RAPID-PROTOTYPING OF ARTIFICIAL NEURAL NETWORKS

BY

ROGER K.W. NG

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg, Canada

RAPID-PROTOTYPING OF ARTIFICIAL NEURAL NETWORKS

BY

ROGER K.W. NG

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

ABSTRACT

This thesis explores Field Programmable Gate Array (FPGA) implementations of artificial neural networks employing pulse-code arithmetic. Pulse-code arithmetic uses values encoded as probabilitic pulse streams. Artificial neural networks employing pulse-code arithmetic require only simple digital logic gates to perform multiplication and addition which are the essential operations for these networks. As such, pulse-code techniques offer considerable potential to construct very large neural networks using FPGA technology. The implementation results presented in this thesis show that each neuron and synapse element use an average of 23 CLBs on Xilinx XC4000 series FPGAs for the XOR problem. One of the advantages of using FPGAs for implementing neural networks is that they allow the overall network to be easily modified or replaced, by simply downloading new circuity. In addition, this thesis describes a top-down design flow methodology from abstracted simulation through to implementation in Xilinx FPGAs. The use of top-down design and FPGA technologies shorten the overall development cycle. In addition, as these networks are extremely compact there is the potential for experimentation during prototyping.

ACKNOWLEDGEMENTS

When everything in your life seems to be leading you towards some unknown yet strangely familiar conclusion and its not until you reach the end that you realize that coincidence has been conspiring to guide you into this particular window of the eternal now. My three-year graduate studies was sort of like that. So thanks to all the guides and earth angels who generously contributed and supported throughout my graduate studies.

I would like especially thank my advisor, Professor Robert McLeod for his advice, encouragement, and assistance throughout this thesis.

I also would like to acknowledge all the members of the VLSI Laboratory at the University of Manitoba, specifically Hart Poskar, Dean McNeill, Richard Wieler and Ken Ferens. In addition, I would like to acknowledge the support of Computer Services at University of Manitoba,

Support provided by the Natural Sciences and Engineering Research Council of Canada, the Federal Government's Centers of Excellence Micronet Program and the Canadian Microelectronics Corporation is highly appreciated.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Almost everything in the field of neural networks has been done by simulating the networks on serial computers. There has been comparatively little study of hardware implementations. General purpose computers are not optimized for neural network calculations; they require specialized hardware in order to utilize the inherent parallelism. The alternative approach is to build special hardware for neural networks on a single chip or multi-chip system. A neural network architecture can be implemented as an integrated circuit using analog, digital, or mixed analog/digital structures. The analog circuitry permits high density implementation as the multiplication is based on modified Gilbert multipliers [1] and summation on Kirchhoff's current law. However, analog hardware does not produce high accuracy arithmetic and the storage of analog weight values required for the synapse is difficult. On the other hand, the digital hardware can perform arithmetic operations with a high degree of accuracy and the storage of the weight values is easy in the digital form. Also, digital hardware can take advantage of some of the benefits of current VLSI technology, such as well understood and advanced design techniques, as well as prototying in *Field Programmable Gate Array* FPGA technologies. However one of the major constraints of digital implementations of neural networks is the amount of circuity required to perform the multiplication. This problem is especially acute in high speed digital designs, where parallel multipliers are extremely expensive in terms of circuity. Adopting an equivalent bit serial architecture significantly reduces this complexity, but still tends to result in large and complex designs. In addition a single multiplier would consume a significant proportion of a current state of the art FPGA, thus making the use of such devices impractical for this approach.

This thesis describes an alternative neural network architecture which may be implemented using standard VLSI technology, but also maps extremely efficiently to FPGAs such as those of Xilinx [2]. The central idea is to represent the real-valued signals passing between neurons using encoded binary pulse streams. Pulse streams arithmetic requires only simple digital logic gates to perform multiplication and addition. The main advantage of such an approach is structural simplicity of the artificial synapse and neuron, comparable to analog implementations, thus allowing very efficient space usage of the fine grained FPGAs.

The work presented in this thesis examines pulse-code neural network implementations on FPGAs using top-down design methodology. In the remainder of this chapter background material of neural networks is presented which will serve to familiarize the reader with some of the general concepts. This will help establish a common reference from which to base the discussions in the later chapters.

1.1 Artificial Neural Network

Modern neural network theories can be traced backed to ideas first introduced in the 1940s and 1950s. In 1943, McCulloch and Pitts proposed a simple model of *neuron* operation [3]. This model attracted much interest because of its simplicity. In the late 1950s, Rosenblatt developed networks that could learn to recognize simple patterns [4]. The *perceptron*, as it was called, could decide whether an input belonged to one of two classes. A single neuron would compute the weighted sum of binary-type inputs, subtract a *threshold*, and pass the result through a non-linear hard limiting threshold that classified the input. In 1969, Minsky and Papert [5] showed that a small class of perceptrons could not perform certain tasks in pattern recognition. The simple example is the *exclusive or* (XOR) problem: a single output neuron is required to turn on if one or the other of two input lines is on, but not when neither or both inputs are on. They believed that structures with more layers of neurons could solve the problem, but they could not find a learning rule to train a multi-layer network. With this roadblock, researchers left the neural network paradigm for almost 20 years. It was not until 1986, when Rumelhart, Hinton and Williams introduced a new learning algorithm, known as *backpropagation* [6] to the problem of the networks discussed in Perceptrons [5] that neural networks regained their popularity.

Neural networks are usually characterized by the way in which neurons are interconnected. There are two major classes of neural network topologies: *multi-layer feedforward* networks and *feedback* networks. Feedback networks are beyond the scope of this thesis which will focus on multi-layer feedforward networks only. The general form of the multi-layer feedforward network consists of an input layer, one or more hidden layers, and an output layer of *neurons* (see Figure 1.1).

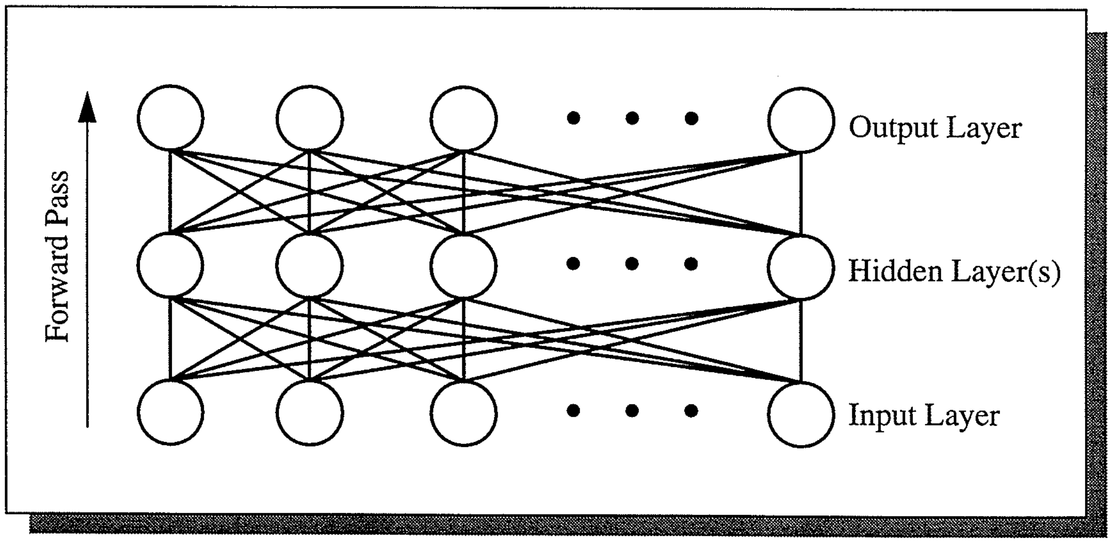


Figure 1.1: Multi-layer feedforward network.

Complete bipartite graphs are constructed between each adjacent layer of neurons using *weighted connections.* Data is presented to the input layer, each is multiplied by a weight and summed at neurons of the connecting layer. These weighted sums are then passed through a *reversible nonlinear transfer function* forming the input to the following layer. This procedure is repeated until reaching the output layer thereby completing a *forward pass.*

Without a program of instructions a computer is a useless machine. The program usually instructs the computer to perform specific tasks on a set of input data to create some sort of output. Therefore, the program is an essential part in a computer environment. Neural networks are not programmed in the conventional sense, they are *taught.* Teaching a neural network cognitive knowledge is basically a modification of the synaptic weights according to some learning algorithm or rule. Therefore, the knowledge or "program" of a neural network is in the weights. There are two main types of learning rule: *supervised* learning and *unsupervised* learning. In supervised learning, an example set of input/output pairs is necessary, and the error between the actual response and the target response is used to correct or modify the network. In contrast to supervised learning, unsupervised learning is not given any information about whether its outputs are right or wrong. Instead, the network must decide what characteristics of the training set are relevant, and modify the weights in order to extract those features. This thesis will focus on the supervised learning neural network. A popular supervised learning algorithm is *the backpropagation learning algorithm*, which was introduced by Rumelhart, Hinton and Williams in 1986 [6].

The backpropagation learning algorithm involves the presentation of a *training set* of input/output pattern pairs. The objective is to find a set of weights that ensures that the output produced by the network is the same as, or close to, the target output pattern for each of the input patterns. During the *backward pass* (learning phase), the actual output is compared to the target output and an error vector is created. These errors are then *backpropagated* through the network modifying the connection weights according to an *iterative gradient descent algorithm.* After many iterations of the training set the connection weights settle to a local minimum of the output error over the training set. Better minima may be found repeated training with randomly selected initial connection weights.

2.2 Summary

This chapter has provided a quick overview of the advantages and disadvantages of analog and digital implementations of neural networks. In order to implement these networks in FPGAs, the area of the circuity is the most important criteria. The use of pulse stream arithmetic was proposed as it allows low area implementation of the hardware required to perform the arithmetic for neural networks. A brief history of neural networks, neural network topologies and learning rules was overviewed. Multi-layer feedforward networks and the learning algorithm known as Backpropagation was also discussed.

2.3 Organization of the Thesis

The next chapter discusses the implementation of pulse-coded neural networks. It begins with the fundamentals of pulse stream arithmetic followed by a discussion of applying pulse stream arithmetic to neural networks. Chapter 3 presents the software simulation of these networks. Chapter 4 describes the design process of the pulse-coded neural networks onto Xilinx FPGAs. Finally conclusions are drawn and proposals for future work is presented.

Chapter 2

Pulse-Code Neural Networks

Pulse-code neural networks use pulse streams to perform the network calculations. The idea of using pulse streams to communicate information between neurons is motivated from biological models, although biological pulse streams are much more complex than the simple pulse representation considered in this thesis. The main motivation here of applying pulse-code arithmetic to neural networks is the ability to implement high density arithmetic operations using digital circuitry. Specifically, pulse-code representations are able to use simple digital hardware to perform addition and multiplication, which are the two most important operations in a neural network. Also, pulse-code implementations offer the advantages of both analog and digital computation. Like analog, pulse representation requires only one line to carry the values, and the size of the hardware (digital gates) needed to perform arithmetic computations is comparable to analog hardware. Like digital, the design methods and implementations are well established.

In this chapter the fundamentals of pulse-code arithmetic are presented and this discussion leads to application of pulse-code arithmetic for neural networks.

2.1 Fundamentals of Pulse-Code Arithmetic

The fundamental idea of pulse-code arithmetic is to use probabilities to carry information [7, 8]. Here the probability p is defined experimentally by considering the frequency of the occurrence of an event (pulse in a time slot). A small number of time slots results in an erroneous assessment of the probability and the number which it represents. In the limiting case of an infinite number of time slots: if there are n pulses in N slots for a given time, and if n/N tends towards a limit as $N \to \infty$ we set

$$p = \lim_{N \to \infty} \frac{n}{N} \tag{2.1}$$

Figure 2.1 shows a synchronous random pulse sequence. At the top of the Figure 2.1, 3 pulses are in 10 time slots, leading to the conclusion that the number transmitted is 0.3. At the bottom, 3 pulses are arranged in different time slots, leading again to 0.3. The order of the pulses in the time slots does not affect the outcome of the representation. The probability can be transformed into some physical quantity by an appropriate mapping. This thesis only considers a linear mapping, although it should be noted that nonlinear mappings exist which permit computations with numbers in an infinite range with logarithmic error characteristics [8]. There are two kinds of linear mapping: *unipolar mapping* and *bipolar mapping*.

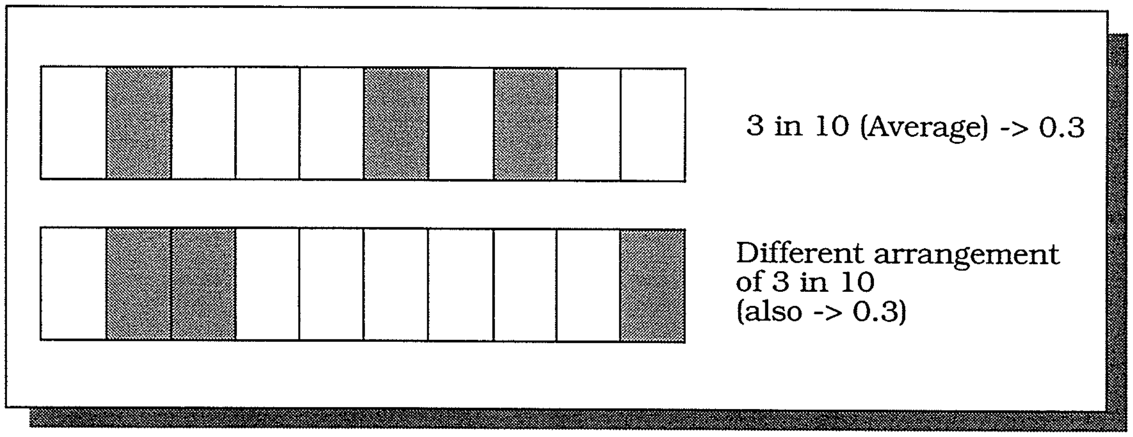


Figure 2.1: Random pulse sequence

Unipolar linear mapping is used as the implementation method in this thesis. In unipolar mapping, the values are encoded between 0 and 1. An example of unipolar pulse representation is shown in Figure 2.2. The value of the unipolar pulse stream is represented by number of pulses, *n*, being ON within the time interval divided by length of the time interval, *N*, or

$$UnipolarValue = \frac{n}{N} \tag{2.2}$$

A unipolar pulse stream with N bits can represent N+1 unique values. For example, a 10-bit pulse stream can represent 11 values from 0.0 to 1.0, in increments of 0.1. The resolution of value depends on the length of the pulse stream. The longer the pulse stream, the higher the resolution that can be achieved.

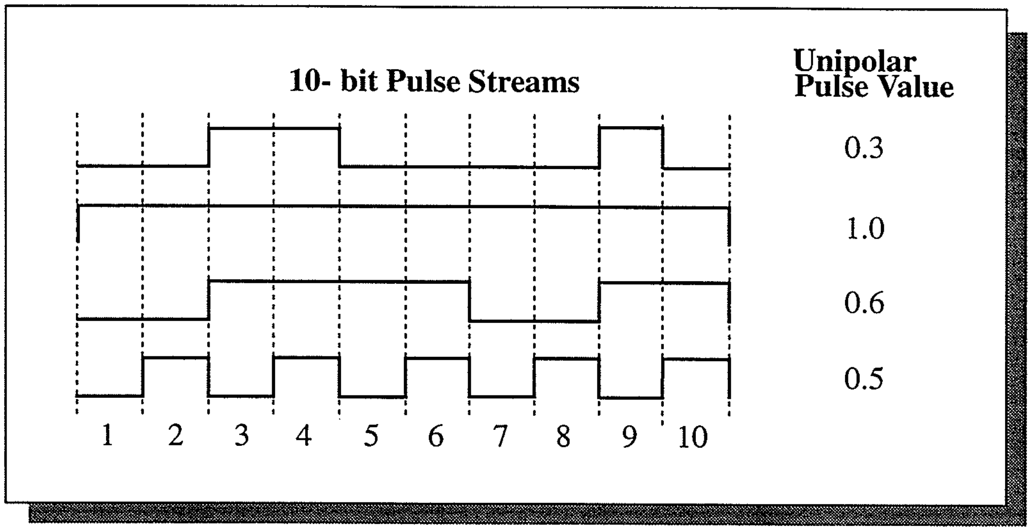


Figure 2.2: Unipolar pulse stream representation.

