National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

Architectural Synthesis for VLSI Neural Networks

Elie Torbey

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science at
Concordia University
Montreal, Quebec, Canada

August, 1992

National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

ISBN 0-315-80928 0

Canada

Abstract

Architectural Synthesis for VLSI Neural Networks

Elie Torbey

The growing use of artificial neural networks for real time adaptive applications in robotics, signal and image processing, creates the need for VLSI ASIC designs that satisfy response time and silicon area constraints. Using high level synthesis techniques, the design of digital ASIC implementations of neural networks can be automated, thus reducing the design cycle while extensively searching the design space for optimal architectures. This thesis presents a synthesis methodology for the automated design of single and multi-chip processors implementing neural networks. Special, effective heuristics and synthesis algorithms specific to neural networks and VLSI bus style data paths are proposed. The synthesized architectures employ multiple busses and functional units with internal parallelism in single, or multi-processor configurations, which exploit the inherent parallelism of neural networks. Pipelined functional units with internal storage are designed to match neural network requirements. Our novel approach of investigating neural network hardware using synthesis, results in high performance architectures that are better than other architectures previously published. The synthesis results of several real-time networks are presented and architectural optimization and trade-off techniques are demonstrated. A verification methodology of the resulting synthesized systems that involves VHDL simulation is developed and used to verify our results. Moreover, VLSI implementations are used to provide insight and guidelines for the synthesis system as to the silicon areas and delays involved in neural network digital hardware. These issues are incorporated within a general synthesis CAD framework presented in this thesis.

Acknowledgments

I would like to thank my supervisor Dr. Baher Haroun for his support and encouragement. None of this work would have been possible without his vision, drive and enthusiasm. Special thanks go to my parents for their love and understanding. Several fellow students at Concordia University helped me in one way or another to accomplish this work and I thank them and would like to acknowledge the fact that they made my stay here enjoyable. In particular I would like to thank the students of the undergraduate and graduate VLSI Design courses for the design of the VLSI library units.

Dedication

To my parents

Table of Contents

List of Figures

List of Tables

List of Acronyms

ANN	Artificial neural networks
BP	Backpropagation
CAD	Computer-aided-design
CPC	Cycles per connection
CP	Couterpropagation
CPS	Connections per second
CPUS	Connection updates per second
DSP	Digital signal processing
ECG	Electro-cardiogram
FJM	Flexible joint manipulator
FU	Functional unit
FUNN	Functional units for neural networks
I/O	Input/Output ports
lr	Local register
MA	Multiply-accumulate operation
MAC	Multiply-accumulate unit
O/I	Output-Input operation
PE	Processing element
PR	Pattern recognition
RTL	Register transfer language
SFG	Signal flow graph

Chapter 1:

Introduction and Motivation

As a result of the recent growth in using neural networks for real time applications, the design of efficient architectures satisfying hard performance requirements is needed. The requirements of neural systems in processing time differ with the type of application. Typically, signal processing and adaptive control applications, for example, have constraints on the order of 50 μs to 10-100 ms respectively. Differences between applications that affect hardware implementations, are also manifested in the size of networks. While typical signal processing problems require networks on the order of 3-400 neurons, image recognition systems can require up to 64,000 neurons. The speed requirements suggest the need for high performance architectures. The differences in the types and sizes of networks imply the need for ASIC implementations of such systems. A whole class of applications also require a certain degree of adaptability. Such applications as adaptive filtering and adaptive control will necessitate the implementation of on-chip learning while others may not require such algorithms. These reasons make the design of neural network architectures subject to certain trade-offs and optimizations. An effective architecture is one that strikes a compromise between computational power, memory size and input-output bandwidth.

Several digital neurocomputer designs were proposed for the implementation of neural algorithms. Most of these, however, use large and expensive general purpose processors in single, or multi-processor configurations, which provide more flexibility than is required in specific applications. Systolic arrays that use simple processing elements with limited internal parallelism were also proposed, but did not investigate the use of parallelism within each processing element. These shortcomings have been recognized in DSP real time applications and the use of automated ASIC design and architectural exploration techniques have achieved unequaled performances. Neural

networks are special signal processing applications with a high degree of connectivity, special transfer function considerations, and large memory storage requirements. The use of DSP style architectures for the implementation of neural networks is therefore justified. Further, the parallelism in neural network algorithms lends itself to efficient parallel implementations and the investigation and exploration of the different degrees of parallelism for a neural processor is warranted.

This thesis presents automated synthesis methodologies and architectures targeting those real-time adaptive applications with high performance constraints. A synthesis tool is therefore presented in this thesis that uses novel methodologies that are specifically designed and oriented for neural network synthesis. The tool automates the exploration of the different degrees of parallelism of proposed architectures and generates one that satisfies the hard real time constraints and fulfills the VLSI silicon area limitations. The synthesis methodologies are further extended to implement multi-processor environments for regular algorithms. Several architectural design techniques and optimizations are investigated using the synthesis tool to achieve optimum architectures for specific applications. The synthesis is further based on a multiple-bus data path using pipelined functional units specifically designed for neural network implementations. The architecture, which can consist of single or multi-chip implementations, is flexible and allows for several degrees of pipelining.

Novel issues addressed in this thesis include the neural network architecture design search using synthesis techniques similar to ones used for DSP applications and the design of special pipelined units for the proposed architecture. Scheduling of pipelined units with varying pipeline depth is implemented and special compound operations are also introduced that result in more compact schedules. Several optimizations include neuron splitting and the use of local storage and local interconnections. Multi-processor synthesis is implemented, based on a systolic architecture. Several neural network specific heuristics are proposed and implemented in the synthesis system.

The organization of the following chapters in this thesis is as follows:

Chapter 2 is an introduction to artificial neural networks and their hardware implementations. It presents the basics of neural networks in terms of topology and algorithms. The requirements for real-time adaptive neural networks in terms of their hardware implementations are provided. Based on these requirements, different implementations including analog and digital techniques are compared. A survey of a variety of digital implementations is also provided.

Chapter 3 is a review of high-level synthesis and introduces the problem of neural network synthesis. Those issues involved in synthesis that are integral to this thesis are explained and various scheduling techniques are compared. The issues involved in the choice of synthesis methodologies used in this thesis are mentioned.

Chapter 4 proposes a bus style architecture to implement artificial neural networks. To that end, several special functional units, specific to neural network requirements are presented. Multi-processing systems are also proposed. Different architectural trade-offs are presented and discussed.

Chapter 5 explains the architectural synthesis methodology used to optimize the designs and investigate the architectural trade-offs. It contains descriptions of the algorithms involved in the synthesis as well as the description of the synthesis methodology as a whole. Novel approaches such as compound operations are introduced.

Chapter 6 presents the results of the design space exploration of typical neural networks used in real-time applications. The architectural trade-offs which are presented in chapter 4 assume a quantitative nature based on the results of several simulations of typical architectures.

Chapter 7 details the VHDL synthesis verification tool and provides the VLSI implementation of an example implementation. The VHDL modelling of the architectures is given and thus, the correctness of the synthesis can be checked by simulating the resulting architectures. VLSI implementations provide accurate area and delay

measurements and provide an idea of the silicon area involved in systems based on the proposed architectures.

Chapter 8 is a conclusion and an overview of the future directions that the extension of this work can follow. The novelty and importance of the different approaches used in this thesis are emphasized. Further, this chapter sets the foundations for future research geared towards completing and further optimizing the synthesis tool and the different architectures used.

Appendix I provides an insight into what is involved in terms of the Prolog implementation of our synthesis tool. The different lists and data structures used are shown.

Appendix II shows the VHDL modelling of an example architecture implementing a XOR using the backpropagation network.

Appendix III presents the synthesis output of our developed tool for the XOR network and its corresponding microcode used in the VHDL simulation.

Chapter 2:

Artificial Neural Network Algorithms and Implementations

2.1. Introduction

This chapter presents an overview of artificial neural networks that includes descriptions of network's topologies, neuronal computations and a presentation of the ideas involved in training artificial neural networks. The model of an artificial neuron is introduced. Feedforward and recursive topologies are presented. The most common neuronal activation functions in use are explained and various supervised and unsupervised training mechanisms are given. The chapter also discusses the hardware requirements of neural network systems as far as performance, memory, communications and flexibility. It also compares digital and analog implementations and advances several arguments for choosing digital systems for hardware intended for adaptive real-time networks. A survey covering several types of digital implementations, ranging from bit-serial and stochastic implementations to parallel and general-purpose DSP processors, is included. We attempt here to survey the different paradigms in order to define the terms and create the proper context to subsequent discussions. Further detailed surveys can be found in [1-11].

2.2. Introduction to artificial neural networks

2.2.1. Overview of neural networks

In recent years, the use of Artificial Neural Networks (ANNs) in *adaptive real-time applications* has seen a dramatic increase. The special features offered by ANNs such as the capability to learn from examples, their ability to generalize and adapt, their extensive parallelism, fault tolerance and noise resistance, made their use in a number of real-world applications extremely attractive. Their successful use in an impressive number of applications ranging from signal processing to robotic control applications has lead the

way to their integration into existing real-time applications.

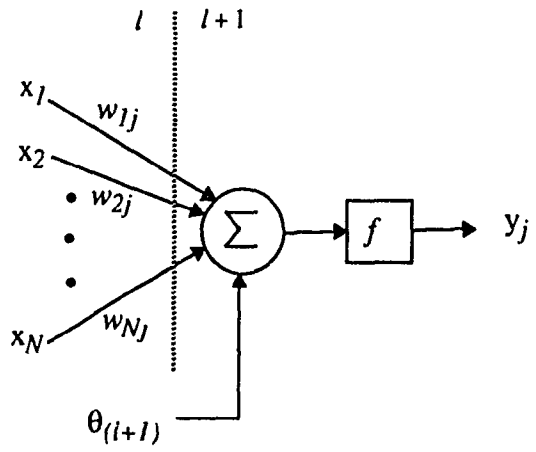


Figure 2.1: Artificial neuron with activation function

ANNs are loosely modeled after the brain. The attempt to model neurons in the brain gave rise to a class of systems known as perceptrons which are single layers of artificial neurons. The artificial neuron whose basic functional architecture is shown in Figure 2.1 is a simple model of the biological neuron. Input activations from other neurons are designated by x_1 to x_N. The weighted edges w_{1j} to w_{Nj} (weights) correspond to the synaptic connections between neurons in the brain. In this model, the weights are multiplied by their corresponding states (neuronal activation states) and summed. An added threshold adjust θ is sometimes used to offset the neuronal summations. The neuron activation y_j is then calculated using an activation or thresholding function. This reduces the summation result to within the boundaries of the activation function. The artificial neuron then, functions as follows:

$$y_j(l+1) = f\left(\sum_{i=1}^{N} x_i(l) w_{ij}(l+1) + \theta(l+1)\right) \tag{2.1}$$

where $l+1$ is a layer of neurons including the one which activation is being calculated and

l corresponds to a preceding layer of neurons.

These systems gave rise to a whole generation of ANN models some closer to their biological inspiration and others having mathematical and statistical conceptions. ANNs use the matrix of synaptic weights as the primary medium for representing and manipulating information. These weights are usually adjusted during training until the matrix converges towards the best values yielding the desired network. This training process is often guided by an energy function. The different types of ANNs available can be characterized by the following properties:

- The topology of the network and interconnections (fully connected, layered) and the propagation of information (feedforward, recurrent)
- The type of neuronal computation (sigmoidal, linear)
- Learning algorithm (auto-associative, delta rule, etc...)

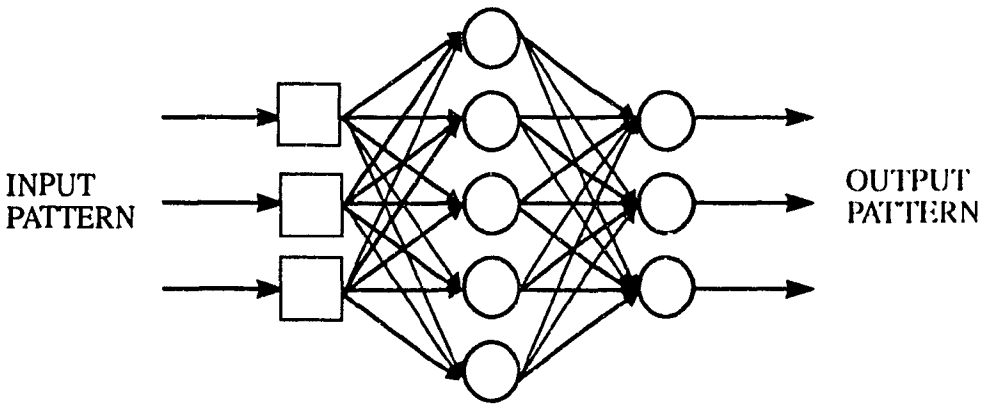


Figure 2.2: Backpropagation network (Feedforward multi-layer perceptron)

2.2.2. Network topology

ANNs are usually arranged in layers of artificial neurons. The Perceptron (as well as similar models such as the adaptive linear element or Adaline [11]) consists of a single layer in which each input connects to all neurons. Single layered networks such as the Perceptron are severely limited in their computational ability, notably demonstrated by the XOR example, a linearly inseparable problem. The linear separability problem was solved

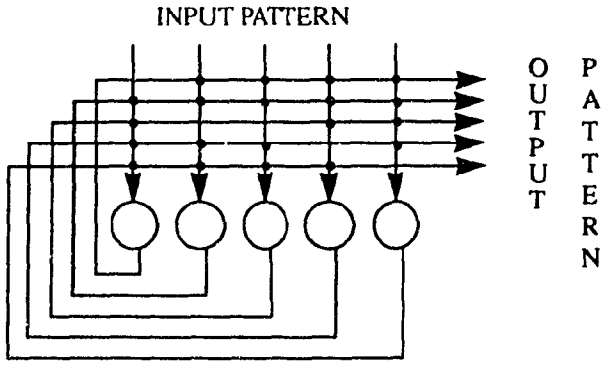


Figure 2.3: Hopfield Network (Recurrent, auto-associative)

by extending the idea of Perceptrons into multilayer networks which have the ability to perform more general classifications. Multilayer perceptrons are the most widely used networks. They have several layers of neurons (usually 3 or 4; a number of layers greater than 4 usually results in a network that is difficult to train). The input layer receives patterns of data and the output layer presents the network's response. The layers in between are called hidden layers. These ANNs are called feedforward networks, the most popular of which is the Backpropagation network (BP), shown in Figure 2.2.

Another type of networks is recurrent, such as the Hopfield network shown in Figure 2.3, where the outputs of the network are fed back to the input [12]. The capability of calculating outputs based on current inputs and previous outputs can exhibit behaviors approximating short-term memory in humans. While the Hopfield network consists of one layer only, other recurrent networks can consist of more than one layer as the Bidirectional Associative Memory [16].

Most neural networks are fully connected (between pairs of layers), some like the Neocognitron [13] (shown in Figure 2.4) have different interconnection styles to distinguish different features of the input patterns. Even networks that are commonly fully connected such as the BP, can assume any interconnection pattern between consecutive layers depending on the specific problem at hand.

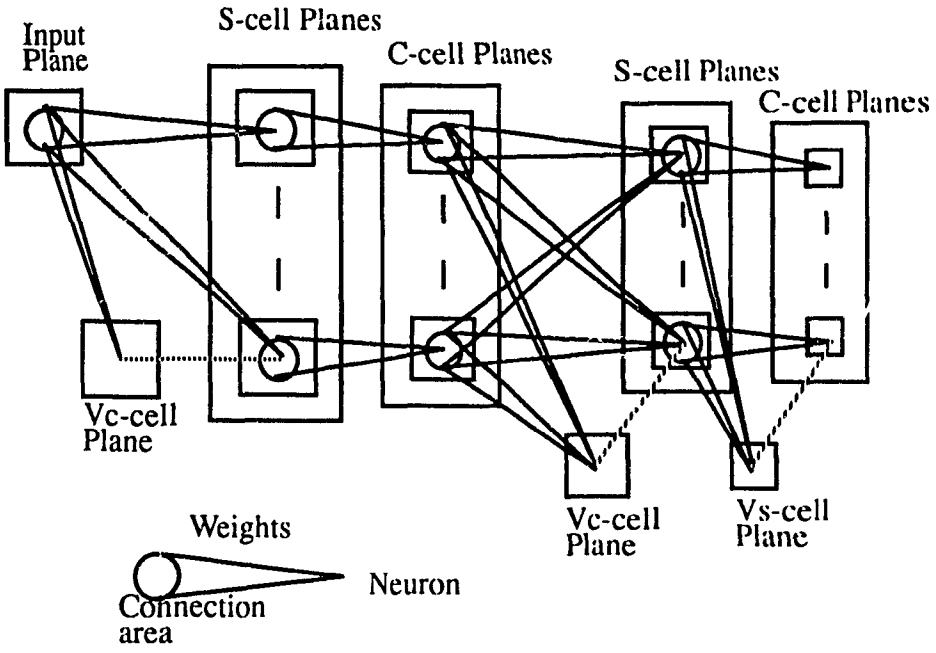


Figure 2.4: Neocognitron topology [14]

2.2.3. Type of neuronal computation

The adaline uses an activation function that produces a binary ± 1 output. The hard limiting quantizer used as the activation function being:

$$f(x) = \text{sgn}(x) \tag{2.2}$$

A bias weight connected to a +1 activation state is used to provide a threshold level to the activation function. Networks such as the Madaline used additional boolean logic function such as *and* gates, and *or* gates as well as a *majority-vote-taker*. Other forms of activation functions have been used in ANNs, notably the sigmoid (logistic) function (shown in Figure 2.5):.

$$f(x) = \frac{1}{1 + e^{-\alpha x}} \tag{2.3}$$

This type of nonlinear activation function provides saturations for decision making

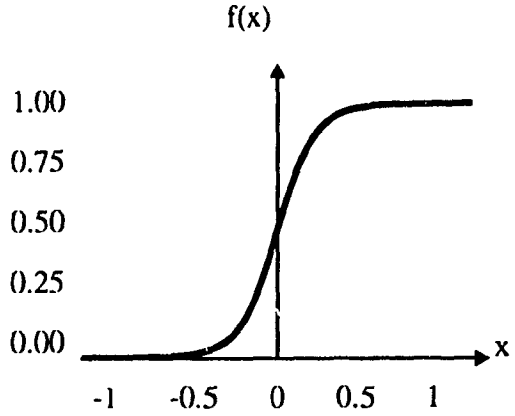


Figure 2.5: Sigmoid function

and has differentiable input-output characteristics that facilitate adaptability. Another nonlinear, sigmoid style, function is the hyperbolic tangent which is also used in today's ANN models:

$$f(x) = \tanh(x) = \left(\frac{1 - e^{-2x}}{1 + e^{-2x}}\right) \tag{2.4}$$

The advantage of the hyperbolic tangent over the logistic function is that it produces bipolar outputs which has proven beneficial for a number of applications.

2.2.4. Training algorithms

Artificial neural networks can modify their behavior and self-adjust in response to a training (learning) mechanism. During training, the network is presented with a set of patterns and produces a certain output. The network's weights are then readjusted to get an output closer to the one expected. While supervised training presents the network with the desired outputs, and unsupervised training does not, both have to adjust the weights of the network in order for it to converge towards a solution. The network's learning rule determines the method and rate of convergence by specifying the procedure to adjust the weights. Learning algorithms are typically iterative processes. Some models use

modifications of the Hebbian rule which strengthens the weights between neurons according to their activities:

$$\Delta w_{ij} = x_i x_j \tag{2.5}$$

where w_{ij} is the weight of the connection between neurons i and j which activation states are x_i and x_j. Statistical training methods make pseudorandom changes in the weight values, retaining those changes that result in improvements. Competitive learning methods usually adjust the weights of a *winning* (one with the highest activation state) neuron. Learning rules include the generalized delta rule, competitive learning, Hopfield minimum-energy rule and the Boltzmann learning algorithm. The training dynamics are usually controlled by an energy function that describes the system's stability.

Model	Recall algorithm	Learning rule
Backpropagation	$y_j = \Sigma w_{ij} x_i + \theta_j$	$\Delta w_{ij} = \eta \Sigma_p (d_{i,p} - y_{i,p}) x_{j,p}$
Hopfield	$y_j = \Sigma w_{ij} x_i + \theta_j + w_i I_i$	$w_{ij} = \Sigma_p (2x_i^{(p)} - 1)(2x_j^{(p)} - 1)$
Boltzmann	$y_j = (\Sigma w_{ij} x_j + \theta_j)\lambda_j$	$\Delta w_{ij} = \frac{\eta}{T(P_1(s_i s_j) - P_0(s_i s_j))}$

Table 2.1: Three of the most common ANN models

Table 2.1 shows three of the most common ANNs in use today. **Backpropagation** is a *feedforward* network that uses *supervised* training implemented by the *generalized delta rule*. The **Hopfield** network is a *recurrent auto-associative* system that uses an *unsupervised* training algorithm. The **Boltzmann** network is a *stochastic* network that uses a noise function to reach a global minimum energy state. While these three networks define the three techniques most commonly used to attack real world problems, a large number of different, more complicated networks, is available. Networks like

Counterpropagation [15], Bidirectional Associative Memory [16], Adaptive Resonance Theory [17] and the Neocognitron [13] all have been successfully used. More biologically oriented networks include the Retina and the Cochlea [18].

2.3. Typical real-time ANN applications

In order to evaluate the performance of the architectures presented in this thesis, as well as the performance of the synthesis tool, three main fully connected backpropagation (BP) networks and one counterpropagation (CP) network have been used. The first BP network is a one DOF robotic flexible joint manipulator (FJM) with 6, 21, 12 and 1 neurons used in adaptive plant dynamics [19]. The real-time requirements of such a network can vary from 10 to 20 μs for the recall phase. The second BP network is a an electro-cardiogram (ECG) biological signal processing network with 40,10 and 1 neurons used for filtering the high frequency noise of electro-cardiogram signals [20]. While an ECG with 8 probes has requirements on the order of hundreds of Hertz sampling frequency, it is typical of a class of biomedical signal processing problems that may require faster systems such as EMG and EEG with up to 10 KHz sampling frequencies. Reasons for using neural networks for signal processing applications are given in [20]. The third BP network is a Pattern Recognition (PR) network with 16,5,9 and 4 neurons [21]. In a typical postal code recognition problem, such a network would need to read up to 10,000 characters per second translating to a requirement of 100 μs for the recall mode. The CP network used (intended to prove the flexibility of the architecture and synthesis tools proposed) is a small version of the NASA space station robot arm guiding network using 64, 8 and 1 neurons respectively [15]. The size is reduced to keep it in line with the BP networks. Its speed requirements are of the same order as the FJM network. All these networks are real-time adaptive ANNs with specific requirements. They have approximately the same size and are typical of the types of networks that are being used in industrial applications today. These networks, the robotic network in particular, further

require on-chip training and a certain degree of flexibility.

2.4. ANN hardware requirements

2.4.1. Performance constraints

Real-time applications require high processing speed. A typical Radar Pulse Identification problem requires a sampling period of 50 μs. This entails a performance on the order of hundreds of Mega Connections Per Second (MCPS). CPS is the most popular speed measure of ANN hardware (Refer to Section 2.7). Increasing the speed of a network requires an increase in the concurrence of computation and therefore parallelism at the hardware level [2].

2.4.2. Memory storage requirements

One of the major problems in ANN hardware implementations is the size of the networks. While this is a constraint on the minimum achievable speed, it is more restrictive for memory storage requirements. The memory size is directly proportional to the number of synapses which usually grows by a square order of the number of neurons. A typical ANN system requires between 5 KWord for robotic applications to 5 MWord of memory storage for low level pattern recognition algorithms, using anything from one bit to 32 bit words.

2.4.3. Parallelism and virtual implementation

ANNs attractiveness is due to a certain extent to their massive parallelism. Architectures implementing them in hardware need to provide a certain degree of parallelism. This also leads to faster architectures. The use of large, parallel systems requires the design of a simple architecture that could be scaled up easily. The middle ground between highly flexible, serial simulations and rigid, highly parallel implementations of neural networks lies in virtual implementations of neurons where more than a single neuron is executed on a processing element allowing for internal

parallelism among the hardware units on the processor as well as inter-processor parallelism (Section 2.6).

2.4.4. Communication

For any degree of parallelism in a hardware implementation, different components have to connect to and transfer data among each other. The high performance requirements of ANN applications enforce strict speed requirements on the communication between the neurons. Due to their massive structure and their large number of interconnections, ANNs rely heavily on interneuronal communications and effective transmission methods should be implemented.

2.4.5. Flexibility and adaptability

A number of real-time applications require adapting to the environment which would involve slight modifications in the size or training data of an ANN's lifetime. Non-adaptive systems are fast since their implementations use fixed weights and consequently result in less overhead. Adaptive systems, on the other hand require programmable storage of the weights which complicates their hardware implementations. While there are applications where off line learning is practical, the ability to learn as more information becomes available to the system is invaluable. Adaptability is important for these types of systems. A completely non-adaptive customized implementation may not be suitable in such cases and general purpose neural-computing machinery may loose some of needed performance. Customized implementations with a certain degree of flexibility are then needed. This implies some programmability in the architecture.

Algorithm development has stabilized to a certain degree, the optimization of learning algorithms however, has opened the way to a large amount of learning techniques and additives to each algorithm that may be used in an application. A good hardware implementation needs therefore, to take into account all these variations by allowing a certain degree of programmability.

2.4.6. Minimum silicon area

The large ANN algorithms require large hardware areas in order to attain real-time speeds. Silicon area is important in terms of cost of the units and the manufacturing yield. Area is also important in terms of fault tolerance and testability. All these issues common to any VLSI design are even more important in the design of ANN hardware since their direct, parallel implementations consume large areas. One of the ways of minimizing the implementation areas on the system and architectural levels consists of using virtual implementations and mapping several neurons on each available processor rather than provide one processor per neuron.

2.5. Analog versus digital implementations

Implementations of ANNs have used digital, analog, optical and combinations of these technologies. While optical implementations seem ideal for ANNs due to their parallel nature, several problems exist in the difficulty of storing photons and weight modification by light switching. Additional problems include cost, size and criticalness of alignment. Coupled with the fact that real-time applications are usually electrical in nature which requires electrical interfaces, optical computing is yet to be employed in practical, cost-effective implementations. A large amount of research into ANN optical implementations is being performed, but analog and digital technologies are still more mature approaches to ANN system hardware implementations.

Analog implementations of ANNs are typically designed using MOSFETs for synapses and operational amplifiers in a comparator configuration as shown in Figure 2.6. These implementations as well as other more complicated analog neuronal integrations suffer from several problems:

- The difficulty in programming analog weights requires designers to use digital techniques to implement on-chip learning algorithms, or even export the training to a host computer.

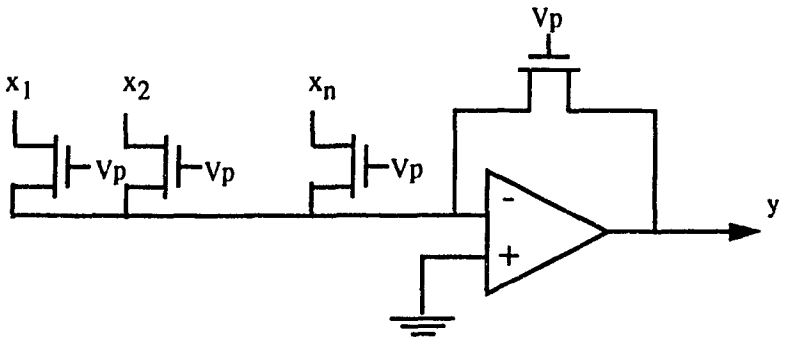


Figure 2.6: Typical analog implementation of a neuron

- The size, resolution and design are all interdependent making the design process more difficult and increasing the design cycle. Accuracy is highly dependent on chip area.
- Difficulty of fabrication of precision resistors and capacitors results in the use of limited resolution systems that may not be suited for several applications.
- Crosstalk and susceptibility to coupled-in interference as well as noise and temperature dependence require special considerations.
- Noise and current consumption limit the size of the network (or the number of neurons) that can be implemented on a chip.
- The lack of design tools that are available to digital design makes analog implementations unfavorable.

A number of researchers advocate the use of analog hardware to implement neural networks based on similarities between the structure of the brain's neurons and today's transistors [22]. Whether the arguments for analog neural networks are well directed or not, a clear advantage of digital implementations is that the analysis and modelling of the network's characteristics can be made independently of circuit design. This reduces the amount of time spent on a system's design and allows the use of computer-aided design (CAD) tools to optimize the architecture and further reduce the design cycle. A clear

boundary between the advantages or disadvantages of the two technologies does not exist but certain applications with specific requirements have been proven successful for both.

Some of the problems with digital implementations, however, include limit cycles (parasitic oscillations) [3]. This is a result of amplitude quantization in recursive digital structures. This problem can be eliminated by increasing computing accuracy and numeric range or by modifying certain algorithms. Another problem is the large silicon area required as compared to analog implementations. The large number of transistors used in digital ANN implementations would require these systems to have a certain amount of fault tolerance.

As a general comparison of digital and analog implementations of ANNs, Table 2.2 presents the advantages and disadvantages of each. Digital implementations are definitely the choice for adaptive real-time systems that require a certain degree of flexibility. It is also usual to implement biologically styled networks in analog because of their connectionist nature and mathematically oriented ones in digital since they are more suitable for mathematically intensive algorithms.

Implementation	Advantages	Disadvantages
Digital	High precision Smaller transistor sizes Ease of memory storage Ease of learning algorithm implementation Parallelism and scalablility Flexibility Generality Ease of Testability/ reliability Availability of CAD support	Quantization errors Large area Difficult nonlinear function implementations Large number of transistors

Table 2.2: Comparison of digital and analog implementations

Implementation	Advantages	Disadvantages
Analog	Small area Nonlinear functions easy to implement Small number of transistors	Limited resolution Poor noise immunity Crosstalk susceptibility Difficulty of memory storage Difficulty of learning implementations Rigidity, inflexibility Temperature dependence

Table 2.2: Comparison of digital and analog implementations

2.6. Digital architecture survey

Digital implementations of ANNs use parallel, bit-serial or stochastic arithmetic. The majority of these have one processor per neuron as shown in Figure 2.7. The neuron

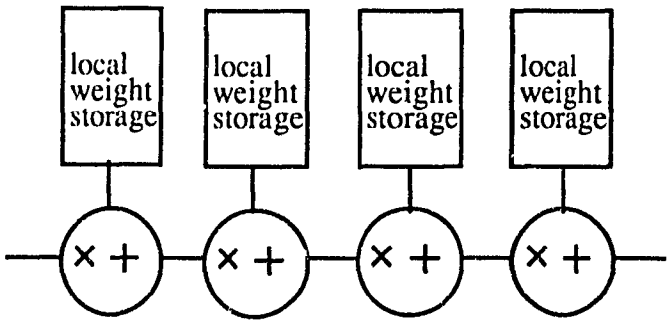


Figure 2.7: One processor per neuron implementation

processors are designated by a multiply-accumulate unit symbol. The weights are stored in memory local to each neuron processor. The communication in such systems can use either systolic interconnections [23,24] or broadcast busses.

The second type has one processor per synapse as shown in Figure 2.8. This implementation has a low synaptic storage density but a high throughput. Systolic interconnection are usually implemented for such systems.

The third type uses one processor for many neurons in DSP fashion as shown in Figure 2.9. This number of processors that can be put on one chip is then I/O limited. The

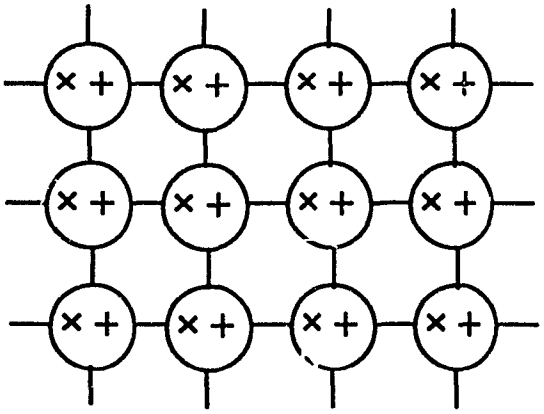


Figure 2.8: One processor per synapse implementation

processor architecture can assume a variety of structures. The weights and other intermediate variables can be stored on-chip or the processor can be interfaced to a RAM chip. The advantages of using such system include minimizing the silicon area and reducing the massive connections problem to one of inter-processor communication. Implementing one neuron per processor requires a number of connections equal to or greater than the number of network connection. This is dramatically decreased in virtual implementations and depends on the relatively small number of processors. Parallel systems can use several processors each using virtual implementations.

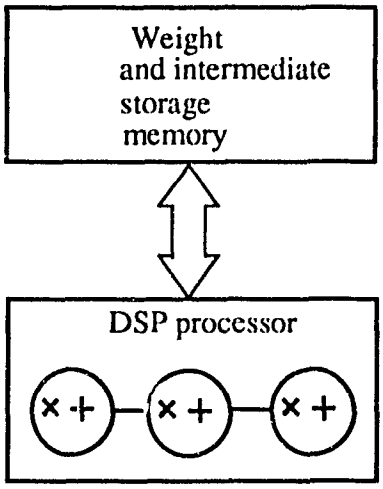


Figure 2.9: Virtual implementation of neurons

The following is a survey of digital architectures for ANNs intended to provide an idea of the diversity of the digital implementations that have been proposed.

NETSIM

TI's NETSIM includes a *solution engine* chip that does the neural computations and a *communication handler* chip to route the neural activations. The NETSIM card includes the solution engine, the communication handler, a microprocessor and memory. The solution engine can perform an 8x8 bit multiply-accumulate in 250ns resulting in a performance of 4 MCPS (Mega Connections Per Second) [25].

Digital neurochips

Duranton and Sirat's digital neurochip is a fully digital architecture storing 16 bit synaptic weights in an on-chip RAM. On-chip learning is implemented but the sigmoid function is exported. It uses a bit serial technique to implement the inner product [26]. Hirai et al's digital neuro-chip is a 1.2 μ CMOS gate array implementation, of 6 neurons and 84 6-bit synapses using a variant of pulse-stream arithmetic [27].

Ouali and Saucier's Neuro-ASIC

Ouali and Saucier's neuro-ASIC includes a local memory for storing synaptic weights and activation function parameters, a multiplier, an adder/subtracter, a controller, input, output and state registers for interfacing to a multi-chip network [28].

DNNA

Neural Semiconductor's DNNA is based on stochastic pulse trains. Each synapse includes a separate stochastic pulse train generator. Each synaptic pulse stream is *anded* with an activation output stream to produce a synaptic product which are *wire-ored* to produce activation input streams. A two chip set can implement 32 neurons and 1024 connections [29].

STONN and TInMANN

North Carolina State's STONN is an 100K transistor CMOS implementation of the Hopfield network using stochastic logic and bit-level pipelining. STONN stores the

weights in an on-chip shift register and generates stochastic samples of N weights per cycle [30]. TInMANN (The integer markovian artificial neural network) is a stochastic architecture proposed by the STONN group to implement competitive learning using stochastic computation. It updates the weights with a probability proportional to the neural input, causing neurons closest to an input vector to move toward it and push others away [31].

CNAPS

Adaptive Solutions' CNAPS is a general purpose SIMD multiprocessor architecture developed for ANN applications. A single chip contains 64 processors (80 processors for fault tolerance), a 32 bit instruction bus, an 8 bit global output bus, an 8 bit global input bus, and a 4 bit inter-processor bus. Each processor includes 4 kbytes of weight memory, a memory base address unit, 32 16-bit registers, an input unit, an output unit, a 8x16 bit multiplier, a 32 bit saturating adder, and a logic and shift unit [32].

Kung's systolic implementation

Kung presented a systolic mapping of ANNs. A board level prototype of a processing element was designed using microprogrammable, commercially available chips. The architecture includes a multiplier, an ALU, 2 memory banks, a 5 port register file., a RAM and address generation units and input/output units [23,33].

GCN

Hiraiwa et al's GCN is a two level pipeline processor array of PEs that use Intel 80860 processors with local memory. Communications between processors can be asynchronous and use a high bandwidth FIFO. The local memories are used to store the weights, data and intermediate results [24,34].

Delta

The Delta processor from Science Applications Intl. is a floating point processor with 1900 instructions supporting 32 and 64 bit integer and floating point operations. Most instructions execute in one cycle [35].

Neurocomputer implementations on parallel machines

Mappings of ANN algorithms on the Hypercube [36], the Connection machine [37], the CMU Warp [38], as well as other parallel machines were also implemented. The networks are usually pre-partitioned before the actual mapping is done.

