5-1-1995

A VHDL model of a digi-neocognitron neural network for VLSI

Troy Brewster

A VHDL Model of a Digi-Neocognitron Neural Network for VLSI

by

Troy D. Brewster

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Computer Engineering

Approved by:

Graduate Advisor - George A. Brown, Professor

Peter G. Anderson, Professor

Tony H. Chang, Professor

Department of Computer Engineering
College of Engineering
Rochester Institute of Technology
Rochester, New York

May 1995

ABSTRACT

Optical character recognition is useful in many aspects of business. However, the use of conventional computers to provide a solution to this problem has not been very effective. Over the past two decades, researchers have utilized artificial neural networks for optical character recognition with considerable success. One such neural network is the neocognitron, a real-valued, multi-layered hierarchical network that simulates the human visual system. The neocognitron was shown to have the capability for pattern recognition despite variations in size, shape or the presence of deformations from the trained patterns. Unfortunately, the neocognitron is an analog network which prevents it from taking full advantage of the many advances in VLSI technology. Major advances in VLSI technology have been in the digital medium. Therefore, it appears necessary to adapt the neocognitron to an efficient digital neural network if it is to be implemented in VLSI.

Recent research has shown that through preprocessing approximations and definition of new model functions, the neocognitron is well suited for implementation in digital VLSI. This thesis uses this methodology to implement a large scale digital neocognitron model. The new model, the digi-neocognitron, uses supervised learning and is trained to recognize ten handwritten numerals with widths of one pixel.

The development of the neocognitron and the digi-neocognitron software models, and a comparison of their performance will be discussed. This is followed by the development and simulation of the digital model using the VHSIC Hardware Description Language (VHDL). The VHDL model is used to demonstrate the functionality of the hardware model and to aid in its design. The model functions of the digi-neocognitron are then implemented and simulated for a 1.2 μm CMOS process.

THESIS RELEASE PERMISSION FORM

ROCHESTER INSTITUTE OF TECHNOLOGY
COLLEGE OF ENGINEERING

Title : A VHDL Model of a Digi-Neocognitron Neural Network for VLSI

I, Troy D. Brewster, hereby deny permission to the Wallace Memorial Library of RIT to reproduce this thesis in whole or in part.

Signature : _____

Date : _____

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY OF TERMS

ANN	Artificial Neural Networks
DNC	Digi-Neocognitron
CMOS	Complementary Metal Oxide Semiconductor
EEPROM	Electrically Erasable Programmable Read Only Memory
EPROM	Erasable Programmable Read Only Memory
IC	Integrated Circuit
I/O	Input/Output
LGB	Lateral Geniculate Body
NC	Neocognitron
OCR	Optical Character Recognition
PDP	Parallel Distributed Processing
RAM	Random Access Memory
ROM	Read Only Memory
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration

CHAPTER 1

1.0 Introduction

1.1 What are Artificial Neural Networks?

The task of pattern processing is present in many real-world problems. This problem set includes areas such as image processing, speech processing, natural language processing, planning, forecasting, and optimization. Such problems deal with large amounts of data with mutually interacting factors. They also require that the information to be processed be precisely specified. Due to these two factors, conventional computers are not well suited for such types of problems. However, artificial neural networks (ANNs) or parallel distributed processing (PDP) models addresses both problems. These networks attempt to simulate the functionality of natural, biological brains in solving pattern processing problems.

ANNs are parallel signal processing networks comprised of a large number of processing elements or neurons which interact via weighted connections. These weighted connections are representative of the biological synapses in the neural system; they can be excitatory, causing the neurons to become active, or inhibitory, suppressing the neurons' outputs. Altering the value of the weights associated with the connections enables the network to adapt to changes in the environment. This procedure is referred to as learning and can be either supervised or unsupervised.

During supervised and unsupervised learning, numerous sets of training patterns are repeatedly presented to the network until it develops the ability to recognize those patterns. Typically, in supervised learning, the trainer presents a pair of patterns to the network consisting of an input pattern and the target output. The network then adjusts the weights of the processing elements based on a calculated error value. This is usually the difference between the expected output and the computed output of each processing element. In contrast, unsupervised learning attempts to classify the input patterns with no information about the expected output. The network detects the patterns' regularities and the grouping for

each to produce a consistent output. This type of model is referred to as biological model, and the process described above is known as "self-organization" [1].

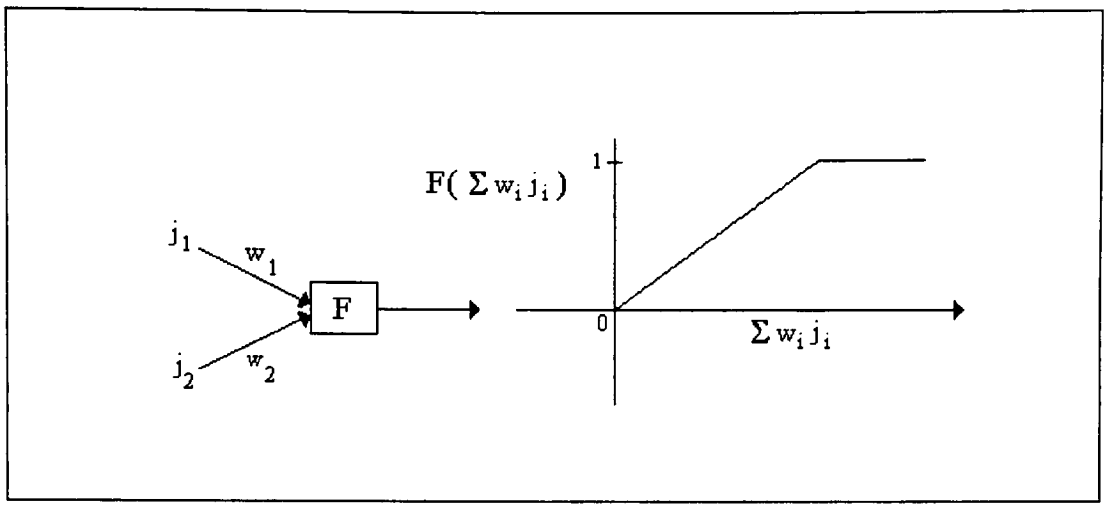

Figure 1-1 : A single processing element of a neural network and the non-linear function used to limit its response to the interval [0, 1].

A trained neural network recalls patterns based on information derived from the associations established between the input and output patterns during training. The activation of a processing element either induces or hinders the activation of the neurons to which it is connected, depending on whether the interconnections are excitatory or inhibitory. Consequently, the response of a processing element indicates the degree of confidence that its associated feature is present in, or absent from the input pattern.

During training and the pattern recognition task, the output of each neuron is passed through a non-linear function called an activation function. This function typically limits the response of the neuron to a particular interval. An example of an activation function that limits the neuron's output to the interval [0,1] is illustrated in Figure 1-1 [2].

1.2 Optical Character Recognition and the Neocognitron

The pattern processing task that this thesis investigates is optical character recognition (OCR). In OCR, a character must be recognized from its image pixels. This task can be fairly

straightforward or extremely difficult, depending on the nature of the image. For images consisting of cleanly printed, fixed-point symbols, simple template matching would suffice. However, when using multiple fonts or when the images are noisy, more sophisticated techniques such as neural networks are required. In handwritten character recognition the images can vary significantly, making this task very difficult even for neural networks. To solve the difficult task of handwritten character recognition, Fukushima designed an artificial neural network called the cognitron [3]. This neural network had the ability for hand-written character recognition, but its accuracy was dependent on the position of the stimulus pattern on the input plane. In essence, the same pattern presented at different positions on the input plane were judged to be different by the cognitron. Several years later, he extended the concept of the cognitron to develop what he called the neocognitron [4]. In its original design, the neocognitron is a real-valued, multi-layered self-organizing network which when trained, recognizes visual patterns with a high degree of accuracy. Unlike the cognitron, the neocognitron is not affected to some degree by deformations, scaling or shifting of the input patterns.

1.3 Digital vs. Analog VLSI Artificial Neural Networks

Due to their intrinsic parallelism, the regularity of the processing elements, and the size of typical networks, ANNs are well suited for hardware implementation. Even though a neural network can comprise thousands of processing elements, each element is relatively simple and identical in all stages of the network. Therefore, a single processing element of each type can be constructed and replicated, leaving the greatest construction difficulty to be the interconnections of these components. Unfortunately, the hardware implementation is normally analog and does not take full advantage of the benefits and major advances in VLSI technologies. To do so, ANNs must be adapted to facilitate an all digital VLSI implementation.

Using digital VLSI to implement artificial neural networks offers many advantages over analog implementations. High density circuits with high speeds are possible with analog

implementations. However, analog circuits are susceptible to noise and temperature changes, and unavoidable inter-chip variations make manufacturing functionally equivalent circuits very difficult. In addition, the need for long-term storage for the connection weights requires special fabrication techniques. The largest shortfall however, is that the construction of such systems typically requires several integrated circuits (ICs). In such cases, care must be taken to guarantee the compatibility of the off-chip electrical environments. It is extremely difficult, if not impossible, to match board-level capacitive loads and time constants to their on-chip counterparts. Even though analog VLSI is well suited for parallel processing using locally connected networks, the system interface difficulties make it cumbersome for multi-chip ANNs [5].

In contrast, using digital VLSI to implement ANNs can be much easier because digital circuits are more tolerant of intra-chip and inter-chip variations, and are easily manufactured to be functionally identical. Also several devices such as static RAMs, EPROMs, EEPROMs and ROMs are available to enable long-term weight storage. Most neural networks utilize a convolution-like operation to compute the output of the neurons. Digital VLSI elements have been widely used to perform similar convolution operations, and with the locally connected architecture of the neocognitron, this convolution-like operation suggests construction using time-division multiplexing. I/O bottlenecks may also exist in ANN solutions, but they are better resolved by digital techniques than analog. The use of input buffers, shift registers and pipelining to process the data also offer significant advantages relative to performance and the need for external storage [6], [7].

One concern with digital VLSI implementations of neural networks is the multiplication and division that must be performed on fixed-point digital values. These circuits consume a significant amount of silicon area which has a negative impact on performance. In addition to the multiplication and division operations, the neocognitron also requires square and square-root operations. The digital implementation of the neocognitron overcomes these potential problems by implementing the square and square-root functions

using look-up tables, and by using shift operations to perform multiplication and division. Hence, the neocognitron model implemented is ideal for digital VLSI.

This thesis utilizes the procedure described by White and Elmasry [8] for adapting a neocognitron to a digital neocognitron model. The network implemented consisted of nine stages with the last stage containing ten processing elements representative of the ten digits to be recognized. Unlike, the model described in [4], this neocognitron model utilized supervised training (learning with a teacher) to reinforce the modifiable synapses or weights of the network.

CHAPTER 2

2.0 Theory of the Neocognitron Neural Network

2.1 The Neocognitron Model Functions

Unlike most ANNs which tend to have general purpose architectures, the neocognitron was designed primarily for handwritten character recognition, even though it has been applied to a variety of pattern recognition tasks. The neocognitron is a multi-layered hierarchical network consisting of several layers of neuron-like cells intended to simulate the human visual system. Its architecture is based on research performed by Hubel and Wiesel on the functionality and structure of the visual nervous system [9] - [11]. Figure 2-1 shows the relation between the Hubel and Wiesel model and the neocognitron [4]. Similar to the human visual system, upon completion of training, the neocognitron recognizes patterns with a high degree of generalization relatively independent of shifts in position, changes in size or shape, or the presence of deformations from the learned patterns. The neocognitron consists of a cascaded connection of a number of modular structures preceded by an input plane. The input plane, denoted by u_0, is a two-dimensional photoreceptor array of cells corresponding to the lateral geniculate body (LGB) shown in Figure 2-1. Each of the remaining modular structures in the network is composed of two layers of cells connected in cascade.

The first layer of the module consists of S-cells which correspond to simple cells or lower-order hypercomplex cells. This is called the S-layer, and is denoted by u_{Sl} representing the S-layer in the l^{th} module. The second layer of the module consists of C-cells and correspond to complex or higher-order hypercomplex cells. This is referred to as the C-layer, and the C-layer in the l^{th} module is denoted by u_{Cl}..

The S-cells and C-cells in a layer are alternately arranged into subgroups according to the stimulus features of their receptive field. Each subgroup is a two-dimensional array and is referred to as a cell-plane. Consequently, S-planes and C-planes represent cell planes consisting of S-cells and C-cells, respectively. All the cells in a single cell plane have input

synapses of the same spatial distribution, and only the positions of the presynaptic cells are shifted in parallel from cell to cell. Therefore, all the cells in a single cell plane have receptive fields of the same function, but at different positions. The layer of C-cells at the highest stage of the network, is the recognition layer representing the final result of the pattern recognition task. Figure 2-2 illustrates the architecture of the neocognitron model implemented.

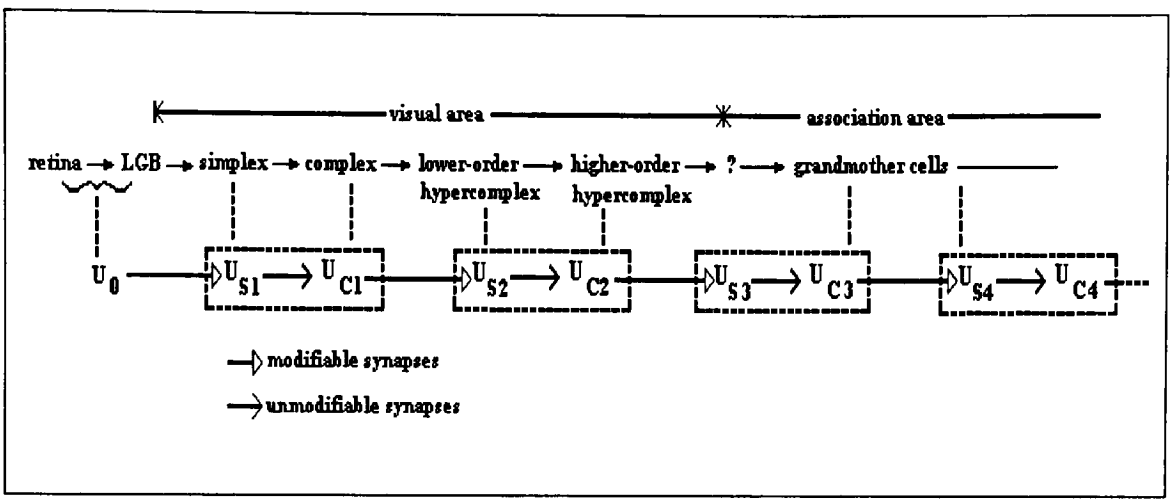

Figure 2-1 : Correspondence between the model by Hubel and Wiesel and the neocognitron neural network.

In Figure 2-2 each of the large tetragons represents an S-layer or a C-layer, and the interior tetragons are S-planes or C-planes, respectively. The labeling below each layer indicates the dimensions of each cell-plane and the number of cell-planes in that layer. For example, 19x19x12 is interpreted as 12 cell-planes of dimension 19x19 pixels. In the neocognitron, S-cells and C-cells receive afferent connections from cells in its receptive field in the layer preceding it. This is indicated by the areas enclosed by the ellipses.

The S-cells of the neocognitron are feature extracting cells. These cells receive input from the preceding layer, detect the presence of specific features, and pass them to the following layer of C-cells. The input connections of the S-cells are variable and are modified during training, while those to the C-cells are unmodifiable and determined by the architecture. Through training, the S-cells come to extract features from the input patterns presented at the input layer. At lower stages of the network, the S-cells extract lines at

different orientations, and at higher stages they extract higher-order features of the training patterns. Eventually, the complete pattern is recognized by the last layer of the network. The layer of C-cells following each S-layer is included in the neocognitron's architecture to allow for variability in the position of the features extracted by the S-layer. Each C-cell receives input from a group of S-cells that extract a specific feature from the stimulus pattern from slightly different positions. At least one cell in this group being active is sufficient to activate the corresponding C-cell. This reduces the dependence on the position of the feature detected, which is further reduced as the number of stages in the network increases.

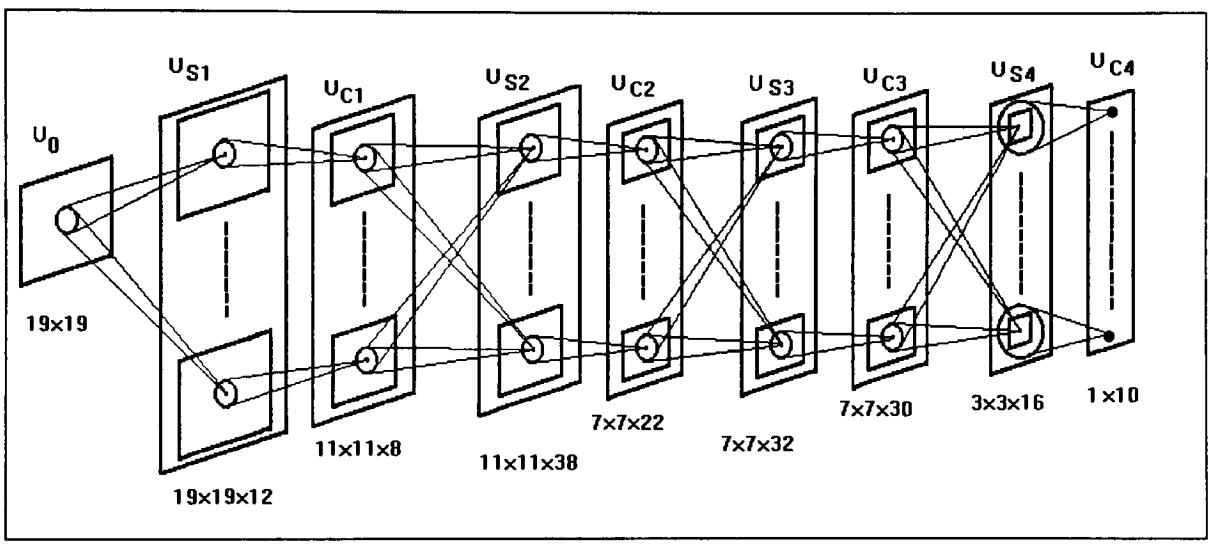

Figure 2-2 : The architectural organization of the neocognitron neural network.

Not shown in Figure 2-2 are the inhibitory Vc-cells and Vs-cells of the neocognitron. These cells are included in the neocognitron to provide a shunting affect to reduce the likelihood of the excitatory S-cells and C-cells responding to irrelevant features.

The output of the four cell types of the neocognitron are determined by the proceeding numerical expressions [8]. In these equations $\boldsymbol{n}=(x,y)$ is the two-dimensional coordinate representing the cell's location, k is the current cell plane, and l represents the current layer. u_0 is the input plane of the neocognitron, and the output of a cell in this plane is denoted by $u_0(\boldsymbol{n})$. u_{Sl} and u_{Cl} represents the S-layer and C-layer, respectively, in the l^{th} module of the

network. However, in layer 1, u_{Cl} is equivalent to u_0, the input plane. The corresponding inhibitory cells, Vc-cells and Vs-cells are denoted by v_{Cl} and v_{Sl}, respectively. The output of an S-cell in the k^{th} S-plane of the l^{th} module is described mathematically as follows:

$$u_{Sl}(k,n) = r_l \cdot \Phi\left[\frac{1+\sum_{K_l=1}^{K_{Cl-1}}\sum_{v\in A_l} a_l(K,v,k)\cdot u_{Cl-1}(K,n+v)}{1+\frac{r_l}{1+r_l}\cdot b_l(k)\cdot v_{Cl}(n)} - 1\right], \tag{1}$$

where

$$\Phi[x] = \begin{cases} x & (x \geq 0) \\ 0 & (x < 0) \end{cases} \tag{2}$$

In Equation (1), $a_l(K,v,k)$ and $b_l(k)$ represents the strength of the variable excitatory and inhibitory connections from the preceding stage. During training the modifiable weights associated with an excitatory S-cell are reinforced only when its output is larger than the neighboring cells in its competitive region. When this occurs, the excitatory synapses $a_l(K,v,k)$ and inhibitory synapses $b_l(k)$ which are afferent to the S-cells of the $\hat{k}^{th}$ S-plane are reinforced by

$$\Delta a_l\left(K,v,\hat{k}\right) = q_l \cdot c_l(v)\cdot u_{Cl-1}\left(K,\hat{n}+v\right), \tag{3}$$

and

$$\Delta b_l\left(\hat{k}\right) = q_l \cdot v_{Cl-1}\left(\hat{n}\right), \tag{4}$$

where

$c_l(v)$ represents the efficiency of the unmodifiable excitatory weights and is a monotonically decreasing function with respect to $|v|$, and

q_l is a positive constant that determines the speed at which the excitatory and inhibitory synapses are reinforced. This parameter is commonly referred to as the learning rate.

This procedure implements a form of Hebbian learning that was first introduced in [3] to train the original cognitron. The representative S-cell selected for reinforcement is denoted by $u_{Sl}(\hat{k}, \hat{n})$, where $\hat{k}$ is the cell plane being trained, and $\hat{n}$ is the two-dimensional coordinate representing the cell's location on the plane. Another important parameter in Equation (1) is the selectivity parameter, r_l, which controls the intensity of the inhibition. A large value for r_l makes the S-cells' response more selective to a specific feature or pattern, and endows the network with a high ability to discriminate between patterns of different classes. However, this increase in selectivity also reduces the network's ability for deformation independent pattern recognition. Therefore, the value of r_l must be chosen at a point of compromise between these two contradictions. A detailed discussion on selecting values for r_l can be found in [12] and [13].

To facilitate the approximations required for the digital neocognitron model, modification of the neocognitron's inhibitory Vc-cell output was required. In the neocognitron described by Fukushima, the $c_l(v)$ fixed weights which are usually taken as two-a dimensional Gaussian such that

$$K_{Cl} \cdot \sum_{v \in A_l} c_l(v) = 1, \tag{5}$$

was changed so that

$$c_l(0) = 1. \tag{6}$$

This change in the normalization provides a consistent range of values for the $c_l(v)$ fixed weights but it requires division by the value *csum* in the Vc-cell output equation. This ensures that the Vc-cell output remains unaffected by the normalization, and that the output of a Vc-cell is consistent with the original neocognitron. The new Vc-cell output equation of the neocognitron model is a weighted root-mean-square average of the preceding level C-cells' output over the same connection area A_l, and is given by

$$v_{Cl}(n) = \sqrt{\frac{1}{csum} \sum_{K=1}^{K_{Cl-1}} \sum_{v \in A_l} c_l(n) \cdot u_{Cl-1}^2(K, n+v)}\ , \tag{7}$$

where

$$csum = \sum_{K=1}^{K_{Cl-1}} \sum_{v \in A_l} c_l(v). \tag{8}$$

With the normalization requirement of Equation (6), the first learning rule shown in Equation (3) becomes

$$\Delta a_l\left(K, v, \hat{k}\right) = q_l \cdot \frac{c_l(v)}{csum} \cdot u_{Cl-1}\left(K, \hat{n} + v\right). \tag{9}$$

The excitatory C-cells of the neocognitron have a shunting-type inhibitory input, $v_{Sl}(n)$, similar to S-cells, but their outputs show a saturation characteristic. The output of a C-cell in the k^{th} C-plane of the l^{th} module is given by

$$u_{Cl}(k, n) = \Psi\left[\frac{1 + \sum_{K=1}^{K_{Sl}} j_l(K, k) \sum_{v \in D_l} d_l(v) \cdot u_{Sl}(k, n+v)}{1 + v_{Sl}(n)} - 1\right], \tag{10}$$

where $\Psi[\ ]$ is the threshold characteristic of the C-cell defined by

$$\Psi[x] = \begin{cases} \dfrac{x}{\alpha_l + x} & (x \geq 0), \\ 0 & (x < 0). \end{cases} \tag{11}$$

This function limits the response of the C-cells to the interval [0,1]. The $d_l(v)$ term in Equation (10) is a two-dimensional Gaussian similar to $c_l(v)$, and denotes the strength of the fixed excitatory connections. The normalization change of Equation (6) is also applicable to the $d_l(v)$ fixed weights. The term $j_l(K,k)$ is used to indicated how the S-planes which extract similar features are combined for input to a C-plane. $j_l(K,k)$ takes on a value of one or zero depending on whether or not the synaptic connections really exist from the K^{th} S-plane to the K^{th} C-plane. Finally, the parameter α_l in Equation (11) is a positive constant which specifies the degree of saturation of the C-cells.

In Equation (10), $v_{Sl}(\boldsymbol{n})$ is the output of the inhibitory Vs-cells defined as a weighted average of the S-cells' output of the preceding stage within the receptive field of $\boldsymbol{n}$. The Vs-cell provides a form of lateral inhibition and are optional in the neocognitron. Typically, only the final layer in the neocognitron contains Vs-cells. The output of a Vs-cell is given by the following equation:

$$v_{Sl}(\boldsymbol{n}) = \frac{1}{K_{Sl}} \sum_{K=1}^{K_{Cl-1}} \sum_{\boldsymbol{v} \in A_l} d_l(\boldsymbol{v}) u_{Sl}(K, \boldsymbol{n} + \boldsymbol{v}) \tag{12}$$

2.2 The Digi-Neocognitron Model Functions

To implement the neocognitron neural network using digital VLSI, the model functions and fixed weight values described in the previous section were modified. This resulted in a new neural network model that the authors of [8] called the digi-neocognitron (DNC). In the DNC model, multiplications and divisions are replaced with shift operations by converting the multiplication and division factors to powers of 2. This is a significant benefit since shifters are easily implemented in digital hardware; as such, they require less silicon area and reduce the propagation delays associated with multipliers. The complex functions, square and square root, of the neocognitron are replaced by look-up tables, and can be implemented with simple combinatorial logic or memory arrays.

To understand the DNC model functions, a discussion on performing power of 2 representations and the notations used in these functions is necessary. The DNC model uses a single-term power of 2 approximation in all cases except one, for which a two-term approximation is required. The notation used to describe the DNC model functions is similar to that used in [14], with modifications to accommodate positive values and fractions.