2.1.1 Pulse Stream Addition

The addition of two unipolar pulse can be performed by using an OR gate. However, OR gate addition does not perform exact addition with pulse streams because of limitations imposed by the representation of the pulse streams, furthermore it cannot handle a sum greater than 1. The output of the OR gate is given by

$$\begin{aligned} A \cup B &= A\bar{B} + \bar{A}B + AB \\ &= A + B - AB \end{aligned} \qquad \texttt{Define:}\ \bar{A} \equiv 1 - A$$

Thus for A<<1 and B<<1 the AB term is small and the output of the OR gate is approximately A + B. For large A and B, the output of the OR gate saturates to 1. This result of a saturating nonlinearity will be useful to the implementation of neural networks. The output for an OR gate with *n* inputs is given by

$$Output = 1 - \prod_{n} (1 - i_n) \tag{2.3}$$

The n-input OR gate can be easily implemented in hardware by using wired-OR logic. Figure 2.3 shows the output probability of the OR gate addition. As the number of the inputs increases, the output of the OR gate addition saturates for a greater range of inputs. Also, as the value of average input increases, the output of the OR gate addition saturates. Therefore, it is desirable to keep the fan-in of the inputs as well as the value of the input small.

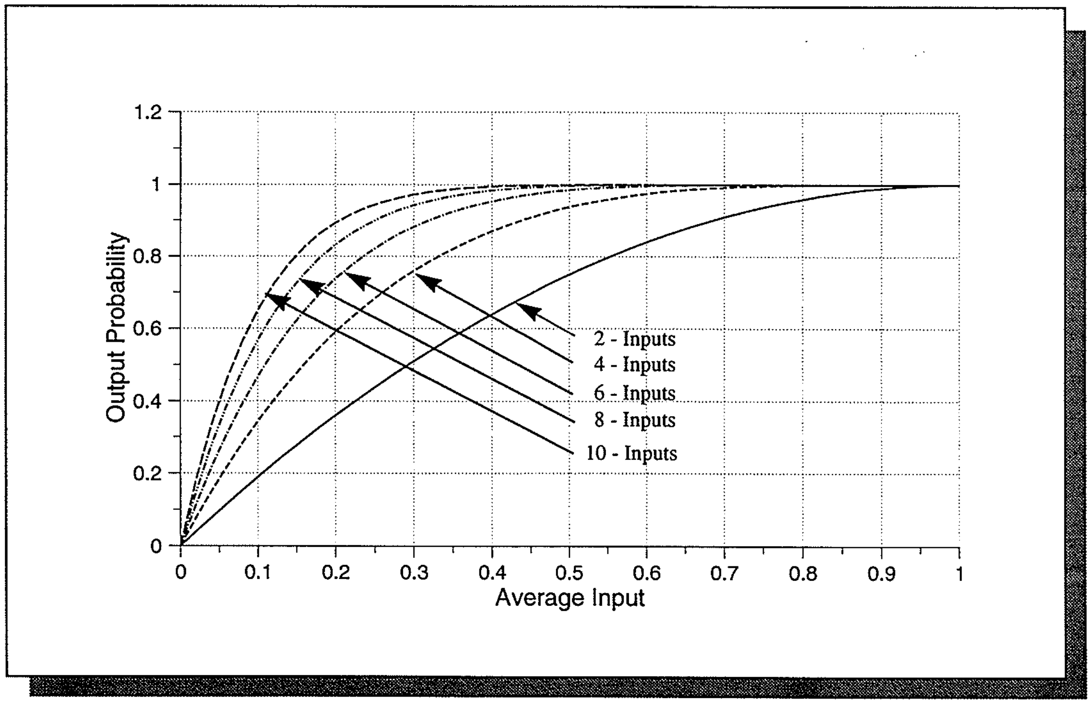


Figure 2.3: OR gate addition.

2.1.2 Pulse Stream Multiplication

To multiply two unipolar pulse streams, an AND gate may be used if two pulse sequence are statistically uncorrelated [8]. Since the unipolar value is always less than or equal to one, the product of two numbers is guaranteed to be at most one and the result will not saturate as it did with OR-gate addition. Assuming the pulse sequences A and B are statistically uncorrelated, the output sequence of an AND gate multiplication is given by

$$A \cap B = AB \tag{2.4}$$

Figure 2.4 shows an example of the OR-gate addition and AND-gate multiplication.

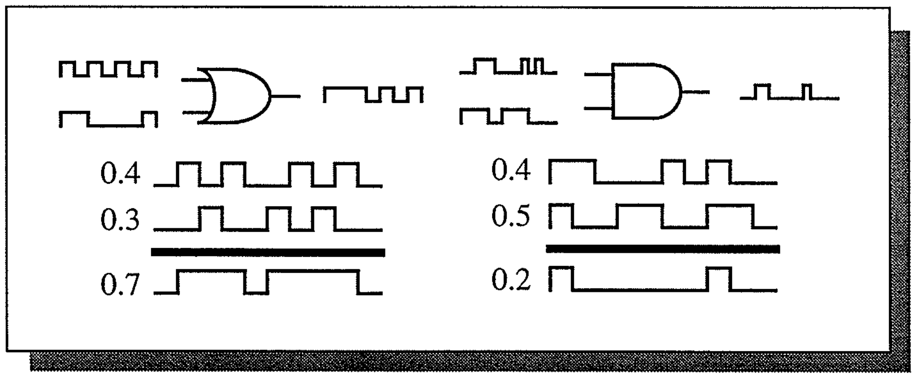


Figure 2.4: Example of the OR-gate addition and AND-gate multiplication.

2.1.3 Pulse Stream Generation

An essential component of pulse-code arithmetic is the generation of the pulse streams for use in the arithmetic operations. As mentioned earlier, the information is carried by the probability of occurrence of a ON logic level within a time slot. Each logic level is generated from a random variable, and the statistically independent results form a pulse sequence whose average pulse rate is determined by the variable to be represented. It should be noted that the validity of the pulse streams arithmetic relies heavily on the assumed property of statistical independence between operating variables. Hence, of vital importance is generators for the provision of independent uniformly distributed random numbers.

In general, a random pulse stream is generated with a uniform random number generator and a digital comparator. Figure 2.5 shows a block diagram of a rate multiplier to generate weighted pulse streams. The following procedure can be used to produce a random pulse stream with probability P(ON)=W:

- Generate a random number R, such that $0 \le R \le 1$.
- If W>R, output a 1 else output a 0.

In digital hardware R and W are usually represented as binary integers. If the maximum possible weight value is M, and the value stored in the weight register is W, then the probability of a pulse should be P(ON) = W/M.

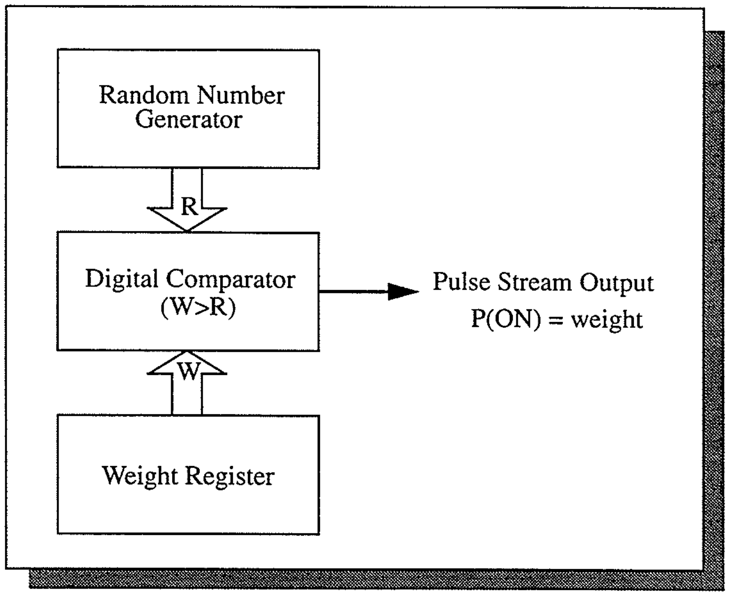


Figure 2.5: Block diagram of a pulse stream generation

A common technique to generate a pseudorandom number in digital hardware is using a linear feedback shift register (LFSR)[9]. Another method to produce a digital random number is to employ a particular configuration of a one-dimensional Cellular Automata (CA) array[10]. Hortensius[11] has shown that certain arrangements of CAs possess maximal length sequences with superior random number properties compared to the LFSR. A CA is a set of registers whose next state is governed by nearest neighbour connections. A CA can yield a maximal-length binary sequence from each site (i.e. $2^n - 1$), like the maximal-length LFSR by combining rules 90

$$a_i(t+1) = a_{i-1}(t) \oplus a_{i+1}(t)$$

and rules 150,

$$a_i(t+1) = a_{i-1}(t) \oplus a_i(t) \oplus a_{i+1}(t)$$

where $a_i(t)$ is the value of the register at position i at time t.

The ordering of the rules for construction of a maximal-length binary sequence is irregular, with complexity similar to that involved in determining the polynomial for a maximal-length LFSR. Table 2.1 gives a sample of possible constructions for producing CAs with maximal cycle length up to length 15. Here, "1" refers to CA rule 150 and "0" refers to CA rule 90. Hence, a length-5 maximal-length CA would be constructed by using rules 90 and 150 in the following order: 150, 150, 90, 90 150.

Table 2.1: Construction of CA with maximal cycle length

Length n	Construction	Cycle Length
5	11001	31
6	010101	63
7	1101010	127
8	11010101	255
9	110010101	511
10	0101010101	1023
11	11010101010	2047
12	010101010101	4095
13	1100101010100	8191
14	01111101111110	16383

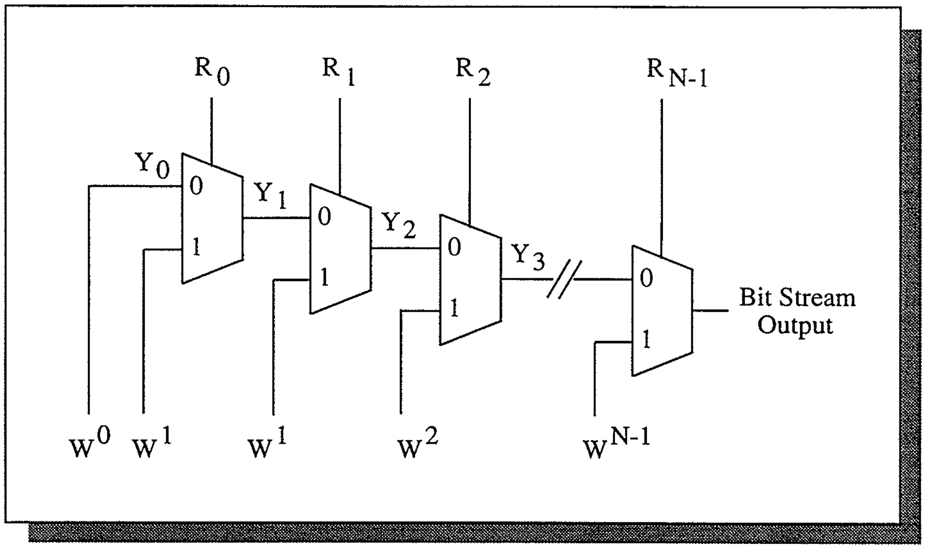


Figure 2.6: Rate multiplier schematic

A rate multiplier can be used to compare a binary set of weights and a set of random bit streams with P(ON)=0.5, and produces a weighted bit stream with P(ON)=$w/2^{N-1}$. The rate multiplier as shown in Figure 2.6 originated out of research in VLSI pseudo-random test pattern generation [12].

2.2 Pulse-Code Neural Network

The idea of using pulse streams has been tried by Tomberg and his co-workers, who published two papers on neural networks using pulse-density modulation [13, 14]. The implementation was a Hopfield type fully connected neural network architecture based on bipolar pulse density modulation. Tomlinson [15] and Dickson [16] also studied in-situ learning neural network using unipolar pulse streams. The work presented in this section is based on the research from Tomlinson and Dickson, extending the design methodology for these implementations.

2.2.1 Pulse-code Feedforward Neural Network

The basic computational operations required in feedforward neural networks are multiplication, summation, and a non-linearity function. Each neuron computes a weighted sum of its inputs from other neurons, and passes this summation through a nonlinear function to produce an output. This output forms the input to the following layer. Tomlinson [15] has proposed a new neural activation function where the summation and the nonlinear activation are performed simultaneously using the OR logic. As mentioned earlier the OR gate addition saturates to 1 with either an increase in the number of inputs or an increase in the value of those inputs. The saturating effect of the OR gate addition requires no extra hardware to implement a nonlinear activation function. Also, the logical OR can be easily implemented in hardware using wired-OR logic. The multiplication of the weight (w_{ij}) and the input (o_i) are performed with a simple AND gate as previously described, assuming that these pulse sequences are statistically uncorrelated.

Let n_j be the probability of a pulse occurrence in the output sequence of an n-input OR gate. The inputs of an OR gate are the product of w_{ij} and o_i produced from the AND gates. This is represented mathematically by the following equation:

$$n_j = 1 - \prod_{i=1}^{n} (1 - w_{ij} o_i) \tag{2.5}$$

Since the unipolar nature of the pulse stream representation does not support negative values, each synaptic weight is separated into two distinct nets: the *excitatory* and the *inhibitory nets*. Therefore, there are two dedicated wired-OR lines per neuron ANDed together to form the activation function and the net inputs variables are defined as

$$n_j^+ = 1 - \prod_{w_{ij} > 0} (1 - w_{ij} o_i) \tag{2.6}$$

$$n_j^- = 1 - \prod_{w_{ij} < 0} (1 + w_{ij} o_i) \tag{2.7}$$

Each neuron j combines the excitatory net input n_j^+ and the inhibitory net input n_j^- to determine the neuron output o_j. Since there is no means to perform subtraction in pulse-code arithmetic, the net output of the neuron is not simply $o_j = n_j^+ - n_j^-$. Moreover negative and positive nets would require the accommodation of negative neuron outputs. If the excitatory net input n_j^+ and inhibitory net input n_j^- are statistically uncorrelated, the probability of output pulse occurrence o_j is

$$o_j = P\left(\left(n_j^+ = 1\right) \wedge \left(n_j^- = 0\right)\right)$$
$$o_j = n_j^+\left(1 - n_j^-\right) \tag{2.8}$$

While the mathematics implies that a weight can have a positive (w_{ij}^+) components and a negative (w_{ij}^-) components, it is not necessary to accommodate both simultaneously. Therefore, it only requires one register to store the weight value and one bit to indicate the sign of the weight. The hardware required for this computation is shown in Figure 2.7.

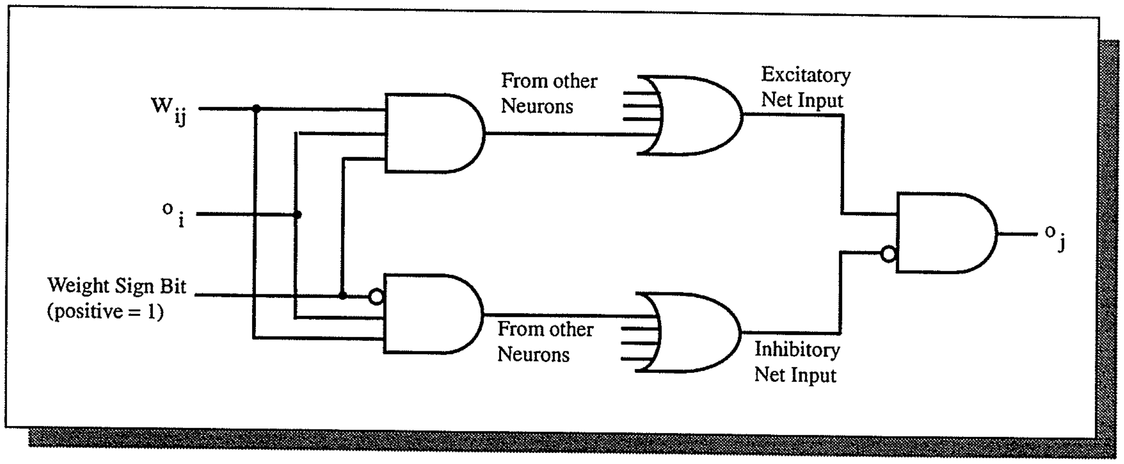


Figure 2.7: Negative and positive weight.