A good survey of existing digital architectures and circuit design techniques involved in digital VLSI implementations is presented in [3].

2.7. Performance measures

The variety of ways that ANNs have been implemented in hardware and even the variety of ANN algorithms and configurations themselves make it extremely difficult to evaluate the performance of a particular implementation. A relatively good indication of the network's speed is the number of connections processed per second (CPS) in recall and the number of connection updates per second (CUPS) in training.

Connection refers to the number of synapses in the network. In feedforward networks, the execution of a connection corresponds to the multiplication of the activation value with the weight of the connection. Several other operations are included in the calculation of the network, mainly the accumulation of the connection results and the activation function calculation. The BP network of Figure 2.2, has 30 connections. This entails 30 multiplications and 30 additions. The activation operations on the other hand, depend on the number of neurons and for the same example, only 8 activation function operations are used. Other operations such as inputs and outputs may be required in a typical architecture. However, the large number of connections being the dominant factor in ANNs, the number of connections per second of execution gives a good indication of the speed of the network. A connection could therefore be considered as a single multiplication, followed by a single addition. All the other operations are implicit and for most cases, when large networks are considered, negligible. CUPS refers to a more complicated algorithm, the one used in training. These algorithms usually entail a larger

number of operations and a wider variety. The fact that the majority of operations is related to the number of connections still allows the number of connection updates (the correction or modification of weight values) to be an effective measure of comparison.

These measures are still dependent on the network's style. Even for the same network style, such as backpropagation, these measures are relatively inadequate to offer a good indication of the system's speed because of the type of architecture and the type of interconnection system used. The operations performed in such similar ANNs as BP and CP networks are still quite different. CP, for example requires some type of winner take all mechanism that is not present in BP networks. Even BP networks can be using slight modifications in their learning rules or their nonlinear functions.

On the architectural level, some systems will perform better for fully connected networks whereas others exploit the interconnection pattern in their partitioning. Some will perform better activation function calculations whereas others achieve faster multiplications and additions and would perform better for networks that use simple activation functions. An implementation of CP network, for instance, can implement competitive activation functions in a variety of ways. Special purpose architectures may be optimized for this style of computation whereas a flexible system will generally use standard operations to implement the competitive algorithms. Again, some architecture are optimized for the recall phase whereas others perform just as well in training.

Nevertheless CPS and CUPS present the only available method of comparing different architectures and they are analogous to the general purpose and super-computing standards of MOPS and MFLOPS.

For systems allowing more than one unique clock speed, or for systems that have not been implemented, it may be useful to evaluate the performance in terms of the number of cycles needed to execute the computations of a single interconnection or CPC (cycles per connection) in either the recall or the training modes. The relation between CPC and CPS (or CUPS) is

$$CPC = \frac{NC \times Clock}{CPS} \tag{2.6}$$

where *NC* is the number of cycles for the execution of the network and *Clock* is the clock frequency of the architecture. This provides a reasonable measure of comparing different architectures with different clock speeds. This measure is analogous to the *Number of Operations per Cycle* for RISC and super-scalar machines.

2.8. Conclusion

The use of ANNs in real world applications requires the design of highly efficient hardware. While the need for both customized and general purpose processors is evident, a greater need exists for customized implementations with a certain degree of flexibility and adaptability. It is also evident that the inherent parallelism in ANN algorithm should be exploited. For applications requiring high precision calculations, or on-chip implementation of learning algorithms, digital implementations are usually necessary. The need to balance performance, flexibility and affordability creates a need for design tools that can investigate the architectural trade-offs of parallel implementations of digital neural networks. This thesis presents such a tool that offers optimized architectures for real-time adaptive ANNs.

Chapter 3:

Review of High-level Synthesis

3.1. Introduction

This chapter presents a review of high-level synthesis. It outlines the need for automating the design process, and discusses all the steps involved in synthesizing digital architectures including scheduling, allocation and binding. It investigates different techniques that include partitioning algorithms and pipeline scheduling. It also presents several scheduling algorithms in use and discusses their advantages and disadvantages. It further discusses the suitability of those algorithms for the synthesis of neural networks. High-level synthesis includes data path and control synthesis. Data path synthesis involves the automated design of the arithmetic units whereas control synthesis is concerned with the automatic generation of a system's control units. The main emphasis in this review is towards data path synthesis which is more relevant to the synthesis system proposed in this thesis.

3.2. Introduction to high-level synthesis

Automatic datapath synthesis has become an important area in computer-aided design. The synthesis transforms an abstract behavioral description of a system into a structural hardware description respecting certain constraints and requirements such as area, speed and functionality. Figure 3.1 shows the different design representation levels of a system. Descriptions of a system can be at *behavioral*, *structural* or *physical* levels. Each of these is further divided into an *architecture*, a *register transfer*, *logic* or *device* level. The design process usually involves a spiral from the higher, more abstract representations to lower, more specific descriptions. High-level synthesis is shown as a transformation from an algorithmic description to a netlist or register-transfer-level description using functional units, registers and interconnects. The use of automation

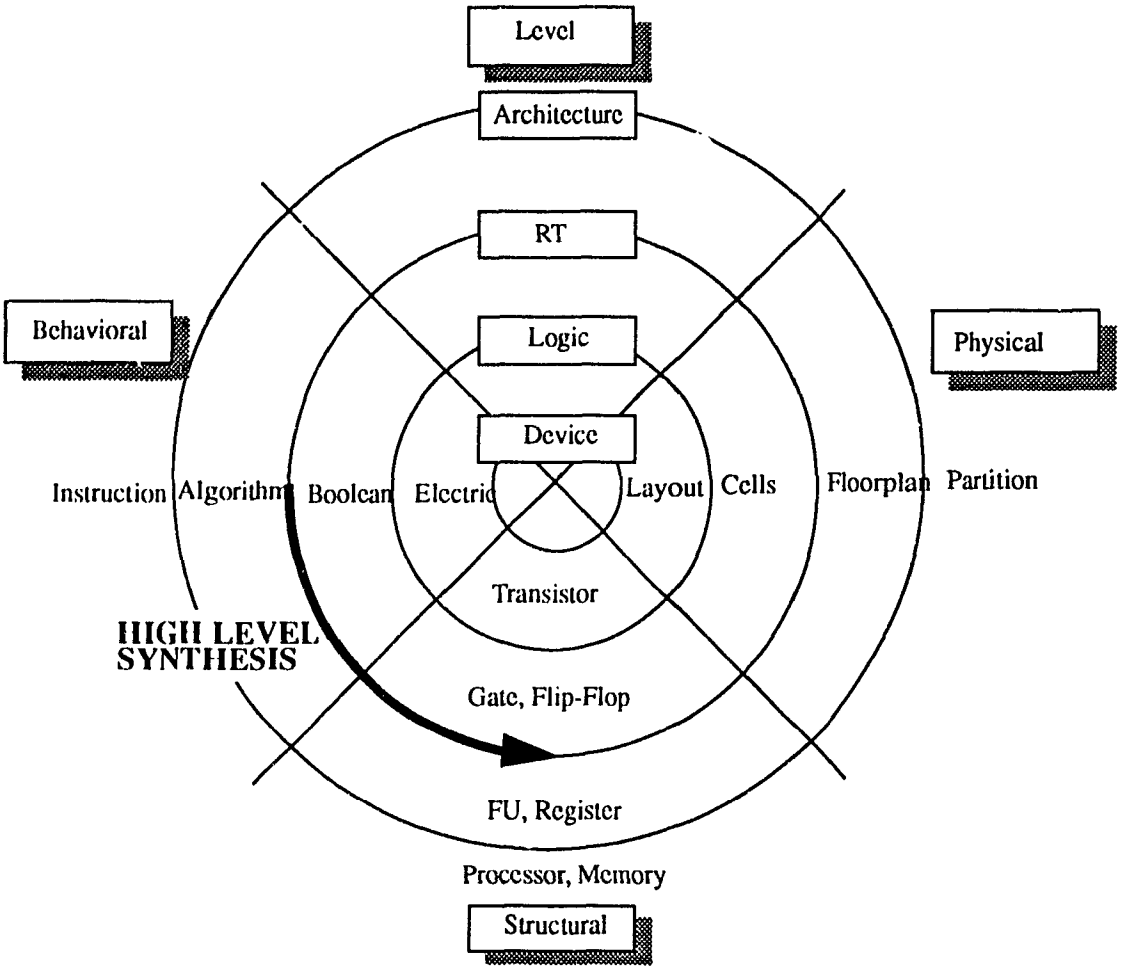


Figure 3.1: The different levels of a system's description

allows for shorter design cycles and pre-defined methodologies result in fewer design errors. Synthesis also provides the ability to search the design space and optimize designs. Synthesis includes solving NP-complete problems inherent in most of the issues involved, hence heuristics are employed to obtain a reasonable execution time for the CAD tool. Most synthesis systems available today are geared towards specific applications in order to find effective heuristics tuned to the application.

The first step in high-level synthesis is the compilation of the initial specification. This is followed by several synthesis procedures including scheduling, allocation and binding. After scheduling, the datapath and control generation is done [39]. A brief definition of these issues follows.

3.2.1. Compilation

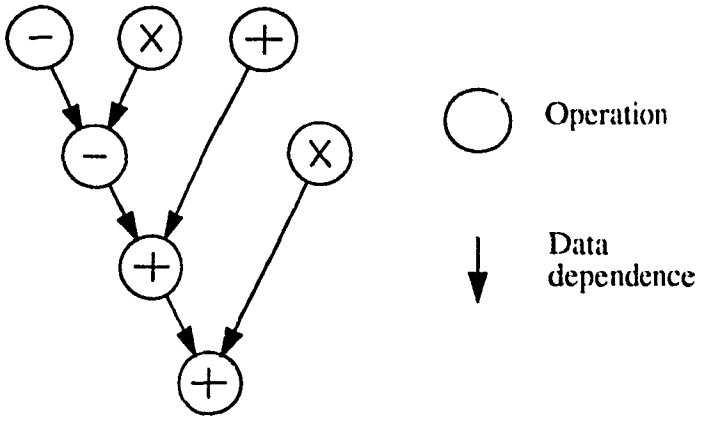


Figure 3.2: Example signal flow graph

The compilation procedure consists of translating the initial specification into the language used by the synthesis systems. This language can be a procedural (sequential) language or a declarative (functional or a logical) language. Languages that have been used include everything from C and Hardware C [40] to VHDL [41] to LISP and Prolog. The resulting specification of the problem is usually a parse tree or a signal flow graph (SFG). The advantages of using SFG descriptions is the nature of algorithms targeted for synthesis (DSP) where the designers are familiar with block diagram representations of the algorithms. Another reason is the ease of graphical interfacing in general. An example is shown in Figure 3.2. For data/signal flow graphs, the nodes designate the types of operations and the arcs specify the data dependencies. Separate data-flow and control-flow graphs can be included in the system's description [39].

3.2.2. Scheduling

Scheduling is the assignment of operations to control steps (clock cycles) given certain constraints and minimizing a cost function. The object is usually to minimize the number of control steps needed to execute the algorithm (i.e. speed) given constraints on hardware resources. These resources can be totally specified a priori or dynamically

allocated respecting upper limits. Operation scheduling determines the cost-speed tradeoffs of the design. In *time-constrained scheduling* the object is to minimize the area (the cost) required to meet a predetermined maximum number of time steps. In *resource-constraint scheduling* on the other hand, the object is to find the fastest schedule given constraints on the hardware resources, while in *feasible-constraint scheduling* the object is to find a solution, if one exists, given both time step and hardware resource constraints [42]. A number of scheduling techniques have been investigated and the following is an overview of the most common ones.

3.2.2.1. ASAP

As soon as possible scheduling is the simplest scheduling algorithm. It schedules the operations from a topologically ordered list and assigns them to the first possible control steps. Figure 3.3 shows the ASAP schedule of the SFG of Figure 3.2. A modified

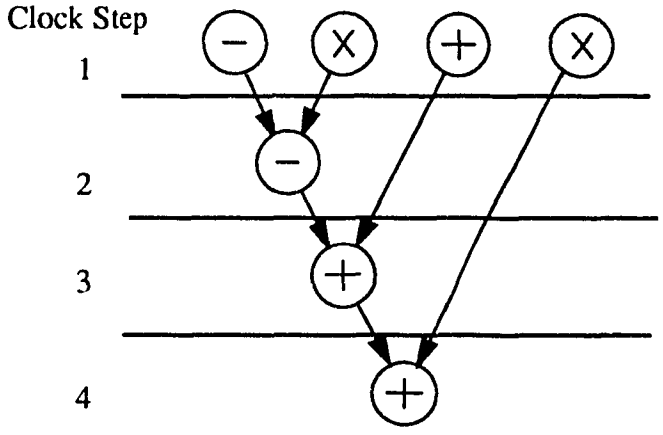


Figure 3.3: Example ASAP schedule

ASAP algorithm can enforce some hardware constraints reducing the number of operations of the same type in the same control step. if, in the given example, only one multiplier is available, the resulting resource constraint schedule becomes as shown in Figure 3.4.

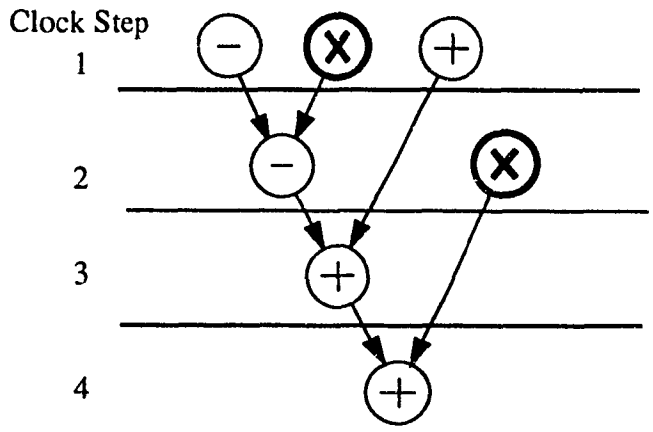


Figure 3.4: Modified ASAP schedule

3.2.2.2. ALAP

As late as possible scheduling is similar to ASAP but schedules the operations in reverse order (from the last operation to the first). Figure 3.5 shows the ALAP schedule of

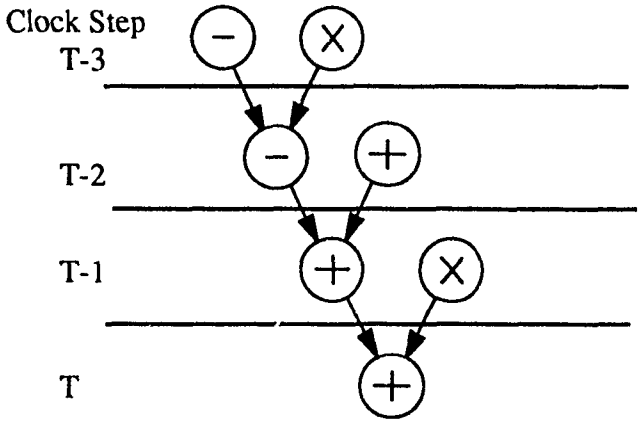


Figure 3.5: Example ALAP schedule

the example. Modified ALAP algorithm also exist that incorporate hardware constraints. The *mobility* of an operation can be calculated from the ALAP and ASAP schedules and is an indication of the affordable flexibility in the scheduling of that operation. This is used in other scheduling approaches.

3.2.2.3. List scheduling

List scheduling is an approach that considers resource constraints. It orders the operations in a list using a priority function according to certain heuristic rules and then schedules them to control steps. Conditional postponement is also used to avoid resource conflicts. The selection of the next operation or the ordering criteria is more global than ASAP and ALAP. This technique requires that the number of FUs be specified. List scheduling yields good results and is simple and fast. It is useful for long, complex algorithms since it is of the order *O(n)*. Criteria for list scheduling can include the mobility of the operation or the path length (Figure 3.6). Mobility measures give priority to the operations with the smallest mobility whereas path length measures give priority to the operations with the longest path length.

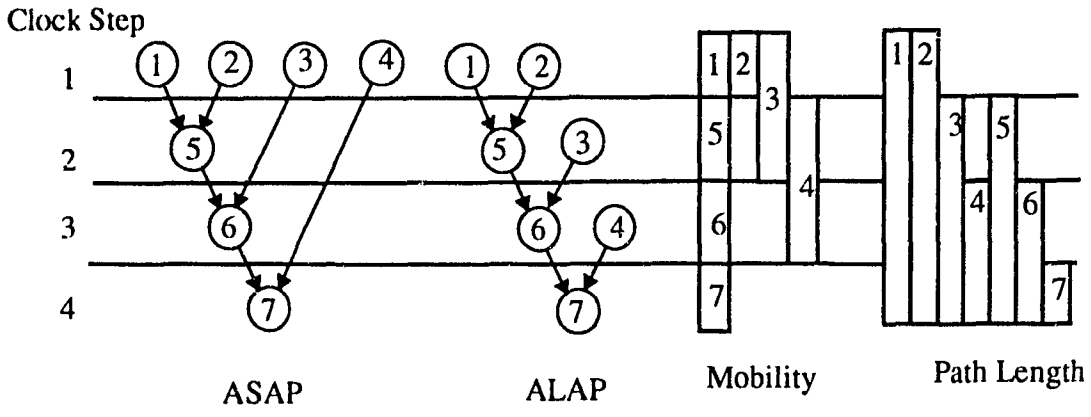


Figure 3.6: Mobility and path length criteria

3.2.2.4. Force directed

Force directed scheduling creates a *distribution graph* (Figure 3.7), based on the mobility of the operations, showing the load for each control step. Forces for every operation are calculated. An operation is then scheduled in the control step that provides it with the highest force. The forces are then re-calculated to account for the scheduled operation. This technique requires that the maximum number of control steps be specified.

Force-directed scheduling yields better results than list scheduling, but is much more complex and computational, on the order of $O(n^3)$.

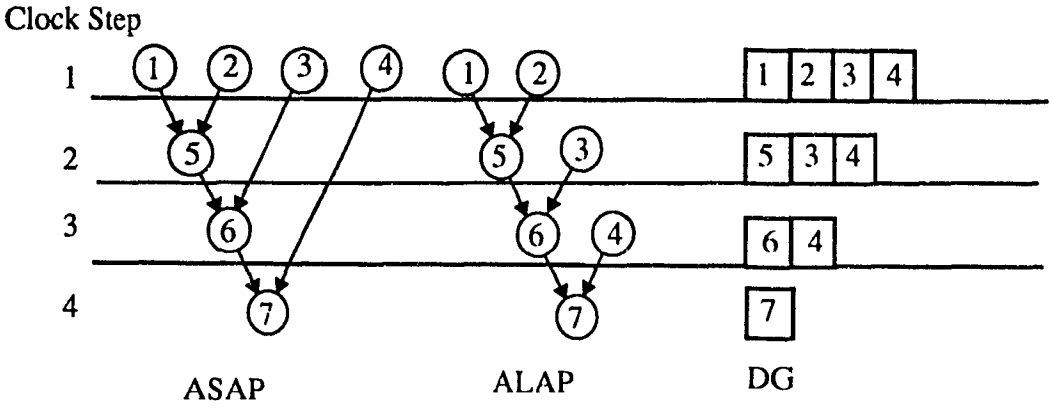


Figure 3.7: Example distribution graph

3.2.2.5. Simulated annealing

Scheduling based on simulated annealing randomly generates modifications in the schedule and accepts or rejects them according to a random rule and a temperature parameter, that decreases gradually as the annealing process proceeds, similar to the physical annealing problem. Simulated annealing is computation-intensive. The advantage of using simulated annealing, however, is that it provides *hill climbing moves*, that helps the optimization problem not to get stuck in local minima.

3.2.2.6. Integer linear programming

In integer linear programming the scheduling problem is formulated as an optimization problem meeting certain constraints. ILP will then minimize the cost of the resources. The objectives of the optimizations and the resource constraints are expressed in integer linear programming formulations. These can include a variety of scheduling problems including pipelining, chaining and multi-cycling. The advantage of ILP is that it provides optimal solutions. The disadvantage is the difficulty of formulating large problems as ILP problems and the computational nature of these optimizations. Further,

some specific scheduling problems may not be suitable for ILP formulation.

3.2.2.7. Synthesis considerations for neural networks

When handling algorithms with large numbers of operations and high parallelism, such as ANNs, approaches such as ILP are not desirable since they would require a massive number of constraints that cannot necessarily be handled by optimization software. In addition, highly parallel algorithms present high mobility of operations making force-directed scheduling an unreasonable approach. In general, any synthesis approach that does not enforce hardware constraints is not desirable if the underlying heuristics can be tuned to the type of algorithm to be synthesized. In chapter 5, a heuristic list scheduling approach is presented that is most suited for large algorithms with high parallelism, specifically, layered neural networks.

3.2.3. Allocation and binding

Allocation refers to assigning hardware resources to execute the behavior. Binding is the assignment of operations to the allocated hardware, according to a given schedule and respecting certain constraints. The object is to minimize the hardware which consists of functional units (FUs), memory elements (Registers, register files, RAMs), and communication paths (busses, local interconnects). Register allocation deals with minimizing the number of registers used in a given system. Functional unit allocation involves scheduling on a minimum number of FUs required to meet the speed requirements. Interconnect allocation entails allocating busses, local links, multiplexers and demultiplexers for all data transfers. At extremes, allocation will serialize the whole algorithm providing the smallest area and lowest speed or completely parallelize the system achieving the highest speed with the largest useful area. A good synthesis system should find an optimum architecture between those two implementations. Scheduling, binding and allocation are interdependent and the three are usually separated to produce easier algorithms [43].

3.2.4. Synthesis hardware

In general, the types of FUs usually used in synthesis systems are simple arithmetic units from available libraries. These units are seldom pipelined and allow single operation execution at a time. The pipelined units are generally simple and do not constrain the scheduling. These units are also general purpose units that are not optimized for specific applications such as ALUs, adders and multipliers. Local storage is not usually implemented and accumulators are relegated to storage registers in register files, latches or even in RAMs.

Special applications may need special FUs. These can affect the quality of the design since they account for specific requirements. That aspect is usually not addressed in general purpose synthesis systems and specifying the kind of FU that are most suited becomes a main task for the specialized synthesis system.

3.2.5. Partitioning

Partitioning refers to the division of an algorithm into several parts that are assigned to or synthesized on different processors. Partitioning is used to exploit the characteristics of large algorithms and allow for more parallelism and overlapping of operations between different processors. The resulting partitions can be implemented on several chips.

Partitioning is done before architectural synthesis and involves determining the number of chips to be used and dividing the behavior of the system among the chips as well as on separate on-chip modules. An advantage of partitioning is that it reduces the amount of hardware needed for every partition It may also reduce the global wirelengths by grouping the interdependent parts together. Partitioning can also improve the parallelism of the architectures [44]. The several methods used to implement partitioning of algorithms mimic the solutions proposed for the scheduling problem, the most common of which are presented in Section 3.2.2 [45].

3.2.6. Functional pipelining

Pipelined hardware allows the execution of more than one operation on the same unit. The use of pipelined FUs complicates the synthesis procedure where additional constraints need to be taken into account. Data dependencies and resource conflicts should be resolved and resynchronization should be used to avoid stalling the pipeline. This fact makes the pipeline scheduling problem NP-Complete [46]. Pipeline scheduling can be achieved when scheduling the operations by overlapping their SFGs as in *functional pipelining*. *Behavior pipelining*, on the other hand, uses retiming transformations [47] to guarantee the correctness of the algorithm. Several methods to obtain sub-optimal pipeline synthesis results are used most of which use some sort of allocation table to check resource conflicts [48].

3.3. Conclusion

Synthesis has been used in DSP and numerical algorithms. ANNs have not been addressed explicitly. Due to their high parallelism, regular nature and the large number of operations, special attention for their synthesis algorithms and hardware resources need to be addressed.

The use of automated synthesis tools in the design of ANN hardware can reduce the design time and enhance the performance requirements of networks that have to meet stringent real-time constraints. Several scheduling methods exist for the synthesis of hardware from high-level descriptions and heuristic-based *list scheduling* is the best choice for synthesizing the large algorithms typical of neural networks.

Chapter 4:

Single and Multi-Processor Neural Network Architecture

4.1. Introduction

This chapter introduces the architecture model proposed in this thesis to implement neural networks. The discussion highlights the similarities and differences between ANN and DSP algorithms. An overview of the architectural styles used in DSP hardware implementation and synthesis is presented. The proposed architecture for ANNs is then introduced, the architectural trade-offs achieving different degrees of parallelism and the optimizations involved are presented and discussed. Examples of special functional units which are specially tuned to ANN implementations are shown and an example data path is given along with its synthesis description. The issues involved in the pipelining of the units used are also laid out. Moreover, this chapter serves as a precursor for the next chapter which describes an automated approach to compile a general ANN onto this architecture. The synthesis results showing the trade-offs and optimizations introduced in this chapter are shown in chapter 6.

4.2. DSP-oriented architectural styles

ANNs are special signal processing algorithms with a high degree of interconnectivity, special activation function implementations and large memory storage requirements. Some neural networks such as the *adaline* evolved directly from adaptive filtering applications. The adaptive digital filter shown in Figure 4.1 is similar to single layer perceptrons (or adalines) as described in [49]. The difference is that the inputs x_1 to x_L in the adaptive filter are delayed versions of the input x_0 whereas in an adaline the inputs in general can be independent. The similarities in the algorithms make DSP style architectures suitable for implementing ANNs especially if specific care is taken to accommodate the differences and optimize the architectures for neural networks. Figure

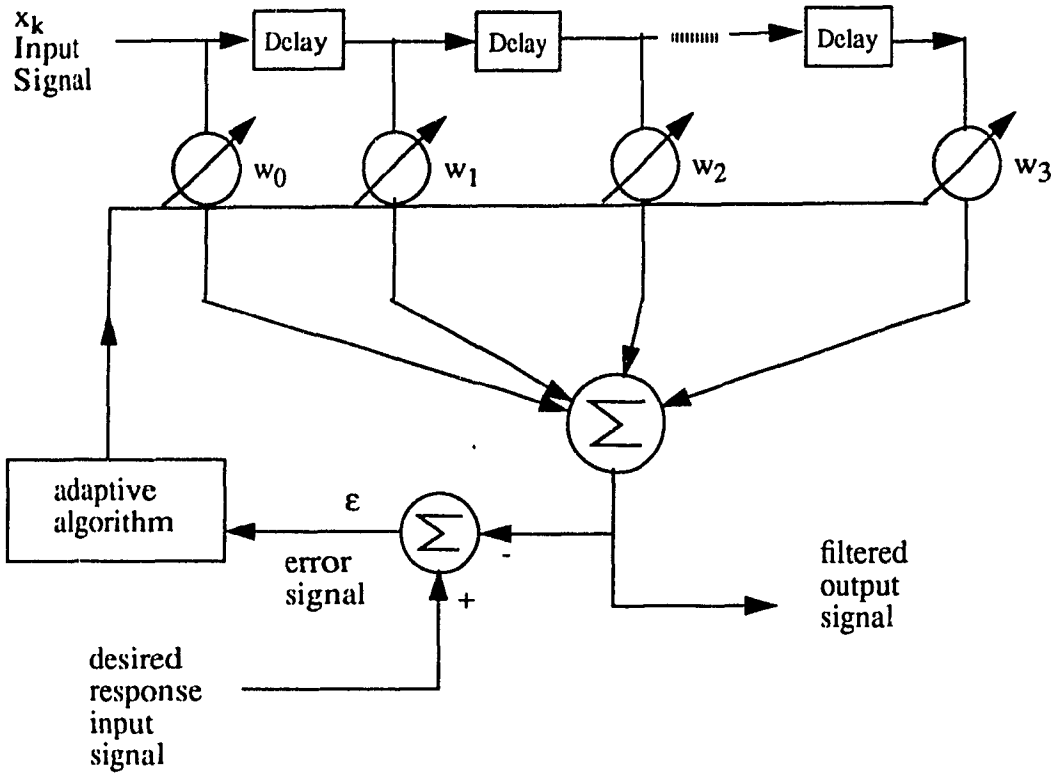


Figure 4.1: Adaptive digital filter

4.2 shows the SFG of a typical second order filter [50] where the boxes correspond to unit delays. Figure 4.3, on the other hand, shows the SFG of a small 2,2,2 BP network (the recall and learning modes). The boxes indicate delays for the update of the weight values. Even though the filter is not adaptive, it is typical of the DSP algorithms used in synthesis and the comparison with the general BP algorithm to be synthesized need to be made. The differences between the two algorithms can be seen in the sheer number of operations even in a very small BP network and in the fanout of some operations that suggest the parallelism involved. ANN algorithms as such involve a large number of variables and the large number of operations necessitate a large number of temporary variables and their corresponding storage.

DSP architectures as investigated by synthesis tools have evolved in four different directions as discussed in [51].

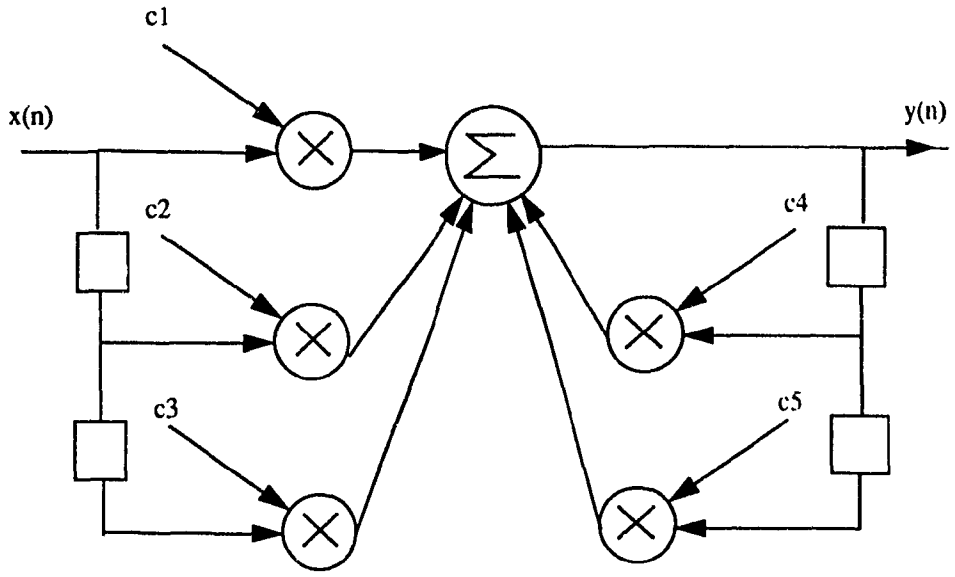


Figure 4.2: SFG of a typical second-order filter

Hard-wired bit-serial architectures: These architectures are suitable for applications with low sample rates. Signals are processed bit by bit. This style has the advantage of consuming less area. Problems with this style of architecture, however, include the difficulties in synchronization at higher clock rates. While the number of FUs in a bit-serial implementation is independent of the word length, the number of registers is increased with the word length. This can dramatically increase the storage requirements of operations.

Microcoded processors: These are suitable for low to medium rate algorithms. Processors in these architectures are dedicated FUs controlled by a microcode controller. The FUs include ALUs, multiplier-accumulators, RAMs, address-computation units, comparators and ROMs. These units can be pipelined. The architectures belonging to this style are programmable and may provide for more flexibility than is generally required in specific DSP algorithms.

Bit-Sliced multiplexed data-paths: These are suitable for irregular and recursive high-speed algorithms. They use a hierarchical control. Timesharing of hardware in this

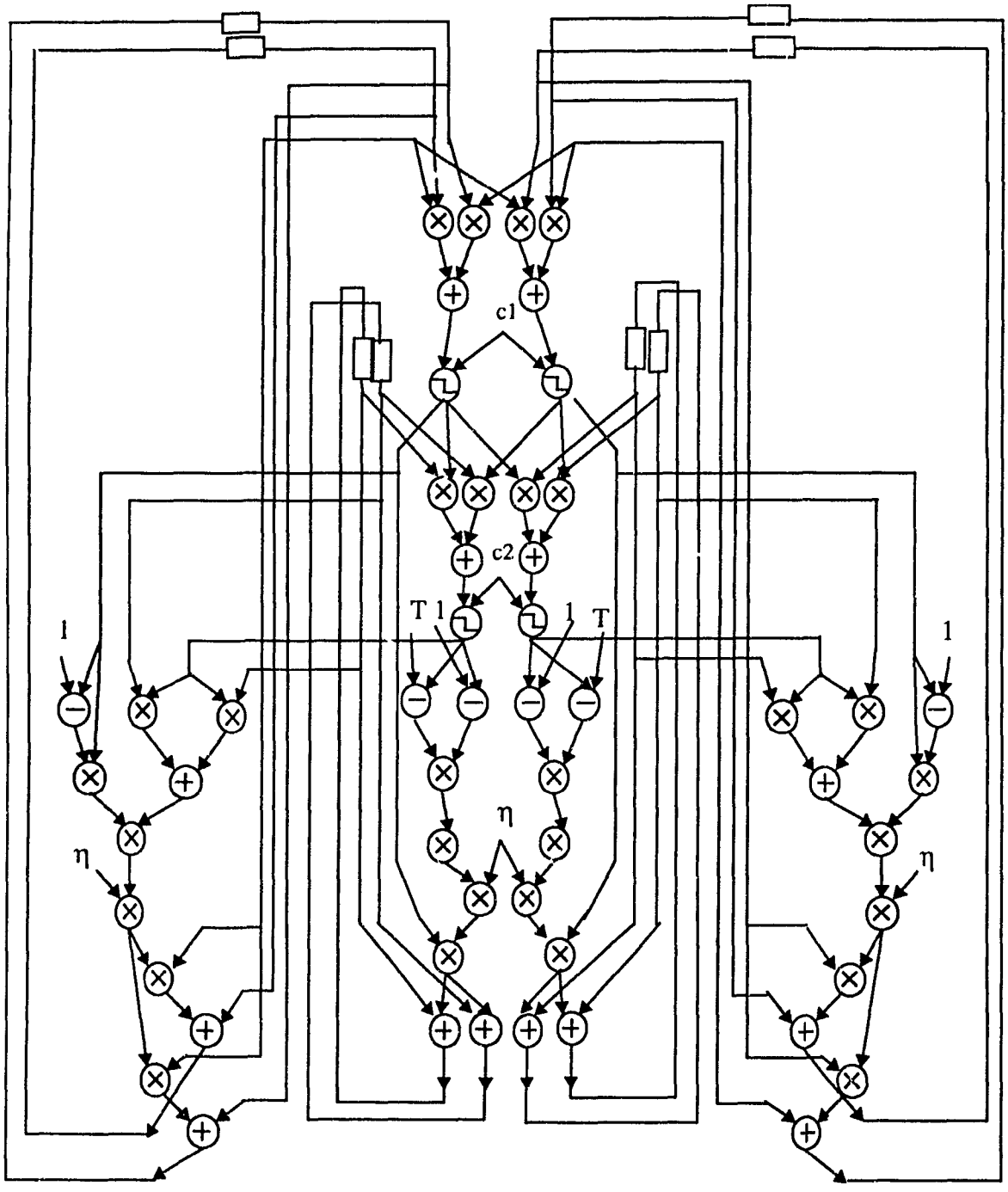


Figure 4.3: SFG of a 2,2,2 BP network

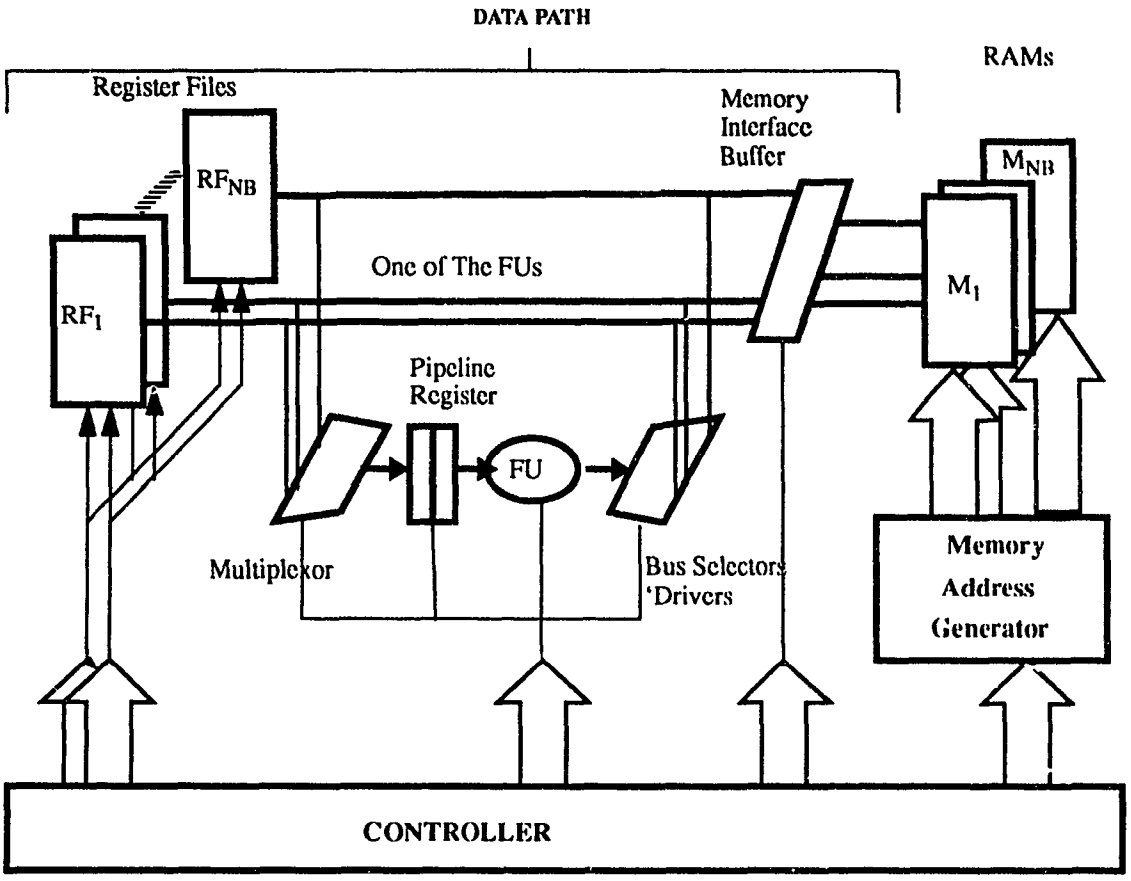


Figure 4.4: Multiple Bus/FU architecture

style of architectures is limited. The control flow is simple and typical implementations of the controller use a finite state machine rather than a microcode controller.

