesis with enough information on the sequencing and signal activations necessary to run the FSM.

It must be noted that at this stage the number of storage elements has been optimised by the previous transformation steps. All temporary variables have been eliminated, except those that are essential to the correct computation. The number of functional units is minimum, i.e., only one unit for each different operator is provided. Although this approach exploits parallelism in a limited way, it generates a highly optimised data path structure in terms of silicon area.

To illustrate the hardware allocation process, Table 8.3 shows the steps taken by the algorithm to allocate registers and bind them into appropriate busses. The allocation of other storage elements is performed in a similar way.

In State 1 three operations can be performed in parallel (see also Figure 8.9). Firstly, the counter *i* and the register *neu_error* are both assigned with the constant zero. Secondly, because *neu_acc* is the output of a multiplication, it is initially loaded with the constant zero (see section 7.7).

In State 2, an input port is created for *neu_acc*, and connected to *Bus B*, since it is the result of a multiplication.

In State 3, register *neu_error* is read and written in the same cycle. This is possible due to the multi-port structure of the registers (see Figure 7.7). In this case, one output port is created to read the current data into *Bus A*, and one input port is created to write a new value from *Bus C*, which is computed by the adder.

State	Register	Type of Port	Bus
S_1	neu_error	Input	local to constant 0
	neu_acc	Input	local to constant 0
S_2	neu_acc	Input	Main Bus B
S_3	neu_error	Output	Main Bus A
	neu_acc	Output	Main Bus B
	neu_error	Input	ALU local Bus C
S_4	✕	✕	✕
S_5	neu_error	Use Output	Main Bus A
	neu_acc	Input	Main Bus A
S_6	neu_acc	Use Input	Main Bus A
	neu_state	Input	Main Bus B

Table 8.3 — Allocation and Binding of Registers in the Data Path

State 4 does not involve any activity in the registers. It is only used by the FSM to decide which next state should be executed.

In State 5 the same output port created before for register neu_error is used, and an input port is created to register neu_acc.

Finally, in State 6 the input port previously created to register neu_acc is used, while an input port is created to the register neu_state.

This process continues until all states are computed. Since several registers have been merged, the entire structure is simplified. Figure 8.11 shows the data path for the execution and memory units automatically created during data path synthesis, according to the algorithm described in the previous chapter. It is essentially the same structure developed manually in chapter 6. However, some minor differences arise due to the characteristics of the synthesis algorithm. For example, a local bus at the output of the ALU is created and shared by the registers. This strategy creates smaller FSMs, since a reduced number of cycles is employed. This is possible because any register can share the output of the ALU. In the manual design, a load cycle of the operands into the ALU is required before the operation can be accomplished.

The scheduling performed upon these allocated structures is shown in Figure 8.10. It is essentially the sequential schedule in which simple assignment operations are executed in parallel. This reduces the number of states necessary to build the FSM. Assuming that this design does not meet a particular timing constraint provided by the user, then the ASAP scheduling algorithm is performed. In this case, the algorithm cannot start from the

previous schedule and allocation strategy, because registers have been merged. Therefore, the ASAP algorithm is applied upon the graph presented in Figure 8.8.

The ASAP algorithm, as defined in section 7.6.4, can only be effectively applied if the output of a particular state is not the input of the subsequent state. Figure 8.8 shows clearly that the ASAP algorithm would not improve this design, because the CDFG is fully sequential. In fact, the obtained design can be considered optimal in the sense that no extra parallelism can be added to it.

After hardware allocation and scheduling have been accomplished, all hardware structures are established, and therefore the module generation tool can be executed. This comprises a set of function calls to the components supported in the VHDL library. This tool searches the entire graph and for each storage element and functional unit the appropriate function is called, which generates the actual module from the parameters defined by the data path.

The creation of the communication unit's data path is straightforward, resulting in a structure identical to the one presented in section 6.5.1. The *ICR*'s *DataPart* contains information regarding the networks topology. Therefore, the *NSC* knows that a fully connected network containing three layers have been defined, and creates the necessary registers to hold the previous layer and next layer address registers, in addition to the *PE*'s own address register. Then the next step is to create busses. Two initial busses are provided (*Bus A* and *Bus B*) to provide communication between this unit and the other two units. A third bus is provided to send the execution unit's results (state and error values) outside the *PE*. A fourth and last bus is needed to address the *PE* and its internal memory.

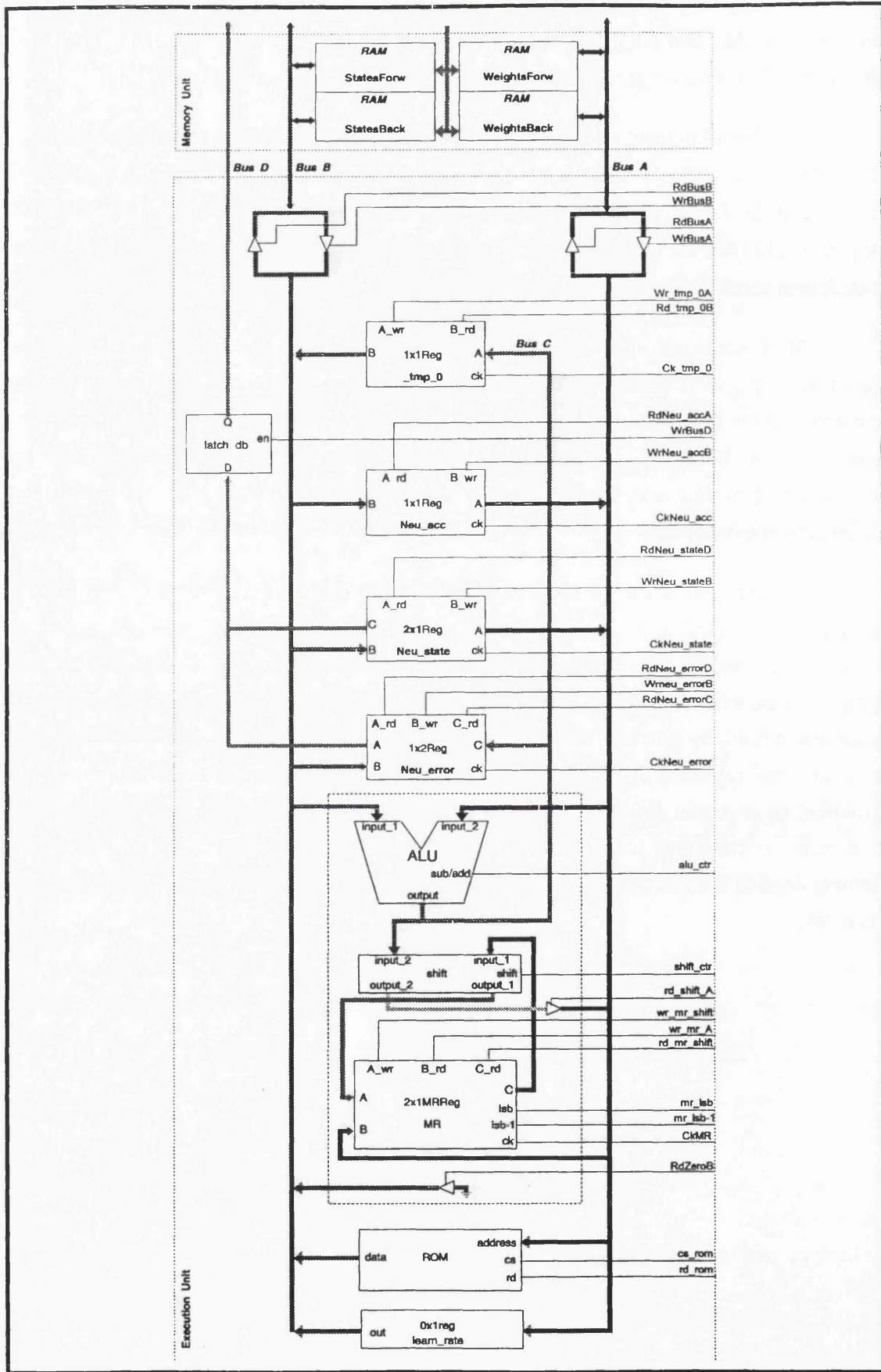


Figure 8.11 — Synthesised Data Path for the Execution and Memory Units

8.4.3. Control Synthesis

The control synthesis task follows the data path synthesis. It consists of creating a FSM, which can be implemented by, for example, a PLA. The result of the scheduling is explicitly stored in a specific data structure, in which elements represent the control states in the FSM. In each of these states, a list of signal activations is specified. The task of the control synthesis is to read this list and generate the control signals as well as the *next state* information.

Figure 8.12 presents the list of actions to be performed by the FSM in each of the given control states, according to the register allocation shown in Table 8.3.

<i>State</i>	<i>Variable</i>	<i>Action</i>
S ₁	neu_error	Activate Awr
	_tmp_dp	Activate Awr
S ₂	_tmp_dp	Activate Bwr
	i	Activate PtrRd
	States_Forw	Activate RdState and CsState
	Weights_Forw	Activate RdWeight and CsWeight
S ₃	_tmp_dp	Activate Bwr
	neu_error	Activate Ard and Bwr
S ₄	_tmp_dp	Activate Crd
	i	Activate IncrPtr and Test EndCount: True: Jump S ₂ False: Jump S ₅
S ₅	neu_error	Activate ARd
	neu_acc	Activate AWr
S ₆	neu_acc	Activate BRd
	neu_state	Activate AWr

Figure 8.12 — Generated List of Actions for the FSM

The list of actions is fed into the control synthesis tool, which is enough to create a FSM. However, to correctly generate a controller, some assumptions are made, such as whether a particular signal is active either on '1' or '0', and if memories, counters, and flip-flops are edge triggered or level sensitive.

After the execution and memory units have been synthesised, the communication unit's can be created. The structure of the communication unit's control module is identical to the one developed in section 6.5.1. The FSM obtained through synthesis is created according to the necessary actions, which are obtained through the algorithms described in section 7.8. These actions are divided in two phases: initialisation and execution.

The initialisation phase consists in initialising registers, holding parameters, and memories, holding the initial weight values needed to start the neural algorithm. The

information concerning which storage element should be initialised is implicitly provided by activity list (see Figures 8.9 and Table 8.2). Clearly, all four memory block (StatesForw, WeightsForw, StatesBack, and WeightsBack) plus the registers `neu_acc` and `learn_rate` must be initialised, since they are first read without having been previously written. Therefore, the control states necessary to read these initial values are built into the FSM. The execution phase is fixed, and consists of commanding the other two units and sending results to other PEs, as described in section 6.5.1.

8.5. VHDL Description

The VHDL description of the entire chip is created in almost every phase of the *NSC*. After data path synthesis, the module generation function is invoked. The data path structure is then created according to the definition of the storage elements, functional units, and busses.

The module generation phase simply consists in the construction of a VHDL entity for each component in the design according to the algorithms defined in the previous chapter. The interconnection structure created before is then used to instantiate these modules and build VHDL entities that link the lower level components among them. This hierarchical design is exactly the same one used in chapter 6, and is greatly favoured by the use of VHDL.

8.6. Summary

This chapter has provided a step-by-step introduction of all actions performed by the *NSC* during the synthesis of a Back Propagation Neural Network. This in-depth study has demonstrated the strengths anticipated in the previous chapter. In particular, it has been shown that the design developed in chapter 6 could be further optimised through the techniques employed by the *NSC*.

The immediate consequence of the techniques developed in the previous chapter is that the number of registers used to implement the execution unit has been reduced from seven, in the manual design, to four, in this automatic design. In addition, it has been shown that a lower number of cycles is needed to implement the execution unit's controller.

These improvements in the chip design are possible due to the data path synthesis algorithms developed in the previous chapter. Moreover, the transformations on the *nC* data structure are fundamental for the success of this operation.

Finally, it must be stressed that the final design conforms fully with the *Generic Neuron* target architecture. This conformity is based upon the heuristics developed during the synthesis algorithms.

Chapter 9

Assessment

This chapter assesses the work developed in this thesis in each of its principal investigation topics, which are centred in the integration between software and hardware tools for neurocomputing. This includes the hardware extensions implemented in the Pygmalion system and the design of the neural silicon compiler.

9.1. Software and Hardware Integration

As stated in Chapter 1, the neural computing research area requires two specialised tools for executing artificial neural models: firstly, a flexible software tool that permits experimentation with different aspects of neural networks, thus providing a framework to execute existing as well as new algorithms; secondly, a massively parallel application-specific neurocomputer, which properly explores the intrinsic parallelism of neural networks, hence granting the necessary high performance for the neural application execution.

Several software simulators and several ASNNs have been designed, with many commercial products already available. However, these products have been designed independently, with no integration between software and hardware tools. The incompatibility between these two tools has had as a consequence that a neural network application tested in the software environment has to be fully re-designed from scratch if high performance VLSI neural chips are required for the computation of the particular application.

Therefore, a neural network programming environment encompassing integrated software and hardware tools, with the capacity of automatically generating ASNNs from a high level specification of the neural network application, is demanded. Such a complete programming environment permits a neural network application to be described and tested using the software tool. After all optimal parameters (both algorithm-specific and hardware-specific) have been chosen, ASNNs can be automatically produced without requiring from the user any knowledge of VLSI design.

The main goal of this thesis focuses precisely on the issue of software and hardware integration for neural computing, in which automatic generation of ASNNs is achieved from a high level description of a neural network application. This involves the adoption of an existing neural network programming environment, namely the *Pygmalion* system, and the extension of this environment to encompass a hardware route. Therefore, the research developed in this thesis has concentrated on the extension of the *Pygmalion* environment as a whole, and the development of the neural silicon compiler.

The extension of the *Pygmalion* system includes the following stages:

- extensions of the *nC* neural network language;
- incorporation of a hardware library for neural models, which represents the hardware counterpart of the software library; and
- introduction of a simulation tool for neural networks' hardware, which uses the hardware library to mimic the exact way neural networks are executed in hardware.

The design of the neural silicon compiler represents the essence of this dissertation, and basically includes the following tasks:

- development of high level transformations performed upon the *nC* input specification;
- design of a hardware-specific intermediate code representation for neural networks, namely *ICR*, which dissociates the neural network environment from the *NSC*, thus allowing the usage of the *NSC* in any neural network programming environment; and
- development of the hardware synthesis tools, comprising basically data path and control synthesis.

The following sections provide an evaluation of all these components according to the established goals.

9.2. *Pygmalion* Extensions

Following the basic proposal of this research, there are some extensions to this software environment that are clearly necessary. The principal reason is that the *Pygmalion* environment (and the vast majority of existing neural network software simulation tools) has been conceived as a software-oriented tool, in which the user can experiment with several different applications and neural models. After the neural network is fully configured and tested in this environment, there is no guarantee that the computation performed in hardware will be exactly the same as the one employed in

software. While the *Pygmalion* software components employ floating-point calculation, digital hardware generally uses fixed-point arithmetic. Along with the type of calculation performed in hardware, several other constraints are imposed in the design, as analysed in chapter 5.

Therefore, the simple provision of a neural silicon compiler to the *Pygmalion* environment is clearly not sufficient. Several hardware-specific extensions needed to be carried out in the *Pygmalion* system. Firstly, the neural network language needed to be extended to provide the same type of computation (during simulation) employed by the hardware's target architecture (during hardware execution). Secondly, the algorithm library needed to be replicated and modified to include all particularities present in the hardware's target architecture. Finally, the simulation tool needed to be expanded to provide an assessment of the neural network's target architecture, considering that finite precision arithmetic is carried out by the hardware.

These extensions have been developed with the following design goals previously established:

- The functionality of the original *Pygmalion* software environment should not be modified — the existence of every module should be maintained, and their functionality preserved;
- The new features should complement the environment — hardware extensions are *incremental*, i.e., none of the original functions are lost; instead, new features are added to the functionality of the environment; and
- The philosophy of the system should be maintained — this assumes that the user continues to use the system regardless of the new hardware features. Therefore, *nC* continues to be used to program neural algorithms and applications, and the simulation environment is used to test the particular application.

The next sub-sections provide an evaluation of the extensions performed in the *Pygmalion* system according to the established objectives.

9.2.1. *nC* Extensions

As discussed in section 4.3, *nC* is a software-oriented language, which is adequate for programming a neural network. However, *nC* does not support any hardware construct that permits the user to guide the process of generating ASNNs. Therefore, the language is extended to support such features. Nevertheless, to keep the design philosophy, in which hardware details are hidden as much as possible from the user, the adopted extensions to

the language include only the introduction of fixed-point arithmetic, as discussed in chapter 5, and the incorporation of new fields in the **system** data structure, presented in chapter 4.

The introduction of fixed-point computation is done internally in every built-in function defined in the language. The user's task concerns only the specification of the type of data handled throughout the simulation of the neural network. Conversely, the incorporation of hardware-specific fields in appropriate parts of the **system** data structure provides the user with the required flexibility to choose hardware constraints.

It must be noted that the introduction of hardware-specific parameters does not defeat the original philosophy of keeping hardware issues transparent to the user, since these new parameters are kept at a high level of abstraction. Hence, the user may specify that the `State_Update` function must be accomplished within a certain time, or that the whole recall or learning procedure should not exceed a determined amount of time. Similarly, the user may wish to have its entire neural network implemented in not more than a given number of chips. The decision of how these restrictions should be met is made by the NSC.

Another important issue for neural algorithms is the activation function. The language is extended to support a lookup table mechanism, which is commonly used in digital implementations of neural chips. The extension includes the definition of an additional built-in function, called `lookuptbl`, which when invoked, performs the table's indexing mechanism according to its size (specified by the user). The activation function is specified by a function that contains the actual description of the activation function to be performed by each neuron. This macro is then used to build the contents of the digitised table for the specified activation function. No provision was made to implement an activation function directly in hardware, since this represents high cost in terms of silicon area. However some activation functions, such as the hard limiter (which only requires a comparator in hardware) are more cost effective than lookup tables. In this case, the system permits the definition of macros that specify threshold values.

9.2.2. Hardware-Specific Algorithm Library

The *Pygmalion* system includes a library of the most popular neural network models. This library was expanded to include hardware-specific features, which are defined by the target architecture of the ASNNs. The expanded library, named hardware library, is functionally identical to the software counterpart, but internally it is modified to perform calculations according to the defined target architecture. This library was built because software simulations (generally employing floating-point calculations) differ considerably

from hardware simulations (performed in fixed-point). The differences between the results obtained through software and hardware simulations were analysed in chapter 5.

Although this approach is opened to several different target architectures, for the purposes of this dissertation, only the *Generic Neuron* target architecture was implemented. However, the hardware library of neural algorithms can be gradually expanded. This is done in conjunction with the extensions performed upon the *NSC* to support the synthesis of ASNNs based upon different target architectures.

The hardware-specific algorithm library is used during simulations of the neural network, as well as during the synthesis of the ASNNs through the *NSC*. This approach guarantees that the simulated network is executed in the same way as the neural network in hardware, thus yielding an *a priori* measure of the actual performance of the hardware.

9.2.3. Simulation of Neural Networks Hardware

The incorporation of a simulation tool for the actual hardware is extremely important for two basic reasons. Firstly, it provides a way of assessing if the user's choice for the hardware parameters, in particular the data precision, is adequate to correctly perform the neural computation. Since hardware constraints are often employed in digital implementations, such as the ones discussed in chapter 5, correct calculations are not guaranteed. Secondly, it permits the tuning of the ASNNs both in terms of speed and area. By experimenting with several different configurations, the user may find a set of hardware parameters that correctly implement the particular application in an optimal fashion.

The implemented simulation tool is fully parameterised, so that several different experiments can be performed by the user. This includes, for example, the experimentation with different data precision, several different sizes for the lookup table, different schemes for realising the activation function, and various mechanisms for dealing with overflows during calculations. As a consequence, the tool proved to be a useful platform to investigate the influence of basic hardware-related parameters upon the neural computation. Such investigation has been developed in chapter 5 for the Back Propagation neural network algorithm.

9.2.4. Summary

It is believed that all main objectives have been achieved by incorporating the above extensions in the *Pygmalion* environment. The new system has gained a new dimension, which permits that, after the network has been configured, tested, and satisfied

the user regarding the solution of the final application, a hardware route can be taken to yield neural chips that provide extremely high performance during the two phases of neurocomputing.

The functionality of the system is maintained, meaning that the original approach for designing a neural network application has not been changed by the introduction of hardware-related extensions. Therefore, if the user decides to take a hardware route after the network has been fully configured and tested, then all that is needed is:

- to specify the employment of fixed-point calculation — the present implementation considers only digital architectures for the hardware;
- to specify what target architecture is envisaged for the final chips — the present system supports only the *Generic Neuron* architecture;
- to define the hardware parameters — this involves the definition of the data precision, the chosen method for realising activation functions, etc.;
- to run the hardware simulator until the obtained results are satisfactory — this involves simulating the recall and learning (if required by the application) phases of the specified neural model according to the adopted target architecture; and
- to run the *NSC* — this results in the synthesis of a set of neural network chips, according to the specified application and the hardware-related parameters.

Finally, the hardware simulator can also be used to find the best target architecture (if more than one is supported by the *NSC*) for the particular application.

9.3. Neural Silicon Compiler

The design of the neural silicon compiler concludes the extension of the *Pygmalion* environment, and indeed, represents the core of this research. It effectively provides the hardware route needed to implement massively parallel neurocomputing systems. The development of the *NSC* followed the goals stated below:

- *High level synthesis* — it should start from the high level neural network programming language *nC* and create a hardware structure for the neural chips at the register transfer level;
- *Independence* — from any particular neural network programming system, meaning that its design should be flexible enough to be integrated into several different existing software environments for neural networks;

- *Target Architecture* — the synthesised hardware structure should conform to a pre-defined target architecture, which is held in the hardware library of the *Pygmalion* environment. Data path and control synthesis algorithms should be capable of generating a structure that conforms with the target architecture; and
- *VHDL chips* — to allow an easy integration of the *NSC* with commercial low level synthesis tools (which ultimately generate the layout mask for the chips) the register transfer level of the synthesised chips should be described in a standard format. For this reason the IEEE standard hardware description language (VHDL) was chosen.

According to the above design goals, the following sub-sections evaluate the major components of the *NSC*: the high level transformations performed upon the *nC* input specification; the definition of the hardware-specific *ICR*; the adoption of the *Generic Neuron* target architecture; the development of the data path and control synthesis; and the final generation of VHDL neural chips.

9.3.1. *nC* High Level Transformations

As discussed throughout this work, the *nC* language was extended to include hardware-related parameters. However, *nC* was not transformed into an HDL. Consequently, special mechanisms to efficiently map a *nC* program into a compact hardware structure are required.

Therefore, this research concentrated much of its efforts into the high level transformations performed at the *nC* level. Parts of these transformations are software-like, but the most important ones are hardware-like optimisations that basically transform the complex (regarding hardware implementation) *nC system* data structure into a simpler data representation for mapping variables into efficient hardware structures.

As explained in chapter 7, the *tag* mechanism incorporated into the **RULE** data structure is paramount for these transformations. Therefore, generic and extended parameters, held in the memory as a string of data values can be extracted as actual named variables.

The importance of this approach is that each element in the generic and extended parameters' list can be assigned to a particular type of storage, which eases enormously the task of the *NSC*'s hardware allocation algorithm.

9.3.2. Intermediate Code Representation

To attain a complete separation between the *NSC* and the neural network programming environment, the *ICR* was defined. It is a very simple format for describing a neural network at the hardware level. It is not, however, supposed to be the users' input language.

By using such a representation, virtually any existing neural network programming environment can be extended to get the benefit of a hardware route to the neural network application.

The *ICR* proved to be an effective format for the specification of complete neural networks including their data and functions to be performed by hardware. Its definition is in conformity with neuron-based architectures [190], such as the *Generic Neuron*, which emulates the neuron's functionality in each processor. Its suitability for weight-based architectures, which focus on the synapse functionality by providing a matrix of multipliers, remains to be investigated.

9.3.3. Target Architecture

One of the requirements defined during this work is that the target architecture for the neural chips should be capable to implement the vast majority of the existing neural models. The final implementation of these chips should be, however, targeted to a particular neural model, as specified by the neural network's application designer. Nevertheless, the target architecture should be flexible enough to allow the synthesis of different chips according to new neural models specified by the user.

The *Generic Neuron* model was conceived for this purpose [190], being a natural choice for the *NSC*'s target architecture. Indeed, in chapter 5 this architecture was assessed and proved to be effective for designing neural chips. Furthermore, a VLSI prototype implementation was fabricated in 2 μm CMOS technology, as part of another PhD thesis [190].

The basic strengths of the *Generic Neuron* are:

- *Flexibility* — the PE's functionality is able to perform several different learning algorithms, and the bus interconnection strategy permits the implementation of a wide spectrum of network topologies used by neural models;

- *Scalability* — the internal architectural framework allows that, in view of the constant improvements in VLSI processing technology, the number of PEs per integrated circuit be increased without affecting the number of pins; and
- *Simplicity versus High Performance* — by employing simple design alternatives, very compact structures are obtained for the PEs, and high parallelism is explored, yielding the required performance.

The *Generic Neuron* achieves this features by dividing its internal organisation in three distinct units: communication, memory, and execution. The task of the *NSC* is to synthesise ASNNs based upon a hardware structure composed of these three units.

The execution and memory units are responsible for the execution of the neural algorithm. Their synthesis comes directly from the functionality defined in the neural algorithm. The communication unit controls the activities in the PE and commands data exchange among the PE, the rest of the network, and the central controller. Its synthesis is derived from the implicit information held in the *ICR's* CDFG, and performed after the other two units have been synthesised.

9.3.4. Data Path Synthesis

To achieve the user's requirements regarding the hardware-related parameters specified in the extended *nC* input language, the algorithms for allocation and scheduling are performed in two basic forms. Firstly, the *default mode* attempts to produce the most possible compact structure. This is achieved by focusing the synthesis on the hardware allocation. The scheduling is greatly simplified, consisting of a simple sequential strategy, which is derived from the *nC* input description. Parallelism is explored in a limited way, where simple assignment operations are grouped together into a single control state. Nevertheless, since hardware components are not duplicated, the goal of compactness is achieved. Secondly, the *user-driven mode* implements the *ASAP* scheduling algorithm to explore parallelism, so that timing constraints defined by the user can be met. Scheduling is thus the focus of the synthesis algorithms. As a consequence, the produced structure may lead to larger silicon area.

The implementation of such approach increases greatly the power of the *NSC* regarding the user's specified constraints. However, it is felt that a further investigation is necessary, in particular for the *user-driven mode*. For the purposes of this dissertation, only extreme cases have been realised. If the user does not specify timing constraints, but specifies area constraints, then the default mode is activated, which results in minimal hardware resources and slower execution. Conversely, if timing constraints are specified,

than the opposite approach is followed, i.e., the *ASAP* strategy is implemented resulting in a faster circuit, but larger in terms of area. An alternative for the above strategy is to employ a trade-off between the two modes.

Concerning the internal algorithms employed by the synthesis tools, it is believed that the heuristic approach has contributed massively to obtain a result that it is virtually identical to that developed manually. The *activity list*, which provides important information on the data flow and lifetime of every variable in the *nC* program, is extremely important during the execution of these algorithms. Similarly, the *activity list* provides information upon which variable should send values to other PEs and when this should occur.

9.3.5. Control Synthesis

The task of generating a controller for the PE's execution and communication units is performed after the data path is constructed. This already involves the scheduling of operations to specific control states. Therefore, the control synthesis' role is to generate a finite state machine that accurately implements the desired schedule.

The communication unit's controller is synthesised as the last step in the synthesis, since its controller is heavily dependent upon the number of initialisations' steps required by the storage elements allocated in the execution unit's data path. Again, the *activity list* plays an important role in deciding which variables need to be initialised. This initialisation is monitored by the communication unit that takes data from the outside of the PE. The data is supplied by the central controller, which is responsible for commanding the PE's initialisation phase.

9.3.6. VHDL Description of Neural Chips

The generation of the VHDL code for the PEs employing the *Generic Neuron* model permits that a complete simulation of the neural network hardware is attained. Special attention has been taken to produce a code that is easy to synthesise by current low level synthesis tools based on VHDL.

However, it cannot be guaranteed that the VHDL output will be input into these low level tools without any modifications, since tests could not be performed due to the lack of such tools in the Department.

Nevertheless, the whole design has been simulated and proved to work satisfactorily, according to the targets established for this research.

Chapter 10

Conclusions and Future Work

This final 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 potential future work.

10.1. Summary

The primary goal of this thesis was to investigate the integration between software and hardware tools for neurocomputing. The integration consists of incorporating a neural silicon compiler into existing neural network programming environments. The motivation for developing such an integrated system has been the ultimate goal for producing a complete neural programming system that provides the user with two complementary tools: a software tool to investigate neural network algorithms and applications; and a hardware design tool to automatically generate ASNNs from a high level specification of the user's neural application.

This thesis concentrated on the design of the high level synthesis of the neural silicon compiler. Nevertheless, as discussed throughout this thesis, a complete and integrated system requires further components in the environment. This included basically the addition of a hardware library and a simulator for the neural network hardware. Furthermore, for performance reasons, the input language had to be expanded, so that the hardware synthesis tools could be supplied with some guidance from the user.

The *nC* language was specified as part of the *Pygmalion* project [24]. The language's design was focused upon the software aspects of the neural network's execution. This execution could be either in sequential or parallel machines. However, *nC* was not conceived to be the input language for a silicon compiler. The implications of this lack of hardware-related features are that the silicon compiler is left with very little information of how efficient hardware structures are created.

The extensions performed in *nC* addressed the issue of hardware-specific parameters, and simultaneously preserved the basic characteristics of the language. This approach guarantees that the user will require very little or no knowledge of VLSI design.

The introduction of a hardware library into the *Pygmalion* environment permits the functionality of the neural models to be described at the software and hardware level separately, thus providing simulation facilities for both end applications. While the software simulation is used to configure the application and test the learning of the neural model adopted, the hardware simulation gives an accurate prediction of the hardware performance.

The design of the parameterised simulator for neural networks' hardware provides an important tool to investigate the influence caused by hardware constraints during the execution of the neural network.

The neural silicon compiler was developed with the primary goal of automatically generating highly optimised neural chips from the high level language *nC*. The use of heuristic procedures during the hardware synthesis, aiming at the construction of the *Generic Neuron* framework for the processing elements, guaranteed that designs virtually identical to the ones expected from a skilful hardware designer are successfully achieved.

The implemented high level synthesis algorithms demonstrated their effectiveness in producing optimised hardware structures according to the *Generic Neuron* definition. The high level transformations performed at the *nC* level of the specification play an important role, since they remove every redundancy introduced by the user and during the compilation of the *nC*. The definition of an *activity list* proved to be extremely important, and it is used throughout the remaining phases of the hardware synthesis process.

The adopted strategy for data path and control synthesis showed the ability to produce very optimised neural chips, both in terms of area and speed. The use of multiple busses along with the two-phase clock mechanism and multi-port storage structures, which are used in the prototype of the *Generic Neuron*, proved to be very effective. Although some investigation is still required to analyse the possibility of further improvements in the quality of the circuits, the implemented algorithms have completely fulfilled the goals of this research.

10.2. Research Contributions

In pursuing the research described in this thesis, the prime consideration was to investigate the possibility of expanding neural network environments to allow an easy, fast, and reliable process to generate high performance ASNNs. It is believed that such a design framework proved feasible. In particular, this thesis provided a simple, yet powerful, hardware environment for the implementation of neural chips. In summary, it is felt that the basic research contributions of this thesis are:

- ***Extension of the Pygmalion Environment*** — A methodology for extending the *Pygmalion* system to provide a hardware path from the *nC* language was proposed and implemented. Considering the generality of the adopted approach, the same extensions can be introduced to other neural network environments.
- ***Simulation of Neural Networks Hardware*** — The implemented simulator for assessing the hardware performance during execution of neural network models demonstrated to be a useful tool for studying the effects caused by hardware constraints. In this respect, exhaustive simulations of the Back Propagation learning and recall procedures were accomplished for digital implementations employing fixed-point calculation, such as the *Generic Neuron* architecture. The results obtained validated the feasibility of the target architecture in realising the learning algorithm. Moreover, bounds for hardware parameters were identified and used as the basis for the implementation of the neural chips. An important characteristic of the simulation tool is its generality, meaning that the tool can be used to assess the hardware performance of any neural model previously specified.
- ***Intermediate Code Representation*** — The definition of a hardware-related intermediate representation for neural networks provides a complete separation between software and hardware tools. This permits a general system to be constructed following a *many-to-many* approach. In this case, the system would support *many* software environments at the top and *many* target architectures at the bottom.
- ***VHDL Implementation of Neural Chips*** — The development of a *Generic Neuron* prototype of the Back Propagation neural network in VHDL confirmed the results anticipated during the hardware simulation. Furthermore, it served as the basis for the definition of the output specification to be generated by the *NSC*. In developing such a prototype, a library of VHDL modules been built, which is used during the high level synthesis of the *NSC*.
- ***Neural Silicon Compiler*** — The design of the *NSC*, which ultimately creates the hardware structure for the neural chips, follows a heuristic approach, which guarantees that designs close to the ones specified by expert hardware designers are successfully obtained. The development of several rules for transforming *nC*'s internal data structure into simpler ones demonstrated its importance in reducing complex and time consuming operations into concise and effective ones, thus allowing the synthesis of compact and fast circuits.