2.2.2 Training Pulse-Code Neural Networks

Because a pulse-code neural network uses a non-traditional activation function, it is instructive to consider a modified learning algorithm where this activation function is incorporated into a popular learning algorithm. Equations 2.5 and 2.8 are continuous and differentiable, indicating that the backpropagation learning algorithm can be used for training. Backpropagation is an iterative technique that performs a gradient descent, typically over a sum squared error measure:

$$E = \frac{1}{2}\sum_j (t_j - o_j)^2 \tag{2.9}$$

where t_j is the desired output, and o_j is the actual output of the neuron j. The weights should be modified along the negative gradient of this error with respect to each weight:

$$\Delta w_{ij} \propto -\frac{\partial E}{\partial w_{ij}} \tag{2.10}$$

The goal of the backpropagation learning algorithm is to reduce the total error by adjusting the weights. Since the output of the neurons are computed from the excitatory nets and inhibitory nets, the derivative must be considered separately for positive and negative weights. Using the chain rule, the positive and negative equations governing the change of weights is as follows

$$\Delta w_{ij}^{+} = -\frac{\partial E}{\partial w_{ij}^{+}} = -\frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial n_j^{+}}\frac{\partial n_j^{+}}{\partial w_{ij}^{+}} \tag{2.11}$$

$$\Delta w^{-}_{ij} = -\frac{\partial E}{\partial w^{-}_{ij}} = -\frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial n^{-}_j}\frac{\partial n^{-}_j}{\partial w^{-}_{ij}} \tag{2.12}$$

Let us define,

$$\varepsilon_j = -\frac{\partial E}{\partial o_j} \tag{2.13}$$

The result of the positive and negative equations become:

$$\Delta w^{+}_{ij} = -\frac{\partial E}{\partial w^{+}_{ij}} = \varepsilon_j\left(1 - n^{-}_j\right)\frac{o_i\left(1 - n^{+}_j\right)}{1 - w^{+}_{ij}o_i} \tag{2.14}$$

$$\Delta w^{-}_{ij} = -\frac{\partial E}{\partial w^{-}_{ij}} = \varepsilon_j n^{+}_j \frac{o_i\left(1 - n^{-}_j\right)}{1 + w^{-}_{ij}o_i} \tag{2.15}$$

Equation 2.9 shows that the error at the output neurons is simply the difference between the training data and the network output:

$$\varepsilon_j = -\frac{\partial E}{\partial o_j} = t_j - o_j \tag{2.16}$$

For the hidden layers, the error is propagated back through the network. Each of the K output neurons is connected to hidden neuron j, and will contribute to this error. This error has two components, from the excitatory and inhibitory net inputs to each output neuron.

$$\varepsilon_j = -\frac{\partial E}{\partial o_j} = \sum_{k;w_{ij}>0} -\frac{\partial E}{\partial n_k^+}\frac{\partial n_k^+}{\partial o_j} + \sum_{k;w_{ij}<0} -\frac{\partial E}{\partial n_k^-}\frac{\partial n_k^-}{\partial o_j} \tag{2.17}$$

Therefore, the result of the error for the hidden neuron becomes

$$\varepsilon_j = -\frac{\partial E}{\partial o_j} = \sum_{k;w_{ij}>0} \varepsilon_k\left(1 - n_k^-\right)\frac{w_{jk}^+\left(1 - n_k^+\right)}{1 - w_{jk}^+ o_j} + \sum_{k;w_{ij}<0} \varepsilon_k n_k^+ \frac{w_{jk}^-\left(1 - n_k^-\right)}{1 + w_{jk}^- o_j} \tag{2.18}$$

2.3 Summary

This chapter has presented a method of arithmetic using pulse streams. As discussed pulse streams allow computation using only simple gates. In addition this chapter has discussed the suitability of pulse stream implementations for neural networks. The theoretical analysis showed that the backpropagation learning algorithm can be used for training this network, using the OR-gate activation function as a continuous and differentiable non-linear function. It may be possible to implement neural networks of high density due to the use of simple digital gates for performing arithmetic.

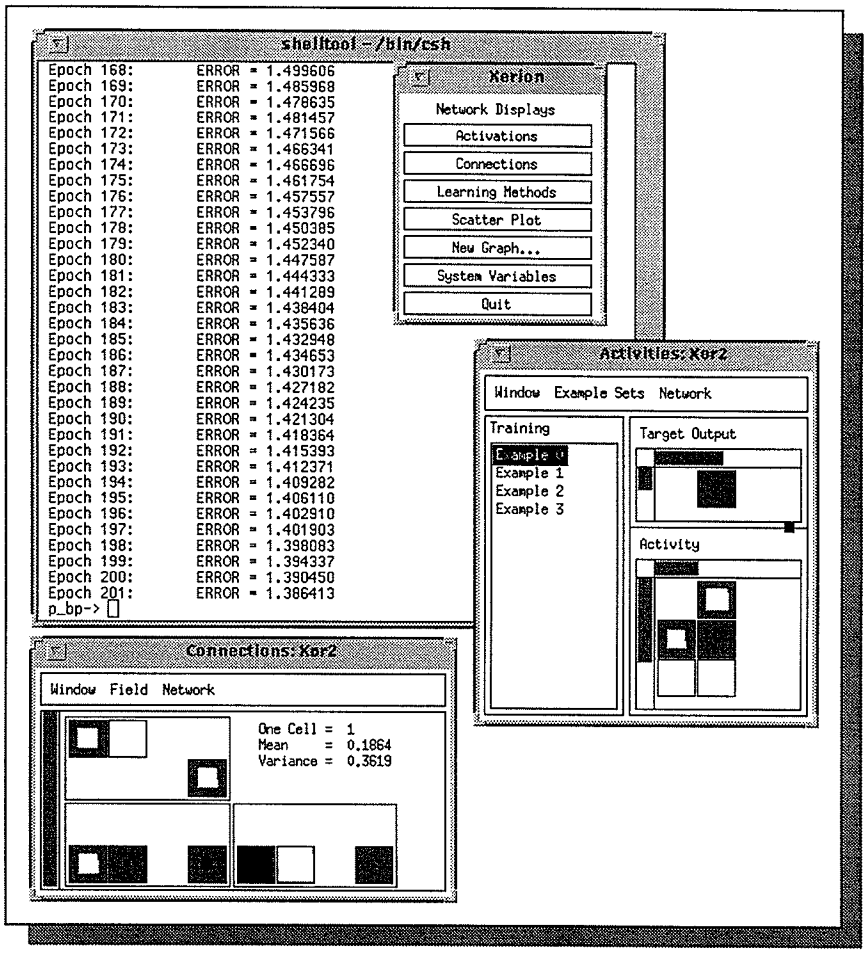
Chapter 3

Simulation of Pulse-Code Neural Networks

This chapter examines the simulation of pulse stream neural networks. Four examples will be discussed in this section. The first three examples are "toy problems" which are often used for testing and benchmarking neural networks. Typically the training set contains all possible input patterns so there is no question of generalization. The last example is a more real-world classification problem in which a network is trained to recognize the digits of a cheque book.

3.1 Simulation Environment

The networks can be simulated on two levels: probability and pulse stream. The probability level models the network activation as probabilities; the pulse level models all activations as pulses and directly emulates a hardware implementation of this network. In our simulator the probability model is chosen for the training of the networks, since it produces more accurate results without worrying about the hardware limitation of the network. A backpropagation algorithm is employed for training as described in the previous section.



```
shelltool - /bin/csh
Epoch 168:          ERROR = 1.499606
Epoch 169:          ERROR = 1.485968
Epoch 170:          ERROR = 1.478635
Epoch 171:          ERROR = 1.481457
Epoch 172:          ERROR = 1.471566
Epoch 173:          ERROR = 1.466341
Epoch 174:          ERROR = 1.466696
Epoch 175:          ERROR = 1.461754
Epoch 176:          ERROR = 1.457557
Epoch 177:          ERROR = 1.453796
Epoch 178:          ERROR = 1.450385
Epoch 179:          ERROR = 1.452340
Epoch 180:          ERROR = 1.447587
Epoch 181:          ERROR = 1.444333
Epoch 182:          ERROR = 1.441289
Epoch 183:          ERROR = 1.438404
Epoch 184:          ERROR = 1.435636
Epoch 185:          ERROR = 1.432948
Epoch 186:          ERROR = 1.434653
Epoch 187:          ERROR = 1.430173
Epoch 188:          ERROR = 1.427182
Epoch 189:          ERROR = 1.424235
Epoch 190:          ERROR = 1.421304
Epoch 191:          ERROR = 1.418364
Epoch 192:          ERROR = 1.415393
Epoch 193:          ERROR = 1.412371
Epoch 194:          ERROR = 1.409282
Epoch 195:          ERROR = 1.406110
Epoch 196:          ERROR = 1.402910
Epoch 197:          ERROR = 1.401903
Epoch 198:          ERROR = 1.398083
Epoch 199:          ERROR = 1.394337
Epoch 200:          ERROR = 1.390450
Epoch 201:          ERROR = 1.386413
p_bp->
```

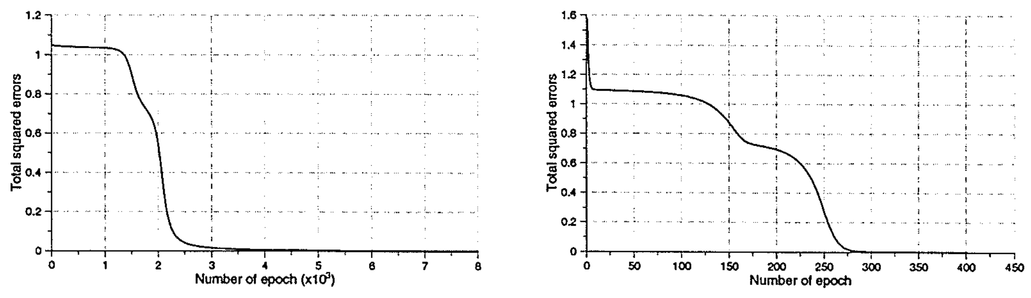
```
One Cell = 1
Mean     = 0.1864
Variance = 0.3619
```

Figure 3.1: The user interface of the pulse-code neural network simulator

The training of the pulse-code neural network has been performed off-line using C code interfaced with the Xerion neural network simulator library [17]. Xerion's libraries contain routines for constructing, displaying and training networks. The software also contains routines to graphically display neuron outputs and weights using Hinton diagrams. The pulse-code neural network simulator written using the Xerion libraries has access to a command line interface with built in commands for: creating and training networks, examining and modifying data structures, as well as miscellaneous utilities. Once the training is completed, the simulator can generate a network description file which includes the network topology and the weights of the synapses for hardware implementation.

3.2 XOR Problem

The XOR problem [6] is a frequently applied test of neural networks since it cannot be solved by a single layer network because it is not a linearly separable problem. A network consisting of two input neurons, two hidden neurons, and one output neuron was applied to the pulse-code neural network.



(a) Regular BP

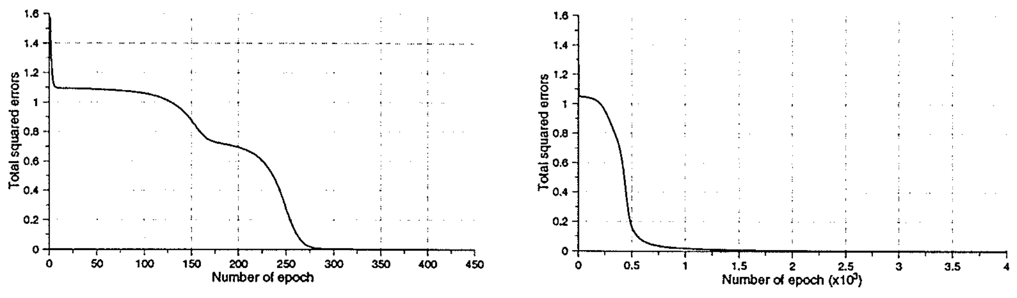
(b) Pulse-Code BP

Figure 3.2: Training evolution of the XOR problem.

Figure 3.2 shows the learning curve of the XOR problem trained with the regular and pulse-code backpropagation algorithms. It has an interesting result that the pulse-code network learns the XOR problem almost 10 times faster than the normal backpropagation network. The normal backpropagation network uses the sigmoid function as the nonlinear transfer function, while the pulse-code network uses the wired-OR gate addition saturation to obtain the nonlinearity. In addition, the initial weights of the pulse-code network must be very small to prevent immature saturation. In general the wired-OR gate addition saturation is not uniform throughout the network, depending on the number of inputs to the neuron (see Figure 2.3).

The learning equations for pulse-code networks contain division. Although division can be performed using pulse-code arithmetic [8], it is undesirable for a number of reasons. For example there are problems associated with division by zero, in addition to being a time consuming operation and requiring more hardware. Dickson[18] has proposed to omit division for training pulse-code network in order to reduce hardware overhead for implementing on chip learning capability.

To determine whether division is necessary the networks were trained with the division operation omitted. The results of the simulation are shown Figure 3.3. Although the XOR problem trained in more cycles without the division, in contrast again to regular backpropagation it still required only half the time. The results show that eliminating division does not impair the ability of the network to minimize the error.



(a) Training for XOR problem with denominator

(b) Training for XOR problem without denominator

Figure 3.3: The impact on division on network training.

3.3 Parity Problem

The parity problem is well known to researchers doing performance evaluation of neural networks. This problem is essentially a generalization of the XOR problem to N inputs [5]. Here the network is presented with binary inputs, 0's and 1's, and the single output neuron must output a high value if the input pattern has odd parity (an odd number of 1's), and a low value if the inputs have even parity. The parity problem is one of the most difficult problems for an artificial neural network to solve, because the network must look at all the input signals in order to determine whether the pattern has even or odd parity. This is untypical of most real-world classification problems which usually have much more regularity and allow generalization within classes of similar input patterns. Four inputs were used in this study. It is known that a network with four hidden neurons[19] can solve this problem using the standard backpropagation algorithm, but the pulse-code network failed to do so because of the limiation of the pulse stream representation. By trying different network topologies, the final network topology for the four bit parity problem consisted of two hidden layers of 12 and 8 units respectively. Figure 3.4 shows the error during training.

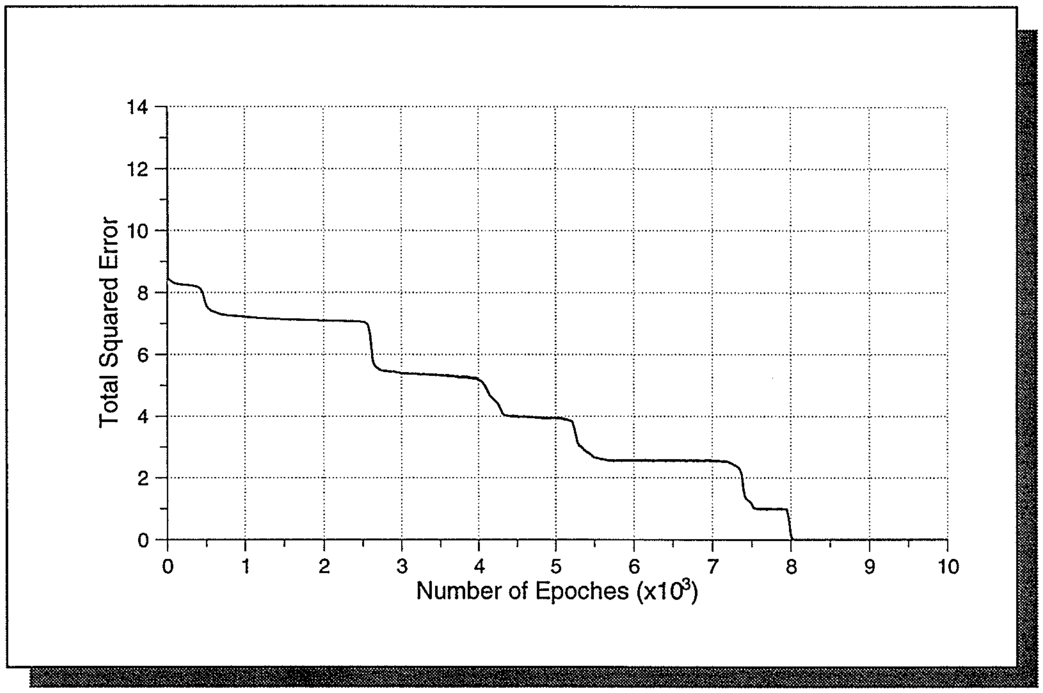


Figure 3.4: Training for the four bit parity problem

This problem has shown a limitation of the pulse-code representation, specifically the limited range of the synaptic weights. The other potential problem is using the OR gate for addition. As the number of inputs to a neuron (the OR gate) increases, the probability of a 1 output for a given set of inputs rises. This result suggests that the weights in a neural network must be very small to prevent the neuron from constantly saturating. The accommodation of the limited synaptic connections and premature neuron saturation is crucial for the success of pulse-code networks. While the complexity of the network is greater for these networks, the hardware complexity is still significantly less than the conventional digital networks. Since the hardware requirement is significantly less, the cost of adding more computing elements is not severe.

3.4 Encoder

The general encoding problem involves finding an efficient set of hidden neuron patterns to encode a large number of input/output patterns. The number of hidden neurons is intentionally made small to force an efficient encoding. The specific problem usually considered involves auto-association, using identical unary input and output patterns. A three-layer network consists of N inputs, N outputs and M hidden neurons, with $M < N$. This is often referred to as an N-M-N encoder. There are exactly N members of the training set, each having one input and the corresponding target on, and the rest off.