Regular array architectures: These are modular and can achieve very high rate. Regularity of the processing elements (PEs) and of the interconnection networks is important for modular, flexible systems. Systolic architecture are a special case of these systems. They are highly modular and heavily pipelined [33].

4.3. Bus style neural network data path

The architecture proposed, shown in Figure 4.4, is a multiple-bus architecture that uses the technologically proven bus-oriented interconnections. It originates from a synthesis system for DSP algorithms [52], and is used in this thesis to implement ANN

algorithms [53]. It is a *virtual implementation* of neural networks and partitions the neurons onto the available processors rather than providing a processor for each neuron. Virtual implementation of ANNs is the mapping of more than one neuron onto the same processor as explained in [4]. The architecture consists of *nb* busses, *nfu* functional units (FUs) (adders, multipliers, etc...), and *nr* storage registers that could be grouped in register files or RAMs.

Other architectures were used for the design of artificial neural networks such as the one proposed by Vlontzos and Kung [54] to implement the processing elements (PEs) of their systolic design. Such proposals, however, implemented the network as one processor per neuron rather than allowing for some parallelism as in virtual implementations. The architecture proposed in these thesis includes internal parallelism in the processor providing for multiplexing different neuronal computations on the same processors.

After the synthesis, the number of FUs, busses and registers as well as the interconnections is reduced from an initial maximum allocation to an optimum value. The resulting processor architecture would have one RAM (or register file) connected to each bus. FU inputs are connected to some or all of the busses through multiplexers, their outputs through tristate bus buffers. The architecture uses a two phase clock cycle (Φ_1: Read, Φ_2: Write) which results in an efficient use of the busses and in the grouping of registers into one register file per bus. A microcode controller (generated by the synthesis tool) produces one control word per clock phase.

4.4. Suitability for synthesis

The advantages of using bus style architectures for synthesis include the accuracy of the delay estimates. The critical path delay in random data path topologies cannot be accurately estimated. Furthermore, this architecture is flexible. It is also possibly programmable which is important in adaptive applications of ANN implementations. It

allows several degrees of parallelism giving more freedom in selecting an optimum architecture to implement the massive parallelism of ANNs. It implements straight code algorithms avoiding the overheads incurred by the generality of providing for branching which is not needed for the most common ANN algorithms. It further provides parallel access to the stored values during execution, an important issue during the training phase of ANNs where the network's weights are to be updated. This architecture is scalable and expandable into a multiprocessing environment as will be shown in Section 4.7, which is important for the implementation of large neural networks. The architecture is also easily testable where testing can be done directly through the I/O ports (Section 4.8). A certain amount of fault tolerance can be easily integrated with this style of architectures as shown in Section 4.9.

4.5. Special functional units

The above architecture can be used to implement any computational algorithm. Using it in artificial neural networks, however, provides for optimizations based on exploiting the characteristics of neural network algorithms. The most important optimizations are those that involve the use of special functional units such as the ones explained below.

4.5.1. Multiply-accumulate unit

The abundance of vector-matrix operations in ANN algorithms make it necessary to provide a special FU for these computations. Forcing the columns of the matrix w_{ij} in Equation (2.1) to be accumulated on one unit saves on data transfers and temporary storage of intermediate values. Thus the required FU is a pipelined multiply-accumulator (**MAC**) such as the one shown in Figure 4.5. The MAC has a local storage register *lr* or accumulator (Section 4.6.2) for the MA (multiply-accumulate operation). Figure 4.6 has a different MAC organization that includes a threshold function which is essentially a truncator used to implement activation functions such as the sigmoid. It also has more than

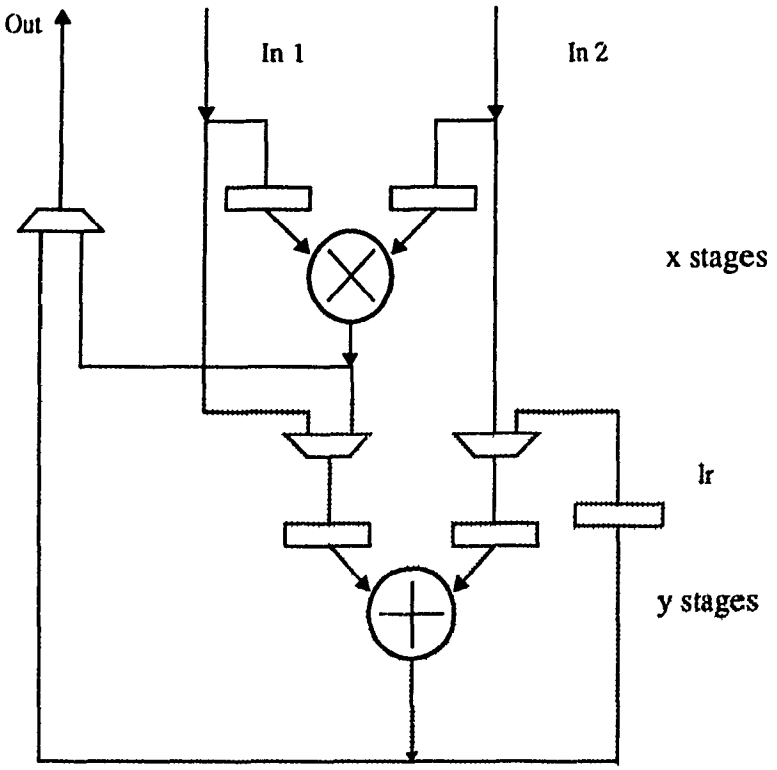


Figure 4.5: Regular MAC

one accumulator. The advantage of using more than one accumulator is shown in Section 4.6.2. Adding the thresholding unit in the MAC rather than providing a separate FU results in a more efficient transfer function implementation. Separate FUs, such as Comparators, Lookup Tables, ALUs and others are also supported. Providing for separate interconnects (other than the busses) between different FUs enables MACs to be constructed such as shown in Figure 4.7, where a distinct multiplier and a distinct adder are interconnected. This will enable these units to function as separate functional units or as a MAC. Overlapping the operations between the MAC, multiplier and adder units results in better utilization of these units. It should be noted that a bus-style architecture using such complicated functional units with local interconnects approaches random data path architectures while still providing all the advantages of fixed data path and bus-style architectures.

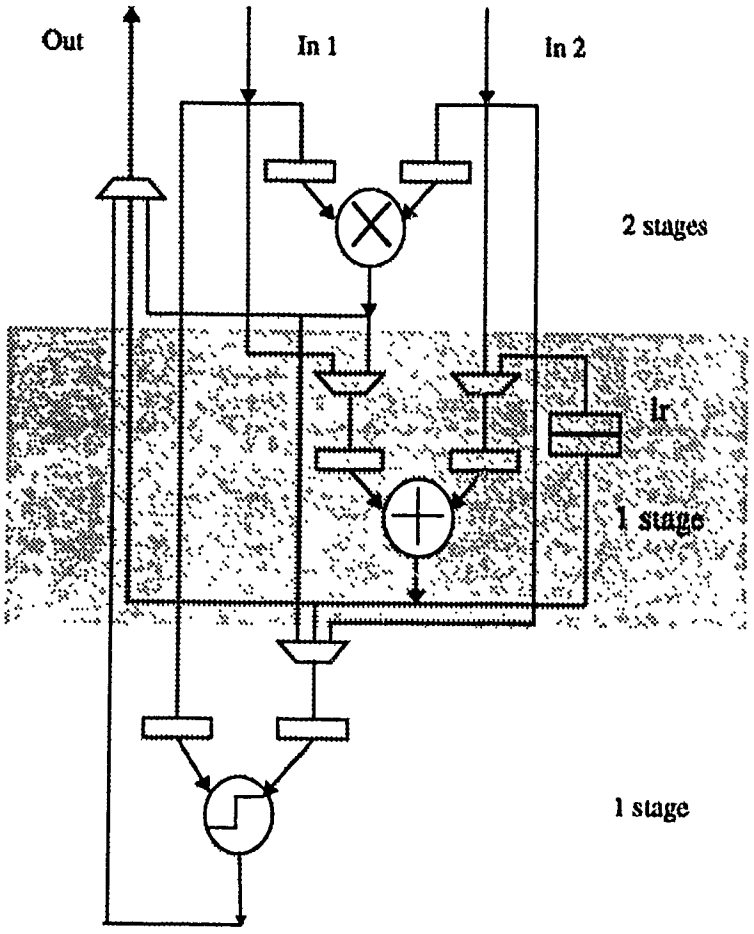


Figure 4.6: Multi-purpose MAC with threshold unit

4.5.2. Activation function implementation

The transfer functions used in ANNs can consist of simple hard limiters (truncators) which can be implemented on the MAC or geometrical functions such as the sigmoid which can be implemented by a lookup table or on the MAC as will be explained next.

A truncation or a thresholding function can be implemented by saturating the function at the required values. A general truncation unit, however, would require that these values be easily modified. For this reason a proposed scheme is shown in Figure 4.8 that can accommodate changes. To truncate a value S using the saturation values $\pm\alpha$ with a

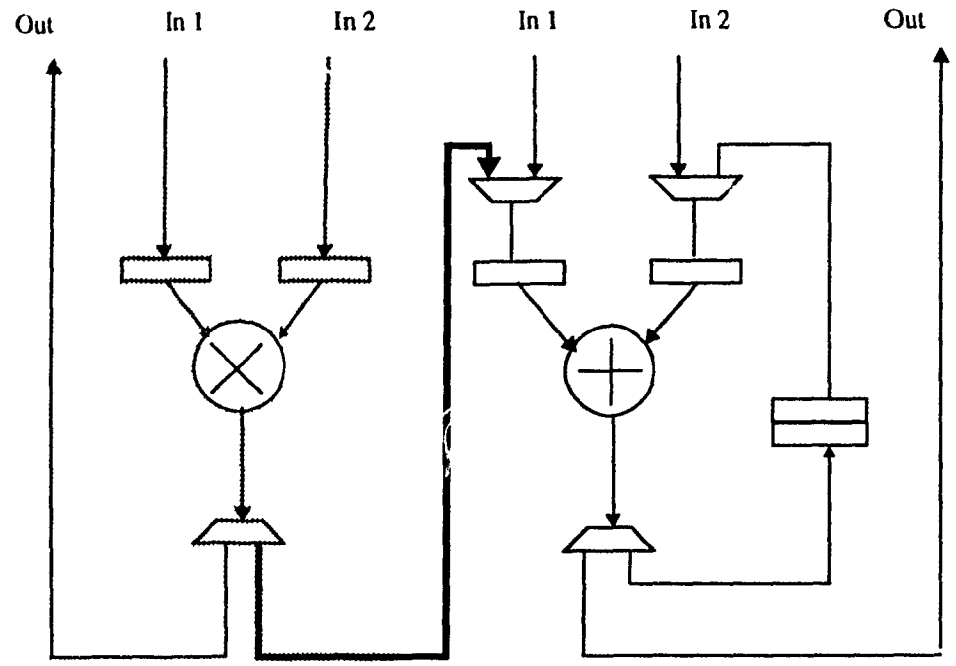


Figure 4.7: Multiplier and adder with local interconnection

hardware truncator at $\pm\beta$, then two multiplications are needed as shown in the figure. $\tilde{S}$ corresponds to the truncated value of S (i.e. + or - α). Appropriate biases could be added to obtain general truncations which are translations of the one shown in both the x and y directions.

To obtain a function with a linear region, rather than a simple cutoff, the truncation in the above description is multiplied by a certain coefficient representing the slope. A combination of several linear regions constitutes an approximation to the sigmoid function. In this thesis, a piecewise linear approximation of the sigmoid function is proposed, as the 3-region sigmoid, shown in Figure 4.14 along with its signal flow graph (SFG). The first three operations for each region correspond to those used in the general truncation unit of Figure 4.8. Appropriate shifting and linearization (multiplication by a slope) is done on all linear segments which, superimposed (added together) produce the sigmoid approximation. The linear approximation of the sigmoid function (or similar

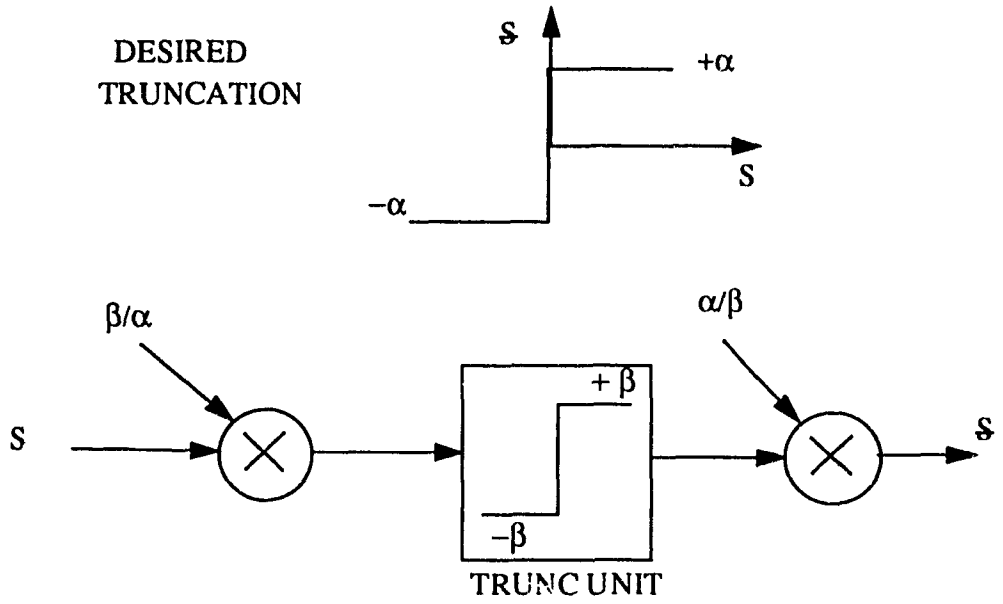


Figure 4.8: Truncation implementation

functions such as tanh) is then based on the following:

$$\begin{aligned} y_n &= \mu_n (x_n + \sigma_n) \qquad \text{for the linear region} \\ y_n &= \pm\alpha_n \qquad \text{for the cutoff regions} \end{aligned} \tag{4.1}$$

where n is the number of linear regions. μ_n is the slope of the linear region n, σ_n is the value of the shift for segment n, and α_n is the truncation value. The final result of the activation is then:

$$f(x) = \sum_n y \tag{4.2}$$

Recall that a shift in the y-direction is also required to obtain a sigmoid function with outputs between 0 and 2α (or other values). The values α_n, μ_n and σ_n are specified by the user according to the requirements of the function.

This is the most flexible implementation of the sigmoid possible and can be tailored to the user's specifications and modified accordingly for each neuron whereas most sigmoid implementations are rigid approximations of a fixed sigmoid function.

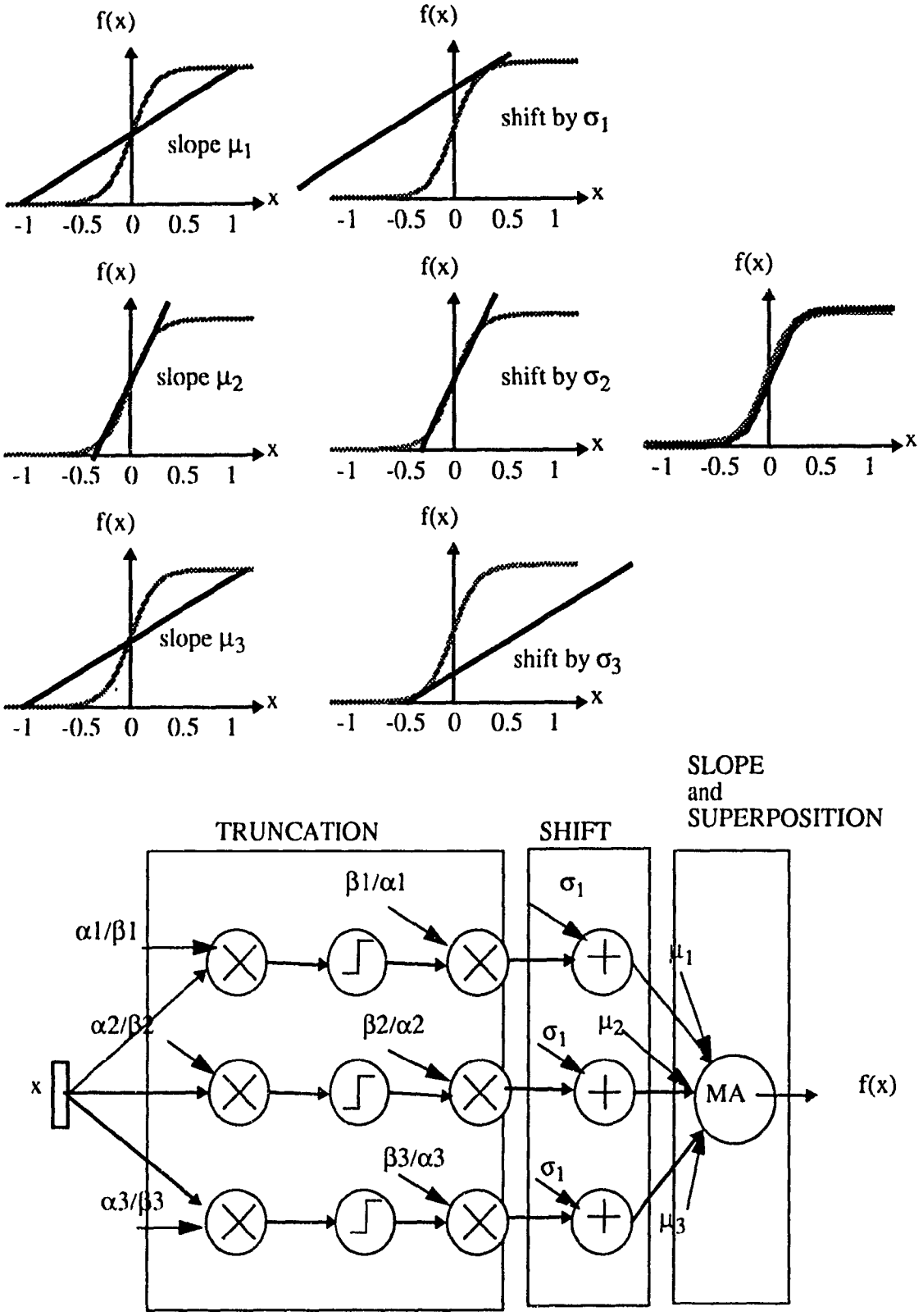


Figure 4.9: SFG for a linearized Sigmoid

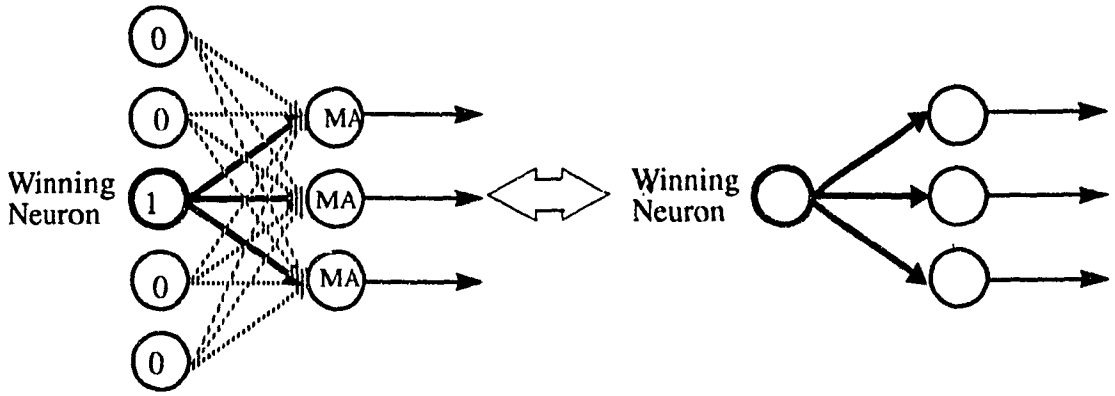


Figure 4.10: Output of competitive layer

4.5.3. Comparator for competitive neurons

Counterpropagation, and similar networks, require a different style of activation functions than the ones previously described, namely, winner-take-all mechanisms where neurons receive inhibitory signals from neighboring neurons and, as a result, only one neuron (the one with the largest value) is activated. An implementation of such a mechanism normally requires a *sort* procedure on all neurons that changes the value of all loosing neurons in the layer to *zero*. The activation value of the winner is then set to *one*. The outputs are then calculated using MAs as in a regular layered perceptron.

Such an implementation in hardware requires information on the storage locations of all neurons. This constrains the controller or, alternatively, requires the transmission of corresponding addresses along with the needed data. These schemes do not fit into the architectural style proposed in this thesis which implements ANN algorithm as straight code segments. In order to implement CP networks with winner-take-all mechanisms, a different approach to the sorting process is used.

It should first be noted that, in CP networks, since only one neuron is activated with a value of one, only the weights corresponding to that neuron are output as shown in Figure 4.10. A result of this implication is that a substantial saving can be achieved by avoiding the MA operations (since most of the constituents are zero) and simply output the

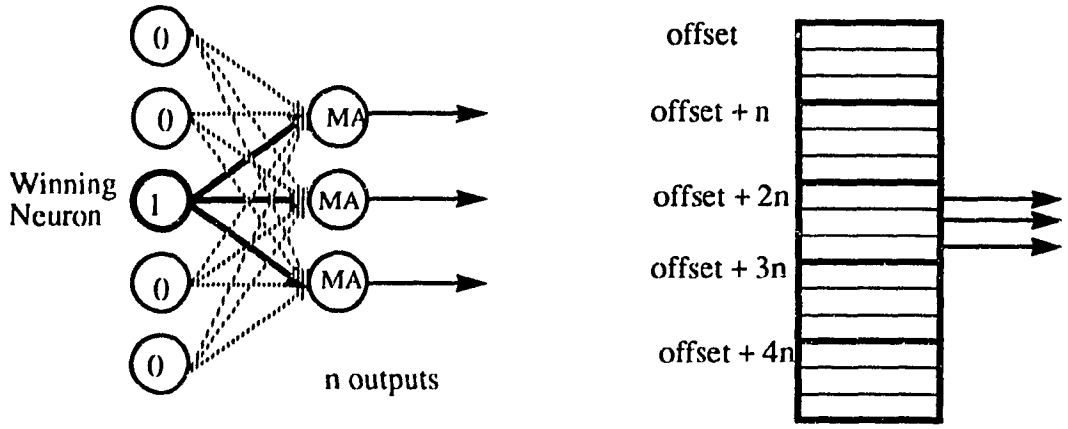


Figure 4.11: RAM location for output weights of competitive layers

weights corresponding to the winning neuron. This eliminates all the unnecessary multiplications by 0 and 1. So the sorting procedure need only find the right neuron (no requirements to inhibit the others or set their values to zero) and its corresponding output weights. For this, all the weights are stored in blocks with known address offsets as shown in Figure 4.11. A comparator is used to select between neurons and adjust the weight address offsets accordingly.

The comparator used in thesis to implement the winner-take-all algorithm in CP networks is shown Figure 4.12. The greater of the two inputs could be stored in the local register and compared with a subsequent input. The address is incremented when a new value is stored in the register. This requires the weights connecting the Grossberg and Kohonen layers of a CP networks to be stored in the order of the comparisons. Indeed effective addressing can be guaranteed by the use of the synthesis tool. Variations on this scheme are possible. Comparing 3 values at once (2 inputs and the value in the local register) with appropriate indexing and the use of several constant for the addition can speed up the procedure. Using more than one comparator is possible and the synthesis tool can generate a schedule for appropriate indexing. The use of such a scheme, specifically designed for CP networks, is clearly superior to sort procedures as explained above.

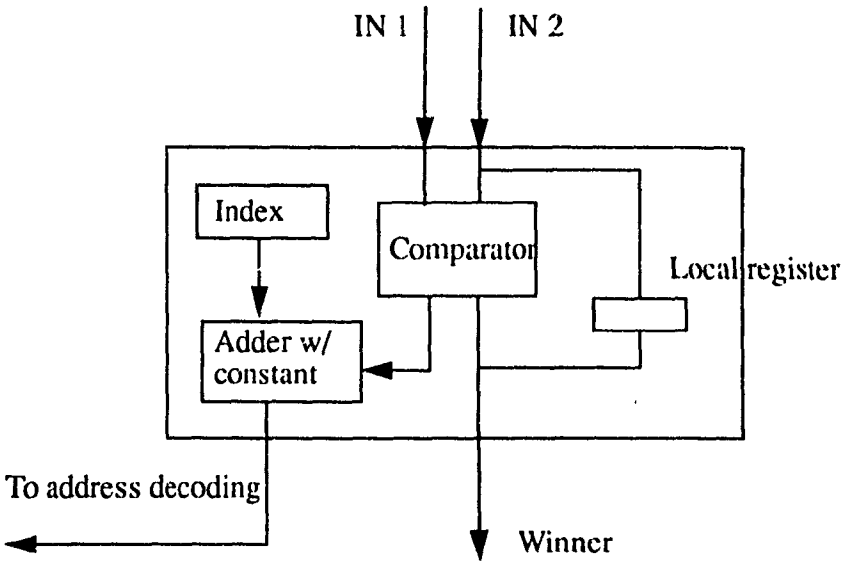


Figure 4.12: Comparator used in winner-take-all mechanisms

4.5.4. Table lookups and other special units

Table lookups can also be implemented to approximate activation functions. These units can be used as regular functional units with one input and one output each or can be interfaced on the busses and register files where they can be used on the write clock phase when a value of an output neuron is to be written in a register file and then retrieved, its activation value calculated and then is stored again.

Other units may be required for specific ANN algorithms and the architecture is flexible enough to accommodate a large style of special unit organizations.

4.6. Proposed architectural optimizations and trade-offs

4.6.1. Neuron splitting

Flattening the MA operations of neurons completely into series of multiplications and additions is not beneficial since it adds to the bus load and requires more time for intermediate data transmission, and extra storage. However, there are cases where the number of neurons in one layer does not divide evenly into the number of MACs available. This asymmetry increases the number of cycles for the ANN computation when

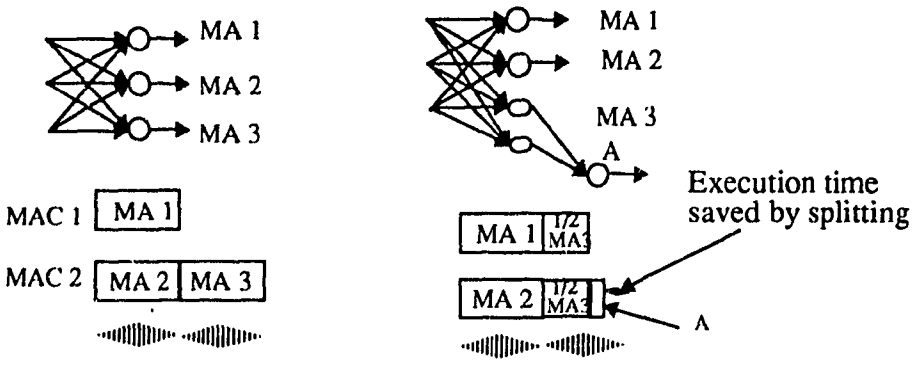


Figure 4.13: Splitting neuron to resolve asymmetry

idle MACs are present. In this case, splitting the MA evenly onto the number of MACs increases the speed (Figure 4.13).

4.6.2. Multiple word storage

MACs with multiple word storage allow more than one MA to be performed on the MAC at the same time. This tends to increase the performance of the system over the use single accumulator MACs when the number of busses is constricted. On the other hand, adding local registers for each MAC constrains the scheduling and could result in lost cycles. In the case of multi-stage adders within the MAC, special consideration need to be observed in order to prevent Read before Write hazards. This is shown in Figure 4.14 where the cases for one and two local registers are shown. For one *lr*, an extra delay is needed. This is guaranteed by a dynamic resource limitation list that scans the FU scheduling lists backwards by the total FU stage delay to adjust the initiation list. For the case of two *lrs*, however, other MA operations can be executed on the same FU. Since a MA is divided into small atomic MAs which can be interleaved with those of another operation, the delay is not needed. The use of local interconnects is beneficial in the case of MAs. Indeed, this approach can be generalized to any combination of operations saving on bus transfers. Section 5.3.1.1 shows several cases where combinations of operations can be used as a pre-scheduling optimization technique. The use of local interconnects, however transforms the bus style architecture being used into one that is practically a

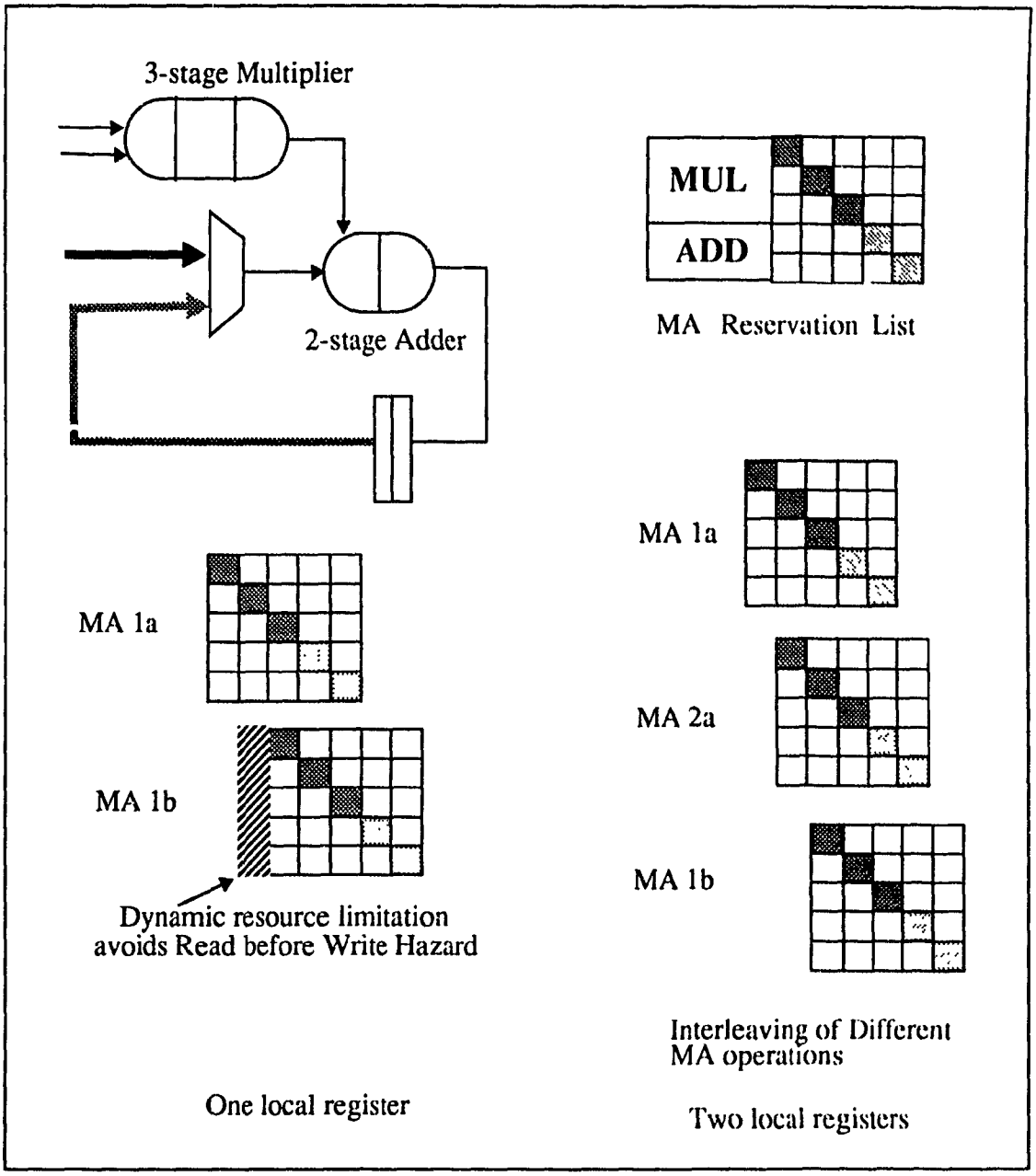


Figure 4.14: SFG for a linearized Sigmoid

random topology architecture. The architecture can, therefore be optimized further while the synthesis approach remains simple as explained in the next chapter.

4.6.3. Deep pipelining

FUs can have different pipeline levels depending on the details of their implementations. Increasing the number of stages of the multi-purpose MACs described

in Section 4.5.1 allows for more flexibility and possible higher throughput. The cycle time for a typical bus style architecture is calculated as follows:

$$T_{cy} = T_{Rd} + T_{Wr} + T_{delay} \tag{4.3}$$

where T_{cy} is the total cycle time, T_{Rd}, T_{Wr} the bus read and write times, T_{delay} the largest delay of the FU stages. For comparison purposes, typical delays are taken as follows:

$$T_{Rd} = 15ns, T_{Wr} = 15ns, T_{delay} = 30ns \Rightarrow T_{cy} = 60ns \tag{4.4}$$

When the number of stages of the MAC is doubled T_{delay} is reduced by half resulting in $T_{cy} = 45ns$. In this case, the performance of the whole system is enhanced by 25%.

4.6.4. Separate activation function units

Providing a separate unit to perform the calculation of the activation function rather than using the multi-purpose MAC yields a slight improvement and having more than one unit can further enhance the performance. For more complicated transfer functions, such as the one described above, the presence of additional threshold units does not improve the performance since multipliers and adders are also used.

4.6.5. Architecture saturation

The maximum number of busses that can possibly be used in this architecture is

$$NB_{max} = \sum_{t=1}^{M} I_t \sum_{i=1}^{N} FU_{ti} \tag{4.5}$$

where M designates the different types of FUs in terms of the number of inputs (The assumption is that the number of inputs is always less than or equal to the number of outputs), and N the number of FUs of each type used. I is the number of input ports per operation for the FU.

It is noted that the efficient use of the busses, however, results in the performance

of the architectures *saturating* well below that upper limit. Architecture saturation refers to the saturation of the speedup curves where the addition of busses does not increase the speed of the system.

4.6.6. Network topology

ANNs with the same number of neurons and connections can still have different topologies. The same architecture, then results in different speeds. This is particularly noticeable in BP networks where the algorithms for the recall and learning phases differ greatly in the amount and types of operations needed. Synthesis tools like the one proposed in this thesis are needed to select the best architecture for each network implementation.