Finally, this hardware exploitation of neural networks was fully integrated with the software environment. It has been pointed out that this integrated system represents the

current trend in the development of neural network applications, which has a well-defined design cycle. Firstly, the user specifies the application, configures the neural network, and submits the network to the learning procedure. After this phase has been finished, the network can be directly mapped onto hardware. However to tune the design of the neural chips, a further simulation step is accomplished, taking into consideration hardware-specific parameters. Then, the *NSC* is finally executed to produce the specified integrated circuits.

10.3. Future Work

Regarding the development of a system that automatically produces ASNNs at the layout level of the integrated circuits, there are still a number of tasks to be carried out. Some of the most interesting ones are listed below:

- ***Integration of the Low Level Synthesis Tools*** — To achieve a complete route to the fabrication of ASNNs, the output of the *NSC*'s high level synthesis should be input into one of the several existing low level tools. This integration should be such that a complete feedback from the low level tools is provided to the high level synthesis tools, so that an accurate decision can be made at the high level of the process, which considers the adopted technology process.
- ***Investigation of More Elaborated Data Path and Control Synthesis*** — Although the algorithms implemented in this research proved to be simple and efficient, a further investigation is required to accurately evaluate the quality of the final integrated circuits.
- ***Expansion of Possible Target Architectures*** — Further investigation is needed to assess the suitability of the development of a generic neural silicon compiler, where multiple target architectures are allowed. In particular, when different techniques, such as analogue neural network chips, and even novel technologies, such as optical and opto-electronics are envisaged, the entire development of the *NSC* should be revised.
- ***Mapping Virtual Neurons onto PEs*** — It is interesting to investigate the *Generic Neuron* architecture's suitability in mapping more than one neuron (*virtual neuron*) onto a single PE. This task is extremely appropriate for silicon compilation. Based upon the mapping strategy, the silicon compiler could create different structures, which would encompass the extra control logic and additional storage elements to PEs implementing more than one neuron.

- **System synthesis** — As briefly discussed in chapter 7, the DataPart of the *ICR* may include information about the hierarchy of a neural network. Then, this information may be used to implement a system synthesis tool, which would be capable of not only synthesising neural chips, but also a complete neurocomputer system, comprising the network of PEs, the central controller, and additional circuitry to execute a particular application.

Other future work, based on the scope of this research comprises: firstly, different neural network programming environments can be added to the present system, so that a multi-environment system is obtained, as proposed in this dissertation and illustrated in Figure 6.2; secondly, the expansion of the hardware simulation studies to other neural network models can be accomplished, which will represent a broader analysis in executing neural network in hardware.

Finally, with the experience achieved during this work, another interesting investigation is the adaptation of the *NSC* to different paradigms, such as genetic algorithms and fuzzy logic systems. This would produce a silicon compilation system capable of synthesising *intelligent* chips, according to the paradigm specified by the user.

References

- [1] "80170NX Electrically Trainable Analog Neural Network", *Intel Data Sheet*, June 1991
- [2] "Directory of Silicon Compilers", *VLSI Systems Design*, pp. 52-56, March 1988
- [3] "Draft Standard VHDL Language Reference Manual", *IEEE P1076-1992/A*, 1992
- [4] "Electronic Design Interchange Format Version 2.0.0 - EIA Interim Standard No. 44", *Electronic Industries Association, Engineering Department*, May 1987
- [5] "HardwareC - A Language for Hardware Design - Version 2.0", Technical Report Stanford University, *Technical Report CSL-TR-90-419*, pp. 1-50, April 1990
- [6] "IEEE Standard VHDL Language Reference Manual", *IEEE Std 1076-1987*, March 31, 1988
- [7] "UDL/I Language Reference - Version 1.0m", Japan Electronic Industry Development Association, May 15, 1992
- [8] Abu-Mostafa, Y. S. and Psaltis, D., "Optical Neural Computers", *Scientific American*, vol. 256, no. 3, pp. 66-73, March 1987
- [9] Ackley, D.H., Hinton, G.E., and Sejnowski, T.J., "A Learning Algorithm for Boltzmann Machines", *Cognitive Science*, vol. 9, pp. 147-169, 1985
- [10] Acosta, R.D., Alexandre, M., Imken, G., and Read, B., "The Role of VHDL in the MCC CAD System", *25th ACM/IEEE Design Automation Conference*, pp. 34-39, 1988
- [11] Aho, A.V. and Ullman, J.D., *Principles of Compiler Design*, Addison-Wesley Series, 1977
- [12] Aleksander, I. and Morton, H., *An Introduction to Neural Computing*, Chapman and Hall, 1990
- [13] Aleksander, I., "Adaptive Pattern Recognition Systems and Boltzmann Machines: A Rapprochement", *Pattern Recognition Letters*, vol. 6, no. 2, pp. 113-120, July 1987
- [14] Aleksander, I., Thomas, W.V., and Bowden, P.A., "WISARD - A Radical Step Forward in Image Recognition", *Sensor Review*, pp. 120-124, July 1984
- [15] Alippi, C. and Nigri, M.E., "Hardware Requirements for Digital VLSI Implementation of Neural Networks", *Proceedings of the IJCNN'91*, pp. 1873-1878, Singapore, November 18-21, 1991
- [16] Alippi, C. and Storti-Gajani, G., "Simple Approximation of Sigmoidal Functions: Realistic Design of Digital Neural Networks Capable of Learning", *Proceedings of the International Symposium on CIRCUITS AND SYSTEMS*, Singapore, 11-14 June 1991
- [17] Alippi, C., "Weight Representation and Network Complexity Reductions in the Digital VLSI Implementation of Neural Nets", Department of Computer Science, University College London, *Research Note RN/91/22*, pp. 1-20, February 1991

- [18] Anderson, C.W., "Learning to Control an Inverted Pendulum Using Neural Networks", *IEEE Control Systems Magazine*, pp. 31-37, 1989
- [19] Angeniol, B. and Treleaven, P., "PYGMALION - Neural Network Programming & Applications", *ESPRIT II - Project 2059*, 1989
- [20] Angeniol, B. et al., "The Galatea Project", *Proceedings of the NeuroNimes'92*, 1992
- [21] Angeniol, B., "Pygmalion: ESPRIT II Project 2059, Neurocomputing", *IEEE Micro*, pp. 28-31, 99-102, December 1990
- [22] Asanovic, K. and Morgan, N., "Experimental Determination of Precision Requirements for Back-Propagation Training of Artificial Neural Networks", *Proceedings of the 2nd International Conference on Microelectronics for Neural Networks*, pp. 9-15, Munich, Germany, October 16-18, 1991
- [23] Athale, R. A., Friedlander, C. B., and Kushner, B. G., "Attentive Associative Architectures and their implications to optical computing", The BDM Corporation, *Proc. SPIE Optical Computing*, vol. 625, pp. 179-185, 1986
- [24] Azema-Barac, M., Hewetson, M., Recce, M., Taylor, J., Treleaven, P., and Vellasco, M., "PYGMALION Neural Network Programming Environment", *International Neural Network Conference*, Paris, France, July 9-13, 1990
- [25] Bailey, J. and Hammerstrom, D., "Why VLSI implementations of Associative VLCNs Require Connection Multiplexing", *IEEE International Joint Conference on Neural Networks*, vol. II, pp. 173-180, San Diego, 1988
- [26] Balakrishnan, M., Majumdar, A.K., Banerji, D.K., and Linders, J.G., "Allocation of Multi-Port Memories in Data Path Synthesis", *IEEE Transactions on Computer-Aided Design*, vol. 7, no. 4, pp. 536-540, April 1988
- [27] Batchman, T.E. and Parrish, E.A. Jr., "Integrated Optical Computing", *IEEE Computer*, pp. 7-8, 1987
- [28] Bell, T.E., "Optical Computing: a Field in Flux", *IEEE Spectrum*, vol. 23, no. 8, pp. 34-57, August 1986
- [29] Brayton, R.K. et al., "MIS: A Multiple-Level Logic Optimization System", *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, no. 6, pp. 1062-1081, November 1987
- [30] Brayton, R.K. et al., "The Yorktown Silicon Compiler System", in *Silicon Compilation*, ed. D.D. Gajski, Addison-Wesley Publishing Company, Inc., pp. 204-310, University of California at Irvine, 1988
- [31] Brayton, R.K., Hachtel, G.D., and Sangiovanni-Vincentelli, A.L., "Multilevel Logic Synthesis", *Proceedings of the IEEE*, vol. 78, no. 2, pp. 264-300, February 1990
- [32] Burich, M.R., "The Role of Logic Synthesis in Silicon Compilation", *Semicustom Design Guide*, pp. 56-61, 1988
- [33] Camposano, R. and Tabet, R.M., "Design Representation for the Synthesis of Behavioural VHDL Models", Elsevier Science Publishers, *Proceedings of the CHDL'89*, 1989

- [34] Camposano, R., "Design Process Model in the Yorktown Silicon Compiler", *25th ACM/IEEE Design Automation Conference*, pp. 489-494, 1988
- [35] Camposano, R., "From Behavior to Structure: High-Level Synthesis", pp. 8-19, October 1990
- [36] Camposano, R., Bergamaschi, R.A., Haynes, C.E., Payer, M., and Wu, S.M., "The IBM High-Level Synthesis System", in *Trends in High-Level Synthesis*, ed. R. Camposano and Wayne Wolf, Kluwer Academic Publishers, pp. 79-104, 1991
- [37] Camposano, R., Saunders, L.F., and Tabet, R.M., "VHDL as Input for High-Level Synthesis", *IEEE Design & Test of Computers*, pp. 43-49, March 1991
- [38] Carlson, S., *Introduction to HDL-Based design using VHDL*, Synopsis, Inc., 1991
- [39] Carpenter, G.A. and Grossberg, S., "The ART of Adaptive Pattern Recognition by a Self-Organizing Neural Network", *IEEE Computer*, pp. 77-88, March 1988
- [40] Carre, B., *Graphs and Networks*, 1979
- [41] Cheng, E.K. and Mazor, S., "The Genesil Silicon Compiler", in *Silicon Compilation*, ed. D.D. Gajski, Addison-Wesley Publishing Company, Inc., pp. 361-405, University of California at Irvine, 1988
- [42] Collins, E., Ghosh, S., and Scofield, C., "An Application of a Multiple Neural Network Learning System to Emulation of Mortgage Underwriting Judgements", *IEEE International Joint Conference on Neural Networks*, vol. II, pp. 459-466, 1988
- [43] Corbin, V. and Snapp, W., "Design Methodologies of the Concorde Silicon Compiler", in *Silicon Compilation*, ed. D.D. Gajski, Addison-Wesley Publishing Company, Inc., pp. 406-445, University of California at Irvine, 1988
- [44] Dayhoff, J.E., *Neural Network Architectures: An Introduction*, Van Nostrand Reinhold, New York, 1990
- [45] Dettmer, R., "ELLA - A Language for VLSI", *Electronic & Power*, pp. 517-522, July 1986
- [46] Devadas, S. and Newton, R., "Algorithms for Hardware Allocation in Data Path Synthesis", *IEEE Transactions on Computer-Aided Design*, vol. 8, no. 7, pp. 768-781, July 1989
- [47] Dewey, A., "VHDL and Next-Generation Design Automation", *IEEE Design & Test of Computers*, pp. 6-7, June 1992
- [48] Dillinger, T.E. et al., "A Logic Synthesis System for VHDL Design Descriptions", *IEEE International Conference on Computer-Aided Design*, pp. 66-69, 1988
- [49] Durfee, D.A. and Shoucair, F.S., "Comparison of Floating Gate Neural Network Memory Cells in Standard VLSI CMOS Technology", *IEEE Transactions on Neural Networks*, vol. 3, no. 3, pp. 347-353, May 1992
- [50] Dutt, N.D., Handley, T., and Gajski, D.D., "An Intermediate Representation for Behavioural Synthesis", *Proceedings of the 27th Design Automation Conference*, pp. 14-19, Orlando, Florida, June 1990

- [51] Dutta, S. and Skekar, S., "Bond Rating: A Non-Conservative Application of Neural Networks", *IEEE International Joint Conference on Neural Networks*, vol. II, pp. 443-450, 1988
- [52] Farhat, N.H., Psaltis, D., Prata, A., and Paek, E., "Optical Implementations of the Hopfield Model", *Applied Optics*, vol. 24, no. 10, pp. 1469-1475, 15 May 1985
- [53] Feldman, M.R. and Guest, C.C., "Computer Generated Holographic Optical Elements for Optical Interconnection of Very Large Scale Integrated Circuits", *Applied Optics*, vol. 26, no. 20, pp. 4377-4384, 15 October 1987
- [54] Forrest, J. and Edwards, M.D., "The Automatic Generation of Programmable Logic Arrays from Algorithmic State Machine Descriptions", *VLSI 83 Elsevier Science Publishers B. V. (North-Holland)*, pp. 183-193, 1983
- [55] Foster, C. and Iberall, T., *Computer Architecture*, Van Nostrand Reinhold Co., 3rd Edition, 1985
- [56] Fukushima, K., "A Neural Network for Visual Pattern Recognition", *IEEE Computer*, pp. 65-75, March 1988
- [57] Fukushima, K., "Neural Networks for Visual Pattern Recognition", *IEICE Transactions*, vol. E.74, no. 1, pp. 179-190, January 1991
- [58] Gajski, D.D. and Thomas, D.E., "Introduction to Silicon Compilation", in *Silicon Compilation*, ed. D.D. Gajski, Addison-Wesley Publishing Company, Inc., pp. 1-48, University of California at Irvine, 1988
- [59] Gajski, D.D., "Module Generation and Silicon Compilation", in *Physical Design Automation of VLSI Systems*, ed. B. Preas and M. Lorenzetti, The Benjamin/Cummings Publishing Company, Inc., pp. 283-345, 1988
- [60] Gebotys, C.H. and Elmasry, M.I., "A Global Optimization Approach for Architectural Synthesis", *Proceedings of the International Conference on Computer-Aided Design'90*, pp. 258-261, Santa Clara, CA, November 1990
- [61] Geus, A.J. de, "Logic Synthesis Speeds ASIC design", *IEEE Spectrum*, pp. 27-31, August 1989
- [62] Ghosh, S., "Using ADA as an HDL", *IEEE Design & Test*, pp. 30-42, February 1988
- [63] Graf, H.P. and deVegar, P., "A CMOS Implementation of a Neural Network Model", in *Advanced Research in VLSI*, ed. P. Losleben, *Proc. of the 1987 Stanford Conferece*, 1987
- [64] Graf, H.P., Hubbard, W., Jackel, L.D., and deVegvar, P.G.N., "A CMOS Associative Memory Chip", *Proc. IEEE First Int. Conf. on Neural Networks*, vol. III, pp. 461-468, June 1987
- [65] Graf, H.P., Jackel, L.D., and Hubbard, W.E., "VLSI Implementation of a Neural Network Model", *IEEE Computer*, pp. 41-49, March 1988
- [66] Graf, H.P., Jackel, L.D., Howard, R.E., Straughn, B., Denker, J.S., Hubbard, W., Tennant, D.M., and Schwartz, D., "VLSI Implementations of a Neural Network Memory with Several Hundreds of Neurons", *American Institute of Physics*, pp. 182-187, 1986

- [67] Graf, H.P., Sackinger, E., Boser, B., and Jackel, L.D., "Recent Developments of Electronic Neural Nets in the US and Canada", *Proceedings of the 2nd International Conference on Microelectronics for Neural Networks*, pp. 471-488, Munich, Germany, October 16-18, 1991
- [68] Greenwood, D., "An Overview of Neural Networks", *Behavioral Science*, vol. 36, pp. 1-33, 1991
- [69] Gutschow, T., "AXON: The Researchers Neural Network Language", *Proceedings of the international Neural Network Symposium INNS'88*, 13-17 September 1988
- [70] Gyrzyc, E.F., Buhr, R.J., and Knight, J.P., "Applicability of a Subset of ADA as an Algorithmic Hardware Description language for Graph-Based Hardware Compilation", *IEEE Transactions on CAD*, vol. CAD-4, no. 2, April 1985
- [71] Hafer, L.J. and Parker, A.C., "A Formal Method for the Specification, Analysis and Design of Register-Transfer Level Digital Logic", *IEEE Transactions on Computer-Aided Design*, vol. CAD-2, no. 1, pp. 4-18, January 1983
- [72] Hall, D.G., "Survey of Silicon-Based Integrated Optics", *IEEE Computer*, pp. 25-32, December 1987
- [73] Hall, D.G., "The Role of Silicon in Integrated Optics", *Optics News*, pp. 12-15, February 1988
- [74] Hamilton, A., Murray, A.F., Baxter, D.J., Churcher, S., Reekie, H.M., and Tarassenko, L., "Integrated Pulse Stream Neural Networks: Results, Issues, and Pointers", *IEEE Transactions on Neural Networks*, vol. 3, no. 3, pp. 385-393, May 1992
- [75] Hammerstrom, D., "A VLSI Architecture for High-Performance, Low-Cost, On-Chip Learning", *IEEE International Joint Conference on Neural Networks*, vol. II, pp. 537-544, San Diego, California, June 17-21, 1990
- [76] *Handbook of Neural Computing Applications*, ed. A.J. Maren, C.T. Harston, and R.M. Pap, Academic Press, Inc., 1990
- [77] Hayati, S. and Parker, A., "Automatic Production of Controller Specifications from Control and Timing Behavioral Descriptions", *26th IEEE/ACM Conference on Design Automation Conference*, pp. 75-80, 1989
- [78] Hebb, D., *The organization of Behaviour*, New York Wiley, 1949
- [79] Hecht-Nielsen, R., "Counterpropagation Networks", *IEEE First International Conference on Neural Networks*, vol. 2, pp. 19-32, 1987
- [80] Hecht-Nielsen, R., "Neurocomputing: Picking the Human Brain", *IEEE SPECTRUM*, vol. 25, no. 3, pp. 36-41., 1988
- [81] Hecht-Nielsen, R., "Performance Limits of Optical, Electro-Optical, and Electronic Neurocomputers", *Proc. SPIE Optical and Hybrid Computing SPIE*, vol. 634, pp. 277-306, 1986
- [82] Hecht-Nielsen, R., *Neurocomputing*, Addison-Wesley Publishing Company, 1990

- [83] Hirai, Y., "Hardware Implementatin of Neural Networks in Japan", *Proceedings of the 2nd International Conference on Microelectronics for Neural Networks*, pp. 435-446, Munich, Germany, October 16-18, 1991
- [84] Holler, M., Tam, S., Castro, H., and Benson, R., "An Electrically Trainable Artificial Neural Network (ETANN) with 10240 'Floating Gate' Synapses", Intel Corporation, Technology Development, *Proc. IEEE First International Joint Conference on Neural Networks - IJCNN 89*, vol. II, pp. 191-196, June 18-22 1989
- [85] Holt, J.L. and Hwang, J.N., "Finite Precision Error Analysis of Neural Network Hardware Implementations", Department of Electrical Engineering, University of Washington, *Internal Report, FT-10*, pp. 1-29, Seattle, March 1991
- [86] Hopfield, J.J., "Neural Networks and Physical Systems with Emergent Collective Computational Abilities", *Proc. Nat. Acad. Sci. USA*, vol. 79, pp. 2554-2558, April 1982
- [87] Howard, R.E., Schwartz, D.B., Denker, J.S., Epworth, R.W., Graf, H.P., Hubbard, W.E., Jackel, L.D., Straughn, B.L., and Tennant, D.M., "An Associative Memory Based on an Electronic Neural Network Architecture", *IEEE Transactions on Electron Devices*, vol. ED-34, no. 7, pp. 1553-1556, July 1987
- [88] Hubbard, W., Schwartz, D., Denker, J., Graf, H.P., Howard, R., Jackel, L., Straughn, B., and Tennant, D., "Electronic Neural Networks", *American Institute of Physics*, pp. 227-234, 1986
- [89] Hutcheson, L.D., "Integrated Optics: Evolution and Prospects", *Optics News*, p. 7, February 1988
- [90] Jackel, L.D., Graf, H.P., and Howard, R.E., "Electronic Neural Network Chips", *Applied Optics*, vol. 26, pp. 5077-5080, December 1987
- [91] Jelemensky, J. et al., "The MC68332 Microcontroller", *IEEE Micro*, vol. 9, no. 4, pp. 31-50, August 1989
- [92] Johannsen, D., "Bristle Blocks: A Silicon Compiler", *Proc. 16th Design Automation Conf.*, pp. 310-313, 1979
- [93] Jones, R.D., Lee, Y.C., Barnes, C.W., Flake, G.W., Lee, K., Lewis, P.S., and Qian, S., "Function Approximation and Time Series Prediction with Neural Networks", *Center for Non Linear Studies*, Los Alamos, 1989
- [94] Josin, G., Charney, D., and White, D., "Robot Control Using Neural Networks", *IEEE International Joint Conference on Neural Networks*, vol. II, pp. 625-631, 1988
- [95] Karatsu, O., "VLSI Design language Standardization Effort in Japan", *26th IEEE/ACM Conference on Design Automation Conference*, pp. 50-55, 1989
- [96] Kernighan, B.W. and Ritchie, D.M., *The C Programming Language*, Prentice Hall, 1978
- [97] Kim, J.J., Kurdahi, F.J., and Park, N., "Automatic Synthesis of Time-Stationary Controllers for Pipelined Data Paths", *IEEE Conference on Computer-Aided Design*, pp. 80-83, 1991

- [98] Kitayama, K. and Yoshinaga, H., "Experiments of Learning in Optical Perceptron-Like and Multilayer Neural Networks", *International Joint Conference on Neural Networks*, 1989
- [99] Kohonen, T., "The Self-Organizing Map", *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1464-1480, September 1990
- [100] Kowalski, T.J., "The VLSI Design Automation Assistant: An Architecture Compiler", in *Silicon Compilation*, ed. D.D. Gajski, Addison-Wesley Publishing Company, Inc., pp. 122- 152, University of California at Irvine, 1988
- [101] Krishnamoorthy, A.V., Yayla, G., and Esener, S.C., "A Scalable Optoelectronic Neural System Using Free-Space Optical Interconnects", *IEEE Transactions on Neural Networks*, vol. 3, no. 3, pp. 404-413, May 1992
- [102] Kuh, E.S. and Ohtsuki, T., "Recent Advances in VLSI Layout", *Proceedings of the IEEE*, vol. 78, no. 2, pp. 237-263, February 1990
- [103] Kyuma, K. et al., "Optical Neural Networks: System and Device Technologies", *SPIE - Optical Computing*, vol. 963, pp. 475-484, 1988
- [104] Lee, J., Hsu, Y., and Lin, Y., "A new Integer Linear Programming Formulation for the Scheduling Problem in Data-Path Synthesis", *Proceedings of the International Conference on Computer-Aided Design'89*, Santa Clara, CA, November 1989
- [105] Ling, E.R. Khan ., "Systolic Architectures for Artificial Neural Nets", *Proceedings of the IJCNN'91*, pp. 620-627, Singapore, November 18-21, 1991
- [106] Lippmann, R.P., "An Introduction to Computing with Neural Nets", *IEEE ASSP Magazine*, pp. 4-22, April 1987
- [107] Lippmann, R.P., "Review of Neural Networks for Speech Recognition", *Neural Computation*, vol. 1, pp. 1-38, 1989
- [108] Man, H. De, Rabaey, J., Meerbergen, J. van, and Huisken, J., "Silicon Compilation of DSP Systems with Cathedral II", *Proceedings of the 4th Annual ESPRIT Conference*, pp. 207-217, Brussels, September 28-29, 1987
- [109] Man, H. et al., "CATHEDRAL-II: A Silicon Compiler for Digital Signal Processing", *IEEE Design and Test*, pp. 13-25, December 1986
- [110] Marcade, E., Canut, F., Revault, N., and Moulinoux, C., "N: A Language Dedicated to Neural Algorithms Design", Thomson CSF/DSE, *Esprit II - Pygmalion 2059, Formal HLL Definition - Release 1 D-110-2*, October 8, 1990
- [111] Martinetz, T.M., Hitter, H.J., and Schulten, K.J., "3D Neural Net for learning Visuomotor coordination of a robot arm", *IEEE International Joint Conference on Neural Networks*, vol. II, pp. 351-356, 1989
- [112] Mauduit, N., Duranton, M., Gobert, J., and Sirat, J., "Lneuro 1.0: A Piece of Hardware LEGO for Building Neural Network Systems", *IEEE Transactions on Neural Networks*, vol. 3, no. 3, pp. 414-422, May 1992
- [113] McFarland, M.C., "The Value Trace: A Data Base for Automated Digital Design", *Department of Electrical Engineering Carnegie-Mellon University RC-01-04-80*, pp. 1-49, December 1978

- [114] McFarland, M.C., Parker, A.C., and Camposano, R., "The High-Level Synthesis of Digital Systems", *Proceedings of the IEEE*, vol. 78, no. 2, pp. 301-319, February 1990
- [115] Mead, C. and Conway, L., *Introduction to VLSI Systems*, Addison-Wesley Publishing Company, Inc., 1980
- [116] Mead, C., *Analog VLSI and Neural Systems*, Addison-Wesley Publishing Company, 1989
- [117] Meyer, E.L., "On Module Generation", *VLSI Systems Design*, pp. 48-63, March 1987
- [118] Micheli, D. De et al., "A Design System for PLA-Based Digital Circuits", *Advances in Computer-Aided Engineering Design*, vol. 1, pp. 285-364, 1985
- [119] Micheli, G. De and Ku, D., "HERCULES - A System for High-Level Synthesis", *Proceedings of the 25th ACM/IEEE Design Automation Conference, Anaheim*, pp. 483-488, 1988
- [120] Micheli, G. De, "High-Level Synthesis of Digital Circuits", *IEEE Design & Test of Computers*, pp. 6-7, October 1990
- [121] Micheli, G. De, Ku, D., Mailhot, F., and Truong, T., "The Olympus Synthesis System", *IEEE Design & Test of Computers*, pp. 37-53, October 1990
- [122] Minsky, M. and Papert, S., *Perceptrons an Introduction to Computational Theory*, MIT Press, Cambridge MA, 1969
- [123] Morton, S.G., "Electronic Hardware Implementations", in *Handbook of Neural Computing Applications*, ed. A.J. Maren, C.T. Harston, and R.M. Pap, Academic Press, Inc., pp. 251-269, 1990
- [124] Murray, A.F. and Smith, A.V.W., "Asynchronous VLSI Neural Networks using Pulse Stream Arithmetic", *IEEE Journal of Solid-State Circuits and Systems*, pp. 1-24, 1988
- [125] Murray, A.F., "Analogue Neural VLSI: Issues, Trends and Pulses", in *Artificial Neural Networks*, ed. I. Aleksander and J. Taylor, Elsevier Science Publishers B.V., vol. 2, pp. 35-43, 1992
- [126] Murray, A.F., "Pulse Arithmetic in VLSI Neural Networks", *IEEE Micro*, pp. 64-74, December 1989
- [127] Myers, D.J. and Hutchinson, R.A., "Efficient Implementation of Piecewise Linear Activation Function for Digital VLSI Neural Networks", *Electronics Letters*, vol. 25, no. 24, pp. 1662-1663, 23rd November 1989
- [128] Nagasamy, V., Berry, N., and Dangelo, C., "Specification, Planning, and Synthesis in a VHDL Design Environment", *IEEE Design & Test of Computers*, pp. 58-68, June 1992
- [129] Nigri, M., "Folding software: A Preliminary Analysis and Implementation", University College London, Galatea Internal Report REP-C21-M18, 1992
- [130] Nigri, M., "Hardware Emulation of Back-Propagation Neural Networks", *Research Note RN/91/21, Department of Computer Science, University College London*, pp. 1-12, February 1991

- [131] Nigri, M., "nC_code Machine independent Neural Network Specification Language", *Pygmalion Documentation*, April 1990
- [132] Nigri, M., Treleaven, P., and Vellasco, M., "Silicon Compilation of Neural Networks", *Proceedings of the IEEE CompEuro'91*, pp. 541-546, Bologna, Italy, May 13-16, 1991
- [133] Nigri, M.E., Rocha, P.V., and Treleaven, P., "An Integrated Neurocomputing System", *Proceedings of the IEEE/INNS IJCNN-91-SEATTLE*, pp. 547-552, July 8-12, 1991
- [134] Obrebska, M., Chuquillanqui, S., and Derantonian, H., "PLA and Custom Design", in *Design Methodologies*, ed. S. Goto, North-Holland, pp. 83-122, 1986
- [135] Ohta, J. et al., "Optical Implementation of an Associative Neural Network Model with a Stochastic Process", *Applied Optics*, vol. 28, no. 12, pp. 2426-2428, 15 June 1989
- [136] Orailoglu, A. and Gajski, D.D., "Flow Graph Representation", *23rd IEEE/ACM Conference on Design Automation Conference*, pp. 503-509, 1986
- [137] Ouali, J. and Saucier, G., "Silicon Compiler for neuro-ASICs", *IEEE International Joint Conference on Neural Networks*, vol. II, pp. 557-561, 1990
- [138] Ouali, J., Saucier, G., and Trille, J., "A Flexible, Universal Wafer Scale Neural Network", *Proceedings of the 3rd Workshop on Wafer Scale Integration*, Como, Italy, June 1989
- [139] Ousterhout, J.K., "The Magic VLSI Layout System", *IEEE Design & Test*, pp. 19-30, February 1985
- [140] Pacheco, M. and Treleaven, P., "A VLSI Word-Slice Architecture for Neurocomputing International Symposium on Computer Architecture and Digital Signal Processing", pp. 29-34, Hong Kong, October 1989
- [141] Palmer, R., "High Performance Digital Neural Network Implementation for Small-Scale Portable Applications", *Proceedings of the 2nd International Conference on Microelectronics for Neural Networks*, pp. 207-216, Munich, Germany, October 16-18, 1991
- [142] Pangrle, B.M. and Gajski, D.D., "Design Tools for Intelligent Silicon Compilation", *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, no. 6, pp. 1098-1111, November 1987
- [143] Papachristou, C.A. and Konuk, H., "A Linear Program Driven Scheduling and Allocation Method Followed by an Interconnect Optimisation Algorithm", *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 77-83, Orlando, Florida, June 24-28, 1990
- [144] Parker, A.C. et al., "Unified System Construction (USC)", in *High Level VLSI Synthesis*, Kluwer Academic Publishers, 1991
- [145] Parker, A.C., Pizarro, J., and Mlinar, M., "MAHA: A Program for Datapath Synthesis", *Proceedings of the 23rd Design Automation Conference*, pp. 461-466, 1986