An 8 input encoder was used in this study. Conventional backpropagation could solve this problem with an 8-3-8 encoder network [19]. The activation pattern of the hidden neurons give the binary representation of the training pattern number. This can be achieved with the connection strengths patterned after the binary numbers. Clearly pulse code networks will fail as the weight connection is bounded by [-1, +1]. In order to solve this problem, a more complex pulse code network is needed. The final network uses 6 hidden neurons, which is an 8-6-8 encoder. Figure 3.5 shows the learning of the 8-6-8 encoder. Once again, the limited range of the synaptic weights is the roadblock of the learning capability in pulse code network.

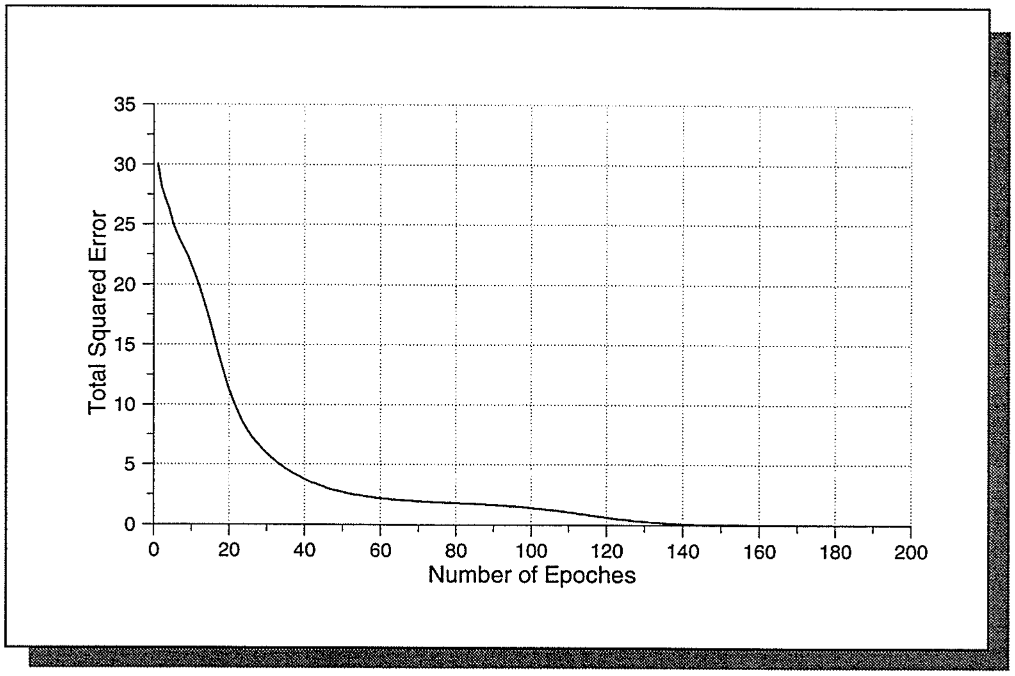
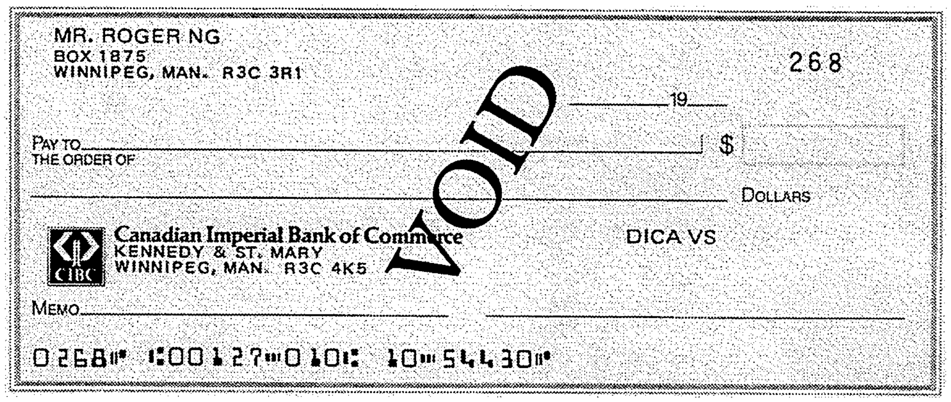


Figure 3.5: Training for 8-6-8 encoder

3.5 Cheque Character Recognition

In the proceeding three examples the training set normally includes all possible input patterns, so no generalization issues arises. Cheque character recognition is a more real-life problem in which noisy patterns can be used for testing the generalization ability. Most cheques have some strange characters indicating the account number, as shown in Figure 3.6. These characters are mapped to a 5x5 matrix as the input to this network with 10 output neurons corresponding to each of the possible character (0,1, ...,9). One hidden layer of six neurons is used to train the pulse-code network. The training error is shown in Figure 3.7 and the input activation is shown in Figure 3.8.



MR. ROGER NG
BOX 1875
WINNIPEG, MAN. R3C 3R1

268

19

PAY TO
THE ORDER OF $

DOLLARS

Canadian Imperial Bank of Commerce
KENNEDY & ST. MARY
WINNIPEG, MAN. R3C 4K5

DICA VS

VOID

MEMO

0268⑈ ⑆00127⑉010⑆ 10⑉54430⑈

Figure 3.6: Sample cheque

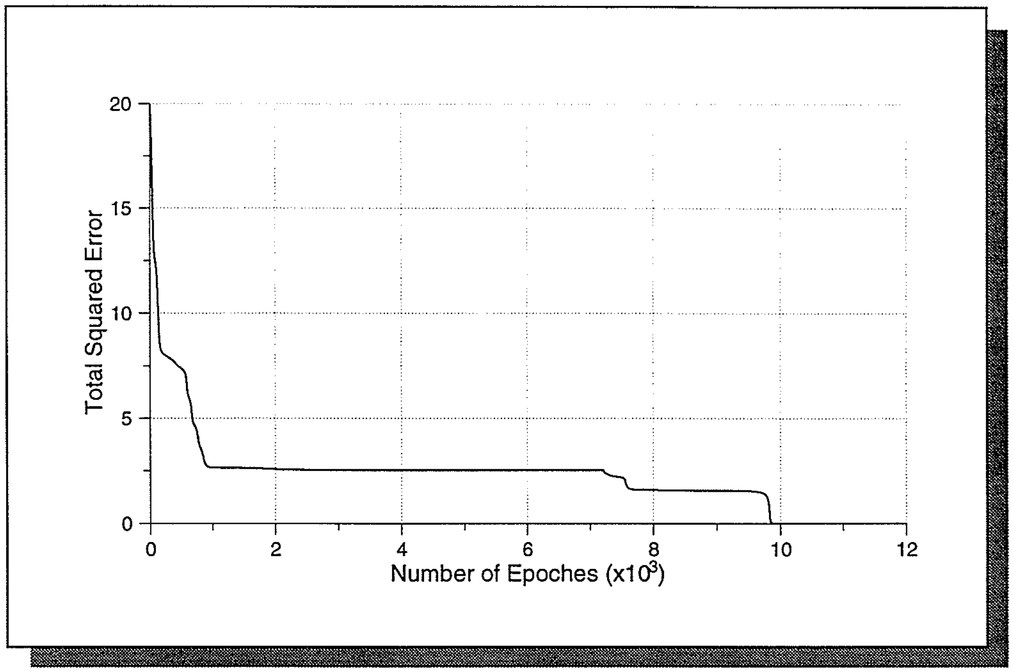


Figure 3.7: Training for the cheque character recognition

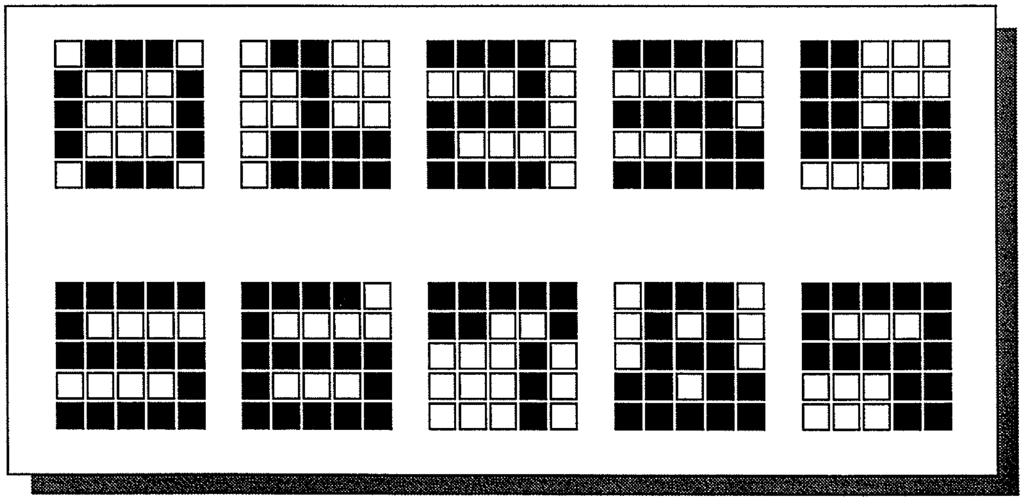


Figure 3.8: Input activation for the cheque character recognition problem

3.6 Summary

This chapter presented a number of simulation contrasting pulse-code neural network implementations to more traditional networks. Two potential problems arose using pulse-code representations. The first concerns the use of the OR gate for addition. As the nonlinearity of the OR gate depends on the number of inputs, some of the neurons could immaturely saturate. This problem can be solved using multi-layered architecture and limiting the number of inputs to each neuron. Rumelhart et. al [19] stated that "A simple method for overcoming the fan-out limitation is simply to use multiple layers of units." The second potential problem with pulse-code neural networks is the limited range of the weight connections. Increasing network topology or complexity could accommodate these limitations.

Chapter 4

Implementation of Pulse-Code Neural Networks in Xilinx FPGAs

This chapter describes the FPGA implementation of the pulse stream neural networks. It starts off by describing the overall hardware architecture of these networks, followed by the brief overview of Xilinx XC4000 series FPGAs. The design process of the network implementation is then described in detail. Finally, several examples of the networks implemented in Xilinx FPGAs are examined.

4.1 FPGAs Implementation

This section discusses the implementation of pulse stream neural networks in FPGAs.

4.1.1 Modular Design of Pulse Stream Neural Networks

There are two main components required to construct a pulse stream neural network; a random number generator and a neuron/synapse element.

Figure 4.1 shows the block diagram of the network structure. Each neuron synapse layer has one CA based random number generator. This random number is used to generate a weighted pulse stream of each synapse. The output of neuron/synapse elements are passed to the inputs of next layer.

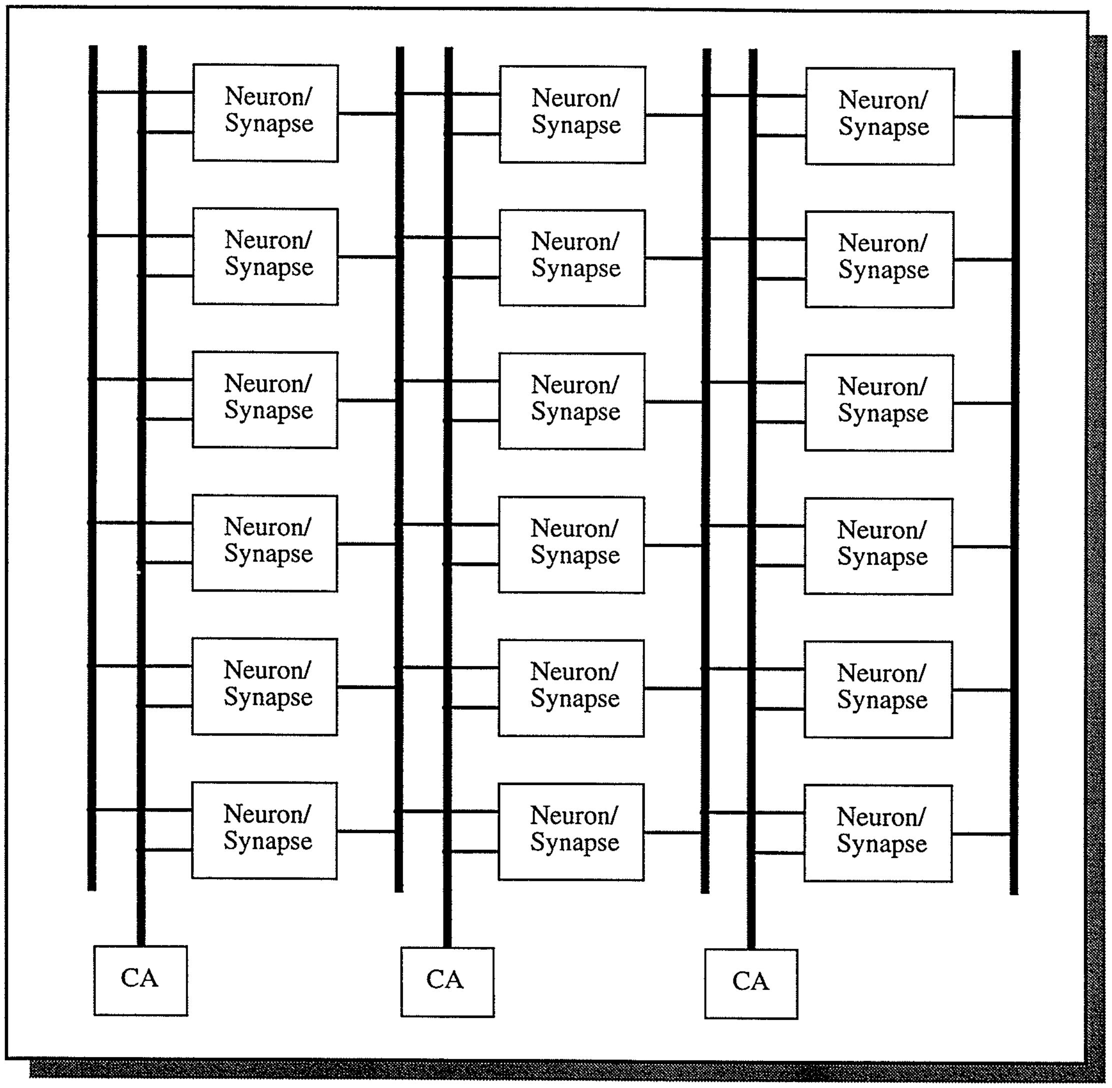


Figure 4.1: The top-level of pulse stream neural networks

4.1.2 Neuron Synapse Units

Each neuron synapse unit consists of one neuron element and n synapse elements. The number of synaptic elements depends on the number of input neurons from the previous layer. Figure 4.2 shows a block diagram of neuron/ synapse element.

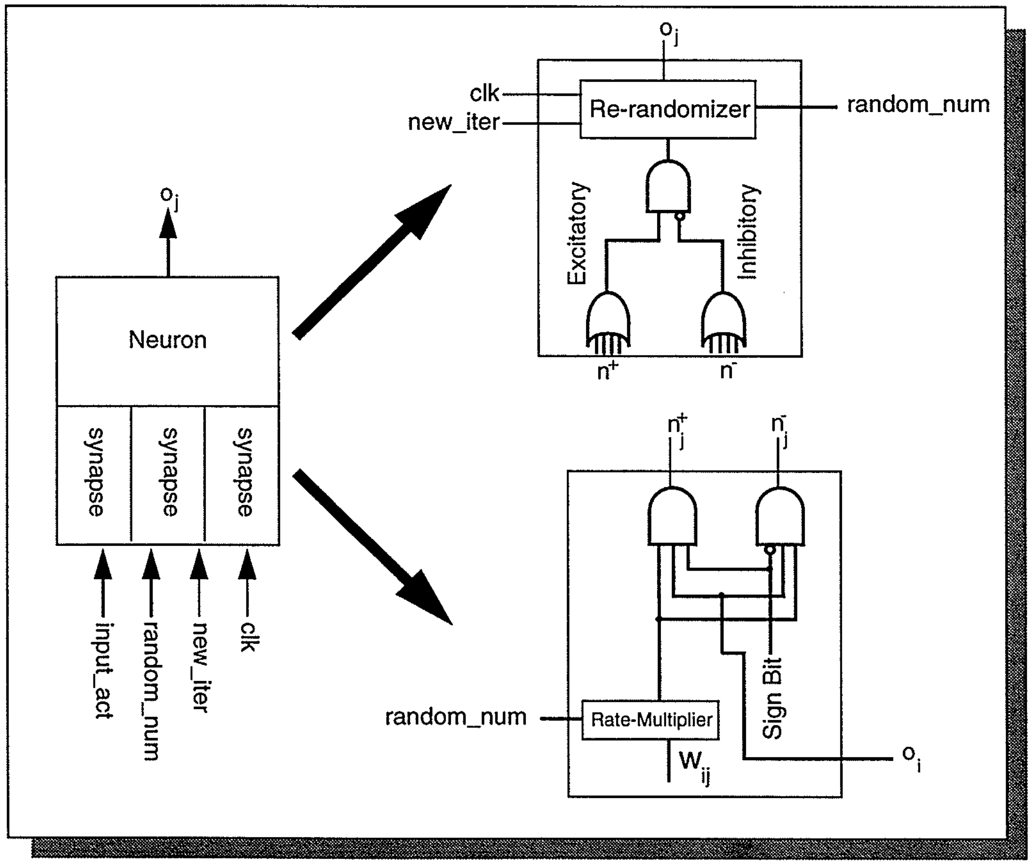


Figure 4.2: Neuron Synapse Unit

Each synaptic element has a preset weight value. A rate multiplier compares this weight value and a random number from a CA to produce a weighted pulse stream. This weighted pulse stream is ANDed with an input pulse stream from a previous layer to produce a synaptic multiplication. The product is transmitted to an excitatory net-input line or an inhibitory net-input line. If the sign bit of the weight is '0', the product pulse stream is transmitted to an excitatory net input line. Otherwise, it is transmitted to an inhibitory net input line.