4.7. Proposed multi-processor implementation

4.7.1. Systolic embedding

The regularity of ANNs makes them amenable to systolic implementations. Several architectures based on this approach have shown that systolic mappings of neural algorithms present a suitable solution. The existing systolic architectures, however, either use simple processing elements (PEs) with limited internal parallelism [23] or large, expensive general purpose processors [24]. Networks are usually implemented on parallel arrays with each PE implementing one neuron. Such parallelism may not be needed and the addition of an I/O port for each neuron results in significant cost to the overall system in area and power. As the technology advances, the speed of a simple processor will be very high which will make global synchronization difficult. By increasing the parallelism within each processor, the total number of processors is decreased thus maintaining a good ratio between internal clocks and data transmission clocks. *Our approach is investigating internal parallelism and the implementation of several neurons on each PE of the systolic array.*

Present systolic implementations of neural networks opt for simple processors

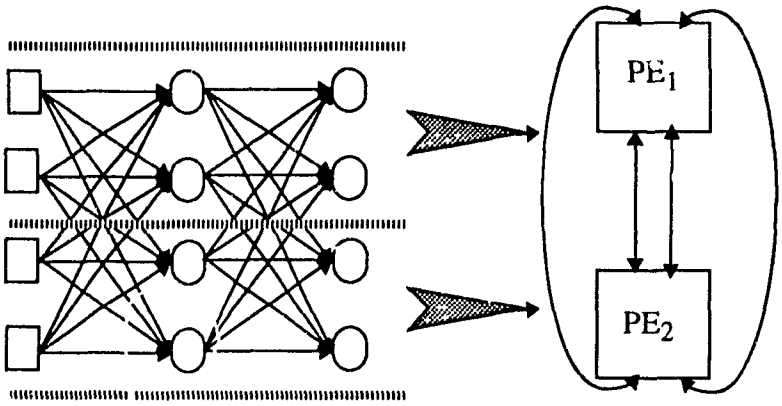


Figure 4.15: One-dimensional linear systolic configuration

(with no internal parallelism) rather than ones such as the architecture proposed in this chapter. They also map each neuron onto a different PE and cannot take advantage of virtual implementations. the reason for this is that systolization in this manner becomes simple whereas the use of complex PEs constrains the problem that it cannot be solved mathematically. The use of a synthesis tool, such as the one presented in this thesis, enables these complicated systolizations where the tool will systolize the algorithm as part of synthesis procedures. The tool, is guided by the user and information about partitioning and communication need to be specified. These issues will be discussed in detail in Chapter 5.

In this thesis, a systolic implementation is proposed which consists of single or multi-dimensional circular arrays. It uses the PEs with internal parallelism as described in Section 4.3 with added I/O ports for systolic interconnection. In a single circular array, the network is partitioned horizontally as shown in Figure 4.15 where a 4,4,4 Backpropagation network is partitioned onto 2 PEs. For multi-dimensional arrays, additional vertical layer-based partitioning is done. In its simplest form the partitioning for multi-dimensional arrays is done on perfectly symmetrical networks. In cases where the networks are not symmetrical special modifications such as splitting neurons or inserting dummy operations need to be performed. The intensity of the computations involved in

ANNs allows for the systolic communications to take place without affecting the performance of the system (Section 5.4.3). From the synthesis of a number of systolic architectures it is concluded that the number of I/O ports per PE does, however, influence the performance of the network due to the large inter-processor data transfers and the transmission delay. For BP networks in recall, the number of I/O ports should be equal to the transmission delay in terms of the PE clock. Details of the multi-processor synthesis are given in Section 5.4.

4.7.2. Input and output ports

The systolic interconnection requires efficient communication between the processors. This is allowed by input and output units using the same busses that the FUs use. To that end, descriptions of the I/O ports are similar to that of FUs. An input unit is therefore a functional unit that receives an input from outside on read bus clock and uses the write bus clock to write it to the register. An output unit receives the output value on a write bus clock and outputs it on a read bus clock. The delay between the reads and writes of the I/O units is in fact the interconnection delay and is modelled appropriately. The synthesis tool decides which busses are used and guarantees synchronization as will be explained in the next chapter.

4.8. Testability

Design for testability is an important issue in VLSI design. Structured approaches that allow for partitioning the design to obtain easier *controllability* and *observability* are needed [55]. The importance of partitioning systems provides bus-style designs with a clear advantage. This architecture allows access to busses which go to different modules. Testing a particular FU, for instance, would necessitate the use of the busses only. Therefore, the global busses used in the architecture proposed provide a way to both control the units they are connected to and observe the outputs of test sequences. Since the architecture does not assume any local interconnects, the testing procedures are much

easier and can be done from one particular test point interfaced to all the busses. Special considerations have to be taken into account so that all FUs are fully testable from the busses.

4.9. Fault-tolerance

The same reasons that allow easy testability of the architecture provide it with inherent fault-tolerance mechanisms. The first immediate solution to the fault-tolerance problem concerns the FUs where redundant units can be used. An architecture using 4 MACs, for example may include 6 MACs on chip for reliability purposes. This is a practice not uncommon in similarly styled architectures with complicated FUs [32]. A RAM controller can easily allow for disregarding certain memory words that may be dysfunctional. A slightly oversized RAM may be a good solution.

4.10. Conclusion

Several architectures have been proposed for the implementation of ANNs that try to exploit the parallelism of those algorithms. While most digital architectures implement one neuron per processing element it is clear that there is a need to investigate different degrees of parallelism within each processor. To that end a flexible bus-style architecture is proposed that can optimize the parallelism of neural networks with area constraints. This DSP oriented architecture is optimized for neural network applications by the use of special functional units. It is further extended into a systolic multi-processing environment suitable for large networks. The architectural trade-offs observed give clear indications of the different optimizations possible. It is then up to an automated synthesis tool to investigate these trade-offs for a specific applications and optimize the corresponding architecture.

Chapter 5:

Architecture Synthesis Methodology

5.1. Introduction

This chapter presents the architectural synthesis methodology conceived for the synthesis of neural network hardware. The synthesis framework is presented as a complete computer-aided design system for digital neural networks. The architecture synthesis steps are then explained and the various issues involved in the compilation, scheduling and allocation steps are presented. A new approach to the synthesis of compound operations such as the multiply-accumulate operations abundant in neural network algorithms is discussed. The synthesis methodology is then extended to a novel multi-processing synthesis environment based on regular structures and the issues dealing with partitioning and synchronization of such systems are presented.

5.2. Synthesis framework

The synthesis system presented in this thesis and diagrammed in Figure 5.1 starts with a specification of the network to be synthesized. This specification is written in an Axon-like language [4] that describes the size, topology and behavior of the network (A model and example of the description are shown in Appendix 1). The specification could also consist of a signal flow graph (SFG) or a combination of both descriptions. Architectural or high-level synthesis then consists of finding the optimum (fastest) schedule for the given network for a certain hardware allocation specified by the user. The synthesis further binds all operations involved in the execution of the network to the hardware units. The outcome of the synthesis is a register transfer (RT) description of the system. If the resulting schedule does not satisfy any or some of the constraints then the network is *redesigned* by providing different hardware allocations. The initial hardware allocations are done by the user (or expert system) and further refined by the high level

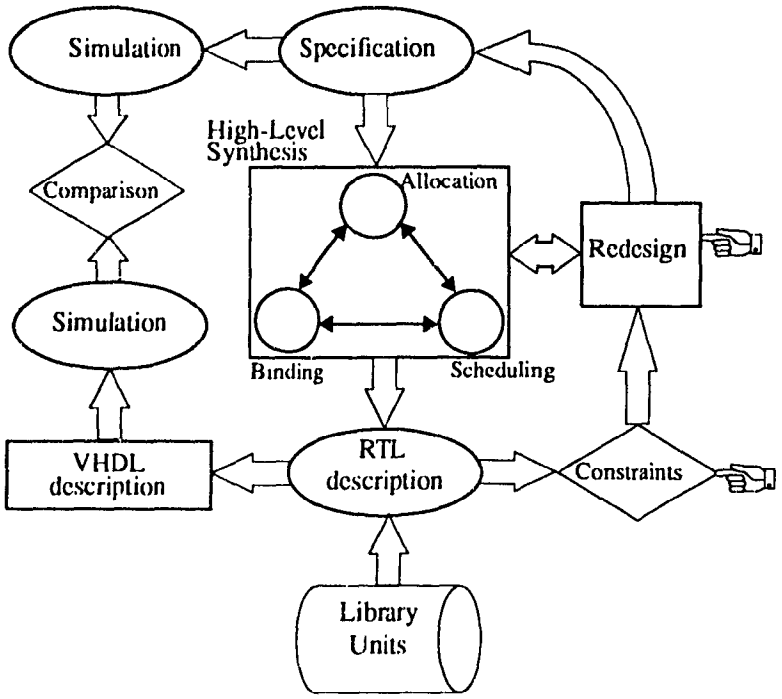


Figure 5.1: Architectural synthesis framework

synthesis procedures. The decision to accept the outcome of the synthesis or further search the design space is made by the user. The system, however, can be used to search the design space and produce area delay trade-offs for a range of architectures. The system can further be interfaced to a set of library units to get accurate measurements of the resulting VLSI implementations and eventually mapping the RT description onto the technology specific components present in the library.

An important aspect of any synthesis system is the ability to verify its correctness. The RT description obtained as a result of the synthesis is used to generate a VHDL description of the system at hand. Simulations of the networks at both the high abstract level (ANN simulator) and at the architectural hardware level (VHDL) are then compared. Further, the VHDL system description can check for inconsistencies and errors such as memory and bus contentions.

The synthesis tool presented as part of this synthesis framework was implemented

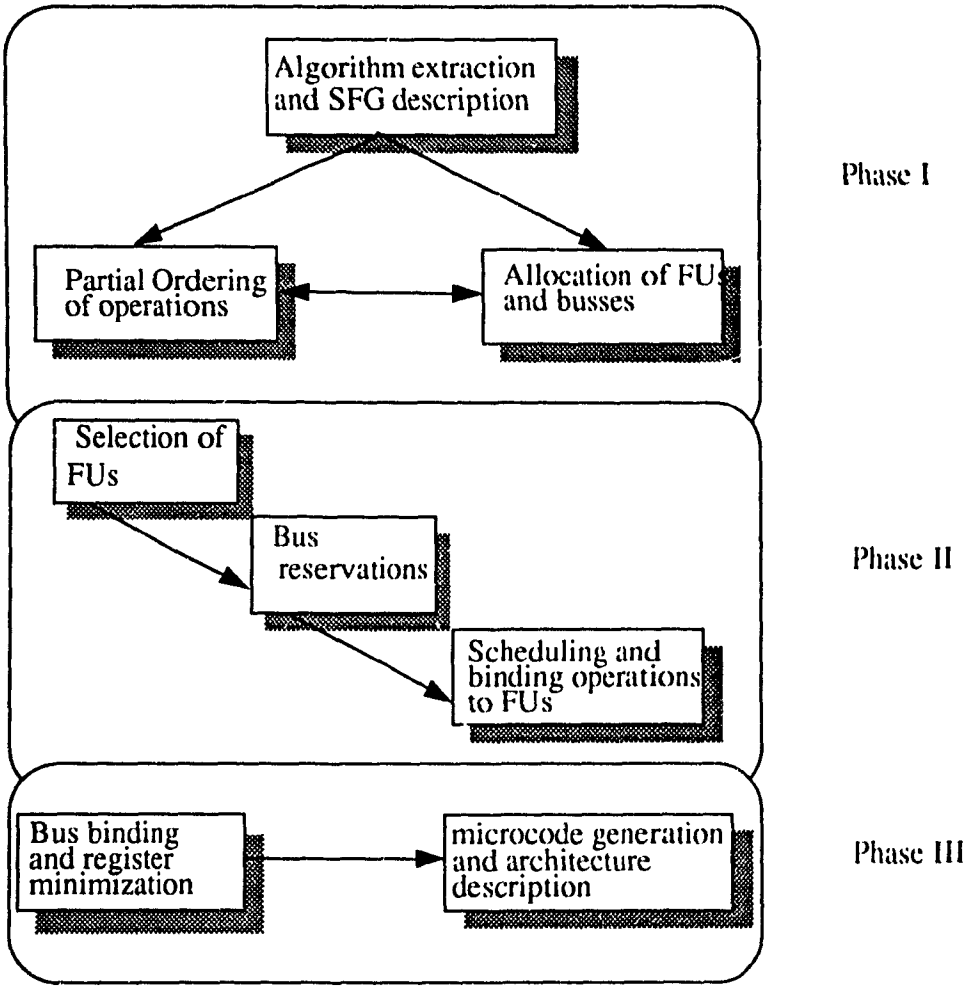


Figure 5.2: High-level synthesis phases

in Prolog and the algorithms used as well as all the issues involved in this synthesis tool are discussed below. The results of example simulation runs are discussed in chapter 6.

5.3. Architecture synthesis methodology

5.3.1. Synthesis of artificial neural network hardware

The high performance requirements of ANNs add to the complexity of their silicon implementations. In order to meet the high computational power demand, design aids have to be developed. The architectural synthesis tool presented is written in Prolog and based on the multiple-bus/FU architecture described in chapter 4. The tool is interactive and uses

user-defined heuristics to obtain an optimal design. It is divided into three major synthesis phases described below and shown in Figure 5.2.

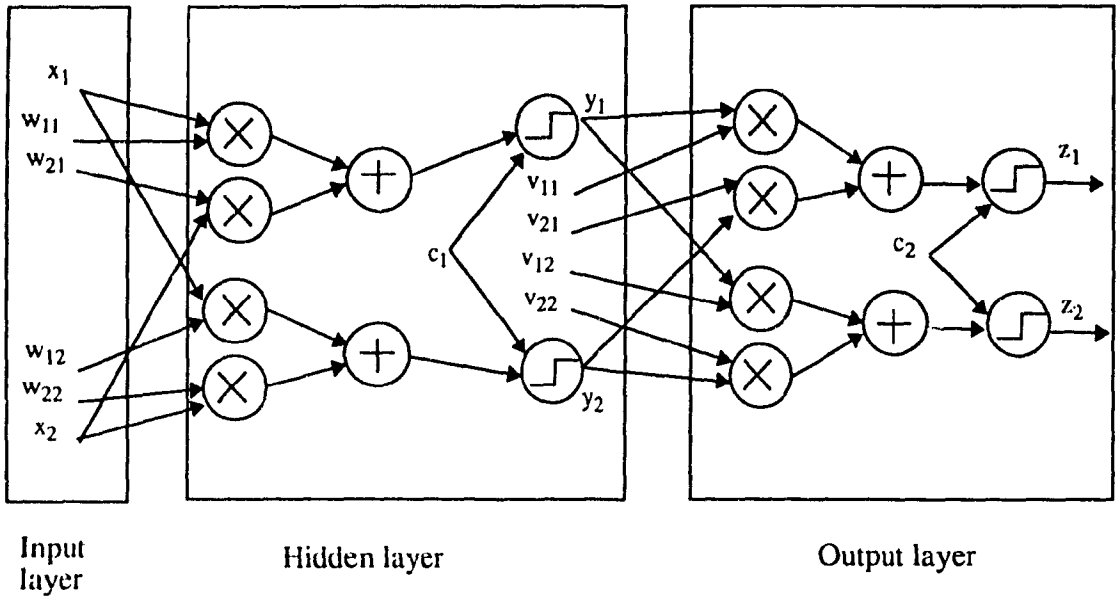


Figure 5.3: SFG for the recall phase of a BP network

5.3.1.1. Network description and pre-scheduling transformations

The first phase (Extractor) transforms a neural network description, specifying the type, topology and interconnections, into a SFG as an intermediate form (which can also be used as input). The network description is modelled after the Axon language presented by Hecht-Nielsen but is oriented for hardware syntnesis rather than algorithm simulation. A model description is shown in Appendix I. The input to the tool can also consist of a general SFG, formed of the operation set and the connections set, describing the network's operation or a combination of both descriptions allowing easy modification of the standard ANN algorithms. An example SFG for a 2, 2, 2 BP network in the recall phase is shown in Figure 5.3. The activation function implementations are shown as simple truncations according to a constant value. In the case of BP networks and most layered networks, the

$$W_{rp,j}(n+1) = W_{rp,j}(n) + \eta \delta_{p,j} X_{r,i}$$

$$\delta_{p,j} = X_{p,j}(1 - X_{p,j}) \left(\sum_{q} \delta_{q,k} W_{pq,k} \right)$$

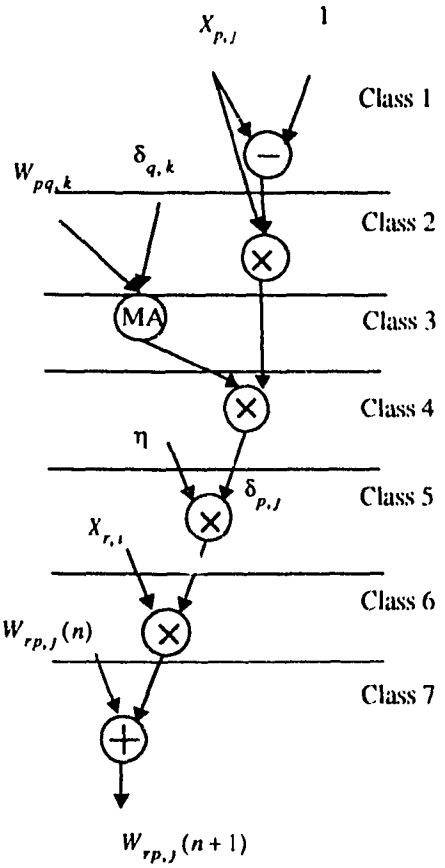


Figure 5.4: Generalized Delta rule (hidden layers)

data dependence is imposed by the layered structure of the network. The set of operations for each layer associated with each neuron is further divided into classes of operations performed by all neurons as shown in Figure 5.4 for the generalized delta rule (applied to the hidden layers) used to train BP networks. A class is a set of similar operations that can be executed concurrently. The use of classes reduces the complexity of the generation of operations and their pre-ordering since the classes incorporate a number of similar

operations. The classes are pre-ordered to achieve better reservation schemes based on the number of interconnections of a neuron $I(n_i)$ which is assumed to correspond to its execution time $T(n_i)$ for a neuron n_i. The number of operations per neuron increases with the number of connections and therefore this measure gives priority to neurons with the longest corresponding operations delay (Section 5.3.1.2).

The ordering of the operations within a class ensures a good scheduling scheme by taking advantage of the inherent parallelism of classes while the ordering of the classes themselves ensures correct data dependence. For the recall modes, the ordering includes all the MA operations of a layer followed by the set of activation function operations. The operations of the following layers follow in the same manner. In the learning phase, the operations are ordered as partially shown in Figure 5.3. The forward propagating phase with operations being the same as in recall is followed by the backward propagating phase with classes of subtract, multiply, MA and add operations. The classes 3 and 4 of Figure 5.3 are interleaved so as to achieve correct data dependence since the output of the multiplication for class 4 is needed in the input of the MA of class 3 for other neurons. The rest of the classes are ordered consecutively.

The reservations corresponding to two consecutive classes can overlap without violating the data dependencies. The types of operations as well as the type of hardware units to be used is determined at this phase. For instance, a pre-scheduling optimization may consist of grouping the multiplications and additions in the SFG of Figure 5.3 into MA operations (described in Section 5.3.1.2) which will result in the SFG shown in Figure 5.5. Indeed, any two or more operations can be grouped together to save on bus transfers as explained in Chapter 4. .In Figure 5.4, it can be seen that a number of operations can be grouped together such as the multiplications in classes 5 and 6 creating a MM (multiply-multiply operation) or classes 1 and 2 where one of the inputs is common resulting in a SM (subtract-multiply) operation. Even classes 5, 6 and 7 can be grouped together producing a MMA (multiply-multiply-add). While these operations may not be as

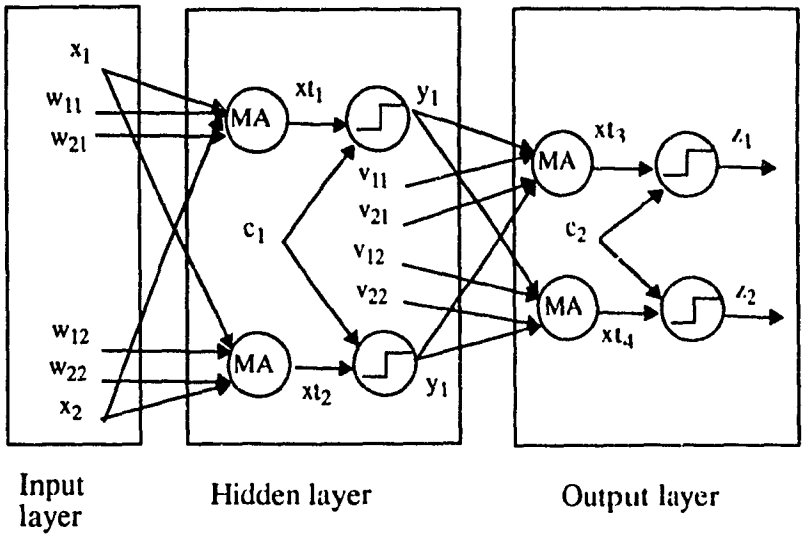


Figure 5.5: SFG of recall phase of BP network using MA operations

beneficial as the large MAs (which benefit is shown in chapter 6), they nevertheless result in more compact schedules and should indeed be investigated.

Further optimizations and algorithmic transformations are performed at this time such as neuron splitting which is described in section 3.3.

5.3.1.2. Scheduling, FU binding and bus reservation

The SFG is the input to the second phase (Scheduler). It is a finite directed graph *G(N,E)* where $N = O \cup C$. $O = \{o_1, ..., o_n\}$ is the set of executable operations including input and output port operations. $C = \{c_1, ..., c_m\}$ is the set of non-executable operations such as constant value assignments. Each operation $o \in O$ is described by $(t(o), i(o), u(o), s(o), e(o), h(o))$ where:

t(o) is the type of the operation,

i(o) is the input list usually containing two values except for the compound MA operations

u(o) is the output list with one datum except for complex numbers (Real and Imaginary parts) and special functional units

$s(o)$ is the starting time of the execution of the operation

$e(o)$ is the end time of the execution of the operation

$h(o)$ is the identifier of the hardware unit to which o is bound.

$E = R \cup D$ is the set of edges consisting of a register set $R = \{r_1, ..., r_e\}$ and a dependence set $D = \{d_1, ..., d_e\}$. Each register $r \in R$ is described by $(rd(r), wr(r), op(r), rf(r))$ where

$rd(r)$ is the register's read time from the register files

$wr(r)$ is the register's write time from the FUs to the register files

$op(r)$ identifies the operation in which this register is used

$rf(r)$ defines the register file in which it is stored and its index within that register file (address)

Registers are subject to the constraint $wr(r) < rd(r)$ except the weight registers, for backpropagation (BP) networks for instance, that will be updated after they are read or any other registers used in a feedback-style operation where the write time corresponds to a *write* in the previous sample period. The registers correspond to each edge in the signal flow graphs. The data dependence set is also used to describe compound operations such as MAs. An MA operation is described by its own SFG in terms of multiplications and additions as shown in Figure 5.6 . A is the set of edge lists (arc lists) describing each MA. $a \in A$ is described by $((x_1,w_1),(x_2,w_2),...,(x_n,w_n))$ where the calculation performed by the MA is $y = \sum_{i}^{n} x_i w_i$.

Scheduling consists of assigning each of the nodes in the SFG to a hardware unit minimizing the number of cycles T_{cycle} it takes to execute the whole set, respecting data dependency. For a N-layer network $L_0, L_1, ..., L_{N-1}$ the scheduling time of the tasks corresponding to layer $L_k \in \{L_0, ..., L_{N-1}\}$ should satisfy $\sigma(S_k) < \sigma(S_{k+1})$ where S_k is the schedule and $\sigma(S_k)$ is the schedule initiation time. The hardware description $H = FU \cup B \cup I \cup O$ consists of a set of FUs a number of busses and a set of I/O ports. The operation set is scheduled on the available FUs and busses using a list scheduling

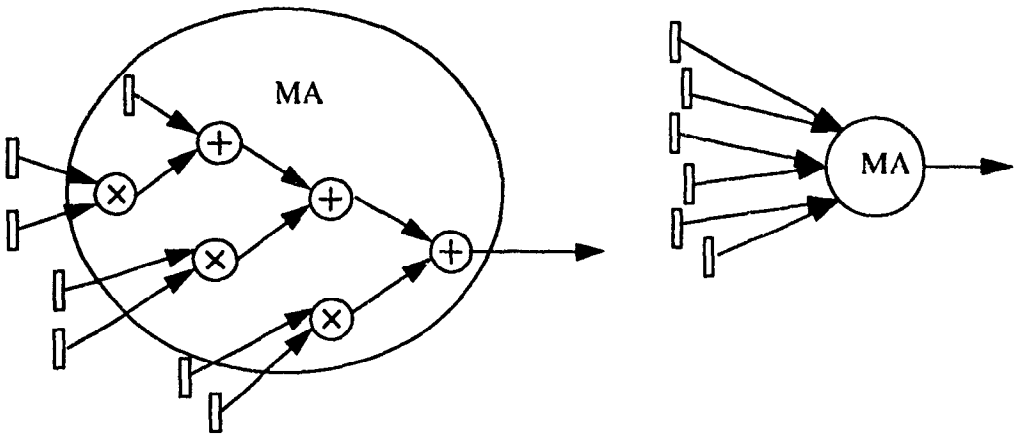


Figure 5.6: SFG of a MA operation

approach. The list of operations is pre-ordered according to their priorities as explained above. The operation with the highest priority is scheduled on the first FU of its type. In order to ensure correct scheduling several issues have to be addressed.

The operation to be scheduled (usually) has two input values from registers with availability times *write-clock(1)* and *write-clock(2)* corresponding to the time they were written (those operations with write-clock times in the previous sampling period would not include this extra constraint and would have write-clock times of zero until they are written at a later time in the scheduling process). The availability time clocks of all FUs corresponding to the operation being executed are then obtained and binpacking heuristics are used to select the FU that results in the best possible utilization as explained later. The FU's availability clock is then *FU-clock*. The new reservation clock of the FU is therefore subject to *reservation-clock* = *max(write-clock(x), FU-clock)* for *x=[1,2]*. This translates into initiating the operation as soon as the second input is latched by the FU without constraining the arrival time of the first input. Further, each of the variable inputs needs to be transmitted from the register files to a FU latch. The availability of the busses need to be guaranteed. While the FU availability lists are scanned to find the last available times, the bus lists are scanned from the availability times of the register *write-clock(1)* and

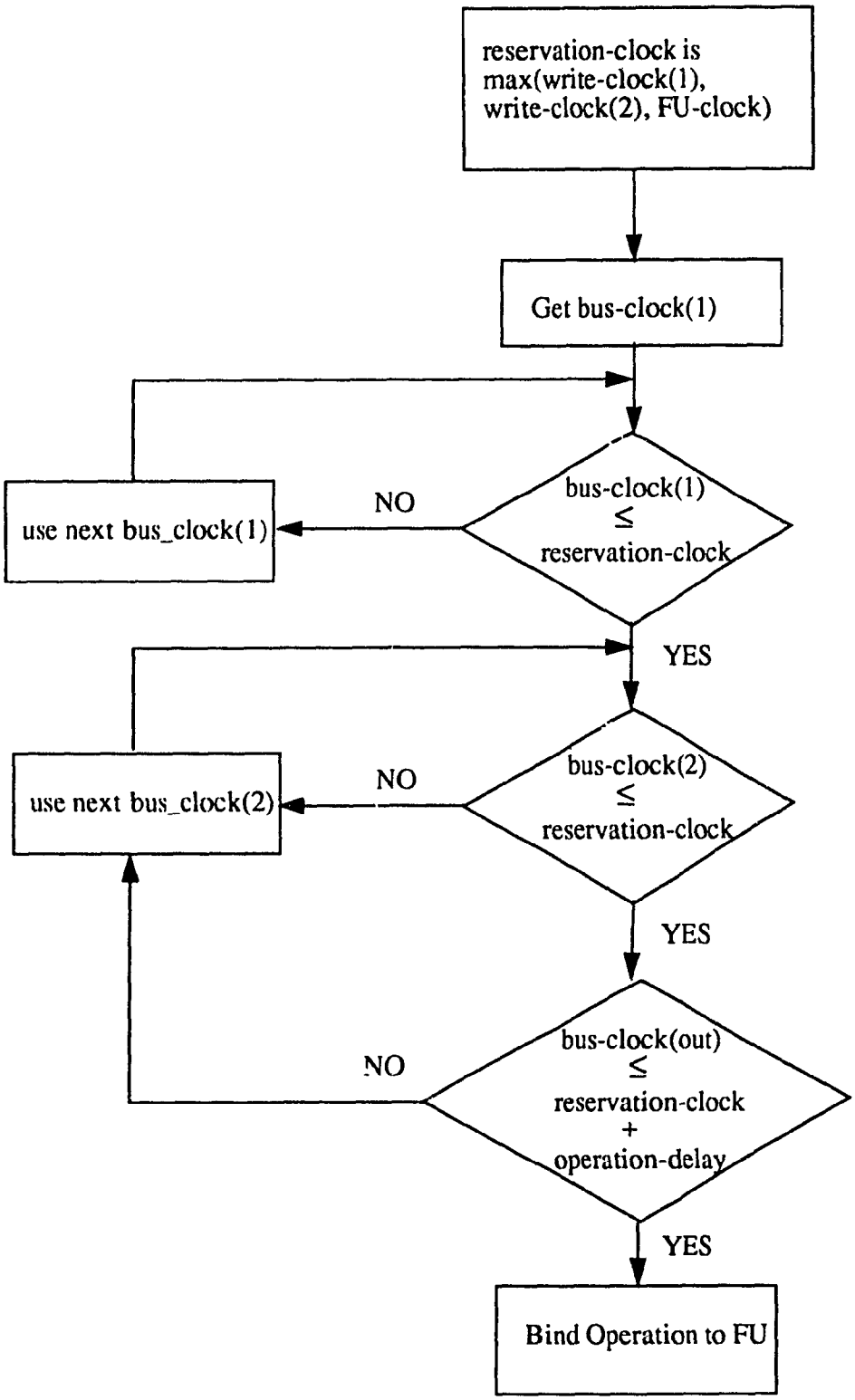


Figure 5.7: Reservation clock flowchart

write-clock(2) onwards. Inputs to the FU have to be guaranteed access through the busses on the read clock phase and the output of the FU needs to be guaranteed access to the busses on the write clock phase. Rather than use FIFO techniques at the output of a FU, the operation will only be scheduled when the input and output words can all access busses. The execution of the operation is thus conditionally delayed. Bus cycle times on the read phase are subject to the following: *bus-clock(1)* ≤ *reservation-clock* and *bus-clock(2)* ≤ *reservation-clock*. The bus cycle time reserved on the write phase is then subject to *bus-clock(out)* = *reservation-clock* + *Operation-delay*. All the busses have to be reserved before the operation can be bound to the FU. The transfers can be done on any of the busses unless bit size and additional constraints are taken into account. The bus reservations are later bound in the optimization phase. The procedure given above is diagrammed in the flowchart of Figure 5.7. For pipelined units, additional constraints are involved in determining the reservation and bus clocks as explained in Section 5.3.2.

The selection of FUs minimizes the idle times without scanning the FU list for possible gaps where an operation can fit. For an operation that can be initiated at time T, where the availability times in clock cycles of FU_1 to FU_N are A_1 to A_N, the chosen FU is the one that satisfies the following selection criteria:

$$\text{If } \neg\exists (i \in (1,N)) \mid [A_i < T] \text{ then select } FU_i \mid [(A_i - T) \text{ is minimum}] \quad (5.1)$$

$$\text{else } \forall (i \in 1,N) \mid [A_i < T] \text{ select } FU_i \mid [(T - A_i) \text{ is minimum}] \quad (5.2)$$

This is shown in Figure 5.9where the shaded boxes designate the use of the FU and the white box designates the execution period of the operation to be scheduled.

The MA operations are *compound* operations (made up of smaller operations) and are dealt with in a manner slightly different than regular operations. Each MA operation $ma \in O$ consists of n operations $o_1[ma], o_2[ma], \ldots, o_n[ma]$. First, $o_1[ma]$ is scheduled as explained above except that the selection of FUs requires information on the availability of local registers (accumulators) to store the intermediate results. Once a MAC has been

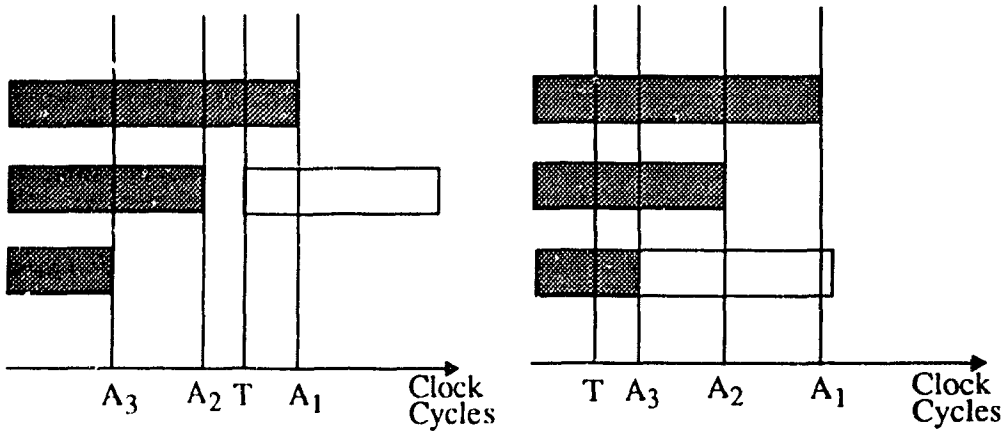


Figure 5.8: Selection of FUs

reserved for $o_1[ma]$, using a specific local register, all the corresponding (n-1) MA operations are bound to that MAC where binding of pipelined operations is performed. The way the MAs are scheduled depends on their first sub-operation. All FUs and local registers are scanned to find the MA *tag* (identifier of MA operations) and the remaining operations are scheduled accordingly. MA sub-operations $o_1[ma]$ through $o_{n-1}[ma]$ do not require the reservation of a output (only $o_n[ma]$ does), since the results are stored internally and accumulated, which relieves part of the constraints on scheduling and results in a more compact schedule.

5.3.1.3. Bus bindings and register minimizations

The third phase (Optimizer), performs several optimizations on the architecture. It involves a modified bipartite edge coloring algorithm to assign the registers used to a register file for each bus, minimizing duplication of storage. Minimizations are also performed on each register file by having temporary registers re-using the same hardware registers. The time sharing of registers is performed by an algorithm that colors a circular arc graph of the registers lifetimes [52]. Initially each link in the SFG is assigned a register. Even though the bus scheduling approach allows for duplication, the optimizations consistently reduce the number of registers (for BP in recall) to within 10%

of the lower bound:

$$lb = M + W + \sum_{i=1}^{K} C_i - P \qquad \text{where} \qquad M = \max_{i=1}^{n} \left(\sum_{i=k}^{k+1} N_i \right) \tag{5.3}$$

$$\text{and} \qquad P = N_{FU} \times N_{LR}$$

where N_i is the number of the neurons in layer i, W is the total number of weights and C_i is the number of constants in layer i. P refers to the neurons being performed in parallel using the local registers in the FUs. N_{FU} and N_{LR} are the number of FUs (MACs) and the number of local registers respectively. (The assumption is that $N_i > N_{LR}, \forall i$).

The microcode generation is performed after all the synthesis and optimizations are completed. It is specific to the architecture used and highly dependent on hardware specifications such as the number of bits/word, the resulting number of multiplexers, the register count, the number and size of the register files, the number and types of FUs as well as the number of operations supported by each. One control word is generated for each clock phase.

5.3.2. Scheduling of pipelined functional units

Pipelined units such as the MACs explained in Section 4.5.1, add to the complexity of the synthesis procedures. A special hazard avoidance algorithm is used to ensure that no conflicts occur in the allocation of hardware resources.