- [146] Paulin, P.G. and Knight, J.P., "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's", *IEEE Transactions on Computer-Aided Design*, vol. 8, no. 6, pp. 661-679, June 1989
- [147] Paulin, P.G., Knight, J.P., and Girczyc, E.F., "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis", *23rd IEEE/ACM Conference on Design Automation Conference*, pp. 263-270, 1986
- [148] Peng, Z., "Synthesis of VLSI Systems with the CAMAD Design Aid", *23rd IEEE/ACM Conference on Design Automation Conference*, pp. 278-284, 1986
- [149] Poechmueller, W. and Glesner, M., "Evaluation of State-of-the-art Neural Network Customised Hardware", Elsevier, *Neurocomputing*, no. 2, pp. 209-231, 1991
- [150] Psaltis, D. and Farhat, N., "Optical Information Processing based on an Associative-Memory Model of Neural Nets with Thresholding and Feedback", *Optics Letters*, vol. 10, no. 2, February 1985
- [151] Psaltis, D., "Optical Realizations of Neural Network Models", *SPIE - International Optical Computing Conference*, vol. 700, pp. 278-282, 1986
- [152] Rabaey, J., "Silicon Compilation and Design Synthesis for Digital Systems", *Proceedings of the CERN European Organisation for Nuclear Research*, pp. 234-259, Oxford, United Kingdom, 15-26 August 1988
- [153] Ramacher, U., "SYNAPSE-X: A General-Purpose Neurocomputer Architecture", *Proceedings of the IJCNN'91*, pp. 2168-2176, Singapore, November 18-21, 1991
- [154] Recce, M., Rocha, P.V., and Treleaven, P.C., "Neural Network Programming Environments", in *Artificial Neural Networks*, ed. I. Aleksander and J. Taylor, Elsevier Science Publishers B.V., vol. 2, pp. 1237-1244, 1992
- [155] Rocha, P.V., "A Fully Integrated Neural Computing System", University of London, PhD Thesis, Department of Computer Science, University College London, July 1992
- [156] Roth, M.W., "Survey of Neural Network Technology for Automatic Target Recognition", *IEEE Transactions on Neural Networks*, vol. 1, no. 1, pp. 28-43, March 1990
- [157] Roy, J., Kumar, N., Dutta, R., and Vemuri, R., "DSS: A Distributed High-Level Synthesis System", *IEEE Design & Test of Computers*, pp. 18-32, June 1992
- [158] Rudell, R., Sangiovanni-Vincentelli, A.L., and Micheli, G. De, "A Finite State Machine Synthesis System", *Proc. ISCAS*, 1985
- [159] Rumelhart, D.E. and McClelland, J.L., *Parallel Distributed Processing - Explorations in the Microstructure of Cognition*, vol. 1 & 2, 1986
- [160] Rumelhart, D.E. and Zipser, D., "Feature Discovery by Competitive Learning", in *Parallel Distributed Processing*, vol. 1, pp. 151-193, 1986
- [161] Rumelhart, D.E., Hinton, G.E., and Williams, R.J., "Learning Internal Representations by Error Propagation", in *Parallel Distributed Processing*, vol. 1, pp. 318-362, 1986

- [162] Sangiovanni-Vincentelli, A., "Towards Automatic Synthesis and Verification of Complex Electronic Systems", *Proceedings of IEEE CompEuro'91*, pp. 888-893, Bologna, Italy, May 13-16, 1991
- [163] Saucier, G., Paulet, M. De, and Sicard, P., "ASYL: A Rule-Based System for Controller Synthesis", *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, no. 6, pp. 1088-1097, November 1987
- [164] Sedgewick, R., *Algorithms*, Addison-Wesley Publishing Company, 1988
- [165] Sejnowski, T.J. and Rosenberg, C.R., "Parallel Networks that learns to Pronounce English Text", *Complex Systems*, vol. 1, pp. 145-168, 1987
- [166] Shahdad, M., "An Interface between VHDL and EDIF", *24th IEEE/ACM Conference on Design Automation Conference*, pp. 472-478, 1987
- [167] Shepherd, G., "Synaptic Organisation of the Brain", *John Wiley & Sons*
- [168] Sivilotti, M.A., Emerling, M.R., and Mead, C.A., "VLSI Architectures for Implementation of Neural Networks", *Proceedings of the American Institute of Physics Conference*, pp. 408-413, 1986
- [169] Sluss, J.J. Jr., Veasey, D.L., Batchman, T.E., and Parrish, E.A. Jr., "An Introduction to Integrated Optics for Computing", *IEEE Computer*, pp. 9-23, December 1987
- [170] Smith, S.D., "Optical Bistability, Photonic Logic, and Optical Computation", *Applied Optics*, vol. 25, no. 10, pp. 1550-1564, 15 May 1986
- [171] Southard, J.R., "MacPitts: An Approach to Silicon Compilation", *Computer*, vol. 16, no. 12, pp. 74-82, December 1983
- [172] Stallman, R.M., *Using and Porting GNU CC - Version 1.39*, Free Software Foundation, Inc., January 1989
- [173] Stoll, A. and Duzy, P., "High-Level Synthesis from VHDL with Exact Timing Constraints", *29th ACM/IEEE Conference on Design Automation Conference*, pp. 188-193, 1992
- [174] Szu, H.H., "Optical Neuro-Computing", in *Handbook of Neural Computing Applications*, ed. A.J. Maren, C.T. Harston, and R.M. Pap, Academic Press, Inc., pp. 271-293, 1990
- [175] Tanaka, T., Kobayashi, T., and Karatsu, O., "HARP: Fortran to Silicon", *IEEE Transactions on Computer-Aided Design*, vol. 8, no. 6, pp. 649-660, June 1989
- [176] Thomas, D.E., "Chapter 13 - Automatic Data Path Synthesis", in *Design Methodologies*, ed. S. Goto, North-Holland, pp. 401-439, 1986
- [177] Thomas, D.E., Lagnese, E.D., Walker, R.A., Nestor, J.A., Rajan, J.V., and Blackburn, R.L., *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*, Kluwer, 1990
- [178] Thomsen, A. and Brooke, M.A., "A Low-cost Application-specific Neural Network Implementation with Floating Gate Weights", *Proceedings of the IJCNN'92*, vol. II, pp. 565-570, Baltimore, Maryland, June 7-11, 1992

- [179] Tomlinson, M.S. Jr., Walker, D.J., and Sivilotti, M.A., "A Digital Network Architecture for VLSI", *International Joint Conference on Neural Networks*, vol. II, pp. 545-550, 1990
- [180] Treleaven, P., Pacheco, M., and Vellasco, M., "VLSI Architectures for Neural Networks", *IEEE Micro*, pp. 8-27, December 1989
- [181] Treleaven, P.C., "PYGMALION Neural Network Programming Environment", *Proceeding of the International Conference on Artificial Neural Networks (ICANN-91)*, pp. 569-578, Espoo, Finland, 24-28 June, 1991
- [182] Treleaven, P.C., "Parallel Architectures for Neurocomputers", *European Seminar on Neural Computing*, February 1988
- [183] Treleaven, P.C., Recce, M., and Wang, C., "Neural Network Programming Environments: a Review", *The 4th European Seminar on Neural Computing - Putting Neural Nets to Work*, London Marriott Hotel, February 21-22, 1991
- [184] Trickey, H., "Flamel: A High-Level Hardware Compiler", *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, no. 2, pp. 259-269, March 1987
- [185] Tseng, C. and Siewiorek, D.P., "Automated Synthesis of Data Path in Digital Systems", *IEEE Transactions on Computer-Aided Design*, vol. CAD-5, no. 3, pp. 379-395, July 1986
- [186] Tseng, C. et al., "Bridge: A Versatile Behavioral Synthesis System", *25th ACM/IEEE Design Automation Conference*, pp. 415-420, 1988
- [187] Turnbull, K.W., "An Overview of High Level Synthesis Technologies for Digital ASICs", in *Algorithmic and Knowledge Based CAD for VLSI*, ed. G. Taylor and G. Russel, pp. 47-75, 1992
- [188] Vanhoof, J., "Architecture Synthesis for Application-Specific Medium-Throughput Digital Signal-Processing Chips", Katholieke Universiteit Leuven, PhD Thesis, Departement Elektrotechniek, Fakulteit Toegepaste Wetenschappen, February 1992
- [189] Vellasco, M. and Treleaven, P., "A Neurocomputer Exploiting Silicon Compilation", *Proc. Neural Computing Meeting, The Institute of Physics*, London, April 1989
- [190] Vellasco, M., "High-Level Silicon Compiler", University College London, Galatea Internal Report REP-C21-M12, 1991
- [191] Vellasco, M., "The nC Neural Network Programming Language - Manual (Version 1.02)", University College London, *Pygmalion Project 2059*, 1990
- [192] Verleysen, M. and Jespers, P.G.A., "An Analog VLSI Implementation of Hopfield's Neural Network", *IEEE Micro*, pp. 46-55, December 1989
- [193] Verleysen, M., Sirletti, B., and Jespers, P., "A New VLSI Architecture for Neural Associative Memories", *Proceedings of the nEuro'88*, pp. 692-700, Paris, France, June 6-9, 1988

- [194] Verleysen, M., Sirletti, B., Vandemeulebroecke, A.M., and Jespers, P.G.A., "Neural Networks for High-Storage Content-Addressable Memory: VLSI Circuit and Learning Algorithm", *IEEE Journal of Solid-State Circuits*, vol. 24, no. 3, pp. 562-569, June 1989
- [195] Wagner, K. and Psaltis, D., "Multilayer optical learning networks", *Applied Optics*, vol. 26, no. 23, pp. 5061-5076, 1 December 1987
- [196] Waibel, A., "Consonant Recognition by Modular Construction of Large Phonemic Time-Delay Neural Networks", in *Neural Information Processing Systems*, Kaufman Publishers, pp. 215-223, 1988
- [197] Waibel, A., Hanazawa, T., Hinton, G., Schikano, K., and Lang, K., "Phoneme Recognition Using Time-Delay Neural Networks", *IEEE Transactions on Acoustics, Speech, and Signal Processing (ASSP)*, no. 37, March 1989
- [198] Walker, R.A. and Camposano, R., *A Survey of High-level Synthesis Systems*, Kluwer Academic Publishers, 1991
- [199] Wasserman, P.D., *Neural Computing Theory and Practice*, Van Nostrand Reinhold, 1989
- [200] Webb, A.R., "Applications of Neural Networks in Military Systems", *Military Microwaves'90*, pp. 356-361, London, July 11-13, 1990
- [201] White, H., "Economic Prediction Using Neural Networks: The Case of IBM Daily Stock Returns", *IEEE International Joint Conference on Neural Networks*, vol. II, pp. 451-458, 1988
- [202] Widrow, B. and Lehr, M.A., "30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation", *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1415-1442, September 1990
- [203] Yatagai, T., "Optical Computing in Japan", *Future Generation Computer Systems*, vol. 4, pp. 177-187, 1988

Appendix A

Published Work

- [1] Alippi, C. and Nigri, M.E., "Hardware Requirements for Digital VLSI Implementation of Neural Networks", in *Proc. IJCNN'91 Singapore, November 18-21 1991*, IEEE, pp. 1873-1878, New York, N.Y., 1991
- [2] Nigri, M., Treleaven, P., and Vellasco, M., "Silicon Compilation of Neural Networks", *Proceedings of the IEEE CompEuro'91*, pp. 541-546, Bologna, Italy, May 13-16, 1991
- [3] Nigri, M.E., Rocha, P.V., and Treleaven, P., "An Integrated Neurocomputing System", *Proceedings of the IEEE/INNS IJCNN-91-SEATTLE*, pp. 547-552, July 8-12, 1991
- [4] Nigri, M.E. and Treleaven, P.C., "High Level Synthesis of Neural Network Chips", (To be published) *the International Workshop on Artificial Neural Networks*, Sitges (Barcelona), Spain, June 9-11, 1993

Appendix B

nC Language Syntax and Example

This appendix presents the syntax of the nC language. This syntax definition is used to implement the parser of the NSC, upon which the compilation process starts. In addition, the implementation of the Back Propagation network in nC is given, which is used throughout this dissertation to assess its execution in hardware.

nC Syntax Definition

The following syntax definition is used for parsing a *nC* rule. The parser is called after the entire *nC* system data structure is read. Then, for each neural algorithm rule defined by the user, the *NSC* parses these rules according to the syntax presented below.

```
rules
    func_def_list
func_def_list
    oldc
oldc
    oldc_func_def
    oldc_oldc_func_def
oldc_func_def
    one_of_types func_def
    func_def
func_def
    function_name PAREN_OPEN arg_list
arg_list
    oldc_arg_list_opt PAREN_CLOSE
oldc_arg_list_opt
    identifier
    oldc_arg_list_opt COMMA identifier
oldc_arg_decl
    typed_spec var_name SEMI_COLON
    oldc_arg_decl typed_spec var_name SEMI_COLON
typed_spec
    type_specifier
var_name
    declarator
```

```
compound_stat
    CBRK_OPEN decl_list_opt stat_list_opt CBRK_CLOSE

decl_list_opt
    decl_list

decl_list
    declaration
    decl_list declaration

declaration
    typed_spec init_decl_list SEMI_COLON

init_decl_list
    init_declarator
    init_decl_list COMMA init_declarator

init_declarator
    var_name
    var_name EQUAL initializer

declarator
    dir_declarator
    STAR dir_declarator
    STAR STAR dir_declarator

dir_declarator
    identifier
    PAREN_OPEN declarator PAREN_CLOSE
    dir_declarator BRK_OPEN int_const_opt BRK_CLOSE
    dir_declarator PAREN_OPEN PAREN_CLOSE

initializer
    additive_expr
    CBRK_OPEN initializer_list CBRK_CLOSE

initializer_list
    initializer
    initializer_list COMMA initializer

stat_list_opt
    stat_list

stat_list
    statement
    stat_list statement

statement
    expr_statement
    selection_stat
    iteration_stat
    jump_stat
    control_stat

expr_statement
    SEMI_COLON
    assign_expr SEMI_COLON

expression
    assign_expr

assign_expr
    logic_or_expr
    unary_expr EQUAL logic_or_expr
    unary_expr ASSIGN_OPERATORS logic_or_expr

logic_or_expr
    logic_and_expr
    logic_or_expr LOGIC_OR logic_and_expr

logic_and_expr
    incl_or_expr
    logic_and_expr LOGIC_AND incl_or_expr

incl_or_expr
    excl_or_expr
    incl_or_expr OPERATOR_OR excl_or_expr
```

```

excl_or_expr
    and_expression
    excl_or_expr OPERATOR_XOR and_expression

and_expression
    equality_expr
    and_expression AMPERSAND equality_expr

equality_expr
    relational_expr
    equality_expr EQUAL_COMPARE relational_expr

relational_expr
    shift_expr
    relational_expr LOGIC_LESS shift_expr
    relational_expr LOGIC_GREATER shift_expr
    relational_expr LOGIC_LESS_EQUAL shift_expr
    relational_expr LOGIC_GREATER_EQUAL shift_expr

shift_expr
    additive_expr
    shift_expr LEFT_SHIFT additive_expr
    shift_expr RIGHT_SHIFT additive_expr

additive_expr
    multipl_expr
    additive_expr PLUS multipl_expr
    additive_expr MINUS multipl_expr

multipl_expr
    cast_expr
    multipl_expr STAR cast_expr
    multipl_expr DIVIDE cast_expr
    multipl_expr MOD cast_expr

cast_expr
    unary_expr
    PAREN_OPEN type_name PAREN_CLOSE cast_expr

unary_expr
    postfix_expr
    STAR cast_expr UNARY
    unary_operator cast_expr UNARY
    SIZEOF unary_expr

unary_operator
    PLUS
    MINUS
    PLUS_PLUS
    MINUS_MINUS
    AMPERSAND
    NOT
    TILDA

postfix_expr
    primary_expr
    postfix_expr BRK_OPEN additive_expr BRK_CLOSE
    postfix_expr PAREN_OPEN arg_expr_list PAREN_CLOSE
    postfix_expr PAREN_OPEN PAREN_CLOSE
    postfix_expr DOT identifier
    postfix_expr POINTER identifier
    postfix_expr PLUS_PLUS
    postfix_expr MINUS_MINUS

primary_expr
    any_constant1
    AMPER_IDENT
    PAREN_OPEN expression PAREN_CLOSE

arg_expr_list
    assign_expr
    arg_expr_list COMMA assign_expr

type_name
    typed_spec abstr_decl_opt

abstr_decl_opt
    abstract_decl

abstract_decl
    pointer

```

```

    pointer dir_abstr_decl
    dir_abstr_decl

double_star
    STAR STAR

pointer
    STAR
    double_star

dir_abstr_decl
    PAREN_OPEN abstr_decl_opt PAREN_CLOSE
    dir_abstr_decl BRK_OPEN const_expr_opt BRK_CLOSE
    BRK_OPEN const_expr_opt BRK_CLOSE
    dir_abstr_decl PAREN_OPEN PAREN_CLOSE

const_expr_opt
    constant_expr

constant_expr
    additive_expr

selection_stat
    IF PAREN_OPEN cond_expression PAREN_CLOSE body_statement IF
    IF PAREN_OPEN cond_expression PAREN_CLOSE body_statement ELSE body_statement

iteration_stat
    for_loop

for_loop
    FOR PAREN_OPEN opt_assign_expr SEMI_COLON opt_cond_expr SEMI_COLON opt_assign_expr
    PAREN_CLOSE body_statement

body_statement
    CBRK_OPEN stat_list CBRK_CLOSE
    statement

opt_assign_expr
    assign_expr

opt_cond_expr
    cond_expression

cond_expression
    logic_or_expr

opt_ret_expr
    expression

jump_stat
    RETURN opt_ret_expr SEMI_COLON
    CONTINUE SEMI_COLON
    BREAK SEMI_COLON

control_stat
    PAR_CONTROL CBRK_OPEN stat_list CBRK_CLOSE
    PAR_CONTROL for_loop

any_constant1
    int_constant
    FLOAT_CONSTANT1
    FLOAT_CONSTANT2
    STRING
    CHAR_CONSTANT

one_of_types
    TYPESPEC
    CADDR_T
    TAGVAL

one_of_strucs
    TYPENAME

int_constant
    INT_CONSTANT
    HEX_CONSTANT
    identifier

int_const_opt
    int_constant

```

```

identifier
    UPPER_IDENT
    LOWER_IDENT
    PARA_LIST
    SYS

function_name
    UPPER_IDENT

type_specifier
    one_of_strucs
    one_of_types

```

Back Propagation Network

The following code is the *nC* specification of the Back Propagation neural network used throughout this thesis. The first part of this specification shows the functions that are required to program an algorithm in *nC*. The second part shows the definition of a typical application.

```

/* ----- */
/* Function bodies for the back-propagation rule classes - */
/* ----- */
/* Silicon Compilation Version - Meyer 20/06/91 */
/* ----- */

/* ----- */
float Learn()
{
    EXEC(&sys->net[c_net]->rules[ NET_R_learn ]);
    return(sys->net[c_net]->parameters[ NET_P_score ].parameter.value.UNION_MEMBER);
}

/* ----- */
float Learn_o()
{
    EXEC(&sys->net[c_net]->rules[ NET_R_learn_o ]);
    return(sys->net[c_net]->parameters[ NET_P_score ].parameter.value.UNION_MEMBER);
}

/* ----- */
float Step_learn()
{
    EXEC(&sys->net[c_net]->rules[ NET_R_step_learn ]);
    return(sys->net[c_net]->parameters[ NET_P_score ].parameter.value.UNION_MEMBER);
}

/* ----- */
float Step_learn_o()
{
    EXEC(&sys->net[c_net]->rules[ NET_R_step_learn_o ]);
    return(sys->net[c_net]->parameters[ NET_P_score ].parameter.value.UNION_MEMBER);
}

/* ----- */
float Recall()
{
    EXEC(&sys->net[c_net]->rules[ NET_R_recall ]);
    EXEC(&sys->net[c_net]->rules[ NET_R_tol_test ]);
    return(sys->net[c_net]->parameters[ NET_P_score ].parameter.value.UNION_MEMBER);
}

```

```

/* ----- */
int State_update (p)
TAGVAL **p;      /* [*state, *acc, *SIZE, *statel, *weightl, ... ] */
{
    p[ 1 ]->value.UNION_MEMBER = dp( p + 2 );
    p[0]->value.UNION_MEMBER = lookuptbl (p[1]->value.UNION_MEMBER);
    return( 0 );
}

/* ----- */
int State_update_output (p)
TAGVAL **p;      /* [*state, *acc, *SIZE, *statel, *weightl, ... ] */
{
    p[ 1 ]->value.UNION_MEMBER = dp( p + 2 );
    if ( threshold ) {
        p[ 0 ]->value.UNION_MEMBER = lookuptbl (p[1]->value.UNION_MEMBER);
    }
    else {
        p[ 0 ]->value.UNION_MEMBER = p[ 1 ]->value.UNION_MEMBER; /*no squashing*/
    }
    return( 0 );
}

/* ----- */
int Err_cal_output (p)
TAGVAL **p;      /* [*err, *target, *state] */
{
    if ( threshold ) {
        p[ 0 ]->value.UNION_MEMBER = (p[ 1 ]->value.UNION_MEMBER -
                                       p[ 2 ]->value.UNION_MEMBER) *
                                       (1 - p[ 2 ]->value.UNION_MEMBER) *
                                       p[ 2 ]->value.UNION_MEMBER;
    }
    else {
        p[ 0 ]->value.UNION_MEMBER = p[ 1 ]->value.UNION_MEMBER -
                                       p[ 2 ]->value.UNION_MEMBER;
    }
    return( 0 );
}

/* ----- */
/* err_cal_hidden_class */
/* Meyer 8/11/90 - For the Generic Neuron Architecture */

int Err_cal_hidden (p)
TAGVAL **p;      /* [*err, *state, *SIZE, *errorl, *weightl, ... ] */
{
    p[0]->value.UNION_MEMBER = dp(p+2) * (1 - p[1]->value.UNION_MEMBER) *
                                p[1]->value.UNION_MEMBER;
    return( 0 );
}

/* ----- */
/* weight_update_class */
/* Meyer 8/11/90 - For the Generic Neuron Architecture */

int Weight_update (p)
TAGVAL **p;      /* [*error, *learn_rate, *SIZE, *weight0, *state0, ... ] */
{
    register      int      i, size, patterns;
    register      DATA_TYPE f;

    size = *(int *)p[ 2 ];

    PAR for (i = 3; i < ( 2 * size + 3); i += 2) {
        p[i]->value.UNION_MEMBER += p[0]->value.UNION_MEMBER *
                                    p[1]->value.UNION_MEMBER *
                                    p[i+1]->value.UNION_MEMBER;
    }
}

/* ----- */
int Tolerance (p)
TAGVAL **p;      /* [*tolerance, *net.score, *net.measure, *SIZE, ... ] */
{
    register      int      size, control;
    register      DATA_TYPE tol, *score;
    tol = (*p++)->value.UNION_MEMBER;
    score = &(*p++)->value.UNION_MEMBER;
}

```

```

    control = (int) (*p++)->value.UNION_MEMBER;

    if ( control == MERR ) {
        *score = max_err_cal(p);
#ifdef DEBUG_BP
        printf ( "Tolerance test MERR : score %f tolerance %f\n", *score, tol );
#endif
        if ( *score < tol ) {
            return( TERM );
        }
    }
    else if ( control == HAM ) {
        size = *(int *)p;
        *score = ham_dis_cal(p) * adjust / (DATA_TYPE) size;
#ifdef DEBUG_BP
        printf ( "Tolerance test HAM : score %f tolerance %f\n", *score, tol );
#endif
        if ( *score < tol ) {
            return( TERM );
        }
    }
    else if ( control == EUCL ) {
        size = *(int *)p;
        *score = eucl_dis_cal(p) * adjust / (DATA_TYPE) size;
#ifdef DEBUG_BP
        printf ( "Tolerance test EUCL : score %f tolerance %f\n", *score, tol );
#endif
        if ( *score < tol ) {
            return( TERM );
        }
    }
    else if ( control == ANGL ) {
        *score = angl_cal(p);
#ifdef DEBUG_BP
        printf ( "Tolerance test ANGL : score %f tolerance %f\n", *score, tol );
#endif
        if ( *score < tol ) {
            return( TERM );
        }
    }
    else {
        error("Illegal tolerance test control [%d]\n", control);
        return( NOTOK );
    }
    return( OK );
}

/* ----- */
/*      Main body for the back-propagation rule classes      */
/* ----- */
/*      Silicon Compilation Version - Meyer 20/06/91          */
/* ----- */
#ifdef OMIT_MAIN
main (argc, argv)
int    argc;
char   *argv[];
{
    int          i, cycle_limit, n_fail, max_i;
    int          wh[4];
    DATA_TYPE   e, max_e, sum_e;
    pat_elem     *p;
    int          load_flags;
    int          pid, nolearn = FALSE, noshow = FALSE, show_save = FALSE;
    int          save_freq = 10;          /* save every 10 cycles */
    int          tolerance_rate = 0;      /* default : no automatic decrease */
    int          tolerance_limit = 0;     /* default : no lower limit */
    char         *c;
    FILE         *fd;
    char         message_file[ FILESIZE ];
    int          loop_count = 0;
    int          pat_select;
    DATA_TYPE   learn_rate_low = LEARN_RATE;
    DATA_TYPE   learn_rate_high = (DATA_TYPE) 0;
    int          first_message = TRUE;

    printf ("\nSIMPLE BP ALGORITHM\n");
/* ----- */
#ifdef D_T == D_INT

```



```

if ( getenv("BP_GENERIC_NEURON") ) {
    generic_neuron = TRUE;
    printf ("GENERIC_NEURON EMULATION\n");
}
else
    printf ("IDEAL FIXED-POINT EMULATION\n");
if ( getenv("BP_USE_THR_TBL") ) {
    use_thr_tbl = TRUE;
    printf ("THRESHOLD TABLE ACTIVATED\n");
}
else
    printf ("IDEAL SIGMOID FUNCTION\n");
if ( c = getenv("BP_INT_PART") ) {
    int_part = atoi( c );
}
else {
    printf ("Enter number of Integer bits -> ");
    scanf ("%d", &int_part);
}
if ( c = getenv("BP_FRAC_PART") ) {
    frac_part = atoi( c );
}
else {
    printf ("Enter number of fractionary bits -> ");
    scanf ("%d", &frac_part);
}
printf ( "BP_INT_PART set : %d\n", int_part);
printf ( "BP_FRAC_PART set : %d\n", frac_part);

precision = frac_part + int_part;
adjust = (long) pow (2.0, (float) frac_part); /* Meyer 19/11/90 */
signal_bit = (int) pow (2.0, (float) int_part + frac_part - 1); /* Meyer 19/11/90 */
max_value = (int) pow (2.0, (float) int_part + frac_part); /* Meyer 27/04/91 */

int_mult = 2 * int_part;
frac_mult = (2 * frac_part) - precision;
adjust_mult = (int) pow (2.0, (float) precision - int_mult);

if (use_thr_tbl) {
    if ( c = getenv("BP_INT_INP_TBL") ) {
        int_input_tbl = atoi( c );
    }
    if ( c = getenv("BP_FRAC_INP_TBL") ) {
        frac_input_tbl = atoi( c );
    }
    printf ( "BP_INT_INP_TBL set : %d\n", int_input_tbl);
    printf ( "BP_FRAC_INP_TBL set : %d\n", frac_input_tbl);
}

#if VERSION == 0
    if (generic_neuron) {
        if (frac_input_tbl > frac_mult) {
            printf ("Error: frac_mult must be greater than frac_input_tbl\n");
            exit (1);
        }
        if (int_input_tbl > int_mult) {
            printf ("Error: int_mult must be greater than int_input_tbl\n");
            exit (1);
        }
    }
    else {
        if (frac_input_tbl > frac_part*2) {
            printf ("Error: frac_part must be greater than frac_input_tbl\n");
            exit (1);
        }
        if (int_input_tbl > int_part*2) {
            printf ("Error: int_part must be greater than int_input_tbl\n");
            exit (1);
        }
    }
}
#endif

size_thr_tbl = (int) pow (2.0, (float) frac_input_tbl + int_input_tbl);
thr_tbl_gen(); /* Initialise threshold table */
/* for (i=0; i<size_thr_tbl; i++)
printf ("%d\n", thr_tbl[i]); */
printf ("Threshold Table size = %d\n", size_thr_tbl);
}
#endif

/*-----*/
tolerance = TOLERANCE * adjust;

```

```

learn_rate = LEARN_RATE * adjust;
input_scale = INPUT_SCALE * adjust;
target_scale = TARGET_SCALE * adjust;
/*-----*/

measure = MEASURE;

if ( argc > 2 ) {
    printf ( "\nBack propagation settings : \n\n");
    if ( getenv("BP_THRESHOLD") ) {
        threshold = TRUE;
    }
    if ( getenv("BP_LINEAR") ) {
        threshold = FALSE;
    }
    if ( threshold ) {
        printf ( "THRESHOLD function on output layer\n");
    }
    else {
        printf ( "LINEAR function on output layer\n");
    }
    if ( c = getenv("BP_LEARN") ) {
        if ( ! lexequ ( c, "Learn" ) ) {
            learn_flag = NET_R_learn;
            printf ( "learn rule : Learn\n");
        }
        else if ( ! lexequ ( c, "Step_learn" ) ) {
            learn_flag = NET_R_step_learn;
            printf ( "learn rule : Step_learn\n");
        }
        else if ( ! lexequ ( c, "Learn_o" ) ) {
            learn_flag = NET_R_learn_o;
            printf ( "learn rule : Learn_o\n");
        }
        else if ( ! lexequ ( c, "Step_learn_o" ) ) {
            learn_flag = NET_R_step_learn_o;
            printf ( "learn rule : Step_learn_o\n");
        }
        else {
            printf ( "Invalid value for environment variable LEARN : %s\n", c );
            exit ( 1 );
        }
    }
    else {
        learn_flag = NET_R_step_learn; /* default rule is Step_learn() */
        printf ( "default learn rule : Step_learn\n");
    }
    if ( c = getenv("BP_CYCLE_LIMIT") ) {
        cycle_limit = atoi( c );
        printf ( "BP_CYCLE_LIMIT set : %d\n", cycle_limit);
    }
    else {
        printf ( "no cycle limit\n" );
        cycle_limit = -1;
    }
    if ( c = getenv("BP_SAVE_FREQ") ) {
        save_freq = atoi( c );
        printf ( "BP_SAVE_FREQ set : saving every %d cycles\n", save_freq);
    }
    else {
        printf ( "saving every %d cycles\n", save_freq );
    }
    if ( input_pattern_mask = getenv("BP_INPUT_MASK") ) {
        printf ( "BP_INPUT_MASK : %s\n", input_pattern_mask);
    }
    if ( target_pattern_mask = getenv("BP_TARGET_MASK") ) {
        printf ( "BP_TARGET_MASK : %s\n", target_pattern_mask);
    }
    if ( getenv("BP_NOLEARN") ) {
        nolearn = TRUE;
        printf ( "BP_NOLEARN set : recall only\n");
    }
    if ( getenv("BP_NOSHOW") ) {
        noshow = TRUE;
        printf ( "BP_NOSHOW set : states will not be displayed\n");
    }
    pat_select = FALSE;
    if ( getenv("BP_PAT_SELECT") ) {
        pat_select = TRUE;
        printf ( "BP_PAT_SELECT set : learning for selected patterns\n");
    }

```

```

    }
    if ( c = getenv("BP_SHOW_SAVE") ) {
        show_save = TRUE;
        printf ( "BP_SHOW_SAVE set\n" );
    }
    if ( c = getenv("BP_TOL_RATE") ) {
        tolerance_rate = atoi( c );
        printf ( "BP_TOL_RATE set : tolerance will automatically decrease by %d
%%\n", tolerance_rate );
    }

    if ( c = getenv("BP_TOL_LIMIT") ) {
#if D_T == D_INT
        tolerance_limit = float2fix (atof( c ));
        printf ( "BP_TOL_LIMIT set : termination after learning to %g\n", fix2float
(tolerance_limit ));
#else
        tolerance_limit = ( float ) atof( c );
        printf ( "BP_TOL_LIMIT set : termination after learning to %g\n",
            tolerance_limit );
#endif
    }

    if ( c = getenv("BP_TOLERANCE") ) {
#if D_T == D_INT
        tolerance = float2fix (atof( c ));
        printf ( "BP_TOLERANCE set : %g\n", fix2float (tolerance ));
#else
        tolerance = ( float ) atof( c );
        printf ( "BP_TOLERANCE set : %g\n", tolerance );
#endif
    }
    else {
#if D_T == D_INT
        printf ( "default tolerance : %g\n", fix2float (tolerance ));
#else
        printf ( "default tolerance : %g\n", tolerance );
#endif
    }

    if ( c = getenv("BP_LEARN_RATE") ) {
#if D_T == D_INT
        learn_rate = float2fix (atof( c ));
        printf ( "BP_LEARN_RATE set : %g\n", fix2float (learn_rate ));
#else
        learn_rate = ( float ) atof( c );
        printf ( "BP_LEARN_RATE set : %g\n", learn_rate );
#endif
    }
    else {
#if D_T == D_INT
        printf ( "default learn_rate : %g\n", fix2float (learn_rate ));
#else
        printf ( "default learn_rate : %g\n", learn_rate );
#endif
    }

    if ( c = getenv("BP_LEARN_HIGH") ) {
#if D_T == D_INT
        learn_rate_high = float2fix (atof( c ));
        printf ( "BP_LEARN_HIGH set : %g\n", fix2float (learn_rate_high ));
#else
        learn_rate_high = ( float ) atof( c );
        printf ( "BP_LEARN_HIGH set : %g\n", learn_rate_high );
#endif
    }

    if ( c = getenv("BP_MEASURE") ) {
#if D_T == D_INT
        measure = float2fix (atof( c ));
        printf ( "BP_MEASURE set : %g\n", fix2float (measure ));
#else
        measure = ( float ) atof( c );
        printf ( "BP_MEASURE set : %g\n", measure );
#endif
    }
    else {
#if D_T == D_INT
        printf ( "default measure : %g\n", fix2float (measure ));
#else
        printf ( "default measure : %g\n", measure );
#endif
    }