The neuron preforms addition of all the net input signals from synapses through an OR gate. The outcome of these excitatory and inhibitory net input signals are ANDed to form the activation signal. This signal passes through a re-randomizer circuit to generate the output pulse stream.

4.1.3 Re-randomizer

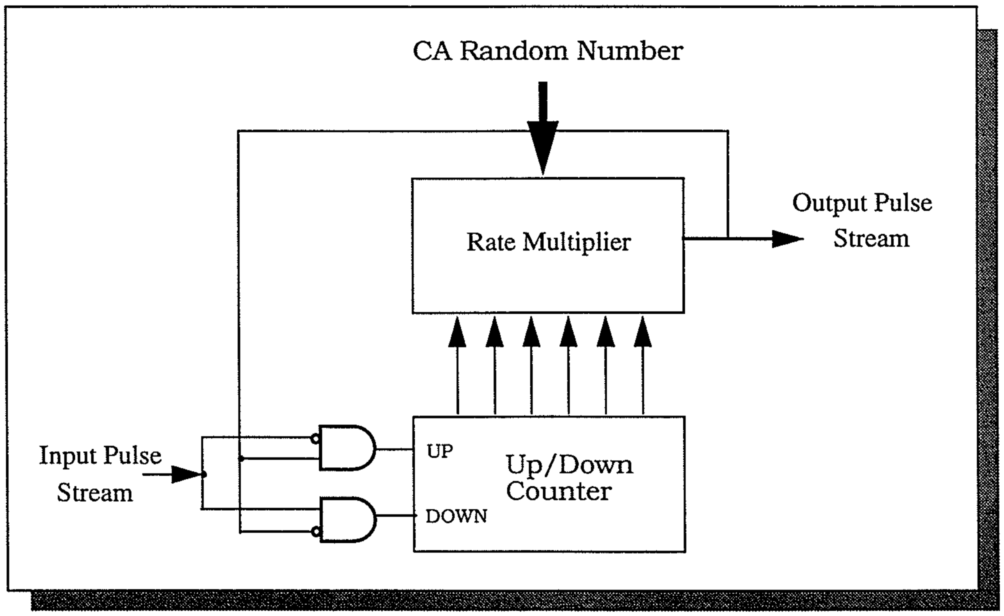


Figure 4.3: Block diagram of the re-randomizer circuit.

As mentioned in the earlier chapter the pulse stream arithmetic relies heavily on the assumed property of statistical independence between pulse streams. The re-randomizer re-orders the neuron output in order to prevent the correlation between the output pulse streams from the earlier layers.

The re-randomizer [18] consists of an up-down counter and a rate multiplier, as shown in Figure 4.3. The up-down counter controls the density of the output stream. If the output is high when the input is low, the counter is decremented. If the output is low and the input is high, then the counter is incremented. If the input and the output are equivalent then there is no change. The re-randomizer uses the random number from the neuron/synapse shifted one bit to the left. The rate multiplier compares the counter value and the random number to form the re-randomized output pulse stream.

4.1.4 Weight Resolution

The weights are stored in the form of an n-bit fractional sign-magnitude number. Each pulse represents the magnitude of $1/2^{n-1}$. A 9 bit weight resolution is chosen, which has 8 bits magnitude and 1 sign bit. Thus there are 2^8 -2 possible positive values, 2^8 - 2 negative values, and zero. Increased resolution requires more hardware due to a larger random number generator, neuron re-randomizer and rate multiplier. Kim et al.[20] has shown that a 9 bit weight provides an acceptable result for neural network classification and generalization.

4.2 Overview of Xilinx Field Programmable Gate Arrays

In 1985, a new technology for implementing digital logic was introduced, *Field Programmable Gate Arrays* (FPGAs). These devices could be viewed as a cross between *Mask-Programmable Gate Arrays* (MPGAs) and *Programmable Logic Devices* (PLDs). FPGAs are capable of implementing significantly more logic than PLDs, because they can implement multi-level of logic, while most PLDs are optimized for two-level logic. While they do not have the capacity of MPGAs, they also do not have to be custom fabricated, greatly lowering the costs for low-volume parts and avoiding long fabrication delays. One of the best know FPGAs is the Xilinx Logic Cell Arrays(LCAs)[2]. In this section their third generation FPGA, the Xilinx 4000 series will be discussed.

Xilinx FPGAs consist of an array of uncommitted logic elements that can be interconnected in a general way like MPGAs. It uses *static RAM* (SRAM) cells as the programmable element so that it can be re-programmed as many times as the designer wishes. Figure 4.4 shows a typical architecture of a Xilinx FPGA. It is a symmetrical array architecture, consisting of a two-dimensional array of *Configurable Logic Blocks* (CLBs) that can be connected by *programmable interconnection* resources. The interconnect comprises segments of wire, where the segments may be of various lengths. Present in the interconnect are programmable switches that serve to connect the CLBs to the wire segments, or one wire segment to another. Logic circuits are implemented in the FPGA by partitioning the logic into individual CLBs and then interconnecting the blocks as required via the switches. The *I/O Blocks* (IOBs) surround the boundary of the FPGAs, providing the interface between the packages pins and internal signal lines.

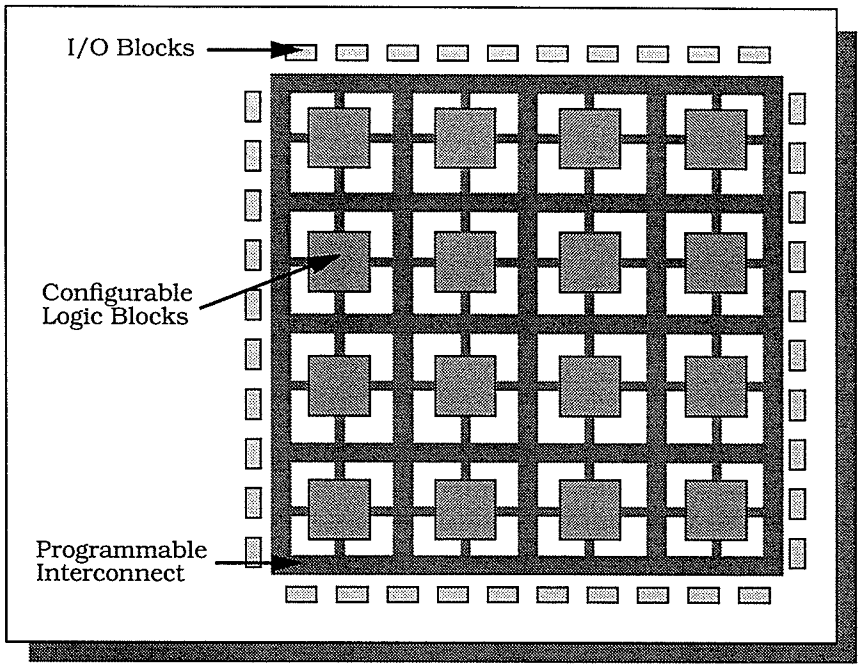


Figure 4.4: Xilinx FPGAs architecture.

A Xilinx 4000 series CLB, as shown in Figure 4.5, is made up of three *Lookup Tables* (LUTs), two programmable flip-flops, and multiple programmable multiplexers. The LUTs allow arbitrary combinational functions of their inputs to be created. Thus, the structure can perform any function of five inputs (using all three LUTs, with the F & G inputs identical), any two functions of four inputs (the two 4-input LUTs used independently), or some functions of up to nine inputs (using all three LUTs, with F & G inputs different). SRAM controlled multiplexers then can route these signals out the X and Y outputs, as well as to the two flip-flops. The inputs at top (C1-C4) provide the third input to the 3-input LUT, enable and set or reset signals to the flip-flops, and a direct connection to the flip-flop inputs. This structure yields a very powerful method of implementing arbitrary, complex digital logic. Note that there are several additional features of the Xilinx FPGA not shown in these figure, including support for embedded memories and carry chains.

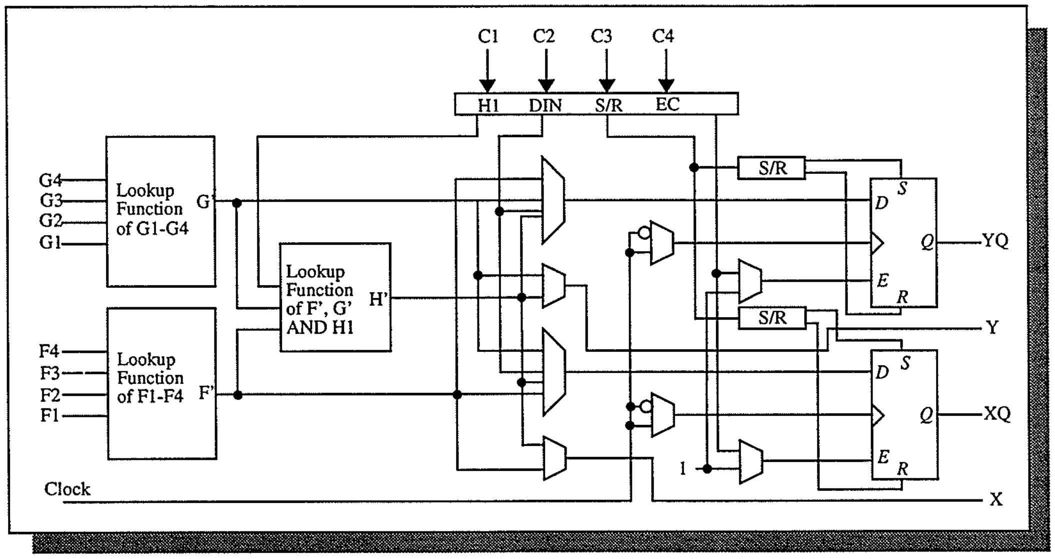


Figure 4.5: XC4000 CLB

The CLBs are surrounded by horizontal and vertical routing channels that permit arbitrary point-to-point communication. All internal connections are composed of metal segments with programmable switching points to implement the desired routing. There are three main types of interconnect, distinguished by the relative length of their segments: single-length lines, double-length lines, and longlines. Single-length lines travel the height of a single CLB, where they then enter a switch matrix. The switch matrix allows this signal to travel out vertically and/or horizontally from the switch martix. Thus, multiple single-length lines can be cascaded together to travel longer distances. Double-length lines are similar, except that they travel the height of two CLBs before entering a switch matrix, thus double-length lines are useful for longer-distance routing, traversing two CLB heights without the extra delay and the wasted configuration sites of an intermediate switch matrix. Finally, longlines are lines that go half the chip height, and do not enter the switch matrix. In this way, very long-distance routes can be accommodated efficiently. With this rich sea of routing resources, the Xilinx 4000 series is able to handle fairly arbitrary routing demands, though mappings that emphasize local communication will still be handled more efficiently.

4.3 Design Flow of Pulse Code Neural Network Hardware

Although the Xerion neural network simulator is valuable tool for simulating pulse-code networks, another goal of this thesis is to search for a design flow generating FPGA hardware from a high level network description. Ideally the design flow progresses from the Xerion neural network simulation to the generation of a Xilinx bit file for programming the FPGA device. Xerion is used to simulate and train the pulse-code neural network, iterate the design, and then generate a network description file including the network topology and final

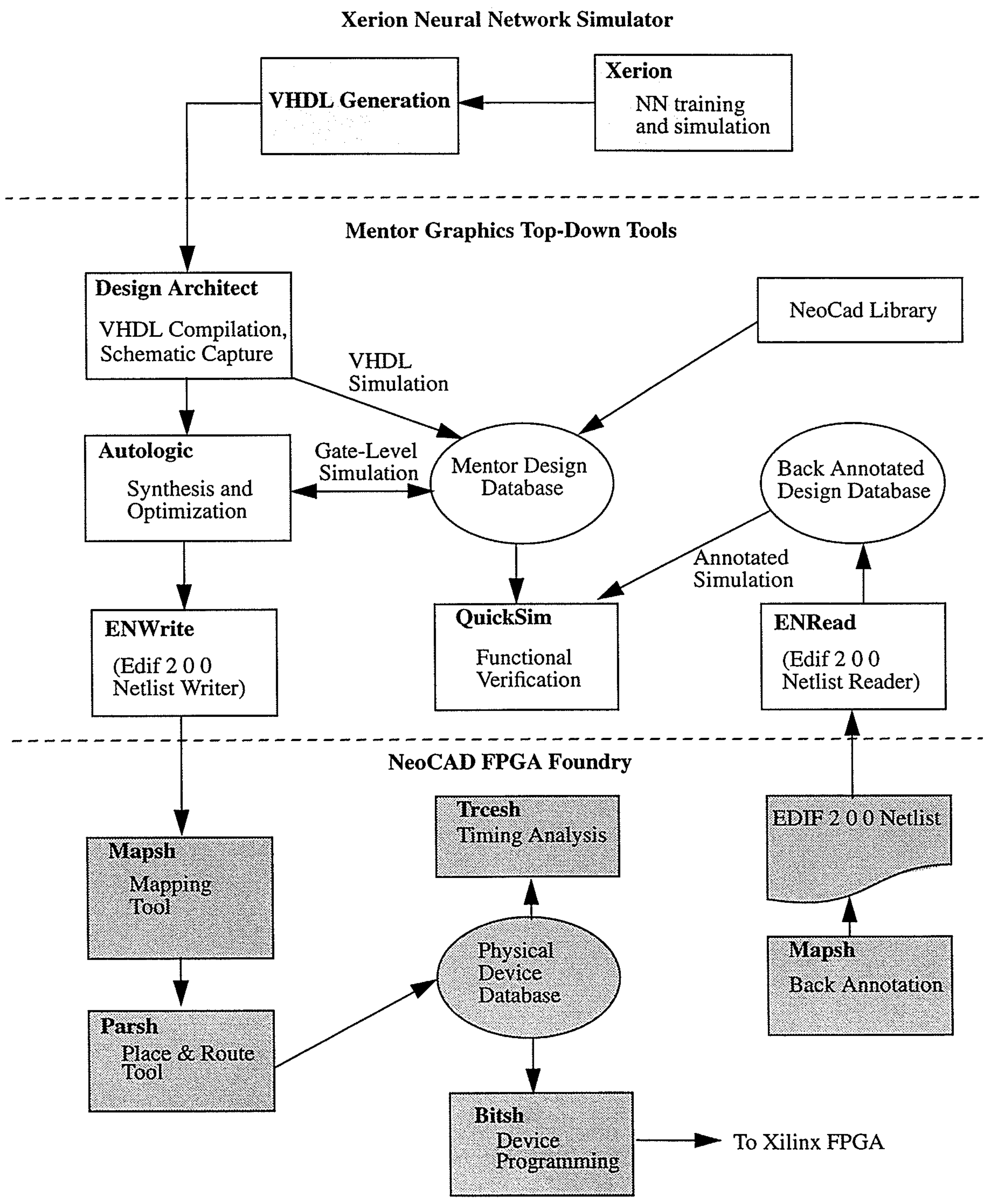


Figure 4.6: Pulse-code neural network design process

weights. Using this description, a custom 'C' program converts it to a VHDL description. Mentor Graphics Top-Down Tools [21] are used for VHDL[1] compilation, syntax verification, synthesis, optimization, and simulation. NeoCad FPGA Foundry tools[22] are used to map the design to a physical FPGA device and to create the bit file for programming the Xilinx chip. Static timing analysis of the placed and routed design is also done within the NeoCad tools. Finally, the design information is back-annotated to a Mentor Graphics database and functionally tested against the top-level VHDL testbench. The complete design flow is shown in Figure 4.6

4.3.1 Database Structure

As the design goes through various tools, it is very important to organize the database properly. Many procedural problems can be avoided by planning the directory structure. Figure 4.7 illustrates one example of how to organize design database for the design.