The set of admissible values in a single-term power of 2 representation is denoted as $P_{m,n} = \{2^m, 2^{m+1}, \ldots, 2^{n-1}, 2^n\}$ where m and n are integers and $m \leq n$. Also,

$$\langle x \rangle_{2^m, 2^n; b} \tag{13}$$

denotes rounding x to the nearest term in the representation $P_{m,n}$ with bias $b \in [0,1]$ to rounding up. If the value of x is outside the range of the representation, the appropriate endpoint is chosen as the approximation. A bias of 0.0 selects the power of 2 that is less than or equal to the number; a bias of 1.0 selects that which is greater than or equal, and a bias of 0.5 selects the closest power of 2. These representations are denoted by the following set of equations.

$$\langle x \rangle_{2^m,2^n:le} = \langle x \rangle_{2^m,2^n:0.0} \qquad \langle x \rangle_{2^m,2^n:ge} = \langle x \rangle_{2^m,2^n:1.0} \tag{14}$$

$$\langle x \rangle_{2^m,2^n} = \langle x \rangle_{2^m,2^n:0.5}$$

An extension of the concepts for a single-term power of 2 representation is used to define the set of possible values for the two-term power of 2 approximations. The representation used is

$$P^2_{m,n} = \left\{ y | y = 2^p + s \cdot 2^k, y > 0 \right\}$$

where m and n are integers, with $m \leq n$, p, $k \in \{m, m+1, \ldots, n-1, n\}$ and $s \in \{-1, 0, 1\}$ The rounding used to determine the nearest value for x in the representation $P^2_{m,n}$ is

$$\langle\langle x \rangle\rangle_{2^m,2^n} \tag{15}$$

with the usage of the bias equivalent to that for the single-term approximation.

Following is a summary of the preprocessing performed on the neocognitron model functions to obtain the output functions for the digi-neocognitron model. A detailed discussion of the steps described below can be found in [8]. The bit ranges of the inputs and outputs of the four cell types are represented with 4 bits to the right of the binary point. In cases where the output of a cell is more than four bits it is truncated. This restriction also requires a change in the lower level C-cells or the input layer. In the input pattern, each pixel is represented by four bits to the right of the binary point, resulting in 0.0000 (0.0) for a zero pixel and 0.1111 (0.9375) for a one pixel, where zero indicates a white pixel and one a black pixel. Also, the s-cell's output is represented as seven bits (with 4 binary places) allowing outputs up to 8. This provides a safety margin because the digi-neocognitron model

approximation of the modifiable weights as powers of 2 can result in S-cell outputs greater than 1 if the excitatory weights are rounded up, and the inhibitory weights are rounded down. In the model implemented, the S-cell's output was always less than 2.

Table 2-1 : Preprocessing approximations for the fixed weight connections.

Fixed Weight c_l, d_l **Interval**	**Fixed Weight** $\overline{c_l}, \overline{d_l}$ **Approximation**	**Interpretation**
[0.0, 0.1)	0	Force 0
[0.1, 0.4)	1/4	Shift 2 R
[0.4, 0.75)	1/2	Shift 1 R
[0.75, 1.0]	1	Shift 0

To obtain the output of the inhibitory Vc-cell, the normalization change of Equation (6) is required. This change provides a consistent range of values for the c_l fixed weights to enable approximations by values restricted to zero and powers of 2. These approximations shown in Table 2-1 are used to replace the multiplication by a shift operation in the Vc-cell output function. With these approximations, Equation (8) now becomes

$$C_\Sigma = \sum_{\kappa=1}^{K_{Cl-1}} \sum_{v \in A_l} \overline{c_l(v)}, \tag{16}$$

where the overbar indicates a power of 2 approximation. The division in Equation (7) is then replaced by a shift operation using a power of 2 approximation of Equation (16) according to the following rules:

$$\overline{C_\Sigma} = \begin{cases} \langle C_\Sigma \rangle_{1,2048,le,} & l = 1 \\ \langle C_\Sigma \rangle_{1,2048,ge,} & l > 1 \end{cases} \tag{17}$$

where l represents the layers of the network.

Table 2-2 : Power of 2 approximation for squaring the output of a c-cell.

Input $\overline{u_C}$ (binary)	Input $\overline{u_C}$ (decimal)	Power of 2 ($\overline{u_C}$) Shift Control
0000	0.0000	Force 0
0001	0.0625	Force 0
0010	0.1250	Force 0
0011	0.1875	Shift 2 R
0100	0.2500	Shift 2 R
0101	0.3125	Shift 1 R
0110	0.3750	Shift 1 R
0111	0.4375	Shift 1 R
1000	0.5000	Shift 1 R
1001	0.5625	Shift 0
1010	0.6250	Shift 0
1011	0.6875	Shift 0
1100	0.7500	Shift 0
1101	0.8125	Shift 0
1110	0.8750	Shift 0
1111	0.9375	Shift 0

With the preprocessing approximations described above, the Vc-cell output of the DNC model is given by

$$\overline{v_{Cl}(n)} = \sqrt{\frac{1}{\overline{C_\Sigma}} \sum_{K=1}^{K_{Cl-1}} \sum_{v \in A_l} \overline{c_l(v)} \cdot \overline{u_{Cl-1}(K, n+v)}^2} \tag{18}$$

In this equation, the square of the C-cell output is obtained by using a power of 2 approximation of $\overline{u_C}$, as shown in Table 2-2, to control the shift operation eliminating the need for a multiplier. A shifter is used here as opposed to a look-up table because the square operation is done for every term in the summation and can be calculated in parallel using some form of pipelining. Also, in this instance a shifter requires less area than a look-up table.

Table 2-3 : Square root approximation for the output of a Vc-cell.

Input (decimal)	**Input (binary)**	**Output (binary)**	**Output (decimal)**
0.0000	0.0000	0.0011	0.1875
0.0625	0.0001	0.0100	0.2500
0.1250	0.0010	0.0110	0.3750
0.1875	0.0011	0.0111	0.4375
0.2500	0.0100	0.1000	0.5000
0.3125	0.0101	0.1001	0.5625
0.3750	0.0110	0.1010	0.6250
0.4375	0.0111	0.1011	0.6875
0.5000	0.1000	0.1011	0.6875
0.5625	0.1001	0.1100	0.7500
0.6250	0.1010	0.1101	0.8125
0.6875	0.1011	0.1101	0.8125
0.7500	0.1100	0.1110	0.8750
0.8125	0.1101	0.1110	0.8750
0.8750	0.1110	0.1111	0.9375
0.9375	0.1111	0.1111	0.9375

However, because the square root is computed once for each Vc-cell, it is implemented using the look-up table shown in Table 2-3.

To obtain the S-cell output function for the DNC model, the modifiable weights $a_l(K,v,k)$ and $b_l(k)$ are approximated by powers of 2. To improve alignment with the power of 2 representation, it is suggested that these weights be scaled by a factor f_l as shown in the equations below. The weight factor $r_l \cdot b_l(k)/(1+r_l)$ from Equation (1) is approximated by one or two power of 2 terms defined by

$$\overline{r_l \cdot b_l(k)/(1+r_l)} = \langle f_l \cdot r_l \cdot b_l(k)/(1+r_l) \rangle_{1,64} \tag{19}$$

or

$$\overline{r_l \cdot b_l(k)/(1+r_l)} = \langle\langle f_l \cdot r \cdot_l b_l(k)/(1+r_l) \rangle\rangle_{1,64}. \tag{20}$$

At lower levels, a single-term approximation is sufficient, however, at higher levels, a two-term power of 2 approximation is required to avoid significant loss of precision due to rounding. The approximation for the a_l weights is done after scaling by f_l using the following equation:

$$\overline{a_l} = \begin{cases} \langle f_l \cdot a_l \rangle_{1/8,8} & f_l \cdot a_l \geq \dfrac{1}{16} \\ 0 & \textit{Otherwise.} \end{cases} \tag{21}$$

r_l, which is multiplied by the function $\Phi[x]$ in Equation (1), is also represented by a power of 2 approximation. This is obtained using

$$\overline{r_l} = \langle r_l \rangle_{1,16,2/3}\ , \tag{22}$$

which selects the closest power of 2 representation in the range, but rounds up if r_l is in the upper two-thirds of the range and down otherwise. Applying these approximations to Equation (1), the digi-neocognitron's S-cell output is given by

$$\overline{u_{Sl}(k,\boldsymbol{n})} = \overline{r_l} \cdot \Phi\left[\frac{E - IC}{1 + IC}\right] \tag{23}$$

where

$$\Phi\left[x\right] = \begin{cases} x & (x \geq 0) \\ 0 & (x < 0)\ . \end{cases}$$

The excitatory term E, is given by

$$E = \sum_{K=1}^{K_{Cl-1}} \sum_{\nu \in A_l} \overline{a_l(K,\nu,k)} \cdot \overline{u_{Cl-1}(K,\boldsymbol{n}+\nu)}, \tag{24}$$

and the inhibitory term IC, is given by

$$IC = \overline{r_l \cdot b_l(k)/(1 + r_l) \cdot v_{Cl}(\boldsymbol{n})}. \tag{25}$$

The function $1/(1+IC)$ in Equation (23) is approximated by the function $1/IC2$ where $IC2$ is a power of 2 as defined in Table 2-4. The resulting S-cell output function is

$$\overline{u_{Sl}(k,n)} = \begin{cases} \bar{r}_l \cdot \left[\dfrac{E-IC}{IC2}\right] & E > IC \\ 0 & Otherwise. \end{cases} \tag{26}$$

The output of the optional Vs-cells, which provides a form of lateral inhibition to the C-cells, is given by

$$\overline{v_{Sl}(n)} = \frac{1}{K_{Sl}} \sum_{K=1}^{K_{Sl}} \sum_{v \in D_l} \overline{d_l(v)} \cdot \overline{u_{Sl}(K, n+v)}. \tag{27}$$

Table 2-4 : Power of 2 approximations for 1/(1+I) inhibit function for the S-cells and C-cells.

IC, IS Interval	**IC2, IS2 Approximation**	**Interpretation**
< 0.5	1	Shift 0
[0.5, 2.0)	2	Shift 1 R
[2.0, 4.5)	4	Shift 2 R
[4.5, 10.0)	8	Shift 3 R
[10.0, 21.0)	16	Shift 4 R
≥ 21.0	32	Shift 5 R

As with the $c_l(v)$ modifiable weights, the $d_l(v)$ fixed weights are two-dimensional Gaussian and use the same normalization, $d_l(0) = 1$. The approximation for $d_l(v)$ is identical to that used for $c_l(v)$ as shown in Table 2-1. K_{Sl} represents the number of S-planes in the previous level, and is approximated using

$$\overline{K_{Sl}} = \langle K_{Sl} \rangle_{1,64;le} \tag{28}$$

so that the division $1/K_{Sl}$ can be replaced by a shift operation.

The power of 2 approximation required for the C-cell output is performed on the $d_l(v)$ fixed weights as discussed above, and on the saturation parameter α_l by

$$\overline{\alpha_l} = \langle \alpha_l \rangle_{1/32,1;le}. \tag{29}$$

The resulting C-cell output equation is

$$\overline{u_{Cl}(k,n)} = \Psi\left[\frac{E - IS}{1 + IS}\right], \tag{30}$$

where the excitatory term E, is defined as

$$E = \sum_{K=1}^{K_{Sl}} j_l(K,k) \sum_{v \in D_l} \overline{d_l(v)} \cdot \overline{u_{Sl}(k, n+v)}, \tag{31}$$

and the inhibitory term IS, is given by

$$IS = \overline{v_{Sl}(n)}. \tag{32}$$

Similar to the S-cell output, the function $1/(1 + IS)$ in Equation (30) is approximated by the function $1/IS2$, where $IS2$ is a power of 2 as defined in Table 2-4. This results in the following output equation for the C-cells:

$$\overline{u_{Cl}(k,n)} = \begin{cases} \Psi\left[\dfrac{E - IS}{IS2}\right] & E > IS \\ 0 & Otherwise. \end{cases} \tag{33}$$

The final output of the C-cells is obtained by approximating the saturation function $\Psi[x]$, which is implemented using the look-up table shown in Table 2-5. This function is redefined for the digi-neocognitron model as

$$\overline{\Psi[x]} = \frac{z}{1+z}, \quad z = x/\overline{\alpha_l}. \tag{34}$$

Table 2-5 : Z/(1+Z) look-up table for the output of a C-cell.

Input (decimal)	**Input (binary)**	**Output (binary)**	**Output (decimal)**
0.00	00.00	0.0000	0.0000
0.25	00.01	0.0011	0.1875
0.50	00.10	0.0101	0.3125
0.75	00.11	0.0111	0.4375
1.00	01.00	0.1000	0.5000
1.25	01.01	0.1001	0.5625
1.50	01.10	0.1010	0.6250
1.75	01.11	0.1010	0.6250
2.00	10.00	0.1011	0.6875
2.25	10.01	0.1011	0.6875
2.50	10.10	0.1011	0.6875
2.75	10.11	0.1100	0.7500
3.00	11.00	0.1100	0.7500
3.25	11.01	0.1100	0.7500
3.50	11.10	0.1100	0.7500
3.75	11.11	0.1101	0.8125
>	>	0.1111	0.9375

CHAPTER 3

3.0 Software Models for the Neocognitron and the Digi-Neocognitron

Prior to designing the digital VLSI version of the neocognitron neural network model, some preliminary steps were required. The basic neocognitron model was constructed, trained, simulated and subsequently used to develop the digi-neocognitron. This also provides a comparative tool for the DNC model, which is necessary to determine if the pattern recognition capability of the digi-neocognitron is similar to that of the neocognitron. Constructing a DNC model is a viable alternative to an neocognitron model only if its capability for pattern recognition is similar to that of the neocognitron model. Both the neocognitron and digi-neocognitron were implemented using the C-programming language on a SUN SPARCStation 20.

3.1 Converting from a Neocognitron to a Digi-Neocognitron Model

To obtain the digital version of the neocognitron with similar pattern recognition capability, White and Elmasry developed a procedure for converting a neocognitron model to a digi-neocognitron [8]. However, this method utilizes the inherent fault tolerant characteristics of the neocognitron, and its unsupervised learning capability to compensate for inaccuracies that result from implementing the required approximations. Because the model implemented utilized supervised learning, some modifications to this methodology were required. The procedure developed consists of the following steps.

1) Do unconstrained simulation, including learning, with the neocognitron model in order to get something that works for the particular problem. If there is not already an existing neocognitron model also incorporate step 2.

2) Change the neocognitron model to use the approximate fixed weights in Table 2-1. If feasible, also use appropriate powers of 2 for the selectivity (r_l) and saturation (α_l) parameters. This recognizes that some of the DNC model preprocessing approximations can be treated as just another set of

neocognitron model parameters. Repeat simulation, including learning, with the neocognitron model and adjust the learning rate (q_l) to develop weights that are appropriate for the DNC representations, as in Equations (19), (20) and (21).

3) Preprocess level 1 to create level 1 DNC model approximations.

4) Run the DNC model simulator on level 1 for all input patterns in the training set and compare with level 1 neocognitron model outputs.

5) If necessary, repeat steps 2, 3, and 4 in order to obtain good weight approximations, trying different scaling factors, changing the amount of training, or adjusting the cell's inhibitory input for each S-plane.

6) Use the level 1 outputs from the DNC model to retrain level 2 of step 2 in the neocognitron Model. This is a critical because the level 2 feature detectors must be developed with inputs from the lower levels that are representative of what will be seen in the DNC model. Otherwise, there will not be enough similarity between lower-level outputs and higher-level feature detectors to have a robust conversion of the neocognitron model pattern recognition capabilities to the DNC model.

7) Preprocess level 2 to create level 2 DNC model approximations.

8) Repeat this process for all levels in the architecture; e.g., if there are three or more levels, next run the DNC simulator on level 2, and use these outputs to retrain level 3 in the neocognitron model.

9) Use the DNC model to check final level outputs for all input patterns using all levels in the neural network.

3.2 Implementation of the Neocognitron Model

The neocognitron model constructed consisted of four levels of S-cell and C-cell planes, as shown in Figure 2-2. Also shown at the bottom of Figure 2-2 are the dimensions of the cell planes in each layer. Though not shown, each layer contains a Vc-cell plane, which provides inhibitory input to the S-cells. In addition, the last layer of the network which represents the patterns to be recognized, contains a Vs-cell plane which provides a form of lateral inhibition to the C-cells. The dimensions of these inhibitory planes are identical to the

excitatory planes that receive their outputs. This network is similar to those described in [9] and [10] with some modifications to the architecture and parameters.

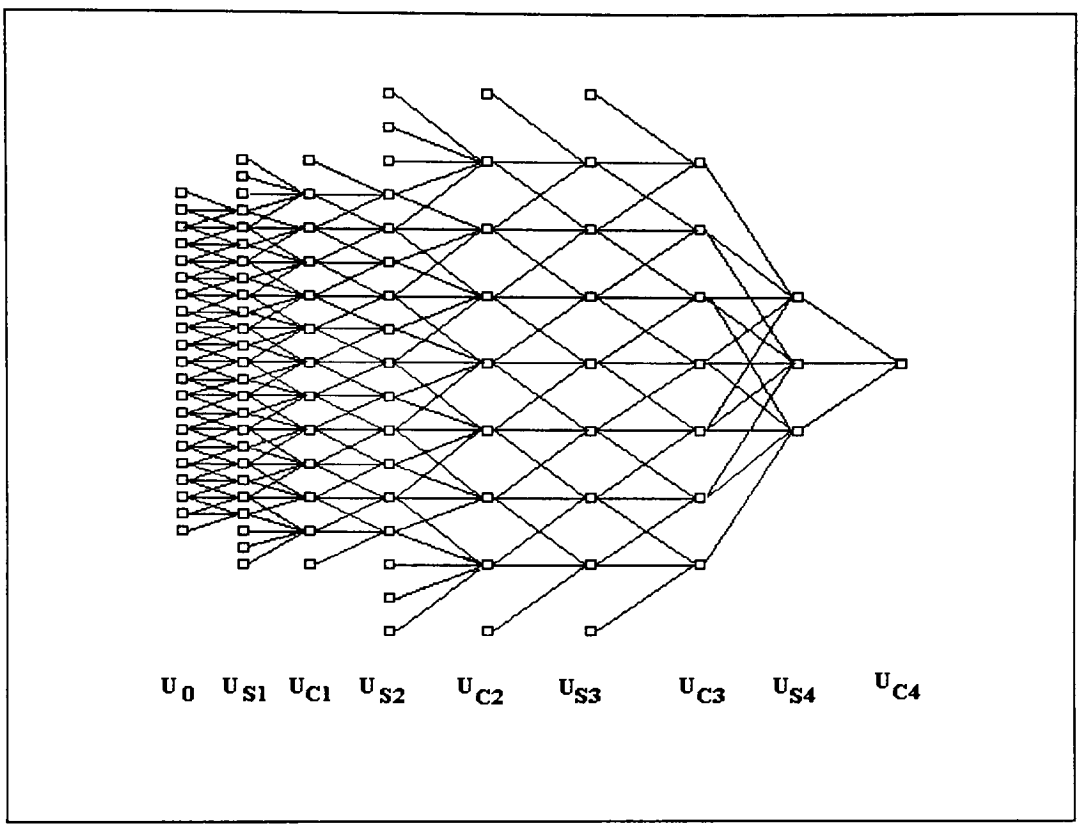

Figure 3-1 : A one-dimensional view of the interconnections between the cells of the different layers in the neocognitron.

As shown in Figure 2-2, the neocognitron is interconnected in a cascaded manner in which the size of the receptive field of a cell is dependent on the layer in which the cell is located. The deeper the cell is in the network, the larger is the cell's receptive field. Consequently, the density of the cells in each plane decreases as the size of the receptive field increases. Figure 3-1 shows a one-dimensional view of how the cell planes of each layer are inter-connected. Only a single plane is shown for each layer since the connection scheme is identical for all planes in the layer. Figure 3-1 also indicates the size of the receptive field of each layer. This connection scheme must be followed if the neocognitron is to be effective in its pattern recognition capability. The thinning out of the cell planes as the number of stages

increases is also necessary. This is required since the information in adjacent cells is almost identical due to smearing of the S-cell outputs through the receptive field averaging.

To calculate the output of a neuron, a non-linear convolution-like operation is performed on two 3x3 or 5x5 masks. However, edge pixels have a connection area that is smaller than the mask size of the remaining pixels in the plane thus complicating this convolution operation. Therefore, care must be taken when computing the output of pixels on the edge of a cell plane to avoid loosing any feature detector outputs. To resolve this dilemma, some planes are oversized to ensure that all cells in the plane have identical size receptive fields. The outputs of these cells are always zero, therefore no computational time is required. In Figure 3-1, these cells do not receive inputs from the preceding plane.

In a software implementation, the cost of oversizing the cell planes is negligible. However, in a hardware implementation, a trade-off between the cost of oversizing, and the cost of the additional circuitry that would be required to determine a cell's receptive field must be considered.

In performing the convolution operation, one of three different two-dimensional masks is required, an $\boldsymbol{a}$, $\boldsymbol{c}$ or $\boldsymbol{d}$ mask. A mask is associated with each cell in the plane and, if implemented in this manner, would require a substantial amount of memory. However, the cells of each plane all extract the same feature from the input pattern and have an identical set of connections. This can be easily exploited in a software simulation to reduce memory requirements. In the model constructed, a different $\boldsymbol{a}$ mask and a common $\boldsymbol{c}$ mask is used for each plane in an S-layer. For example, in layer u_{S1} there are 12 planes and one in the input plane u_0. Correspondingly, there are 12 different $\boldsymbol{a}$ masks and one $\boldsymbol{c}$ mask. A single $\boldsymbol{c}$ mask is required in all layers, but the number of $\boldsymbol{a}$ masks depends on the number of S-planes in the current stage and the number of C-planes in the preceding stage. Layer u_{S2} consists of 38 planes and u_{C1} contains 8 planes; therefore, the number of distinct $\boldsymbol{a}$ masks required is 38x8. Also, a $\boldsymbol{d}$ mask is associated with each C-plane and S-plane pair, but each mask is identical. Therefore, like the $\boldsymbol{c}$ mask, a single $\boldsymbol{d}$ mask is used for all cell planes in each C-layer.

Critical to the training and the performance of the neocognitron is the selection of the values for its parameters. The learning rate q_l, the saturation parameter, α_l, the selectivity parameter r_l, and the fixed excitatory weights, $c_l(v)$ and $d_l(v)$, must be chosen carefully to provide an acceptable balance between the training time, and the neocognitron's pattern recognition capability.

The learning rate q_l, determines the amount of reinforcement of the modifiable synapses $b_l(k)$ and $a_l(K,v,k)$. For unsupervised training, the value chosen for the learning rate is paramount if the network is to be trained successfully. However, for supervised training as used in this implementation, q_l is often chosen sufficiently large so that the training of each layer can be completed in a single iteration. Since the primary purpose of constructing the neocognitron is to facilitate the development of the digi-neocognitron, the learning rate for each layer is chosen so that the final weight values will be similar to those of the digi-neocognitron. The values utilized for the learning rate were $q_1 = 45.0$, $q_2 = 175.0$, $q_3 = 285.0$ and $q_4 = 625.0$.

The values chosen for the saturation parameter, α_l, were identical to those used for the neocognitron described in [13]. The model implemented here was based on that described by the authors of [13]. Therefore, these values were an appropriate starting point, which proved adequate for this implementation. The values used were $\alpha_1 = \alpha_2 = \alpha_3 = 0.25$ and $\alpha_4 = 1.0$.

The selectivity parameter was chosen as follows: $r_1 = 1.7$, $r_2 = 5.0$, $r_3 = 1.5$ and $r_4 = 1.0$. This value determines the efficiency of the inhibitory inputs to the S-cells and controls the selectivity during feature extraction. $r_1 = 1.7$ is chosen for u_{S1} because the connection area for this layer is 3x3. The S-cell will yield a non-zero output when the stimulus feature detected contains up to two additive elements of noise, or one additive and one subtractive element. However, the cell yields a zero output if the stimulus feature has two or more subtractive elements of noise. With $r_2 = 5.0$ the S-cells' output will be non-zero with up to one additive or one subtractive element of noise, and zero with both. Such selectivity is necessary because

the training patterns are specific portions of the 10 numerals to be recognized. In layers 3 and 4 the selectivity parameter is chosen to be $r_3 = 1.5$ and $r_4 = 1.0$. The s-cells' selectivity should not be too large, since these layers are trained on complete patterns and must be more tolerant of deformations.

The exact values chosen for the fixed weights, $c_l(v)$ and $d_l(v)$, are not important as long as Equation (6) is satisfied. The values for $c_l(v)$ is given by

$$c_l(v) = \gamma_l^{|v|} \tag{35}$$

where $\gamma_1 = \gamma_2 = \gamma_3 = 0.7$ and $\gamma_4 = 0.6$. $d_l(v)$ is given by

$$d_l(v) = \overline{\delta_l} \cdot \delta_l^{|v|} \tag{36}$$

where $\overline{\delta_1} = \overline{\delta_2} = 1.0$, $\overline{\delta_3} = 0.6$ and $\overline{\delta_4} = 0.3$; $\delta_1 = 0.7$, $\delta_2 = 0.6$, $\delta_3 = 0.5$ and $\delta_4 = 0.8$. Both equations are taken from [15] but are used with different parameters since the patterns to be recognized and the architecture of this implementation differs from that described in [15]. In Equations (35) and (36), $|v|$ is the normalized distance between the cell located at the position v and the center of the receptive field.

The software for neocognitron's model functions are shown in Appendix A-1. These functions utilizes two routines that are not included in the Appendix. The first, *getCfield()*, is utilized by the Vc-cell and S-cell output functions to determine the cell's receptive field on the C-planes in the preceding stage. Similarly, the function *getSfield()* determines the receptive field of a Vs-cell and C-cell on the S-planes in the preceding stage.