```

```

    }
    printf ( "default: weight update for each pattern\n");
    if ( ! getenv("BP_GO") ) {
        printf ( "\nsettings correct ( y/n ) ? ");
        if ( ! YES() ) {
            exit ( 1 );
        }
    }
    printf ( "\n");
}
random(0xd5a4793c); /* Initialize drand48 */
srand48(random());

/* Construct the network */

if ( rc_read () ) {
    printf ( "Problem with .pgmrc file\n");
    exit ( 1 );
}
printf ( "local_user %s\n", rc [ RC_local_user ] );
printf ( "local_host %s\n", rc [ RC_local_host ] );

#ifdef CODE_GENERATED /* Meyer 12/3/91 - This is compiled only under bp.c and not under
bp_nc_CODE.c which is generated by code_gen() */
    if (argc == 1) {
        if (system_connect(&system_config)) /* use internal configuration defaults */
            exit(FAIL);
        if (build_rules())
            exit(FAIL);
    }
    else {
#endif
        if (argc > 1)
            strcpy( system_filename, argv[ 1 ] );
#ifdef CODE_GENERATED /* Meyer 12/3/91 - This is compiled only under bp.c */
        /* load system file - if configuration not specified,
        connect() and build_rules() must be called here */
        if ( ! ( load_flags = sys_load( system_filename ) & LD_CONFIG ) ) {
            if (system_connect(&system_config))/* use internal configuration defaults */
                exit(FAIL);
            if (build_rules())
                exit(FAIL);
        }
    }
#endif
    /* 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.UNION_MEMBER;
    target_pattern_control = (int) sys->net[c_net]->parameters[ NET_P_target_control
].parameter.value.UNION_MEMBER;
    sys->net[c_net]->parameters[ NET_P_measure ].parameter.value.UNION_MEMBER =
measure;
    sys->net[c_net]->parameters[ NET_P_tolerance ].parameter.value.UNION_MEMBER =
tolerance;
    sys->net[c_net]->parameters[ NET_P_learn_rate ].parameter.value.UNION_MEMBER =
learn_rate;
    learn_rate_low = sys->net[c_net]->parameters[ NET_P_learn_rate
].parameter.value.UNION_MEMBER;
    if ( learn_rate_high == (DATA_TYPE) 0 ) {
        learn_rate_high = learn_rate_low;
    }

    /* Initialise patterns specified on command line */

    if ( argc > 2 ) {
        i = init_patterns ( argv [ 2 ] );
        printf ( "init_patterns returns %d from %s\n", i, argv [ 2 ] );
    }

    if ( pattern_list == NULL ) /* new system - randomise the weights */
        rand_weight();

/* ----- Meyer 14/11/90 Convert all synapse weights to integer -----*/
#ifdef D_T == D_INT
    { int l, c, n, s;

```

```

        for (l=1; l<sys->net[0]->layers; l++)
        for (c=0; c<sys->net[0]->layer[l]->clusters; c++)
        for (n=0; n<sys->net[0]->layer[l]->cluster[c]->neurons; n++)
        for (s=0; s<sys->net[0]->layer[l]->cluster[c]->neuron[n]->synapses; s++){
            sys->net[0]->layer[l]->cluster[c]->neuron[n]->synapse[s]-
>weight.value.UNION_MEMBER = float2fix (sys->net[0]->layer[l]->cluster[c]->neuron[n]-
>synapse[s]->weight.value.f); /* 12/3/91 UNION_MEMBER */
        }
    }
/*      exit (1); */
#endif
    } /* end if (argc == 1) */
/* ----- Meyer 14/11/90 End of conversion -----*/

    if ( argc == 1 || nolearn ) {
        printf ( "\nskipping learn phase\n");
    }
    else {
        while ( cycle_limit-- ) {

/*****
*                               Test phase                               *
*****/
            n_fail = 0;
            max_e = (DATA_TYPE) 0;
            sum_e = (DATA_TYPE) 0;
            tolerance = sys->net[c_net]->parameters[ NET_P_tolerance
].parameter.value.UNION_MEMBER;
            for ( pattern_num = 0; pattern_num < pattern_count; pattern_num++ ) {
                current_pattern = indextop ( pattern_num );
                read_input      ( pattern_num );
                scale_input     ( sys->net[c_net]->parameters[ NET_P_input_scale
].parameter.value.UNION_MEMBER);
                read_target     ( pattern_num );
                scale_target    ( sys->net[c_net]->parameters[ NET_P_target_scale
].parameter.value.UNION_MEMBER);
                result = Recall ();
                current_pattern->score = result;
                if ( result > tolerance ) {
                    n_fail++;
                    e = result - tolerance;
                    sum_e += e;
                    if ( e > max_e ) {
                        max_e = e;
                        max_i = pattern_num;
                    }
                }
            }
            if ( ! n_fail && tolerance_rate ) {
                if ( tolerance_limit != (DATA_TYPE) 0 &&
                    tolerance < tolerance_limit ) {
#if D_T == D_INT
                    printf("\nTolerance limit %f reached\n\n",fix2float(tolerance_limit
));
#else
                    printf( "\nTolerance limit %f reached\n\n", tolerance_limit );
#endif
                    break;
                }
            }
            if D_T == D_INT
                tolerance= (tolerance * ((1*adjust - ( tolerance_rate ) / 100) * adjust))
/ adjust;
            else
                tolerance*= 1.0 - ( (float) tolerance_rate ) / 100.0;
            endif
            sys->net[c_net]->parameters[NET_P_tolerance].parameter.value.UNION_MEMBER
                = tolerance;
            if D_T == D_INT
                printf( "\nResetting TOLERANCE to %f\n\n", fix2float (tolerance) );
            else
                printf( "\nResetting TOLERANCE to %f\n\n", tolerance );
            endif
        }

/*****
*                               Progress message                               *
*****/
            printf ( "cycle [%4d] ", cycle_count );
            switch ( learn_flag ) {
                case NET_R_learn:

```

```

        case NET_R_learn_o:

            printf ( "loops [%6.2f] ",(float) ( loop_count )/pattern_count);
            break;

        case NET_R_step_learn:
        case NET_R_step_learn_o:
            break;
    }
    if ( pat_select ) {
#if D_T == D_INT
        printf ("learn rate [ %5.3f ] ",
                fix2float(sys->net[c_net]->parameters[ NET_P_learn_rate
].parameter.value.UNION_MEMBER));
#else
        printf("learn rate [ %5.3f ] ",
                sys->net[c_net]->parameters[NET_P_learn_rate].parameter.value.f );
#endif
    }
    if ( n_fail ) {
        printf ( "%d of %d fail ( %7.5f ) : ave [ %f ] max [ %d ] by %f\n",
                n_fail,
                pattern_count,
#if D_T == D_INT
                fix2float (tolerance),
#else
                tolerance,
#endif
                fix2float ( sum_e / n_fail),
                ( sum_e/(float) n_fail),
                max_i,
                fix2float (max_e)
#if D_T == D_INT
                max_e
#endif
            );
    }
    else {
        printf ( "\n" );
    }
    /* ----- */
    if ( first_message ) {
        printf("\n");
        first_message = FALSE;
    }
    else {
        if ( ! ( (cycle_count) % save_freq ) ) {
            printf("\n");
            /* (void) set_interrupts(); */
            sys_save( system_filename );
            /* (void) clear_interrupts(); */
        }
    }
    if ( last_interrupt ) {
        printf ( "\ninterrupt - exiting\n");
        break;
    }
    if ( ! n_fail ) { /* none fail - exit */
        break;
    }
    if ( n_fail && pat_select ) {
#if D_T == D_INT
        sys->net[c_net]->parameters[ NET_P_learn_rate
].parameter.value.UNION_MEMBER = learn_rate_low + ( 1*adjust - ((DATA_TYPE) n_fail /
(DATA_TYPE) pattern_count) ) * (learn_rate_high - learn_rate_low) / adjust;
#else
        sys->net[c_net]->parameters[ NET_P_learn_rate
].parameter.value.UNION_MEMBER = learn_rate_low + ( 1 - ((DATA_TYPE) n_fail / (DATA_TYPE)
pattern_count) ) * (learn_rate_high - learn_rate_low);
#endif
    }
    #ifdef DEBUG_BP
        printf( "Resetting LEARN_RATE to %f\n", sys->net[c_net]->parameters[
NET_P_learn_rate ].parameter.value.f);
    #endif
}

```

```

/*****
*          Learn phase
*****/

    loop_count = 0;          /* clear loop counter for sexec_r LEARN rules */
    for ( pattern_num = 0; pattern_num < pattern_count; pattern_num++ ) {
        current_pattern = indextop ( pattern_num );
        if ( pat_select && ( current_pattern->score != (DATA_TYPE) 0 ) &&
            ( current_pattern->score < sys->net[c_net]->parameters[
NET_P_tolerance ].parameter.value.UNION_MEMBER ) ) {
            continue;
        }
        read_input      ( pattern_num );
        scale_input      ( sys->net[c_net]->parameters[ NET_P_input_scale
].parameter.value.UNION_MEMBER);
        read_target      ( pattern_num );
        scale_target      ( sys->net[c_net]->parameters[ NET_P_target_scale
].parameter.value.UNION_MEMBER);
        switch ( learn_flag ) {
            case NET_R_learn:
                result = Learn();
                loop_count += get_loop ( &sys->net[c_net]->rules[ NET_R_learn ] );
                break;
            case NET_R_learn_o:
                result = Learn_o();
                loop_count += get_loop(&sys->net[c_net]->rules[ NET_R_learn_o ] );
                break;
            case NET_R_step_learn:
                result = Step_learn();
                break;
            case NET_R_step_learn_o:
                result = Step_learn_o();
                break;
        }
    } /* end for */
    cycle_count++;
} /* end while ( cycle_limit ) */
printf ( "\nterminated at cycle [ %d ]\n", cycle_count);
} /* end if ( nolearn ) */
if ( argc > 3 ) { /* may specify second test set in argv[ 3 ] */
    i = init_patterns ( argv [ 3 ] );
    printf ( "init_patterns returns %d from %s\n", i, argv [ 3 ] );
}
if ( system_filename[0] == '\0' ) {
    sys_save ( "test" );
}
else {
    sys_save( system_filename );
}
#endif
code_gen(0,"");/* generate debugging rule analysis IFF environment variable
PGMREPORT set */
if (getenv ("CODE_GEN")) {
    printf ("\nGenerating bp_nc_CODE.c ...\n");
    code_gen(1, "bp"); /* generate bp_nc_CODE.c */
}
if (getenv ("ICR_GEN")) { /* Meyer 05/06/91 */
    printf ("\nGenerating Intermediate Code Representation for Silicon
Compilation...\n");
    gen_icr("bp.icr"); /* generate Intermediate Code Representation */
    printf ("Done!\n\n");
}
if ( ! noshow ) {
    for ( pattern_num = 0; pattern_num < pattern_count; pattern_num++ ) {

/*****
*          Recall phase
*****/

```

```

*****/
    current_pattern = indextop ( pattern_num );
    read_input      ( pattern_num );
    scale_input     ( sys->net[c_net]->parameters[ NET_P_input_scale
].parameter.value.UNION_MEMBER );
    read_target     ( pattern_num );
    scale_target    ( sys->net[c_net]->parameters[ NET_P_target_scale
].parameter.value.UNION_MEMBER);
    Recall();
    shownet();
    if ( show_save ) {
        sys_save( system_filename );
    }
}
}
rc_write ();
#if D_T == D_INT
    if (argc > 1) {
        printf ("Total number of Underflows under recalling = %d\n", underflow);
        printf ("Total number of Overflows under recalling = %d\n", overflow);
        underflow = overflow = 0L;
        printf ("Total number of up-saturation under recalling = %d\n", highsaturate);
        printf ("Total number of low-saturation under recalling = %d\n", lowsaturate);
        lowsaturate = highsaturate = 0L;
    }
#endif
}
#endif
/* ----- */
void scale_input( factor )
DATA_TYPE factor;
{
    #if D_T == D_INT
        if ( factor != (DATA_TYPE) 1*adjust) {
    #else
        if ( factor != (DATA_TYPE) 1) {
    #endif
        int i, input_size = sys->net[c_net]->fanin;
        DATA_TYPE **f, **port = (DATA_TYPE **) sys->net[c_net]->input_port;
        for ( f = port, i=0; i< input_size; i++, f++ ) {
    #if D_T == D_INT
            **f *= factor / adjust;
    #else
            **f *= factor;
    #endif
        }
    }
}
/* ----- */
void scale_target( factor )
DATA_TYPE factor;
{
    #if D_T == D_INT
        if ( factor != (DATA_TYPE) 1*adjust ) {
    #else
        if ( factor != (DATA_TYPE) 1 ) {
    #endif
        int i, target_size = sys->net[c_net]->fanout;
        DATA_TYPE **f, **port = (DATA_TYPE **) sys->net[c_net]->target;
        for ( f = port, i=0; i< target_size; i++, f++ ) {
    #if D_T == D_INT
            **f *= factor / adjust;
    #else
            **f *= factor;
    #endif
        }
    }
}
/* ----- */

```


Appendix C

Hardware Simulation Results

This Appendix reports in depth the results obtained during the simulation of the Back Propagation neural network under the hardware constraints discussed in the chapter 5.

Overview

The simulations of neural networks' hardware carried out in this work, and presented in chapter 5, are directed to the analysis of hardware constraints during the neural computation. Several issues are considered, which have produced a large amount of data during the simulation of the Back Propagation neural network for the Optical Character Recognition (OCR) application. This large amount of data is fully presented in this appendix through several tables, which are organised in a way to provide a clear understanding about the obtained results. Additionally, to allow a clear analysis of each issue and its own influence on the hardware execution of the algorithm, the remaining sections of this appendix are organised according to the context by which the investigations are developed.

The tables are organised as follows:

- **Data Precision** — determines the number of bits used for data (it is assumed that every data has the same precision);
- **Table Index** — specifies how the result of the propagation rule is fed into table to obtain the neuron's output;
- **Back Propagation Parameters** — determine each parameter employed in the Back Propagation neural model, in which, η is the learning rule, ϵ is the error associated with the learning algorithm, and μ is the *momentum* term;
- **Results** — give the final assessment of the hardware performance by checking whether learning and recall have been successfully achieved, and the number of cycles required to train the network; and

- **Data Saturation** — assesses how the lack of enough number of bits in the integer part of the data affects the results. Saturations occur when the result of the propagation rule ($\Sigma s.w$) is greater than the maximum value permitted. In this case, the result is saturated to the maximum (or minimum) value.

Three special symbols are used in the tables to describe special situations, either for the simulation inputs or the simulation results:

- ✓ means that the current option applies (for example, that the *Generic Neuron* architecture is fully simulated);
- ✱ means that either the options is not valid (for example, when the *momentum* term is not used, this symbol is inserted in the appropriate column to indicate that the *simple* version of the Back Propagation model is enforced), or the result of the simulation indicates impossibility of being achieved (for example, when no learning is possible, this symbol is used to indicate that after the number of cycles specified, the network was unable to learn the patterns for the given tolerance ϵ); and
- ☒ indicates that although the network has not been successfully trained, the recall phase has been possible for all characters except one or two, which are explained by special comments.

Analysis of Data Precision

Data precision is analysed by fixing the number of bits in the integer part and varying the number of bits in the decimal part, keeping all other parameters unchanged. Table C.1 shows the results for the data precision varying from 7 to 14 bits, and fixing the number of bits for the integer part to 4. Lookup table is not employed, so that the analyses can be focused on the data precision.

Data Precision		Table Index		Back Prop. Parameters			Archi- tecture	Simulation Results			Errors during $\Sigma s.w$ due to data limited precision (Learn / Recall)	
Int	Dec	Int	Dec	η	ϵ	μ	Gen Neu	Learn	Recall	Cycles	Up-Saturation	Low-Saturation
4	7	*	*	.3	.25	*	✓	*	✓	800	0 / 0	0 / 0
4	8	*	*	.3	.25	*	✓	✓	✓	196	0 / 0	0 / 0
4	9	*	*	.3	.25	*	✓	✓	✓	197	0 / 0	0 / 0
4	10	*	*	.3	.25	*	✓	✓	✓	269	0 / 0	0 / 0
4	11	*	*	.3	.25	*	✓	✓	✓	192	0 / 0	0 / 0
4	12	*	*	3	.25	*	✓	✓	✓	686	2,512 / 3	3,161 / 5
4	13	*	*	3	.25	*	✓	✓	✓	348	75 / 1	2,566 / 12
4	14	*	*	3	.25	*	✓	*	☒ ¹	800	2,656 / 3	7,314 / 9

Table C.1 — Data Precision Influence without Employing Lookup Table.

¹Pattern 7 failed by just one neuron (pixel).

These results show that a minimum of 8 bits is required to achieve learning. Note that with 7 bits, the network is said to have failed to perform learning. However, even after failing to learn the ten patterns, the network has successfully recalled all patterns. This means that the learning was not able to satisfy the tolerance parameter, but after 800 cycles, the weights had been adjusted to a point that allowed the use of the network. It is clearly expected that the recall phase can be used with lower precision. This is discussed later in this appendix.

Other interesting effects showed by Table C.1 is that there is a minimum precision value and beyond that, learning is no longer improved. In this case, 8 bits are enough to guarantee perfect functioning of the network. The employment of greater precision required more cycles to achieve convergence. This may be attributed to the fact that very fine steps were taken during the learning phase. In addition, the saturation mechanism started to be employed, which clearly diverted the convergence trajectory of the algorithm.

Data Precision		Table Index		Back Prop. Parameters			Architecture	Simulation Results			Errors during $\Sigma s.w$ due to data limited precision (Learn / Recall)	
Int	Dec	Int	Dec	η	ϵ	μ	Gen Neu	Learn	Recall	Cycles	Up-Saturation	Low-Saturation
4	7	4	4	.3	.25	*	✓	*	✓	800	0 / 0	0 / 0
4	8	4	4	.3	.25	*	✓	✓	✓	162	0 / 0	0 / 0
4	9	4	4	.3	.25	*	✓	✓	✓	186	0 / 0	0 / 0
4	10	4	4	.3	.25	*	✓	✓	✓	210	0 / 0	0 / 0
4	11	4	4	.3	.25	*	✓	✓	✓	199	0 / 0	0 / 0
4	12	4	4	3	.25	*	✓	✓	✓	456	537 / 1	869 / 4
4	14	4	4	.3	.25	*	✓	✓	✓	224	0 / 0	144 / 2

Table C.2 — Data Precision Influence Employing Lookup Table

Table C.2 shows the same analysis, but employing the lookup table mechanism. It may be observed that little effect was caused by the use of such a technique for realising the activation function in hardware. The learning algorithm was badly affected when the saturation mechanism started to be employed. Interesting to note is that for 14-bit precision, only Low-Saturation has occurred, which represents a better result if compared with 12-bit precision. This suggests that in the first case, the saturation mechanism has benefited the algorithm, reducing the number of cycles to obtain convergence, while in the second case, possible saturations have been prematurely performed.

Influence of the Table's Size

Table C.3 presents the analysis of the influence caused by the Lookup Table mechanism. It can be seen that little effect has been caused by varying the size of the table. Table C.1 showed that for the same data representation, 196 cycles were required to achieve learning without the use of a lookup table. The results below show that the use of a lookup table caused loss of precision in realising the activation function. As a

consequence, some steps were skipped, and therefore, fewer cycles were necessary to obtain convergence.

Data Precision		Table Index		Back Prop. Parameters			Architecture	Simulation Results			Errors during $\Sigma s.w$ due to data limited precision (Learn / Recall)	
Int	Dec	Int	Dec	η	ε	μ	Gen Neu	Learn	Recall	Cycles	Up-Saturation	Low-Saturation
4	8	4	2	.3	.25	*	✓	✓	✓	168	0/0	0/0
4	8	4	3	.3	.25	*	✓	✓	✓	140	0/0	0/0
4	8	4	4	.3	.25	*	✓	✓	✓	162	0/0	0/0
4	8	4	5	.3	.25	*	✓	✓	✓	163	0/0	0/0
4	8	4	6	.3	.25	*	✓	✓	✓	167	0/0	0/0
4	8	4	7	.3	.25	*	✓	✓	✓	191	0/0	0/0
4	8	4	8	.3	.25	*	✓	✓	✓	158	0/0	0/0

Table C.3 — Impact Caused by the Employment of the Lookup Table (12 bits data)

Table C.4 shows the same result observed before. Fewer cycles were necessary to achieve learning. As an exception, when the saturation mechanism was used, the learning trajectory was diverted, causing an increase in the number of cycles necessary to train the network.

Data Precision		Table Index		Back Prop. Parameters			Architecture	Simulation Results			Errors during $\Sigma s.w$ due to data limited precision (Learn / Recall)	
Int	Dec	Int	Dec	η	ε	μ	Gen Neu	Learn	Recall	Cycles	Up-Saturation	Low-Saturation
4	10	4	4	.3	.25	*	✓	✓	✓	210	0/0	0/0
4	10	4	5	.3	.25	*	✓	✓	✓	229	0/0	0/0
4	10	4	6	.3	.25	*	✓	✓	✓	235	0/0	40/1
4	10	4	7	.3	.25	*	✓	✓	✓	274	284/1	756/4
4	10	4	8	.3	.25	*	✓	✓	✓	223	0/0	0/0
4	10	4	9	.3	.25	*	✓	✓	✓	235	0/0	0/0
4	10	4	10	.3	.25	*	✓	*	☒ ²	800	0/0	2,114/2

Table C.4 — Impact Caused by the Employment of the Lookup Table (14 bits data)

The same effect is observed in Table C.5, where greater precision is used. Comparisons with the result obtained in Table C.1 shows that fewer cycles were needed to obtain the learning convergence.

Data Precision		Table Index		Back Prop. Parameters			Architecture	Simulation Results			Errors during $\Sigma s.w$ due to data limited precision (Learn / Recall)	
Int	Dec	Int	Dec	η	ε	μ	Gen Neu	Learn	Recall	Cycles	Up-Saturation	Low-Saturation
4	12	4	4	.1	.2	*	✓	✓	✓	529	0/0	0/0
4	12	4	5	.1	.2	*	✓	✓	✓	517	0/0	0/0
4	12	4	6	.1	.2	*	✓	✓	✓	529	0/0	0/0
4	12	4	7	.1	.2	*	✓	✓	✓	575	0/0	0/0
4	12	4	8	.1	.2	*	✓	✓	✓	617	0/0	0/0
4	12	4	9	.1	.2	*	✓	✓	✓	507	0/0	0/0
4	12	4	10	.1	.2	*	✓	✓	✓	534	0/0	0/0
4	12	4	11	.1	.2	*	✓	✓	✓	524	0/0	0/0
4	12	4	12	.1	.2	*	✓	✓	✓	576	0/0	0/0

Table C.5 — Data Precision Influence Employing Lookup Table (16 bits Data)

²Pattern 7 failed by just one neuron (pixel).

Table C.6 shows the results for the expanded version of the Back Propagation algorithm, in which a *momentum* term is introduced. According to the computation involved in this version [158], more multiplications are employed, and therefore the need for greater precision should be expected. Nevertheless, for the precision used in the Table C.6, the same influence of the lookup table shown before is observed.

Data Precision		Table Index		Back Prop. Parameters			Archi- tecture	Simulation Results				Errors during $\Sigma s.w$ due to data limited precision (Learn / Recall)	
Int	Dec	Int	Dec	η	ε	μ	Gen Neu	Learn	Recall	Cycles		Up-Saturation	Low-Saturation
4	8	4	2	.3	.25	.5	✓	✓	✓	93		0 / 0	0 / 0
4	8	4	3	.3	.25	.5	✓	✓	✓	106		0 / 0	0 / 0
4	8	4	4	.3	.25	.5	✓	✓	✓	83		0 / 0	0 / 0
4	8	4	5	.3	.25	.5	✓	✓	✓	78		0 / 0	0 / 0
4	8	4	6	.3	.25	.5	✓	✓	✓	92		0 / 0	0 / 0
4	8	4	7	.3	.25	.5	✓	✓	✓	127		0 / 0	0 / 0
4	8	4	8	.3	.25	.5	✓	✓	✓	87		0 / 0	0 / 0

Table C.6 — Data Precision Influence Employing Lookup Table (with μ term)

Complex Computation

In this section the *simple* version of the Back Propagation model is compared with the more *complex* version, which includes the *momentum* term. Since the latter involves a considerable greater amount of multiplications than the former, it is interesting to analyse the influence that this term has in hardware.

Table C.7 is organised as pair of rows, which have the same input parameters, except that the first line has no *momentum* term and the second line uses $\mu = 0.5$.

The results given by Table C.7 show that for low precision data the *simple* version has better performance than the *complex* version, while for higher precision, the effect of the *momentum* term is effectively felt. This follows the theoretical predictions [158], in which the momentum term is added to speed up the convergence of the Back Propagation algorithm. However, since this involves a more intensive computation than the *simple* version, greater precision was also required.

Data Precision		Table Index		Back Prop. Parameters			Architecture	Simulation Results			Errors during $\Sigma s.w$ due to data limited precision (Learn / Recall)	
Int	Dec	Int	Dec	η	ϵ	μ	Gen Neu	Learn	Recall	Cycles	Up-Saturation	Low-Saturation
3	8	3	4	.3	.25	*	✓	✓	✓	173	5,776 / 38	7,098 / 42
3	8	3	4	.3	.25	.5	✓	*	*	800	4,159,477 / 3,549	4,470,957 / 3,729
3	9	3	4	.3	.25	*	✓	✓	✓	276	80,361 / 229	74,348 / 261
3	9	3	4	.3	.25	.5	✓	*	*	800	5,600,199 / 4,596	5,023,500 / 3,768
3	10	3	4	.3	.25	*	✓	*	✓	800	652,891 / 1,074	918,221 / 1,350
3	10	3	4	.3	.25	.5	✓	*	*	800	7,502,939 / 5,429	12,741,356 / 6,180
3	12	3	4	.3	.25	*	✓	*	✗ ³	800	2,325,481 / 2,687	2,211,466 / 2,639
3	12	3	4	.3	.25	.5	✓	*	*	800	7,763,730 / 5,186	10,997,677 / 5,056
3	14	3	4	.3	.25	*	✓	*	*	800	3,074,253 / 2,850	3,511,016 / 3,791
3	14	3	4	.3	.25	.5	✓	*	*	800	4,331,969 / 2,332	15,394,788 / 11,462
4	9	4	4	.3	.25	*	✓	✓	✓	186	0 / 0	0 / 0
4	9	4	4	.3	.25	.5	✓	*	✗ ⁴	800	0 / 0	0 / 0
4	11	4	4	.3	.25	*	✓	✓	✓	199	0 / 0	0 / 0
4	11	4	4	.3	.25	.5	✓	✓	✓	77	0 / 0	0 / 0
4	12	4	4	.3	.25	*	✓	✓	✓	456	537 / 1	869 / 4
4	12	4	4	.3	.25	.5	✓	✓	✓	113	622 / 2	23 / 0
4	14	4	4	.3	.25	*	✓	✓	✓	224	0 / 0	144 / 2
4	14	4	4	.3	.25	.5	✓	✓	✓	105	250 / 3	462 / 6
5	13	5	4	.3	.25	*	✓	✓	✓	246	0 / 0	0 / 0
5	13	5	4	.3	.25	.5	✓	✓	✓	79	0 / 0	0 / 0
6	10	6	4	.3	.25	*	✓	✓	✓	210	0 / 0	0 / 0
6	10	6	4	.3	.25	.5	✓	✓	✓	90	0 / 0	0 / 0
6	13	6	4	.3	.25	*	✓	✓	✓	246	0 / 0	0 / 0
6	13	6	4	.3	.25	.5	✓	✓	✓	79	0 / 0	0 / 0
6	14	6	4	.3	.25	*	✓	✓	✓	224	0 / 0	0 / 0
6	14	6	4	.3	.25	.5	✓	✓	✓	105	0 / 0	0 / 0

Table C.7 — Simple vs. Complex Back Propagation Computation

Effects of the Back Propagation Parameters

This section discusses how the parameters ϵ and η influence the hardware performance, in order to verify the theoretical studies held in chapter 5. Table C.8 shows the results obtained for this study for a variety of configurations.

It can be noted that the choice of these parameters is important and has a direct effect on the behaviour of the learning trajectory. These results are in accordance with the theoretical prediction given in chapter 5, except when data saturation is employed, since this has not been considered by the theoretical approach.

³Pattern 4 failed by just one neuron (pixel).

⁴Pattern 7 failed by just one neuron (pixel).

Data Precision		Table Index		Back Prop. Parameters			Architecture	Simulation Results			Errors during $\Sigma s.w$ due to data limited precision (Learn / Recall)	
Int	Dec	Int	Dec	η	ε	μ	Gen Neu	Learn	Recall	Cycles	Up-Saturation	Down-Saturation
3	10	3	7	.1	.1	*	✓	*	☑	800	562,363 / 1,086	202,010 / 379
3	10	3	7	.1	.2	*	✓	*	☑	2,000	5,755,089 / 3,087	2,466,505 / 1,468
3	10	3	7	.5	.2	*	✓	✓	✓	400	588,728 / 935	438,149 / 1,035
4	10	4	6	.1	.1	*	✓	*	✓	2,000	0 / 0	0 / 0
4	10	4	6	.5	.1	*	✓	✓	✓	370	1,270 / 4	3,796 / 10
4	10	4	6	.1	.2	*	✓	✓	✓	945	0 / 0	0 / 0
4	10	4	6	.5	.2	*	✓	✓	✓	200	150 / 2	1,000 / 5
5	10	5	5	.1	.1	*	✓	*	✓	2,000	0 / 0	0 / 0
5	10	5	5	.5	.1	*	✓	✓	✓	307	0 / 0	0 / 0
5	10	5	5	.1	.2	*	✓	✓	✓	820	0 / 0	0 / 0
5	10	5	5	.5	.2	*	✓	✓	✓	142	0 / 0	0 / 0
5	13	5	5	.1	.1	*	✓	✓	✓	1,309	0 / 0	0 / 0
5	13	5	5	.5	.1	*	✓	✓	✓	851	0 / 0	0 / 0
5	13	5	5	.1	.2	*	✓	✓	✓	500	0 / 0	0 / 0
5	13	5	5	.5	.2	*	✓	✓	✓	454	0 / 0	0 / 0
6	9	6	4	.1	.1	*	✓	*	✓	2,000	0 / 0	0 / 0
6	9	6	4	.5	.1	*	✓	✓	✓	316	0 / 0	0 / 0
6	9	6	4	.1	.2	*	✓	*	✓	800	0 / 0	0 / 0
6	9	6	4	.5	.2	*	✓	✓	✓	120	0 / 0	0 / 0
6	10	6	4	.1	.1	*	✓	*	✓	2,000	0 / 0	0 / 0
6	10	6	4	.5	.1	*	✓	✓	✓	506	0 / 0	0 / 0
6	10	6	4	.1	.2	*	✓	✓	✓	1,096	0 / 0	0 / 0
6	10	6	4	.5	.2	*	✓	✓	✓	189	0 / 0	0 / 0
6	11	6	4	.1	.1	*	✓	*	✓	2,000	0 / 0	0 / 0
6	11	6	4	.5	.1	*	✓	✓	✓	396	0 / 0	0 / 0
6	11	6	4	.1	.2	*	✓	✓	✓	666	0 / 0	0 / 0
6	11	6	4	.5	.2	*	✓	✓	✓	260	0 / 0	0 / 0

Table C.8 — Effects of the Back Propagation Parameters

Recall Phase

This section analyses the precision requirements for the recall phase. It is assumed the weights were adjusted using high precision during the learning phase. It can be seen from that at least 4 bits are required for the integer part and decimal part. This result confirms initial predictions, in which a network executing only the recall phase can be implemented with fewer bits than required by the learning phase.