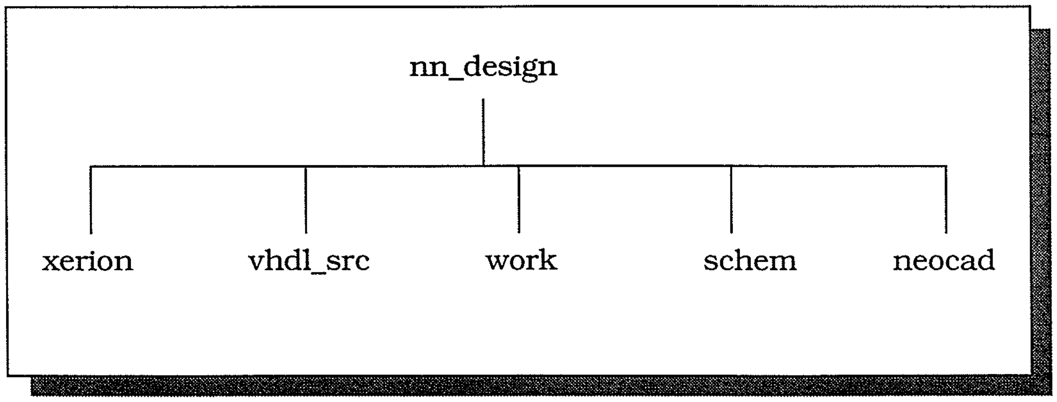


Figure 4.7: Database organization

1. VHDL - VHSIC Hardware Description Language is a language for designing integrated circuits.

The neural network design directory is divided into five sub-directories for Xerion simulation, VHDL source, synthesis and optimization, gate-level schematic and the physical FPGA layout.

4.3.2 Xerion Neural Network Simulator

Xerion is used to train and simulate the pulse code neural networks, as mentioned in the previous chapter. The input of the simulator is a text file with the description of the network topology and example training data. The networks train with the backpropagation algorithm. Once the training is completed, Xerion generates a network specification for hardware implementation. This specification includes the topology and the weights of synapses.

4.3.3 VHDL Code

The network specification generated from Xerion is converted into VHDL code for hardware implementation. VHDL is a language for designing Integrated Circuits, which can describe the circuits at the behaviour and/or structure level. In order to ensure that the VHDL code is synthesizable, the designs must be described at Register Transfer Language (RTL) level. In addition, there are certain code styles to use when targeting Xilinx FPGAs. Appendix A provides design hints for writing VHDL for Xilinx FPGA designs. A custom "C" program is used to generate the synthesizable VHDL code for Xilinx FPGAs from the network specification.

The VHDL description of the network is hierarchically organized. The top-level circuit represents the connections between the neuron/synapse units and CA random number generators. It is described in structural VHDL. The neuron/synapse units and CAs are described in RTL descriptions.

Each VHDL component must have an entity which defines the I/O of the model. Each VHDL component in the design also has several architectures. For neuron/synapse units and CAs, the RTL description of the circuit is in an architecture called *RTL*. The top-level circuit uses a structural VHDL model which should be in an architecture called *struct*.

Other VHDL architectures are also necessary. In order to facilitate the creation of a hierarchical schematic for the top-level circuit, a *dummy* architecture has to be created for each low-level circuit (neuron/synapse elements and CAs). These *dummy* architectures have nothing between the BEGIN and END statements in the VHDL. They are just place holders for the schematic generation process. A *schem* architecture is required of the top-level circuit. The architecture is the same as the struct architecture except that it calls only the *dummy* VHDL architecture for the circuits beneath it.

Mentor Graphics' system-1076 compiler performs the syntax checking and database generation of the VHDL design. Once syntactically correct, the compiler creates a *Mentor Eddm database* from the VHDL code which can be simulated in Quicksim for simulation and read into Autologic for synthesis.

4.3.4 Synthesis and Optimization

After compiling the VHDL files, the Mentor Eddm database is read into Autologic and synthesized to the Xilinx XC4000 FPGAs. The NeoCad Xilinx XC4000 library is used and the target environment variables are set to commercial derating factors.

The neuron/synapse elements and CA circuits in the low-level hierarchy are written in RTL level VHDL. This code is synthesized directly to XC4000 gates. These circuits are synthesized separately. All hierarchy implied in the VHDL code at this level is flattened to improve the area optimization. The optimization recipe in Autologic used is AREA(LOW) with an AREA REPORT. Since timing optimization is not available this is all that is required at this level.

Symbols must be created for each low-level circuit so that they can be referenced by the top-level hierarchical schematic. These symbols are automatically generated when the VHDL entities are compiled. To save these symbols, they must be opened within *Design Architect* and saved.

The top-level for this design is only the connection between the neuron/synapse and CAs, which is defined in a structural VHDL netlist. In order to generate a hierachical schematic for the top-level circuit in *Autologic*, the following multiple step process is used:

- All neurons/synapse units and CAs circuit must have a dummy architectures.
- The low-level circuit must have been previously synthesized to gates and a symbol created to represent the circuits.
- The top-level circuit should have a *schem* architecture which calls the dummy architecture of the low-level circuits. This was previously mentioned.
- The *schem* architecture of the top-level circuit is synthesized and optimized into a schematic.
- On the resulting schematic, the dummy components are replaced with symbols for the real circuits that have been previously synthesized.
- Back in Autologic, the resulting schematic with real componets is re-optimized. The I/O ports and buffers are added to the final schematic.

4.3.5 Design Verification

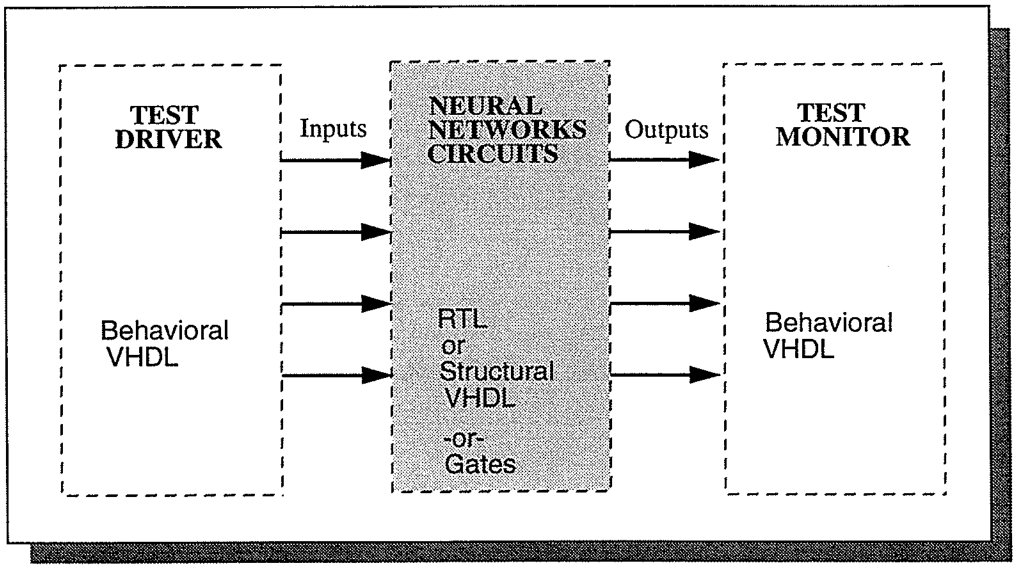


Figure 4.8: VHDL testbenches

Quicksim is used to simulate the operation of the circuits in both VHDL and XC4000 gate representations. To functionally verify the design, VHDL test-benches are created for the top-level circuits. After synthesis, the same test-benches are used to simulate the gate-level circuit. The VHDL and a gate operation are compared to make sure they match.

The VHDL test-benches are only a piece of behavioural VHDL to drive the input. If the design is complex, then a piece of VHDL can also be written to monitor the outputs and report if an error is seen. The advantage to the VHDL driver/monitor test-bench is that it can be used to test the gate-level models as well as the VHDL. Figure 4.8 shows the connections of a VHDL testbench and how it can be used to drive and monitor the operation of a circuit block. For pulse stream neural network design, the test-driver provides the input example to the network, and the test-monitor uses a number of up counters to monitor the pulse density of the output neurons.

4.3.6 Neocad FPGA Foundry

The Neocad foundry tools are used to map the Xilinx XC4000 gates into a physical array. The NeoCad tools accept data from Mentor Graphics in the form of EDIF 2 0 0 netlist. *Mapsh*, mapping tool, maps the EDIF file into the specific Xilinx part and package, performs the design rule checking. and generates a NeoCad database file for place and route. *Parsh*, place and route tool, performs the place the and route of the FPGA. *Trcesh* is then used to analysize the timing of the design. The layout related timing information is back-annotation to Mentor Graphics design verification tools. Finally, Neocad can create a bitstream file which is used to physically program the Xilinx FPGA chip

4.4 FPGAs Design Examples

Three neural networks example problems are implemented into Xilinx FPGAs: the XOR, the encoder and the cheque character recognition. The first two examples are implemented onto Xilinx XC4010 PG -6 FPGAs. Appendix B provides a quick reference of different Xilinx 4000 series FPGAs that are available. The XC4010 part has 4000 CLBs equivalent to 10,000 gates. Therefore, it is able to accommodate a significantly large design in a single FPGA. The speed grade of this part is 6 which means there is a 6ns delay of each CLB. The last problem is implemented on a Xilinx XC4013 FPGA. In this section, the simulation of the XOR problem will be discussed and the area and timing of all three problems will also be examined.

4.4.7 XOR Problem

The XOR problem, as mentioned in the chapter 3, is implemented into Xilinx XC4010 FPGAs. The networks to solve this problem consists of two input neurons, two hidden neurons and output neuron.

A top-level testbench was created to verify the VHDL model and the gate level representation. This testbench presented inputs to the network and monitored the output pulse density using and up-counter. Figure 4.9 shows the simulation result of the network. The curve in the figure represents the value of the counter in the re-randomizer of the output neuron. The initial value of re-randomizers has a "precharging" value, 1/2 of the maximum counter value. With each presentation of an input vector the value of re-randomizer is reset to "precharging" value. The simulation result has proved that the design is functioning as expected.

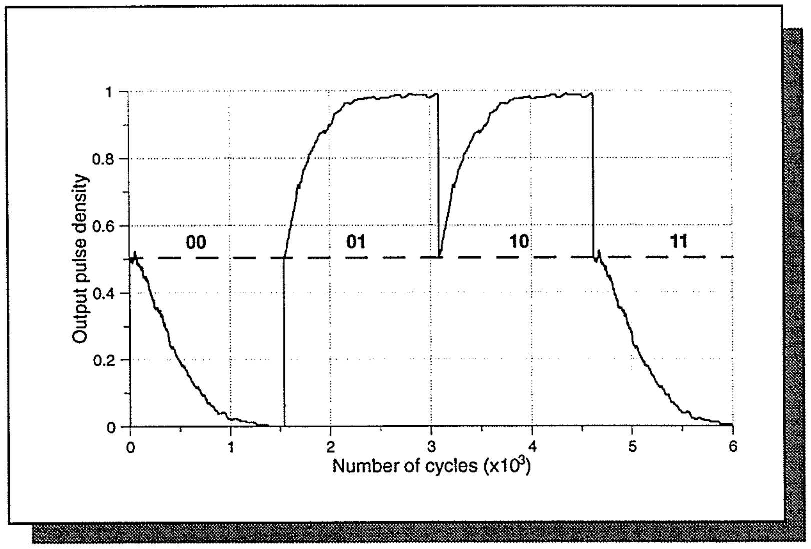


Figure 4.9: XOR simulation results

The complete layout of the design is shown in Figure 4.10. The design required 69 CLBs and 5 IOBs. It has the interesting result that there are two clusters of CLBs in the layout. The small cluster of the CLBs is the output neuron and the larger one is the two hidden neurons. It only takes 23 CLBs per neuron in this design.

Neocad's timing analysis tools, *Trcesh*, is used to perform the static timing analysis. Figure 4.11 shows the timing report of the XOR design. It details the maximum delay path of the design. This report identifies both logic delays and route delays. The 'R' next to the delay entry indicates a delay based on rising edge timing of signals through a IOB or CLB. The maximum delay path of this design is 171.585ns. The report also shows a breakdown of the percentage of delay attributed to logic vs. routing delays. In this case 53.6% is logic delay and 46.4% is routing delay. The total delay of the design is 171.585ns logic and routing delay plus 8.0ns setup, which is 179.585ns. Therefore the maximum operating frequency of the XOR design is 5.568MHz.

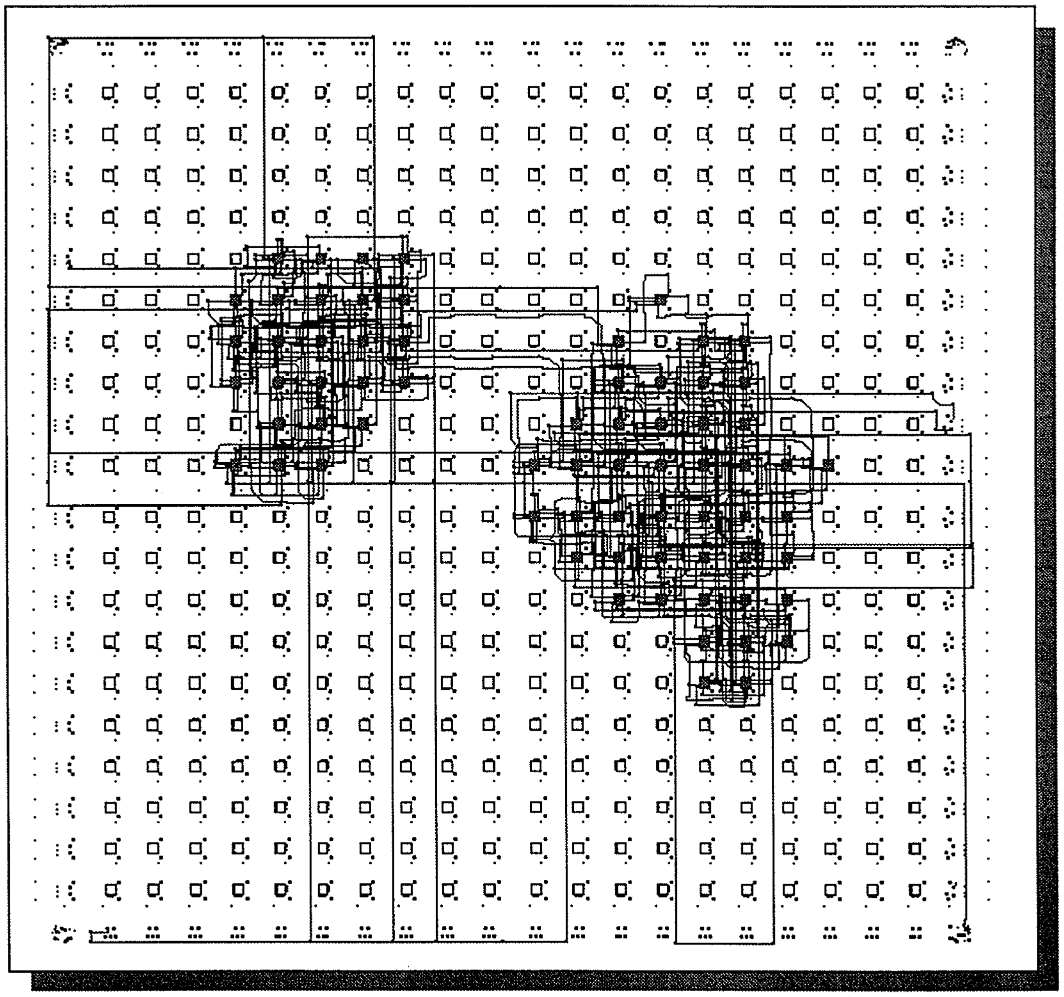
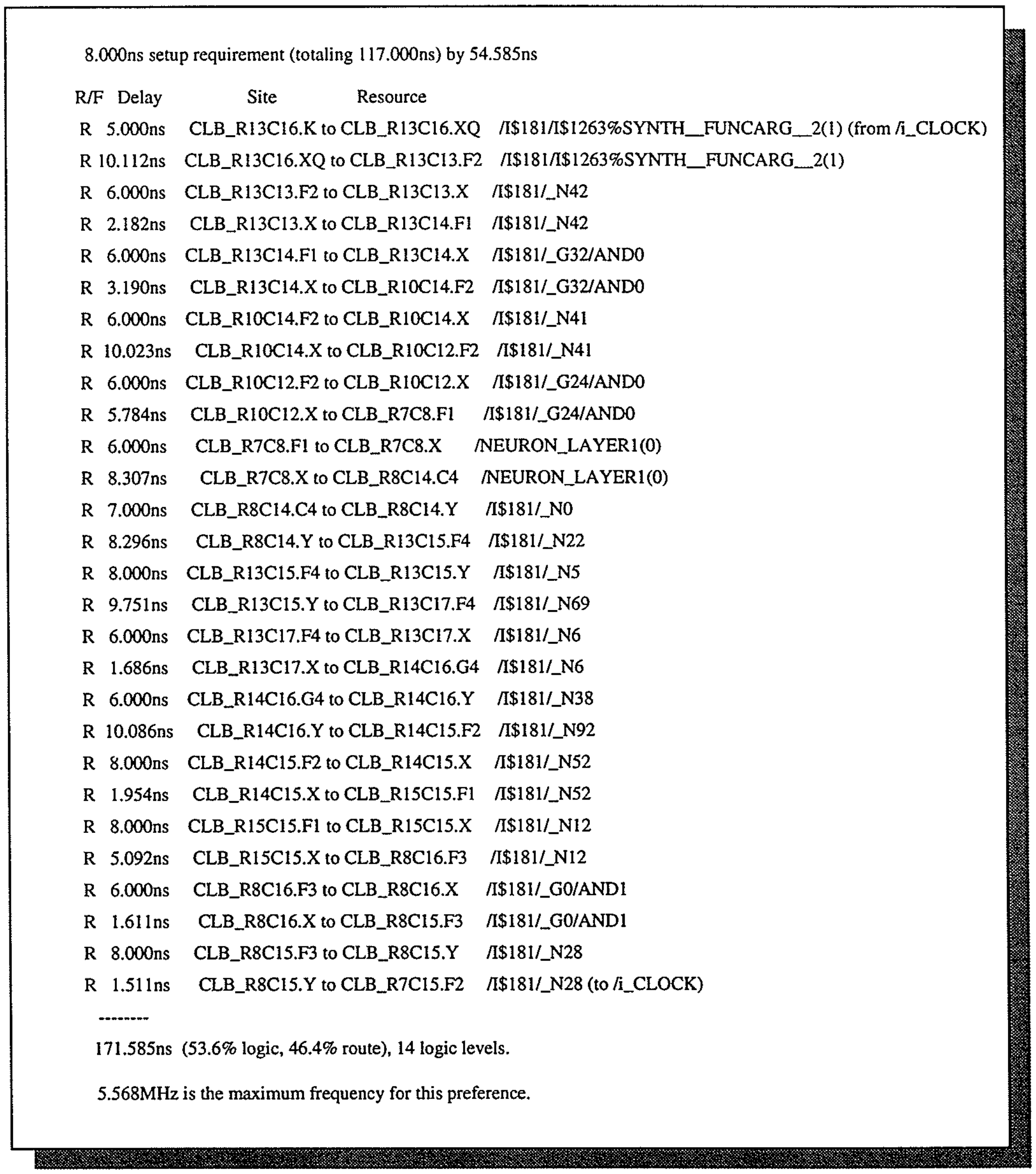