3.2.1 Training the Neocognitron

The neocognitron can learn to recognize a set of patterns via unsupervised or supervised training. The latter training methodology was used to train the software model constructed. Training of the neocognitron was performed layer by layer, beginning at the lowest level and proceeding to the highest level. A training pattern was presented to the input layer and the layer to be trained was selected. The inhibitory Vc-cell outputs are first

computed since they are required in the S-cell computation. After the outputs of all S-planes are calculated, the weight updates to the modifiable inhibitory and excitatory connections are made according to Equations (4) and (9), respectively. The cell at the center of each cell plane is always selected as the representative, $u_{Sl}(\hat{k}, \hat{n})$, for that plane. This requires that the training pattern be centered on the input plane to ensure that the complete pattern is within the receptive field of the representative cell. It is necessary to train only a single cell in each plane since all the cells in a given plane have the same receptive field and extract the same feature from the input pattern. For layers other than the first this training procedure is also used. However, in those layers it is necessary to calculate the output of all preceding S-layers and C-layers even though no updates to the weights are performed. This is possible since these layers have already been trained and will recognize those features, for which they were trained, that are present in the current training pattern.

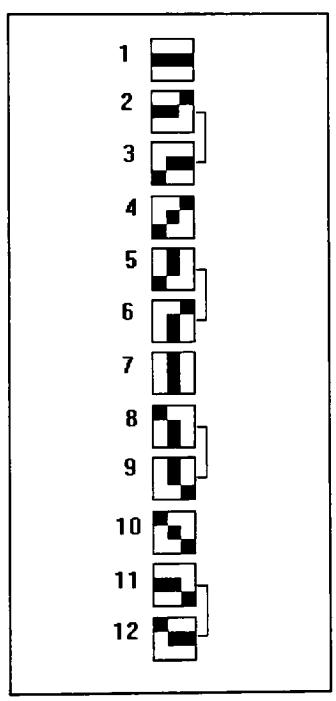

Figure 3-2 : Patterns used to train the 12 cell planes of layer u_{Sl}. How these planes are combined at the input stage of layer u_{Cl} is indicated on the right of the training patterns.

The u_0 input plane is oversized to 21x21 as a result of the 3x3 receptive field of the cell in layer u_{S1}. This photoreceptor array is the input to the 12-25x25 S-planes in layer u_{S1}. The cell planes in this stage are oversized to accommodate the 5x5 receptive field of the cells in the next stage. Shown in Figure 3-2 are the 12 patterns used to train layer u_{S1}. These 12 S-planes, with a 3x3 connection area, extracts line components at different orientations from the input pattern. Due to the similarity between some of these lines, they are combined at layer u_{C1} thus requiring only 8 C-planes in this stage. These 8 cell planes are thinned out from the previous stage to 13x13 and have a 5x5 connection area. In this and the remaining stages of the network, the value $j_l(K,k)$ in Equation (10) indicates how the S-planes are combined for input to the C-planes. K indicates the current C-plane, and k is a one if the S-planes are to be joined and zero otherwise.

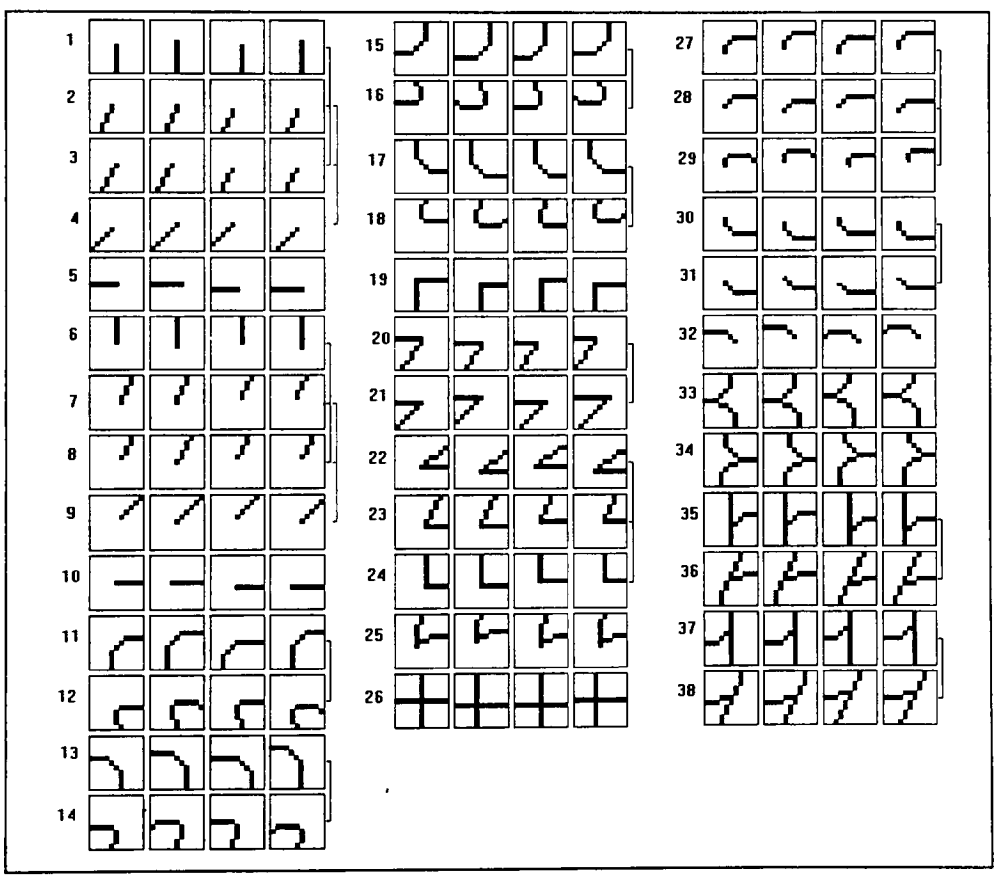

Figure 3-3 : Patterns used to train the 38 cell planes of layer u_{S2}. How these planes are combined at the input stage of layer u_{C2} is indicated on the right of training patterns.

There are 38-17x17 S-planes in stage u_{S2} which receive input from the 8 C-planes in layer u_{C1} through a 3x3 connection area. Thinning out is not performed in this stage but, all cell planes are oversized. Figure 3-3 illustrates the 38 sets of training patterns used to train this stage. Due to the 2-to-1 thinning of the cells in layer u_{C2} from u_{S1}, four patterns are used to train each S-plane at this stage. Each training set consists of the identical pattern presented at four different locations, by shifting it both vertically and horizontally on the input plane. Training each cell plane using four patterns was necessary to ensure each plane's ability to extract its assigned feature. How these 38 cell planes are combined for input to the 22 cell planes in layer u_{C2} is indicated to the right of the training patterns in Figure 3-3.

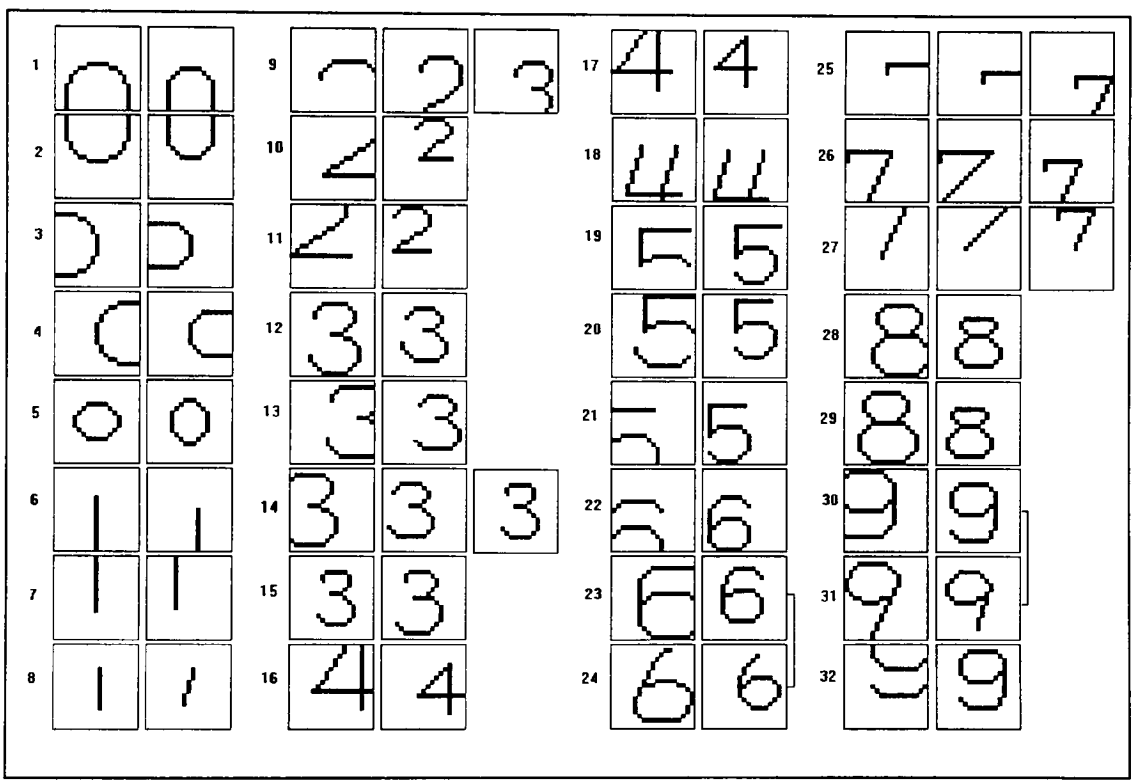

Figure 3-4 : Patterns used to train the 32 cell planes in layer u_{S3}. How these planes are combined at the input stage of layer u_{C3} is indicated on the right of the training patterns.

At the third stage of the neocognitron, the 32-9x9 S-planes in layer u_{S3} receive input from the 22 C-planes in layer u_{C2} through a 3x3 connection area. The training patterns for the 32 S-planes in this stage are shown in Figure 3-4. This training set consists of partial patterns

and some of the actual patterns that are to be recognized by the network. Two or three patterns of varying size, shape and deformations are used to train the 32 planes in this layer. These 32 S-planes are combined as indicated to the right of the cell planes in Figure 3-4 for input to the 30 u_{C3} planes. Thinning out of the cells in the planes from those in the previous layer is not performed at this stage but each plane is oversized.

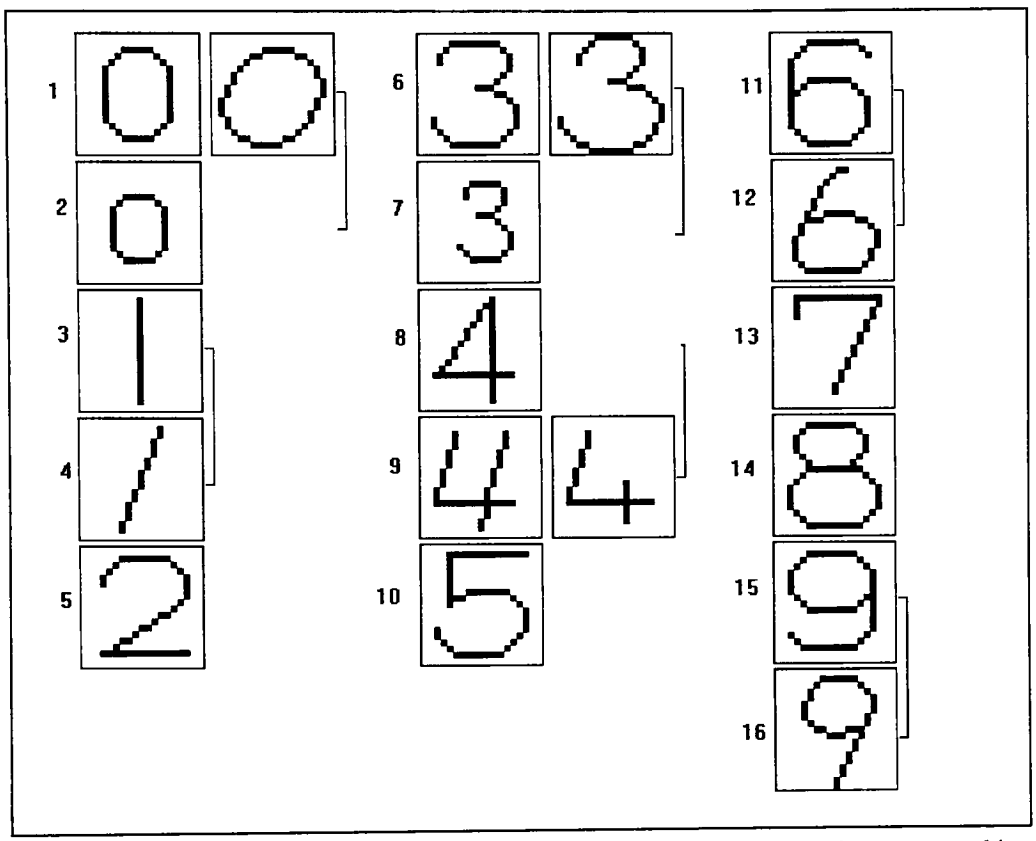

Figure 3-5 : Patterns used to train the 16 cell planes in layer u_{S4}. How these planes are combined at the input stage of layer u_{C4} is indicated on the right of the training patterns.

The patterns used for training the final S-layer, u_{S4}, are illustrated in Figure 3-5. There are 16-3x3 cell planes in this layer, each with a 5x5 receptive field. Each of these cell are trained using one or two complete patterns. The outputs of these 16 S-planes are the inputs to 10 C-planes in layer u_{C4}, each consisting of a single cell representing the 10 numerals to be recognized. How the 16 u_{S4} planes are combined for input to the 10 u_{C4} planes

is indicated at the right of the cell planes in Figure 3-5. Also, as in layer 3, no thinning of the cells in this layer is performed.

3.2.2 *Testing the Neocognitron*

In testing the pattern recognition capability of the neocognitron network, the following criteria was used. If the cell plane corresponding to the input character yields the largest output of the 10 u_{C4} planes, the response is judged to be correct. If no cell plane responds or this value is identical to that of a different plane, the character is classified as unknown. This will prevent the network from making incorrect guesses when it is uncertain. Finally, a wrong classification is used when the appropriate cell plane exhibit a weaker response than a cell in another plane.

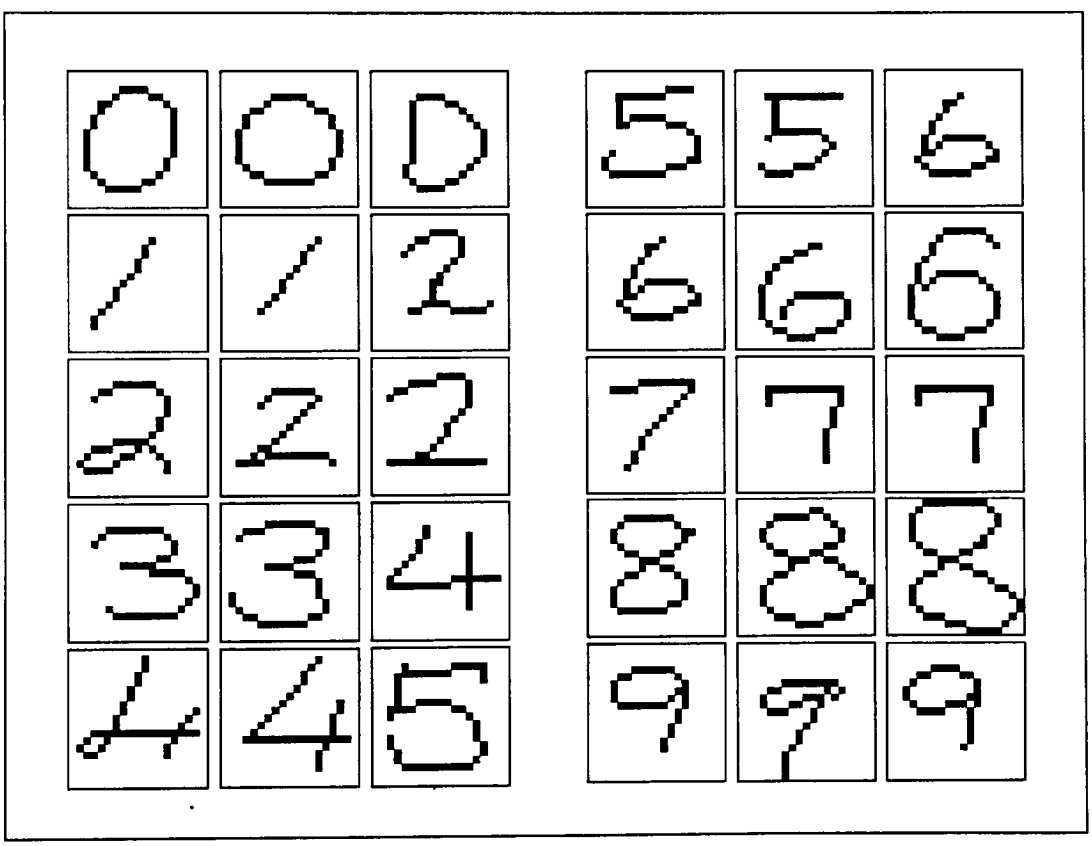

Figure 3-6 : Sample patterns that were correctly classified by the neocognitron.

Training the neocognitron as described in the previous section took approximately 4 minutes 36 seconds on a SUN SPARCStation 20. However, during the recognition phase processing a single character took approximately 4 seconds. The model was tested with numerals of different writing styles, with varying degrees of deformations from the training patterns, and with patterns placed on different positions on the input plane. In addition, testing was also conducted on the complete patterns utilized during training. These patterns are illustrated in Figures 3-4 and 3-5. The neocognitron correctly classified all the patterns in this test set. Other patterns that the neocognitron correctly classified are illustrated in Figure 3-6. Patterns that the neocognitron incorrectly classified or judged unknown are shown in Figure 3-7.

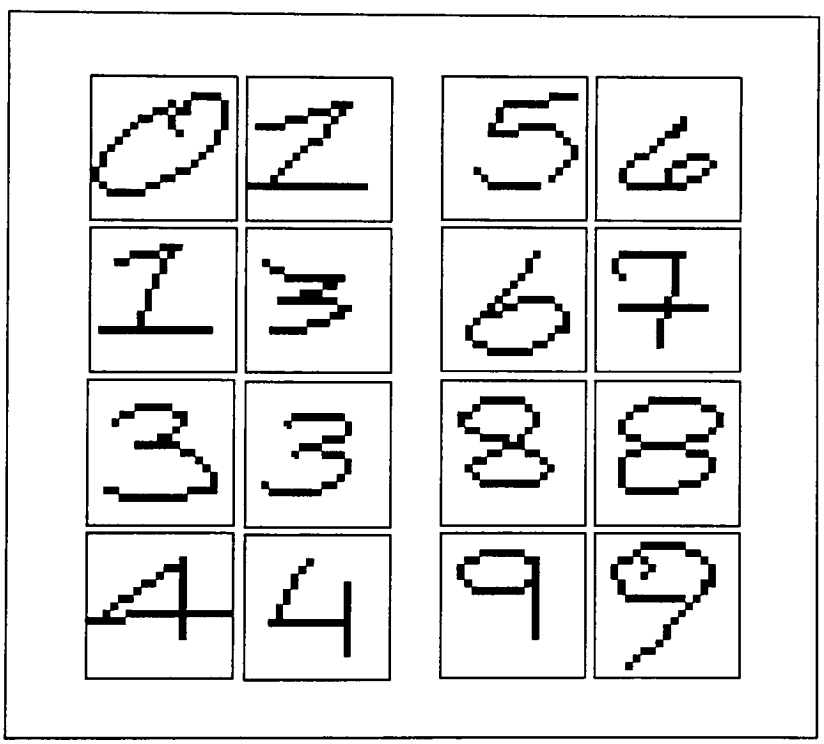

Figure 3-7 : Sample test patterns that the neocognitron failed to correctly classify.

The variety of patterns that can be recognized by the neocognitron is dictated by the numbers of layers and planes in the network, and the variations of the training patterns. Since this implementation has nine levels of C-cells and S-cells, increasing the number of layers is

not necessary. It would suffice to increase the number of planes in layers two and three, and/or the number and variations of the training patterns. These additional training patterns should include partial and complete patterns of the new styles of patterns to be recognized.

3.3 Implementation of the Digi-Neocognitron Model

Architecturally, the digi-neocognitron is identical to the neocognitron model described in Section 3.2. However, the learning rates and saturation parameters were changed in order to improve the weights developed and to satisfy the required bit restrictions. Also, to implement the DNC model functions as described in Equations (18), (26), (27) and (33), modifications to the C-routines in section 3.2 were required.

Like the neocognitron, the learning rate varied for all the layers. However, these values were developed during training to ensure that the computed weights adhered to the digi-neocognitron's bit-width restrictions. The final values for the learning rates were $q_1 = 40$, $q_2 = 135$, $q_3 = 185$ and $q_4 = 525$. The procedure utilized to obtain these results is discussed in Section 3.3.1. Also developed through training were the values for the saturation parameter. These values were $\alpha_1 = \alpha_2 = \alpha_3 = 0.125$ and $\alpha_4 = 1.0$.

The digi-neocognitron model functions required power of 2 approximations, the use of look-up tables, and shifters to implement multiplications and divisions. With the exception of the shift operations, all the required changes for converting the neocognitron to a digi-neocognitron model were made. Shifters could not be used since the calculations are performed on floating point values, and the C-compiler utilized did not allow bit-wise operations on floating point values. The C-routines for the DNC model functions are shown Appendix A-2.

3.3.1 Training the Digi-Neocognitron

With the inclusion of the procedure described in Section 3.1, training the digi-neocognitron is identical to the training described for the neocognitron. Using the procedure described in Section 3.1, the planes in the first layer of S-cell are trained. The neocognitron

model is first used and the learning rate is adjusted to ensure that the weights are within the prescribed limits. This is necessary since the approximations used by the DNC model assigns the maximum allowable values if the results are greater than or equal to these limits. In such a situation the excitatory weights will be rounded down even though the inhibitory weights are increasing. As a result the digi-neocognitron would loose its ability for pattern recognition. It is preferable to develop excitatory weights with the neocognitron that are at the minimum value required by the digi-neocognitron for rounding up. When this is achieved, two criteria are used to determine if the weights computed achieves an acceptable pattern recognition capability. First, the excitatory and inhibitory weights are compared with those for the neocognitron and the learning rate is modified; and the simulation is repeated until an acceptable correlation is obtained. Next, the pattern recognition capability of this layer is tested using known good patterns. If the results of this test are unacceptable, the saturation parameter is modified and the networks response is rechecked. If this also fails to yield an acceptable response, the inhibitory input $b_l(k)$ is adjusted. The process is repeated until the desired response is achieved or some compromise is reached. This procedure is used to train all the layers of the network. Modification of the inhibitory inputs were not required for the first layer in this model. However, it was required for the remaining layers in the network.

3.4 Performance Comparison of the Neocognitron and Digi-neocognitron Models

Both the neocognitron and the digi-neocognitron neural network models were tested using patterns from the training set, and patterns of different individual hand-writing styles. These patterns varied in size, shape, the presence of deformations, and their position on the input plane. Patterns that the neocognitron failed to recognized or classified as unknown contained significant deformations from those in the training set, or were extremely similar one of the other ten numerals. For the test set utilized, the neocognitron correctly classified 93% of the patterns, and classified 7% as unknown. In addition, for patterns that were included in the training set the neocognitron correctly classified 100% of these patterns.

As expected, the performance of the digi-neocognitron was lower than that of the neocognitron. For patterns that were included in the training set, the digi-neocognitron correctly classified 90% of the patterns, and classified 10% as unknown. For the complete set of test patterns the digi-neocognitron's classifications were 80% correct and 20% unknown. Some of the patterns that were correctly classified by the digi-neocognitron model are illustrated in Figure 3-8.

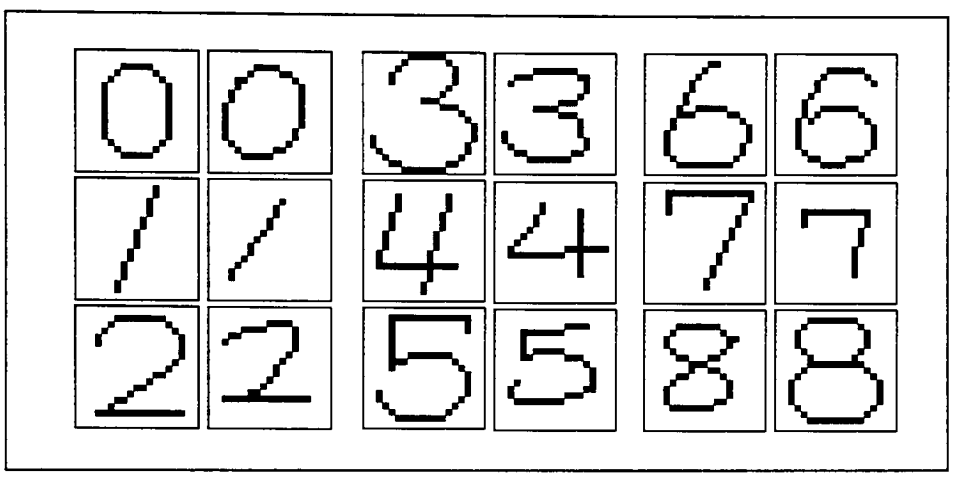

Figure 3-8 : Sample test patterns that the digi-neocognitron correctly classified.

The digi-neocognitron's performance shortfall is due to the type of training method used, the set of training patterns used, and the need for further refinement of the weight values developed.

Using the supervised training methodology, each plane is trained to recognize a specific feature from each pattern. If the power of 2 approximations are too gross, some of the cell planes will not learn to recognize its assigned feature. These cell planes will either learn to recognize a different feature or will fail to extract any features from the patterns on its input planes. As this failure to recognize an assigned feature propagates to higher stages in the network, the problem worsens and the pattern will be incorrectly classified.

The other problem with this DNC model was the set of patterns utilized during training. The training patterns used to train the last two layers of the digi-neocognitron made

it difficult to train these layers. Because the number of patterns used to train the cell planes in these layers varied, it was difficult to control the inhibitory inputs to the S-cells. For the few planes with more training patterns than the remaining planes in the layer, the inhibit output would reach the maximum upper limit before adequate excitatory weights could be developed for the remaining cell planes.