The first part of this algorithm uses the reservation tables obtained from the hardware descriptions of the FUs augmented by information on resource limitations to generate initiation lists for all pairs of operations. A typical FU description of the MAC unit presented in Section 4.5.1 is shown in Figure 5.9. The filled boxes indicate the stage usage of an operation. The arrows indicate hardware resource limitations due to input and output ports available. The pairs of initiation lists are then calculated based on an allocation table method, where one operation is shifted in time with respect to an existing

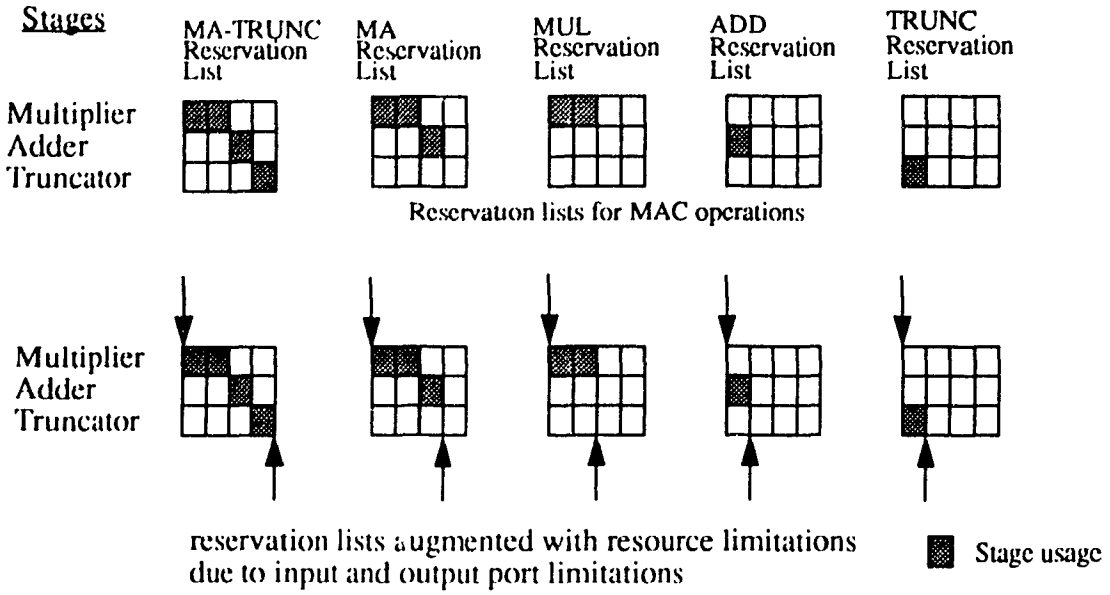


Figure 5.9: Example pipelined FU description

operation. The positions that do not result in any hazards or conflicts are inserted in corresponding initiation lists, examples of which are shown in Figure 5.10. The shaded regions indicate the initiation times that cannot be used due to some conflicts in the stage usage or hardware resources. This part is executed prior to the start of the synthesis procedures on all the available pipelined units.

The second part of the algorithm is invoked when an operation is to be scheduled on a pipelined FU. The purpose of this algorithm is to combine the initiation lists to allow for multiple operations to be executed on the same FU at overlapping times. For an operation to be scheduled at time T, the set of initiation vectors, for the operations executing on the FU relative to the new operation is:

$$V_i \quad \text{for} \quad i = \{T_0, \ldots, T\} \quad \text{where} \quad T_0 = T - S \times Ck \tag{5.4}$$

Ck being the number of clock cycles used by a stage and S being the number of stages of the pipelined FU. The initiation vector of the new operation will be:

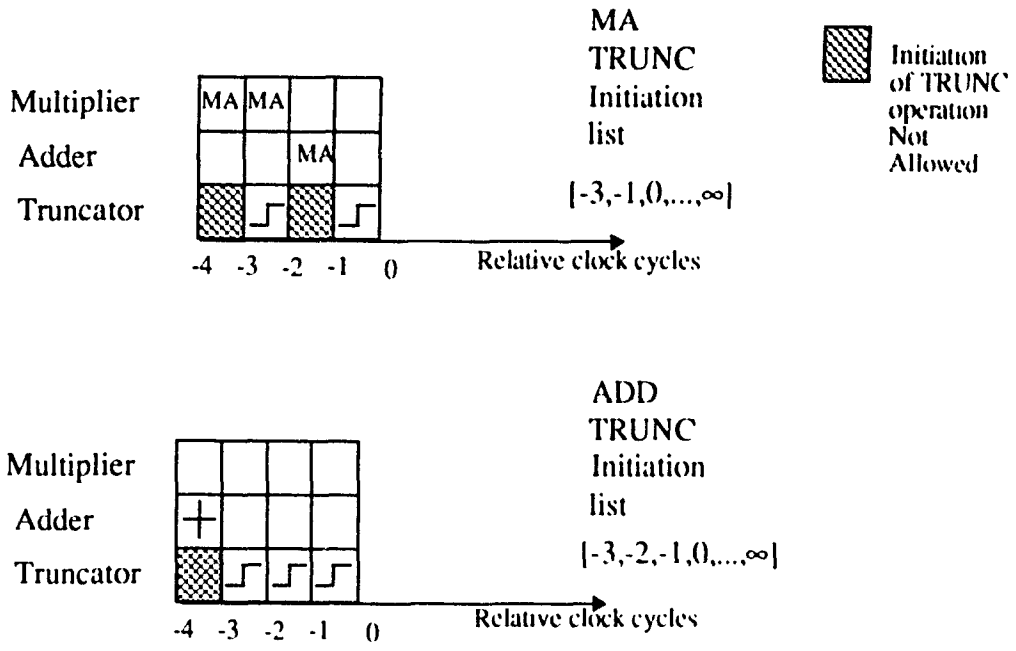


Figure 5.10: Example initiation lists of the MAC of Figure 5.9

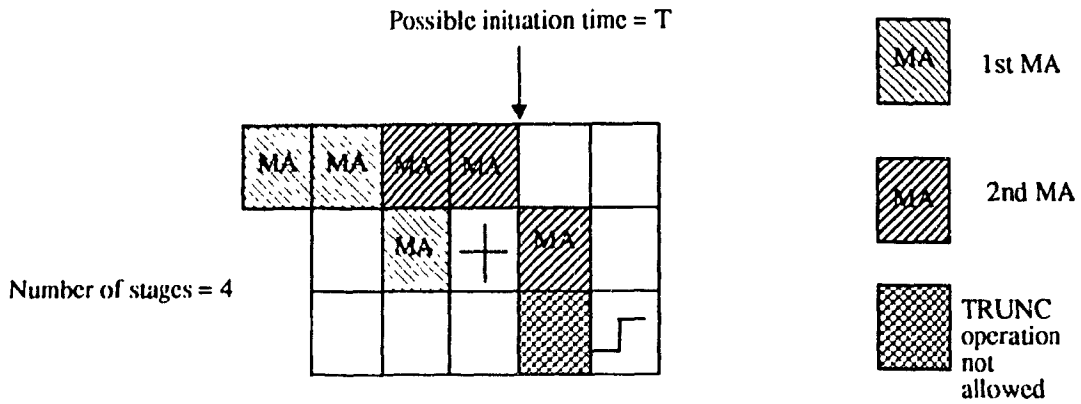
$$X = \bigcap_{i=T_0}^{T} (V_i - \delta_i) \tag{5.5}$$

where δ is the set of delays between the initiation of existing operations and the new one (adjusted by $S \times Ck$ to produce a value relative to T).

Thus, if a TRUNC operation whose inputs are available at clock cycle T is to be scheduled on the MAC of Figure 5.9, where two MAs and one addition are executing as shown, the resulting initiation vector becomes [1,2,...,∞], or [T+1,T+2,...,∞] as shown in Figure 5.10.

5.3.3. Pre-scheduling optimization using neuron splitting

Compound operations such as MAs when bound to a FU constitute long operations that may include idle times between their distinct parts. Optimum scheduling of these operations cannot be obtained when their number is not a multiple of the available MACs since this may result in uneven use of the MACs and additional cycle time as shown in



Initiation list for TRUNC with 2 MAs and 1 ADD

$[-3,-1,0,\ldots,\infty]$ $\quad \delta_{1=4-4=0} \Rightarrow \quad [-3,-1,0,\ldots\infty]$

$[-3,-1,0,\ldots,\infty]$ $\quad \delta_{2=2-4=-2} \Rightarrow \quad [-1,1,2,\ldots\infty]$

$[-3,-2,-1,0,\ldots,\infty]$ $\quad \delta_{3=1-4=-3} \Rightarrow \quad [0,1,2,3\ldots,\infty]$

$\cap \Rightarrow \quad X=[1,\ldots,\infty]$ with respect to T

Figure 5.11: Initiation list for TRUNC with MA and ADD

Figure 5.12. This *asymmetry* can result in loss of cycles and eventually a bad schedule. Splitting the neurons that cause asymmetry into *pseudo-neurons* which execution can be evenly partitioned on all available MACs can result in substantial savings. This approach requires additional temporary storage registers and could result in delays due to the additional bus traffic needed to transmit the intermediate results on the busses. However, for most neural networks, the MAs are large operations (since a neuron's inputs can consist of all the previous layer's state and weight values), and the results of splitting neurons to resolve asymmetry saves clock cycles. It is clear, as will be shown in Section 6.3.1, that some networks benefit from this optimization more than others due to their topologies. It is also clear that the optimization is most effective when applied to the last layer where overlapping of the last MAs in the layer with the first ones in the following layer cannot occur. The speedups obtained are not high for some cases due to the

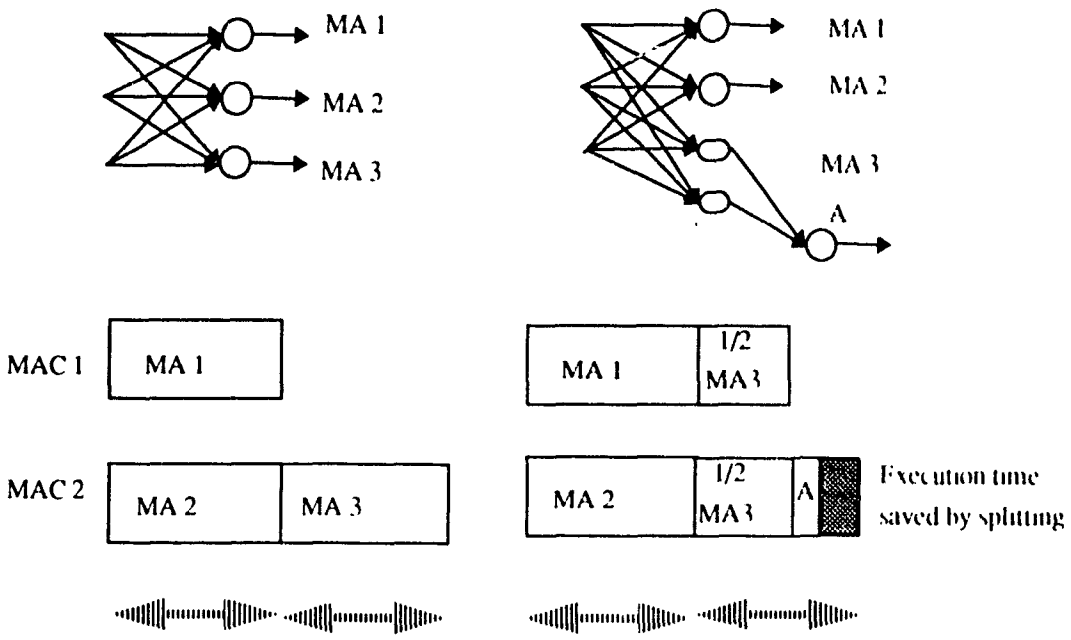


Figure 5.12: Splitting neuron to resolve asymmetry.

additional constraints on bus transfers and FU usage that splitting will induce, which is the argument against splitting the neurons completely and using multiplications and additions to perform the MA operations.

5.3.4. Implementing loop folding by augmenting the network's SFG

In order to reduce the controller size, the network is divided into similar partitions and looped execution (FOR... DO loops) of the network is implemented. ANN algorithms are highly regular and thus suitable for folding into similar partitions. Two types of folding are discussed which can be combined to implement specific folding patterns for specific networks: *vertical* and *horizontal* folding. Vertical or layered folding is achieved by executing one layer at a time. This is accomplished by the addition of feedback edges to the SFG as shown in Figure 5.15. This adds to the complexity of the controller in terms of word length (microcode controller) but results in a greatly reduced schedule and therefore reduces the total controller size. The addition of the feedback edges does not modify the datapath scheduling. The second type of partitioning uses horizontal or neuronal folding

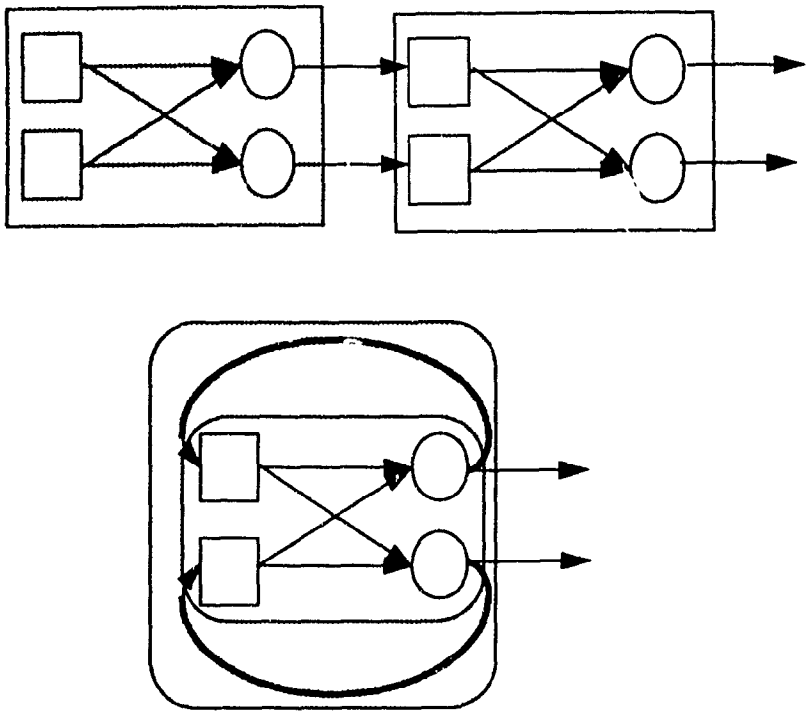


Figure 5.13: Vertical folding of a network

which divides each layer into similar partitions as shown in Figure 5.15. Dividing a layer into two different partitions will necessitate the addition of two partial sums or the use of a storage register to store the values of the same neuron and initializing the local storage registers for each subsequent MA operation with the appropriate sum. It should be noted that in such a folding scheme the nonlinear operations cannot be executed until all the partitions are done and the final sum of MAs is computed. Classes of MAs and nonlinear function implementations are then separated in the synthesis.

5.4. Multi-processor Synthesis

The methodology used for multi-processor synthesis is general and can be applied to any signal flow graph but is most suitable for regular algorithms such as ANNs. The following discussion, however, concentrates on ANN synthesis issues, specifically feedforward-style networks. Examples and explanations presented are based on BP

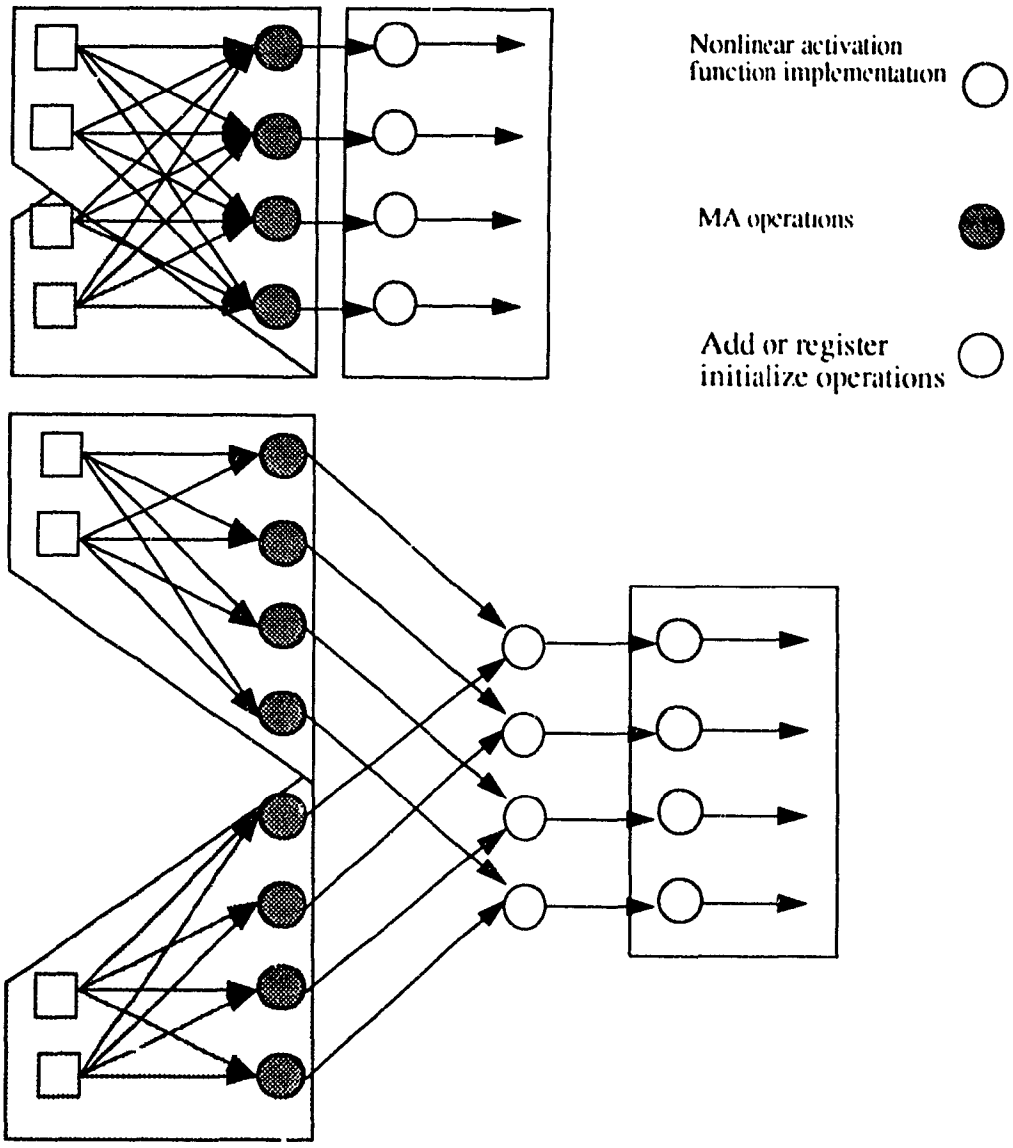


Figure 5.14: Horizontal folding of a network

networks in the recall mode.

ANNs are large systems that are inherently parallel. This parallelism should be exploited beyond the area boundaries of one processor implementations. An extension of loop folding as described above is to implement each partition on a separate processor. All the processors would execute the same schedule on different partitions resulting in a SIMD structure. In order to achieve this, however, several issues such as partitioning, synchronization and inter-processor communication have to be addressed.

5.4.1. Partitioning the network on the available processors

The network is partitioned symmetrically so that the same schedule is used for different partitions on different processors. The types of partitioning used in loop folding (Section 5.3.4) can be used, however the SFG must be modified to allow for I/O operations to accommodate for the interconnection which is not the case in folding the network on the same processor.

A *symmetrical* network is one in which the number of neurons (and interconnections) can be divided evenly among the number of processing elements used. In the case of *asymmetrical* networks, either dummy operations are inserted or neuron splitting is implemented as detailed in Section 5.3.3 or a combination of both methods is used. Dummy operations are analogous to NOOP in microprocessors and their sole purpose is to delay the processing in a PE to allow for the synchronization. The easiest way to implement these operations is to implement redundant calculations that may be executed on another PE. This will guarantee synchronization (the operations execution times are identical) without affecting the behavior of the processor.

The partitioning is done by the user and according to the number of processors available. The synthesis tool takes one partition, augments its corresponding SFG and synthesizes it using the same methods used for architectural synthesis.

Once a network has been partitioned as shown in Figure 5.15, the SFG corresponding to each partition is augmented and modified as explained in Section 5.4.2. The resulting graph is then synthesized. The partitioning shown, includes input and output operations at the interfaces of the two partitions. In cases where more than two partitions exist the neuron activation values input to a partition will have to be output to the following partition as will be explained in details in Section 5.4.2.

The multiprocessing model used as an example to execute the network of Figure 5.15 is a circular systolic array as shown in Figure 5.16. Any type of systolic configuration can be used with each PE implementing one of the partitions. The synthesis as outlined

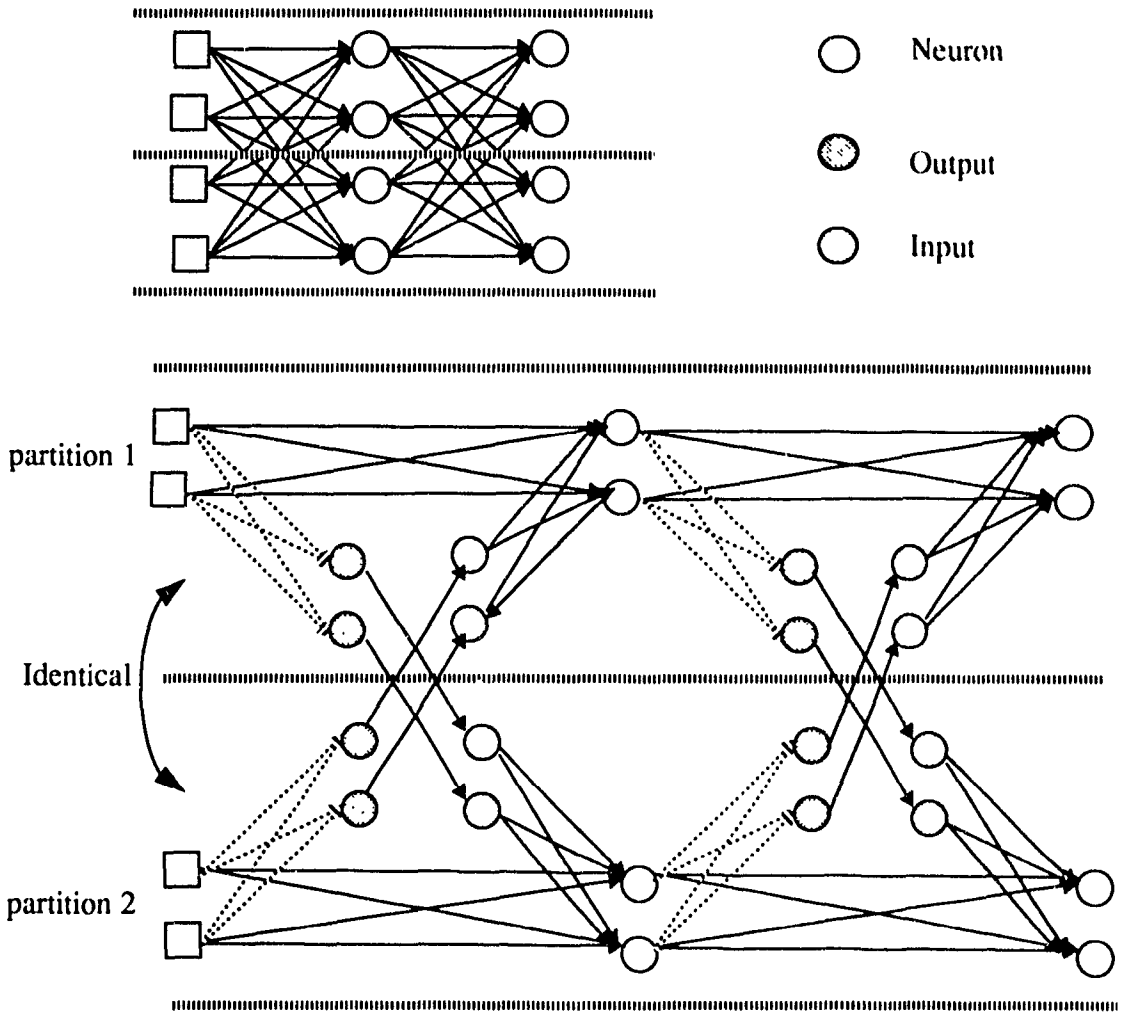


Figure 5.15: Example network partitioning for multi-processing

above will ensure correct operation. The interconnections between PEs can consist of several links which are set by the number of input and output ports used as part of the hardware configuration for each partition.

5.4.2. Modifying the SFG for multi-processing

Multi-processor synthesis is an extension of the architectural synthesis methodologies presented in Section 5.3. In order to simplify the synthesis task and exploit the parallelism and regularity of ANNs, a systolic model of the multi-processor architecture is proposed as explained in Section 4.7. Extending architectural synthesis to

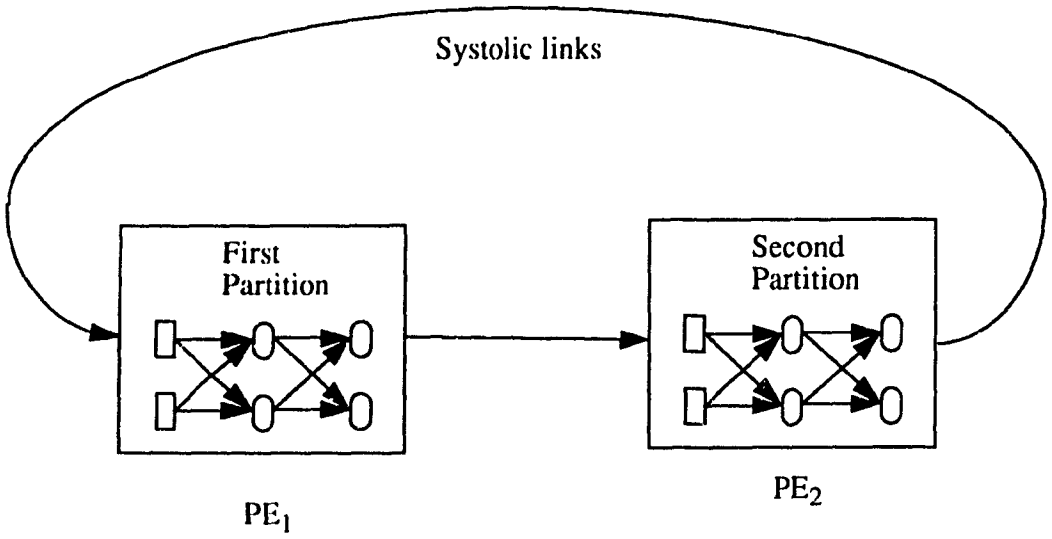


Figure 5.16: Circular systolic configuration example

accommodate processor inter-communication necessitates the following modifications to the SFG:

Let α_L: be the number of *internal neurons* for layer L

β_L: the number of *external neurons* for layer L

N_L: the number of neurons used to generate the SFG for layer L

Net: the set of layers in the network

then:

$$\forall (L \in Net), \begin{cases} N_{L-1} = \alpha_{L-1} + \beta_{L-1} \\ \text{and} \\ N_L = \alpha_L \end{cases} \tag{5.6}$$

Internal neurons are those which operations are implemented on a single partition and therefore would be executed on one PE and are thus internal to that PE. External neurons on the other hand are those neurons (for each layer) that are not internal to a PE and which activation values need to be input from the neighboring PEs. Equation 5.6 is a formalization of the number of neurons needed in each layer. It basically states that the

number of neurons used in the SFG, for each layer, is actually the number of internal neurons whereas the number of neurons in the previous layer is the total number of neurons in that layer for the whole network. The distinction need to be made in order to generate the correct neuronal and input/output operations. In fact, the number of inputs to PE_{Li}, where L denotes the current layer and i is an index for the PEs in that layer, is β_{L-1}. The number of outputs is given by:

$$\text{number of outputs} = \alpha_L + \beta_{L-1} - \gamma_{L-1} \tag{5.7}$$

where γ_{L-1} denotes the neurons internal to the $PE_{(L+1)i}$.

The number of PEs used to configure a layer L is $PE_L = \left\lceil \frac{\alpha_L + \beta_L}{\alpha_L} \right\rceil$. The number $\left\lceil \frac{\alpha_L + \beta_L}{\alpha_L} \right\rceil$ can also be modified by splitting neurons to reduce cycle time (as explained for the single processor case in section 3.3) or inserting dummy operations to obtain a more symmetrical network which is needed in order for the controllers of all the partitions to be identical.

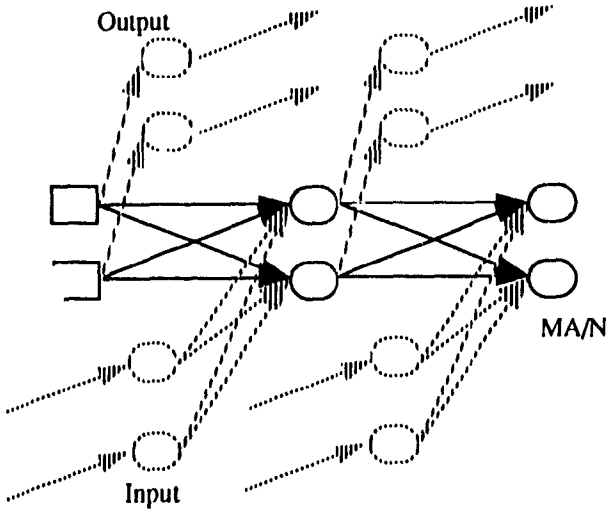


Figure 5.17: SFG with I/O considerations for multiprocessing

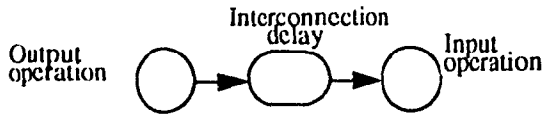


Figure 5.18: O/I delay operation for multiprocessing

5.4.3. Scheduling modifications to accommodate multi-processing

Once the network has been partitioned, the modification of the SFG is done by adding the appropriate input and output operations. For the network of Figure 5.15, additional input and output operations are included. The modifications are as shown in Figure 5.17 for each partition. Each output operation in a partition corresponds to an input operation in a second partition. The input operations need to be performed before their values are needed in the calculation of the regular operations such as the MAs. These input and output operations have to be synchronized in order for the multi-chip system to function correctly. One way of achieving this is by adding appropriate delays on the systolic links. This, however, could result in extreme delays and a poor execution time. The other method is to include the synchronization in the synthesis itself. In order to achieve this, a new operation is introduced. This operation as shown in Figure 5.18. The input to this operation is the output value and will be transmitted on one of the busses and outputted through the O/I (Output-Input) unit to another partition on another PE on the Read clock phase. The output is an input value and is transmitted on one of the busses through the O/I unit to the partition implemented on the PE itself. The delay is the interconnection delay. Using this operation, synchronization is guaranteed and the SFG of Figure 5.17 is transformed into the one shown in Figure 5.19. The number of systolic interconnections is equal to the number of O/I units. Different types of O/I units can be modelled and used in the synthesis as FU. This allows any systolic interconnection to be modelled appropriately. The transformed SFG is synthesized using the methods outlined

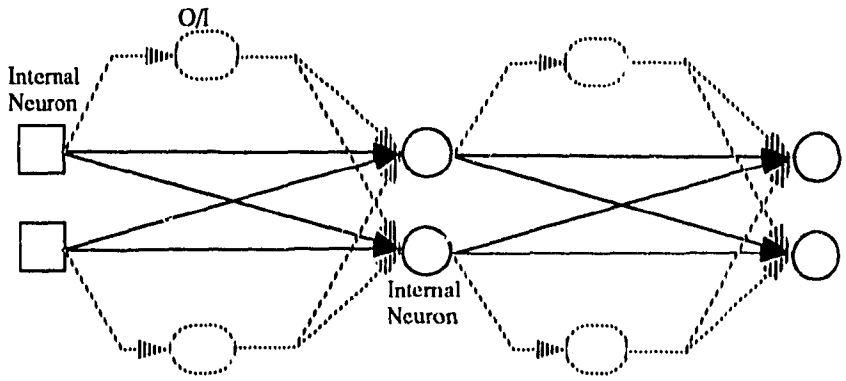


Figure 5.19: SFG with added O/I delay operation for multiprocessing

before. The modifications are automated and based on the number of PEs allocated by the user. Once a synthesized network exists the same schedule can be used for all partitions.

5.5. Conclusion

This chapter introduced the methodologies used in the architecture and multi-processing synthesis of ANN algorithms and implemented in a Prolog synthesis tool. The issues involved in the extraction of the network's description, the pre-scheduling optimizations that included neuron splitting and operation type allocations were explained. The scheduling and binding problems were discussed including compound operations such as the multiply-accumulate operation. Issues involved in the scheduling of pipelined units were introduced. For multiprocessor synthesis, a new output/input operation is used to synchronize communications and modifications to the partitions' SFGs are used to incorporate multiprocessing. All the issues proposed in chapter were implemented in a Prolog synthesis tool and results from synthesized architectures showing the different optimizations and trade-offs are presented in chapter 6.

Chapter 6:

Architectural Trade-offs and Synthesis Results

6.1. Introduction

This chapter presents the synthesis results using the architecture described in chapter 4 and the algorithm detailed in chapter 5. The developed synthesis tool is used on several neural networks detailing the architectural trade-offs that were discussed in chapter 4. The design space of the four networks presented in chapter 2 is searched and the corresponding optimum architectures are presented. The architectural optimizations discussed in chapter 4 are then investigated. Comparisons with existing neural network hardware is made proving the quality and importance of the design search provided by the tool and the proposed architecture synthesized as described in this thesis.

6.2. Performance Evaluation

6.2.1. Design examples

Several networks have been synthesized using the synthesis methodologies as described in Chapter 5. Three fully connected BP networks and one counterpropagation network presented in Chapter 2 were used as examples of searching the design space, others were used to illustrate specific architectural trade-offs. The first network used in the design space explorations is a robotic one Degree of Freedom Flexible Joint Manipulator with 6, 21, 12 and 1 neurons respectively. The second is an ECG signal processing network with 40,10, and 1 neurons respectively and the third is a pattern recognition (PR) network with 16,5,9 and 4 neurons. The counterpropagation network used is a small version of the NASA space station robot arm guiding network using 64, 8 and 1 neurons respectively. All these networks are real-time adaptive ANNs with specific requirements. They have approximately the same size and are typical of the types of networks that are being used in industrial applications today.

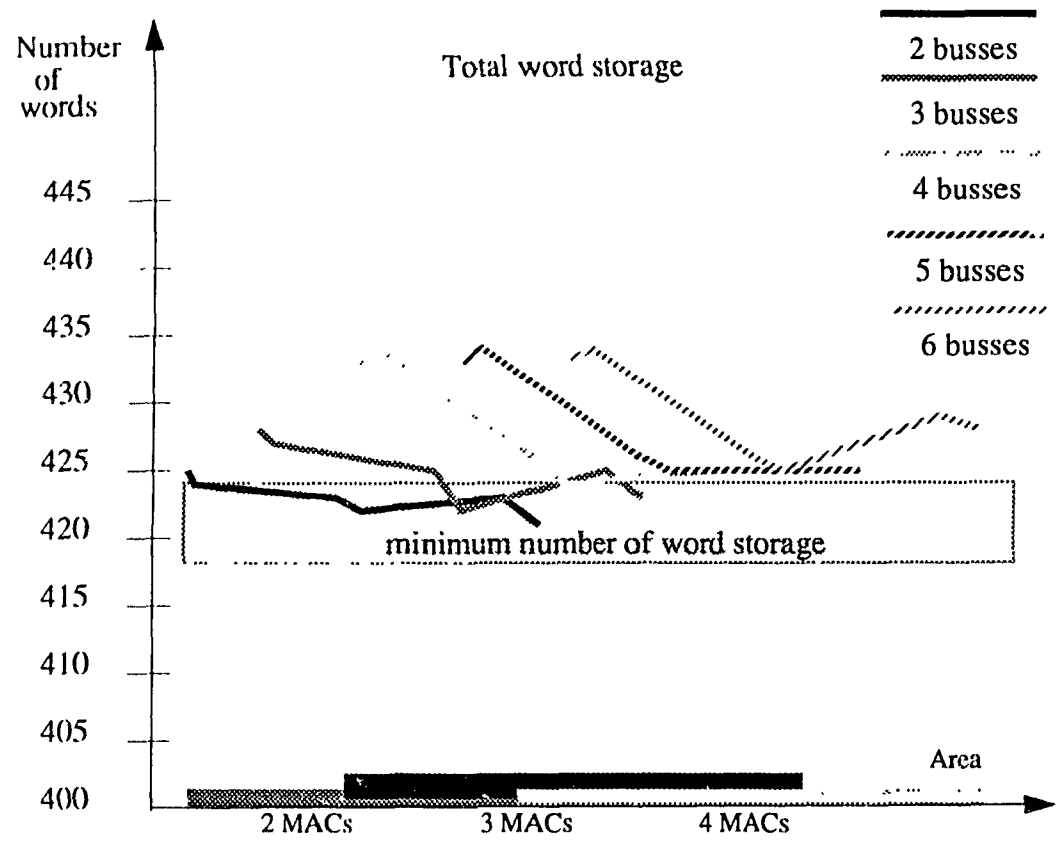


Figure 6.1: Total word storage for the number of busses

The architectural performance evaluations were done in two different ways. The first is an architectural top-bottom approach that assumes typical values for the execution times and speeds (based of course, on technological ranges available in today's technologies). The second is a VLSI bottom-up approach that uses technologically specific measurements of area and speed to evaluate architectural trade-offs. The first two BP networks (Robotic and ECG) as well as the CP network were synthesized using the first approach while the PR was synthesized using the second approach.