Figure 4.10: The FPGA layout of XOR problem



```
8.000ns setup requirement (totaling 117.000ns) by 54.585ns

R/F  Delay          Site               Resource
R  5.000ns   CLB_R13C16.K to CLB_R13C16.XQ   /I$181/I$1263%SYNTH__FUNCARG__2(1) (from /i_CLOCK)
R 10.112ns  CLB_R13C16.XQ to CLB_R13C13.F2   /I$181/I$1263%SYNTH__FUNCARG__2(1)
R  6.000ns  CLB_R13C13.F2 to CLB_R13C13.X    /I$181/_N42
R  2.182ns   CLB_R13C13.X to CLB_R13C14.F1   /I$181/_N42
R  6.000ns  CLB_R13C14.F1 to CLB_R13C14.X    /I$181/_G32/AND0
R  3.190ns   CLB_R13C14.X to CLB_R10C14.F2   /I$181/_G32/AND0
R  6.000ns  CLB_R10C14.F2 to CLB_R10C14.X    /I$181/_N41
R 10.023ns   CLB_R10C14.X to CLB_R10C12.F2   /I$181/_N41
R  6.000ns  CLB_R10C12.F2 to CLB_R10C12.X    /I$181/_G24/AND0
R  5.784ns   CLB_R10C12.X to CLB_R7C8.F1    /I$181/_G24/AND0
R  6.000ns    CLB_R7C8.F1 to CLB_R7C8.X     /NEURON_LAYER1(0)
R  8.307ns     CLB_R7C8.X to CLB_R8C14.C4    /NEURON_LAYER1(0)
R  7.000ns   CLB_R8C14.C4 to CLB_R8C14.Y     /I$181/_N0
R  8.296ns    CLB_R8C14.Y to CLB_R13C15.F4   /I$181/_N22
R  8.000ns  CLB_R13C15.F4 to CLB_R13C15.Y    /I$181/_N5
R  9.751ns   CLB_R13C15.Y to CLB_R13C17.F4   /I$181/_N69
R  6.000ns  CLB_R13C17.F4 to CLB_R13C17.X    /I$181/_N6
R  1.686ns   CLB_R13C17.X to CLB_R14C16.G4   /I$181/_N6
R  6.000ns  CLB_R14C16.G4 to CLB_R14C16.Y    /I$181/_N38
R 10.086ns   CLB_R14C16.Y to CLB_R14C15.F2   /I$181/_N92
R  8.000ns  CLB_R14C15.F2 to CLB_R14C15.X    /I$181/_N52
R  1.954ns   CLB_R14C15.X to CLB_R15C15.F1   /I$181/_N52
R  8.000ns  CLB_R15C15.F1 to CLB_R15C15.X    /I$181/_N12
R  5.092ns   CLB_R15C15.X to CLB_R8C16.F3    /I$181/_N12
R  6.000ns   CLB_R8C16.F3 to CLB_R8C16.X     /I$181/_G0/AND1
R  1.611ns    CLB_R8C16.X to CLB_R8C15.F3    /I$181/_G0/AND1
R  8.000ns   CLB_R8C15.F3 to CLB_R8C15.Y     /I$181/_N28
R  1.511ns    CLB_R8C15.Y to CLB_R7C15.F2    /I$181/_N28 (to /i_CLOCK)
 --------
 171.585ns  (53.6% logic, 46.4% route), 14 logic levels.

 5.568MHz is the maximum frequency for this preference.
```

Figure 4.11: The timing report of the XOR FPGA design

4.4.8 Encoder Problem

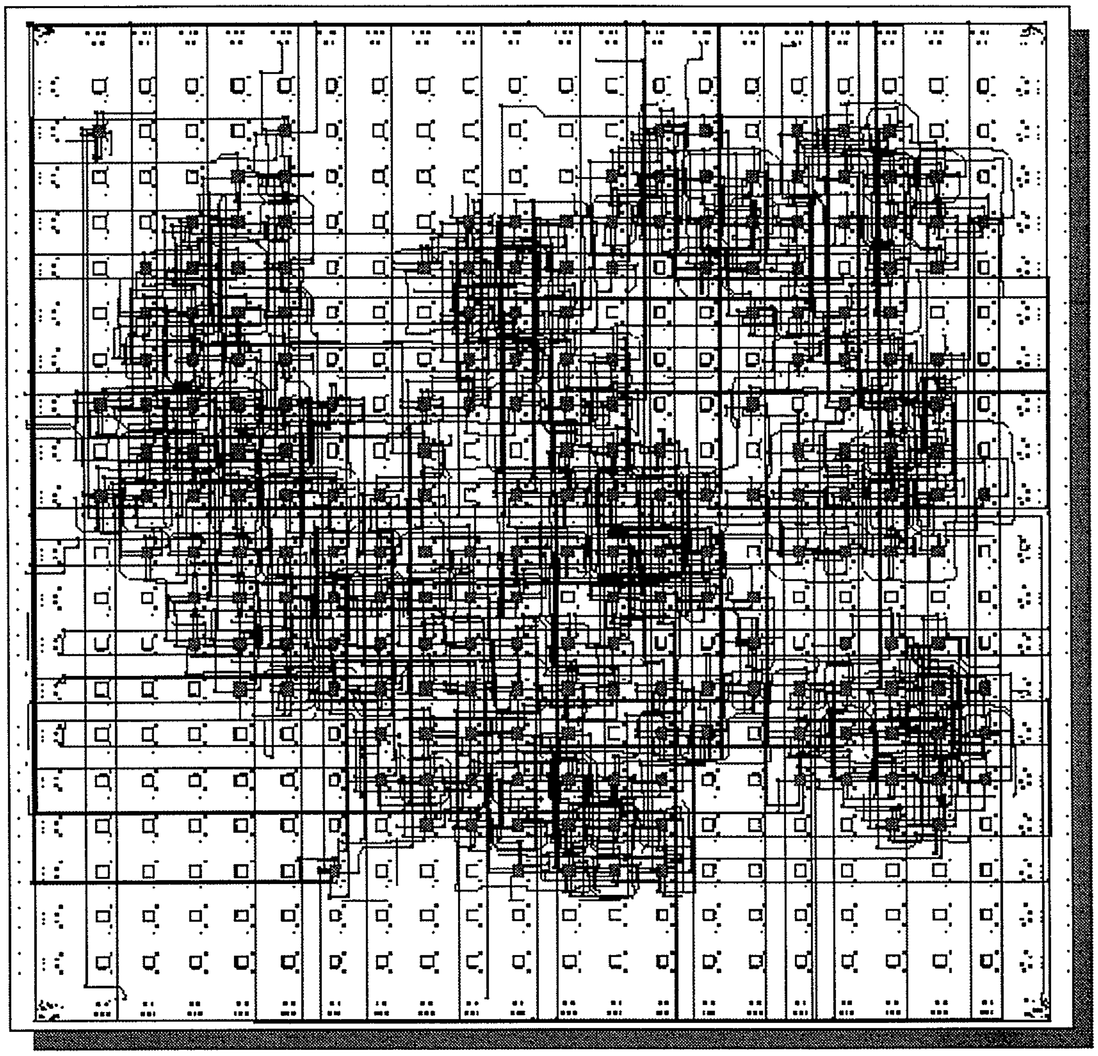


Figure 4.12: The FPGA layout of 5-4-5 encoder

The other design example is an encoder. The network has five inputs, four hidden and five output neurons. This network is smaller than the one discussed in chapter 3, but it is the same class of problem. The FPGA layout of the encoder design is shown in Figure 4.12. There are 215 CLBs and 12 IOBs used. The total logic and routing delay is 249.346ns with 44.9% from logic and 55.1% from routing. The routing delay in this design has higher percentage than the XOR as it has more synaptic connections between the layers. The maximum operating frequency is 3.886Hz. The average number of CLBs per neurons are 23.89. The design has more synapses per neuron than the XOR but this average is still very close to that of the XOR. As all the synapses are combinational logic, the place and route is able to efficiently pack them into the CLBs.

4.4.9 Cheque Character Recognition

This problem was discussed in chapter 3 and the network has 25 inputs, 6 hidden and 10 output neurons. This design is implemented into a Xilinx XC4013 FPGA. It used 506 CLBs and 37 IOBs and the average number of CLBs per neurons is 31.63. The maximum delay path of this design is 340.09 with 32.6% logic and 67.4% routing. The maximum operating frequency is 2.873MHz.

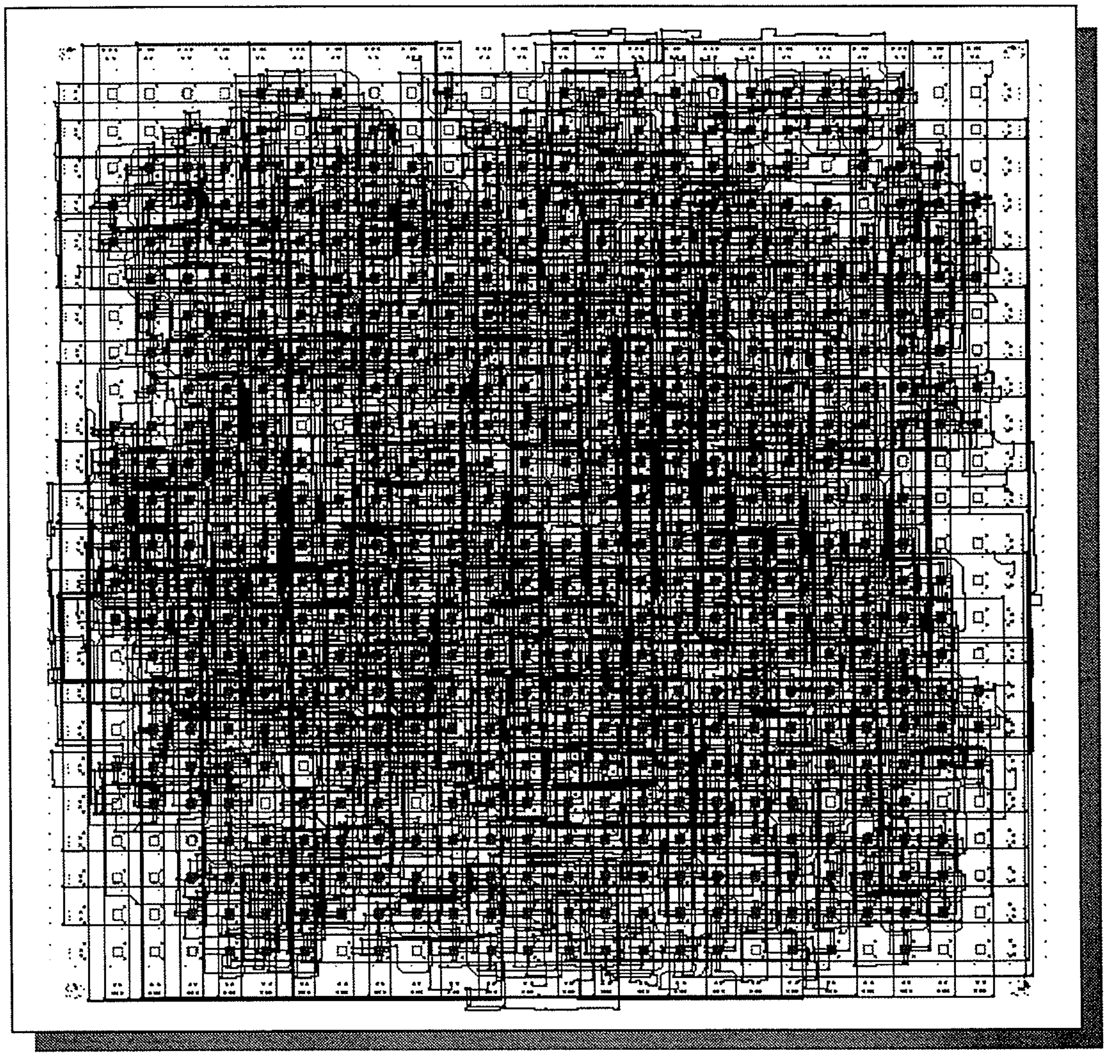


Figure 4.13: The FPGA layout of cheque character recognition

4.5 Summary

This chapter has presented a hardware implementation of multi-layer neural networks using pulse-code arithmetic. The design of the networks is hierarchically organized so that they results in more optimized circuity during logic synthesis and optimization. The networks are divided into neuron/synapse units and random number generators. This chapter also discussed a re-randomizer for the neuron output in order to prevent conrrelation between the neuron pulse streams.

A top-down design flow for constructing these networks has been discussed. This flow progresses from a high-level network description to the generation of a Xilinx bit stream for programming the Xilinx FPGA device. The use of a VHDL testbench for design verification was also discussed. Following this, three example problems were implemented on Xilinx 4000 series FPGAs. The implementation results showed that the network are extremely compact and use only 23 CLBs per neuron/synapse unit for XOR problem.

Chapter 5

Conclusions and Future Work

This thesis has demonstrated the implementation of pulse-code neural networks in Xilinx FPGAs. The hardware requirements of these networks was shown to be minimal; only simple digital gates were required to perform the arithmetic. Also, the use of the backpropagation learning algorithm for training, as well the simulation of these networks was discussed. The simulation results suggested that the weights in a neural network must be very small to prevent the neuron from constantly saturating and multi-layers networks should be used in order to overcome the fan-in limitation of the neuron.

The hardware architecture of these networks was described as well the top-down design flow for implementing these networks in Xilinx FPGAs. In addition, two design examples, the XOR and encoder, were implemented and examined. The implementation results have shown that the average number of CLBs per neuron/synapse unit was only 23 for XOR problem. The increase in the number of the synapse connections of the neuron did not significantly contribute to this hardware cost. As a result a significantly large network can be implement on a single Xilinx FPGA. For example, approximately 44 neuron/synapse units could be implemented on a Xilinx XC4025 part, the largest Xilinx 4000 series part which has 1024 CLBs on a single chip. With the aid of the current state of art CAD tools, the design cycle took only days to complete as opposed to traditional design methodology of weeks or months.

Continued work in this area should investigate the use of time multiplexing to further increase the number of neurons per device. The idea is to have one single physical layer of neurons and re-use them for different layers emulating multiple layers of neurons.