The original implementation of the digi-neocognitron model [8] utilized the inherent fault tolerant characteristics of the neocognitron, and unsupervised training to compensate for inaccuracies that are introduced as a result of the preprocessing approximations. In the unsupervised training methodology, the digi-neocognitron has the freedom to determine what features each cell plane will learn to recognize. This allows the digi-neocognitron to better compensate for the inaccuracies resulting from the power of 2 approximation. Also, the DNC model described in [8] was a much smaller model with a smaller character set.

The purpose of developing a digi-neocognitron model was to facilitate a digital VLSI implementation of the neocognitron. From the DNC performance achieved in the model constructed, this may appear to be a bad implementation choice. However, this is not necessarily the case. In the VLSI model of digi-neocognitron, the training is performed off-line. Also, the storage of the connection weights is implemented using either SRAMs, EPROMs or EEPROMs. Therefore, once the desired architecture is established, the weight values can be changed at a later date to utilize weights with a better character recognition performance. Thus, the poor performance of this DNC model does not negate the benefits that a digital VLSI implementation has to offer.

CHAPTER 4

4.0 VHDL Implementation of the Digi-neocognitron

4.1 VHDL Overview

This section provides a brief overview of VHDL (VHSIC Hardware Description Language) and it capabilities as found in [16]. It is not intended to be a comprehensive overview of the language. However, it provides some of the basic terminology required to understand the concepts that will be discussed in this chapter. For a complete discussion of VHDL please refer to the text referenced.

Two concepts of importance are an entity and a component. An entity, as used in this chapter, is a hardware abstraction of a digital system, and a component is an entity that is used within another entity. VHDL provides several constructs to describe an entity, but the two of significance are the entity declaration and the architecture body. An entity is modeled using an entity declaration and an architecture body. The entity declaration is used to described the external view of the entity, and the architecture contains the internal description of that entity. The entity declaration contains the input and output signal names. These signals through which an entity communicates with other models in its external environment are referred to as ports. The architecture body that accompanies an entity can consist of a set of interconnected components representing the structure of the entity, or a set of concurrent or sequential statements that represent the behavior of that entity.

VHDL supports several styles of modeling: structural, behavioral, dataflow and any combination of these three. In structural modeling, an entity is described as a set of interconnected components. It specifies the interface ports for the accompanying architecture body which consist of component declarations in the declarative section, and component instantiations in the statement section. This style of modeling maps directly to the hardware (gate level) implementation, but can prove extremely difficult when attempting to determine

the behavior of a very large model. An entity declaration and architecture body for a half adder which demonstrates this modeling style is shown in Figure 4-1.

```
ENTITY half_adder IS
  PORT(a, b : IN; sum, carry : OUT bit);
END half_adder;

ARCHITECTURE structural OF half_adder IS
  -- component declarations
  COMPONENT xor2
    PORT(x, y : IN bit; z : OUT bit);
  END COMPONENT;
  COMPONENT and2
    PORT(l, m : IN bit; n : OUT bit);
  END COMPONENT;
BEGIN          -- component instantiations
  x1 : xor2 PORT MAP(a, b, sum);
  a1 : and2 PORT MAP(a, b, carry);
END structural;
```

Figure 4-1 : Structural VHDL model of a half-adder circuit.

The next modeling style, behavioral modeling, specifies the behavior of an entity as a set of sequential statements. These sequential statements are specified within a process statement and specifies the functionality of the entity not its structure. This process statement which appears within the architecture body is a concurrent statement even though its contents are executed sequentially. Figure 4-2 illustrates a behavioral implementation of a half-adder circuit.

```
ENTITY half_adder IS
  PORT(a, b : IN; sum, carry : OUT bit);
END half_adder;

ARCHITECTURE behavioral OF half_adder IS
BEGIN
  adder : PROCESS(a, b)
  BEGIN
    sum <= a XOR b;
    carry <= a AND b;
  END PROCESS adder;
END behavioral;
```

Figure 4-2 : Behavioral VHDL model of a half-adder circuit.

In Figure 4-2, the list of signals in parentheses after the keyword process is termed the sensitivity list. This process is invoked whenever there is an event on any of the signals in this list. Also common to this style of modeling in the signal assignment operator, <=. Unlike the variable assignment operator, :=, which assigns a value to a variable instantaneously, the signal assignment operator assigns a value to a signal after a user-specified or the default delta delay.

```
ENTITY half_adder IS
  PORT(a, b : IN; sum, carry : OUT bit);
END half_adder;

ARCHITECTURE dataflow OF half_adder IS
BEGIN
  sum <= a XOR b AFTER 8 ns;
  carry <= a AND b AFTER 4 ns;
END dataflow;
```

Figure 4-3 : Dataflow VHDL model of a half-adder circuit.

In the dataflow modeling style, the flow of data through an entity is expressed using concurrent signal assignment statements. The structure of an entity which uses this modeling style is not specified explicitly; however, it can be implicitly deduced from the set of statements. Figure 4-3 demonstrates the half-adder circuit using the dataflow modeling style.

VHDL also allows combining these three styles of modeling in a single architecture to describe an entity. This is referred to as mixed-style modeling. In an architecture body this can be achieved through the use of component instantiations that represent structural modeling, process statements which represent behavioral modeling, and concurrent signal assignments that represent dataflow modeling. An example of mixed-style modeling is demonstrated for a full-adder in Figure 4-4. Note the inclusion of signal and variable declarations, and variable assignments. Signal declarations can only occur in an architecture body, and variable declarations and assignments can occur only within a process statement.

```
ENTITY full_adder IS
  PORT(a, b, cin : IN bit; sum, cout : OUT bit);
END full_adder;

ARCHITECTURE mixed_style OF full_adder IS

  COMPONENT xor2
    PORT(a, b : IN bit; z : OUT bit);
  END COMPONENT;

  SIGNAL s1 : bit;

BEGIN
  x1 : xor2 PORT MAP(a, b, s1);          -- structural modeling

  PROCESS(a, b, cin)                     -- behavioral modeling
    VARIABLE t1, t2, t3 : bit;
  BEGIN
    t1 := a AND b;
    t2 := b AND cin;
    t3 := a AND cin;
    cout <= t1 OR t2 OR t3;
  END PROCESS;

  SUM <= s1 XOR cin;                     -- dataflow modeling
END mixed_style;
```

Figure 4-4 : Mixed-style VHDL model of a full-adder circuit.

Also, with VHDL it is possible to exercise and verify the correctness of the hardware model implemented using VHDL. This is performed by a test bench. The purpose of this test bench is to generate stimulus (waveforms) for the simulation, to apply these stimulus to the entity that is being tested, monitor the output responses, and to compare the output responses with expected known values. The application of the stimulus to the entity-under-test is done automatically by instantiating the entity in the test bench model and then specifying the appropriate interface signals. A typical format of a test bench model is shown in Figure 4-5. The test bench models used to verify the functionality of the digi-neocognitron model functions are listed in Appendix A-4. .

```
ENTITY test_bench IS
END test_bench;

ARCHITECTURE tb_behavior OF test_bench IS

  COMPONENT entity_under_test
    PORT(list-of-ports-their-types-and-modes);
  END COMPONENT;

local signal declarations;

BEGIN
  Generate-waveforms-using-behavioral-constructs;

Apply-to-entity-under_test;
  EUT : Entity_Under_Test PORT MAP(port-associations);

BEGIN
    Monitor-values-and-compare-with-expected-values;
END tb_behavior;
```

Figure 4-5 : Typical format of a VHDL test bench model.

4.2 Hardware Description of the DNC Model Functions

The cells of the digi-neocognitron were modeled using VHDL prior to implementing these four processing elements in VLSI. These models were used to develop the required timing sequences and verify the functionality of the processing elements, to validate the functionality of the complete design, and subsequently to synthesize the gate and transistor level circuits of the DNC model functions. VHDL is hardware description language that can be used to model digital systems at various levels of abstraction. A design can be modeled algorithmically, as with other high-level programming languages, or it can be modeled at the gate level similar to what would be done using a circuit design tool. Like other high-level programming languages, VHDL supports two design methodologies, top-down and bottom-up, and it provides support for modeling a system hierarchically. It is possible to model a system and its subsystems from the architectural level to the gate level. The model developed for the digi-neocognitron used the behavioral style of modeling and a bottom-up design methodology. A bottom-up approach was required in order to develop the timing sequences

necessary for the model functions. These timing sequences were then utilized by the controllers of the computation processes for the four cell types.

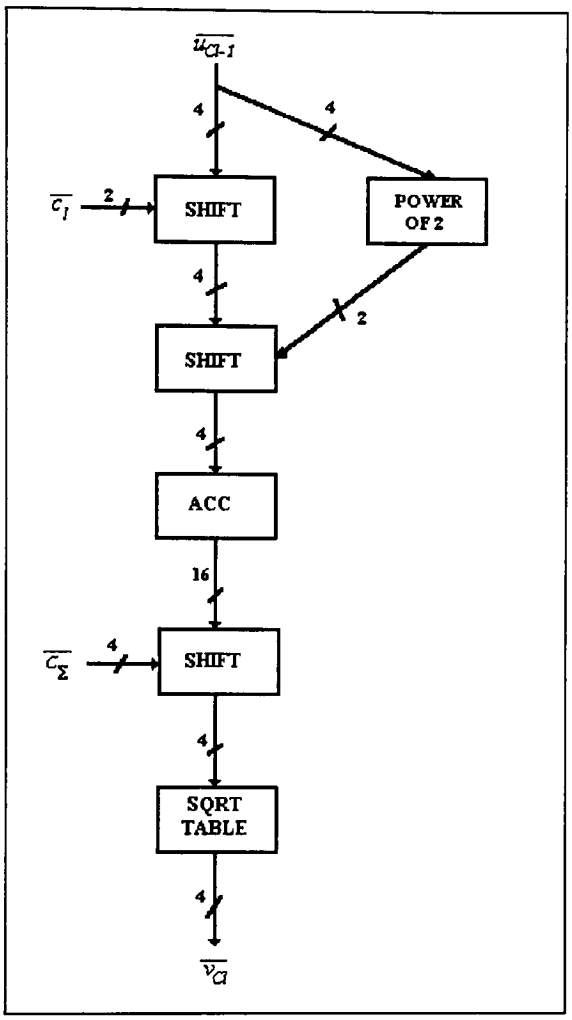

Figure 4-6 : Block diagram for the Vc-cell output calculation of the digi-neocognitron.

Equations (16) - (34) describes the digi-neocognitron model functions. It was not possible to implement these functions as would be done in hardware using the C programming language; however, VHDL provides the facilities to model the hardware directly. The VHDL models of the four output functions of the dig-neocognitron were carefully designed to minimize the logic circuits that would be constructed in hardware. This is discussed in more

detail in chapter 5. The VHDL models of the four functions described in this section are listed in Appendix A-3.

The inhibitory Vc-cell output given by Equation (18) is implemented as shown in Figure 4-6. At level 1, $u_{Cl\text{-}1}$ represents the pixels in the input pattern. As indicated previously, a 0 pixel is input as a 0.0000 binary and a 1 pixel is input as a 0.1111. The squaring of u_{Cl} is implemented by the POWER OF 2 block as defined in Table 2-2, and can be implemented using simple combinatorial logic. Multiplication by c_l and division by C_Σ are implemented using shifters, the summation over the connection areas is performed by an accumulator (adder plus register), and the square root is approximated by the look-up table shown in Table 2-3. Similar to the square function, this look-up table can be implemented using combinatorial logic.

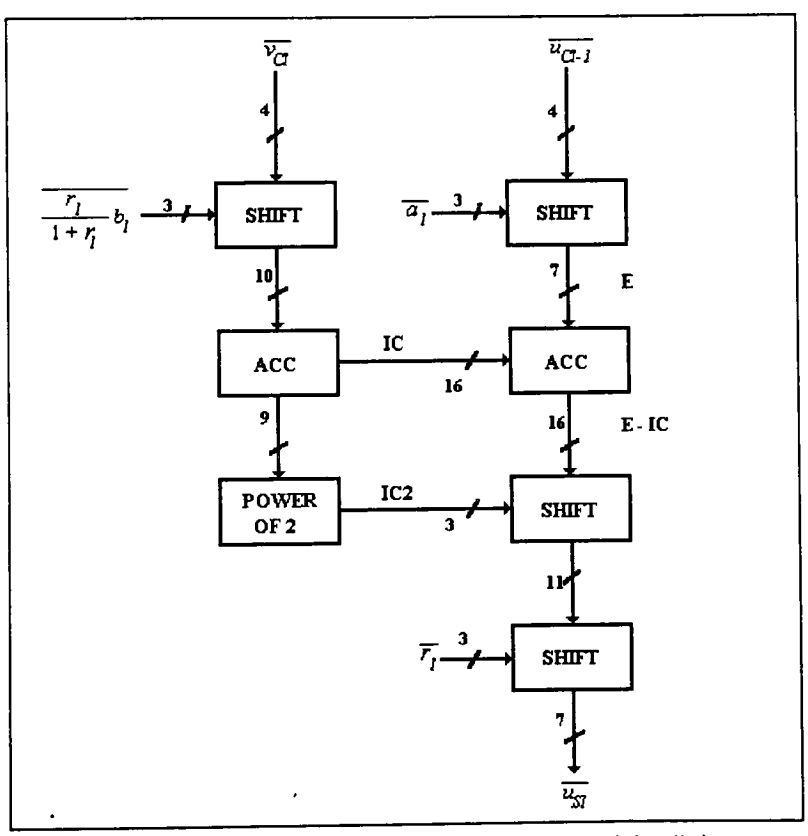

Figure 4-7 : Block diagram for the S-cell output calculation of the digi-neocognitron.

The SHIFT blocks shown in Figure 4-6 were modeled as multiplexer-based right shifters. These shifters are not clocked but they do require an enable signal to latch the outputs. The ACC block represents a synchronous accumulator with an asynchronous reset.

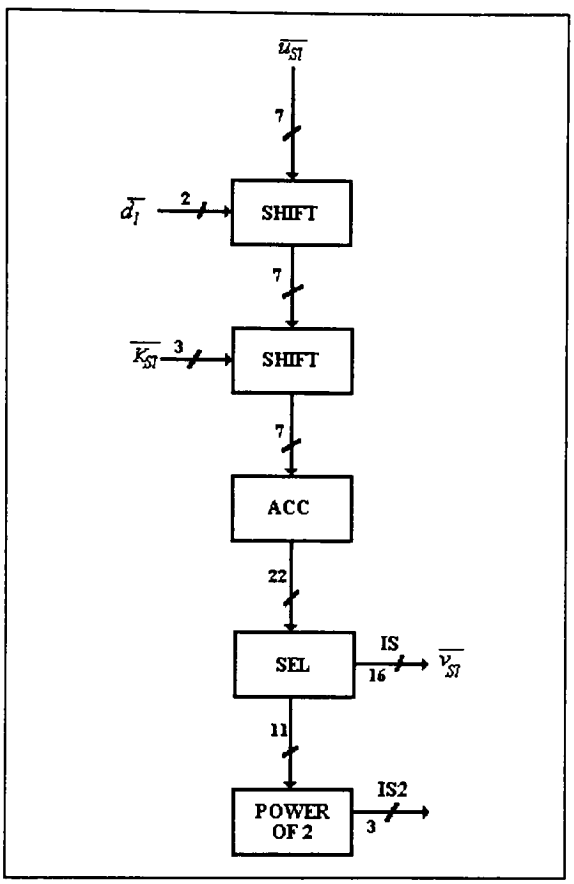

Figure 4-8 : Block diagram for the Vs-cell output calculation of the digi-neocognitron.

Figure 4-7 shows the implementation for the S-cell output function. The left hand side of Figure 4-7 implements the inhibitory portion (*IC*) of Equation (25), and the excitatory portion (*E*), given by Equation (24), is implemented by the right hand side. The accumulator used to compute the inhibitory portion of the S-cell output is similar to that used in Vc-cell calculation. However, an enable signal is required to limit the summation to the two terms used for the approximation. In this implementation, the two terms used to approximate *IC* are always positive, requiring only an adder. To compute the excitatory portion *E* requires both an adder and subtractor, and a comparator. This accumulator calculates *E*, but performs the

subtraction E - IC only if $E > IC$. The POWER OF 2 block can be implemented using combinatorial logic. This block performs the approximation for the $^1/_{(1+I)}$ inhibit function shown in Table 2-4. The two remaining SHIFT blocks are right and left shifters, respectively.

With the preprocessing approximations, the Vs-cell output of the digi-neocognitron, given by Equation (27), is implemented as shown in Figure 4-8. Both SHIFT blocks represent right shifters and the accumulator (ACC) is similar to that used in the Vc-cell calculation. The SEL block is used as a delay circuit for the inhibit output (IS) and the POWER OF 2 block approximates $^1/_{(1+I)}$. The final block which approximates $IS2$ occurs in the C-cell output calculation. However, it is implemented here since this value contributes to the C-cell output in the final layer only.

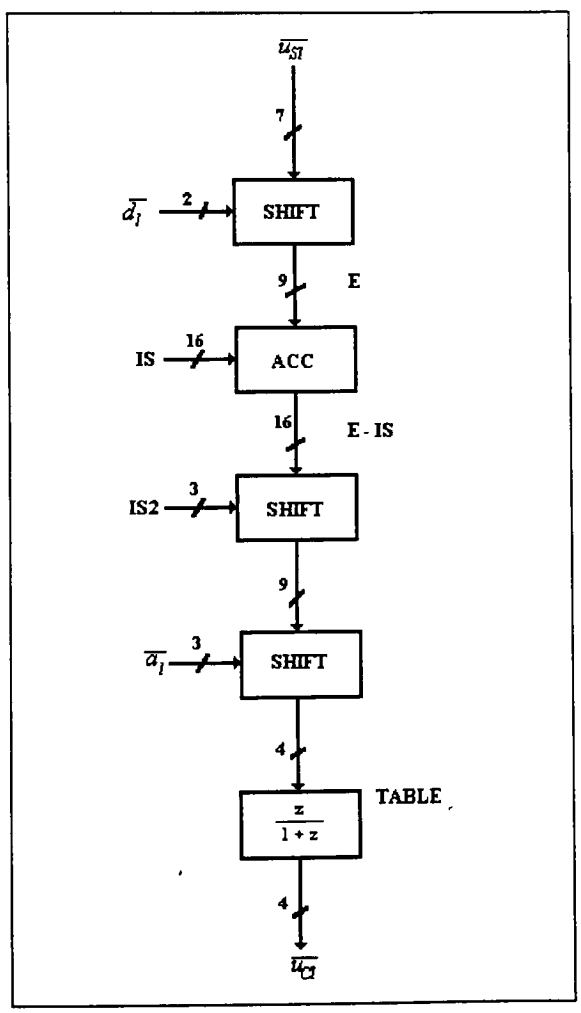

Figure 4-9 : Block diagram for the C-cell output calculation of the digi-neocognitron.

The final processing element of the digi-neocognitron, the C-cell, is implemented as shown in Figure 4-9. The first two SHIFT blocks are right shifters and the third is a left shifter. The ACC block is an accumulator similar to that used in the S-cell calculation; it computes E - IC from Equation (33) if $E > IC$. The final block of Figure 4-9 implements $^{Z}/_{(1+Z)}$ using the look-up table given by Table 2-5.

4.3 VHDL Simulation of the DNC Model Functions

The digi-neocognitron model functions were simulated using the Mentor Graphics QuickVHDL simulation environment. Simulation at this stage was required for several reasons: (1) to verify the functionality of the VHDL models implemented; (2) to develop the timing information required by the components that controlled the computations, and (3) to provide a comparative tool for the logic simulation of the circuit design of these model functions.

As discussed earlier in this chapter, it is possible to used VHDL to apply stimulus to the component that is being tested. Consequently, test bench models were constructed for the four output functions of the digi-neocognitron. These test bench models were used only to supply stimulus to the components under test, and not to monitor or compare the results with expected results. Such detailed test bench models were not required since the models being tested were relatively small. Also, monitoring of the intermediate results were required which would have made the test bench models excessively complicated.

The four models were first tested using arbitrary inputs to determine the correct transitions points of the inputs and the results at each stage of the computations. The test bench models were subsequently modified to utilize values that were calculated from the previous stages for a given pattern. This aided in the verification of the computations performed in the digi-neocognitron model. These models utilize a clock with a 10 ns period and a 50% duty cycle. All signal changes occur at clock boundaries and the synchronous accumulators operate on the rising edge of the clock. The clock period utilized during

simulations was extremely optimistic. However, due to the length of time required to perform the simulations, a 10 ns clock period was beneficial during the debug stage.

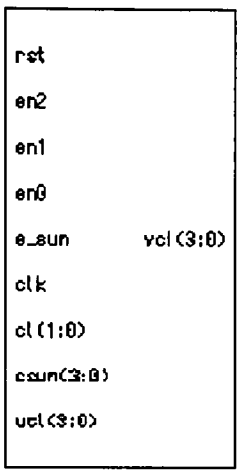

Figure 4-10 : Block diagram of the Vc-cell VHDL component.

A block diagram illustrating the component interface of the Vc-cell output calculations is shown in Figure 4-10, and the simulation results for the Vc-cell computation of a single 3x3 receptive field is shown in Figure 4-14. To complete the computation for a single 3x3 connection area required 140 ns (14 clock cycles). With the exception of the first layer and some of the C-planes, computations are performed over several connection areas of size 3x3 or 5x5 in the DNC model. However, summation over a single plane was sufficient to verify the correctness of the model functions.

The test bench model listed in Appendix A-4 was used to provide the stimulus to the Vc-cell component. Appendix A-4 also contains the listings of the test bench models for the three remaining output functions of the digi-neocognitron.

The S-cell component was the second model to be simulated. The order of simulation was important due to the desire to use the computed outputs in subsequent layers. This helped significantly during the functional verification of the complete digi-neocognitron model in which the order of the calculations is fixed.

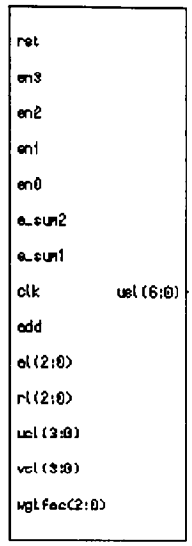

Figure 4-11 : Block diagram for the S-cell VHDL component.

Shown in Figures 4-11 and 4-15 is the block diagram for the S-cell component and the simulated results, respectively. The S-cell calculations were also performed using a 3x3 receptive field. Note that in the test bench models, the inputs from the preceding C-layer are identical to those used in the Vc-cell calculation. This reflects what actually occurs in the digi-neocognitron model. In addition, the inhibitory input from Figure 4-15 is utilized in the S-cell output calculation. Similar to the Vc-cell model, the S-cell computation for a 3x3 connection required 14 clock cycles.

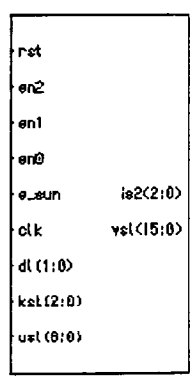

Figure 4-12 : Block diagram for the Vs-cell VHDL component.

The Vs-cell and C-cell functions were the last models simulated. The inputs that were used to simulate these models were generated by the digi-neocognitron VHDL model. As stated earlier, the objective during the simulation of the four models was to utilize realistic input values. Therefore, it was necessary to compute the results for an entire Vc-plane and a layer of S-planes. This process provided the desired inputs for testing the Vs-cell and C-cell models, and simultaneously verified the computation models for the Vc-layer and the S-layer.

The Vs-cell computation was performed on a 3x3 receptive field. The component block diagram containing the I/O signals is illustrated in Figure 4-12, and the resulting waveforms in Figure 4-16. However, for the C-cell calculation a 5x5 connection area was used. This is the receptive field size of the first two layers of the network, and this was an opportunity to assess the processing time required for a larger field size. The component block diagram and simulation results of the C-cell components are shown in Figures 4.13 and 4.17, respectively. For the 5x5 connection area utilized in the C-cell test bench, the calculation required 30 clock cycles.

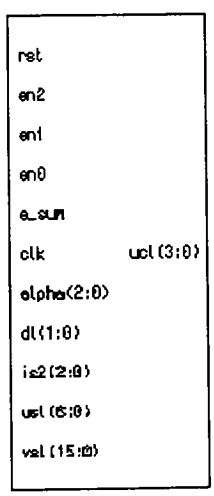

Figure 4-13 : Block diagram for C-cell VHDL component.

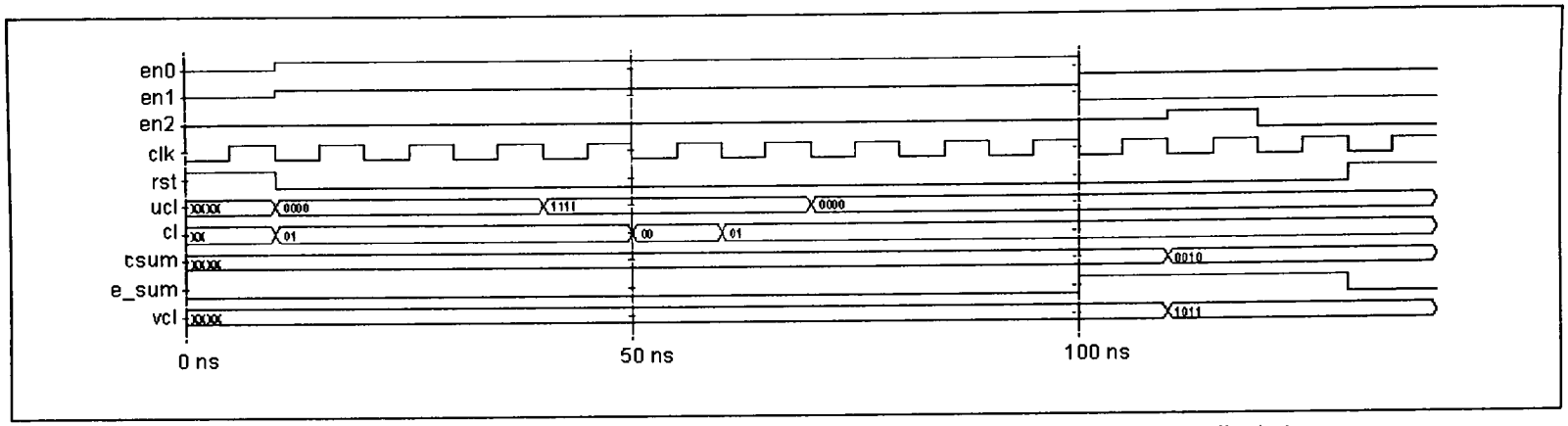

Figure 4-14 : Simulation results for the Vc-cell component using the test bench in Appendix A-4.