For the first approach, the evaluations were based on the speedup with respect to a reference architecture as well as AT^2 where A corresponds to the increase in area with respect to the reference architecture which is a configuration with a minimum allocation of one MAC (one accumulator), 1 bus and 1 register file. The number of MACs, busses and

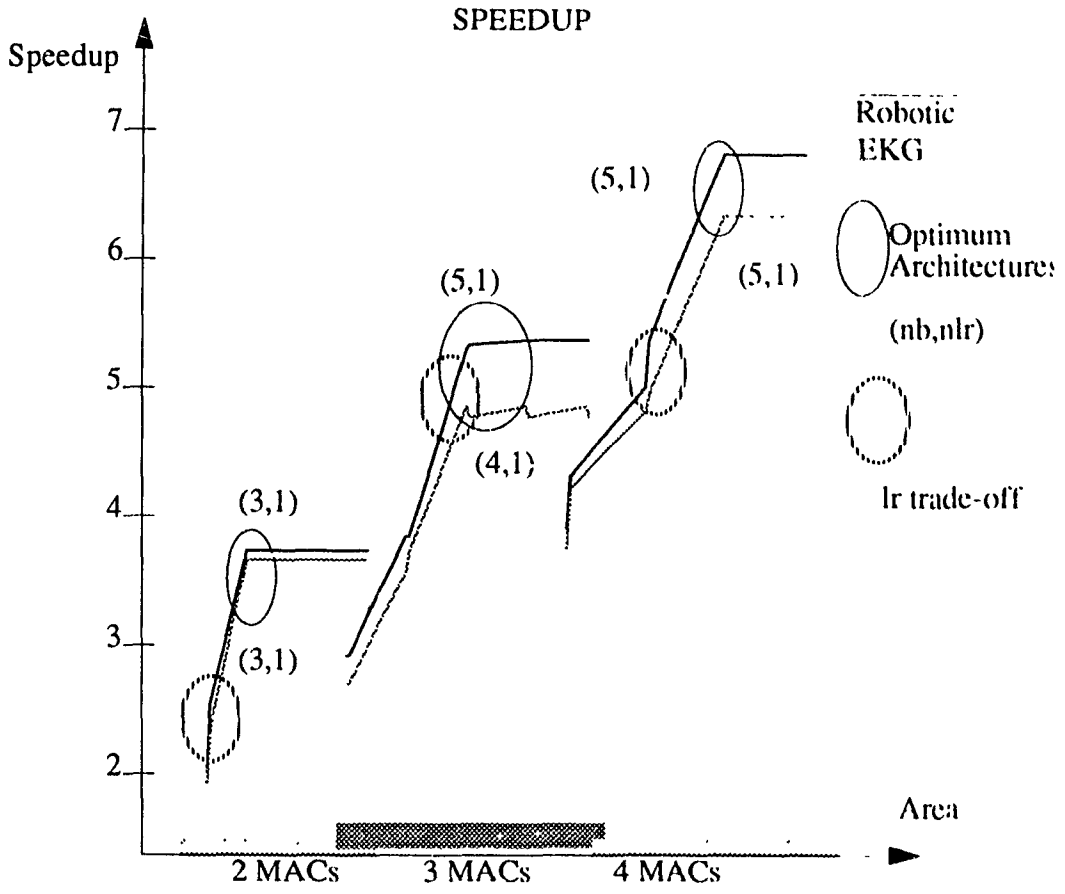


Figure 6.2: Recall Performance of BP networks (Speedup) for increase in MACs/area

local registers is then increased to investigate the design space. The speedup relative to the increase in area is $S_a = \frac{T_0}{T_a}$ where T_0 and T_a are the execution times in clock cycles of the reference architecture and the one being evaluated respectively. The percentage increase in area relative to the reference architecture is based on the area assumptions as follows:

$$A = 30N_{MAC} + N_R + 6N_{MAC}N_B \tag{6.8}$$

where N_{MAC}, N_R and N_B are the numbers of MACs, registers and busses used. These assumptions are used to indicate cost trends and not absolute cost values as these depend on layout details and technology. This area estimate is for the data path area including the

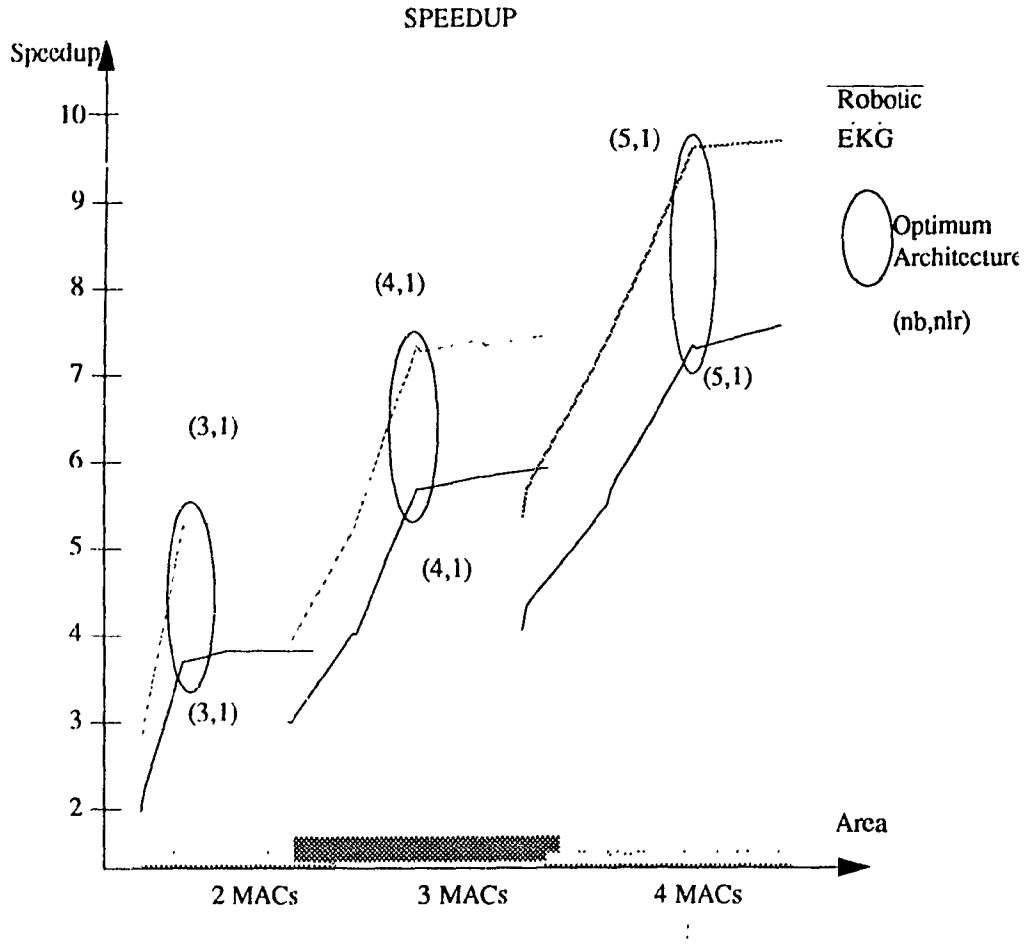


Figure 6.3: Learning performance of BP networks (Speedup) for increase in MACs/ area

MACs (with their local registers) and the busses area The controller and the register file areas are not included. This is due to the fact that their inclusion would smooth out the trade-off curves, that will be explained shortly, without adding any insights since their areas are relatively similar as can be seen in Figure 6.1 for the Robotic network. It should also be noted that the total number of word storage is between 0 and 0.028% over the *minimum* calculated as explained in chapter 5. Since the register files are implemented as RAMs with 2^N words. the small differences do not result in actual differences in the hardware areas at all.

In the examples presented here, the FUs considered are the general MACs specified in chapter 4 with operation delays taken to have typical values as follows:

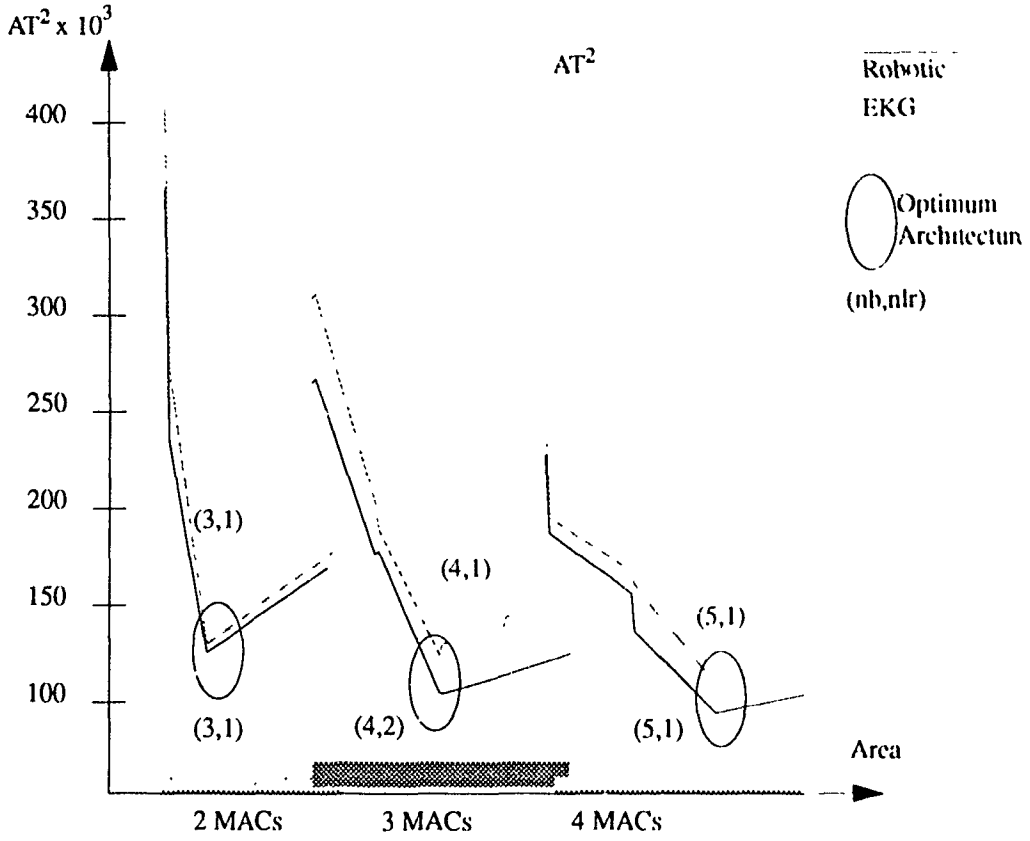


Figure 6.4: Recall Performance of BP networks (AT2) for increase in MACs/area

MA	3 cycles
Multiplication	2 cycles
Addition/Subtraction	1cycle
Threshold	1cycle

The transfer functions for the examples shown are implemented as simple truncations with respect to a constant value.

For the second approach, the areas and delays are calculated as explained in chapter 7 and are actual measurements in mm^2 and μs. The area calculated includes the data path area as well as that of the controller and register files. The delay is based on the processor clock cycle time as explained in chapter 7. Measures such as AT^2 can then be obtained to investigate the architectural design trade-offs.

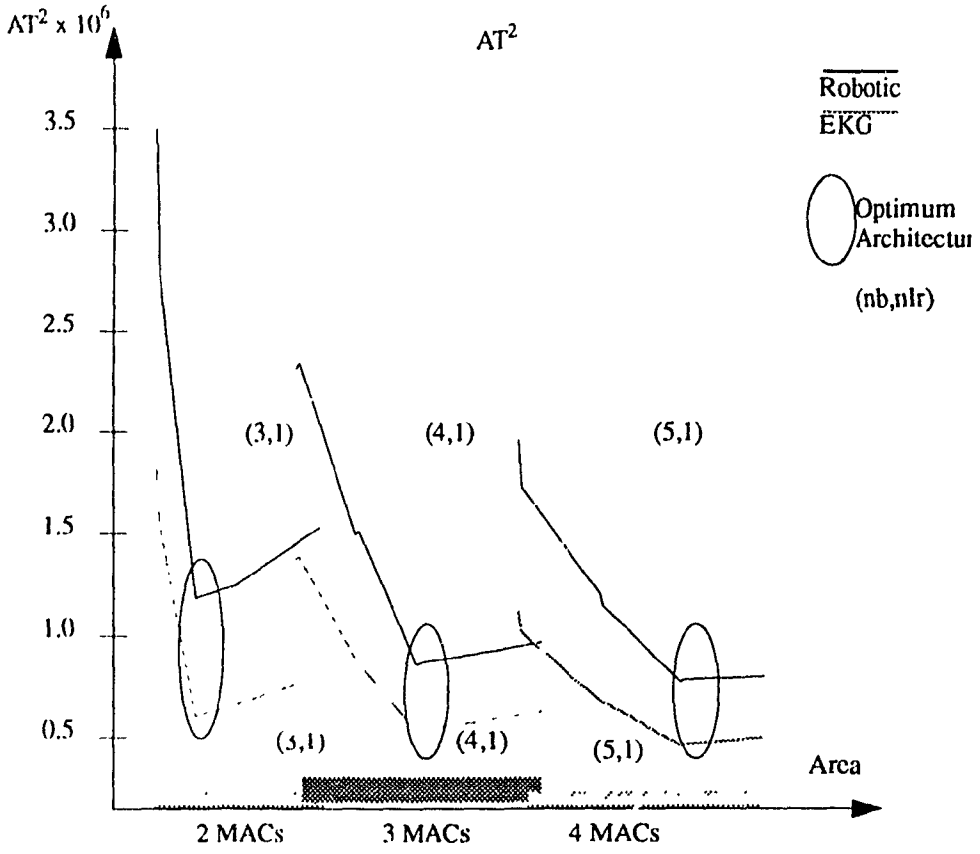


Figure 6.5: Learning Performance of BP networks (AT^2) for increase in MACs/area

6.2.2. Design space search

For the Top-bottom architectural design the Robotic and ECG networks are used. Figure 6.2 and 6.3 show the synthesis results for the networks in the recall and training modes respectively for the speedup measure. Figure 6.4 and 5 present the same for the AT^2 measure. The graphs show distinctly the difference between using 2, 3 or 4 MACs. The optimum architectures with the number of busses and local registers (*NB, NLR*) used are shown. Figure 6.2 further outlines the trade-off relating to the number of *lr*s. Figure 6.6 and 6.7 present the recall and learning performance of the CP network. All these figures have similarities in their general shapes but outline specific differences between the networks as will be explained later.

For the bottom-up approach, Figure 6.8 presents the AT^2 behavior of the PR

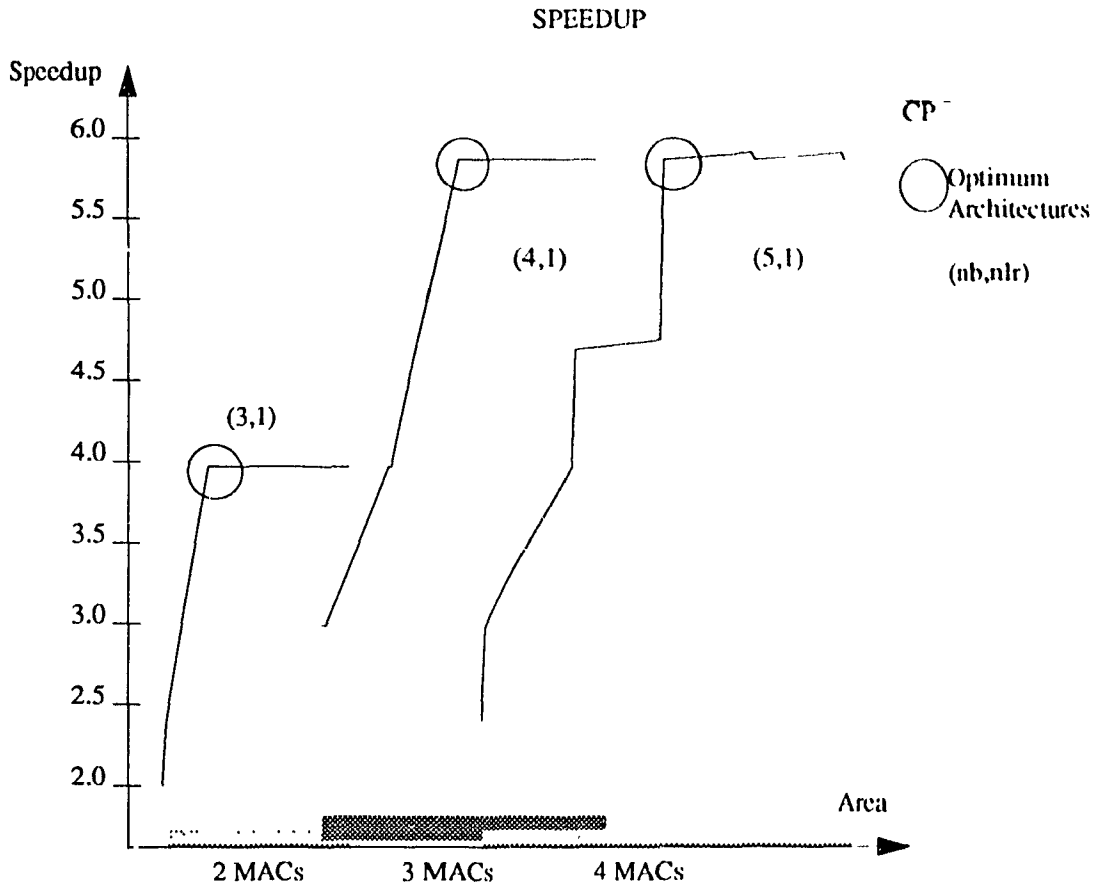


Figure 6.6: Recall Performance of CP network (Speedup) for increase in MACs/area

network from actual VLSI area and delay measurements. The general shape is again similar in nature to the curves obtained using the first method proving the relative accuracy of such broad assumptions as were made to evaluate the areas and the delays.

6.2.3. FU saturation for the number of busses

The *saturations* of the speedup curves occur when the addition of busses does not improve the speed and they indicate the highest speedup that could be achieved with the number of FUs provided. These saturations occur well below the limit of two b ses per FU. This limit is due to the fact that all the FUs used have two inputs and one output requiring a maximum of $N_B = 2 \times N_{FU}$ busses on the read clock phase to provide complete parallel access to the FUs. It is clear that the optimum architectures for different networks

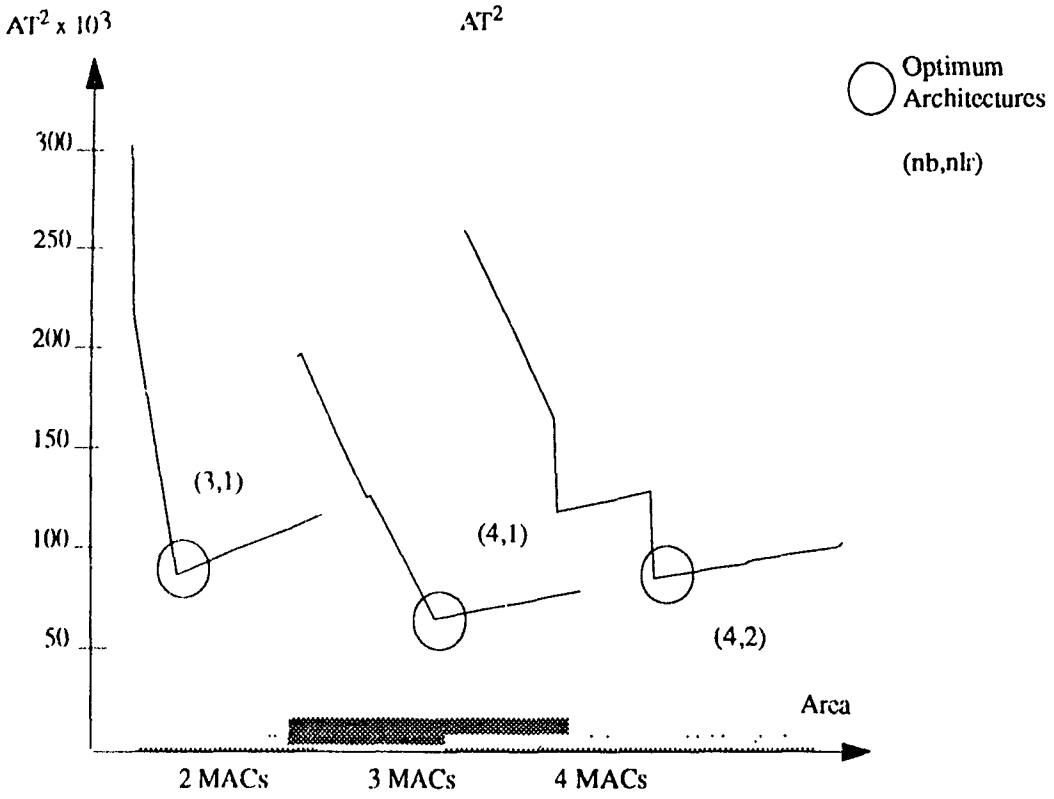


Figure 6.7: Recall Performance of CP network (AT^2) for increase in MACs/area

are not necessarily identical even though these networks were selected in the same size range.

6.2.4. Multiple word storage

The *lr* register trade-offs play an important role where the number of the busses is constricted and when operations are interleaved as explained in Section 4.6.2. Figure 6.2 shows several cases where one additional register results in 10 to 25% speedup over the previous architecture. Adding more *lr*s, however could result in worse performance due to the added scheduling constraints.

Table 6.2 shows several cases where adding one local register to each MAC can result in substantial savings in terms of speed. In the table, Mode designates the Recall (R) or the Learning (L) modes, NMAC, NB and NC are the number of MACs, busses and total

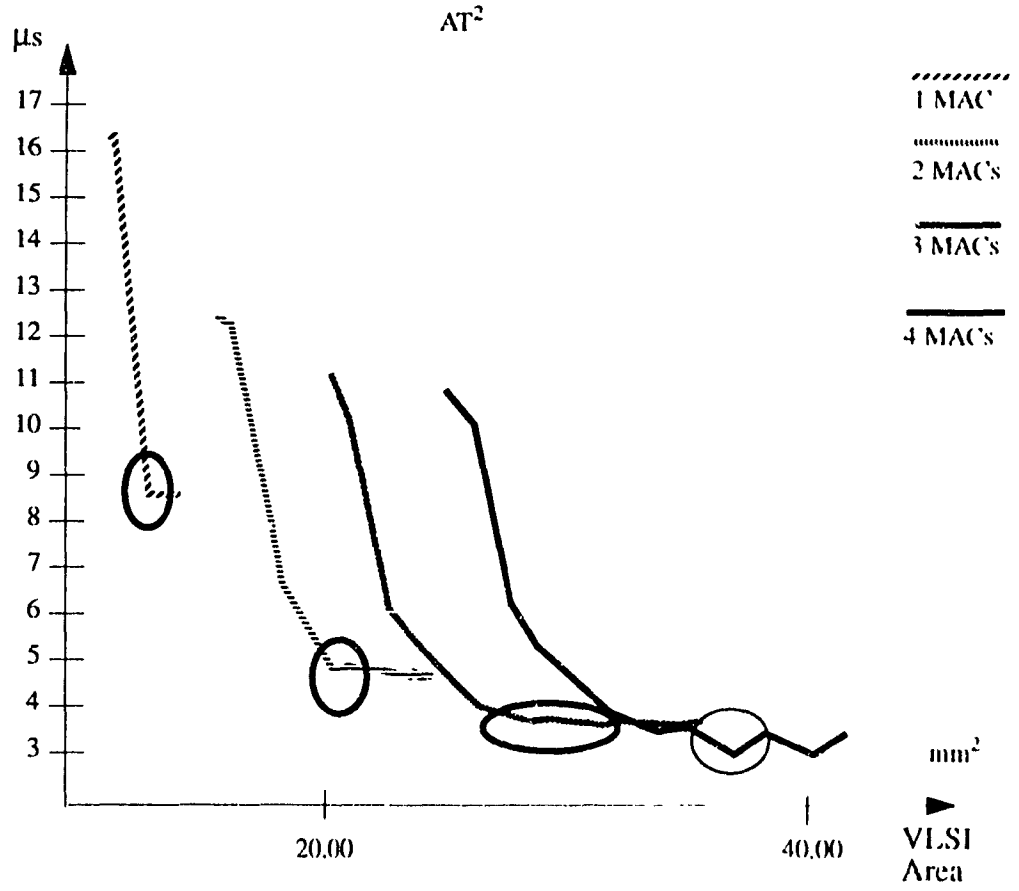


Figure 6.8: AT^2 performance from technological area-delay measurements

clock cycles respectively. CPC is the number of cycles per connection. The percentage speedup designates the speedup achieved as a result of the optimization (i.e. adding one local register per MAC).

6.2.5. Topology differences' effect on speed

The networks presented (with approximately the same number of neurons) achieve dramatically different performances in the learning mode due to the difference in their topologies. This is due to the fact that learning algorithms behave differently than recall algorithms. The propagation itself in BP networks suggests the difference in the number of operations since the MAs involved are being calculated in an opposite direction than in the recall.Table 6.2 compares the ECG and robotic networks in recall and learning. NLR

Network	Mode	NMAC	NB	NC		CPC		Percentage Speedup
				1 lr	2 lrs	1 lr	2 lrs	
Robotic	R	2	2	398	317	1.02	0.81	20.6%
Robotic	R	4	4	208	188	0.53	0.48	9.4%
Robotic	L	2	2	1230	1091	3.15	2.80	11.1%
Robotic	L	4	4	450	437	1.15	1.12	2.6%
ECG	R	3	3	227	218	0.55	0.53	3.6%
ECG	L	4	3	462	436	1.13	1.06	6.2%
PR	R	3	3	100	90	0.62	0.56	9.7%
CP	R	4	4	163	132	0.31	0.25	19.3%

Table 6.1: Effect of adding one local register to each MAC

designates the number of local registers used. * denotes the reference architecture for the speedup calculation (in this case the slower network).It can be seen that the networks achieve almost the same speeds in recall (Rows A and C), with the robotic network being slightly faster, but a large difference exists in learning (Rows B and D), and the ECG network becomes faster.

	Network	Mode	NMAC	NB	NI R	NC	CPC	Percentage Speedup
A	Robotic	R	4	5	1	119	0.31	0%
	ECG	R	4	5	1	128	0.31	*
B	Robotic	L	4	5	1	339	0.87	*
	ECG	L	4	5	1	259	0.63	24%
C	Robotic	R	3	3	2	211	0.54	*
	ECG	R	3	3	2	218	0.53	1.8%
D	Robotic	L	3	3	2	615	1.58	*
	ECG	L	3	3	2	463	1.13	28.5%

Table 6.2: Topology effect on speed for recall and learning

6.2.6. FU utilization

Table 6.3 contains the utilization percentages for the MAC stages for the Robotic and ECG network. The utilization of the threshold is low since it is performed once for

Network	NMAC	NB	NLR	Recall Utilization (%)		Learning Utilization (%)		
				Multiplier/ Adder	Threshold	Multiplier	Adder	Threshold
Robotic	2	4	1	86.29	7.52	61.24	83.64	2.64
	3	4	1	82.28	7.17	57.62	78.70	2.48
	4	6	1	79.27	6.9	57.42	78.42	2.47
ECG	2	4	1	92.76	2.49	49.26	89.38	1.16
	3	4	1	81.83	2.20	46.58	84.54	1.10
	4	6	1	80.08	2.15	45.85	83.20	1.09

Table 6.3: Average percentage utilization of the MAC stages

every neuron. The multiplier and the adder have the same utilizations in recall since they are only being used in MAs. In learning, the number of additions and subtractions is larger than multiplications (even though multiplication classes outnumber addition/subtraction classes) resulting in better utilization of the adder. This type of architectural search can provide several important guidelines for the design. It is clear that more care need to be taken when implementing the nonlinear function for the Robotic than the ECG network since the utilization is 4 times as much. Another important point concerns the possible use of additional adders in the learning phase since their utilization is much higher than other units' utilizations.

6.2.7. Multi-processing results

The ECG network is synthesized using 2 and 5 processors in a linear systolic configuration. For 2 processors, the partitions used, had 20, 5 and 1 neurons each. For 5 processors, the partitions had 8, 2 and 1 neurons each. Neuron splitting is not implemented. The speedup graph for the recall performance in comparison to the one processor implementations are shown in Figure 6.9. For 2 processors a maximum speedup of 1.91 is achieved. For a 5 processor configuration, a speedup of 3.68 is obtained. The

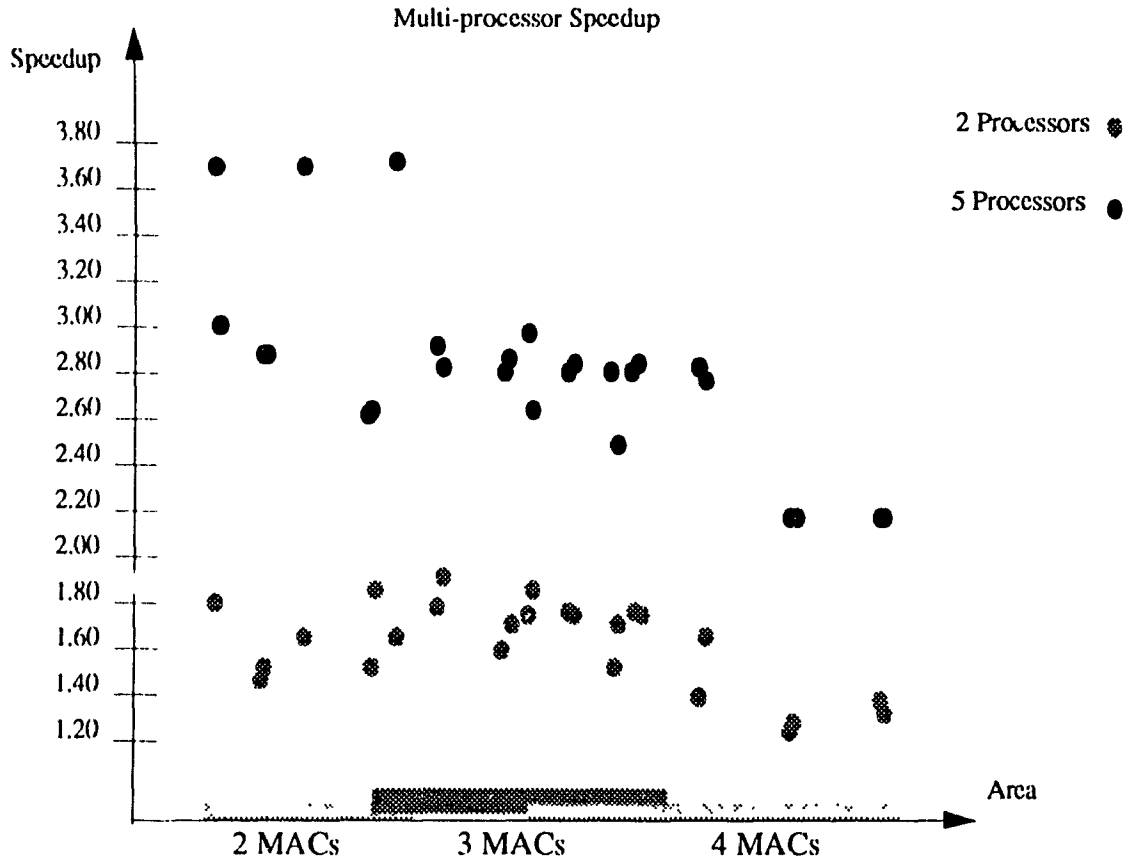


Figure 6.9: Speedup for EKG for 2 and 5 processors with respect to one

reason the speedup for the 5 processor configuration is not as high as expected lies in the fact that the partition (8,2,1) is too small so that additional hardware within a processor does not yield better results. Further splitting neurons rather than using dummy operations can enhance the performance. Table 6.4 further shows an important result of investigating multi-processor configurations. A trade-off exists between the number of hardware units used per processor and the number of processors used. It may be of use, therefore, to use 2 processors with 2 MACs each (Row b) rather than one processor with 4 MACs (Row g), as can be seen by the small difference of the number of clock cycles for the execution time in Table 6.4. Using less hardware per processor further reduces the complexity of the system and may result in a slightly faster clock. The use of the synthesis tool for multi-

	Network	Number of					CPC	Speedup
		PEs	MAC	busses	I/O	cycles		
a	40,10,1	1	2	4	1	221	0.54	*
b	40,10,1	2	2	4	1	135	0.33	1.6
c	40,10,1	5	2	4	1	60	0.15	3.6
d	40,10,1	1	3	5	1	167	0.41	*
e	40,10,1	2	3	5	1	96	0.23	1.3
f	40,10,1	5	3	5	1	60	0.15	2.7
g	40,10,1	1	4	6	1	128	0.31	*
h	40,10,1	2	4	6	1	95	0.23	1.3
i	40,10,1	5	4	6	1	60	0.15	2.1

Table 6.4: Multiprocessing trade-offs and their effect on speed

processing can determine these trade-offs. Table 6.5 shows the effect of adding I/O ports on the system's speed. Adding I/O ports can speedup execution by reducing constraints on communication.

Network	Number of					CPC	Percentage Speedup
	PEs	MAC	busses	I/O	cycles		
24,24,24	12	4	6	1	100	0.09	*
24,24,24	12	4	6	2	57	0.05	44%

Table 6.5: I/O ports' effect on speed

6.3. Architecture optimization techniques

6.3.1. Neuron Splitting

Splitting the MA operations of neurons when the number of neurons in one layer does not divide evenly into the number of MACs available can enhance the performance of the system. The asymmetry of the network increases the number of cycles for the ANN computation as explained in Section 4.6.1. Table 6.6 presents the results of using neuron splitting as an optimization measure for several networks. This shows improvements as high as 40% over the networks synthesized without splitting neurons.

Architectural trade-offs	Network	NMAC	NB	NC	CPC	Speedup
No splitting	1,20,1	4	6	36	0.90	*
Splitting	1,20,1	4	6	25	0.62	40%
No splitting	1,20,1	2	4	46	1.15	*
Splitting	1,20,1	2	4	37	0.93	20%
No splitting	20,3,1	2	4	49	0.78	*
Splitting	20,3,1	2	4	42	0.67	15%
No splitting	40,10,1	3	5	167	0.41	*
Splitting	40,10,1	3	5	154	0.38	7.3%

Table 6.6: Effect of splitting neurons

6.3.2. Deep Pipelining

FUs with different degrees of pipelining can dramatically modify the behavior of the system. Considering the MAC of Chapter 4 that includes the threshold unit, the stages for the different hardware units being as follows:

Multiplier 2 clock cycles

Adder 1 clock cycle

Truncator 1 clock cycle

MAC stages	Clock cycle	Network	NMAC	NB	NC	Delay	Speedup
{2,1,1}	60 ns	40,10,1	4	6	128	7.68 µs	*
{4,2,2}	45 ns	40,10,1	4	6	135	6.08 µs	21%
{2,1,1}	60 ns	40,10,1	3	4	167	10.02 µs	*
{4,2,2}	45 ns	40,10,1	3	4	174	7.83 µs	22%
{2,1,1}	60 ns	40,10,1	2	3	221	13.26 µs	*
{4,2,2}	45 ns	40,10,1	2	3	222	9.99 µs	25%

Table 6.7: Effect of deeper pipelining of MACs

If deeper pipelining is introduced such as doubling the stage clock cycles to {4,2,2}, then the result is as shown in Table 6.7. The speedups shown are a result of the decrease in the total cycle time as explained in Section 4.6.3. The improvement seen is close to the gain in the clock cycle of 25%. The use of separate multipliers and adders with

local interconnects as suggested in chapter 4 can further enhance the performance by providing a pipeline with less constraints.

6.3.3. Activation function implementation

The use of a separate FU for the nonlinear function calculation may be required. Its use rather than the use of the unit provided on the MAC may result in a slight improvement. In the examples shown in Table 6.8, a slight improvement is shown when a single separate threshold is used. Using 4 of these units may not, however, be desirable since the resulting speedup is not high. This is due to the fact that nonlinear operations can be performed in parallel with MAs which take a longer time to execute. One threshold can therefore suffice to finish the nonlinear operations of all MACs while they are executing new operations. This is specifically true when the activation function is implemented by a look-up table.