The use of multiple FPGA environments for implementing these networks should also be investigated. In this case, very large scale neural networks can implemented for prototying. As well, other FPGAs device should be considered. One potential candidate is the new Xilinx 8000 series FPGA, which is a sea of gate architecture. Since neural network structures are highly regular with little global wiring, the basic architecture is similar to the architecture of the XC8000 series FPGAs, therefore better utilization of the FPGA can be achieved.

A sophisticated high level interface that compiles a given neural architecture directly to a single FPGA or multi-FPGA based hardware system should be developed. However, the logic synthesis tools could be a problem for such systems as the current logic synthesis tools do not work well when the design exceeds 3000 gates. Further work should be done on partitioning the networks into small pieces for logic synthesis as well as on the use the Mentor Graphics' design management software, *WorkXpert*, to automate the design flow and design capture.

Appendix A

Targeting VHDL Design to Xilinx FPGAs

As the density and complexity of Xilinx FPGA designs increase to 20,000 gates and beyond, the traditional schematic capture design entry is often cumbersome. The use of *hardware description languages* (HDLs), such as VHDL and Verilog HDL, can raise designer productivity. High-level languages combined with logic synthesis can provide a consistent design methodology across a range of technologies. By raising the level of design abstraction, synthesis tools can increase productivity, ensuring error-free gate level realizations and freeing designers for more creative tasks. However, the designer should not ease up on hardware implementation consideration when synthesis tool aids are available. The methods for designing ASICs do not always apply to designing with Xilinx FPGAs. ASICs have more gates and routing resources than Xilinx FPGAs. Since ASICs have a large number of available resources, the designer can easily create inefficient code that results in a large number of gates. When designing with Xilinx FPGAs, the designer must create efficient code.

The VHSIC Hardware Description Language (VHDL) is a language for designing Integrated Circuits (ICs), which can descibe the designs at the behaviour and/or structure level. VHDL designs can be behaviourally simulated and tested to be functionally correct before synthesis. However many VHDL constructs are not supported by synthesis tools. In general, only a subset of VHDL constructs, called *Register Transfer Level* (RTL) constructs, are accepted by the synthesis tools. In addition, systhesis tools intreprete the VHDL code differently when targeting different technologies. The following guidelines ensure VHDL code that takes the best advantages of Xilinx's resources and produces the same functionality after synthesis.

A.1 Wait for XX ns Statement

Wait for XX ns statements specifies the number of nanoseconds that must pass before a condition is executed. This statement does not synthesize to a component. In designs that include this statement, the functionality of the simulated design does not match the functionality of the synthesized design.

A.2 After XX ns Statement

After XX ns statement is usually used as a condition of a signal assignment. This statement is usually ignored by the synthesis tool. An example of this statement is:

```
Q <=0 after  xx ns
```

A.3 Initial Values

Assigning signals and variables initial values are ignored by most synthesis tools. The functionality of the simulated design may not match the functionality of the synthesized design. For example, do not use initialization statements such as the following:

```
variable SUM: INTEGER:=0
```

A.4 Order and Group Arithmetic Functions

The ordering and grouping of arithmetic functions influence design performance. For example, the following two statements are not equivalent:

```
ADD <= A1 + A2 +A3 +A4;
ADD <= (A1 + A2) + (A3 + A4);
```

The first statement cascades three adders in series. The second statement creates two adders in parallel: A1 + A2 and A3 + A4. In the second statement, the two additions are evaluated in parallel and the results are combined with a third adder. RTL simulation results are the same for both statements, however, the second statement results in a faster circuit after synthesis.

A.5 Xilinx Name Conventions

Xilinx has reserve names for their FPGA. The following FPGA resource names are reserved and should not be used to name nets or components:

- Configurable Logic Blocks (CLBs), Input/Output Blocks (IOBs), clock buffers, tristate buffer (BUFTs), oscillators, package pin names, CCLK, DP, GND, VCC, and RST
- CLB names such as AA, AB, and RIC2
- Primitive names such as TD0, BSCAN, M0, M1, M2, or STARTUP
- Do not use pin names such as P1 and P2 for component names
- Do not use pad names such as PAD1 for component names

For further Xilinx naming conventions, Xilinx Data Books [2] provide a more detailed references.

A.6 Latches and Registers

VHDL compilers infer latches from incomplete specifications of conditional expressions. Latch primitives are not available in XC4000 CLBs, however, the IOBs contain input latches. Latches described in VHDL are implemented with gates in the CLB function generators. For example, the D latch shown in Figure A.1 is implemented with one function generator. The D Latch implemented with gates is shown in Figure A.2.

```
LIBRARY mgc_portable ;
USE mgc_portable.qsim_logic.all ;
ENTITY d_latch IS
     PORT ( GATE, DATA: in qsim_state;
          Q: out qsim_state);
end case_ex;

ARCHITECTURE BEHAV OF d_latch IS
begin

  LATCH: process (GATE, DATA)
  begin
    if (GATE = '1') then
      Q <= DATA;
    end if;
  end process; --End LATCH
end BEHAV;
```

Figure A.1: Latch inference

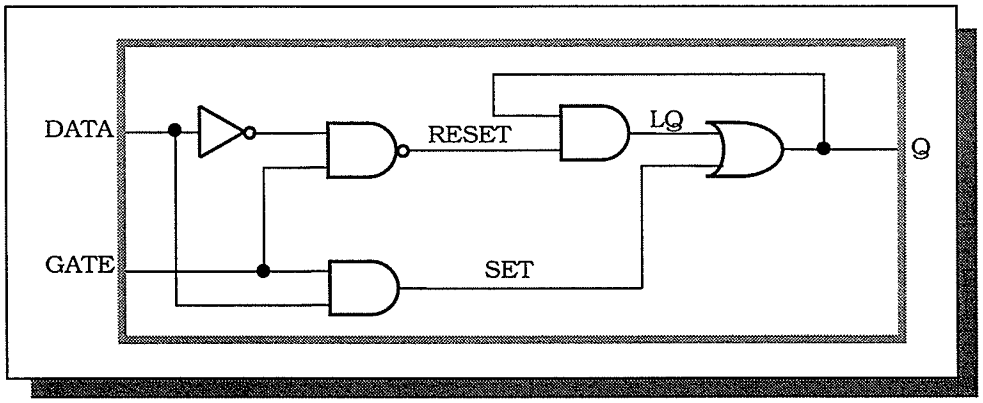


Figure A.2: Latch implemented with gates

In this example, the VHDL code contains an IF statement without the ELSE which always implies a latch in gate-level representation. The drawback of a latch is that it is implemented as a combinatorial feedback loop in a CLB and synthesis tools do not process hold-time requirements because of the uncertainty of routing delays. In order to eliminate unnecessary latches, it is desirable to replace them with D registers, as each CLB has two D flip-flops. To convert a latch to a D register uses an ELSE clause in the IF statement or a WAIT UNTIL statement.

In all other cases (such as latches with reset/set or enable), use the D flip-flop instead of a latch. This rule also applies to JK and SR flip-flops. Table A.1 provides a comparison of area and speed for a D latch implemented with gates and a D flip-flop.

Table A.1: D latch implementation comparison

	D Latch	D Flip-Flop
Advantages/ Disadvantages	VHDL that infers D latch implemented with gates. Combinatorial feed-back loop results in hold-time requirement.	Requires change to VHDL to convert D latches to D flip-flops, No hold time or combinatorial loop
Area	I Function Generator	1 Register
Speed	1 Logic Level Combinatorial feedback loop	1 Logic Level; no combinatorial loop.

A.7 Implementing Multiplexers with Tristate Buffers

A 4-to-1 multiplexer is efficiently implemented in a single XC4000 CLB. The six input signals (four inputs, two select lines) use the F, G, and H function generators. Multiplexers that are larger than 4-to-1 exceed the capacity of one

CLB. For example a 16-to-1 multiplexer requires five CLBs and has two logic levels. These additional CLBs increase area and delay. In order to utilize XC4000 resources, using tristate buffers (BUFTs) is recommended to implement multiplexers larger than 4-to-1.

A VHDL design of a 5-to-1 multiplexer built with gates is shown in Figure A.3. Typically, the gate version of this multiplexer has binary encoded selector inputs and requires three select inputs (SEL<2:0>). The schematic representation of this design is shown in Figure A.3

```
LIBRARY mgc_portable ;
USE mgc_portable.qsim_logic.all ;
ENTITY mux_gate IS
    PORT ( sel: in qsim_state_vector(2 downto 0);
           A,B,C,D,E: in qsim_state;
            MUX_OUT: out qsim_state);
end mux_gate;

ARCHITECTURE BEHAV OF mux_gate IS
begin

  SEL_PROCESS: process (SEL,A,B,C,D,E)
  begin
    case SEL is
      when "000"    =>   MUX_OUT <= A;
      when "001"    =>   MUX_OUT <= B;
      when "010"    =>   MUX_OUT <= C;
      when "011"    =>   MUX_OUT <= D;
      when others   =>   MUX_OUT <= E;
    end case;
  end process; --End SEL_PROCESS
end BEHAV;
```

Figure A.3: Implementing 5-to-1 MUX with gates

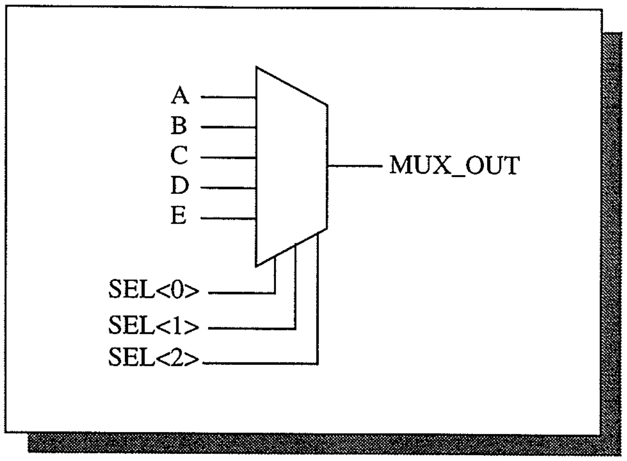


Figure A.4: 5-to-1 MUX implemented with gates

```
LIBRARY mgc_portable ;
USE mgc_portable.qsim_logic.all ;
ENTITY mux_tbuf IS
      PORT ( sel: in qsim_state_vector(4 downto 0);
            A,B,C,D,E: in qsim_state;
            MUX_OUT: out qsim_state_resolved_x);
end mux_tbuf;

ARCHITECTURE BEHAV OF mux_tbuf IS
begin
  MUX_OUT <= A when (SEL(0)='0') else 'Z';
  MUX_OUT <= B when (SEL(1)='0') else 'Z';
  MUX_OUT <= C when (SEL(2)='0') else 'Z';
  MUX_OUT <= D when (SEL(3)='0') else 'Z';
  MUX_OUT <= E when (SEL(4)='0') else 'Z';
end BEHAV;
```

Figure A.5: Implementing 5-to-1 MUX with BUFTs

The VHDL design shown in Figure A.5 is a 5-to-1 multiplexer built with tristate buffers. The tristate buffer version of the multiplexer has one-hot encoded selector inputs and requires five select inputs SEL<4:0>. The schematic representation of this design is shown in Figure A.6.

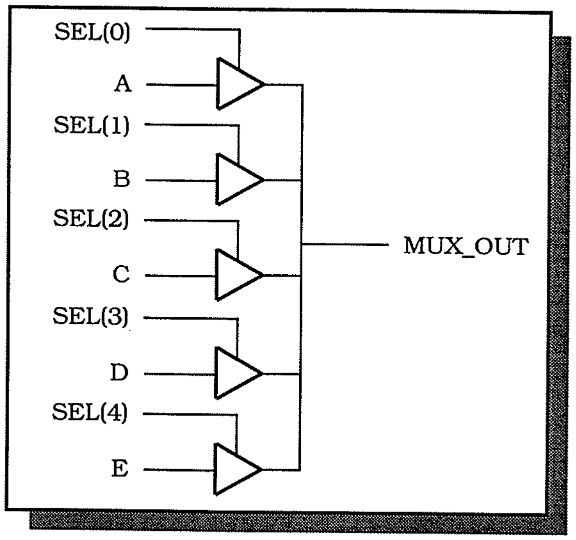


Figure A.6: 5-10-1 MUX implemented with BUFTs

Appendix B

Xilinx Device Quick References

Table B.1: Xilinx Devices, Packages and Speed Grades

Device	Packages	Speed Grades
XC4002A	PC84 PQ100 PG120	-5 -6
XC4003A	PC84 PQ100 CB100 CQ100 PG120	-5 -6
XC4003	PC84 PQ100 CB100 CQ100 PG120	-5 -6
XC4004A	PC84 PQ100 PG120 PQ160	-5 -6
XC4005A	PC84 PG156 PQ160 PQ208	-5 -6
XC4005	PC84 PG156 PQ160 CB164 PQ208	-5 -6 -10
XC4006	PG156 PQ160 PQ208	-5 -6
XC4008	PG191 CB196 PQ208	-5 -6 -10
XC4010	PG191 CB196 PQ208 MQ208	-5 -6 -10
XC4013	PG223 MQ208 PQ240 MQ256	-5 -6 -10
XC4025	PG223 PG299 MQ240	-5 -6 -10

References

[1] B. Gilbert, “A High-Performance Monolithic Multiplier Using Active Feedback,” *IEEE J. Solid-State Circuits*, vol. SC-9, pp. 364-373, 1974.

[2] Xilinx Inc., *The Xilinx Data Book*, 1994.

[3] W. S. McCulloch and W. Pitts, "A Logical Calculus of The Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics 5*, pp. 115-133, 1943.

[4] F. Rosenblatt, "Principles of Neurodynamics," *New York: Spartan Books*, 1959.

[5] M. Minsky and S. Papert, "Perceptrons: An Introduction to Computational Geometry," *Cambridge, MA: The MIT Press*, 1969.

[6] D. Rumelhart, G. Hinton and R. Williams, "Learning Internal Representation by Backpropagating Errors," *Nature: 323*, pp. 533-536, 1986.

[7] B. R. Gaines, “Stochastic Computing Systems,” *Advances in information System Science, volume-2*, Julius T. Tou, editor, Plenum Press, 1969.

[8] P. Mars, *Stochastic and Deterministic Averaging Processors*, The Institution of Electrical Engineers, London and New York, 1981.

[9] S.W. Golomb, "Shift Register Sequences", *Holden-Day Publishing Co., San Franciso*, 1982.

[10] P. Hortensius, R. McLeod, B. Podaima, "Cellular Automata Circuits for Built-In Self-Test", *IBM Journal of Research and Development vol 34*, March, 1990.

[11] P. Hortensius, "Parallel Computation of Non-deterministic Algorithms in VLSI," *Ph.D. thesis,* Department of Electrical and Computer Engineering, University of Manitoba, 1987.

[12] F. Breglez, C. Gloster, and G. Kedem. "Hardware-based Weighted Random Pattern Generation for Boundary Scan," *IEEE International Test Conference*, Aug 1989.

[13] J. Tomberg, T. Ritoniemi, K. Kaski and H. Tenhunen, "Full Digital Neural Network Implementation Based on Pulse Density Modulation," *Proc. IEEE Custom Integrated Circuits Conf.*, (San Diego, CA; May 15-17), pp. 12.7.1-12.7.4, 1989.

[14] J. Tomberg and K. Kaski, "Pulse-density Modulation Technique in VLSI Implementation of Neural Network Algorithms," *IEEE Journal of Solid State Circuits, 25(2)*, pp. 1277-1286, Oct. 1990.

[15] M. Tomlinson Jr., M. Walker and M. Silvilott, "A Digital Neural Network Architecture for VLSI," *Proc. IJCNN-90*, pp. 545-550, San Diego, CA, 1990.

[16] J. Dickson, R. McLeod and H. Card, "Stochastic Arithmetic Implementations of Neural Networks with In Situ Learning," *IEEE International Conference on Neural Networks,* (San Francisco, CA; Mar. 28-Apr. 1), pp. 711-716, 1993.

[17] Drew van Camp, Evan E. Steeg, and Tony Plate. *XERION Neural Network Simulator*. Computer Science Department, University of Toronto, 1991.

[18] J. Dickson, "Stochastic Arithmetic Implementation of Artificial Neural Networks," *MSc. Thesis,* Department of Electrical Engineering, University of Manitoba, 1992.

[19] D. Rumelhart, J. McClelland and PDP Research Group, *Parallel Distributed Processing Volume 1*, The MIT Press, 1986.

[20] Y. C. Kim, and M. Shanblatt, "Random Noise Effects in Pulse-Mode Digital Multilayer Neural Networks", *IEEE Transactions on Neural Networks Vol 6, No. 1*, January 1995.

[21] Mentor Graphics, *Bold Browser*, 1995.

[22] NeoCad Inc, *NeoCad FPGA Foundry Tutorial*, 1994