Int	Dec	Int	Dec	Gen Neu	Recall	Up-Saturation	Down-Saturation
2	8	2	4	✓	*	1,573	4188
3	8	3	4	✓	✓	484	384
4	8	3	4	✓	✓	3	3
4	7	4	4	✓	✓	3	3
4	6	4	4	✓	✓	3	3
4	5	4	4	✓	✓	2	2
4	4	4	4	✓	✓	0	0
4	3	4	4	✓	*	0	0

Table C.9 — Effects of the Back Propagation Parameters

Summary

The results given in this appendix for the simulation of the Back Propagation neural network under several hardware constraints lead to some general conclusions:

- There is a minimum number of bits in the integer part of the data from which no saturation mechanism is needed — the results showed that at least 4 bits are needed;
- There is a minimum number of bits in the decimal part of the data from which enough precision is given to correctly employ the computation — the result showed that at least 8 bits are needed;
- The algorithm-dependent parameters have a direct impact on the required data representation and precision;
- The results are completely dependent on the algorithm itself, as shown by differences between *simple* and *complex* Back Propagation; this results in further simulations if the neural model is modified;
- Implementation of the activation function through a lookup table is feasible and little impact is felt on the results; even very small tables were reported to work well — the results indicate that the size should be at least 64.
- Learning requires more precision than recall;

More importantly, these simulation results show clearly the feasibility of using fixed-point computation with limited precision. This has a direct impact on the effectiveness of neural networks' hardware implementations. Furthermore, the parameters (both hardware-dependent and algorithmic-dependent) can be tuned to the particular application, thus producing cost-effective neural chips.

Appendix D

VHDL Description of Processing Elements

This appendix describes the design of the processing element in VHDL for a Back Propagation neural network. This includes the design of the three basic modules defined in the Generic Neuron model: memory unit, communication unit, and execution unit.

Definition of Registers

The use of multi-port registers is extremely important for the *Generic Neuron* architecture, since a particular variable can be read and written simultaneously in the same cycle through the busses it is connected. Below are the VHDL entities for several configurations of multi-port registers. These registers are used in the execution unit, as part of its data path. The VHDL ARCHITECTURE for these registers is described at the structural domain.

```
-- +-----+
-- | File Name       : 0xlreg.vhd
-- | Author          : Meyer E. Nigri
-- | Date of Creation : 05/12/91
-- | Last Update     : 31/12/91
-- +-----+
-- | Description:
-- | -----
-- | This entity implements the reg0xl block.
-- |
-- +-----+

LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY reg0xl IS
  PORT ( a : OUT mvl_vector (data_bus_lines DOWNT0 1));
END reg0xl;

ARCHITECTURE structure OF reg0xl IS
  COMPONENT latch_db
    PORT ( d : IN mvl_vector(data_bus_lines DOWNT0 1);
          q : OUT mvl_vector(data_bus_lines DOWNT0 1);
          ck : IN BIT
        );
  END COMPONENT;

  SIGNAL value : mvl_vector(data_bus_lines DOWNT0 1) := eta;
  SIGNAL vcc : BIT := '1';

BEGIN
  sw_input_a: latch_db PORT MAP ( d => value,
                                  q => a,
                                  ck => vcc
                                );

END structure;
```

```
-- +-----+
-- | File Name       : 1xlreg.vhd
-- | Author          : Meyer E. Nigri
-- | Date of Creation : 05/12/91
-- | Last Update     : 31/12/91
-- +-----+
```

```
-- | Description:
-- | -----
-- | This entity implements the reg1xl block.
-- |
-- +-----+

LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY reg1xl IS
  PORT ( a : IN mvl_vector (data_bus_lines DOWNT0 1);
        ctr_a_wr : IN BIT;
        b : OUT mvl_vector (data_bus_lines DOWNT0 1);
        ctr_b_rd : IN BIT;
        ck : IN BIT
      );
END reg1xl;

ARCHITECTURE structure OF reg1xl IS
  COMPONENT ff_n
    PORT ( d : IN mvl_vector(data_bus_lines DOWNT0 1);
          q : OUT mvl_vector(data_bus_lines DOWNT0 1);
          ck : IN BIT
        );
  END COMPONENT;

  COMPONENT latch_db
    PORT ( d : IN mvl_vector(data_bus_lines DOWNT0 1);
          q : OUT mvl_vector(data_bus_lines DOWNT0 1);
          ck : IN BIT
        );
  END COMPONENT;

  COMPONENT tri_state_data_bus
    PORT ( inp_3_sta : IN mvl_vector(data_bus_lines DOWNT0 1);
          enable : IN BIT;
          out_3_sta : OUT mvl_vector(data_bus_lines DOWNT0 1)
        );
  END COMPONENT;

  SIGNAL x, y : mvl_vector(data_bus_lines DOWNT0 1);

BEGIN
  reg: ff_n PORT MAP ( d => x,
                      q => y,
                      ck => ck
                    );

  sw_input_a: latch_db PORT MAP ( d => a,
                                  q => x,
                                  ck => ctr_a_wr
                                );
```

```

sw_output_b: tri_state_data_bus PORT MAP ( inp_3_sta => y,
                                             enable => ctr_b_rd,
                                             out_3_sta => b
                                           );

END structure;

-- File Name      : lx2reg.vhd
-- Author         : Meyer E. Nigri
-- Date of Creation : 05/12/91
-- Last Update    : 31/12/91
-- Description:
-- =====
-- This entity implements the reglx2 block.
--

LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY reglx2 IS
  PORT ( a      : IN mvl_vector (data_bus_lines DOWNTO 1);
        ctr_a_wr : IN BIT;
        b      : OUT mvl_vector (data_bus_lines DOWNTO 1);
        ctr_b_rd : IN BIT;
        c      : OUT mvl_vector (data_bus_lines DOWNTO 1);
        ctr_c_rd : IN BIT;
        ck      : IN BIT
      );
END reglx2;

ARCHITECTURE structure OF reglx2 IS
  COMPONENT ff_n
    PORT ( d : IN mvl_vector(data_bus_lines DOWNTO 1);
          q : OUT mvl_vector(data_bus_lines DOWNTO 1);
          ck: IN BIT
        );
  END COMPONENT;

  COMPONENT latch_db
    PORT ( d : IN mvl_vector(data_bus_lines DOWNTO 1);
          q : OUT mvl_vector(data_bus_lines DOWNTO 1);
          ck: IN BIT
        );
  END COMPONENT;

  COMPONENT tri_state_data_bus
    PORT ( inp_3_sta : IN mvl_vector(data_bus_lines DOWNTO 1);
          enable     : IN BIT;
          out_3_sta  : OUT mvl_vector(data_bus_lines DOWNTO 1)
        );
  END COMPONENT;

  SIGNAL x, y : mvl_vector(data_bus_lines DOWNTO 1);

BEGIN
  reg: ff_n PORT MAP ( d => x,
                      q => y,
                      ck => ck
                    );

  sw_input_a: latch_db PORT MAP ( d => a,
                                  q => x,
                                  ck => ctr_a_wr
                                );

  sw_output_b: tri_state_data_bus PORT MAP ( inp_3_sta => y,
                                              enable => ctr_b_rd,
                                              out_3_sta => b
                                            );

  sw_output_c: tri_state_data_bus PORT MAP ( inp_3_sta => y,
                                              enable => ctr_c_rd,
                                              out_3_sta => c
                                            );

END structure;

-- File Name      : lx3reg.vhd
-- Author         : Meyer E. Nigri
-- Date of Creation : 05/12/91
-- Last Update    : 31/12/91
-- Description:
-- =====
-- This entity implements the reglx3 block.
--

LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY reglx3 IS
  PORT ( a      : IN mvl_vector (data_bus_lines DOWNTO 1);
        ctr_a_wr : IN BIT;
        b      : OUT mvl_vector (data_bus_lines DOWNTO 1);
        ctr_b_rd : IN BIT;
        c      : OUT mvl_vector (data_bus_lines DOWNTO 1);
        ctr_c_rd : IN BIT;
        d      : OUT mvl_vector (data_bus_lines DOWNTO 1);
        ctr_d_rd : IN BIT;
        ck      : IN BIT
      );
END reglx3;

ARCHITECTURE structure OF reglx3 IS
  COMPONENT ff_n
    PORT ( d : IN mvl_vector(data_bus_lines DOWNTO 1);
          q : OUT mvl_vector(data_bus_lines DOWNTO 1);
          ck: IN BIT
        );
  END COMPONENT;

  COMPONENT latch_db
    PORT ( d : IN mvl_vector(data_bus_lines DOWNTO 1);
          q : OUT mvl_vector(data_bus_lines DOWNTO 1);
          ck: IN BIT
        );
  END COMPONENT;

  COMPONENT tri_state_data_bus
    PORT ( inp_3_sta : IN mvl_vector(data_bus_lines DOWNTO 1);
          enable     : IN BIT;
          out_3_sta  : OUT mvl_vector(data_bus_lines DOWNTO 1)
        );
  END COMPONENT;

  SIGNAL x, y : mvl_vector(data_bus_lines DOWNTO 1);

BEGIN

```

```

reg: ff_n PORT MAP ( d => x,
                    q => y,
                    ck => ck
                  );

sw_input_a: latch_db PORT MAP ( d => a,
                                q => x,
                                ck => ctr_a_wr
                              );

sw_output_b: tri_state_data_bus PORT MAP ( inp_3_sta => y,
                                             enable => ctr_b_rd,
                                             out_3_sta => b
                                           );

sw_output_c: tri_state_data_bus PORT MAP ( inp_3_sta => y,
                                             enable => ctr_c_rd,
                                             out_3_sta => c
                                           );

sw_output_d: tri_state_data_bus PORT MAP ( inp_3_sta => y,
                                             enable => ctr_d_rd,
                                             out_3_sta => d
                                           );

END structure;

-- File Name      : 2xlmrreg.vhd
-- Author         : Meyer E. Nigri
-- Date of Creation : 05/12/91
-- Last Update    : 31/12/91
-- Description:
-- =====
-- This entity implements the reg2xl block for the MR.
--

LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY reg2xlmr IS
  PORT ( a      : IN mvl_vector (data_bus_lines DOWNTO 1);
        ctr_a_wr : IN BIT;
        b      : IN mvl_vector (data_bus_lines DOWNTO 1);
        ctr_b_wr : IN BIT;
        c      : OUT wire_vector (data_bus_lines DOWNTO 1);
        ctr_c_rd : IN BIT;
        lsb     : OUT mvl;
        lsb_1   : OUT mvl;
        ck      : IN BIT
      );
END reg2xlmr;

ARCHITECTURE structure OF reg2xlmr IS
  COMPONENT or2gate
    PORT ( input1 : IN BIT;
          input2 : IN BIT;
          output  : OUT BIT
        );
  END COMPONENT;

  COMPONENT ff_n
    PORT ( d : IN mvl_vector(data_bus_lines DOWNTO 1);
          q : OUT mvl_vector(data_bus_lines DOWNTO 1);
          ck: IN BIT
        );
  END COMPONENT;

  COMPONENT latch_db
    PORT ( d : IN wire_vector(data_bus_lines DOWNTO 1);
          q : OUT mvl_vector(data_bus_lines DOWNTO 1);
          ck: IN BIT
        );
  END COMPONENT;

  COMPONENT tri_state_data_bus
    PORT ( inp_3_sta : IN mvl_vector(data_bus_lines DOWNTO 1);
          enable     : IN BIT;
          out_3_sta  : OUT wire_vector(data_bus_lines DOWNTO 1)
        );
  END COMPONENT;

  SIGNAL x : mvl_vector(data_bus_lines DOWNTO 1);
  SIGNAL y : mvl_vector(data_bus_lines DOWNTO 1);
  SIGNAL w : wire_vector(data_bus_lines DOWNTO 1);
  SIGNAL ctr_or : BIT;

BEGIN
  ctr_logic: or2gate PORT MAP ( input1 => ctr_a_wr,
                                input2 => ctr_b_wr,
                                output => ctr_or
                              );

  sw_input_a: tri_state_data_bus PORT MAP ( inp_3_sta => a,
                                              enable => ctr_a_wr,
                                              out_3_sta => w
                                            );

  sw_input_b: tri_state_data_bus PORT MAP ( inp_3_sta => b,
                                              enable => ctr_b_wr,
                                              out_3_sta => w
                                            );

  latch_input: latch_db PORT MAP ( d => w,
                                   q => x,
                                   ck => ctr_or
                                 );

  reg: ff_n PORT MAP ( d => x,
                      q => y,
                      ck => ck
                    );

  sw_output_c: tri_state_data_bus PORT MAP ( inp_3_sta => y,
                                              enable => ctr_c_rd,
                                              out_3_sta => c
                                            );

  lsb <= y(2);
  lsb_1 <= y(1);

END structure;

-- File Name      : 2xlreg.vhd
-- Author         : Meyer E. Nigri
-- Date of Creation : 05/12/91
-- Last Update    : 31/12/91
-- Description:
-- =====

```

```
-- | This entity implements the reg2x1 block.
-- |
-- | -----
LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY reg2x1 IS
  PORT ( a      : IN mvl_vector (data_bus_lines DOWNTO 1);
        ctr_a_wr : IN BIT;
        b      : IN mvl_vector (data_bus_lines DOWNTO 1);
        ctr_b_wr : IN BIT;
        c      : OUT wire_vector (data_bus_lines DOWNTO 1);
        ctr_c_rd : IN BIT;
        ctr_e_rd : IN BIT;
        ck      : IN BIT;
  );
END reg2x1;

ARCHITECTURE structure OF reg2x1 IS
  COMPONENT or2gate
    PORT ( input1 : IN BIT;
          input2 : IN BIT;
          output  : OUT BIT;
    );
  END COMPONENT;

  COMPONENT ff_n
    PORT ( d : IN mvl_vector(data_bus_lines DOWNTO 1);
          q : OUT mvl_vector(data_bus_lines DOWNTO 1);
          ck : IN BIT;
    );
  END COMPONENT;

  COMPONENT latch_db
    PORT ( d : IN wire_vector(data_bus_lines DOWNTO 1);
          q : OUT mvl_vector(data_bus_lines DOWNTO 1);
          ck : IN BIT;
    );
  END COMPONENT;

  COMPONENT tri_state_data_bus
    PORT ( inp_3_sta : IN mvl_vector(data_bus_lines DOWNTO 1);
          enable     : IN BIT;
          out_3_sta  : OUT wire_vector(data_bus_lines DOWNTO 1);
    );
  END COMPONENT;

  SIGNAL x, y : mvl_vector(data_bus_lines DOWNTO 1);
  SIGNAL w : wire_vector(data_bus_lines DOWNTO 1);
  SIGNAL ctr_or : BIT;

BEGIN
  ctr_logic: or2gate PORT MAP ( input1 => ctr_a_wr,
                                input2 => ctr_b_wr,
                                output  => ctr_or
  );

  sw_input_a: tri_state_data_bus PORT MAP ( inp_3_sta => a,
                                             enable => ctr_a_wr,
                                             out_3_sta => w
  );

  sw_input_b: tri_state_data_bus PORT MAP ( inp_3_sta => b,
                                             enable => ctr_b_wr,
                                             out_3_sta => w
  );

  latch_input: latch_db PORT MAP ( d => w,
                                   q => x,
                                   ck => ctr_or
  );

  reg: ff_n PORT MAP ( d => x,
                      q => y,
                      ck => ck
  );

  sw_output_c: tri_state_data_bus PORT MAP ( inp_3_sta => y,
                                             enable => ctr_c_rd,
                                             out_3_sta => c
  );

END structure;

-- | File Name      : 3x3reg.vhd
-- | Author         : Meyer E. Nigri
-- | Date of Creation : 05/12/91
-- | Last Update    : 31/12/91
-- | Description:
-- | =====
-- | This entity implements the reg3x3 block.
-- | -----
LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;
```

```
ENTITY reg3x3 IS
  PORT ( a      : IN mvl_vector (data_bus_lines DOWNTO 1);
        ctr_a_wr : IN BIT;
        b      : IN mvl_vector (data_bus_lines DOWNTO 1);
        ctr_b_wr : IN BIT;
        c      : IN mvl_vector (data_bus_lines DOWNTO 1);
        ctr_c_wr : IN BIT;
        d      : OUT wire_vector (data_bus_lines DOWNTO 1);
        ctr_d_rd : IN BIT;
        e      : OUT wire_vector (data_bus_lines DOWNTO 1);
        ctr_e_rd : IN BIT;
        f      : OUT wire_vector (data_bus_lines DOWNTO 1);
        ctr_f_rd : IN BIT;
        ck      : IN BIT;
  );
END reg3x3;

ARCHITECTURE structure OF reg3x3 IS
  COMPONENT or3gate
    PORT ( input1 : IN BIT;
          input2 : IN BIT;
          input3 : IN BIT;
          output  : OUT BIT;
    );
  END COMPONENT;

  COMPONENT ff_n
    PORT ( d : IN mvl_vector(data_bus_lines DOWNTO 1);
          q : OUT mvl_vector(data_bus_lines DOWNTO 1);
          ck : IN BIT;
    );
  END COMPONENT;

  COMPONENT latch_db
    PORT ( d : IN wire_vector(data_bus_lines DOWNTO 1);
          q : OUT mvl_vector(data_bus_lines DOWNTO 1);
          ck : IN BIT;
    );
  END COMPONENT;

  COMPONENT tri_state_data_bus
    PORT ( inp_3_sta : IN mvl_vector(data_bus_lines DOWNTO 1);
          enable     : IN BIT;
          out_3_sta  : OUT wire_vector(data_bus_lines DOWNTO 1);
    );
  END COMPONENT;

  SIGNAL x, y : mvl_vector(data_bus_lines DOWNTO 1);
  SIGNAL w : wire_vector(data_bus_lines DOWNTO 1);
  SIGNAL ctr_or : BIT;

BEGIN
  ctr_logic: or3gate PORT MAP ( input1 => ctr_a_wr,
                                input2 => ctr_b_wr,
                                input3 => ctr_c_wr,
                                output  => ctr_or
  );

  sw_input_a: tri_state_data_bus PORT MAP ( inp_3_sta => a,
                                             enable => ctr_a_wr,
                                             out_3_sta => w
  );

  sw_input_b: tri_state_data_bus PORT MAP ( inp_3_sta => b,
                                             enable => ctr_b_wr,
                                             out_3_sta => w
  );

  sw_input_c: tri_state_data_bus PORT MAP ( inp_3_sta => c,
                                             enable => ctr_c_wr,
                                             out_3_sta => w
  );

  reg: ff_n PORT MAP ( d => x,
                      q => y,
                      ck => ck
  );

  latch_input: latch_db PORT MAP ( d => w,
                                   q => x,
                                   ck => ctr_or
  );

  sw_output_d: tri_state_data_bus PORT MAP ( inp_3_sta => y,
                                             enable => ctr_d_rd,
                                             out_3_sta => d
  );

  sw_output_e: tri_state_data_bus PORT MAP ( inp_3_sta => y,
                                             enable => ctr_e_rd,
                                             out_3_sta => e
  );

  sw_output_f: tri_state_data_bus PORT MAP ( inp_3_sta => y,
                                             enable => ctr_f_rd,
                                             out_3_sta => f
  );

END structure;
```

Definition of Counters

Counters are used to implement the multiplication algorithm and to access memory elements sequentially. Three VHDL entities are described below: count16, counter, and counterl. count16 counts 16 times, after which a signal is set to '1', indicating end of count. This is used in the multiplication of 16-bit values. Its VHDL ARCHITECTURE is described at the behavioural level. counter is also specified at the behavioural level and is used for the

memory addressing mechanism. counter1 is a structural description of a counter that has its output data *latched*. It uses counter as one of its component.

```

-- File Name      : count16.vhd
-- Author         : Meyer E. Nigri
-- Date of Creation : 05/12/91
-- Last Update    : 31/12/91
-- Description:
-- *****
-- This entity implements a counter that outputs '1' when it
-- has counted to 16.
-- *****

LIBRARY xl; USE xl.xl_std.ALL;

ENTITY count_to_16 IS
  PORT (reset : IN BIT;
        ck    : IN BIT;
        count_end : OUT BIT
        );
END count_to_16;

LIBRARY xl; USE xl.xl_std.ALL;

ARCHITECTURE behaviour OF count_to_16 IS
BEGIN
  counting: PROCESS (ck, reset)
    VARIABLE reg : INTEGER := 0;
  BEGIN
    IF (reset = '1') THEN
      reg := 0;
      count_end <= '0';
    ELSIF (ck = '1') THEN
      reg := reg + 1;
      IF (reg = 15) THEN
        count_end <= '1';
      ELSE
        count_end <= '0';
      END IF;
    END IF;
  END PROCESS counting;
END behaviour;

-- File Name      : counter.vhd
-- Author         : Meyer E. Nigri
-- Date of Creation : 05/12/91
-- Last Update    : 31/12/91
-- Description:
-- *****
-- This entity implements a counter for the Memory Addressing
-- module.
-- *****

LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY counter IS
  PORT (reset : IN BIT;
        ck    : IN BIT;
        value : OUT BIT_VECTOR(ram_addr_lines DOWNTO 1)
        );
END counter;

LIBRARY xl; USE xl.xl_std.ALL;

ARCHITECTURE behaviour OF counter IS

```

```

BEGIN
  counting: PROCESS (ck, reset)
    VARIABLE reg : INTEGER := 0;
  BEGIN
    IF (reset = '1') THEN
      FOR j IN 1 TO ram_addr_lines LOOP
        reg := 0;
        value(j) <= '0';
      END LOOP;
    ELSIF (ck = '1') THEN
      reg := reg + 1;
      value <= to_bitv(reg);
    END IF;
  END PROCESS counting;
END behaviour;

-- File Name      : counter1.vhd
-- Author         : Meyer E. Nigri
-- Date of Creation : 05/12/91
-- Last Update    : 31/12/91
-- Description:
-- *****
-- This entity implements a counter which is capable of
-- incrementing and reading previous value at the same time.
-- It is used in the Addressing Module of the Memory Unit.
-- *****

LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY counter1 IS
  PORT (reset : IN BIT;
        ck    : IN BIT;
        tx    : IN BIT;
        value : OUT mvl_vector(ram_addr_lines DOWNTO 1)
        );
END counter1;

ARCHITECTURE structure OF counter1 IS
  COMPONENT counter
    PORT (reset : IN BIT;
          ck    : IN BIT;
          value : OUT mvl_vector(ram_addr_lines DOWNTO 1)
          );
  END COMPONENT;

  COMPONENT latchrab
    PORT (d : IN mvl_vector(ram_addr_lines DOWNTO 1);
          q : OUT mvl_vector(ram_addr_lines DOWNTO 1);
          ck : IN BIT
          );
  END COMPONENT;

  SIGNAL x : mvl_vector(ram_addr_lines DOWNTO 1);
BEGIN
  b1: counter PORT MAP ( reset => reset,
                        ck    => ck,
                        value => x
                        );

  b2: latchrab PORT MAP ( d    => x,
                        q    => value,
                        ck    => tx
                        );
END structure;

```

Definition of Memories

The memories RAM and ROM are described at the behavioural level. The memory's size is defined in the top level configuration file `gn_definition`. The ROM has its contents defined in the file "rom.data", which has been built during hardware simulation to implement the neuron's activation function.

```

-- File Name      : ram.vhd
-- Author         : Meyer E. Nigri
-- Date of Creation : 05/12/91
-- Last Update    : 31/12/91
-- Description:
-- *****
-- This entity implements a RAM.
-- *****

LIBRARY xl; USE xl.xl_std.ALL;

```

```

USE WORK.gn_definition.ALL;
USE STD.textio.ALL, xl.xl_io.ALL;

ENTITY ram IS
  PORT (addr : IN mvl_vector(ram_addr_lines DOWNTO 1);
        data : INOUT mvl_vector(data_bus_lines DOWNTO 1);
        rd_wr : IN BIT;
        cs    : IN BIT
        );
END ram;

ARCHITECTURE behaviour OF ram IS
BEGIN
  PROCESS --(addr, cs, rd_wr)
    VARIABLE mem: mvl_16_memory(0 TO ram_size - 1);
    VARIABLE address: INTEGER;
    VARIABLE l: line;
  BEGIN
    FOR j IN 1 TO data_bus_lines LOOP

```



```

        data[j] <= 'Z';
    END LOOP;
    WAIT UNTIL cs = '1';
    WAIT FOR 1 ns;
    -- remove the following line. This is to test only low addresses...
    address := to_integer(sense(addr AND "0000000000000111"));
    --address := to_integer(SENSE(addr));
    IF (rd_wr = '1') THEN -- it is a read command
        data <= drive(mem(address));
        write (1, "Time is ");
        write (1, NOW, RIGHT, 2);
        write (1, drive(mem(address)), RIGHT, 22);
        write (1, " is being READ from RAM address ");
        write (1, addr, RIGHT, 22);
        writeline (output, 1);
    ELSE -- it is a write command
        mem(address) := sense(data);
        write (1, "Time is ");
        write (1, NOW, RIGHT, 2);
        write (1, sense(data), RIGHT, 22);
        write (1, " is being WRITTEN to RAM address");
        write (1, addr, RIGHT, 22);
        writeline (output, 1);
    END IF;
    WAIT UNTIL cs = '0';
END PROCESS;
END behaviour;

-----
-- | File Name       : rom.vhd
-- | Author          : Meyer E. Nigri
-- | Date of Creation : 05/12/91
-- | Last Update     : 31/12/91
-- |-----
-- | Description:
-- | =====
-- | This entity implements a ROM.
-- |-----

LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY rom IS

```

```

    GENERIC ( rom_file : STRING := "rom.dat");
    PORT ( addr : IN mvl_vector(rom_addr_lines DOWNTO 1);
          data : OUT mvl_vector(data_bus_lines DOWNTO 1);
          rd   : IN BIT;
          cs   : IN BIT
        );
END rom;

LIBRARY xl; USE STD.textio.ALL, xl.xl_io.ALL;

ARCHITECTURE behaviour OF rom IS
    FILE rom_contents: text IS IN rom_file;
BEGIN
    PROCESS (addr, rd, cs)
        VARIABLE mem: mvl_16_memory(0 TO rom_size-1);
        VARIABLE address: INTEGER;
        VARIABLE rline: line;
        VARIABLE first_time: BOOLEAN := TRUE;
    BEGIN
        IF first_time THEN -- Read Rom File Data
            address := 0;
            WHILE NOT endfile(rom_contents) LOOP
                readline (rom_contents, rline);
                read (rline, mem(address));
                address := address + 1;
            END LOOP;
            first_time := FALSE;
        END IF;

        IF (cs = '1' AND rd = '1') THEN
            address := to_integer(addr);
            --Just to test. Remove the 3 next lines...
            IF (address > 10) THEN
                address := 10;
            END IF;
            data <= mem(address);
        ELSE
            FOR j IN 1 TO data_bus_lines LOOP
                data[j] <= 'Z';
            END LOOP;
        END IF;
    END PROCESS;
END behaviour;

```

Basic Components

Several basic components are necessary to support the implementation of the PE's three units. Some of these elements are described in the structural domain. Some are behavioural descriptions.

```

-----
-- | Description:
-- | =====
-- | This entity implements a flip-flop of n bits.
-- |-----

LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY ff_n IS
    PORT ( d : IN mvl_vector(data_bus_lines DOWNTO 1);
          q : OUT mvl_vector(data_bus_lines DOWNTO 1);
          ck : IN BIT
        );
END ff_n;

ARCHITECTURE behaviour OF ff_n IS
BEGIN
    PROCESS (ck, d)
        VARIABLE first_time: BOOLEAN := TRUE;
    BEGIN
        IF first_time THEN
            FOR i IN 1 TO data_bus_lines LOOP
                q(i) <= '0';
            END LOOP;
            first_time := FALSE;
        END IF;

        IF (clear = '1' AND preset = '0') THEN
            q <= '0';
            notq <= '1';
        ELSIF (clear = '0' AND preset = '1') THEN
            q <= '1';
            notq <= '0';
        ELSIF (clear = '0' AND preset = '0') THEN
            IF (ck = '1') THEN
                q <= d;
                notq <= NOT d;
            END IF;
        ELSE
            ASSERT (clear = '1' AND (preset = '1'))
            REPORT "Clear and Preset = 0 at the same time in ff"
            SEVERITY WARNING;
        END IF;
    END PROCESS;
END behaviour;

-----
-- | File Name       : ff_n.vhd
-- | Author          : Meyer E. Nigri
-- | Date of Creation : 05/12/91
-- | Last Update     : 31/12/91
-- |-----

```

```

LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY ff_n IS
    PORT ( d : IN mvl_vector(data_bus_lines DOWNTO 1);
          q : OUT mvl_vector(data_bus_lines DOWNTO 1);
          ck : IN BIT
        );
END ff_n;

ARCHITECTURE behaviour OF ff_n IS
BEGIN
    PROCESS (ck, d)
        VARIABLE first_time: BOOLEAN := TRUE;
    BEGIN
        IF first_time THEN
            FOR i IN 1 TO data_bus_lines LOOP
                q(i) <= '0';
            END LOOP;
            first_time := FALSE;
        END IF;

        IF (ck = '1') THEN
            q <= d;
        END IF;
    END PROCESS;
END behaviour;

-----
-- | File Name       : inverter.vhd
-- | Author          : Meyer E. Nigri
-- | Date of Creation : 05/12/91
-- | Last Update     : 20/12/91
-- |-----
-- | Description:
-- | =====
-- | This entity implements an inverter gate.
-- |-----

LIBRARY xl; USE xl.xl_std.ALL;

ENTITY inverter IS
    PORT ( input : IN BIT;
          output: OUT BIT
        );
END inverter;

ARCHITECTURE behaviour OF inverter IS
BEGIN
    PROCESS (input)
    BEGIN
        output <= NOT input;
    END PROCESS;
END behaviour;