Architectural trade-offs	Network	NB	NC	CPC	Speedup
1 Thresh. on each MAC	12,12,12	6	83	0.29	*
1 separate threshold	12,12,12	6	78	0.27	6%
4 separate thresholds	12,12,12	6	76	0.26	8.4%

Table 6.8: Threshold implementation as part of MAC or separate unit

6.4. Performance Comparison

The Delta II ANS processor achieves **4** CPC (Connections Per Cycles). The Balboa HNC uses an i860 with **4.4** CPC for learning (or 4.4 CUPS) and **1.6** CPC for recall. The GCN, a network of 128 PEs each being implemented by an Intel 80860 achieves **6.4** CPC. The optimum architectures presented in this paper achieve consistently less than **1** CPC (Recall). The best architectures produce results as low as **0.31** CPC (Recall) and **0.63** CPC (Learning). These results are for single processors. For multi-processor performance evaluation, the NetTalk network [59] with 203, 60 and 26 neurons partitioned on 16 processors is executed in the recall phase with **0.039** CPC. With a system

clock of *45 ns*, this amounts to *750 MCPS*.

Our architectures outperform the above mentioned systems since they have been optimized for the specific applications using the synthesis tool. Furthermore, special FUs targeting ANNs have been used. Also, implementing networks as straight line codes eliminates all the overhead associated with branching. Therefore, the synthesis methodology and the architecture presented here result in a better performance than general purpose neural processors by avoiding the overhead induced by general purpose systems and optimizing the architectures for specific applications.

In the case of the robotic BP network presented, the real time recall requirements of *17* μs can be obtained using 3 MACs (with 1 *lr*) and 4 busses with a *100ns* clock resulting in *15.8* μs.

6.5. Conclusion

The architectural trade-offs explored in this chapter prove the importance of using a synthesis tool to investigate the design optimizations that can be performed in the synthesis of neural network digital hardware. The performance evaluation further indicates that the optimized architectures resulting from the synthesis tool can outperform other architectures for the same applications. The ease with which different degrees of parallelism can be investigated allows for customized architectures that are flexible enough for use in the adaptive applications targeted. The use of the tool allows for some non-obvious architectural trade-offs that would otherwise have gone unnoticed by the designer.

Chapter 7:

Verification and Implementation

7.1. Introduction

This chapter completes the synthesis framework by presenting a VHDL verification methodology to ensure the correctness of the synthesis procedures by simulating the resulting hardware. A VHDL behavioral model of the multiple bus/ functional unit architecture is presented. An example XOR backpropagation network is synthesized and the resulting control is used in VHDL simulation proving the correctness of the synthesis procedures. This chapter also presents an example VLSI implementation of the minimum configuration of the proposed architecture and details the area and delay measurements used in chapter 6 for the bottom-up approach for evaluating architectural trade-offs.

7.2. VHDL verification of synthesis

A number of synthesis systems have been implemented using VHDL. The possibility of describing a system at various levels of abstractions in VHDL is particularly useful. The synthesis system proposed uses VHDL to specify the architecture after a complete RT description is obtained. Verification is done through the simulation of the given description and by obtaining accurate estimates of execution time for specific styles of implementations. The VHDL part can detect any errors in the generation of the RT description by detecting memory and bus contentions. The VHDL description of the system is a behavioral one. Further extensions to the verification system include the structural VHDL modelling of the library units to allow for accurate simulations of the intended hardware.

7.2.1. VHDL unit modelling

The units used in the processor are modelled in a behavioral manner. The MACs

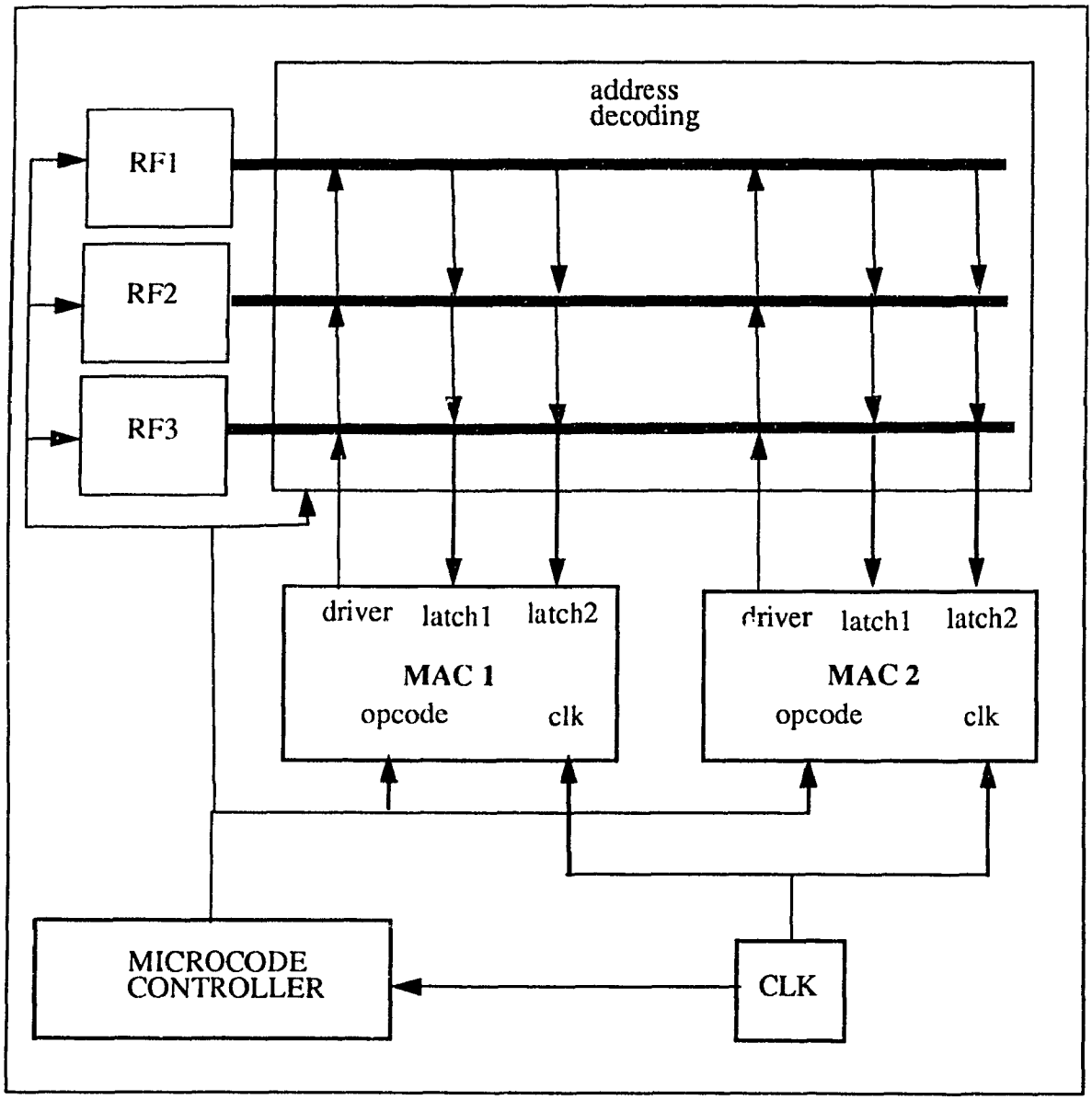
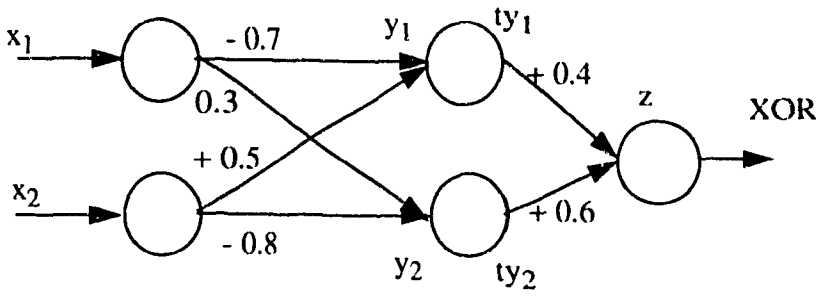


Figure 7.1: Example of a VHDL model

are separate units with ports for two input latches and one output driver interfaced to the busses. They further include ports for the clock and an opcode from the controller which specifies the type of operation to be executed by the MAC. The pipelining is not explicit but is guaranteed by appropriate delays for each operation. The register files (with a fixed number) are modelled as arrays with the indices being read from the microcode. The busses are modelled using an address decoding scheme that selects the appropriate registers in the register files and the appropriate MAC and latch of the FUs. This decoding



x_1	x_2	y_1	y_2	ty_1	ty_2	z	XOR
0	0	0	0	0	0	0	0
0	1	0.5	- 0.8	1	0	0.4	1
1	0	- 0.7	0.3	0	1	0.6	1
1	1	- 0.2	-0.5	0	0	0	0

Figure 7.2: XOR network example

is further divided for the two clock phases. Units that generate clock signal and control the simulation are also used. The model for the example architecture is given in Appendix II.

7.2.2. Example VHDL model

An example model of a system using 3 busses and 2 MACs is shown in Figure 7.1. A small XOR network is synthesized with the synthesis tool presented in this thesis using 2 MACs and 3 busses. The resulting control sequence is converted into VHDL code. The results of the synthesis and the VHDL microcode description are shown in Appendix III. The XOR network has been used as an example of a non-linearly separable problem and as a result, one that a single layer perceptron cannot solve. Its use in this thesis is merely to indicate the correct operation of the control generation and the synthesis procedures and as a working example to demonstrate the VHDL behavioral simulation. The XOR network is then simulated in VHDL and the results evaluated. The XOR network is as shown in Figure 7.2. The weights for the 2,2,1 BP network were set as shown (not derived using BP

algorithm). In the given table, *t* preceding a variable indicates its value after truncation. In the same way XOR is the truncation of z, which in this case may not be needed (a linear neuron could be used), but is included for completeness. The activation function (truncation) is simply:

$$\begin{aligned} trunc(x) &= 0 \quad \text{for} \quad x \leq 0 \\ trunc(x) &= 1 \quad \text{for} \quad x > 0 \end{aligned} \tag{7.1}$$

. The resulting VHDL simulation (using the Valid VHDL interface) proves the correctness of the synthesis algorithms and control generation procedures and is shown in Figure 7.3 (Obtained from the Valid VHDL interface). The cycle time is set at 36ns with a two phase clock (ϕ1: Read and ϕ2: Write). Two MACs, denoted FU1 and FU2, and three busses with three corresponding register files, RF1, RF2 and RF3, are used. For each FU, the values of the opcode, latches and output driver are given. Only the registers that are modified during execution are shown. The final value of every item is given in bold figures. the vertical bars indicate changes in the value of each signal. If the new value of a signal is the same as the old value, no change appears on this graph. This case corresponds to x_1=1 and x_2=0 resulting in z=1 on the tenth cycle as shown by FU1.DRIVER. The large negative numbers are initial values set in the initiation phases. All four inputs to the network produce the correct results according to the XOR truth table.

7.3. PE Chip Implementation

7.3.1. Custom Arithmetic Cells

A set of VLSI modules are designed specifically for the implementation of the architectures presented in this thesis using the NT 1.2 µm CMOS technology[1]. The designs presented here use standard library cells. Future extensions of this work include

1. The cell designs were done by project groups in VLSI graduate and undergraduate courses under the supervision of Dr. B. Haroun.

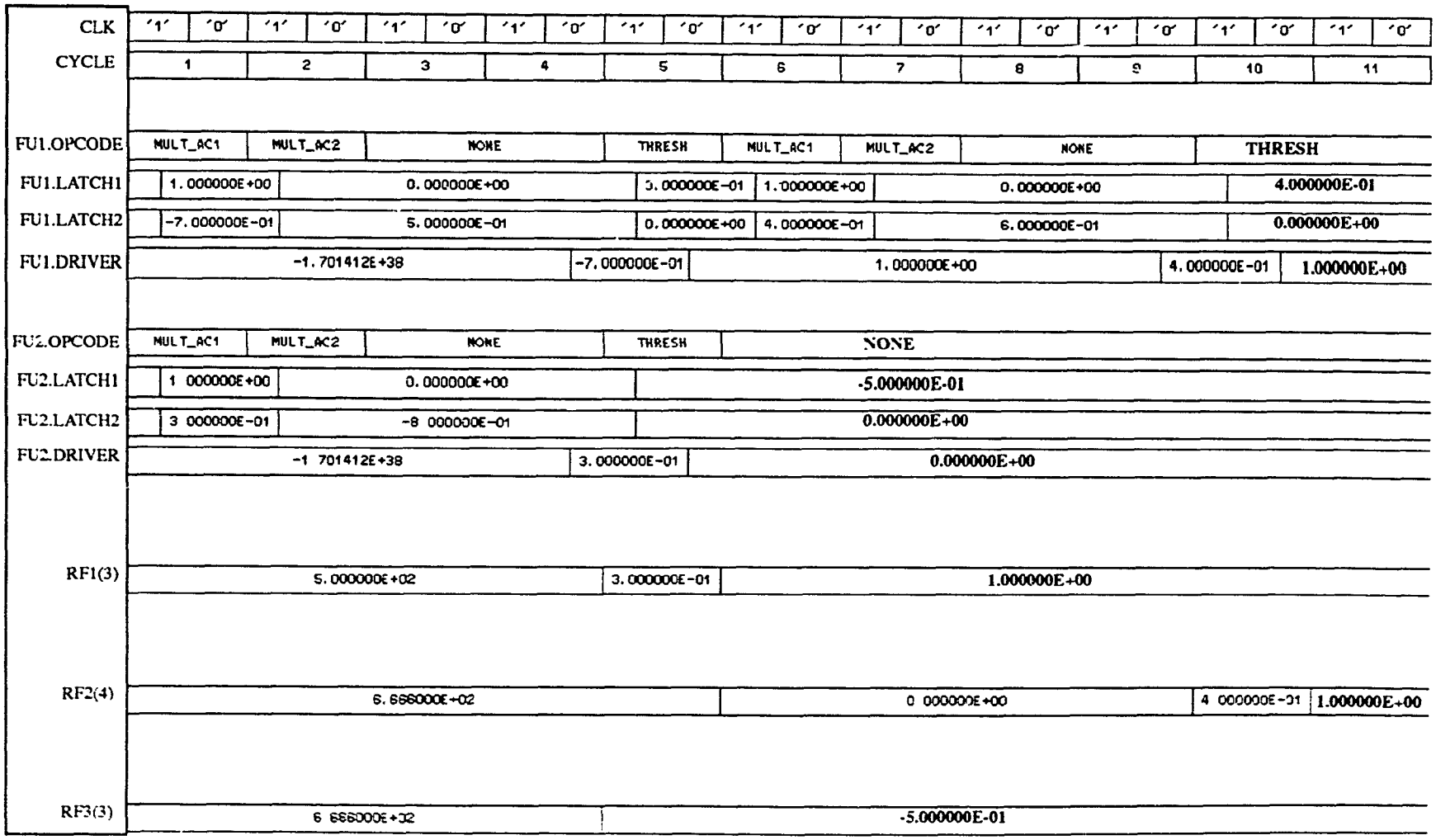


Figure 7.3: VHDL timing diagram for XOR

the design of various customized library units that could be interfaced to the synthesis system. The use of the currently available designs, however is important to gather an understanding of the area versus delay trade-offs as discussed in chapter 6. The main purpose of the PE chip implementation is to present an idea of the silicon area involved in such designs.

7.3.2. Multiplier

The multiplier used is based on the modified booth algorithm [60]. The algorithm involves bit-pair recoding. This scheme decreases the number of rows to be added together at the last stage of the multiplier speeding up the computation. The implementation of a 16 x 9 multiplier yields a regular structure occupying 1.4 mm^2. Its total delay is 18 ns (The area and delay do not include the adder stage). The multiplier is designed with standard library cells but using over the cell routing. It is to note that this multiplier can be highly pipelined.

7.3.3. Adder

The adder implemented was a 25 bit Binary Carry Lookahead adder [61]. In this design, the carry propagation time is reduced by using an associative operator to compute the carry in a binary tree fashion and generate the sum bits using the intermediate carries. The implementation using standard library cells and automatic place and route techniques resulted in a 0.6 mm^2 adder with a delay of 12 ns.

7.3.4. Multiply-Accumulate Unit

The multi-purpose MAC described in chapter 4 is designed as part of the FUNN (Functional Units for Neural Networks) project aiming at the design of hardware units (arithmetic and otherwise) specifically designed for ANNs and the synthesis procedure presented in this thesis. The area of the multi-purpose MAC is 4.6 mm^2. This figure includes the multiplier, adder, truncator, and all the associated multiplexing and decoding logic and local control as well as the routing. The MAC was again designed using

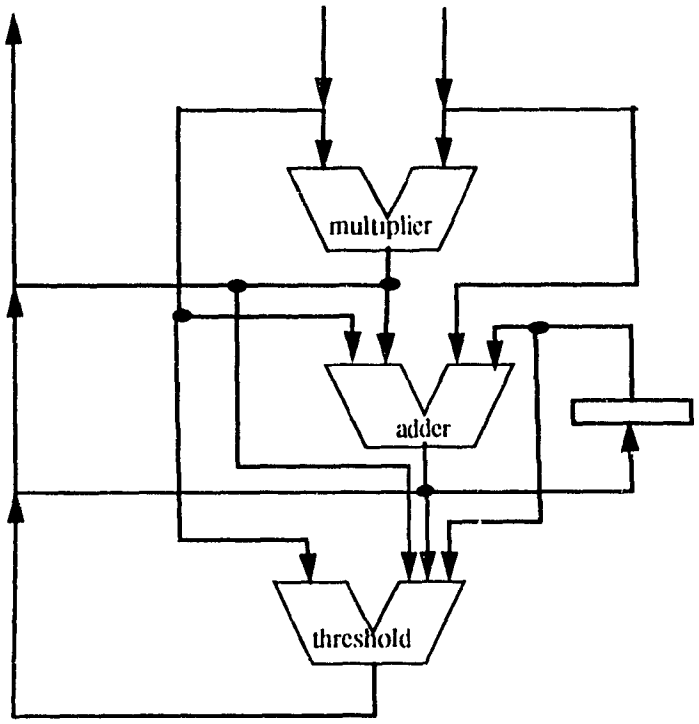


Figure 7.4: MAC organization

automatic place and route algorithms. It is pipelined so that a multiplication takes 2 cycles and an addition takes 1 cycle resulting in 15ns delay per stage. The resulting MAC has the organization as shown in Figure 7.4.

7.4. Neural processor implementation

The areas of the register files/RAMs used in the architecture were calculated from actual SRAM layouts. The areas of the bus latches and tristate drivers were also calculated. These values were used in the architectural trade-off exploration presented in chapter 6. The delays of the busses are estimated to be 15ns on each clock phase for all possible architectures, the resulting system would run at 45ns.

An example minimum configuration (1 bus, 1 register file and 1 MAC) is shown in Figure 7.5. The MAC used is the one described above. The SRAM is a 512x16 RAM. The resulting chip layout is then shown in Figure 7.6. This chip is presented here as an

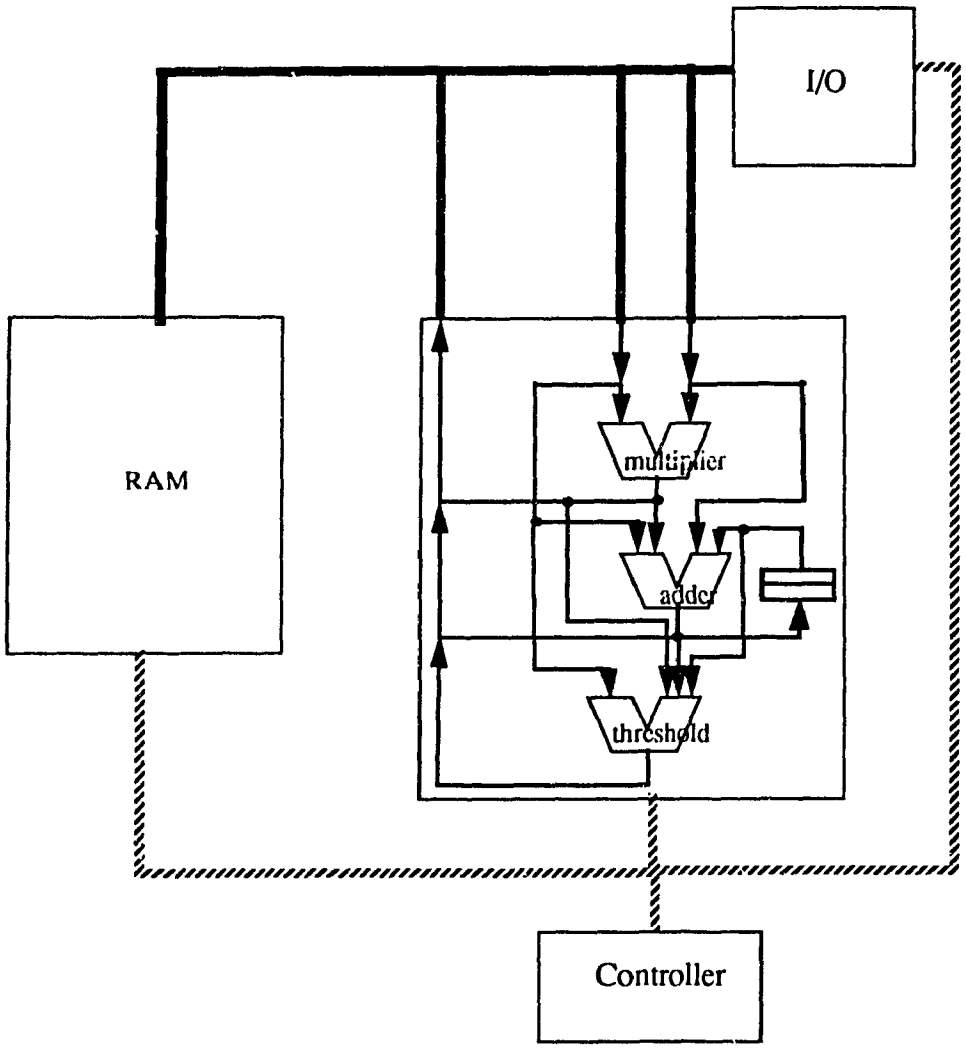


Figure 7.5: Minimum configuration organization

indication of the silicon areas involved in such designs.

It is clear that for the same amount of memory more than one MAC can be used without increasing the area dramatically while allowing for parallelism.

It should be noted that a much more compact design is achievable with the use of customized cells, detailed hand routing and optimization, as well as DRAMs.

7.5. Area measurements for synthesis

The area used for each MAC is the area of the MAC logic added to the area of the local registers and the multiplexing involved resulting in 4.6 mm^2 for a MAC with a single

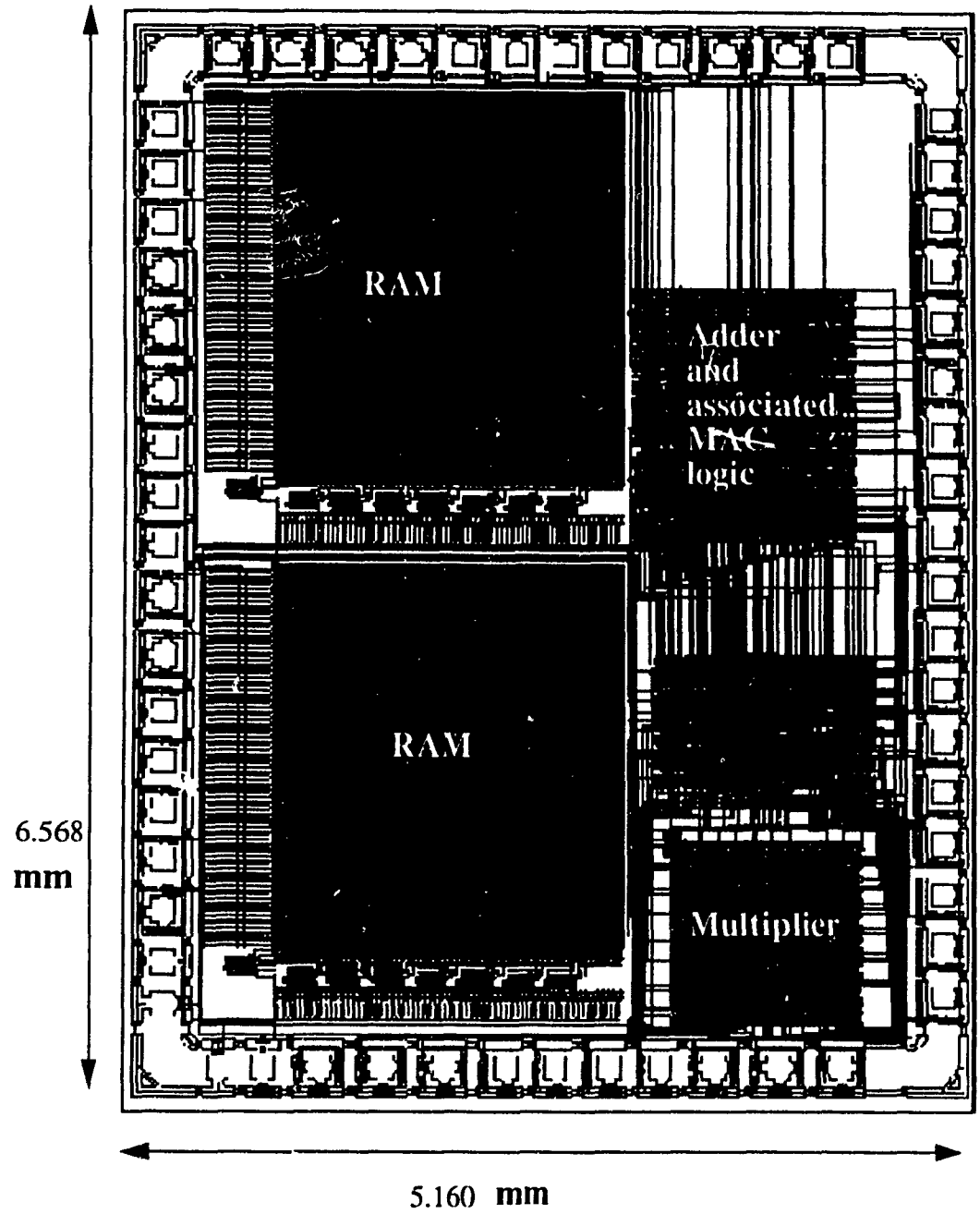


Figure 7.6: Chip layout of minimum configuration

accumulator, 4.9 mm^2 for one using two accumulators.

The bus area is proportional to the number of FUs and includes the areas of the latches and bus drivers.

The RAM area was calculated based on the number of bits N and the number of

words M using a fixed per bit area of the RAM cells in addition to the areas consumed by the sense amplifiers and column and row decoders used for each configuration.

The controller used is also a RAM. The number of words is twice the number of clock periods (one word per clock phase). The number of bits, however depends on the configuration. The number of bits for the architectures used is:

$$nb = NFU\,[\,(\log_2 NOP) + NP \times \log_2 NB\,] + Address \qquad (7.2)$$

where *NFU*, *NP* and *NB* are the number of FUs, I/O ports for each unit and busses respectively. NOP is the number of different operations that could be executed on a FU. *Address* consists of the number of bits needed to address the register files. This is dependent on the number of register files (or busses), the number of words in each register file as well as the type of encoding used.

7.6. Conclusion

This chapter completed the synthesis framework presented in Chapter 5 by introducing the VHDL modeling of the architectures. An example XOR backpropagation network is given to prove the correctness of the synthesis procedures. The VLSI area and delay measurements used in chapter 6 are also given and the design of the minimum configuration of one RAM and one MAC is presented as an indication of the silicon areas involved in the design of such architectures as the ones proposed in this thesis.

Chapter 8:

Conclusion

This thesis presents a novel approach to the design of digital neural networks. Real-time adaptive neural network applications are targeted and an ASIC design methodology is designed to meet their requirements. A flexible bus style data path is suggested that uses pipelined units, such as the multi-purpose MAC, that are specifically designed for neural network applications. This architecture allows a certain degree of parallelism within each processor, something that most digital implementations of ANNs fail to provide. The architecture is further developed in a multi-processing environment using systolic communication. The parallelism involved in such architectures, as well as the design trade-offs involved, are investigated using an architectural design synthesis tool, written in Prolog, that performs FU, bus and memory allocation, reservation and binding. The algorithms involved in this tool take advantage of the nature of ANN algorithms and allow compound operations such as MAs to be used, guaranteeing a more compact schedule. The tool uses heuristics specific to neural networks that help in the pre-ordering of the operations and the scheduling and binding processes. Multi-processing synthesis is also performed where the developed tool systolizes the neural algorithms using complicated PEs with internal parallelism. Several architectural trade-offs can be examined using the tool such as the addition of local storage registers, the use of deeper FU pipelining, neuron splitting, the use of special FUs and the number of busses needed. These trade-offs vary between applications depending on the type of algorithm used and the topology of the networks. The tool is therefore needed to examine the cases where such optimizations are useful. The synthesis framework is completed by the use of a VHDL simulator that verifies the correctness of the synthesis procedures and the microcode generation of the controller, and by VLSI library units that can be used to obtain accurate area and delay measurements as well as the design of the ANN ASICs.

The networks synthesized using this CAD tool result in architectures that outperform other systems for the same applications. The benefits of the optimizations are also shown in the synthesis results proving the importance of the use of automated design techniques.

In general, the use of high-level synthesis techniques to investigate the design of digital neural networks for real-time adaptive applications is proven to result in optimized architectures that outperform other architectures previously designed and published. Also, the synthesis tool developed can be used to implement optimized architectures by extensively searching the design space while reducing the design cycle clearing the way for cheap ASICs for ANN applications.

Completing the full CAD framework presented in this thesis to obtain a general system that accommodates all types of neural networks and provides support from the specification through the chip development is a monumental task. Nevertheless, the work in this thesis provides the foundations for all possible expansions. Suggested future work in this area includes the design of a *complete VLSI library* equipped with several design styles of each unit needed in such implementations. The characteristics of these units such as the areas, delays and power consumption could be used as part of the criteria in judging the synthesized designs. Even though the synthesis methodologies presented in this thesis are general, many important issues in synthesis were not addressed such as *the use of conditional branching*. This issue was avoided partly because it would strip the customized architecture of its advantages and would slow down the overall system. It is the case, however, that some neural networks rely on probabilistic algorithms that require specific considerations for branches and control in general. The synthesis tool presented is mainly a data path synthesis tool. Even though a microcode controller can be generated, it is generally favorable to include the control in the synthesis procedure. This type of *mixed control data path synthesis* is a difficult area that needs, nevertheless, to be examined. The techniques used to implement the MA operations can be used to combine any two or more operations. With appropriate functional units, this type of synthesis would result in much

more compact schedules. A regular SFG can be used as input to the tool, it is beneficial, however, to provide *parsers* that could generate these SFGs from simple descriptions of neural networks. The reason for this being the complication of the SFGs for large networks. It is indeed impractical to propose writing a signal flow graph for each network to be synthesized. Appropriate parsers were implemented for backpropagation and counterpopagation networks. Hopfield networks could also be accommodated. There is, however, a large number of networks which need have the same support. In the multi-processing synthesis, issues like *partitioning* were relegated to the user. While the regularity of fully connected neural networks allows easy partitioning, networks with special interconnection patterns may prove to be more difficult. Synthesis procedures should be developed to attack this sort of problems.

These issues can be investigated and the resulting work can be incorporated in the synthesis framework presented in this thesis. It should be noted that the field of neural networks is still growing and that neural network ASICs are still gaining ground in commercial applications. Whatever the future trends of neural networks may be, it is guaranteed that they can always make use of automated synthesis techniques for the design of their ASIC implementations.

References

[1] P. Treleaven, M. Pacheo and M. Vellasco, "VLSI Architectures for Neural Networks," IEEE MICRO Magazine, pp. 8-27, Dec. 1989.

[2] U. Ramacher, "Guidelines to VLSI Design of Neural Nets," in Ramacher U., Ruckert U., ed. VLSI Design of Neural Networks, Kluwer Academic Publishers, Boston, 1991, pp. 1-18.

[3] J. Burr, "Digital Neural Network Implementations," in Antognetti, P. and Milutinovic, V. eds. Neural Networks: Concepts, Applications and Implementations, Vol II. Prentice-Hall, Toronto, 1991, pp. 237-285.

[4] R. Hecht-Nielsen, Neurocomputing, Addison-Wesley, New York, 1989.

[5] B. Soucek and M. Soucek, Neural and Massively Parallel Computers, John Wiley & Sons, 1988.

[6] P. D. Wasserman, "Neural Computing: Theory and Practice," Van Nostrand Reinhold, New York, 1989.

[7] Eckmiller, R., Ed., "Advanced Neural Computers," Elsevier Science Publishing Company, New York, 1990.

[8] Anderson Dana Z., ed., "Neural Information Processing Systems," American Institute of Physics, New York, 1988.

[9] R. Eckmiller, C. Malsburg, Eds. "Neural Computers," Springer-Verlag, New York, 1988.

[10] I. Aleksander, Ed., "Neural Computing Architectures," North Oxford Academic Publishers, 1989.

[11] B. Widrow and M. Lehr, "30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation," Proceedings of the IEEE, Vol. 78, No. 9, September 1990, pp. 1415-1442.

[12] J. J. Hopfield and D. W. Tank, "Neural Computation of Decisions in Optimization Problems," Biol. Cyber. 52, 141-152, 1985.

[13] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," Biol. Cyber, Vol. 36, pp. 193-202, 1980.

[14] B. White and M. Elmasry, "The Digi-Neocognitron: A Digital Neocognitron Neural Network Model for VLSI," IEEE Transactions on Neural Networks, Vol. 3, No. 1, January 1992.

[15] R. Hecht-Nielsen, "Applications of Counterpropagation Networks," Neural networks, Vol. 1, pp. 131-139, 1988.

[16] B. Kosko, "Bidirectional Associative Memories," IEEE Transactions on Systems, Man and Cybernetics, vol. 18, no.1, pp. 49-60, Neural Networks, Vol. 18, no. 1, pp.49-60, Jan. 1988.

[17] G. Carpenter and S. Grossberg, "Neural Dynamics of Category Learning an Recognition: Attention,

Memory Consolidation, and Amnesia," in J. Davis, et al. Eds. Brain Structure, Learning and Memory, AAAS Symposium Series, pp. 239-285 1986.

[18] C. A. Mead, Analog VLSI and Neural Systems, Addison-Wesley, 1989.

[19] V. Zeman, R. V. Patel and K. Khorasani, "A Neural Network Based Approach for the Control of Flexible-Joint Manipulators," to appear in the 8th International Conference on CAD/CAM, Robotics and Factories of the Future, Metz, France, August 1992.

[20] B. Soucek, Neural and Concurrent Real-Time Systems, Wiley, Toronto, 1989.

[21] J. Lont and W. Guggenbuhl, "Analog CMOS Implementation of a Multilayer Perceptron with Nonlinear Synapses," IEEE Transactions on Neural Networks, Vol. 3, No. 3, May 1992, pp. 457-465.

[22] C. Mead, "Neuromorphic Electronic Systems," Proceedings of the IEEE, Vol. 78, No. 10, October 1990, pp. 1629-1636.

[23] S. Kung and J. Hwang, "Parallel Architectures for Artificial Neural Nets," International Joint Conference on Neural Networks 1990, Vol. II pp. 165-172.

[24] A. Hiraiwa et al. "Implementation of ANN on RISC Processor Array," International Conference on Application Specific Array Processors, 1990, pp. 677-688.

[25] S. Garth, "A Chipset for High Speed Simulation of Neural Network Systems," IEEE First International Conference on Neural Networks, Vol. III, 1987, pp. 443-452.

[26] M. Duranton and J. Sirat, "Learning on VLSI: A General Purpose Digital Neurochip," International Joint Conference on Neural Networks, 1989.

[27] Hirai et. al. "A Digital Neurochip with Unlimited Connectability for Large Scale Neural Networks," International Joint Conference on neural Networks, Vol. II, pp.163-169.

[28] J. Ouali and G. Saucier, "Silicon compiler for neuro-ASICs," International Joint Conference on Neural Network, Vol. II, 1990, pp. 557-561.

[29] M. Tomlinson et. al. "A digital Neural Network Architecture for VLSI," International Joint Conference on Neural Networks, Vol. II, 1990, pp. 545-550.

[30] W. Wike et al. "The VLSI Implementation of STONN," International Joint Conference on Neural Networks," Vol. II, 1990, pp. 593-598.