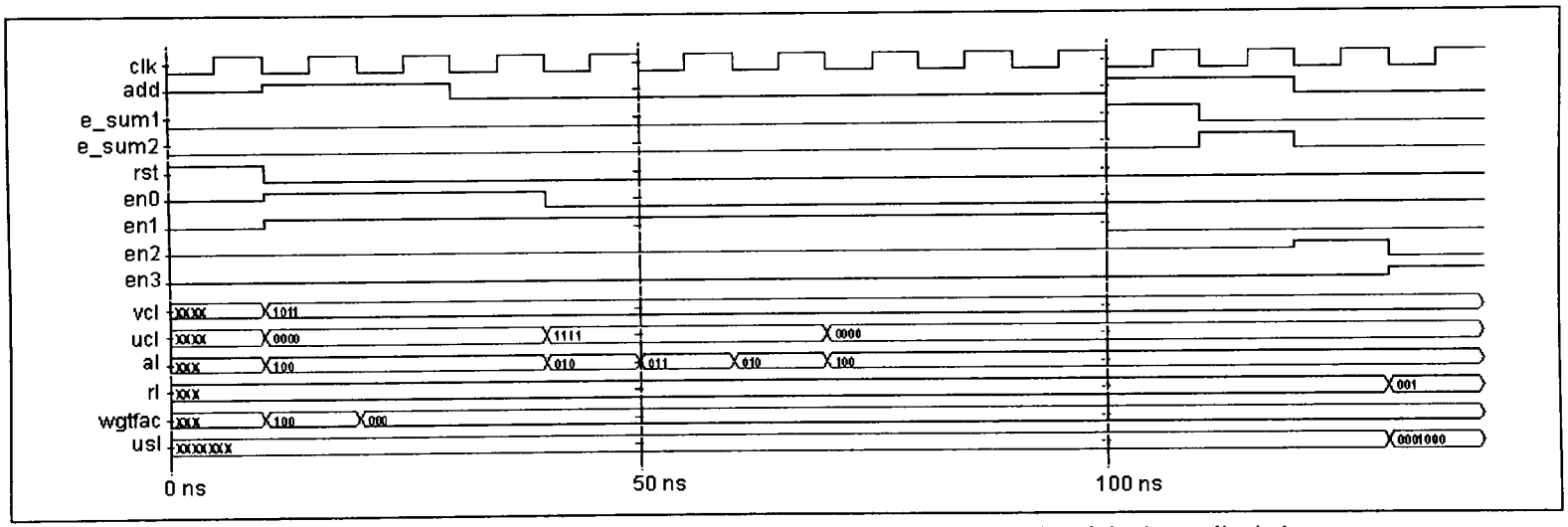

Figure 4-15 : Simulation results for the S-cell component using the test bench in Appendix A-4.

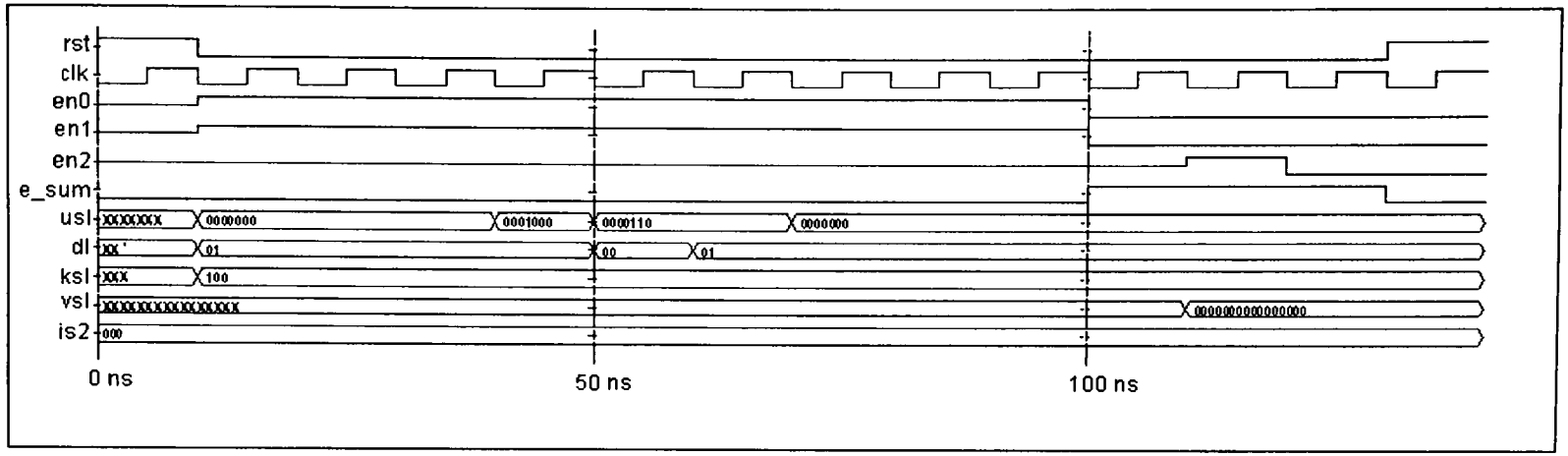

Figure 4-16 : Simulation results for the Vs-cell component using the test bench in Appendix A-4.

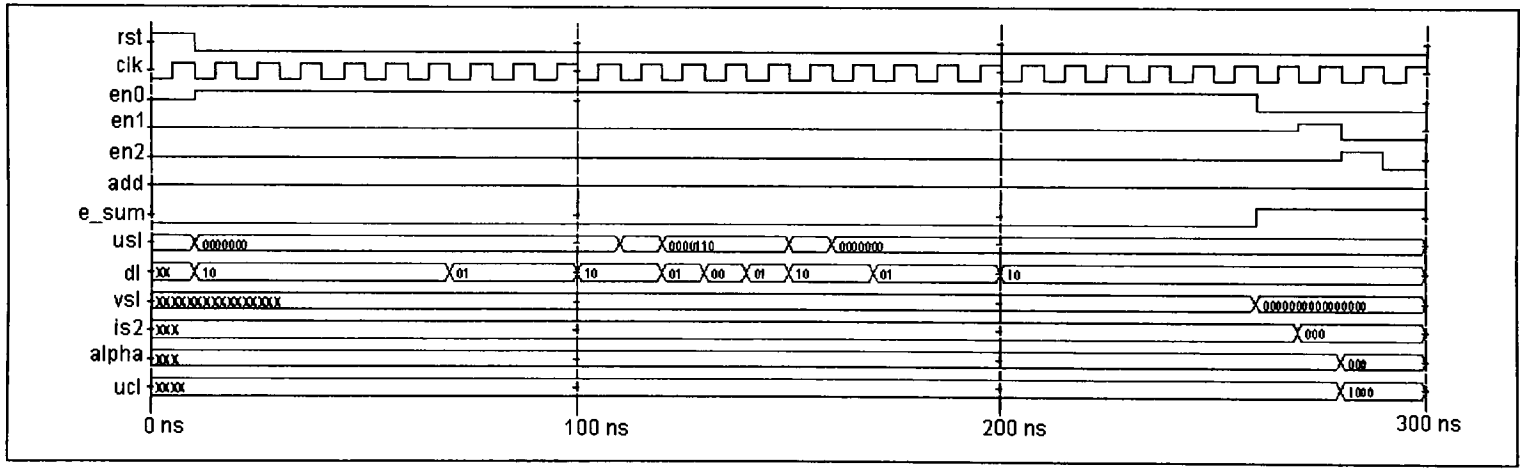

Figure 4-17 : Simulation results for the C-cell component using the test bench in Appendix A-4.

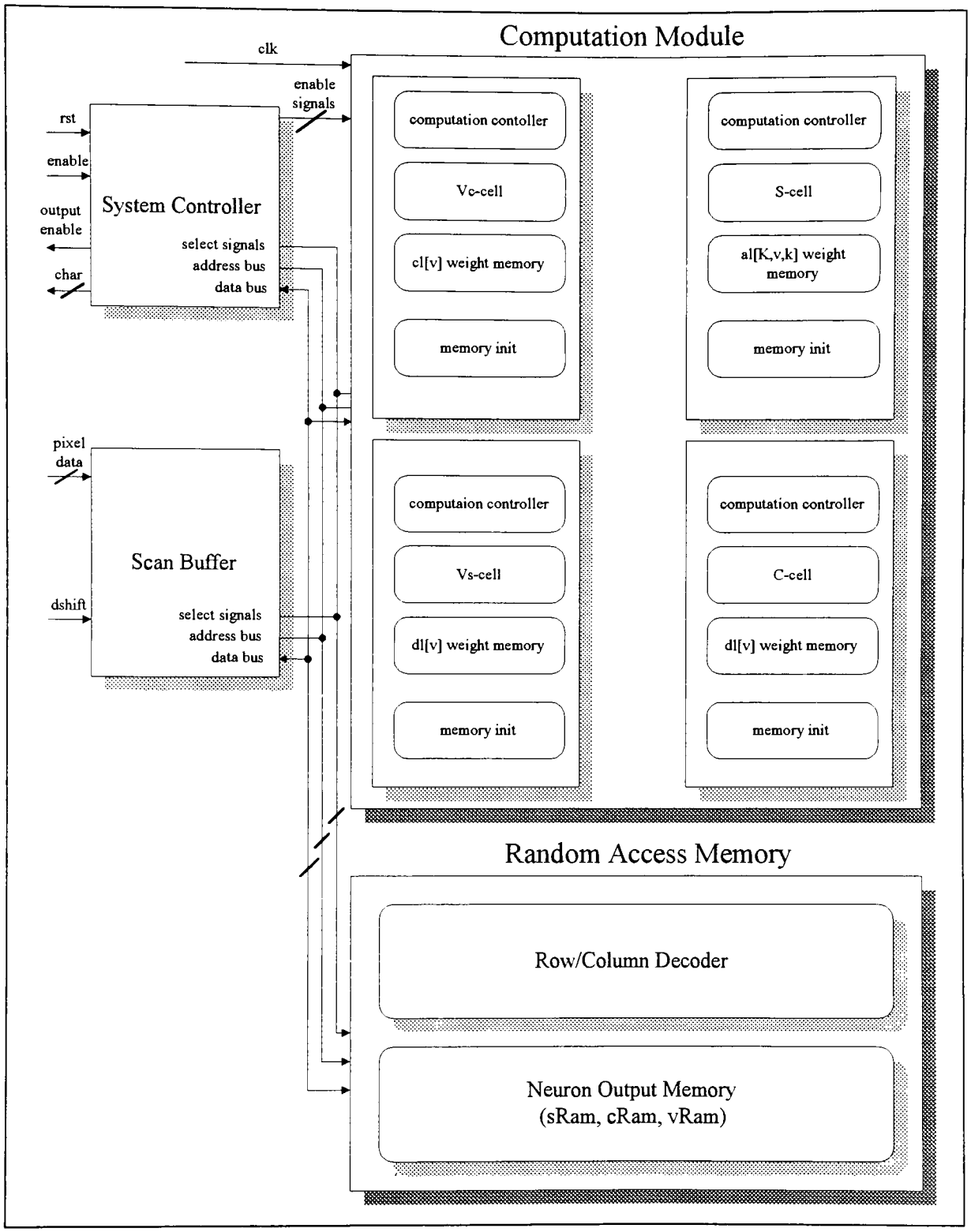

Figure 4-18 : Architecture of the digi-neocognitron VHDL model.

4.4 Architecture of the Digi-Neocognitron Model

The architecture of the digi-neocognitron VHDL model is illustrated in Figure 4-18. This model consisted of four primary functional blocks: a scan buffer, a random access memory , a system controller and a computation module.

SCAN BUFFER

This block reads the pixel data of the pattern to be recognized into the model. It requires 19 clock cycles to read a 19x19 character. This is accomplished by using a single bit (0 or 1) to represent each pixel, since at the input layer a white (off) pixel is represented as 0000 and a black pixel (on) is represented as 1111. Even though the input character is read in 19 clock cycles, this data is not immediately available for processing. The scanner module buffers the 19 scan lines then sequentially writes the data into the RANDOM ACCESS MEMORY block. Consequently, after initialization, input data is not available for processing until (19 x 19) + 19 clock cycles. However, this large initial delay is dependent upon the method used to implement this model. In a hardware implementation, a high speed memory block with its own internal clock can be used to store the input data. Also, it is not necessary to wait until all the input data is read prior to initiating the date store operation. The scan process was implemented in this manner so that the u_0 input plane could be treated as just another C-layer.

RANDOM ACCESS MEMORY

This block contains three RAM blocks that are referred to as vRAM, cRAM and sRAM. The first character in the names of the memory module indicates the cell layers for which they are used. The vRAM is used to store the output of the Vc-cell computations, the cRAM stores the outputs of the C-cells and the input pattern, and the sRAM stores the outputs of the S-cells. These RAM modules are modeled with an access time of 5 ns. A 5 ns access time was used because the input data to the four model functions must be available every 10 ns. This was due to the clock frequency chosen for the simulation.

SYSTEM CONTROLLER

The function of the controller block is to provide the handshake signals that monitor the calculations performed by the compute blocks of the four model functions. It enables the computation for each layer in the network, and performs some post processing on the results from the last layer of C-cells to provide the final result.

COMPUTATION MODULE

The computation block is where all the calculations and results storage are performed. This is not a physical module, but a virtual module for grouping the four computation modules of the digi-neocognitron: vcOutput, usOutput, vsOutput and ucOutput. Each of these modules are organized as parallel processing elements with its own local weight memory. These modules also have access to the shared RANDOM ACCESS MEMORY block described above.

vcOutput

This module consisted of the $c_l(v)$ weight memory, a block which initializes or modifies this memory, a single Vc-cell output module, and a controller for computing the output over a Vc-cell's connection area in any layer of the network.

usOutput

The usOutput module consisted of the $a_l(K,v,k)$ weight memory, a block which initializes or modifies this memory, a single S-cell output module, and a controller for computing the output over an S-cell's connection area in any layer of the network.

vsOutput

The vsOutput module consisted of the $d_l(v)$ weight memory, a block which initializes or modifies this memory, a single Vs-cell output module, and a controller for computing the output over a Vs-cell's connection area in any layer of the network.

ucOutput

The ucOutput module consisted of the $d_l(v)$ weight memory, a block which initializes or modifies this memory, a single C-cell output module, and a controller for computing the output over a C-cell's connection area in any layer of the network.

The organization described above was chosen for the computation modules to increase the use of parallelism. Even though the calculations in the modules that comprise the model functions were pipelined, no parallelism was utilized during the calculations for the cell planes of each layer. Processing the cell planes in parallel adds to the complexity of the VHDL model, but it would significantly reduce the simulation time and the time required to process each pattern.

4.5 VHDL Simulation of the Digi-Neocognitron Model

The VHDL model of the digi-neocognitron was simulated in stages. It was necessary to verify that the calculations performed in the lower stages were correct before proceeding. Also, as noted previously, the inputs to each stage are the results computed in the previous stage. The exception to this is first layer of S-cells in which the inputs to this stage is the pattern being processed. As with the model functions, the clock used to simulate the digi-neocognitron had a period of 10 ns and a 50% duty cycle.

The first step of the simulation was to verify the initialization of the weights developed through off-line training. The initialization of the weight memories of all four layers took 21,181 clock cycles. Since all cell types have their own local memory, it was possible to initialize all of the weight memories concurrently. Also performed concurrently with the weight initialization was the storing of the pattern to be recognized to the input plane. Since the time required to store all the weights was longer that the time required to store the input pattern, this process was transparent to the system. However, with the architecture of this model, it was not possible to input the next character until all processing required for the current pattern was completed. This occurs because the input plane was treated as just another C-plane. In a hardware implementation, the input plane should be separate from the C-planes. Then, the pattern to be processed could be stored immediately after the calculations for the first S-layer is completed. Using this approach the overhead required for storing a pattern would be incurred for the first pattern only.

In subsequent stages of the simulation process, each of the computation modules shown in Figure 4-18 were simulated in the same sequence described for the software models.

At the end of the simulation run for each of these blocks, a 3x3 or 5x5 area from one of the computed planes was processed using one of the models for the four output functions. This was used to verify that the functionality of the computation blocks.

Table 4-1 : Performance comparison of the digi-neocognitron for sequential vs. concurrent calculation of the cell planes in each layer

Cell Layers	Number of Cell Planes	Calculating Cell Planes Sequentially (clock cycles)	Calculating Cell Plane Concurrently (ns)
u_{S1}	12	62,292	5,191
u_{C1}	8	31,349	3,919
u_{S2}	38	319,732	8,414
u_{C2}	22	36,008	1,637
u_{S3}	32	263,552	8,236
u_{C3}	30	18,902	630
u_{S4}	16	108,864	6,804
u_{C4}	10	150	15

As indicated previously, the calculations for the cell planes were performed sequentially. This was done to simplify an already complicated model. However, with the scaling of CMOS VLSI technologies and the possibility for a significant gain in performance from concurrent calculation of the cell planes, this would not be the desired approach for a VLSI implementation. The computation blocks were designed to facilitate parallel processing of all the cell planes in a single layer. All the cell planes within a layer have an identical number of connections from the cells in the previous stage. Therefore, rather than reading the input data 12 times for 12 cell planes, it could be read once and all the cell planes could simultaneously process the data. To enable concurrent calculation of the cell planes in each layer would require a single computation block (vcOutput, usOutput, vsOutput or ucOutput) for each plane in the layer and additional memory I/O controller for reading the required inputs from memory and storing the results to memory. Table 4-1 shows an estimate of the performance that would be achieved by computing the outputs of the cell planes in parallel. This data is based on the VHDL simulation results using a 10 ns clock. To sequentially

calculate the outputs of the cell planes of the S and C-layers of the digi-neocognitron took 340,669 clock cycles. This is in comparison to 34,846 clock cycles that could be achieved by performing the calculations in parallel. To sequentially process a single pattern, the digi-neocognitron took 889,786 clock cycles. If the calculations for the cell planes in the S and C-layers are performed concurrently, this time drops to approximately 83,829 clock cycles. It is clear from these processing times that despite the increase in silicon area, concurrently calculating the cell planes of each layer if the preferred implementation choice.

CHAPTER 5

5.0 Circuit Implementation of the Digi-Neocognitron Model Functions

5.1 Synthesis and Simulation of the DNC Model Functions

The digi-neocognitron model functions are comprised primarily of datapath operators (shifters, adders, subtractors and comparators) and combinatorial logic. This allows the digi-neocognitron to benefit from many of the principles employed in a structured design methodology. These include hierarchy, regularity, modularity and locality.

Generally, n-bits of data are processed enabling the use of n-identical circuits to implement a function. Also, data operations are generally sequenced in time leading to the idea of physically placing linked data operators adjacent to each other. Data can then be arranged to flow in one direction, and control signals can flow in an orthogonal direction to the dataflow. It is these features that makes the DNC functions ideal for logic synthesis. Even though the digi-neocognitron model functions were implemented using behavioral modeling, these models were carefully designed to facilitate logic synthesis. Great effort was made to minimize the circuitry that would be synthesized for each of the logic blocks contained in the models. In addition to the design efforts for optimizing the VHDL code, Mentor Graphics' AutoLogic was used to optimized the circuits for area and in some instances speed (accumulators) prior to synthesis. AutoLogic was then used to synthesize the logic and transistor level circuits for the cells of the digi-neocognitron to a 1.2 μm CMOS process.

5.1.1 Logic Synthesis of the DNC Model Functions

Figures 5-1 through 5-9 illustrates the circuit diagrams of the four model functions generated by the AutoLogic synthesis tool, and the logic simulation results using the QuickSim simulator. The function of the logic blocks in each of the schematic diagrams shown are identical to those described in section 4.3. The external inverters shown are

utilized to provide needed drive capability and to resolve fanout violations. Similar to the VHDL models for these functions, all shifters were multiplexer-based shifters and were constructed from 2:1 and 4:1 multiplexers. These logic blocks also contained a series of D-latches to latch the shift result. Figure 5-1 shows the logic circuit for the *shift1vc* block of the Vc-cell output function illustrated in Figure 5-2. This is a 2:1 multiplexed-based right shifter.

The look-up tables (*sqrt_tbl* and *psi1uc*) and power of 2 blocks (*pow2vc* and *pow2vs*) consisted of simple combinatorial logic. The boolean equations that implemented these functions were reduced prior to implementation. However, AutoLogic also performed logic reduction on these equations during the optimization stage prior to synthesizing the circuit diagrams.

The Vc-cell output function circuit diagram shown in Figure 5-2, the *acc1vc* block is a 16-bit accumulator. This was implemented using a 16-bit full-adder with 2-bit carry-lookahead units. Also required to implement this accumulator were 32-D flip flops (16 with set and clear inputs) and 12 D-latches.

In addition to four multiplexer-based shift circuits, the S-cell output schematic shown in Figure 5-4 required two adders, a subtractor, and comparator circuits for the VLSI implementation. Implementing the accumulators that calculated the inhibitory (Vc-cell) and excitatory inputs to the S-cell, each required a 16-bit adder with 2-bit carry-lookahead units. Also needed to compute the excitatory input to the S-cell was a 16-bit unsigned comparator with 2-bit carry-lookahead units. This comparator was used to implement the function $E > IC$. Comparators were also needed to implement the power of 2 function (*pow2us*). Utilized for this function was a 3-bit and a 9-bit unsigned comparator with 2-bit carry-lookahead units. The S-cell also require 101 sequential components. These consisted of 41-D flip flops, 32-D flip flops with set and clear and 38 D-latches.

To implement the circuit for the Vs-cell output function shown in Figure 5-6, a 16-bit adder and five 11-bit unsigned comparators were synthesized for the accumulator (*acc1vs*), and the power of 2 (*pow2vs*) components, respectively. Both of these circuits were implemented with 4-bit carry-lookahead units. The select component (*sel1vs*) is simply a

buffer circuit and was constructed using D-latches. Overall, the Vs-cell circuit required 32-D flip flops (16 with set and clear inputs) and 41 D-latches.

Shown in Figure 5-8 is the circuit diagram for the C-cell processing element of the digi-neocognitron. The shift circuits implemented here are similar to those used in the Vs-cell, and the *psiofx* component is a look-up table made up of combinatorial logic. To implement the accumulator (*acc1uc*) a 16 bit full-adder, a 16-bit subtractor and a 16-bit unsigned comparator each with 4-bit carry-lookahead units were used. The sequential components that were required for the C-cell circuit were 32 D flip-flops (16 with set and clear inputs) and 21 D latches.

5.1.2 Simulation of the Synthesized Circuits

The functionality of the logic circuits synthesized by AutoLogic for the DNC model functions was verified using Mentor Graphics' QuickSim simulator. These results are shown in Figures 5-3, 5, 7 and 9. The test vectors used in the circuit simulation were identical to those used to test the VHDL models of these functions. The results of the logic simulation were consistent with those obtained for the VHDL models. This verified the functionality of the circuits synthesized by AutoLogic. However, it does not verify the timing requirements or consider the actual circuit delays. Therefore, the 10 ns clock period used in the simulation is still quite optimistic. To verify the timing requirements of the synthesized circuits, a transistor level simulation would be necessary. Also, the types of materials used for interconnect in the circuit layout, and their dimensions should be used along with the transistor level circuit models.

The circuit simulation results for these model functions were performed using the following parameters:

temperature : 25^0C
volts : 5 v
estimated net capacitance : 0.01 pF
unit load capacitance : 0.2 pF
default rise/fall time : 2.0 ns.

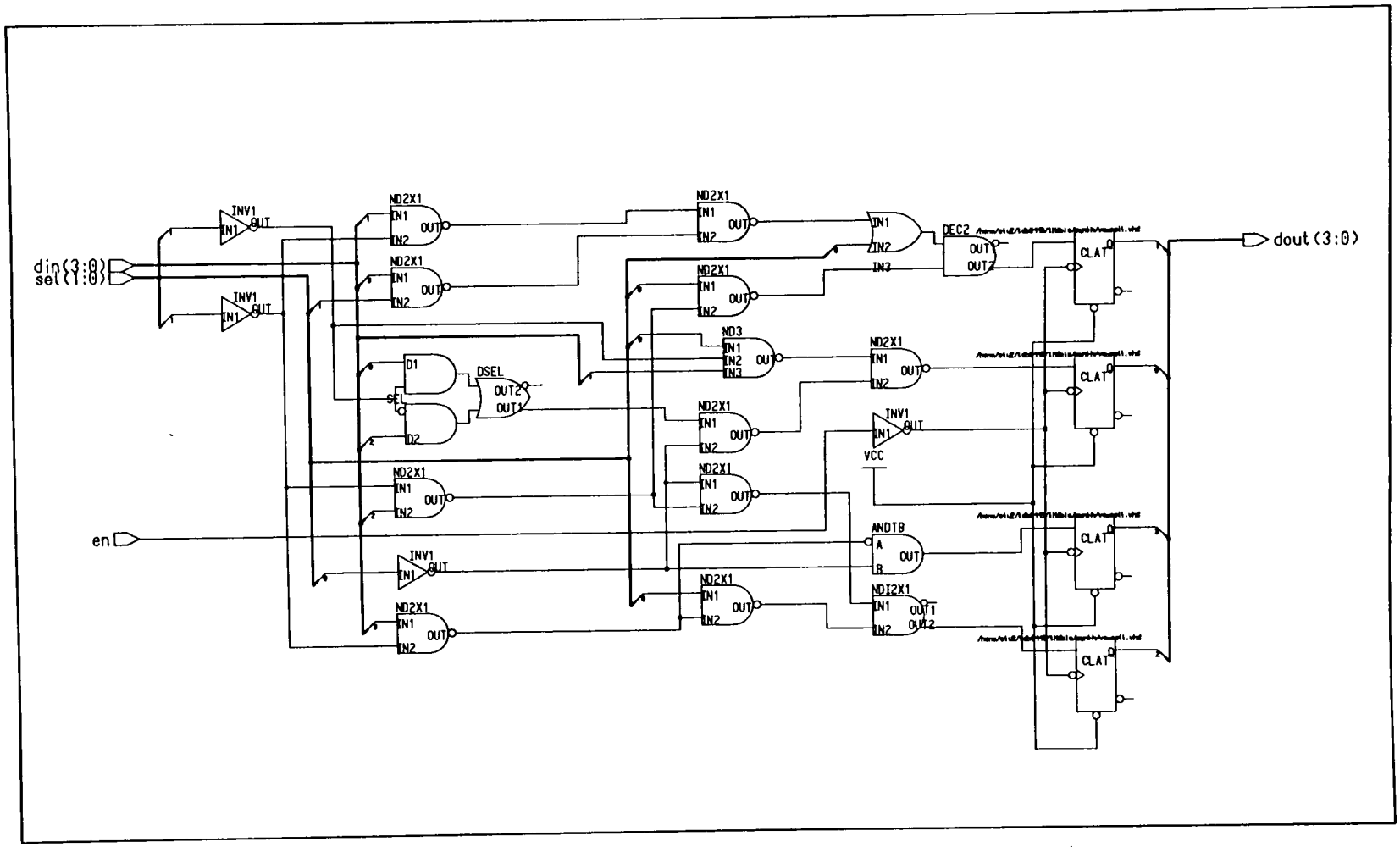

Figure 5-1 : Logic circuit for the SHIFT1VC component of the Vc-cell output function.