```

```
-- | File Name      : or2gate.vhd
-- | Author        : Meyer E. Nigri
-- | Date of Creation : 05/12/91
-- | Last Update    : 20/12/91
-- | Description:
-- | =====
-- | This entity implements an or gate of 2 inputs.
```

```
LIBRARY xl; USE xl.xl_std.ALL;
```

```
ENTITY or2gate IS
  PORT ( input1 : IN BIT;
        input2 : IN BIT;
        output  : OUT BIT
  );
```

```
END or2gate;
```

```
ARCHITECTURE behaviour OF or2gate IS
```

```
BEGIN
  PROCESS (input1, input2)
  BEGIN
    output <= input1 OR input2;
  END PROCESS;
END behaviour;
```

```
-- | File Name      : or3gate.vhd
-- | Author        : Meyer E. Nigri
-- | Date of Creation : 05/12/91
-- | Last Update    : 20/12/91
-- | Description:
-- | =====
-- | This entity implements an or gate of 3 inputs.
```

```
LIBRARY xl; USE xl.xl_std.ALL;
```

```
ENTITY or3gate IS
```

```
  PORT ( input1 : IN BIT;
        input2 : IN BIT;
        input3 : IN BIT;
        output  : OUT BIT
  );
```

```
END or3gate;
```

```
ARCHITECTURE behaviour OF or3gate IS
```

```
BEGIN
  PROCESS (input1, input2, input3)
  BEGIN
    output <= (input1 OR input2) OR input3;
  END PROCESS;
END behaviour;
```

```
-- | File Name      : tri_state.vhd
-- | Author        : Meyer E. Nigri
-- | Date of Creation : 05/12/91
-- | Last Update    : 31/12/91
-- | Description:
-- | =====
-- | This entity implements a tri_state port.
```

```
LIBRARY xl; USE xl.xl_std.ALL;
```

```
USE WORK.ram_definition.ALL;
```

```
ENTITY b_tri_state IS
```

```
  PORT ( inp_3_sta : IN BIT_VECTOR(addr_lines DOWNTO 1);
        enable    : IN BIT;
        out_3_sta  : OUT mvl_vector(addr_lines DOWNTO 1)
  );
```

```
END b_tri_state;
```

```
ARCHITECTURE behaviour OF b_tri_state IS
```

```
BEGIN
  PROCESS (enable, inp_3_sta)
  VARIABLE first_time: BOOLEAN := TRUE;
  BEGIN
    IF first_time THEN
      FOR i IN addr_lines DOWNTO 1 LOOP
        out_3_sta(i) <= 'Z';
      END LOOP;
      first_time := FALSE;
    END IF;

    IF (enable = '1') THEN
      --out_3_sta <= DRIVE(to_mvlv(inp_3_sta));
      --out_3_sta <= inp_3_sta;
      out_3_sta <= to_mvlv(inp_3_sta);
    ELSE
      FOR i IN addr_lines DOWNTO 1 LOOP
        out_3_sta(i) <= 'Z';
      END LOOP;
    END IF;
  END PROCESS;
END behaviour;
```

```
-- | File Name      : xor2gate.vhd
-- | Author        : Meyer E. Nigri
-- | Date of Creation : 05/12/91
-- | Last Update    : 20/12/91
-- | Description:
-- | =====
-- | This entity implements an xor gate of 2 inputs.
```

```
LIBRARY xl; USE xl.xl_std.ALL;
```

```
ENTITY xor2gate IS
```

```
  PORT ( input1 : IN BIT;
        input2 : IN BIT;
        output  : OUT BIT
  );
```

```
END xor2gate;
```

```
ARCHITECTURE behaviour OF xor2gate IS
```

```
BEGIN
  PROCESS (input1, input2)
  BEGIN
    output <= input1 XOR input2;
  END PROCESS;
END behaviour;
```

```
-- | File Name      : latch_db.vhd
-- | Author        : Meyer E. Nigri
-- | Date of Creation : 05/12/91
-- | Last Update    : 31/12/91
-- | Description:
-- | =====
-- | This entity implements a latch of n bits for the DATA BUS
```

```
-- |
```

```
LIBRARY xl; USE xl.xl_std.ALL;
```

```
USE WORK.gn_definition.ALL;
```

```
ENTITY latch_db IS
```

```
  PORT ( d : IN mvl_vector(data_bus_lines DOWNTO 1);
        q : OUT mvl_vector(data_bus_lines DOWNTO 1);
        ck : IN BIT
  );
```

```
END latch_db;
```

```
ARCHITECTURE behaviour OF latch_db IS
```

```
BEGIN
```

```
  PROCESS (d, ck)
  VARIABLE first_time: BOOLEAN := TRUE;
  VARIABLE data : mvl_vector (data_bus_lines DOWNTO 1);
  BEGIN
    IF first_time THEN
      FOR i IN 1 TO data_bus_lines LOOP
        q(i) <= '0';
        data(i) := '0';
      END LOOP;
      first_time := FALSE;
    END IF;
```

```
    IF (ck = '1') THEN
      data := d;
      q <= d;
    ELSE
      q <= data;
    END IF;
  END PROCESS;
```

```
END behaviour;
```

```
-- | File Name      : latchrab.vhd
```

```
-- | Author        : Meyer E. Nigri
```

```
-- | Date of Creation : 05/12/91
```

```
-- | Last Update    : 31/12/91
```

```
-- | Description:
```

```
-- | =====
```

```
-- | This entity implements a latch of n bits for the RAM
```

```
-- | ADDRESS BUS.
```

```
LIBRARY xl; USE xl.xl_std.ALL;
```

```
USE WORK.gn_definition.ALL;
```

```
ENTITY latchrab IS
```

```
  PORT ( d : IN mvl_vector(ram_addr_lines DOWNTO 1);
        q : OUT mvl_vector(ram_addr_lines DOWNTO 1);
        ck : IN BIT
  );
```

```
END latchrab;
```

```
ARCHITECTURE behaviour OF latchrab IS
```

```
BEGIN
```

```
  PROCESS (d, ck)
  VARIABLE first_time: BOOLEAN := TRUE;
  BEGIN
    IF first_time THEN
      FOR i IN 1 TO ram_addr_lines LOOP
        q(i) <= '1';
      END LOOP;
      first_time := FALSE;
    END IF;

    IF (ck = '1') THEN
      q <= d;
    END IF;
  END PROCESS;
```

```
END behaviour;
```

```
-- | File Name      : sw_n.vhd
```

```
-- | Author        : Meyer E. Nigri
```

```
-- | Date of Creation : 05/12/91
```

```
-- | Last Update    : 31/12/91
```

```
-- | Description:
```

```
-- | =====
```

```
-- | This entity implements a switch of n bits.
```

```
LIBRARY xl; USE xl.xl_std.ALL;
```

```
USE WORK.gn_definition.ALL;
```

```
ENTITY sw_n IS
```

```
  PORT ( d : IN mvl_vector(data_bus_lines DOWNTO 1);
        q : OUT mvl_vector(data_bus_lines DOWNTO 1);
        en : IN BIT
  );
```

```
END sw_n;
```

```
ARCHITECTURE behaviour OF sw_n IS
```

```
BEGIN
```

```
  PROCESS (en, d)
  BEGIN
    IF (en = '1') THEN
      q <= d;
    ELSIF (en = '0') THEN
      FOR i IN 1 TO data_bus_lines LOOP
        q(i) <= 'Z';
      END LOOP;
    END IF;
  END PROCESS;
```

```
END behaviour;
```

Communication Unit

This unit is implemented by the VHDL entity `comm_unit`, which is built upon the entities `cu_datapath` and `cu_fsm` in the behavioural domain. The former is responsible for uniquely identifying the PE's address. The latter implements the FSM for this unit, which keeps looking for signals sent by `cu_datapath` and from the external control bus. The logic for realising the `cs/rdy` function is done in the FSM by the component `cs_rdy_logic`.

```

-- File Name      : cu.vhd
-- Author         : Meyer E. Nigri
-- Date of Creation : 05/12/91
-- Last Update    : 20/12/91
--
-- Description:
--
-- This entity implements the Communication Unit of the
-- Generic Neuron's processing element.
--
LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY comm_unit IS
  PORT( in_bus      : IN  mvl_vector(data_bus_lines DOWNTO 1);
        out_bus     : OUT mvl_vector(data_bus_lines DOWNTO 1);
        add_bus     : IN  mvl_vector(ram_addr_lines DOWNTO 1);
        reset       : IN  BIT; -- 0=Initialisation; 1=Execution
        load        : IN  BIT;
        frw_bkw     : IN  BIT;
        phi1        : IN  BIT;
        phi2        : IN  BIT;
        cs_rdy      : INOUT wire;
        mem_add_bus : OUT mvl_vector(ram_addr_lines DOWNTO 1);
        weights_bus : OUT mvl_vector(data_bus_lines DOWNTO 1);
        states_bus  : OUT mvl_vector(data_bus_lines DOWNTO 1);
        z_bus       : IN  mvl_vector(data_bus_lines DOWNTO 1);
        cu_cs_wgt_frw : OUT BIT;
        cu_cs_wgt_bkw : OUT BIT;
        cu_cs_sta_frw : OUT BIT;
        cu_cs_sta_bkw : OUT BIT;
        cu_rd_wr_wgt_frw : OUT BIT;
        cu_rd_wr_wgt_bkw : OUT BIT;
        cu_rd_wr_sta_frw : OUT BIT;
        cu_rd_wr_sta_bkw : OUT BIT;
        cu_inc_ptr   : OUT BIT;
        cu_reset_ptr : OUT BIT;
        eval_state   : OUT BIT;
        eval_error   : OUT BIT;
        state_rdy    : IN  BIT;
        error_rdy    : IN  BIT;
        cu_en_cnr_add : OUT BIT;
        cu_tx_ptr    : OUT BIT;
        cu_en_outbus : OUT BIT
  );
END comm_unit;

ARCHITECTURE structure OF comm_unit IS
  COMPONENT cu_datapath
    PORT( in_bus      : IN  mvl_vector(data_bus_lines DOWNTO 1);
          out_bus     : OUT mvl_vector(data_bus_lines DOWNTO 1);
          add_bus     : IN  mvl_vector(ram_addr_lines DOWNTO 1);
          is_my_add   : OUT BIT;
          next_layer  : OUT BIT;
          prev_layer  : OUT BIT;
          en_mem_add  : IN  BIT;
          en_outbus   : IN  BIT;
          en_weightbus : IN  BIT;
          en_staerrbus : IN  BIT;
          mem_add_bus : OUT mvl_vector(ram_addr_lines DOWNTO 1);
          weights_bus : OUT mvl_vector(data_bus_lines DOWNTO 1);
          states_bus  : OUT mvl_vector(data_bus_lines DOWNTO 1);
          z_bus       : IN  mvl_vector(data_bus_lines DOWNTO 1)
        );
  END COMPONENT;

  COMPONENT cu_control
    PORT( reset      : IN  BIT;
          load       : IN  BIT; -- 0=Initialisation; 1=Execution
          frw_bkw    : IN  BIT;
          phi1       : IN  BIT;
          phi2       : IN  BIT;
          cs_rdy     : INOUT wire;
          is_my_add  : IN  BIT;
          next_layer : IN  BIT;
          prev_layer : IN  BIT;
          en_mem_add : OUT BIT;
          en_outbus  : OUT BIT;
          en_weightbus : OUT BIT;
          en_staerrbus : OUT BIT;
          cu_cs_wgt_frw : OUT BIT;
          cu_cs_wgt_bkw : OUT BIT;
          cu_cs_sta_frw : OUT BIT;
          cu_cs_sta_bkw : OUT BIT;
          cu_rd_wr_wgt_frw : OUT BIT;
          cu_rd_wr_wgt_bkw : OUT BIT;
          cu_rd_wr_sta_frw : OUT BIT;
          cu_rd_wr_sta_bkw : OUT BIT;
          cu_inc_ptr   : OUT BIT;
          cu_reset_ptr : OUT BIT;
          eval_state   : OUT BIT;
          eval_error   : OUT BIT
        );
  END COMPONENT;

  cu_datapath : cu_datapath
    PORT map (
      in_bus      => in_bus,
      out_bus     => out_bus,
      add_bus     => add_bus,
      is_my_add   => is_my_add,
      next_layer  => next_layer,
      prev_layer  => prev_layer,
      en_mem_add  => en_mem_add,
      en_outbus   => en_outbus,
      en_weightbus => en_weightbus,
      en_staerrbus => en_staerrbus,
      mem_add_bus => mem_add_bus,
      weights_bus => weights_bus,
      states_bus  => states_bus,
      z_bus       => z_bus
    );

  cu_control : cu_control
    PORT map (
      reset      => reset,
      load       => load,
      frw_bkw    => frw_bkw,
      phi1       => phi1,
      phi2       => phi2,
      cs_rdy     => cs_rdy,
      is_my_add  => is_my_add,
      next_layer => next_layer,
      prev_layer => prev_layer,
      en_mem_add => en_mem_add,
      en_outbus  => en_outbus,
      en_weightbus => en_weightbus,
      en_staerrbus => en_staerrbus,
      cu_cs_wgt_frw => cu_cs_wgt_frw,
      cu_cs_wgt_bkw => cu_cs_wgt_bkw,
      cu_cs_sta_frw => cu_cs_sta_frw,
      cu_cs_sta_bkw => cu_cs_sta_bkw,
      cu_rd_wr_wgt_frw => cu_rd_wr_wgt_frw,
      cu_rd_wr_wgt_bkw => cu_rd_wr_wgt_bkw,
      cu_rd_wr_sta_frw => cu_rd_wr_sta_frw,
      cu_rd_wr_sta_bkw => cu_rd_wr_sta_bkw,
      cu_inc_ptr   => cu_inc_ptr,
      cu_reset_ptr => cu_reset_ptr,
      eval_state   => eval_state,
      eval_error   => eval_error
    );
END structure;

```

```

state_rdy : IN  BIT;
error_rdy : IN  BIT;
cu_en_cnr_add : OUT BIT;
cu_tx_ptr : OUT BIT
);
END COMPONENT;

SIGNAL a,b,c,d,e,f,g : BIT;

FOR b2: cu_control USE ENTITY WORK.cu_control(structure);
FOR b1: cu_datapath USE ENTITY WORK.cu_datapath(behaviour);

BEGIN
  b2: cu_control PORT MAP ( reset      => reset,
                           load       => load,
                           frw_bkw    => frw_bkw,
                           phi1       => phi1,
                           phi2       => phi2,
                           cs_rdy     => cs_rdy,
                           is_my_add  => a,
                           next_layer => b,
                           prev_layer => c,
                           en_mem_add => d,
                           en_outbus  => e,
                           en_weightbus => f,
                           en_staerrbus => g,
                           cu_cs_wgt_frw => cu_cs_wgt_frw,
                           cu_cs_wgt_bkw => cu_cs_wgt_bkw,
                           cu_cs_sta_frw => cu_cs_sta_frw,
                           cu_cs_sta_bkw => cu_cs_sta_bkw,
                           cu_rd_wr_wgt_frw => cu_rd_wr_wgt_frw,
                           cu_rd_wr_wgt_bkw => cu_rd_wr_wgt_bkw,
                           cu_rd_wr_sta_frw => cu_rd_wr_sta_frw,
                           cu_rd_wr_sta_bkw => cu_rd_wr_sta_bkw,
                           cu_inc_ptr   => cu_inc_ptr,
                           cu_reset_ptr => cu_reset_ptr,
                           eval_state   => eval_state,
                           eval_error   => eval_error,
                           state_rdy    => state_rdy,
                           error_rdy    => error_rdy,
                           cu_en_cnr_add => cu_en_cnr_add,
                           cu_tx_ptr    => cu_tx_ptr
                         );

  b1: cu_datapath PORT MAP (
    in_bus      => in_bus,
    out_bus     => out_bus,
    add_bus     => add_bus,
    is_my_add   => a,
    next_layer  => b,
    prev_layer  => c,
    en_mem_add  => d,
    en_outbus   => e,
    en_weightbus => f,
    en_staerrbus => g,
    mem_add_bus => mem_add_bus,
    weights_bus => weights_bus,
    states_bus  => states_bus,
    z_bus       => z_bus
  );

  cu_en_outbus <= e;
END structure;

-- File Name      : cu_datapath.vhd
-- Author         : Meyer E. Nigri
-- Date of Creation : 05/12/91
-- Last Update    : 20/12/91
--
-- Description:
--
-- This entity implements the Communication Unit of the
-- Generic Neuron's processing element.
--
LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY cu_datapath IS
  PORT( in_bus      : IN  mvl_vector(data_bus_lines DOWNTO 1);
        out_bus     : OUT mvl_vector(data_bus_lines DOWNTO 1);
        add_bus     : IN  mvl_vector(ram_addr_lines DOWNTO 1);
        is_my_add   : OUT BIT;
        next_layer  : OUT BIT;
        prev_layer  : OUT BIT;
        en_mem_add  : IN  BIT;
        en_outbus   : IN  BIT;
        en_weightbus : IN  BIT;
        en_staerrbus : IN  BIT;
        mem_add_bus : OUT mvl_vector(ram_addr_lines DOWNTO 1);
        weights_bus : OUT mvl_vector(data_bus_lines DOWNTO 1);
        states_bus  : OUT mvl_vector(data_bus_lines DOWNTO 1);
        z_bus       : IN  mvl_vector(data_bus_lines DOWNTO 1)
  );
END cu_datapath;

LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ARCHITECTURE behaviour OF cu_datapath IS

```



```

-- It means that there is a state coming from
-- the previous
-- layer, and it must be read by this PE.
-- The actions are:
-- Read state from the in_data_bus and write it
-- onto the state
-- Memory block. Once this condition is true,
-- it will remain so
-- for the number of neuron in the previous
-- layer * clock_period,
-- because every neuron in a certain layer finishes
-- its calculation
-- at the same time.
IF (load = '0' AND frw_bkw = '1' AND
    next_state <= 'B') THEN
    WAIT UNTIL phil = '1';
    en_staerrbus <= '1', '0' AFTER tpw;
    en_mem_add <= '1', '0' AFTER tpw;
    cs_sta_frww <= '1', '0' AFTER tpw;
    rd_wr_sta_frww <= '0';
    eval_state <= '1', '0' AFTER tpw;
ELSE
    next_state <= 'C';
END IF;
WHEN 'C' =>
    -- do nothing unless one of the following
    -- conditions is true
    -- Which means, wait for Execution Unit calculates
    -- Thresh(S * W) and
    -- activates state_rdy. When this happens, set
    -- cs_rdy and send
    -- S; This means that every neuron from this
    -- layer has finished
    -- its execution. Then, wait for cs_rdy from neurons
    -- from the next
    -- layer to initiate backward phase.
    IF (load = '0' AND frw_bkw = '1' AND
        is_my_add = '1' AND state_rdy = '1') THEN
        next_state <= 'C';
        WAIT UNTIL phil = '1';
        cs_rdy_ctr <= '1', '0' AFTER clock_period - tps;
        cs_rdy_flag <= '1', '0' AFTER clock_period - tps;
        en_outbus <= '1', '0' AFTER tpw;
        write (l, "Generating cs_rdy.....");
        writeline (output, l);
        -- wait for backward phase.
    ELSEIF (load = '0' AND frw_bkw = '0' AND
        next_state <= 'D') THEN
        WAIT UNTIL phil = '1';
        en_staerrbus <= '1', '0' AFTER tpw;
        en_mem_add <= '1', '0' AFTER tpw;
        cs_sta_bkw <= '1', '0' AFTER tpw;
        rd_wr_sta_bkw <= '0';
        eval_error <= '1', '0' AFTER tpw;
    ELSE
        next_state <= 'E';
    END IF;
WHEN 'D' =>
    -- It means that there is an error coming from the next
    -- layer, and it must be read by this PE.
    -- The actions are:
    -- Read error from the in_data_bus and write it
    -- onto the Error
    -- Memory block. Once this condition is true,
    -- it will remain so
    -- for the number of neuron in the
    -- next layer * clock_period,
    -- because every neuron in a certain layer finishes
    -- its calculation at the same time.
    IF (load = '0' AND frw_bkw = '1' AND cs_rdy_in = '1') THEN
        next_state <= 'D';
        WAIT UNTIL phil = '1';
        en_staerrbus <= '1', '0' AFTER tpw;
        en_mem_add <= '1', '0' AFTER tpw;
        cs_sta_bkw <= '1', '0' AFTER tpw;
        rd_wr_sta_bkw <= '0';
        eval_error <= '1', '0' AFTER tpw;
    ELSE
        next_state <= 'E';
    END IF;
WHEN 'E' =>
    -- do nothing until the following condition is active.
    IF (load = '0' AND frw_bkw = '0' AND
        is_my_add = '1' AND error_rdy = '1') THEN
        next_state <= 'A';
        WAIT UNTIL phil = '1';
        cs_rdy_ctr <= '1', '0' AFTER tpw;
        cs_rdy_flag <= '1', '0' AFTER tpw;
        en_outbus <= '1', '0' AFTER tpw;
    ELSE
        next_state <= 'E';
    END IF;
END CASE;
END PROCESS finite_state_machine;
END behaviour;

-- File Name : cu_ctr.vhd
-- Author : Meyer E. Nigri
-- Date of Creation : 05/12/91
-- Last Update : 20/12/91
-- Description:
-- This entity implements the Communication Unit's control
-- of the Generic Neuron's processing element.

LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY cu_control IS
    PORT (
        reset : IN BIT;
        load : IN BIT; -- 0=Initialisation; 1=Execution
        frw_bkw : IN BIT;
        phil : IN BIT;
        phi2 : IN BIT;
        cs_rdy : INOUT wire;
        is_my_add : IN BIT;
        next_layer : IN BIT;
        prev_layer : IN BIT;
        en_mem_add : OUT BIT;
        en_outbus : OUT BIT;
        en_weightbus : OUT BIT;
        en_staerrbus : OUT BIT;
        cu_ca_wgt_frww : OUT BIT;
        cu_ca_sta_frww : OUT BIT;
        cu_ca_sta_bkw : OUT BIT;
        cu_rd_wr_wgt_frww : OUT BIT;
        cu_rd_wr_wgt_bkw : OUT BIT;
        cu_rd_wr_sta_frww : OUT BIT;
        cu_rd_wr_sta_bkw : OUT BIT;
        cu_inc_ptr : OUT BIT;
    );
END cu_control;

ARCHITECTURE structure OF cu_control IS
    COMPONENT cu_fsm
        PORT (
            reset : IN BIT;
            load : IN BIT; -- 0=Initialisation; 1=Execution
            frw_bkw : IN BIT;
            phil : IN BIT;
            phi2 : IN BIT;
            cs_rdy_flag : OUT BIT;
            cs_rdy_in : IN BIT;
            cs_rdy_ctr : OUT BIT;
            is_my_add : IN BIT;
            next_layer : IN BIT;
            prev_layer : IN BIT;
            en_mem_add : OUT BIT;
            en_outbus : OUT BIT;
            en_weightbus : OUT BIT;
            en_staerrbus : OUT BIT;
            cs_wgt_frww : OUT BIT;
            cs_wgt_bkw : OUT BIT;
            cs_sta_frww : OUT BIT;
            cs_sta_bkw : OUT BIT;
            rd_wr_wgt_frww : OUT BIT;
            rd_wr_wgt_bkw : OUT BIT;
            rd_wr_sta_frww : OUT BIT;
            rd_wr_sta_bkw : OUT BIT;
            inc_ptr : OUT BIT;
            reset_ptr : OUT BIT;
            eval_state : OUT BIT;
            eval_error : OUT BIT;
            state_rdy : IN BIT;
            error_rdy : IN BIT;
            en_cntr_add : OUT BIT;
            tx_ptr : OUT BIT;
        );
    END COMPONENT;

    COMPONENT cu_combinational
        PORT (
            cs_rdy_flag : IN BIT;
            cs_rdy_in : OUT BIT;
            cs_rdy_ctr : IN BIT;
            cs_rdy : INOUT wire;
            phi2 : IN BIT;
        );
    END COMPONENT;

    SIGNAL a, b, c: BIT;

    --FOR fsm_block: cu_fsm USE ENTITY work.cu_fsm(behaviour);
    --FOR comb_logic: cu_combinational USE ENTITY
    work.cu_combinational(structure);

BEGIN
    fsm_block: cu_fsm PORT MAP (
        reset => reset,
        load => load,
        frw_bkw => frw_bkw,
        phil => phil,
        phi2 => phi2,
        cs_rdy_flag => a,
        cs_rdy_in => b,
        cs_rdy_ctr => c,
        is_my_add => is_my_add,
        next_layer => next_layer,
        prev_layer => prev_layer,
        en_mem_add => en_mem_add,
        en_outbus => en_outbus,
        en_weightbus => en_weightbus,
        en_staerrbus => en_staerrbus,
        cs_wgt_frww => cu_ca_wgt_frww,
        cs_wgt_bkw => cu_ca_wgt_bkw,
        cs_sta_frww => cu_ca_sta_frww,
        cs_sta_bkw => cu_ca_sta_bkw,
        rd_wr_wgt_frww => cu_rd_wr_wgt_frww,
        rd_wr_wgt_bkw => cu_rd_wr_wgt_bkw,
        rd_wr_sta_frww => cu_rd_wr_sta_frww,
        rd_wr_sta_bkw => cu_rd_wr_sta_bkw,
        inc_ptr => cu_inc_ptr,
        reset_ptr => cu_reset_ptr,
        eval_state => eval_state,
        eval_error => eval_error,
        state_rdy => state_rdy,
        error_rdy => error_rdy,
        en_cntr_add => cu_en_cntr_add,
        tx_ptr => cu_tx_ptr
    );

    comb_logic: cu_combinational PORT MAP (
        cs_rdy_flag => a,
        cs_rdy_in => b,
        cs_rdy_ctr => c,
        cs_rdy => cs_rdy,
        phi2 => phi2
    );
END structure;

-- File Name : cu_comb.vhd
-- Author : Meyer E. Nigri
-- Date of Creation : 05/12/91
-- Last Update : 20/12/91
-- Description:
-- This entity implements the combinational logic of the
-- communication unit's control part.

LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY cu_combinational IS
    PORT (
        cs_rdy_flag : IN mvl;
        cs_rdy_in : OUT mvl;
        cs_rdy_ctr : IN BIT;
        cs_rdy : INOUT mvl;
        phi2 : IN BIT;
    );
    -- See comments by the end of this file
    -- tx_ptr_in : IN BIT;
    -- tx_ptr_out : OUT BIT;
END cu_combinational;

ARCHITECTURE behaviour OF cu_combinational IS
    COMPONENT cu_ca_rdy_logic
        PORT (
            cs_rdy_flag : IN mvl;
            cs_rdy_in : OUT mvl;
            cs_rdy_ctr : IN BIT;
            cs_rdy_in_out : INOUT mvl;
        );
    END COMPONENT;

```

```

END COMPONENT;

BEGIN
  cs_logic_bk: cu_cs_rdy_logic PORT MAP
  ( cs_rdy_flag => cs_rdy_flag,
    cs_rdy_in   => cs_rdy_in,
    cs_rdy_ctr  => cs_rdy_ctr,
    cs_rdy_in_out => cs_rdy
  );

  -- This is not needed here, since the "and phi2"
  -- logic is implemented
  -- directly in the FSM process.
  --
  -- PROCESS (phi2, tx_ptr_in)
  -- BEGIN
  --   tx_ptr_out <= tx_ptr_in AND phi2;
  -- END PROCESS;
END behaviour;

-- File Name      : cu_cs_rdy.vhd
-- Author         : Meyer E. Nigri
-- Date of Creation : 05/12/91
-- Last Update    : 20/12/91
--
-- Description:
--
-- This entity implements the cs_rdy logic for the
-- Communication Unit's control of the Generic Neuron's
-- processing element.

LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY cu_cs_rdy_logic IS

```

```

  PORT ( cs_rdy_flag : IN  mvl;
         cs_rdy_in   : OUT mvl;
         cs_rdy_ctr  : IN  BIT;
         cs_rdy_in_out : INOUT mvl
       );
END cu_cs_rdy_logic;

ARCHITECTURE structure OF cu_cs_rdy_logic IS
  COMPONENT tri_state_gate
    PORT ( inp_3_sta : IN mvl;
          enable    : IN BIT;
          out_3_sta  : OUT mvl
        );
  END COMPONENT;
  COMPONENT inverter
    PORT ( input : IN BIT;
          output : OUT BIT
        );
  END COMPONENT;
  SIGNAL tri2_ctr: BIT;

BEGIN
  tri_state_1: tri_state_gate PORT MAP ( inp_3_sta => cs_rdy_flag,
                                         enable   => cs_rdy_ctr,
                                         out_3_sta => cs_rdy_in_out
                                       );

  tri_state_2: tri_state_gate PORT MAP ( inp_3_sta =>
                                         cs_rdy_in_out,
                                         enable   => tri2_ctr,
                                         out_3_sta => cs_rdy_in
                                       );

  invert_ctr: inverter PORT MAP ( input => cs_rdy_ctr,
                                output => tri2_ctr
                              );
END structure;

```

Memory Unit

This unit comprises RAM components for states and weights, which are implemented by the storage module. The addressing module is necessary to address the memories. Each block of states is connected to *Bus A*, while the block of weights are connected into *Bus B*. The addressing is made in such a way that the two blocks of memory are accessed simultaneously.

```

-- File Name      : mu.vhd
-- Author         : Meyer E. Nigri
-- Date of Creation : 05/12/91
-- Last Update    : 31/12/91
--
-- Description:
--
-- This entity implements the Memory Unit of the
-- Generic Neuron's processing element.

LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY mem_unit IS
  PORT ( cu_inc_ptr   : IN BIT;
        cu_inc_ptr   : IN BIT;
        cu_tx         : IN BIT;
        cu_reset_ptr : IN BIT;
        cu_reset_ptr : IN BIT;
        cu_en_cntr_add : IN BIT;
        cu_en_cntr_add : IN BIT;
        end_frv_ph    : OUT BIT;
        end_bkv_ph    : OUT BIT;
        mem_add_bus   : INOUT wire_vector(ram_addr_lines DOWNTO 1);
        weight_bus    : INOUT wire_vector(data_bus_lines DOWNTO 1);
        states_bus    : INOUT wire_vector(data_bus_lines DOWNTO 1);
        cu_cs_wgt_frv : IN BIT;
        cu_rd_wr_wgt_frv : IN BIT;
        cu_rd_wr_wgt_frv : IN BIT;
        cu_cs_wgt_bkv : IN BIT;
        cu_rd_wr_wgt_bkv : IN BIT;
        cu_rd_wr_wgt_bkv : IN BIT;
        cu_cs_sta_frv : IN BIT;
        cu_rd_wr_sta_frv : IN BIT;
        cu_rd_wr_sta_frv : IN BIT;
        cu_cs_sta_bkv : IN BIT;
        cu_rd_wr_sta_bkv : IN BIT;
        cu_rd_wr_sta_bkv : IN BIT
      );
END mem_unit;

ARCHITECTURE structure OF mem_unit IS

```

```

  COMPONENT comparator
    PORT ( value1 : IN mvl_vector(ram_addr_lines DOWNTO 1);
          value2 : IN mvl_vector(ram_addr_lines DOWNTO 1);
          result  : OUT BIT
        );
  END COMPONENT;

  COMPONENT counter1
    PORT ( reset : IN BIT;
          ck     : IN BIT;
          tx     : IN BIT;
          value  : OUT mvl_vector(ram_addr_lines DOWNTO 1)
        );
  END COMPONENT;

  COMPONENT tri_state_ram_addr
    PORT ( inp_3_sta : IN mvl_vector(ram_addr_lines DOWNTO 1);
          enable    : IN BIT;
          out_3_sta  : OUT wire_vector(ram_addr_lines DOWNTO 1)
        );
  END COMPONENT;

  COMPONENT or2gate
    PORT ( in1 : IN BIT;
          in2 : IN BIT;
          output : OUT BIT
        );
  END COMPONENT;

  COMPONENT storage_module
    PORT ( mem_add_bus : IN wire_vector(ram_addr_lines DOWNTO 1);
          weight_bus  : INOUT wire_vector(data_bus_lines DOWNTO 1);
          states_bus  : INOUT wire_vector(data_bus_lines DOWNTO 1);
          cs_wgt_frv  : IN BIT;
          rd_wr_wgt_frv : IN BIT;
          cs_wgt_bkv  : IN BIT;
          rd_wr_wgt_bkv : IN BIT;
          cs_sta_frv  : IN BIT;
          rd_wr_sta_frv : IN BIT;
          cs_sta_bkv  : IN BIT;
          rd_wr_sta_bkv : IN BIT
        );
  END COMPONENT;

  SIGNAL cntr_output: mvl_vector(ram_addr_lines DOWNTO 1);
  -- There are 2 neurons connected to this one from
  -- the previous layer.
  SIGNAL reg1_output: mvl_vector(ram_addr_lines DOWNTO 1);
  := "0000000000000001";
  -- This neuron is connected to 2 neurons in the next layer.
  SIGNAL reg2_output: mvl_vector(ram_addr_lines DOWNTO 1);
  := "0000000000000001";

  SIGNAL inc_ptr, tx, reset_ptr : BIT;
  SIGNAL en_cntr_add : BIT;
  SIGNAL cs_wgt_frv, cs_wgt_bkv, cs_sta_frv, cs_sta_bkv : BIT;
  SIGNAL rd_wr_wgt_frv, rd_wr_wgt_bkv, rd_wr_sta_frv, rd_wr_sta_bkv : BIT;

  --FOR b1: comparator USE ENTITY work.comparator(behaviour);
  --FOR b2: comparator USE ENTITY work.comparator(behaviour);
  --FOR b3: counter1 USE ENTITY work.counter1(structure);
  --FOR b4: tri_state_ram_addr USE ENTITY