[31] D. Van den Bout and T. Miller, "TInMANN: The Integer Markovian Neural Network," International Joint Conference on Neural Network, Vol. II, 1989.

[32] D. Hammerstrom, "A VLSI Architecture for High-Performance, Low-Cost, On-chip Learning,"

International Joint Conference on Neural Networks, San Diego, 1990, Vol. II, pp. 537-543.

[33] S. Kung and J. Hwang, "Digital VLSI Architectures for Neural Networks," International Symposium on Circuits and Systems 1989, pp. 445-448.

[34] A. Hiraiwa et al. "A Two-Level Pipeline RISC Processor Array for ANN," International Joint Conference on Neural Networks, 1990, Vol. II pp. 137-140.

[35] G. Works, "The Creation of Delta: a New Concept in ANS Processing," IEEE Conference on Neural Networks, Vol. II, pp. 158-164, 1988.

[36] D. Jackson and D. Hammerstrom, "Distributing Back Propagation Networks Over the Intel iPSC/860 Hypercube," Vol. I pp. 569-574.

[37] G. Blelloch and C. Rosenberg, "Network Learning on the Connection Machine," The Tenth International Joint Conference on Artificial Intelligence, pp. 323-326, 1987.

[38] D. Pomerlau, G. Gusciora, D. Touretzky and H. T. Kung, "Neural Network Simulation at WARP Speed: How we got 17 Million Connections per Second," IEEE International Conference on Neural Networks, San Diego, CA, 1988, Vol. II, pp. 143-150.

[39] M. McFarland, A. Parker and R. Camposano, "Tutorial on High-Level Synthesis," ACM/IEEE Design Automation Conference, 1988, pp. 330-336.

[40] D. Ku and G. De Micheli, "Hercules- A System for High Level Synthesis," Design Automation Conference, June 1988, pp. 483-488.

[41] J. Lis and D. Gajski, "Synthesis from VHDL," International Conference on Computer Design, 1988, pp. 378-381.

[42] C. Hwang, et al. "A Formal Approach to the Scheduling Problem in High Level Synthesis," IEEE Transactions on Computer-Aided Design, Vol. 10, No. 4, April 1991, pp. 464-475.

[43] S. Devadas and R. Newton, "Algorithms for Hardware Allocation in Data Path Synthesis," IEEE Transactions on Computer-Aided Design, Vol. 8, No. 7, July 1989, pp. 768-781.

[44] E. Lagnese and D. Thomas, "Architectural Partitioning for System Level Synthesis of Integrated Circuits," IEEE Transactions on Computer-Aided Design, Vol. 10, No. 7, July 1991.

[45] B. Kernighan and S. Lin, "An efficient Heuristic Procedure for Partitioning Graphs," Bell Systems Technical Journal, pp. 291-307, February 1970.

[46] C. Ramamoorthy and H. Li, "Some Problems in Parallel and Pipeline Processing," Proceedings of COMPCON, IEEE, pp. 177-180. 1975.

[47] C. Leisersen and J. Saxe, "Optimizing Synchronous Systems," Journal of VLSI and Computer Systems 1, no. 1 pp. 41-67, 1983

[48] N. Park and A. Parker, "SEHWA: A Program For Synthesis of Pipelines," 23rd Design Automation Conference, 1986, pp. 454-460.

[49] B. Widrow and R. Winter, "Neural Nets for Adaptive Filtering and Adaptive Pattern Recognition," IEEE Computer magazine, March 1988, pp. 25-39.

[50] L. Jackson, Digital Filters and Signal Processing. Kluwer Academic Publishers,Birmingham, MA, 1986.

[51] H. De Man et al. "Architecture-Driven Synthesis Techniques for VLSI Implementation of DSP Algorithms," Proceedings of the IEEE, Vol. 78, No. 2, February 1990, pp. 319-335

[52] B. Haroun and M. I. Elmasry, "Architectural Synthesis for DSP Silicon Compilers," IEEE Transactions on Computer-Aided Design, vol. 8, pp. 431-447, April 1989.

[53] E. Torbey and B. Haroun, "Architectural Synthesis for Digital Neural Networks," International Joint Conference on Neural Networks, Baltimore, Maryland, June 1992, Vol. II, pp. 601-606.

[54] J. Vlontzos and S. Kung, "Digital Neural Network Architecture and implementation," in Ramacher U. and Ruckert U, ed. VLSI Design of Neural Networks, Kluwer Academic Publishers, Boston, 1991, pp. 205-228.

[55] T. Williams and K. Parker, "Design for Testability-A survey," Proceedings of the IEEE, Vol. 71, No. 1, January 1983, pp. 98-110.

[56] B. Haroun and E. Torbey, "Synthesis of Multiple Bus/Functional Unit Architectures Implementing Neural Networks," to appear in the proceedings of the International Conference on Computer Design, Cambridge, Massachusetts, October 1992.

[57] B. Haroun and M. I. Elmasry, "VLSI Architectural Synthesis for Large Data Storage Signal Processing Algorithms," International Conference on Microelectronics, pp. 26-29, Dec. 1991.

[58] J. Holt and J. Hwang. "Finite Precision Error Analysis of Neural Network Electronic Hardware Implementations," International Joint Conference on Neural Networks, Vol. I, pp-519-525, 1991.

[59] T. Sejnowski and C. Rosenberg, "Parallel Networks that Learn to Pronounce English Text," Complex Systems 1(1), 1987, pp. 145-168.

[60] M. Anaratone. Digital CMOS Circuit Design. Kluwer Academic Publishers, 1986.

[61] R. Brent and H.T. Kung, "A Regular Layout for Parallel Adders," IEEE Transactions on Computers, Vol. C-31, No. 3, March 1982, pp.260-264.

Appendix I:

Prolog Implementation Issues

1.1. Network description

```
% This is the user's network description
% network([
%       [layer_name, num_of_slabs,
%       slab([...[slab_name,num_of_neurons,
%                       neuron(neuron_type,[...[?neuron, ?priority]...],_,_),
%                       xfer_fctn([...[pass,xfer_fctn_type]...]),
%                       out_weights(connec_type, [...[weight_name]...]),
%                       [error_list]])].

% Example: Backpropagation with one hidden layer
%               with 3,4,2 neurons respectively.
%               no bias

network(backpropagation,[
        [layer-1,input,1,
                slab([[slab-1-a,4, neuron(input_node, Neuron_list),
                        xfer_fctn([fwd,linear],[bwd,back-error]),
                        out_weights(none, []),
                        Error_list]])],
        [layer-2,hidden_1,1,
                slab([[slab-2-a,4, neuron(perceptron, Neuron_list),
                 xfer_fctn([fwd,sigmoid],[bwd,back-error]),
                        out_weights(full, Weight_list),
                        Error_list]])],
         [layer-3,hidden_2,1,
                slab([[slab-3-a,4, neuron(perceptron, Neuron_list),
                        xfer_fctn([fwd,sigmoid],[bwd,back-error]),
```

```
out_weights(full, Weight_list).
Error_list]])]).
```

1.2. Hardware description

```
% H = hardware(functional_units([..,[Type,[operator_id,..]],..]),
busses([..,Bus_id,..])).


hardware_allocation(H):-
eq(H,hardware(fus([[mac,2,[mac1,mac2]],[io,1,[io1]],[output,1,[out1]]],
        busses([bus_1-all,bus_2-all]))).
```

1.3. FU type description

```
%type_def(FU_type,Opn_type,Num_of_inputs,Delay,style(Design_style,N
umber_of_stages,Initiation_list),local_storage(num_of_regs,[reg_list]),Ar
ea,Power,Design_num).


type_def(mac,mul_ac,2,1,6,style(pipelined,3,[[s-1,[1]],[s-2,[2]],[s-
3,[3]],[s-4,[4]],[s-5,[5]],[s-6,[6]],[s-7,[0]],[s-8,[0]]]),1,_,_,_).

type_def(mac,mul,2,1,2,style(pipelined,2,[[s-1,[1]],[s-2,[2]],[s-3,[3]],[s-
4,[4]],[s-5,[5]],[s-6,[6]],[s-7,[0]],[s-8,[0]]]),1,_,_,_).

type_def(mac,add,2,1,1,style(non_pipelined,1,[[s-1,[0]],[s-2,[0]],[s-
3,[0]],[s-4,[0]],[s-5,[1]],[s-6,[2]],[s-7,[0]],[s-8,[0]]]),1,_,_,_).

type_def(mac,sub,2,1,1,style(non_pipelined,1,[[s-1,[0]],[s-2,[0]],[s-
3,[0]],[s-4,[0]],[s-5,[1]],[s-6,[2]],[s-7,[0]],[s-8,[0]]]),1,_,_,_).

type_def(mac,trunc,1,1,1,style(non_pipelined,1,[[s-1,[0]],[s-2,[0]],[s-
3,[0]],[s-4,[0]],[s-5,[0]],[s-6,[0]],[s-7,[1]],[s-8,[2]]]),1,_,_,_).

type_def(sorter,sort,2,0,1,style(non_pipelined,1,[[s-1,[1]]]),0,_,_,_).

type_def(io,in_out,1,1,2,style(non_pipelined,1,[[s-1,[1]]]),0,_,_,_).

type_def(input,in,1,1,1,style(non_pipelined,1,[[s-1,[1]]]),0,_,_,_).

type_def(output,out,1,1,1,style(non_pipelined,1,[[s-1,[1]]]),0,_,_,_).
```

1.4. Operations list

% O =

[...[Op_id,FU_type,Op_type,Input_list,Output,Start_time,End_time,FU_id -LR_id]...]

[ma-0-f-2-a-1,mac,mul_ac,edges,xt-2-a-1,1,4,mac1-1]

[n-2-a-1,mac,trunc,[xt-2-a-1,c-2],x-2-a-1,5,5,mac1]

1.5. Registers list:

[...[Reg_num,[Reg_id,Tag],[Opn_id,Input_tag],Wr_clock,Rd_clock,Bus_id,Register_file_id-Index]...]

[xreg-1,[x-1-a-1,1],[ma-0-f-2-a-1,1],0,1,bus-1,rf-1-r2]

[wreg-1,[w-1-a-1-2-a-1,1],[ma-0-f-2-a-1,2],0,1,bus-2,rf-2-wr1]

[xtreg-2,[xt-2-a-2,1],[n-2-a-2,1],4,5,bus-1,rf-1-r1]

[creg-3,[c-3,1],[n-3-a-1,2],0,10,bus-3,rf-3-cr3]

Appendix II:

VHDL Modelling Issues

2.1. Example architecture

```
use work.all;
use work.Types.all;
use work.all;
entity nnchip is
        generic (clock_period : integer := 50);
end nnchip;

architecture nnchip_arch of nnchip is
signal clk : bit;
signal stop_sim : boolean := FALSE;
signal cycle : word;
signal READ : microcode_rd;
signal WRITE : microcode_wr;
signal rf1,rf2,rf3 : reg_array;
signal
MAC1_l1,MAC1_l2,MAC2_l1,MAC2_l2,MAC1_dr,MAC2_dr:word;

        component CLOCK
                port(clk: out bit);
        end component;
        for all : CLOCK use entity clkgen(generator);

        component SIM
                port(clk: in bit; cycle: inout word);
        end component;
        for all : SIM use entity sim_control(sim_control_arch);
```

```
component MICRO
        port (clk: in bit; cycle: in integer;
        RD: out microcode_rd; WR: out microcode_wr);
end component;
for all : MICRO use entity MICROCODE(MICROCODE_arch);

component READ_BUS
        port (clk: in bit; RD : in microcode_rd;
                reg_file1: in reg_array;
                reg_file2: in reg_array;
                reg_file3: in reg_array;
MAC1_latch1,MAC1_latch2,MAC2_latch1,MAC2_latch2: out word):
end component;
for all : READ_BUS use entity BUS_rd(BUS_rd_arch);

component WRITE_BUS
port (clk: in bit; cycle: in word; WR : in microcode_wr;
        reg_file1,reg_file2,reg_file3: inout reg_array;
        MAC1_driver,MAC2_driver: in word);
end component;
for all : WRITE_BUS use entity BUS_wr(BUS_wr_arch);

component FU
port(clk: in bit; opcode: in op_codes; latch1, latch2 : in word;
        driver : out word);
end component;
for all : FU use entity MAC(MAC_arch);


begin
CLOCK1 : CLOCK port map(clk);
SIM1 : SIM port map(clk,cycle);
```

```
        WRITE_BUS1: WRITE_BUS
port map(clk,cycle,WRITE,rf1,rf2,rf3,MAC1_dr,MAC2_dr),
        MICRO1 : MICRO port map(clk,cycle,READ,WRITE);
        READ_BUS1: READ_BUS
port map(clk,READ,rf1,rf2,rf3,MAC1_l1,MAC1_l2,MAC2_l1,MAC2_l2);
        FU1 : FU
port map(clk, READ.opcode1,MAC1_l1,MAC1_l2,MAC1_dr);
        FU2 : FU
port map(clk, READ.opcode2,MAC2_l1,MAC2_l2,MAC2_dr);
        end nnchip_arch;
```

2.2. Clock and simulation control

```
use work.Types.all;
use work.all;
entity clkgen is
        generic (clock_period : integer := 36);
        port(clk: out bit);
end clkgen;

architecture generator of clkgen is

begin
        CLOCK_CONTROL : process
        begin
                        clk <='1';
                        wait for clock_period/2 * 1ns;
                        clk <='0';
                        wait for clock_period/2 * 1ns;
                        end process CLOCK_CONTROL;
end generator;

use work.Types.all;
```

```
entity MAC is
        port(clk : in bit; opcode: in op_codes; latch1, latch2 : in word;
driver : out word);
end;

use work.Types.all;
use work.all;
entity sim_control is
        port(clk: in bit; cycle: inout word := 0);
end sim_control;

architecture sim_control_arch of sim_control is
        begin
        process(clk)
                        begin
                        if clk='1' then
                        case cycle is
when 15 => assert FALSE report "Stopping simulation" severity failure;
when others => cycle <= cycle + 1;
                        end case;
                        end if;
        end process;
end sim_control_arch;
```

2.3. MAC behaviour

```
use work.Types.all;

entity MAC is
        port(clk : in bit; opcode: in op_codes; latch1, latch2 : in word;
        driver : out word);
end;
```

```
architecture MAC_arch of MAC is
        signal reg: word:=0;
        begin
        process
                        begin
                        wait until latch2'event;
                                        case opcode is
when ADD => driver <= transport latch1 + latch2 after stage_delay ;
when SUB => driver <= transport latch1 - latch2 after stage_delay ;
when THRESH => if latch1 < latch2 then driver <= transport 0 after
stage_delay ;
        else driver <= transport 1 after stage_delay ;
end if;
when MULT => driver <= transport latch1*latch2 after M_delay;
when MULT_AC1 => reg <= transport latch1 * latch2 + reg after
stage_delay;
when MULT_AC2 => reg <= 0 after 3*cycle_delay - Rd_delay - Wr_delay;
                        driver <= transport latch1*latch2 + reg after
3*cycle_delay -Rd_delay - Wr_delay;
when NONE => null;
                                        end case;
        end process;
end MAC_arch;
```

Appendix III:

XOR Synthesis and Simulation Results

3.1. Synthesis output:

The operations list is :

[ma-0-f-2-a-1,mac,mul_ac,edges,xt-2-a-1,1,4,mac1-1]
[ma-0-f-2-a-2,mac,mul_ac,edges,xt-2-a-2,1,4,mac2-1]
[n-2-a-1,mac,trunc,[xt-2-a-1,c-2],x-2-a-1,5,5,mac1]
[n-2-a-2,mac,trunc,[xt-2-a-2,c-2],x-2-a-2,5,5,mac2]
[ma-0-f-3-a-1,mac,mul_ac,edges,xt-3-a-1,6,9,mac1-1]
[n-3-a-1,mac,trunc,[xt-3-a-1,c-3],x-3-a-1,10,10,mac1]

The registers list is :

[xreg-1,[x-1-a-1,1],[ma-0-f-2-a-1,1],0,1,bus-1,rf-1-r2]
[xreg-2,[x-1-a-2,1],[ma-0-f-2-a-1,3],0,2,bus-1,rf-1-r3]
[xreg-3,[x-1-a-1,1],[ma-0-f-2-a-2,5],0,1,bus-1,rf-1-r2]
[xreg-4,[x-1-a-2,1],[ma-0-f-2-a-2,7],0,2,bus-1,rf-1-r3]
[xreg-5,[x-2-a-1,1],[ma-0-f-3-a-1,9],5,6,bus-1,rf-1-r1]
[xreg-6,[x-2-a-2,1],[ma-0-f-3-a-1,11],5,7,bus-2,rf-2-r4]
[xreg-7,[x-3-a-1,1],[out-3-a-1,1],10,0,bus-2,rf-2-r4]
[xreg-8,[xout-3-a-1,1],out,0,0,bus-2,rf-2-r5]
[wreg-1,[w-1-c-1-2-a-1,1],[ma-0-f-2-a-1,2],0,1,bus-2,rf-2-wr1]
[wreg-2,[w-1-a-2-2-a-1,1],[ma-0-f-2-a-1,4],0,2,bus-2,rf-2-wr2]
[wreg-3,[w-1-a-1-2-a-2,1],[ma-0-f-2-a-2,6],0,1,bus-3,rf-3-wr3]
[wreg-4,[w-1-a-2-2-a-2,1],[ma-0-f-2-a-2,8],0,2,bus-3,rf-3-wr4]
[wreg-5,[w-2-a-1-3-a-1,1],[ma-0-f-3-a-1,10],0,6,bus-3,rf-3-wr5]
[wreg-6,[w-2-a-2-3-a-1,1],[ma-0-f-3-a-1,12],0,7,bus-3,rf-3-wr6]
[xtreg-1,[xt-2-a-1,1],[n-2-a-1,1],4,5,bus-3,rf-3-r6]
[xtreg-2,[xt-2-a-2,1],[n-2-a-2,1],4,5,bus-1,rf-1-r1]
[xtreg-3,[xt-3-a-1,1],[n-3-a-1,1],9,10,bus-2,rf-2-r4]

[creg-1,[c-2,1],[n-2-a-1,2],0,5,bus-2,rf-2-cr1]

[creg-2,[c-2,1],[n-2-a-2,2],0,5,bus-2,rf-2-cr2]

[creg-3,[c-3,1],[n-3-a-1,2],0,10,bus-3,rf-3-cr3]

The hardware list is :

hardware(fus([[mac,2,[mac1,mac2]],[io,1,[io1]],[output,1,[out1]]]),buse

ss([bus_1-all,bus_2-all,bus_3-all]))

The bus reservation input list is :

[0,[bus_1-all,bus_2-all,bus_3-all]]

[1,[x-1-a-1,w-1-a-1-2-a-1,w-1-a-1-2-a-2]]

[2,[x-1-a-2,w-1-a-2-2-a-1,w-1-a-2-2-a-2]]

[3,[_288718,_288720,_288722]]

[4,[_288740,_288742,_288744]]

[5,[xt-2-a-1,c-2,xt-2-a-2]]

[6,[x-2-a-1,w-2-a-1-3-a-1,_298637]]

[7,[x-2-a-2,w-2-a-2-3-a-1,_300224]]

[8,[_302266,_302268,_302270]]

[9,[_302298,_302300,_302302]]

[10,[xt-3-a-1,c-3,_302336]]

The bus reservation output list is :

[0,[bus_1-all,bus_2-all,bus_3-all]]

[1,[_285991,_285993,_285995]]

[2,[_286009,_286011,_286013]]

[3,[_286029,_286031,_286033]]

[4,[xt-2-a-1,xt-2-a-2,_286055]]

[5,[x-2-a-1,x-2-a-2,_289067]]

[6,[_300580,_300582,_300584]]

[7,[_300608,_300610,_300612]]

[8,[_300638,_300640,_300642]]

[9,[xt-3-a-1,_300672,_300674]]

[10,[x-3-a-1,_302673,_302675]]

The FU list is :

mac

mac1

[lreg-1,w-1-a-1-2-a-1,w-1-a-2-2-a-1,free,w-2-a-1-3-a-1,w-2-a-2-3-a-1,free|_300882]

in

[1,[xreg-1,wreg-1]]

[2,[xreg-2,wreg-2]]

[5,[xtreg-1,creg-1]]

[6,[xreg-5,wreg-5]]

[7,[xreg-6,wreg-6]]

[10,[xtreg-3,creg-3]]

out

[4,xt-2-a-1]

[5,x-2-a-1]

[9,xt-3-a-1]

[10,x-3-a-1]

[1,2,3,6,7,8,11|_302617]

mac

mac2

[lreg-1,w-1-a-1-2-a-2,w-1-a-2-2-a-2,free/_287493]

in

[1,[xreg-3,wreg-3]]
[2,[xreg-4,wreg-4]]
[5,[xtreg-2,creg-2]]
out

[4,xt-2-a-2]
[5,x-2-a-2]
[1,2,3,6/_290300]

3.2. Corresponding VHDL microcode

```
use work.Types.all ;

entity MICROCODE is

        port (clk : in bit;
                        cycle: in integer;
                        RD: out microcode_rd;
                        WR: out microcode_wr);
        end MICROCODE;


architecture MICROCODE_arch of MICROCODE is

-- read word =[...[reg-file-id, FU-id, Input-latch],reg-file-index,...]
-- write word=[...[reg-file-id,FU-id,Input-latch]...]
begin
process(clk)
begin
if clk='1' then
case cycle + 1 is
```

```
when 1 => RD <=
((RF1,MAC12,1),2,(RF2,MAC1,2),1,(RF3,MAC2,2),1,MULT_AC1,MULT_AC1);
when 2 => RD <=
((RF1,MAC12,1),1,(RF2,MAC1,2),2,(RF3,MAC2,2),2,MULT_AC2,MULT_AC2);
when 3 => RD <=
((NONE,NONE,1),1,(NONE,NONE,1),1,(NONE,NONE,1),1,NONE,NONE);
when 4 => RD <=
((NONE,NONE,1),1,(NONE,NONE,1),1,(NONE,NONE,1),1,NONE,NONE);
when 5 => RD <=
((RF1,MAC1,1),3,(RF2,MAC12,2),3,(RF3,MAC2,1),3,THRESH,THRESH);
when 6 => RD <=
((RF1,MAC1,1),3,(NONE,NONE,1),1,(RF3,MAC1,2),4,MULT_AC1,NONE);
when 7 => RD <=
((NONE,NONE,1),1,(RF2,MAC1,1),4,(RF3,MAC1,2),5,MULT_AC2,NONE);
when 8 => RD <=
((NONE,NONE,1),1,(NONE,NONE,1),1,(NONE,NONE,1),1,NONE,NONE);
when 9 => RD <=
((NONE,NONE,1),1,(NONE,NONE,1),1,(NONE,NONE,1),1,NONE,NONE);
when 10 => RD <=
((NONE,NONE,1),1,(RF2,MAC1,1),4,(RF3,MAC1,2),6,THRESH,NONE);
when others => null;
end case;

elsif clk='0' then
case cycle is
when 1 => WR <=
((NONE,NONE),4,(NONE,NONE),5,(NONE,NONE),5);
when 2 => WR <=
((NONE,NONE),4,(NONE,NONE),5,(NONE,NONE),5);
when 3 => WR <=
((NONE,NONE),4,(NONE,NONE),5,(NONE,NONE),5);
when 4 => WR <=
```

```
((RF1,MAC2),3,(NONE,NONE),5,(RF3,MAC1),3);
when 5 => WR <=
((RF1,MAC1),3,(RF2,MAC2),4,(NONE,NONE),1);
when 6 => WR <=
((NONE,NONE),4,(NONE,NONE),5,(NONE,NONE),5);
when 7 => WR <=
((NONE,NONE),4,(NONE,NONE),5,(NONE,NONE),5);
when 8 => WR <=
((NONE,NONE),4,(NONE,NONE),5,(NONE,NONE),5);
when 9 => WR <=
((NONE,NONE),4,(RF2,MAC1),4,(NONE,NONE),5);
when 10 => WR <=
((NONE,NONE),4,(RF2,MAC1),4,(NONE,NONE),5);
when others => null;
end case;
end if;
end process;
end MICROCODE_arch;
```

3.3. Simulation results

The following are the simulation results for the XOR for the inputs, '0 0', '0 1' and '1 1' obtained through the Valid Software VHDL interface. The case for '10' inputs is shown in chapter 7.

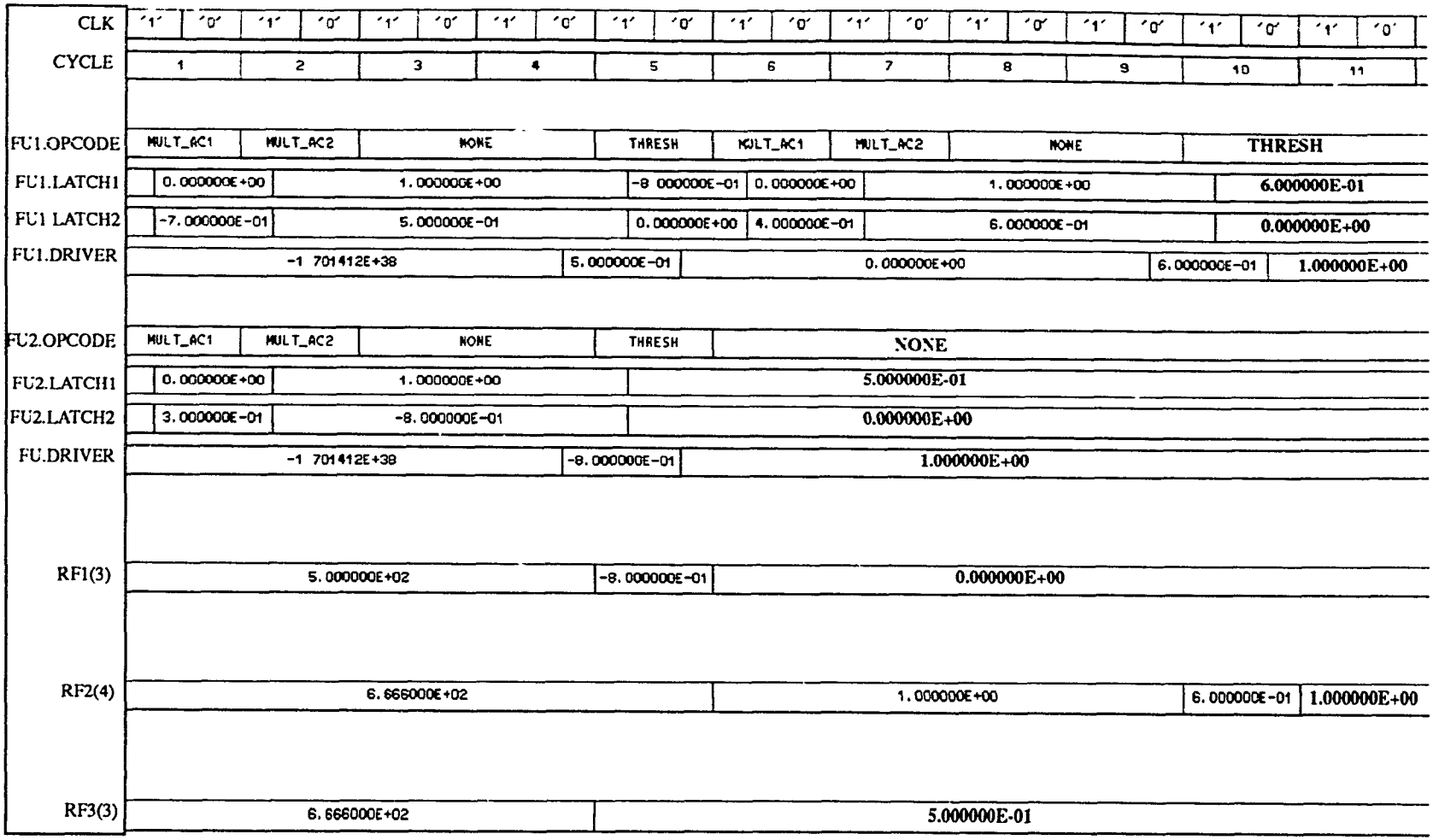


Figure III.1: VHDL timing diagram for XOR example for '0 1' inputs

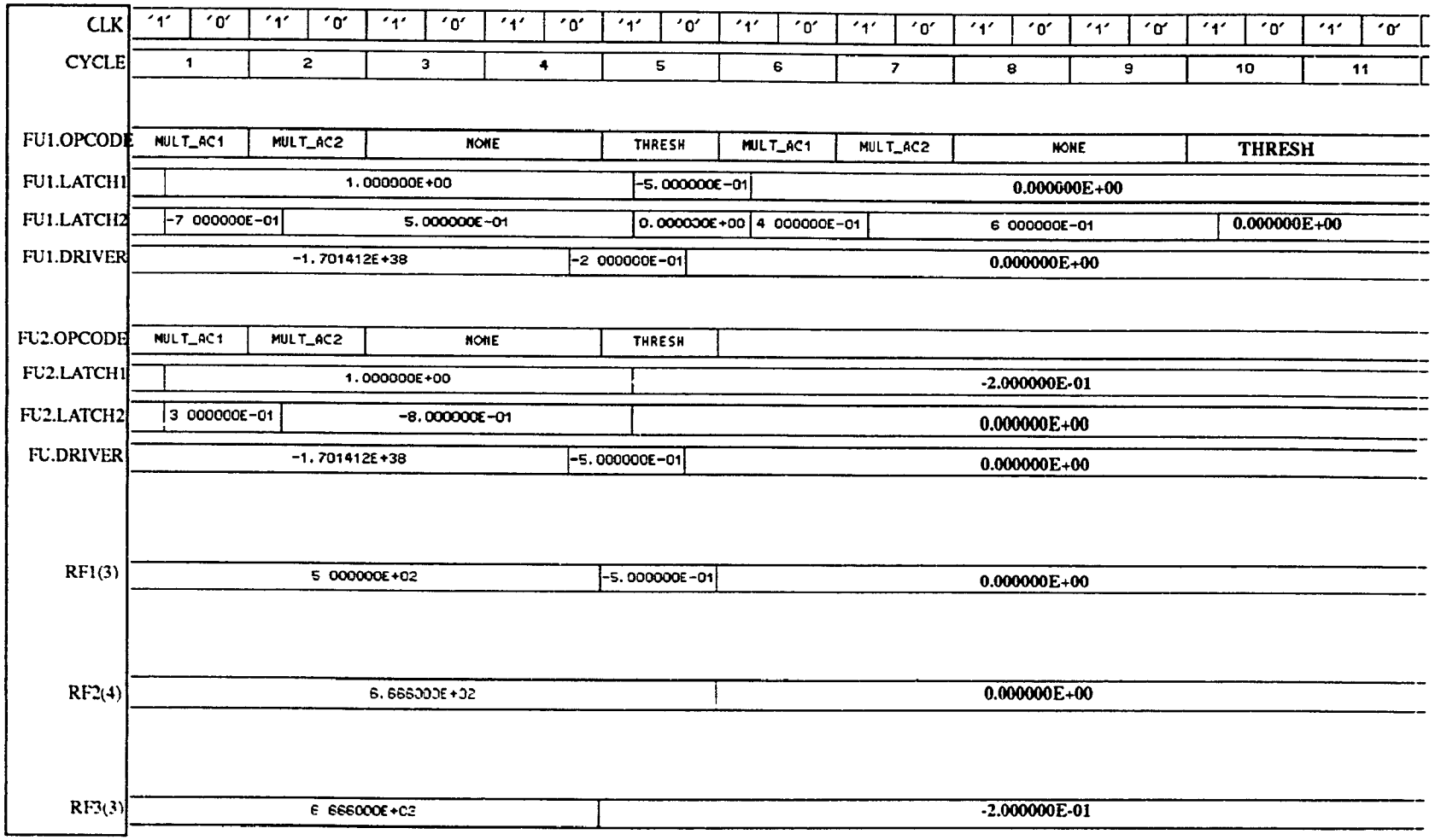


Figure III.2: VHDL timing diagram for XOR example for '1 1' inputs

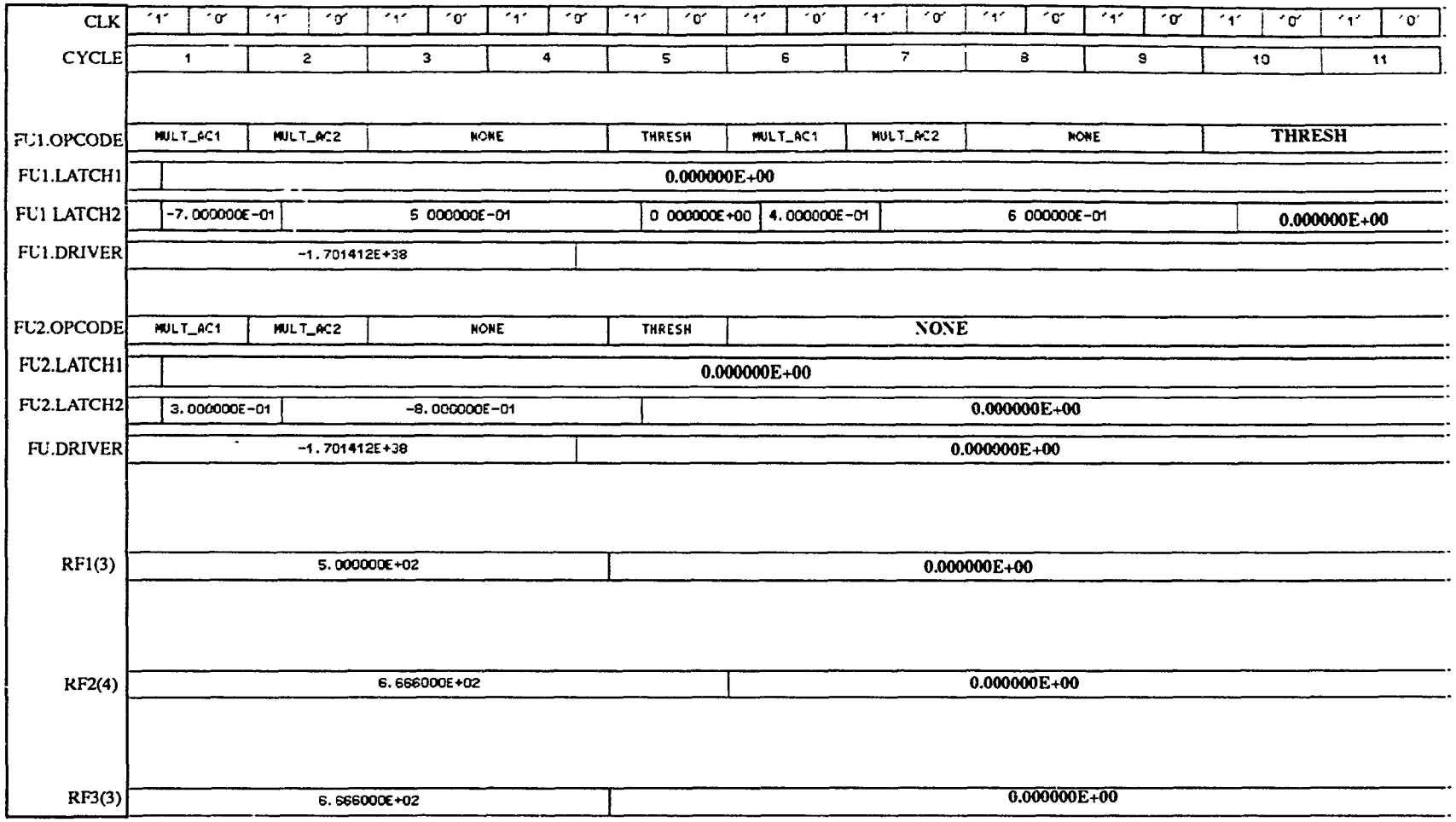


Figure III.3: VHDL timing diagram for XOR example for '0 0' inputs

Publications Resulting from this Research

Elie Torbey and Baher Haroun, *"Architectural Synthesis for Digital Neural Networks,"* **International Joint Conference on Neural Networks**, Vol. II, pp. 601-606, Baltimore, June 1992.

Baher Haroun and Elie Torbey, *"Synthesis of Multiple Bus/ Functional Unit Architectures Implementing Neural Networks,"* to appear in the proceedings of the **International Conference on Computer Design**, Cambridge, Massachusetts, October 1992.

Elie Torbey, Baher Haroun, Asif Zaidi and Rajni Patel, *"Synthesis and Implementation of Digital VLSI Neural Networks,"* to appear in the proceedings of the **Canadian Conference on Very Large Scale Integration**, 1992.