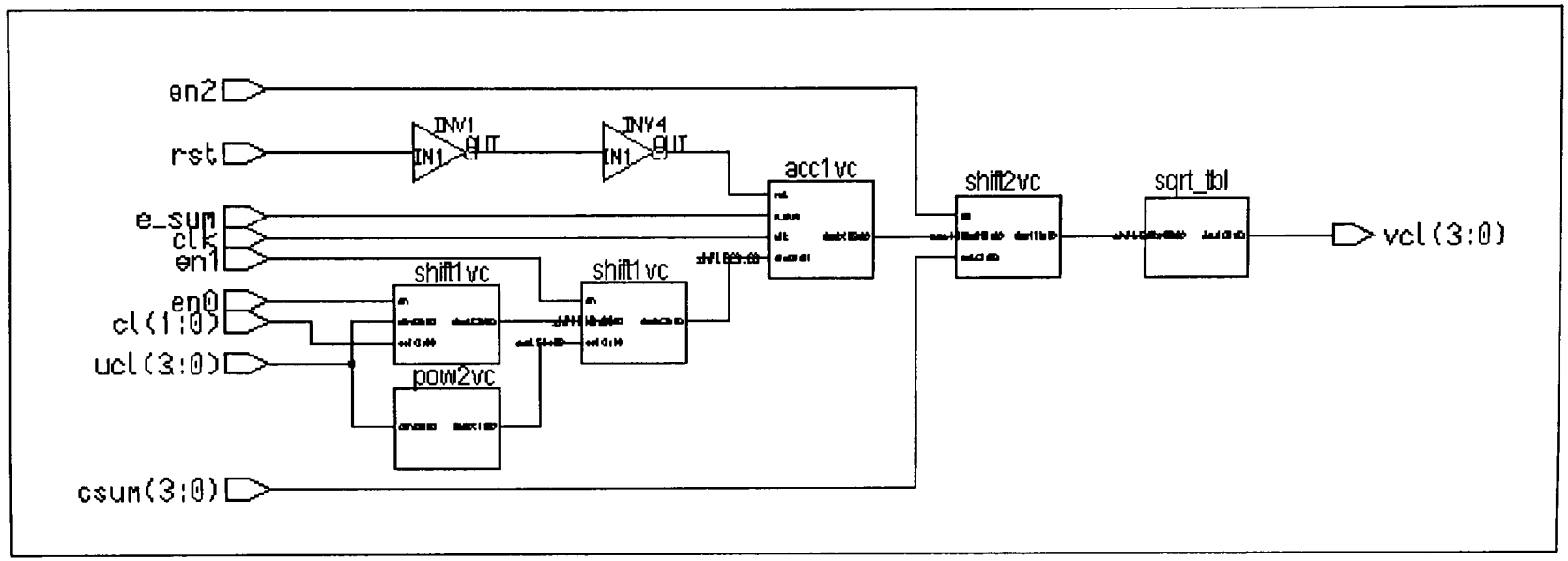

Figure 5-2 : Circuit schematic diagram for the Vc-cell output function of the digi-neocognitron.

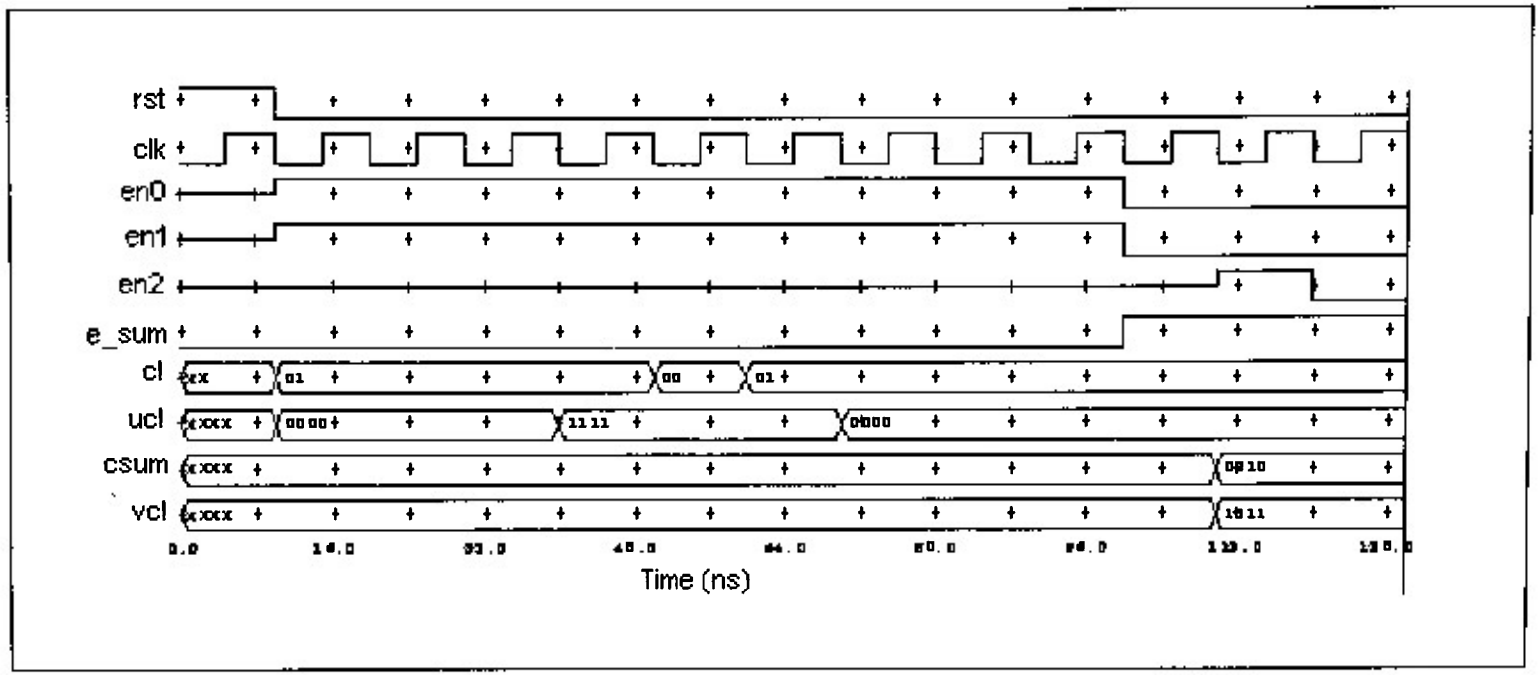

Figure 5-3 : Logic simulation results for the Vc-cell output function of the digi-neocognitron.

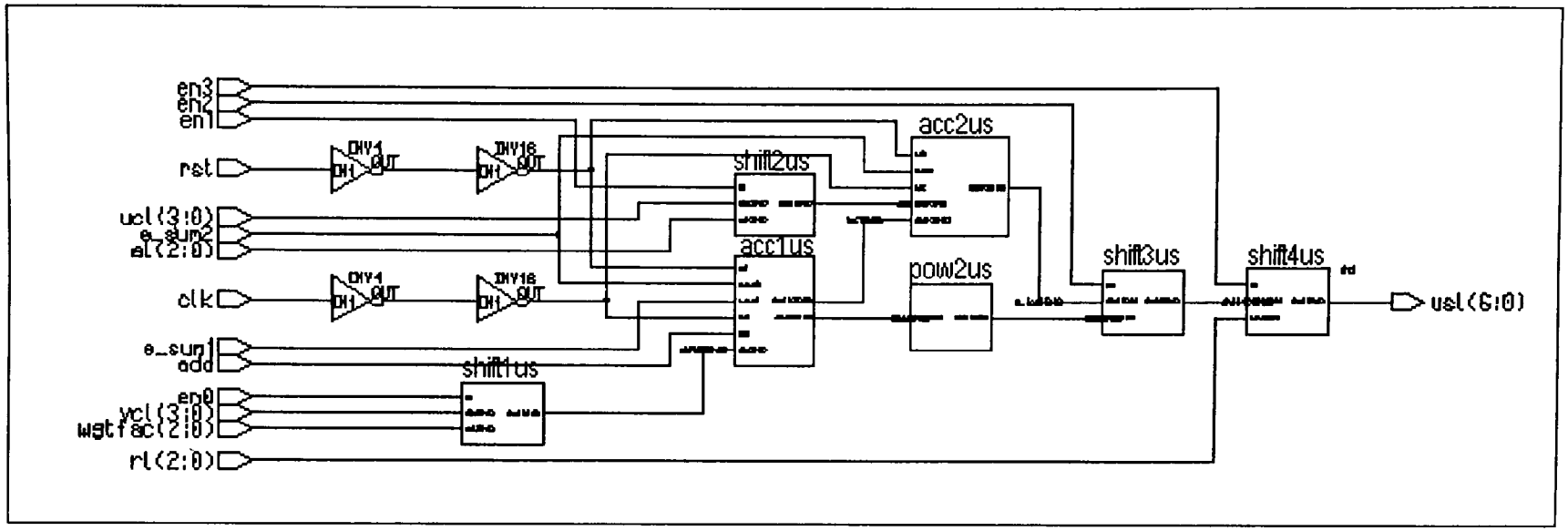

Figure 5-4 : Circuit schematic diagram for the S-cell output function of the digi-neocognitron.

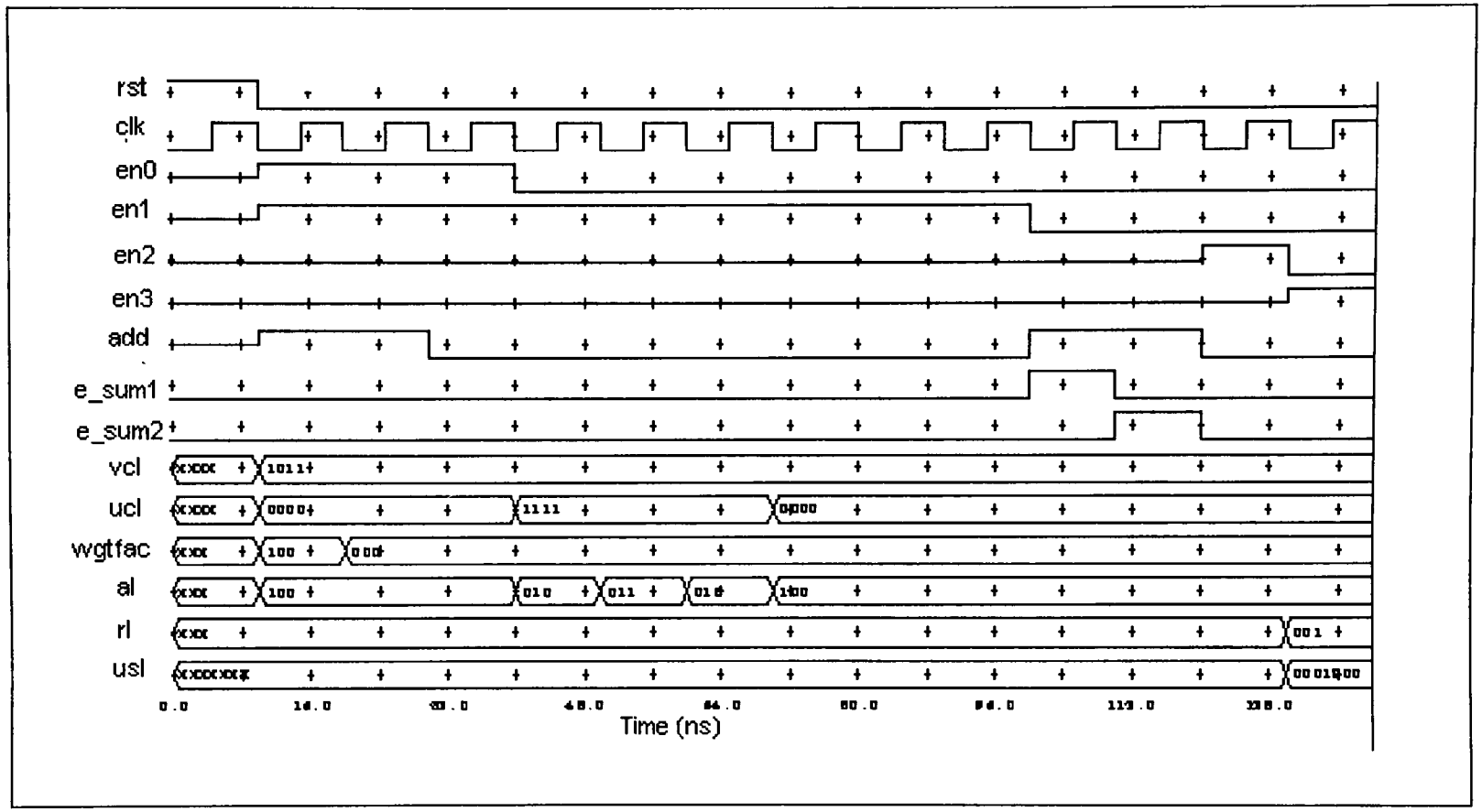

Figure 5-5 : Logic simulation results for the S-cell output function of the digi-neocognitron.

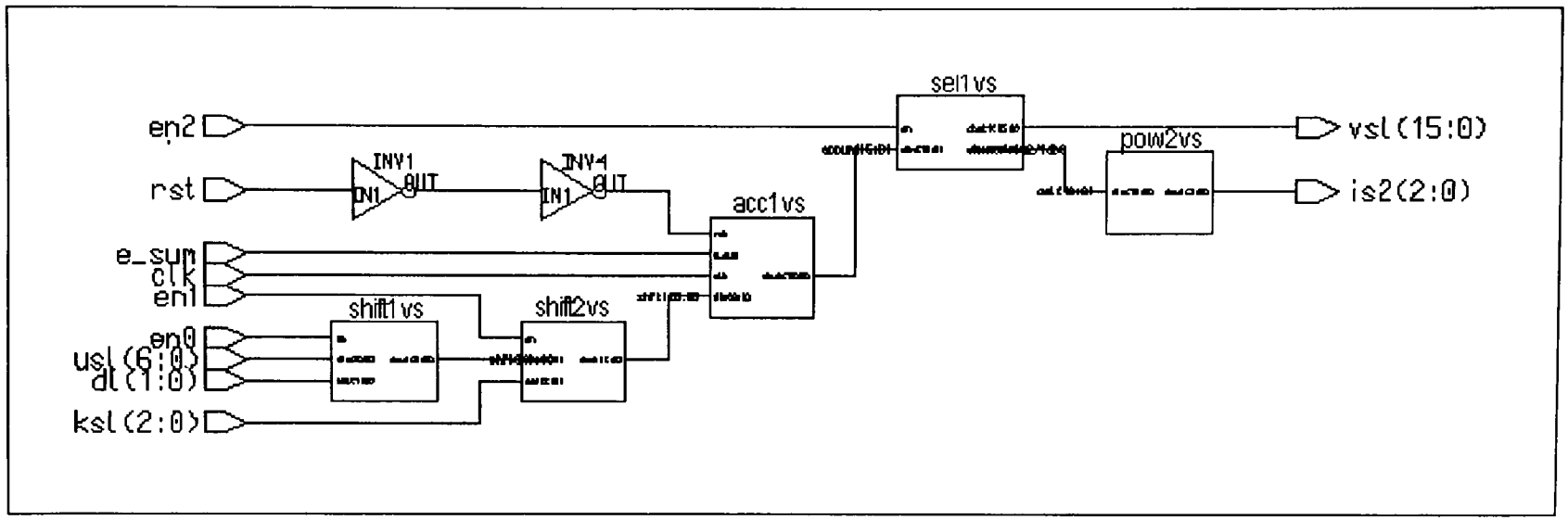

Figure 5-6 : Circuit schematic diagram for the Vs-cell output function of the digi-neocognitron.

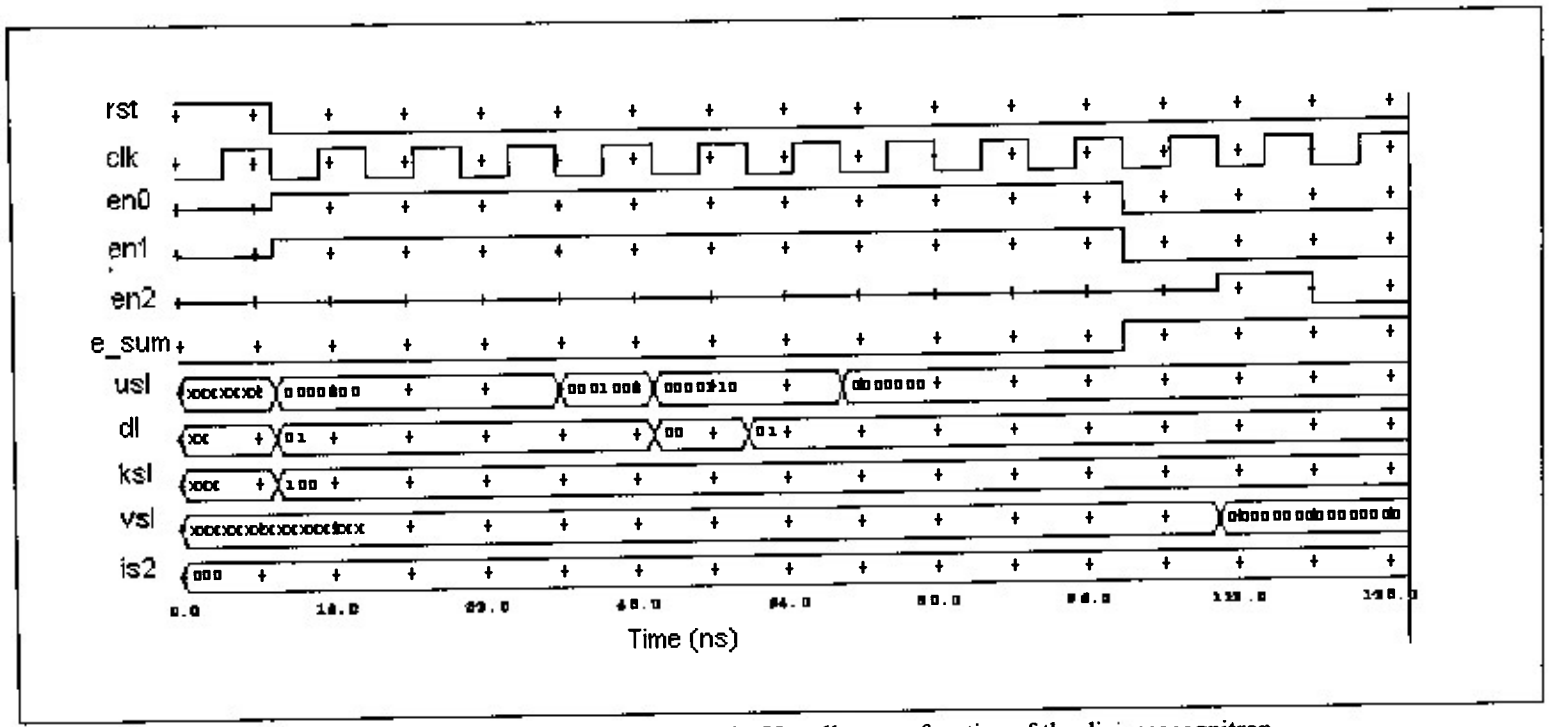

Figure 5-7 : Logic simulation results for the Vs-cell output function of the digi-neocognitron.

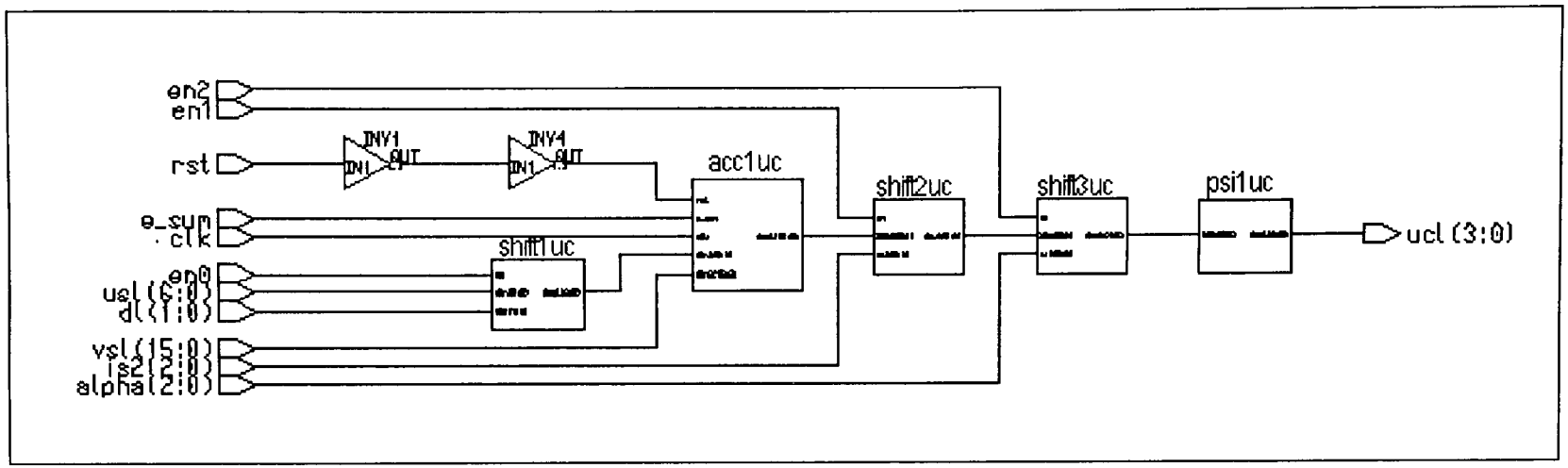

Figure 5-8 : Circuit schematic diagram for the C-cell output function of the digi-neocognitron.

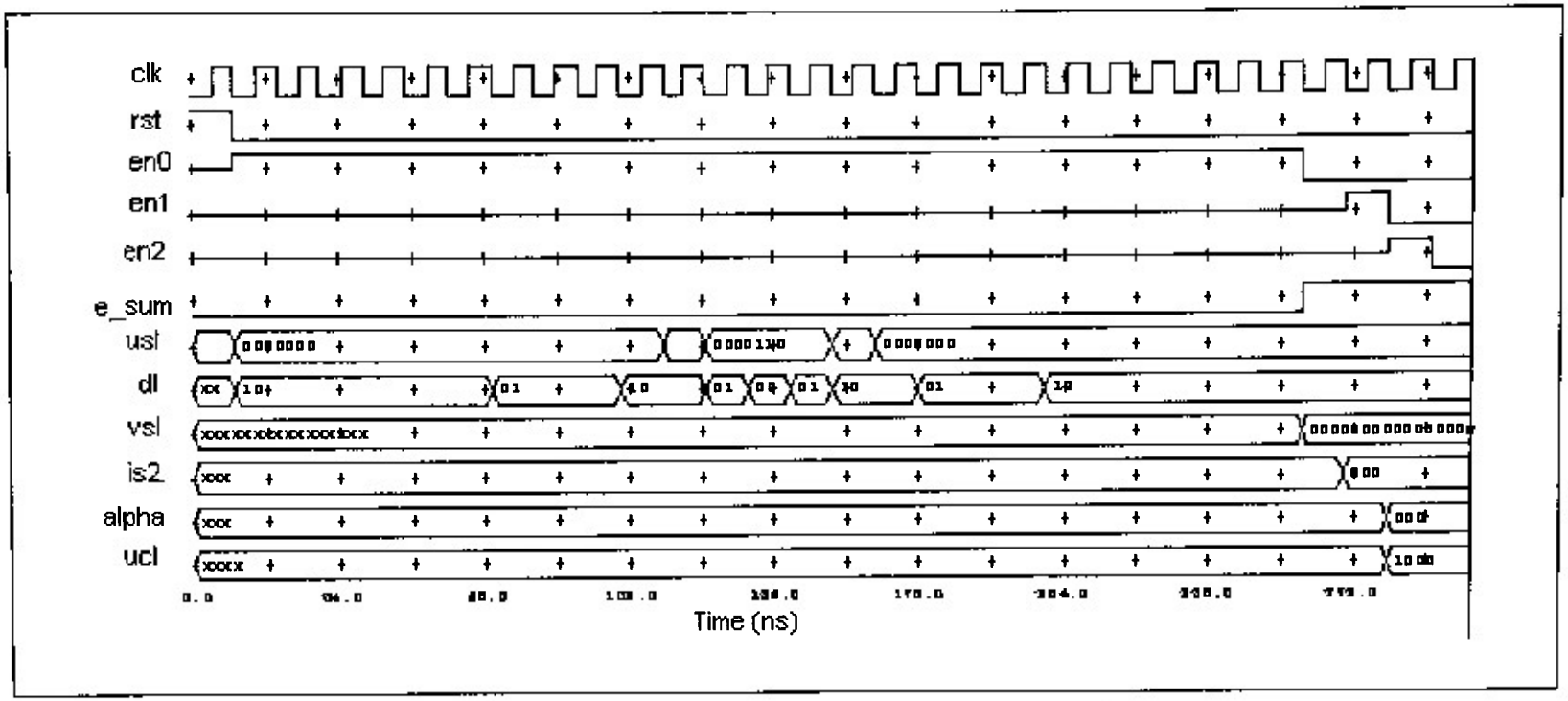

Figure 5-9 : Logic simulation results for the C-cell output function of the digi-neocognitron.