```



```

-- | File Name      : exec_unit.vhd
-- | Author         : Meyer E. Nigri
-- | Date of Creation : 05/12/91
-- | Last Update     : 20/12/91
-- |-----|
-- | Description:
-- | =====
-- | This entity implements the Execution Unit of the Generic
-- | Neuron's processing element.
-- |-----|

LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY exec_unit IS
  PORT (
    a_bus      : INOUT wire_vector(data_bus_lines DOWNTO 1);
    b_bus      : INOUT wire_vector(data_bus_lines DOWNTO 1);
    z_bus      : OUT mvl_vector(data_bus_lines DOWNTO 1);
    state_rdy  : OUT BIT;
    error_rdy  : OUT BIT;
    eval_state : IN BIT;
    eval_error : IN BIT;
    eu_en_outbus : IN BIT;
    eu_en_outph : IN BIT;
    end_frwr_ph : IN BIT;
    end_bkwr_ph : IN BIT;
    eu_en_cntr_add : OUT BIT;
    eu_reset_ptr : OUT BIT;
    eu_inc_ptr   : OUT BIT;
    eu_tx_ptr    : OUT BIT;
    eu_ca_wgt_frwr : OUT BIT;
    eu_ca_wgt_bkwr : OUT BIT;
    eu_ca_sta_frwr : OUT BIT;
    eu_ca_sta_bkwr : OUT BIT;
    eu_rd_wr_wgt_frwr : OUT BIT;
    eu_rd_wr_wgt_bkwr : OUT BIT;
    eu_rd_wr_sta_frwr : OUT BIT;
  )
  -- MU Controls
  phil : IN BIT;
  lrn_rcl : IN BIT;
  load : IN BIT;
  state_rdy : OUT BIT;
  error_rdy : OUT BIT;
  eval_state : IN BIT;
end exec_unit;

ARCHITECTURE structure OF exec_unit IS
  COMPONENT eu_datapath
    PORT (
      a_bus      : INOUT wire_vector(data_bus_lines DOWNTO 1);
      b_bus      : INOUT wire_vector(data_bus_lines DOWNTO 1);
      z_bus      : OUT mvl_vector(data_bus_lines DOWNTO 1);
      wr_bus_a, rd_bus_a : IN BIT;
      wr_bus_b, rd_bus_b : IN BIT;
      aux1_ck, wr_aux1_a, reset_aux1, rd_aux1_b : IN BIT;
      aux2_ck, wr_aux2_a, rd_aux2_b : IN BIT;
      ej_ck, wr_ej_a, rd_ej_z, rd_ej_a : IN BIT;
      sj_ck, wr_sj_a, rd_sj_b, rd_sj_z, rd_sj_a : IN BIT;
      ax_ck, wr_ax_eta, wr_ax_b, rd_ax_b : IN BIT;
      acc_ck, wr_acc_a, preset_acc, reset_acc,
      rd_acc_a, rd_acc_b, rd_acc_alu_2 : IN BIT;
      alu_ctr : IN BIT;
      rd_alu_a : IN BIT;
      shift_ctr : IN BIT;
      rd_shift_a : IN BIT;
      mr_ck, wr_mr_shift, wr_mr_a, rd_mr_shift : IN BIT;
      mr_lsb, mr_lsb_1 : OUT mvl;
      cs_rom, rd_rom : IN BIT;
      rd_zero_b : IN BIT;
    )
  END COMPONENT;

  COMPONENT eu_control
    PORT (
      phil : IN BIT;
      reset : IN BIT;
      lrn_rcl : IN BIT;
      load : IN BIT;
      state_rdy : OUT BIT;
      error_rdy : OUT BIT;
      eval_state : IN BIT;
    )
  END COMPONENT;
end structure;

```

```

eval_error : IN BIT;
en_outbus : IN BIT;
end_fr_w_ph : IN BIT;
end_bkw_ph : IN BIT;
en_cntr_add : OUT BIT;
inc_ptr : OUT BIT;
tx_ptr : OUT BIT;
cs_wgt_fr_w : OUT BIT;
cs_wgt_bkw : OUT BIT;
cs_sta_fr_w : OUT BIT;
cs_sta_bkw : OUT BIT;
rd_wt_wgt_fr_w : OUT BIT;
rd_wt_wgt_bkw : OUT BIT;
rd_wt_sta_fr_w : OUT BIT;
rd_wt_sta_bkw : OUT BIT;
wr_bus_a, rd_bus_a : OUT BIT;
wr_bus_b, rd_bus_b : OUT BIT;
wr_aux1_a, reset_ptr_aux1, rd_aux1_b : OUT BIT;
wr_aux2_a, rd_aux2_b : OUT BIT;
wr_ej_a, rd_ej_z, rd_ej_a : OUT BIT;
wr_sj_a, rd_sj_b, rd_sj_z, rd_sj_a : OUT BIT;
wr_ax_eta, wr_ax_b, rd_ax_b : OUT BIT;
wr_acc_a, preset_acc, reset_acc_mptr,
rd_acc_a, rd_acc_b, rd_acc_alu_2 : OUT BIT;
alu_ctr : OUT BIT;
rd_alu_a : OUT BIT;
shift_ctr : OUT BIT;
rd_shift_a : OUT BIT;
wr_mr_shift, wr_mr_a, rd_mr_shift : OUT BIT;
mr_lsb, mr_lsb_1 : IN mvl;
cs_rom, rd_rom : OUT BIT;
rd_zero_b : OUT BIT;
);
END COMPONENT;

SIGNAL wr_bus_a, rd_bus_a : BIT;
SIGNAL wr_bus_b, rd_bus_b : BIT;
SIGNAL wr_aux1_a, reset_ptr_aux1, rd_aux1_b : BIT;
SIGNAL wr_aux2_a, rd_aux2_b : BIT;
SIGNAL wr_ej_a, rd_ej_z, rd_ej_a : BIT;
SIGNAL wr_sj_a, rd_sj_b, rd_sj_z, rd_sj_a : BIT;
SIGNAL wr_ax_eta, wr_ax_b, rd_ax_b : BIT;
SIGNAL wr_acc_a, preset_acc, reset_acc_mptr,
rd_acc_a, rd_acc_b, rd_acc_alu_2 : BIT;
SIGNAL alu_ctr : BIT;
SIGNAL rd_alu_a : BIT;
SIGNAL shift_ctr : BIT;
SIGNAL rd_shift_a : BIT;
SIGNAL wr_mr_shift, wr_mr_a, rd_mr_shift : BIT;
SIGNAL mr_lsb, mr_lsb_1 : mvl;
SIGNAL cs_rom, rd_rom : BIT;
SIGNAL rd_zero_b : BIT;

FOR b1: eu_datapath USE ENTITY WORK eu_datapath(structure);
--FOR b2: eu_control USE ENTITY work.eu_control(structure);

BEGIN
b1: eu_datapath PORT MAP ( a_bus,
                           b_bus,
                           z_bus,
                           wr_bus_a, rd_bus_a,
                           wr_bus_b, rd_bus_b,
                           phil, wr_aux1_a, reset_ptr_aux1,
                           rd_aux1_b,
                           phil2, wr_aux2_a, rd_aux2_b,
                           phil2, wr_ej_a, rd_ej_z, rd_ej_a,
                           phil2, wr_sj_a, rd_sj_b, rd_sj_z,
                           rd_sj_a,
                           phil2, wr_ax_eta, wr_ax_b, rd_ax_b,
                           phil, wr_acc_a, preset_acc,
                           reset_acc_mptr,
                           rd_acc_a, rd_acc_b, rd_acc_alu_2,
                           alu_ctr,
                           rd_alu_a,
                           shift_ctr,
                           rd_shift_a,
                           phil, wr_mr_shift, wr_mr_a,
                           rd_mr_shift,
                           mr_lsb, mr_lsb_1,
                           cs_rom, rd_rom,
                           rd_zero_b
                           );
b2: eu_control PORT MAP ( phil,
                           phil2,
                           reset,
                           lra_rcl,
                           load,
                           state_rdy,
                           error_rdy,
                           eval_state,
                           eval_error,
                           eu_en_outbus,
                           end_fr_w_ph,
                           end_bkw_ph,
                           eu_en_cntr_add,
                           eu_inc_ptr,
                           eu_tx_ptr,
                           eu_cs_wgt_fr_w,
                           eu_cs_wgt_bkw,
                           eu_cs_sta_fr_w,
                           eu_cs_sta_bkw,
                           eu_rd_wt_wgt_fr_w,
                           eu_rd_wt_wgt_bkw,
                           eu_rd_wt_sta_fr_w,
                           eu_rd_wt_sta_bkw,
                           wr_bus_a, rd_bus_a,
                           wr_bus_b, rd_bus_b,
                           wr_aux1_a, reset_ptr_aux1, rd_aux1_b,
                           wr_aux2_a, rd_aux2_b,
                           wr_ej_a, rd_ej_z, rd_ej_a,
                           wr_sj_a, rd_sj_b, rd_sj_z, rd_sj_a,
                           wr_ax_eta, wr_ax_b, rd_ax_b,
                           wr_acc_a, preset_acc, reset_acc_mptr,
                           rd_acc_a, rd_acc_b, rd_acc_alu_2,
                           alu_ctr,
                           rd_alu_a,
                           shift_ctr,
                           rd_shift_a,
                           wr_mr_shift, wr_mr_a, rd_mr_shift,
                           mr_lsb, mr_lsb_1,
                           cs_rom, rd_rom,
                           rd_zero_b
                           );
eu_reset_ptr <= reset_ptr;

END structure;
-- File Name : eu_dpath.vhd
-- Author : Meyer E. Nigri
-- Date of Creation : 05/12/91
-- Last Update : 20/12/91
-- Description:
-- This entity implements the Execution Unit's datapath
-- for the Generic Neuron.

```

```

ctr_e_wr => reset_acc,
d        => int_bus_a,
ctr_d_rd => rd_acc_a,
e        => int_bus_b,
ctr_e_rd => rd_acc_b,
f        => alu_oper2,
ctr_f_rd => rd_acc_alu_2,
ck       => acc_ck
);

alu_block: alu PORT MAP ( oper1 => int_bus_b,
                           oper2 => alu_oper2,
                           result => alu_result,
                           control => alu_ctr
);

rd_alu_res: tri_state_data_bus PORT MAP ( inp_3_sta =>
alu_result,
                                           enable   => rd_alu_a,
                                           out_3_sta => int_bus_a
);

shift_alu: shift_n PORT MAP ( d1    => multiplier_out,
                               d2    => alu_result,
                               q1    => lsb_shift_out,
                               q2    => msb_shift_out,
                               ck    => shift_ctr
);

rd_shift: tri_state_data_bus PORT MAP ( inp_3_sta =>
msb_shift_out,
                                           enable   => rd_shift_a,
                                           out_3_sta => int_bus_a
);

reg_mr: reg2x1mr PORT MAP ( a      => lsb_shift_out,
                             ctr_a_wr => wr_mr_shift,
                             b      => int_bus_a,
                             ctr_b_wr => wr_mr_a,
                             c      => multiplier_out,
                             ctr_c_rd => rd_mr_shift,
                             lsb    => mr_lsb,
                             lsb_1  => mr_lsb_1,
                             ck     => mr_ck
);

rom_block: rom PORT MAP ( addr => int_bus_b,
                          data  => int_bus_a,
                          rd    => rd_rom,
                          cs    => cs_rom
);

wr_zero_bus_b: tri_state_rom_addr PORT MAP
( inp_3_sta => gnd_bus,
  enable   => rd_zero_b,
  out_3_sta => int_bus_b
);

pullup_bus: one_16 PORT MAP (one_16 => one_bus);

pulldown_bus: gnd_16 PORT MAP (gnd_16 => gnd_bus);

END structure;

-- | File Name           : eu_fsm.vhd
-- | Author              : Meyer E. Nigri
-- | Date of Creation    : 05/12/91
-- | Last Update         : 20/12/91
-- | Description:
-- | =====
-- | This entity implements the Execution Unit's FSM of the
-- | Generic Neuron's processing element.

LIBRARY xl; USE xl.xl_std.ALL; STD.textio.ALL; xl.xl_io.ALL;
USE WORK.gn_definition.ALL;

ENTITY eu_fsm IS
PORT (
    phi1      : IN BIT;
    phi2      : IN BIT;
    eval_error : IN BIT;
    eval_state : IN BIT;
    err_rdy    : OUT BIT;
    st_rdy     : OUT BIT;
    en_cntr_add : OUT BIT;
    end_fr_w_ph : IN BIT;
    end_bkw_ph : IN BIT;
    lrm_rc1    : IN BIT;
    cs_vgt_fr_v : OUT BIT;
    cs_vgt_bkv : OUT BIT;
    cs_sta_fr_v : OUT BIT;
    cs_sta_bkv : OUT BIT;
    rd_wr_vgt_fr_v : OUT BIT;
    rd_wr_vgt_bkv : OUT BIT;
    rd_wr_sta_fr_v : OUT BIT;
    rd_wr_sta_bkv : OUT BIT;
    inc_ptr      : OUT BIT;
    reset_ptr_aux1 : OUT BIT;
    inc_mptr     : OUT BIT;
    reset_acc_mptr : OUT BIT;
    end_multiply : IN BIT;
    rd_bus_a     : OUT BIT;
    wr_bus_a     : OUT BIT;
    rd_bus_b     : OUT BIT;
    wr_bus_b     : OUT BIT;
    wr_aux1_a    : OUT BIT;
    rd_aux1_b    : OUT BIT;
    wr_aux2_a    : OUT BIT;
    rd_aux2_b    : OUT BIT;
    wr_ej_a      : OUT BIT;
    rd_ej_s      : OUT BIT;
    rd_ej_a      : OUT BIT;
    wr_sj_a      : OUT BIT;
    rd_sj_b      : OUT BIT;
    rd_sj_s      : OUT BIT;
    rd_sj_a      : OUT BIT;
    wr_ax_eta    : OUT BIT;
    wr_ax_b      : OUT BIT;
    wr_acc_a     : OUT BIT;
    preset_acc   : OUT BIT;
    rd_acc_a     : OUT BIT;
    rd_acc_b     : OUT BIT;
    rd_acc_alu_2 : OUT BIT;
    alu_ctr      : OUT BIT;
    multiply     : OUT BIT;
    rd_alu_a     : OUT BIT;
    shift_ctr    : OUT BIT;
    rd_shift_a   : OUT BIT;
    wr_mr_shift  : OUT BIT;
    wr_mr_a      : OUT BIT;
    rd_mr_shift  : OUT BIT;
    cs_rom       : OUT BIT;
    rd_rom       : OUT BIT
);

-- +--- Mem. Unit Controls ---+
-- |                           |
-- |                           |
-- |                           |
-- +--- Mem Addressing Cntrl Controls ---+
-- | Multiple Counter Controls |
-- |                           |
-- +--- BUS A Controls ---+
-- | +--- +--- +--- +--- +--- |
-- | +--- BUS B Controls ---+
-- | +--- AUX1 Controls ---+
-- | +--- AUX2 Controls ---+
-- | +--- Ej Controls ---+
-- | |                       |
-- | +--- Sj Controls ---+
-- | |                       |
-- | +--- +--- +--- +--- +--- |
-- | +--- AX Controls ---+
-- | |                       |
-- | +--- ACC Controls ---+
-- | |                       |
-- | +--- +--- +--- +--- +--- |
-- +--- SHIFT Control ---+
-- | +--- MR Controls ---+
-- | |                       |
-- | +--- ROM Controls ---+
-- | +--- +--- +--- +--- +--- |

);
END eu_fsm;

ARCHITECTURE behaviour OF eu_fsm IS
```



```

SIGNAL current_state: CHARACTER := 'A';
SIGNAL next_state: CHARACTER := 'A';

BEGIN
  debug_fam: PROCESS -- (phil, phi2)
  VARIABLE l: line;
  BEGIN
    --WAIT UNTIL phil'event OR phi2'event;
    WAIT UNTIL phi2'EVENT AND phi2 = '1';
    write (l, "Time is ");
    write (l, NOW, RIGHT, 2);
    write (l, " EU is in State ");
    write (l, current_state, RIGHT, 2);
    writeline (output, l);
  END PROCESS debug_fam;

  synch: PROCESS
  VARIABLE flag: BOOLEAN := FALSE;
  BEGIN
    WAIT UNTIL phi2'EVENT AND phi2 = '1';
    current_state <= next_state;
  END PROCESS synch;

  finite_state_machine: PROCESS
  VARIABLE flag: BOOLEAN := FALSE;
  VARIABLE l: line;
  PROCEDURE carry_on_mult_mr_ax IS
  BEGIN
    WAIT UNTIL phi2 = '1';
    multiply <= '1', '0' AFTER tpw;
    rd_acc_alu_2 <= '1', '0' AFTER tpw; -- ALU(2) <- ACC
    -- The 2 lines below are now implemented in the structure
    level.
    --rd_ax_b <= (mr_lsb XOR mr_lsb_1), '0' AFTER Tpw;
    --rd_zero_b <= not (mr_lsb XOR mr_lsb_1), '0' AFTER Tpw;
    rd_mr_shift <= '1', '0' AFTER tpw; -- shift(1) <- MR
    shift_ctr <= '1', '0' AFTER tpw;
    inc_mptr <= '1', '0' AFTER tpw;
    -- Do this during the next phil:
    rd_shift_a <= '1' AFTER tpw+tps;
    -- '0' AFTER tpw+tps+tpw; -- ACC <- shift(2)
    wr_acc_a <= '1' AFTER tpw+tps, '0' AFTER tpw+tps+tpw;
    wr_mr_shift <= '1' AFTER tpw+tps, --MR <- shift(1)
    -- '0' AFTER tpw+tps+tpw;
  END carry_on_mult_mr_ax;

  BEGIN
    -- Every test in this process will be taken during phil,
    -- while the actions performed will be taken during phi2.
    WAIT UNTIL phil'EVENT AND phil = '1';
    WAIT FOR 1 ns; -- Timing problem between EU and CU.
    CASE current_state IS
      WHEN 'A' =>
        IF (reset = '1') THEN
          next_state <= 'A';
          en_cntr_add <= '0';
          cs_wgt_frw <= '0';
          cs_wgt_bkw <= '0';
          cs_sta_frw <= '0';
          cs_sta_bkw <= '0';
          rd_wr_wgt_frw <= '0';
          rd_wr_wgt_bkw <= '0';
          rd_wr_sta_frw <= '0';
          rd_wr_sta_bkw <= '0';
          inc_ptr <= '0';
          reset_ptr_aux1 <= '0';
          inc_mptr <= '0';
          reset_acc_mptr <= '0';
          rd_bus_a <= '0';
          wr_bus_a <= '0';
          rd_bus_b <= '0';
          wr_bus_b <= '0';
          wr_aux1_a <= '0';
          rd_aux1_b <= '0';
          wr_aux2_a <= '0';
          rd_aux2_b <= '0';
          wr_ej_a <= '0';
          rd_ej_z <= '0';
          rd_ej_a <= '0';
          wr_ej_b <= '0';
          rd_ej_z <= '0';
          rd_ej_a <= '0';
          wr_ej_a <= '0';
          wr_ax_eta <= '0';
          wr_ax_b <= '0';
          wr_acc_a <= '0';
          preset_acc <= '0';
          rd_acc_a <= '0';
          rd_acc_b <= '0';
          rd_acc_alu_2 <= '0';
          alu_ctr <= '0';
          multiply <= '0';
          rd_alu_a <= '0';
          shift_ctr <= '0';
          rd_shift_a <= '0';
          wr_mr_shift <= '0';
          wr_mr_a <= '0';
          rd_mr_shift <= '0';
          cs_rom <= '0';
          rd_rom <= '0';
        ELSIF (eval_state = '1') THEN
          next_state <= 'B';
          WAIT UNTIL phi2 = '1';
          reset_ptr_aux1 <= '1', '0' AFTER tpw; -- AUX1 <- 0
        ELSE
          next_state <= 'A';
        END IF;
      WHEN 'B' =>
        -- Prepare registers for
        multiplication
        next_state <= 'C';
        WAIT UNTIL phi2 = '1';
        reset_acc_mptr <= '1', '0' AFTER tpw; -- Address is 0
        en_cntr_add <= '1', '0' AFTER tpw;
        cs_sta_frw <= '1' AFTER 1 ns, '0' AFTER tpw; -- MR <- S
        rd_wr_sta_frw <= '1', '0' AFTER tpw;
        rd_bus_a <= '1', '0' AFTER tpw;
        wr_mr_a <= '1', '0' AFTER tpw;
        cs_wgt_frw <= '1', '0' AFTER tpw; -- AX <- Wf
        rd_wr_wgt_frw <= '1', '0' AFTER tpw;
        rd_bus_b <= '1', '0' AFTER tpw;
        wr_ax_b <= '1', '0' AFTER tpw;
      WHEN 'C' =>
        -- Multiply MR * AX
        IF (end_multiply = '1') THEN
          next_state <= 'D';
        ELSE
          carry_on_mult_mr_ax;
          next_state <= 'C';
        END IF;
      WHEN 'D' =>
        -- Accumulate S * W
        flag := FALSE;
        IF (end_frw_ph = '1') THEN
          next_state <= 'E';
        ELSE
          next_state <= 'B';
          flag := TRUE;
        END IF;
        END IF;
        WAIT UNTIL phi2 = '1';
        IF flag THEN
          inc_ptr <= '1', '0' AFTER tpw;
        END IF;
        rd_aux1_b <= '1', '0' AFTER tpw; -- AUX1 <- AUX1 + ACC
        rd_acc_alu_2 <= '1', '0' AFTER tpw;
        rd_alu_a <= '1', '0' AFTER tpw;
        wr_aux1_a <= '1', '0' AFTER tpw;
        --test only, remove this...
        --wr_bus_a <= '1', '0' AFTER Tpw;
        --wr_bus_b <= '1', '0' AFTER Tpw;
      WHEN 'E' =>
        -- Calculate Threshold
        flag := FALSE;
        IF (lra_rcl = '1') THEN -- This test is done in phil,
          next_state <= 'F'; -- That's why we need a flag
          flag := TRUE;
        ELSE
          next_state <= 'A';
        END IF;
        WAIT UNTIL phi2 = '1';
        IF flag THEN
          preset_acc <= '1',
            '0' AFTER tpw; -- Prepare ACC to do 1 - Sj in
          reset_ptr_aux1 <= '1',
            '0' AFTER tpw; -- the next state...
        END IF;
        rd_aux1_b <= '1',
          '0' AFTER tpw; -- Sj <- threshold(aux1)
        cs_rom <= '1', '0' AFTER tpw;
        rd_rom <= '1', '0' AFTER tpw;
        wr_sj_a <= '1', '0' AFTER tpw;
        rd_sj_a <= '1', '0' AFTER tpw; -- send Sj out
        st_rdy <= '1', '0' AFTER tpw;
        --test only, remove this...
        --wr_bus_a <= '1', '0' AFTER Tpw;
        --wr_bus_b <= '1', '0' AFTER Tpw;
      WHEN 'F' =>
        -- ACC <- 1 - Sj (ACC = 1)
        next_state <= 'f';
        WAIT UNTIL phi2 = '1';
        rd_acc_alu_2 <= '1', '0' AFTER tpw;
        rd_sj_b <= '1', '0' AFTER tpw;
        rd_alu_a <= '1', '0' AFTER tpw;
        wr_acc_a <= '1', '0' AFTER tpw;
        alu_ctr <= '1',
          '0' AFTER tpw; -- Indicates it is a Sub. OP.
        --test only, remove this...
        --wr_bus_a <= '1', '0' AFTER Tpw;
        --wr_bus_b <= '1', '0' AFTER Tpw;
      WHEN 'f' =>
        -- Prepare registers for multiplication
        next_state <= 'G';
        WAIT UNTIL phi2 = '1';
        reset_acc_mptr <= '1', '0' AFTER tpw; -- ACC <- 0
        rd_acc_b <= '1', '0' AFTER tpw; -- AX <- ACC (old)
        wr_ax_b <= '1', '0' AFTER tpw;
        rd_sj_a <= '1', '0' AFTER tpw; -- MR <- Sj
        wr_mr_a <= '1', '0' AFTER tpw;
        --test only, remove this...
        --wr_bus_a <= '1', '0' AFTER Tpw;
        --wr_bus_b <= '1', '0' AFTER Tpw;
      WHEN 'G' =>
        -- Multiply Sj * AX (AX=1-Sj)
        IF (end_multiply = '1') THEN
          next_state <= 'H';
        ELSE
          carry_on_mult_mr_ax;
          next_state <= 'G';
        END IF;
      WHEN 'H' =>
        -- Ej <- ACC
        next_state <= 'I';
        WAIT UNTIL phi2 = '1';
        rd_acc_a <= '1', '0' AFTER tpw;
        wr_ej_a <= '1', '0' AFTER tpw;
        --test only, remove this...
        --wr_bus_a <= '1', '0' AFTER Tpw;
        --wr_bus_b <= '1', '0' AFTER Tpw;
      WHEN 'I' =>
        -- Prepare registers for multiplication
        next_state <= 'J';
        WAIT UNTIL phi2 = '1';
        rd_sj_a <= '1', '0' AFTER tpw; -- MR <- Sj
        wr_mr_a <= '1', '0' AFTER tpw;
        wr_ax_eta <= '1', '0' AFTER tpw; -- AX <- eta
        reset_acc_mptr <= '1', '0' AFTER tpw; -- ACC <- 0
        --test only, remove this...
        --wr_bus_a <= '1', '0' AFTER Tpw;
        --wr_bus_b <= '1', '0' AFTER Tpw;
      WHEN 'J' =>
        -- Multiply eta * Sj
        IF (end_multiply = '1') THEN
          next_state <= 'L';
        ELSE
          carry_on_mult_mr_ax;
          next_state <= 'J';
        END IF;
      WHEN 'L' =>
        -- AUX2 <- ACC
        IF (eval_error = '1') THEN
          next_state <= 'N';
        ELSE
          next_state <= 'L';
        END IF;
        WAIT UNTIL phi2 = '1';
        rd_acc_a <= '1', '0' AFTER tpw;
        wr_aux2_a <= '1', '0' AFTER tpw;
        --test only, remove this...
        --wr_bus_a <= '1', '0' AFTER Tpw;

```

```

--wr_bus_b <= '1', '0' AFTER Tpw;

WHEN 'M' =>
multiplication
    next_state <= 'N';
    WAIT UNTIL phi2 = '1';
    en_cntr_add <= '1', '0' AFTER tpw;
    cs_sta_bkw <= '1', '0' AFTER tpw; -- MR <- Sb
    rd_wr_sta_bkw <= '1', '0' AFTER tpw;
    rd_bus_a <= '1', '0' AFTER tpw;
    wr_mr_a <= '1', '0' AFTER tpw;
    rd_aux2_b <= '1', '0' AFTER tpw; -- AX <- AUX2
    wr_ax_b <= '1', '0' AFTER tpw;
    reset_acc_mptr <= '1', '0' AFTER tpw;

--test only, remove this...

--wr_bus_b <= '1', '0' AFTER Tpw;

WHEN 'N' =>
    IF (end_multiply = '1') THEN
        next_state <= 'O';
    ELSE
        carry_on_mult_mr_ax;
        next_state <= 'N';
    END IF;

WHEN 'O' =>
    next_state <= 'P';
    WAIT UNTIL phi2 = '1';
    en_cntr_add <= '1', '0' AFTER tpw;
    rd_wr_wgt_bkw <= '1', '0' AFTER tpw;
    cs_wgt_bkw <= '1', '0' AFTER tpw;
    rd_bus_b <= '1', '0' AFTER tpw;
    rd_acc_alu_2 <= '1', '0' AFTER tpw;
    rd_alu_a <= '1', '0' AFTER tpw;
    wr_acc_a <= '1', '0' AFTER tpw;

--test only, remove this...

--wr_bus_a <= '1', '0' AFTER Tpw;

WHEN 'P' =>
multiplication
    next_state <= 'Q';
    WAIT UNTIL phi2 = '1';
    en_cntr_add <= '1', '0' AFTER tpw;
    cs_wgt_bkw <= '1', '0' AFTER tpw; -- Wb <- ACC (old)
    -- rd_wr is ZERO already
    wr_bus_b <= '1', '0' AFTER tpw;
    rd_acc_b <= '1', '0' AFTER tpw;
    reset_acc_mptr <= '1', '0' AFTER tpw; -- ACC <- 0
    wr_ax_b <= '1', '0' AFTER tpw; -- AX <- ACC(old)
    rd_wr_sta_bkw <= '1', '0' AFTER tpw;
    cs_sta_bkw <= '1', '0' AFTER tpw; -- MR <- Sb
    rd_bus_a <= '1', '0' AFTER tpw;
    wr_mr_a <= '1', '0' AFTER tpw;

WHEN 'Q' =>
    IF (end_multiply = '1') THEN
        next_state <= 'R';
    ELSE
        carry_on_mult_mr_ax;
        next_state <= 'Q';
    END IF;

WHEN 'R' =>
    flag := FALSE;
    IF (end_bkw_ph = '1') THEN
        next_state <= 'S';
    ELSE
        next_state <= 'M';
        flag := TRUE;
    END IF;
    WAIT UNTIL phi2 = '1';
    IF flag THEN
        inc_ptr <= '1', '0' AFTER tpw;
    END IF;
    rd_aux1_b <= '1', '0' AFTER tpw;
    rd_acc_alu_2 <= '1', '0' AFTER tpw;
    rd_alu_a <= '1', '0' AFTER tpw;
    wr_aux1_a <= '1', '0' AFTER tpw;

--test only, remove this...

--wr_bus_a <= '1', '0' AFTER Tpw;
--wr_bus_b <= '1', '0' AFTER Tpw;

WHEN 'S' =>
    next_state <= 'T';
    WAIT UNTIL phi2 = '1';
    rd_ej_a <= '1', '0' AFTER tpw; -- MR <- Ej [S * (1 - S)]
    wr_mr_a <= '1', '0' AFTER tpw;
    rd_aux1_b <= '1', '0' AFTER tpw; -- AX <- AUX1
    wr_ax_b <= '1', '0' AFTER tpw;
    reset_acc_mptr <= '1', '0' AFTER tpw; -- ACC <- 0

--test only, remove this...

--wr_bus_a <= '1', '0' AFTER Tpw;
--wr_bus_b <= '1', '0' AFTER Tpw;

WHEN 'T' =>
    IF (end_multiply = '1') THEN
        next_state <= 'U';
    ELSE
        carry_on_mult_mr_ax;
        next_state <= 'T';
    END IF;

WHEN 'U' =>
    next_state <= 'V';
    WAIT UNTIL phi2 = '1';
    rd_acc_a <= '1', '0' AFTER tpw; -- Ej <- ACC
    wr_ej_a <= '1', '0' AFTER tpw;
    reset_ptr_aux1 <= '1', '0' AFTER tpw;
    wr_rdy <= '1', '0' AFTER tpw;
    rd_ej_s <= '1', '0' AFTER tpw;

--test only, remove this...

--wr_bus_a <= '1', '0' AFTER Tpw;
--wr_bus_b <= '1', '0' AFTER Tpw;

WHEN 'V' =>
    next_state <= 'W';
    WAIT UNTIL phi2 = '1';
    rd_ej_a <= '1', '0' AFTER tpw; -- MR <- Ej
    wr_mr_a <= '1', '0' AFTER tpw;
    wr_ax_eta <= '1', '0' AFTER tpw; -- AX <- eta
    reset_acc_mptr <= '1', '0' AFTER tpw; -- ACC <- 0

--test only, remove this...

--wr_bus_a <= '1', '0' AFTER Tpw;
--wr_bus_b <= '1', '0' AFTER Tpw;

WHEN 'W' =>
    IF (end_multiply = '1') THEN
        next_state <= 'X';
    ELSE
        carry_on_mult_mr_ax;
        next_state <= 'W';
    END IF;

WHEN 'X' =>
    next_state <= 'Y';
    WAIT UNTIL phi2 = '1';
    rd_acc_a <= '1', '0' AFTER tpw;
    wr_aux2_a <= '1', '0' AFTER tpw;

--test only, remove this...

--wr_bus_a <= '1', '0' AFTER Tpw;
--wr_bus_b <= '1', '0' AFTER Tpw;

WHEN 'Y' =>
    next_state <= 'Z';
    WAIT UNTIL phi2 = '1';
    en_cntr_add <= '1', '0' AFTER tpw;
    cs_sta_fr_w <= '1', '0' AFTER tpw; -- MR <- Sf
    rd_wr_sta_fr_w <= '1', '0' AFTER tpw;
    rd_bus_a <= '1', '0' AFTER tpw;
    wr_mr_a <= '1', '0' AFTER tpw;
    rd_aux2_b <= '1', '0' AFTER tpw; -- AX <- AUX2
    wr_ax_b <= '1', '0' AFTER tpw;
    reset_acc_mptr <= '1', '0' AFTER tpw;

--test only, remove this...

--wr_bus_b <= '1', '0' AFTER Tpw;

WHEN 'Z' =>
    IF (end_multiply = '1') THEN
        next_state <= 'a';
    ELSE
        carry_on_mult_mr_ax;
        next_state <= 'Z';
    END IF;

WHEN 'a' =>
    next_state <= 'b';
    WAIT UNTIL phi2 = '1';
    en_cntr_add <= '1', '0' AFTER tpw;
    cs_wgt_fr_w <= '1', '0' AFTER tpw;
    rd_wr_wgt_fr_w <= '1', '0' AFTER tpw;
    rd_bus_b <= '1', '0' AFTER tpw;
    rd_acc_alu_2 <= '1', '0' AFTER tpw;
    rd_alu_a <= '1', '0' AFTER tpw;
    wr_acc_a <= '1', '0' AFTER tpw;

--test only, remove this...

--wr_bus_a <= '1', '0' AFTER Tpw;

WHEN 'b' =>
    flag := FALSE;
    IF (end_fr_w_ph = '1') THEN
        next_state <= 'A';
    ELSE
        next_state <= 'Y';
        flag := TRUE;
    END IF;
    WAIT UNTIL phi2 = '1';
    IF flag THEN
        inc_ptr <= '1', '0' AFTER tpw;
    END IF;
    en_cntr_add <= '1', '0' AFTER tpw;
    cs_wgt_fr_w <= '1', '0' AFTER tpw; -- Wf <- ACC
    -- rd_wr is already '0' for write
    wr_bus_b <= '1', '0' AFTER tpw;
    rd_acc_b <= '1', '0' AFTER tpw;

--test only, remove this...

--wr_bus_a <= '1', '0' AFTER Tpw;

END CASE;
END PROCESS finite_state_machine;

--
-- File Name      : eu_ctr.vhd
-- Author         : Meyer E. Nigri
-- Date of Creation : 05/12/91
-- Last Update    : 20/12/91
--
-- Description:
-- =====
-- This entity implements the Execution Unit's control part
-- of the Generic Neuron's processing element.
--

LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY eu_control IS
    PORT (
        phi1      : IN BIT;
        phi2      : IN BIT;
        reset      : IN BIT;
        lrn_rcl    : IN BIT;
        load       : IN BIT;
        state_rdy  : OUT BIT;
        error_rdy  : OUT BIT;
        eval_state : IN BIT;
        eval_error : IN BIT;
        en_outbus  : IN BIT;
        end_fr_w_ph : IN BIT;
        end_bkw_ph : IN BIT;
        en_cntr_add : OUT BIT;
        inc_ptr    : OUT BIT;
        tx_ptr     : OUT BIT;
        cs_wgt_fr_w : OUT BIT; -- +--- Mem. Unit Controls ---+
        cs_wgt_bkw : OUT BIT; -- |
        cs_sta_fr_w : OUT BIT; -- |
        cs_sta_bkw  : OUT BIT; -- |
        rd_wr_wgt_fr_w : OUT BIT; -- |
        rd_wr_wgt_bkw : OUT BIT; -- |
        rd_wr_sta_fr_w : OUT BIT; -- |
        rd_wr_sta_bkw : OUT BIT; -- |
        wr_bus_a, rd_bus_a : OUT BIT;
        wr_bus_b, rd_bus_b : OUT BIT;
        wr_aux1_a, reset_ptr_aux1, rd_aux1_b : OUT BIT;
        wr_aux2_a, rd_aux2_b : OUT BIT;
        wr_ej_a, rd_ej_a, rd_ej_s : OUT BIT;
    );
END ENTITY eu_control;

```



```

cs_sta_frw      => cs_sta_frw,
cs_sta_bkw      => cs_sta_bkw,
rd_wr_wgt_frw   => rd_wr_wgt_frw,
rd_wr_wgt_bkw   => rd_wr_wgt_bkw,
rd_wr_sta_frw   => rd_wr_sta_frw,
rd_wr_sta_bkw   => rd_wr_sta_bkw,
inc_ptr          => inc_ptr,
reset_ptr_aux1  => reset_ptr_aux1,
inc_mptr        => int_inc_mptr,
reset_acc_mptr  => int_reset_mptr,
end_multiply    => int_end_multiply,
rd_bus_a        => rd_bus_a,
wr_bus_a        => wr_bus_a,
rd_bus_b        => rd_bus_b,
wr_bus_b        => wr_bus_b,
--ck_aux1       => aux1_ck,
wr_aux1_a       => wr_aux1_a,
rd_aux1_b       => rd_aux1_b,
--ck_aux2       => aux2_ck,
wr_aux2_a       => wr_aux2_a,
rd_aux2_b       => rd_aux2_b,
--ck_ej         => ej_ck,
wr_ej_a         => wr_ej_a,
rd_ej_z         => rd_ej_z,
rd_ej_a         => rd_ej_a,
--ck_sj         => sj_ck,
wr_sj_a         => wr_sj_a,
rd_sj_b         => rd_sj_b,
rd_sj_z         => rd_sj_z,
rd_sj_a         => rd_sj_a,
--ck_ax         => ax_ck,
wr_ax_eta       => wr_ax_eta,
--rd_ax_b       => wr_ax_b, rd_ax_b,
--ck_acc        => acc_ck,
wr_acc_a        => wr_acc_a,
preset_acc      => preset_acc,
rd_acc_a        => rd_acc_a,
rd_acc_b        => rd_acc_b,
rd_acc_alu_2    => rd_acc_alu_2,
alu_ctr         => int_alu_ctr,
multiply        => int_multiply,
rd_alu_a        => rd_alu_a,
shift_ctr       => shift_ctr,
rd_shift_a      => rd_shift_a,
--mr_lsb        => mr_lsb_ff,
--mr_lsb_l      => mr_lsb_l_ff,
--ck_mr         => mr_ck,
wr_mr_shift     => wr_mr_shift,
wr_mr_a         => wr_mr_a,
rd_mr_shift     => rd_mr_shift,
cs_rom          => cs_rom,
rd_rom          => rd_rom,
--rd_zero_b     => rd_zero_b,
);

comb_block: eu_combinational PORT MAP
(
  phil          => phil,
  phi2         => phi2,
  load         => load,
  en_outbus    => en_outbus,
  err_rdy      => int_err_rdy,
  error_rdy    => error_rdy,
  st_rdy       => int_st_rdy,
  state_rdy    => state_rdy,
  end_multiply => int_end_multiply,
  reset_mptr   => int_reset_mptr,
  inc_mptr     => int_inc_mptr,
  multiply     => int_multiply,
  alu_ctr_in   => int_alu_ctr,
  alu_ctr      => alu_ctr,
  mr_lsb       => mr_lsb,
  mr_lsb_l     => mr_lsb_l,
  rd_zero_b    => rd_zero_b,
  rd_ax_b      => rd_ax_b,
  tx_ptr       => tx_ptr,
);

reset_acc_mptr <= int_reset_mptr;

END structure;
-- | File Name      : eu_comb.vhd
-- | Author        : Meyer E. Nigri
-- | Date of Creation : 05/12/91
-- | Last Update    : 20/12/91
-- | Description:
-- | *****
-- | This entity implements the Execution Unit's combinational
-- | logic of the Generic Neuron's processing element.
-- | *****

LIBRARY x1; USE x1.x1_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY eu_combinational IS
  PORT ( phil      : IN BIT;
        phi2     : IN BIT;
        load      : IN BIT;
        en_outbus : IN BIT;
        err_rdy   : IN BIT;
        error_rdy : OUT BIT;
        st_rdy    : IN BIT;
        state_rdy : OUT BIT;
        end_multiply : OUT BIT;
        reset_mptr : IN BIT;
        inc_mptr   : IN BIT;
        multiply   : IN BIT;
        alu_ctr_in : IN BIT;
        alu_ctr    : OUT BIT;
        mr_lsb     : IN BIT;
        mr_lsb_l   : IN BIT;
        rd_zero_b  : OUT BIT;
        rd_ax_b    : OUT BIT;
        tx_ptr     : OUT BIT;
        );
END eu_combinational;

ARCHITECTURE structure OF eu_combinational IS
  COMPONENT count_to_16
    PORT ( reset : IN BIT;
          ck     : IN BIT;
          count_end : OUT BIT
        );
  END COMPONENT;

  COMPONENT ff
    PORT ( d : IN BIT;
          q : OUT BIT;
          notq : OUT BIT;
          clear : IN BIT;
          preset : IN BIT;
          ck : IN BIT
        );
  END COMPONENT;

  COMPONENT and2gate

```

```

cs_sta_frw      => cs_sta_frw,
cs_sta_bkw      => cs_sta_bkw,
rd_wr_wgt_frw   => rd_wr_wgt_frw,
rd_wr_wgt_bkw   => rd_wr_wgt_bkw,
rd_wr_sta_frw   => rd_wr_sta_frw,
rd_wr_sta_bkw   => rd_wr_sta_bkw,
inc_ptr          => inc_ptr,
reset_ptr_aux1  => reset_ptr_aux1,
inc_mptr        => int_inc_mptr,
reset_acc_mptr  => int_reset_mptr,
end_multiply    => int_end_multiply,
rd_bus_a        => rd_bus_a,
wr_bus_a        => wr_bus_a,
rd_bus_b        => rd_bus_b,
wr_bus_b        => wr_bus_b,
--ck_aux1       => aux1_ck,
wr_aux1_a       => wr_aux1_a,
rd_aux1_b       => rd_aux1_b,
--ck_aux2       => aux2_ck,
wr_aux2_a       => wr_aux2_a,
rd_aux2_b       => rd_aux2_b,
--ck_ej         => ej_ck,
wr_ej_a         => wr_ej_a,
rd_ej_z         => rd_ej_z,
rd_ej_a         => rd_ej_a,
--ck_sj         => sj_ck,
wr_sj_a         => wr_sj_a,
rd_sj_b         => rd_sj_b,
rd_sj_z         => rd_sj_z,
rd_sj_a         => rd_sj_a,
--ck_ax         => ax_ck,
wr_ax_eta       => wr_ax_eta,
--rd_ax_b       => wr_ax_b, rd_ax_b,
--ck_acc        => acc_ck,
wr_acc_a        => wr_acc_a,
preset_acc      => preset_acc,
rd_acc_a        => rd_acc_a,
rd_acc_b        => rd_acc_b,
rd_acc_alu_2    => rd_acc_alu_2,
alu_ctr         => int_alu_ctr,
multiply        => int_multiply,
rd_alu_a        => rd_alu_a,
shift_ctr       => shift_ctr,
rd_shift_a      => rd_shift_a,
--mr_lsb        => mr_lsb_ff,
--mr_lsb_l      => mr_lsb_l_ff,
--ck_mr         => mr_ck,
wr_mr_shift     => wr_mr_shift,
wr_mr_a         => wr_mr_a,
rd_mr_shift     => rd_mr_shift,
cs_rom          => cs_rom,
rd_rom          => rd_rom,
--rd_zero_b     => rd_zero_b,
);

comb_block: eu_combinational PORT MAP
(
  phil          => phil,
  phi2         => phi2,
  load         => load,
  en_outbus    => en_outbus,
  err_rdy      => int_err_rdy,
  error_rdy    => error_rdy,
  st_rdy       => int_st_rdy,
  state_rdy    => state_rdy,
  end_multiply => int_end_multiply,
  reset_mptr   => int_reset_mptr,
  inc_mptr     => int_inc_mptr,
  multiply     => int_multiply,
  alu_ctr_in   => int_alu_ctr,
  alu_ctr      => alu_ctr,
  mr_lsb       => mr_lsb,
  mr_lsb_l     => mr_lsb_l,
  rd_zero_b    => rd_zero_b,
  rd_ax_b      => rd_ax_b,
  tx_ptr       => tx_ptr,
);

reset_acc_mptr <= int_reset_mptr;

END structure;
-- | File Name      : eu_comb.vhd
-- | Author        : Meyer E. Nigri
-- | Date of Creation : 05/12/91
-- | Last Update    : 20/12/91
-- | Description:
-- | *****
-- | This entity implements the Execution Unit's combinational
-- | logic of the Generic Neuron's processing element.
-- | *****

LIBRARY x1; USE x1.x1_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY eu_combinational IS
  PORT ( phil      : IN BIT;
        phi2     : IN BIT;
        load      : IN BIT;
        en_outbus : IN BIT;
        err_rdy   : IN BIT;
        error_rdy : OUT BIT;
        st_rdy    : IN BIT;
        state_rdy : OUT BIT;
        end_multiply : OUT BIT;
        reset_mptr : IN BIT;
        inc_mptr   : IN BIT;
        multiply   : IN BIT;
        alu_ctr_in : IN BIT;
        alu_ctr    : OUT BIT;
        mr_lsb     : IN BIT;
        mr_lsb_l   : IN BIT;
        rd_zero_b  : OUT BIT;
        rd_ax_b    : OUT BIT;
        tx_ptr     : OUT BIT;
        );
END eu_combinational;

ARCHITECTURE structure OF eu_combinational IS
  COMPONENT count_to_16
    PORT ( reset : IN BIT;
          ck     : IN BIT;
          count_end : OUT BIT
        );
  END COMPONENT;

  COMPONENT ff
    PORT ( d : IN BIT;
          q : OUT BIT;
          notq : OUT BIT;
          clear : IN BIT;
          preset : IN BIT;
          ck : IN BIT
        );
  END COMPONENT;

  COMPONENT and2gate

```

```

    PORT ( input1 : IN BIT;
          input2 : IN BIT;
          output : OUT BIT
        );
END COMPONENT;

COMPONENT or2gate
  PORT ( input1 : IN BIT;
        input2 : IN BIT;
        output : OUT BIT
      );
END COMPONENT;

COMPONENT xor2gate
  PORT ( input1 : IN BIT;
        input2 : IN BIT;
        output : OUT BIT
      );
END COMPONENT;

COMPONENT inverter
  PORT ( input : IN BIT;
        output : OUT BIT
      );
END COMPONENT;

COMPONENT pulldown
  PORT ( gnd : OUT BIT);
END COMPONENT;

--COMPONENT pullup
--  PORT ( vcc : OUT BIT);
--END COMPONENT;

SIGNAL gnd : BIT;
--SIGNAL vcc : BIT;
SIGNAL mr_and_lsbs, mr_phi2, mult, notmr_lsb_l_ff : BIT;
SIGNAL int_mr_lsb_ff : BIT;
SIGNAL not_load : BIT;
SIGNAL mr_lsb_l_ff : BIT;
SIGNAL int_xor : BIT;
SIGNAL int_inv2 : BIT;

BEGIN

  b1: count_to_16 PORT MAP ( reset => reset_mptr,
                             ck    => inc_mptr,
                             count_end => end_multiply
                           );

  sta_ff: FF PORT MAP ( d    => gnd,
                       q    => state_rdy,
                       notq  => OPEN,
                       clear  => en_outbus,
                       preset => st_rdy,
                       ck    => gnd
                     );

  err_ff: FF PORT MAP ( d    => gnd,
                       q    => error_rdy,
                       notq  => OPEN,
                       clear  => en_outbus,
                       preset => err_rdy,
                       ck    => gnd
                     );

  lsb_ff: FF PORT MAP ( d    => mr_lsb,
                       q    => int_mr_lsb_ff,

```

```

                             notq  => OPEN,
                             clear  => gnd,
                             preset => gnd,
                             ck    => phil
                           );

  lsb_l_ff: FF PORT MAP ( d    => mr_lsb_l,
                         q    => mr_lsb_l_ff,
                         notq  => notmr_lsb_l_ff,
                         clear  => gnd,
                         preset => gnd,
                         ck    => phil
                       );

  xor1: xor2gate PORT MAP ( input1 => int_mr_lsb_ff,
                           input2 => mr_lsb_l_ff,
                           output => int_xor
                         );

  inv2: inverter PORT MAP ( input => int_xor,
                           output => int_inv2
                         );

  and4: and2gate PORT MAP ( input1 => int_inv2,
                           input2 => multiply,
                           output => rd_zero_b
                         );

  and5: and2gate PORT MAP ( input1 => int_xor,
                           input2 => multiply,
                           output => rd_ax_b
                         );

  and1: and2gate PORT MAP ( input1 => int_mr_lsb_ff,
                           input2 => notmr_lsb_l_ff,
                           output => mr_and_lsbs
                         );

  and2: and2gate PORT MAP ( input1 => mr_and_lsbs,
                           input2 => phi2,
                           output => mr_phi2
                         );

  and3: and2gate PORT MAP ( input1 => mr_phi2,
                           input2 => multiply,
                           output => mult
                         );

  or1: or2gate PORT MAP ( input1 => mult,
                         input2 => alu_ctr_in,
                         output => alu_ctr
                       );

  notload: inverter PORT MAP ( input => load,
                              output => not_load
                            );

  nloadphil: and2gate PORT MAP ( input1 => phil,
                                input2 => not_load,
                                output => tx_ptr
                              );

  --vcc_pullup: pullup PORT MAP ( vcc => vcc);
  gnd_pulldown: pulldown PORT MAP ( gnd => gnd);

END structure;

```

Processing Element

The processing element is constructed from the three units above, by simply instantiating each one and providing the necessary connections.

```

-- | File Name      : gn.vhd
-- | Author       : Meyer E. Nigri
-- | Date of Creation : 05/12/91
-- | Last Update  : 20/12/91
-- | Description:
-- | =====
-- | This entity implements the Generic Neuron for the Hidden
-- | layer of a Back Propagation Neural Network.
-- |
-- |
LIBRARY xl; USE xl.xl_std.ALL;
USE WORK.gn_definition.ALL;

ENTITY gn_hidden IS
  PORT ( in_bus      : IN  mvl_vector(data_bus_lines DOWNTO
1);
        out_bus     : OUT  mvl_vector(data_bus_lines DOWNTO
1);
        add_bus     : IN  mvl_vector(ram_addr_lines DOWNTO
1);
        reset       : IN  BIT;
        load        : IN  BIT; -- 0=Initialisation;
1=Execution
        frw_bkw     : IN  BIT;
        phi1        : IN  BIT;
        phi2        : IN  BIT;
        cs_rdy      : INOUT wire;
        mem_add_bus : OUT  wire_vector(ram_addr_lines DOWNTO
1);
        weights_bus : OUT  wire_vector(data_bus_lines DOWNTO
1);
        states_bus  : OUT  wire_vector(data_bus_lines DOWNTO
1);
        z_bus       : IN  mvl_vector(data_bus_lines DOWNTO
1);
        cu_cs_wgt_frw : OUT BIT;
        cu_cs_wgt_bkw : OUT BIT;
        cu_cs_sta_frw : OUT BIT;
        cu_cs_sta_bkw : OUT BIT;
        cu_rd_wr_wgt_frw: OUT BIT;
        cu_rd_wr_wgt_bkw: OUT BIT;
        cu_rd_wr_sta_frw: OUT BIT;
        cu_rd_wr_sta_bkw: OUT BIT;
        cu_inc_ptr    : OUT BIT;
        cu_reset_ptr  : OUT BIT;
        eval_state    : OUT BIT;
        eval_error    : OUT BIT;
        state_rdy     : IN  BIT;
        error_rdy     : IN  BIT;
        cu_en_cntr_add : OUT BIT;
        cu_tx_ptr     : OUT BIT;
        cu_en_outbus  : OUT BIT
      );
END gn_hidden;

ARCHITECTURE structure OF gn_hidden IS
  COMPONENT comm_unit

```

```

PORT ( cu_inc_ptr      : IN BIT;
      eu_inc_ptr      : IN BIT;
      cu_tx           : IN BIT;
      eu_tx           : IN BIT;
      cu_reset_ptr    : IN BIT;
      eu_reset_ptr    : IN BIT;
      cu_en_cntr_add  : IN BIT;
      eu_en_cntr_add  : IN BIT;
      end_frwr_ph     : OUT BIT;
      end_bkw_ph      : OUT BIT;
      mem_add_bus     : INOUT wire_vector(ram_addr_lines

DOWNT0 1);
      weight_bus      : INOUT wire_vector(data_bus_lines
DOWNT0 1);
      states_bus      : INOUT wire_vector(data_bus_lines
DOWNT0 1);
      cu_cs_wgt_frwr  : IN BIT;
      eu_cs_wgt_frwr  : IN BIT;
      cu_rd_wr_wgt_frwr : IN BIT;
      eu_rd_wr_wgt_frwr : IN BIT;
      cu_cs_wgt_bkw   : IN BIT;
      eu_cs_wgt_bkw   : IN BIT;
      cu_rd_wr_wgt_bkw : IN BIT;
      eu_rd_wr_wgt_bkw : IN BIT;
      cu_cs_sta_frwr  : IN BIT;
      eu_cs_sta_frwr  : IN BIT;
      cu_rd_wr_sta_frwr : IN BIT;
      eu_rd_wr_sta_frwr : IN BIT;
      cu_cs_sta_bkw   : IN BIT;
      eu_cs_sta_bkw   : IN BIT;
      cu_rd_wr_sta_bkw : IN BIT;
      eu_rd_wr_sta_bkw : IN BIT
    );
END COMPONENT;

COMPONENT exec_unit
  PORT ( a_bus      : INOUT wire_vector(data_bus_lines
DOWNT0 1);
        b_bus      : INOUT wire_vector(data_bus_lines
DOWNT0 1);
        z_bus      : OUT mvl_vector(data_bus_lines
DOWNT0 1);
        state_rdy   : OUT BIT;
        error_rdy   : OUT BIT;
        eval_state   : IN BIT;
        eval_error   : IN BIT;
        eu_en_outbus : IN BIT;
        end_frwr_ph  : IN BIT;
        end_bkw_ph   : IN BIT;
        eu_en_cntr_add : OUT BIT;
        eu_reset_ptr  : OUT BIT;
        eu_inc_ptr    : OUT BIT;
        eu_tx_ptr     : OUT BIT;
        eu_cs_wgt_frwr : OUT BIT;
        eu_cs_wgt_bkw : OUT BIT;
        eu_cs_sta_frwr : OUT BIT;
        eu_cs_sta_bkw : OUT BIT;
        eu_rd_wr_wgt_frwr : OUT BIT;
        eu_rd_wr_wgt_bkw : OUT BIT;
        eu_rd_wr_sta_frwr : OUT BIT;
        eu_rd_wr_sta_bkw : OUT BIT;
        load         : IN BIT;
        reset        : IN BIT;
        lrn_rcl      : IN BIT;
        phil         : IN BIT;
        phi2         : IN BIT
    );
END COMPONENT;

SIGNAL a_bus : wire_vector(data_bus_lines DOWNT0 1);
SIGNAL z_bus : mvl_vector(data_bus_lines DOWNT0 1);
SIGNAL mem_add_bus : wire_vector(ram_addr_lines DOWNT0 1);
SIGNAL en_outbus : BIT;
SIGNAL state_rdy, error_rdy, eval_state, eval_error : BIT;
SIGNAL cu_cs_wgt_frwr, eu_cs_wgt_frwr : BIT;
SIGNAL cu_rd_wr_wgt_frwr, eu_rd_wr_wgt_frwr : BIT;
SIGNAL cu_cs_wgt_bkw, eu_cs_wgt_bkw : BIT;
SIGNAL cu_rd_wr_wgt_bkw, eu_rd_wr_wgt_bkw : BIT;
SIGNAL cu_cs_sta_frwr, eu_cs_sta_frwr : BIT;
SIGNAL cu_rd_wr_sta_frwr, eu_rd_wr_sta_frwr : BIT;
SIGNAL cu_cs_sta_bkw, eu_cs_sta_bkw : BIT;
SIGNAL cu_rd_wr_sta_bkw, eu_rd_wr_sta_bkw : BIT;
SIGNAL cu_reset_ptr, eu_reset_ptr : BIT;
SIGNAL cu_tx_ptr, eu_tx_ptr : BIT;
SIGNAL cu_en_cntr_add, eu_en_cntr_add : BIT;
SIGNAL end_frwr_ph, end_bkw_ph : BIT;

--FOR cu_hidden: comm_unit USE ENTITY work.comm_unit(structure);
--FOR mu_hidden: mem_unit USE ENTITY work.mem_unit(structure);
--FOR eu_hidden: exec_unit USE ENTITY work.exec_unit(structure);

BEGIN

cu_hidden: comm_unit PORT MAP ( in_bus => in_bus,
                                out_bus => out_bus,
                                add_bus => add_bus,
                                reset => reset,
                                load => load,
                                frwr_bkw => frwr_bkw,
                                phil => phil,
                                phi2 => phi2,
                                cs_rdy => cs_rdy,
                                mem_add_bus => mem_add_bus,
                                weights_bus => b_bus,
                                states_bus => a_bus,
                                z_bus => z_bus,
                                cu_cs_wgt_frwr => cu_cs_wgt_frwr,
                                eu_cs_wgt_frwr => eu_cs_wgt_frwr,
                                cu_cs_wgt_bkw => cu_cs_wgt_bkw,
                                eu_cs_wgt_bkw => eu_cs_wgt_bkw,
                                cu_cs_sta_frwr => cu_cs_sta_frwr,
                                eu_cs_sta_frwr => eu_cs_sta_frwr,
                                cu_cs_sta_bkw => cu_cs_sta_bkw,
                                eu_cs_sta_bkw => eu_cs_sta_bkw,
                                cu_rd_wr_wgt_frwr =>
                                cu_rd_wr_wgt_frwr,
                                eu_rd_wr_wgt_frwr =>
                                eu_rd_wr_wgt_frwr,
                                cu_rd_wr_wgt_bkw =>
                                cu_rd_wr_wgt_bkw,
                                eu_rd_wr_wgt_bkw =>
                                eu_rd_wr_wgt_bkw,
                                cu_rd_wr_sta_frwr =>
                                cu_rd_wr_sta_frwr,
                                eu_rd_wr_sta_frwr =>
                                eu_rd_wr_sta_frwr,
                                cu_rd_wr_sta_bkw =>
                                cu_rd_wr_sta_bkw,
                                eu_rd_wr_sta_bkw =>
                                eu_rd_wr_sta_bkw,
                                cu_inc_ptr => cu_inc_ptr,
                                eu_inc_ptr => eu_inc_ptr,
                                cu_reset_ptr => cu_reset_ptr,
                                eu_reset_ptr => eu_reset_ptr,
                                eval_state => eval_state,
                                eu_tx_ptr => eu_tx_ptr,
                                eu_en_cntr_add => eu_en_cntr_add,
                                eu_tx_ptr => eu_tx_ptr,
                                eu_en_outbus => en_outbus
    );

```

```

mu_hidden: mem_unit PORT MAP ( cu_inc_ptr => cu_inc_ptr,
                                eu_inc_ptr => eu_inc_ptr,
                                cu_tx => cu_tx_ptr,
                                eu_tx => eu_tx_ptr,
                                cu_reset_ptr => cu_reset_ptr,
                                eu_reset_ptr => eu_reset_ptr,
                                cu_en_cntr_add => cu_en_cntr_add,
                                eu_en_cntr_add => eu_en_cntr_add,
                                end_frwr_ph => end_frwr_ph,
                                end_bkw_ph => end_bkw_ph,
                                mem_add_bus => mem_add_bus,
                                weight_bus => b_bus,
                                states_bus => a_bus,
                                cu_cs_wgt_frwr => cu_cs_wgt_frwr,
                                eu_cs_wgt_frwr => eu_cs_wgt_frwr,
                                cu_rd_wr_wgt_frwr =>
                                eu_rd_wr_wgt_frwr =>
                                eu_rd_wr_wgt_frwr,
                                cu_cs_wgt_bkw => cu_cs_wgt_bkw,
                                eu_cs_wgt_bkw => eu_cs_wgt_bkw,
                                cu_rd_wr_wgt_bkw =>
                                eu_rd_wr_wgt_bkw =>
                                eu_rd_wr_wgt_bkw,
                                cu_cs_sta_frwr => cu_cs_sta_frwr,
                                eu_cs_sta_frwr => eu_cs_sta_frwr,
                                cu_rd_wr_sta_frwr =>
                                eu_rd_wr_sta_frwr =>
                                eu_rd_wr_sta_frwr,
                                cu_cs_sta_bkw => cu_cs_sta_bkw,
                                eu_cs_sta_bkw => eu_cs_sta_bkw,
                                cu_rd_wr_sta_bkw =>
                                eu_rd_wr_sta_bkw =>
                                eu_rd_wr_sta_bkw,
                                eu_rd_wr_sta_bkw =>
                                eu_rd_wr_sta_bkw
    );

eu_hidden: exec_unit PORT MAP ( a_bus => a_bus,
                                b_bus => b_bus,
                                z_bus => z_bus,
                                state_rdy => state_rdy,
                                error_rdy => error_rdy,
                                eval_state => eval_state,
                                eval_error => eval_error,
                                eu_en_outbus => eu_en_outbus,
                                end_frwr_ph => end_frwr_ph,
                                end_bkw_ph => end_bkw_ph,
                                eu_en_cntr_add => eu_en_cntr_add,
                                eu_reset_ptr => eu_reset_ptr,
                                eu_inc_ptr => eu_inc_ptr,
                                eu_tx_ptr => eu_tx_ptr,
                                eu_cs_wgt_frwr => eu_cs_wgt_frwr,
                                eu_cs_wgt_bkw => eu_cs_wgt_bkw,
                                eu_cs_sta_frwr => eu_cs_sta_frwr,
                                eu_cs_sta_bkw => eu_cs_sta_bkw,
                                eu_rd_wr_wgt_frwr =>
                                eu_rd_wr_wgt_frwr =>
                                eu_rd_wr_wgt_frwr,
                                eu_rd_wr_wgt_bkw =>
                                eu_rd_wr_wgt_bkw =>
                                eu_rd_wr_wgt_bkw,
                                eu_rd_wr_sta_frwr =>
                                eu_rd_wr_sta_frwr =>
                                eu_rd_wr_sta_frwr,
                                eu_rd_wr_sta_bkw =>
                                eu_rd_wr_sta_bkw =>
                                eu_rd_wr_sta_bkw,
                                load => load,
                                reset => reset,
                                lrn_rcl => lrn_rcl,
                                phil => phil,
                                phi2 => phi2
    );

END structure;

```

Appendix E

List of Abbreviations

<i>AFAP</i>	<i>As Fast As Possible</i>
<i>ALAP</i>	<i>As Late As Possible</i>
<i>ASAP</i>	<i>As Soon As Possible</i>
<i>ASIC</i>	<i>Application Specific Integrated Circuit</i>
<i>ASNNC</i>	<i>Application Specific Neural Network Chip</i>
<i>ATR</i>	<i>Automatic Target Recognition</i>
<i>CDFG</i>	<i>Control and Data Flow Graph</i>
<i>CFG</i>	<i>Control Flow Graph</i>
<i>DFG</i>	<i>Data Flow Graph</i>
<i>DSP</i>	<i>Digital Signal Processing</i>
<i>EDIF</i>	<i>Electronic Design Interchange</i>
<i>ETANN</i>	<i>Electrically Trainable Analogue Neural Network</i>
<i>FPGA</i>	<i>Field Programmable Gate Arrays</i>
<i>FSM</i>	<i>Finite State Machine</i>
<i>HDL</i>	<i>Hardware Description Language</i>
<i>ICR</i>	<i>Intermediate Code Representation</i>
<i>MSB</i>	<i>Most Significant Byte</i>
<i>NSC</i>	<i>Neural Silicon Compiler</i>
<i>OCR</i>	<i>Optical Character Recognition</i>

<i>PE</i>	<i>Processing Element</i>
<i>RISC</i>	<i>Reduced Instruction Set Computer</i>
<i>RTL</i>	<i>Register Transfer Level</i>
<i>SIF</i>	<i>Sequencing Intermediate Form</i>
<i>SLM</i>	<i>Spatial Light Modulators</i>
<i>SSIM</i>	<i>Sequential Synthesis In-Core Model</i>
<i>UDL/I</i>	<i>Unified Design Language for Integrated Circuits</i>
<i>VHDL</i>	<i><u>V</u>HSIC Hardware Description Language</i>
<i>VHSIC</i>	<i>Very High Speed Integrated Circuits</i>
<i>VLSI</i>	<i>Very Large Scale Integration</i>
<i>VML</i>	<i>Virtual Machine Language</i>
<i>WSI</i>	<i>Wafer Scale Integration</i>
<i>YIF</i>	<i>Yorktown Internal Format</i>
<i>YSC</i>	<i>Yorktown Silicon Compiler</i>