CONCLUSION

This thesis discussed an artificial neural network model, the neocognitron, for hand-written character recognition, and its adaptation to an all digital implementation for VLSI. The new model, the digi-neocognitron, was derived from the neocognitron model through preprocessing approximations and definition of new model functions. Multiplications and divisions are eliminated by converting the factors to powers of 2, so that only shift operations are needed. Also, complex functions, square and square-root, are replaced by look-up tables which can be implemented with simple combinatorial logic.

A software model was developed for the neocognitron. This model demonstrated an excellent pattern recognition capability for numerals 0 through 9 despite variations in size, shape, position on the input plane, and the presence of deformations from the training patterns. The neocognitron model was used to develop the digi-neocognitron model using the conversion methodology presented. However, the digi-neocognitron model's character recognition capability was less than that achieved for the neocognitron. Since the digi-neocognitron as a viable alternative to the neocognitron is dependent on it having comparable performance; the DNC model developed would not be used as a replacement for the neocognitron.

The poor performance obtained for the DNC was due to the supervised training method used, the set of training patterns, and the need for further refinement of the weight values developed. Because the weights are developed off-line, the DNC's performance does not negate the benefits offered by the digi-neocognitron for VLSI implementation. The digital VLSI model can be implemented and the weight values, which are stored using RAMs, EPROMs or EEPROMs, can be changed at anytime to utilize weights with a better pattern recognition capability.

To demonstrate the feasibility of a VLSI implementation for the digi-neocognitron a VHDL model was constructed. This model successfully demonstrated the digi-neocognitron's ability for hand-written character recognition, and highlighted opportunities for reducing the

time required to process a character. By sequentially computing the planes in each layer, a single character required approximately 8,898 μs using a clock with a period of 10 ns. A significant improvement in processing speed is attainable if these calculations are done concurrently. Concurrently calculating the outputs of the cell planes in each layer would result in approximately a processing time of 839 μs.

The model functions of the digi-neocognitron were synthesized using Mentor Graphics' AutoLogic, for a 1.2 μm CMOS process. These functions are comprised primarily of datapath operators (shifters, adders, subtractors and comparators), making them ideal for logic synthesis. The functionality of the synthesized circuits was verified using the QuickSim logic simulator. Logic simulation was also used to gain confidence in the AutoLogic tool.

Despite the performance shortfall of the digi-neocognitron model, it is more suitable for VLSI implementation than the neocognitron model. VHDL proved to be an invaluable tool in the implementation of the digi-neocognitron model using digital VLSI. Many of the architectural and design issues could be resolved in the simulation model prior to designing any actual hardware. In addition, logic synthesis tools can utilize the VHDL models to synthesize logic, transistor level and VLSI masks. With a reliable synthesis tool and properly designed VHDL models, circuit synthesis can significantly improve the designed process.

APPENDIX A

This section contains the source code that was required to implement and test the neocognitron and digi-neocognitron model functions. Appendix A-1 and A-2 lists the C source code for the neocognitron and digi-neocognitron model functions, respectively. In section A-3 is the listings for the VHDL models required to implement the processing blocks for constructing these cells in hardware. These processing blocks are described in Chapter 4. Finally, section A-4 lists the VHDL test bench models that were used to verify the functionality of the digi-neocognitron output functions shown in section A-3.

A-1 C Source Code for the Neocognitron Model Functions

```
/* vc_compute: algorithm to compute output of a single Vc-cell
*/
float vc_compute (x, y, layer)
int x, y, layer;
{
   int cplane, row, col, index;
   float csum, vc_cell;
   int *xptr, *yptr;

   vc_cell= csum = 0.0;
   xptr = &x;
   yptr = &y;
   get_cfield(layer, xptr, yptr);

   for (cplane=0; cplane<num_C[layer-1]; cplane++) {
      csum = csum + compute_csum(cl[layer-1], cfield[layer-1]);
      index = 0;
      for (row=x; row<x+cfield[layer-1]; row++) {
         for (col=y; col<y+cfield[layer-1]; col++) {
            vc_cell += (cl[layer-1][index] *
                     pow(uC[layer-1][cplane][row][col], 2.0));
            index++;
         }
      }
   }
   vc_cell = sqrt((1.0 / csum) * vc_cell);
   return (vc_cell);
}



/* us_compute: algorithm to compute the output of a single S-
cell */
float us_compute(x, y, layer, splane)
int x, y, layer, splane;
{
   int cplane, row, col, index, i, j;
   float inhib, select, s_cell;
   int *xptr, *yptr;

   s_cell = 0.0;
   i = x;
   j = y;
   xptr = &x;
   yptr = &y;
   get_cfield(layer, xptr, yptr);

   for (cplane=0; cplane<num_C[layer-1]; cplane++) {
      index = 0;
      for (row=x; row<x+cfield[layer-1]; row++) {
         for (col=y; col<y+cfield[layer-1]; col++) {
            s_cell +=  (al[layer-1][splane][cplane][index] *
                              uC[layer-1][cplane][row][col]);
            index++;
         }
      }
   }
   s_cell = s_cell + 1.0;
   select = rl[layer-1] / (1.0 + rl[layer-1]);
   inhib = 1.0 + (select * bl[layer-1][splane] *
                     vC[layer-1][i][j]);
   s_cell = (s_cell / inhib) - 1.0;

   /* linear threshold function */
   if (s_cell > 0.0)
      s_cell = rl[layer-1] * s_cell;
   else
      s_cell = 0.0;
   return(s_cell);
}



/* vs_compute: algorithm to compute the output of a single Vs-
cell */
float vs_compute(x, y, layer, posx, posy)
int x, y, layer, posx, posy;
{
   int splane, row, col, index;
   float vs_cell;
   int *xptr, *yptr;

   vs_cell = 0.0;
   xptr = &x;
   yptr = &y;
   get_sfield(layer, xptr, yptr, posx, posy);

   for (splane=0; splane<num_S[layer-1]; splane++) {
      index = 0;
      for (row=x; row<x+sfield[layer-1]; row++) {
         for (col=y; col<y+sfield[layer-1]; col++) {
            vs_cell += (dl[layer-1][index] *
                        uS[layer-1][splane][row][col]);
            index++;
         }
```

```
      }
   }
   return (vs_cell / num_S[layer-1]);
}


/* uc_compute: algorithm to compute the output of a single C-
cell */
float uc_compute(x, y, layer, cplane, posx, posy)
int x, y, layer, cplane, posx, posy;
{
   int row, col, splane, index;
   float c_cell, vs_cell;
   int *xptr, *yptr;

   if ( layer == LMAX )
      vs_cell = vS[x][y];
   else
      vs_cell = 0.0;

   c_cell = 0.0;

   xptr = &x;
   yptr = &y;
   get_sfield(layer, xptr, yptr, posx, posy);

   /* combines appropriate S-layers */
   for (splane=0; splane<num_S[layer-1]; splane++) {
      if (jarray[layer-1][cplane][splane] == 1) {
         index = 0;
         for (row=x; row<x+sfield[layer-1]; row++) {
            for (col=y; col<y+sfield[layer-1]; col++) {
               c_cell += (dl[layer-1][index] *
                         uS[layer-1][splane][row][col]);
               index++;
            }
         }
      }
   }
   c_cell = ((1.0 + c_cell) / (1.0 + vs_cell)) - 1.0;

   /* linear threshold function */
   if (c_cell > 0.0)
      c_cell = c_cell / (alpha1[layer-1] + c_cell);
   else
      c_cell = 0.0;

   return(c_cell);
}
```

A-2 C Source Code for the Digi-Neocognitron Model Functions

```
/* vc_compute2: computes the output of the Vc-cells for the DNC
model*/
float vc_compute2(x, y, layer)
int x, y, layer;
{
   int cplane, row, col, index;
   float csum, vc_cell, ucl;
   int *yptr, *xptr;

   vc_cell= csum = 0.0;
   xptr = &x;
   yptr = &y;
   get_cfield(layer, xptr, yptr);

   for (cplane=0; cplane<num_C[layer-1]; cplane++) {
      csum += compute_csum(cl[layer-1], cfield[layer-1]);
      index = 0;
      for (row=x; row<x+cfield[layer-1]; row++) {
         for (col=y; col<y+cfield[layer-1]; col++) {
            ucl = pow2_approx(uC[layer-1][cplane][row][col]);
            vc_cell += (cl[layer-1][index] * ucl);
            index++;
         }
      }
   }
   csum = approx_csum(csum, layer);
   vc_cell = vc_cell / csum;
   vc_cell = sqroot_approx(vc_cell);

   return (vc_cell);
}



/* us_compute2: computes the output of the S-cells of the DNC
model */
float us_compute2(x, y, layer, splane)
int x, y, layer, splane;
{
   int cplane, i, j;
   float weight_factor, s_cell, select, csum = 0.0;
   float E, IC, IC2, low, high1, high2;
   int *yptr, *xptr, row, col, index;

   low = 1.0;
   high1 = 16.0;
   high2 = 64.0;
   E = s_cell = 0.0;

   i = x;
   j = y;
   xptr = &x;
   yptr = &y;
   get_cfield(layer, xptr, yptr);

   /* perform computation of all cells in the cell plane */
   for (cplane=0; cplane<num_C[layer-1]; cplane++) {
      csum +=  compute_csum(cl[layer-1], cfield[layer-1]);
      index = 0;;
      for (row=x; row<x+cfield[layer-1]; row++) {
         for (col=y; col<y+cfield[layer-1]; col++) {
            E += (al[layer-1][splane][cplane][index] *
                                 uC[layer-1][cplane][row][col]);
            index++;
         }
      }
   }

/* determine the value of the inhibitory input to the S-cell */
   weight_factor = (wscale[layer-1] * rl[layer-1] *
                    bl[layer-1][splane]) / (1.0 + rl[layer-1]);
   if (layer == 1) {
      if (weight_factor <= low)
         weight_factor = low;
      else if (weight_factor >= high2)
         weight_factor = high2;
      else
         weight_factor = approximate(int_powers, EQ,
weight_factor);
   }
   else {
      if (weight_factor <= low)
         weight_factor = low;
      else
         weight_factor = two_term_approx(high2, weight_factor);
   }
   IC = weight_factor * vC[layer-1][i][j];
   IC2 = approx_ICS2(IC);

   /* determine S-cell output */
   if (E > IC) {
      if (rl[layer-1] <= low)
         select = low;
      else if (rl[layer-1] >= high1)
         select = high1;
```

```
      else {
         if (rl[layer-1] > 1.0)
            select = approximate(int_powers, GE, rl[layer-1]);
         else
            select = approximate(int_powers, LE, rl[layer-1]);
      }
      s_cell = select * ((E - IC) / IC2);
   }
   return (s_cell);
}


/* vs_compute2: computes the output of the Vs-cells in the DNC
model */
float vs_compute2(x, y, layer, posx, posy)
int x, y, layer, posx, posy;
{
   float vs_cell, numS, low, high;
   int index, splane, row, col, *yptr, *xptr;

   vs_cell = 0.0;
   low = 1.0;
   high = 64.0;

   xptr = &x;
   yptr = &y;
   get_sfield(layer, xptr, yptr, posx, posy);

   /* perform computation of all cells in the cell plane */
   for (splane=0; splane<num_S[layer-1]; splane++) {
      index = 0;
      for (row=x; row<x+sfield[layer-1]; row++) {
         for (col=y; col<y+sfield[layer-1]; col++) {
            vs_cell += (dl[layer-1][index] *
                              uS[layer-1][splane][row][col]);
            index++;
         }
      }
   }

   /* approximate Vs-cell output */
   numS = (float) num_S[layer-1];
   if (numS <= low)
      numS = low;
   else if (numS >= high)
      numS = high;
   else
      numS = approximate(int_powers, LE, numS);

   return (vs_cell / numS);
}
```

```
/* uc_compute2: computes the output of the C-cells of the DNC
model */
float uc_compute2(x, y, layer, cplane, posx, posy)
int x, y, layer, cplane, posx, posy;
{
   float c_cell, low, high, E, IS, IS2;
   float alpha, *array, z;
   int index, splane, row, col, *yptr, *xptr;

   if (layer == LMAX)
      IS = vS[x][y];
   else
      IS = 0.0;

   E = 0.0;
   low = 0.03125;
   high = 1.0;

   xptr = &x;
   yptr = &y;
   get_sfield(layer, xptr, yptr, posx, posy);

   /* perform computation of all cells in the cell plane */
   for (splane=0; splane<num_S[layer-1]; splane++) {
      if (jarray[layer-1][cplane][splane] == 1) {
         index = 0;
         for (row=x; row<x+sfield[layer-1]; row++) {
            for (col=y; col<y+sfield[layer-1]; col++) {
               E += (dl[layer-1][index] *
                              uS[layer-1][splane][row][col]);
               index++;
            }
         }
      }
   }

   /* approximate C-cell output */
   IS2 = approx_ICS2(IS);
   if (E > IS) {
      c_cell = (E - IS) / IS2;
      if (alpha2[layer-1] < 1.0)
         array = frac_powers;
      else
         array = int_powers;
      if (alpha2[layer-1] <= low)
         alpha = low;
      else if (alpha2[layer-1] >= high)
         alpha = high;
      else
         alpha = approximate(array, LE, alpha2[layer-1]);
```

```
      z = c_cell / alpha;
      c_cell = approx_PSI(z / (1.0 + z));
   }
   else
      c_cell = 0.0;

   return (c_cell);
}
```

A-3 VHDL Source Code for the Digi-Neocognitron Model Functions

```
------------------------------------------------------------
-- VHDL components used to model the Vc-cell output function --
------------------------------------------------------------

library mgc_portable;
use mgc_portable.qsim_logic.all;

-- shift1vc Entity Description
entity shift1vc is
   port(en   : in qsim_state;
        din  : in qsim_state_vector(3 downto 0);
        sel  : in qsim_state_vector(1 downto 0);
        dout : out qsim_state_vector(3 downto 0));
end shift1vc;

-- shift1vc Architecture Description
architecture rtl of shift1vc is
   subtype shiftbus is qsim_state_vector (7 downto 0);
begin
   shifter_PROCESS : process(en, din, sel)
      variable s0, s1, s2 : shiftbus;
   begin
      if (en = '1') then
         s2(7 downto 4) := (others => '0');
         s2(3 downto 0) := din;

         -- Build shifting muxes
         if (sel = "11") then
            s0(3 downto 0) := (others => '0');     -- Force to 0
         else
            if (sel(1) = '0') then
               s1 := s2;
            else
               s1(5 downto 0) := s2(7 downto 2);  -- Shift by 2
            end if;
            if (sel(0) = '0') then
               s0 := s1;
            else
               s0(6 downto 0) := s1(7 downto 1);  -- Shift by 1
            end if;
         end if;

         -- assign right shift output
         dout <= s0(3 downto 0);
      end if;
   end process shifter_PROCESS;
end rtl;



library mgc_portable;
use mgc_portable.qsim_logic.all;

-- pow2vc Entity Description
entity pow2vc is
   port(din  : in qsim_state_vector(3 downto 0);
        dout : out qsim_state_vector(1 downto 0));
end pow2vc;

-- pow2vc Architecture Description
architecture rtl of pow2vc is
begin
   power_PROCESS : process(din)
      variable na, b, nb, c, nc, d, nd : qsim_state;
      variable pow : qsim_state_vector(1 downto 0);
   begin
      na := not din(3);
      b  := din(2);
      nb := not din(2);
      c  := din(1);
      nc := not din(1);
      d  := din(0);
      nd := not din(0);

      pow(1) := (na and nb) or (na and nc and nd);
      pow(0) := (nb and nc and nd ) or (na and nc and d)
                  or (nc and b and d)  or (na and c and nd);

      dout <= pow;
   end process power_PROCESS;
end rtl;



library mgc_portable;
use mgc_portable.qsim_logic.all;

-- acc1vc Entity Description
entity acc1vc is
   port(rst   : in qsim_state;
        clk   : in qsim_state;
        e_sum : in qsim_state;
        din   : in qsim_state_vector(3 downto 0);
        dout  : out qsim_state_vector(15 downto 0));
end acc1vc;
```

```
-- acclvc Architecture Description
architecture rtl of acclvc is
   signal reg : qsim_state_vector(15 downto 0);
begin
   adder : process(clk, rst)
   begin
      if (rst = '1') then
         reg <= (others => '0');
      elsif (clk = '1' and clk'event and clk'last_value = '0')
then
         if (e_sum = '1') then
            dout <= reg;
         else
            reg <= reg + din;
         end if;
      end if;
   end process adder;
end rtl;
```

```
library mgc_portable;
use mgc_portable.qsim_logic.all;

-- shift2vc Entity Description
entity shift2vc is
   port(en   : in qsim_state;
        din  : in qsim_state_vector(15 downto 0);
        sel  : in qsim_state_vector(3 downto 0);
        dout : out qsim_state_vector(3 downto 0));
end shift2vc;

-- shift2vc Architecture Description
architecture rtl of shift2vc is
   subtype shiftbus is qsim_state_vector (31 downto 0);
begin
   shifter_PROCESS : process(en, din, sel)
      variable s0, s1, s2, s3, s4 : shiftbus;
   begin
      if (en = '1') then
         s4(31 downto 16) := (others => '0');
         s4(15 downto 0) := din;

         -- Build shifting muxes
         if (sel(3) = '0') then
            s3 := s4;
         else
            s3(23 downto 0) := s4(31 downto 8);   -- Shift by 8
         end if;
         if (sel(2) = '0') then
            s2 := s3;
         else
            s2(27 downto 0) := s3(31 downto 4);   -- Shift by 4
         end if;
         if (sel(1) = '0') then
            s1 := s2;
         else
            s1(29 downto 0) := s2(31 downto 2);   -- Shift by 2
         end if;
         if (sel(0) = '0') then
         s0 := s1;
         else
            s0(30 downto 0) := s1(31 downto 1);   -- Shift by 1
         end if;

         -- assign right shift output
         dout<= s0(3 downto 0);
      end if;
   end process shifter_PROCESS;
end rtl;
```

```
library mgc_portable;
use mgc_portable.qsim_logic.all;

-- sqrt_tbl Entity Description
entity sqrt_tbl is
   port(din  : in qsim_state_vector(3 downto 0);
        dout : out qsim_state_vector(3 downto 0));
end sqrt_tbl;

-- sqrt_tbl Architecture Description
architecture rtl of sqrt_tbl is
begin
   sqrt_PROCESS : process(din)
      variable a, na, b, nb, c, nc, d, nd : qsim_state;
      variable root : qsim_state_vector(3 downto 0);
   begin
      a  := din(3);
      na := not din(3);
      b  := din(2);
      nb := not din(2);
      c  := din(1);
      nc := not din(1);
      d  := din(0);
      nd := not din(0);

      root(3) := a or b or (c and d);
      root(2) := (a and b) or (a and d) or (nb and c and nd)
                     or (na and nb and nc);
```

```
      root(1) := (a and b) or (b and c) or (a and nc and nd)
             or (na  and nb and nc and d) or (na and c and nd);
      root(0) := (a and c) or (na and b and d) or
                         (nb and c and nd) or (a and nb and nd);
      dout <= root;
   end process sqrt_PROCESS;
end rtl;



library mgc_portable;
use mgc_portable.qsim_logic.all;
use work.all;

-- vc_cell Entity Description
entity vc_cell is
   port(rst   : in qsim_state;
        clk   : in qsim_state;
        en0   : in qsim_state;
        en1   : in qsim_state;
        en2   : in qsim_state;
        e_sum : in qsim_state;
        cl    : in qsim_state_vector(1 downto 0);
        ucl   : in qsim_state_vector(3 downto 0);
        csum  : in qsim_state_vector(3 downto 0);
        vcl   : out qsim_state_vector(3 downto 0));
end vc_cell;

-- vc_cell Architecture Description
architecture rtl of vc_cell is
   component shiftlvc
      port(en   : in qsim_state;
           din  : in qsim_state_vector(3 downto 0);
           sel  : in qsim_state_vector(1 downto 0);
           dout : out qsim_state_vector(3 downto 0));
   end component;
   component pow2vc
      port(din  : in qsim_state_vector(3 downto 0);
           dout : out qsim_state_vector(1 downto 0));
   end component;
   component acclvc
      port(rst   : in qsim_state;
           clk   : in qsim_state;
           e_sum : in qsim_state;
           din   : in qsim_state_vector(3 downto 0);
           dout  : out qsim_state_vector(15 downto 0));
   end component;
   component shift2vc
      port(en   : in qsim_state;
           din  : in qsim_state_vector(15 downto 0);
           sel  : in qsim_state_vector(3 downto 0);
           dout : out qsim_state_vector(3 downto 0));
   end component;
   component sqrt_tbl
      port(din  : in qsim_state_vector(3 downto 0);
           dout : out qsim_state_vector(3 downto 0));
   end component;

   signal shftl : qsim_state_vector(3 downto 0);
   signal shft2 : qsim_state_vector(3 downto 0);
   signal shft3 : qsim_state_vector(3 downto 0);
   signal sel   : qsim_state_vector(1 downto 0);
   signal acc   : qsim_state_vector(15 downto 0);
begin
   shiftl : shiftlvc port map(en0, ucl, cl, shftl);
   power2 : pow2vc   port map(ucl, sel);
   shift2 : shiftlvc port map(en1, shftl, sel, shft2);
   accum1 : acclvc   port map(rst, clk, e_sum, shft2, acc);
   shift3 : shift2vc port map(en2, acc, csum, shft3);
   sqroot : sqrt_tbl port map(shft3, vcl);
end rtl;
```

```
-----------------------------------------------------------------
-- VHDL components used to model the S-cell output function  --
-----------------------------------------------------------------

library mgc_portable;
use mgc_portable.qsim_logic.all;

-- shiftlus entity description
entity shiftlus is
   port(en   : in qsim_state;
        din  : in qsim_state_vector(3 downto 0);
        sel  : in qsim_state_vector(2 downto 0);
        dout : out qsim_state_vector(9 downto 0));
end shiftlus;

-- shiftlus architecture description
architecture rtl of shiftlus is
   subtype shiftbus is qsim_state_vector (19 downto 0);
begin

   shifter_PROCESS: process(en, din, sel)
      variable s0, s1, s2, s3 : shiftbus;
      variable dist : integer range 0 to 7;
   begin
      s3(19 downto 10) := "000000" & din;
      s3(9 downto 0) := (others => '0');

      -- build shifting muxes
      if (sel(2) = '0') then
         s2 := s3;
      else
         s2(19 downto 4) := s3(15 downto 0);      -- shift by 4
      end if;
      if (sel(1) = '0') then
         s1 := s2;
      else
         s1(19 downto 2) := s2(17 downto 0);      -- shift by 2
      end if;
      if (sel(0) = '0') then
         s0 := s1;
      else
         s0(19 downto 1) := s1(18 downto 0);      -- shift by 1
      end if;
      if (sel = "000") then
         s0 := (others => '0');
      end if;

      -- check for valid shift distances and assign shift left
output
      if (en = '1') then
         dist := to_integer('0' & sel,0);
         if (dist > 4) then
            dout <= (others => '0');
         else
            dout <= s0(19 downto 10);
         end if;
      end if;
   end process shifter_PROCESS;
end rtl;



library mgc_portable;
use mgc_portable.qsim_logic.all;

-- shift2us entity description
entity shift2us is
   port(en   : in qsim_state;
        din  : in qsim_state_vector(3 downto 0);
        sel  : in qsim_state_vector(2 downto 0);
        dout : out qsim_state_vector(9 downto 0));
end shift2us;

-- shift2us architecture description
architecture rtl of shift2us is
   subtype shiftbus is qsim_state_vector (19 downto 0);
begin
   shifter_PROCESS : process(en, din, sel)
      variable s0, s1, s2 : shiftbus;
   begin
      if (sel(2) = '1') then      -- Build right shifting muxes
         s2(19 downto 10) := (others => '0');
         s2(9 downto 0) := "000000" & din;

         if (sel(1) = '0') then
            s1 := s2;
         else
            s1(17 downto 0) := s2(19 downto 2);  -- Shift by 2
         end if;
         if (sel(0) = '0') then
            s0 := s1;
         else
            s0(18 downto 0) := s1(19 downto 1);   -- Shift by 1
         end if;
         if (sel(1) = '0' and sel(0) = '0') then
            s0(9 downto 0) := (others => '0');    -- Force 0
         end if;

         -- assign right shift output
         if (en = '1') then
            dout <= s0(9 downto 0);
```

```
        end if;

      elsif (sel(2) = '0') then       Build left shifting muxes
         s2(19 downto 10) := "000000" & din;
         s2(9 downto 0) := (others => '0');

         if (sel(1) = '0') then
            s1 := s2;
         else
            s1(19 downto 2) := s2(17 downto 0);    -- Shift by 2
         end if;
         if (sel(0) = '0') then
            s0 := s1;
         else
            s0(19 downto 1) := s1(18 downto 0);    -- Shift by 1
         end if;

         -- assign left shift output
         if (en = '1') then
            dout <= s0(19 downto 10);
         end if;
      end if;
   end process shifter_PROCESS;
end rtl;



library mgc_portable;
use mgc_portable.qsim_logic.all;

-- acc1us Entity Description
entity acc1us is
   port(rst    : in qsim_state;
        clk    : in qsim_state;
        add    : in qsim_state;
        e_sum1 : in qsim_state;
        e_sum2 : in qsim_state;
        din    : in qsim_state_vector(9 downto 0);
        dout1  : out qsim_state_vector(10 downto 0);
        dout2  : out qsim_state_vector(8 downto 0));
end acc1us;

-- acc1us Architecture Description
architecture rtl of acc1us is
   signal reg : qsim_state_vector(10 downto 0);
begin
   accum_PROCESS : process(rst, clk)
      variable sign : qsim_state;
   begin
      if (rst = '1') then
         reg <= (others => '0');
      elsif (clk = '1' and clk'event and clk'last_value = '0')
then
         if (e_sum1 = '1') then
            dout1 <= reg;
         elsif (e_sum2 = '1') then
            dout2 <= reg(8 downto 0);
         elsif (add = '1') then
            reg <= reg + din;
         end if;
      end if;
   end process accum_PROCESS;
end rtl;



library mgc_portable;
use mgc_portable.qsim_logic.all;
use work.package_utils.all;

-- acc2us Entity Description
entity acc2us is
   port(rst    : in qsim_state;
        clk    : in qsim_state;
        e_sum  : in qsim_state;
        din1   : in qsim_state_vector(9 downto 0);
        din2   : in qsim_state_vector(10 downto 0);
        dout   : out qsim_state_vector(15 downto 0));
end acc2us;

-- acc2us Architecture Description
architecture rtl of acc2us is
   signal reg : qsim_state_vector(15 downto 0);
begin
   accum_PROCESS : process(rst, clk)
      variable E, IC : integer;
   begin
      if (rst = '1') then
         reg <= (others => '0');
      elsif (clk = '1' and clk'event and clk'last_value = '0')
then
         if (e_sum = '1') then
            E := to_Integer2(reg);
            IC := to_Integer2(din2);
            if (E > IC) then
               dout <= reg - din2;
            else
               dout <= (others => '0');
            end if;
         else
            reg <= reg + din1;
         end if;
```

```
      end if;
   end process accum_PROCESS;
end rtl;



library mgc_portable;
use mgc_portable.qsim_logic.all;

-- pow2us Entity Description
entity pow2us is
   port(din  : in qsim_state_vector(8 downto 0);
        dout : out qsim_state_vector(2 downto 0));
end pow2us;

-- pow2us Architecture Description
architecture rtl of pow2us is
begin
   pow2_PROCESS : process(din)
      variable pow : qsim_state_vector(2 downto 0);
   begin
      if (din < "000001000") then
         pow := "000";
      elsif (din < "000100000") then
         pow := "001";
      elsif (din < "001001000") then
         pow := "010";
      elsif (din < "010100000") then
         pow := "011";
      elsif (din < "101010000") then
         pow := "100";
      else
         pow := "101";
      end if;

      dout <= pow;
   end process pow2_PROCESS;
end rtl;



library mgc_portable;
use mgc_portable.qsim_logic.all;

-- shift3us entity description
entity shift3us is
   port(en   : in qsim_state;
        din  : in qsim_state_vector(15 downto 0);
        sel  : in qsim_state_vector(2 downto 0);
        dout : out qsim_state_vector(10 downto 0));
end shift3us;
```

```
-- shift3us architecture description
architecture rtl of shift3us is
   subtype shiftbus is qsim_state_vector (31 downto 0);
begin
   shifter_PROCESS : process(en, din, sel)
      variable s0, s1, s2, s3 : shiftbus;
   begin
      s3(31 downto 16) := (others => '0');
      s3(15 downto 0) := din;

      -- Build shifting muxes
      if (sel(2) = '0') then
         s2 := s3;
      else
         s2(27 downto 0) := s3(31 downto 4);     -- Shift by 4
      end if;
      if (sel(1) = '0') then
         s1 := s2;
      else
         s1(29 downto 0) := s2(31 downto 2);     -- Shift by 2
      end if;
      if (sel(0) = '0') then
         s0 := s1;
      else
         s0(30 downto 0) := s1(31 downto 1);     -- Shift by 1
      end if;

      -- assign right shift output
      if (en = '1') then
         dout <= s0(10 downto 0);
      end if;
   end process shifter_PROCESS;
end rtl;



library mgc_portable;
use mgc_portable.qsim_logic.all;

-- shift4us entity description
entity shift4us is
   port(en   : in qsim_state;
        din  : in qsim_state_vector(10 downto 0);
        sel  : in qsim_state_vector(2 downto 0);
        dout : out qsim_state_vector(6 downto 0));
end shift4us;

-- shift4us architecture description
architecture rtl of shift4us is
   subtype shiftbus is qsim_state_vector (21 downto 0);
```

```
begin
   shifter_PROCESS : process(en, din, sel)
      variable s0, s1, s2, s3 : shiftbus;
   begin
      s3(21 downto 11) := din;
      s3(10 downto 0) := (others => '0');

      -- Build shifting muxes
      if (sel(2) = '0') then
         s2 := s3;
      else
         s2(21 downto 4) := s3(17 downto 0);    -- shift by 4
      end if;
      if (sel(1) = '0') then
         s1 := s2;
      else
         s1(21 downto 2) := s2(19 downto 0);    -- shift by 2
      end if;
      if (sel(0) = '0') then
         s0 := s1;
      else
         s0(21 downto 1) := s1(20 downto 0);     -- shift by 1
      end if;

      -- assign left shift output
      if (en = '1') then
         dout <= s0(17 downto 11);
      end if;
   end process shifter_PROCESS;
end rtl;



library mgc_portable;
use mgc_portable.qsim_logic.all;
use work.all;

-- us_cell entity Description
entity us_cell is
   port(rst    : in qsim_state;
        clk    : in qsim_state;
        en0    : in qsim_state;
        en1    : in qsim_state;
        en2    : in qsim_state;
        en3    : in qsim_state;
        add    : in qsim_state;
        e_sum1 : in qsim_state;
        e_sum2 : in qsim_state;
        vcl    : in qsim_state_vector(3 downto 0);
        ucl    : in qsim_state_vector(3 downto 0);
        wgtfac : in qsim_state_vector(2 downto 0);
        al     : in qsim_state_vector(2 downto 0);
        rl     : in qsim_state_vector(2 downto 0);
        usl    : out qsim_state_vector(6 downto 0));
end us_cell;

-- us_cell Architecture Description
architecture rtl of us_cell is
   component shift1us
      port(en   : in qsim_state;
           din  : in qsim_state_vector(3 downto 0);
           sel  : in qsim_state_vector(2 downto 0);
           dout : out qsim_state_vector(9 downto 0));
   end component;
   component shift2us
      port(en   : in qsim_state;
           din  : in qsim_state_vector(3 downto 0);
           sel  : in qsim_state_vector(2 downto 0);
           dout : out qsim_state_vector(9 downto 0));
   end component;
   component acc1us
      port(rst    : in qsim_state;
           clk    : in qsim_state;
           add    : in qsim_state;
           e_sum1 : in qsim_state;
           e_sum2 : in qsim_state;
           din    : in qsim_state_vector(9 downto 0);
           dout1  : out qsim_state_vector(10 downto 0);
           dout2  : out qsim_state_vector(8 downto 0));
   end component;
   component acc2us
      port(rst   : in qsim_state;
           clk   : in qsim_state;
           e_sum : in qsim_state;
           din1  : in qsim_state_vector(9 downto 0);
           din2  : in qsim_state_vector(10 downto 0);
           dout  : out qsim_state_vector(15 downto 0));
   end component;
   component pow2us
      port(din  : in qsim_state_vector(8 downto 0);
           dout : out qsim_state_vector(2 downto 0));
   end component;
   component shift3us
      port(en   : in qsim_state;
           din  : in qsim_state_vector(15 downto 0);
           sel  : in qsim_state_vector(2 downto 0);
           dout : out qsim_state_vector(10 downto 0));
   end component;
   component shift4us
      port(en   : in qsim_state;
           din  : in qsim_state_vector(10 downto 0);
           sel  : in qsim_state_vector(2 downto 0);
```

```
          dout : out qsim_state_vector(6 downto 0));
   end component;

   signal shft0 : qsim_state_vector(9 downto 0);
   signal shftl : qsim_state_vector(10 downto 0);
   signal acc   : qsim_state_vector(8 downto 0);
   signal IC2   : qsim_state_vector(2 downto 0);
   signal IC    : qsim_state_vector(10 downto 0);
   signal E     : qsim_state_vector(9 downto 0);
   signal E_IC  : qsim_state_vector(15 downto 0);
begin
   shiftl : shiftlus port map(en0, vcl, wgtfac, shft0);
   shift2 : shift2us port map(enl, ucl, al, E);
   acc0l  : acclus   port map(rst, clk, add, e_suml, e_sum2,
shft0, IC, acc);
   acc02  : acc2us   port map(rst, clk, e_sum2, E, IC, E_IC);
   power2 : pow2us   port map(acc, IC2);
   shift3 : shift3us port map(en2, E_IC, IC2, shftl);
   shift4 : shift4us port map(en3, shftl, rl, usl);
end rtl;
```

```
------------------------------------------------------------
-- VHDL components used to model the Vs-cell output function --
------------------------------------------------------------

library mgc_portable;
use mgc_portable.qsim_logic.all;

-- shiftlvs Entity Description
entity shiftlvs is
   port(en   : in qsim_state;
        din  : in qsim_state_vector(6 downto 0);
        sel  : in qsim_state_vector(1 downto 0);
        dout : out qsim_state_vector(6 downto 0));
end shiftlvs;

-- shiftlvs Architecture Description
architecture rtl of shiftlvs is
   subtype shiftbus is qsim_state_vector (13 downto 0);
begin
   shifter_PROCESS : process(en, din, sel)
      variable s0, s1, s2 : shiftbus;
   begin
      s2(13 downto 7) := (others => '0');
      s2(6 downto 0) := din;

      -- build shifting muxes
      if (sel = "11") then
         s0 := (others => '0');                    -- force to 0
      else
         if (sel(1) = '0') then
            s1 := s2;
         else
            s1(11 downto 0) := s2(13 downto 2);  -- shift by 2
         end if;
         if (sel(0) = '0') then
            s0 := s1;
         else
            s0(12 downto 0) := s1(13 downto 1);  -- shift by 1
         end if;
      end if;

      -- assign right shift output
      if (en = '1') then
         dout <= s0(6 downto 0);
      end if;
   end process shifter_PROCESS;
end rtl;
```

```
library mgc_portable;
use mgc_portable.qsim_logic.all;

-- shift2vs Entity Description
entity shift2vs is
   port(en   : in qsim_state;
        din  : in qsim_state_vector(6 downto 0);
        sel  : in qsim_state_vector(2 downto 0);
        dout : out qsim_state_vector(6 downto 0));
end shift2vs;

-- shift2vs Architecture Description
architecture rtl of shift2vs is
   subtype shiftbus is qsim_state_vector (13 downto 0);
begin
   shifter_PROCESS : process(en, din, sel)
      variable s0, s1, s2, s3 : shiftbus;
   begin
      s3(13 downto 7) := (others => '0');
      s3(6 downto 0) := din;

      -- build shifting muxes
      if (sel(2) = '0') then
         s2 := s3;
      else
         s2(9 downto 0) := s3(13 downto 4);    -- shift by 4
      end if;
      if (sel(1) = '0') then
         s1 := s2;
      else
         s1(11 downto 0) := s2(13 downto 2);   -- shift by 2
      end if;
      if (sel(0) = '0') then
         s0 := s1;
      else
         s0(12 downto 0) := s1(13 downto 1);   -- shift by 1
      end if;

      -- assign right shift output
      if (en = '1') then
         dout <= s0(6 downto 0);
      end if;
   end process shifter_PROCESS;
end rtl;



library mgc_portable;
use mgc_portable.qsim_logic.all;

-- acclvs Entity Description
```

```
entity acc1vs is
   port(rst   : in qsim_state;
        clk   : in qsim_state;
        e_sum : in qsim_state;
        din   : in qsim_state_vector(6 downto 0);
        dout  : out qsim_state_vector(15 downto 0));
end acc1vs;

-- acc1vs Architecture Description
architecture rtl of acc1vs is
   signal reg : qsim_state_vector(15 downto 0);
begin
   accum_PROCESS : process(clk, rst)
   begin
      if (rst = '1') then
         reg <= (others => '0');
      elsif (clk = '1' and clk'event and clk'last_value = '0')
then
         if (e_sum = '1') then
            dout <= reg;
         else
            reg <= reg + din;
         end if;
      end if;
   end process accum_PROCESS;
end rtl;



library mgc_portable;
use mgc_portable.qsim_logic.all;

-- sel1vs Entity Description
entity sel1vs is
   port(en    : in qsim_state;
        din   : in qsim_state_vector(15 downto 0);
        dout1 : out qsim_state_vector(15 downto 0);
        dout2 : out qsim_state_vector(10 downto 0));
end sel1vs;

-- sel1vs Architecture Description
architecture rtl of sel1vs is
begin
   latch_PROCESS : process(en, din)
   begin
      if (en = '1') then
         dout1 <= din;
         dout2 <= din(10 downto 0);
      end if;
   end process latch_PROCESS;
end rtl;
```

```
library mgc_portable;
use mgc_portable.qsim_logic.all;

-- pow2vs Entity Description
entity pow2vs is
   port(din  : in qsim_state_vector(10 downto 0);
        dout : out qsim_state_vector(2 downto 0));
end pow2vs;

-- pow2vs Architecture Description
architecture rtl of pow2vs is
begin
   pow2_PROCESS : process(din)
      variable f : qsim_state_vector(2 downto 0);
   begin
      if (din < "00000001000") then
         f := "000";
      elsif (din < "00000100000") then
         f := "001";
      elsif (din < "00001000000") then
         f := "010";
      elsif (din < "00010100000") then
         f := "011";
      elsif (din < "00101010000") then
         f := "100";
      else
         f := "101";
      end if;

      dout <= f;
   end process pow2_PROCESS;
end rtl;



library mgc_portable;
use mgc_portable.qsim_logic.all;
use work.all;

-- vs_cell Entity Description
entity vs_cell is
   port(rst   : in qsim_state;
        clk   : in qsim_state;
        en0   : in qsim_state;
        en1   : in qsim_state;
        en2   : in qsim_state;
        e_sum : in qsim_state;
        us1   : in qsim_state_vector(6 downto 0);
```

```
        dl    : in qsim_state_vector(1 downto 0);
        ksl   : in qsim_state_vector(2 downto 0);
        vsl   : out qsim_state_vector(15 downto 0);
        is2   : out qsim_state_vector(2 downto 0));
end vs_cell;

-- vs_cell Architecture Description
architecture rtl of vs_cell is
   component shiftlvs
      port(en   : in qsim_state;
           din  : in qsim_state_vector(6 downto 0);
           sel : in qsim_state_vector(1 downto 0);
           dout : out qsim_state_vector(6 downto 0));
   end component;
   component shift2vs
      port(en   : in qsim_state;
           din  : in qsim_state_vector(6 downto 0);
           sel  : in qsim_state_vector(2 downto 0);
           dout : out qsim_state_vector(6 downto 0));
   end component;
   component acclvs
      port(rst   : in qsim_state;
           clk   : in qsim_state;
           e_sum : in qsim_state;
           din   : in qsim_state_vector(6 downto 0);
           dout  : out qsim_state_vector(15 downto 0));
   end component;
   component sellvs
      port(en    : in qsim_state;
           din   : in qsim_state_vector(15 downto 0);
           dout1 : out qsim_state_vector(15 downto 0);
           dout2 : out qsim_state_vector(10 downto 0));
   end component;
   component pow2vs
      port(din  : in qsim_state_vector(10 downto 0);
           dout : out qsim_state_vector(2 downto 0));
   end component;

   signal shft0 : qsim_state_vector(6 downto 0);
   signal shft1 : qsim_state_vector(6 downto 0);
   signal sel   : qsim_state_vector(10 downto 0);
   signal accum : qsim_state_vector(15 downto 0);
begin
   -- link entities that make up vs-cell component
   shift1 : shiftlvs port map(en0, us1, d1, shft0);
   shift2 : shift2vs port map(en1, shft0, ks1, shft1);
   accum1 : acclvs   port map(rst, clk, e_sum, shft1, accum);
   selct1 : sellvs   port map(en2, accum, vs1, sel);
   power2 : pow2vs   port map(sel, is2);
end rtl;
```

```
------------------------------------------------------------
-- VHDL components used to model the C-cell output function  --
------------------------------------------------------------

library mgc_portable;
use mgc_portable.qsim_logic.all;

-- shiftluc Entity Description
entity shiftluc is
   port(en   :.in qsim_state;
        din  : in qsim_state_vector(6 downto 0);
        sel  : in qsim_state_vector(1 downto 0);
        dout : out qsim_state_vector(6 downto 0));
end shiftluc;

-- shiftluc Architecture Description
architecture rtl of shiftluc is
   subtype shiftbus is qsim_state_vector (13 downto 0);
begin
   shifter_PROCESS : process(en, din, sel)
      variable s0, s1, s2 : shiftbus;
   begin
      if (en = '1') then
         s2(13 downto 7) := (others => '0');
         s2(6 downto 0) := din;

         -- build shifting muxes
         if (sel = "11") then
            s0(6 downto 0) := (others => '0');     -- Force to 0
         else
            if (sel(1) = '0') then
               s1 := s2;
            else
               s1(11 downto 0) := s2(13 downto 2);-- shift by 2
            end if;
            if (sel(0) = '0') then
               s0 := s1;
            else
               s0(12 downto 0) := s1(13 downto 1);-- shift by 1
            end if;
         end if;

         -- assign right shift output
         dout <= s0(6 downto 0);
      end if;
   end process shifter_PROCESS;
end rtl;
```

```
library mgc_portable;
use mgc_portable.qsim_logic.all;
use work.package_utils.all;

-- accluc Entity Description
entity accluc is
   port(rst   : in qsim_state;
        clk   : in qsim_state;
        e_sum : in qsim_state;
        din1  : in qsim_state_vector(6 downto 0);
        din2  : in qsim_state_vector(15 downto 0);
        dout  : out qsim_state_vector(15 downto 0));
end accluc;

-- accluc Architecture Description
architecture rtl of accluc is
   signal reg : qsim_state_vector(15 downto 0);
begin
   accum_PROCESS : process(rst, clk)
      variable E, IS0 : integer;
   begin
      if (rst = '1') then
         reg <= (others => '0');
      elsif (clk = '1' and clk'event and clk'last_value = '0')
then
         if  (e_sum = '1') then
            E := to_Integer2(reg);
            IS0 := to_Integer2(din2);
            if (E > IS0) then
               dout <= reg - din2;
            else
               dout <= (others => '0');
            end if;
         else
            reg <= reg + din1;
         end if;
      end if;
   end process accum_PROCESS;
end rtl;



library mgc_portable;
use mgc_portable.qsim_logic.all;

-- shift2uc entity description
entity shift2uc is
   port(en   : in qsim_state;
        din  : in qsim_state_vector(15 downto 0);
        sel  : in qsim_state_vector(2 downto 0);
```

```
       dout : out qsim_state_vector(8 downto 0));
end shift2uc;

-- shift2uc architecture description
architecture rtl of shift2uc is
   subtype shiftbus is qsim_state_vector (31 downto 0);
begin
   shifter_PROCESS : process(en, din, sel)
      variable s0, s1, s2, s3 : shiftbus;
   begin
      if (en = '1') then
         s3(31 downto 16) := (others => '0');
         s3(15 downto 0) := din;

         -- build shifting muxes
         if (sel(2) = '0') then
            s2 := s3;
         else
            s2(27 downto 0) := s3(31 downto 4);   -- shift by 4
         end if;
         if (sel(1) = '0') then
            s1 := s2;
         else
            s1(29 downto 0) := s2(31 downto 2);   -- shift by 2
         end if;
         if (sel(0) = '0') then
            s0 := s1;
         else
            s0(30 downto 0) := s1(31 downto 1);   -- shift by 1
          end if;

         -- assign right shift output
         dout <= s0(8 downto 0);
      end if;
   end process shifter_PROCESS;
end rtl;



library mgc_portable;
use mgc_portable.qsim_logic.all;

-- shift3uc Entity Description
entity shift3uc is
   port(en   : in qsim_state;
        din  : in qsim_state_vector(8 downto 0);
        sel  : in qsim_state_vector(2 downto 0);
        dout : out qsim_state_vector(4 downto 0));
end shift3uc;

-- shift3uc Architecture Description
architecture rtl of shift3uc is
   subtype shiftbus is qsim_state_vector (17 downto 0);
begin
   shifter_PROCESS : process(en, din, sel)
      variable s0, s1, s2, s3 : shiftbus;
   begin
      if (en = '1') then
         s3(17 downto 9) := (others => '0');
         s3(8 downto 0) := din;

         -- build shifting muxes
         if (sel(2) = '0') then
            s2 := s3;
         else
            s2(13 downto 0) := s3(17 downto 4);   -- shift by 4
         end if;
         if (sel(1) = '0') then
            s1 := s2;
         else
            s1(15 downto 0) := s2(17 downto 2);   -- shift by 2
         end if;
         if (sel(0) = '0') then
            s0 := s1;
         else
            s0(16 downto 0) := s1(17 downto 1);   -- shift by 1
         end if;

         -- assign right shift output
         dout <= s0(6 downto 2);
      end if;
   end process shifter_PROCESS;
end rtl;



library mgc_portable;
use mgc_portable.qsim_logic.all;

-- psiluc Entity Description
entity psiluc is
   port(din  : in qsim_state_vector(4 downto 0);
        dout : out qsim_state_vector(3 downto 0));
end psiluc;

-- psiluc Architecture Description
architecture rtl of psiluc is
begin
   psi_PROCESS : process(din)
      variable a, b, c, d, e : qsim_state;
      variable na, nb, nc, nd, ne : qsim_state;
      variable psi : qsim_state_vector(3 downto 0);
```

```
   begin
      a  := din(4);
      b  := din(3);
      nb := not b;
      c  := din(2);
      nc := not c;
      d  := din(1);
      nd := not d;
      e  := din(0);
      ne := not e;

      psi(3) := c or b;
      psi(2) := (b and c) or (nb and nc and d) or
                    (b and d and e);
      psi(1) := (nb and nc and e) or (nb and c and d) or
                  (b and nc and nd) or (b and nc and ne) or a;
      psi(0) := (nb and nc and d) or (nb and nd and e) or
                  (b and nc and nd) or (b and c and d and e);

      dout <= psi;
   end process psi_PROCESS;
end rtl;



library mgc_portable;
use mgc_portable.qsim_logic.all;
use work.all;

-- c_cell Entity Description
entity c_cell is
   port(rst   : in qsim_state;
        clk   : in qsim_state;
        en0   : in qsim_state;
        en1   : in qsim_state;
        en2   : in qsim_state;
        e_sum : in qsim_state;
        us1   : in qsim_state_vector(6 downto 0);
        d1    : in qsim_state_vector(1 downto 0);
        vs1   : in qsim_state_vector(15 downto 0);
        is2   : in qsim_state_vector(2 downto 0);
        alpha : in qsim_state_vector(2 downto 0);
        uc1   : out qsim_state_vector(3 downto 0));
end c_cell;

-- c_cell Architecture Description
architecture rtl of c_cell is
   component shiftluc
      port(en   : in qsim_state;
           din  : in qsim_state_vector(6 downto 0);
           sel  : in qsim_state_vector(1 downto 0);
           dout : out qsim_state_vector(6 downto 0));
   end component;
   component accluc
      port(rst   : in qsim_state;
           clk   : in qsim_state;
           e_sum : in qsim_state;
           din1  : in qsim_state_vector(6 downto 0);
           din2  : in qsim_state_vector(15 downto 0);
           dout  : out qsim_state_vector(15 downto 0));
   end component;
   component shift2uc
      port(en   : in qsim_state;
           din  : in qsim_state_vector(15 downto 0);
           sel  : in qsim_state_vector(2 downto 0);
           dout : out qsim_state_vector(8 downto 0));
   end component;
   component shift3uc
      port(en   : in qsim_state;
           din  : in qsim_state_vector(8 downto 0);
           sel  : in qsim_state_vector(2 downto 0);
           dout : out qsim_state_vector(4 downto 0));
   end component;
   component psiluc
      port(din  : in qsim_state_vector(4 downto 0);
           dout : out qsim_state_vector(3 downto 0));
   end component;

   signal shft0 : qsim_state_vector(6 downto 0);
   signal shft1 : qsim_state_vector(8 downto 0);
   signal shft2 : qsim_state_vector(4 downto 0);
   signal accum : qsim_state_vector(15 downto 0);
begin
   -- link entities that make up C-cell component
   shift1 : shiftluc port map(en0, us1, d1, shft0);
   accum1 : accluc   port map(rst, clk, e_sum, shft0, vs1,
                              accum);
   shift2 : shift2uc port map(en1, accum, is2, shft1);
   shift3 : shift3uc port map(en2, shft1, alpha, shft2);
   psiofx : psiluc   port map(shft2, uc1);
end rtl;
```

A-4 Test Bench Models for the Output Functions of the Digi-Neocognitron

```
-------------------------------------------------------------
-- Test bench model the Vc-cell VHDL component              --
-------------------------------------------------------------

library mgc_portable;
use mgc_portable.qsim_logic.all;
use work.all;

entity vcTB is
end vcTB;

architecture rtl of vcTB is
   component vc_cell
      port(rst  : in qsim_state;
           clk  : in qsim_state;
           en0  : in qsim_state;
           en1  : in qsim_state;
           en2  : in qsim_state;
           e_sum : in qsim_state;
           cl   : in qsim_state_vector(1 downto 0);
           ucl  : in qsim_state_vector(3 downto 0);
           csum : in qsim_state_vector(3 downto 0);
           vcl  : out qsim_state_vector(3 downto 0));
   end component;

   type data4_type is array(0 to 8) of qsim_state_vector(3
downto 0);
   type data2_type is array(0 to 8) of qsim_state_vector(1
downto 0);
   constant data4 : data4_type := ("0000", "0000", "0000",
                                   "1111", "1111", "1111",
                                   "0000", "0000", "0000");


   constant data2 : data2_type := ("01", "01", "01",
                                   "01", "00", "01",
                                   "01", "01", "01");

   signal en0, en1, en2 : qsim_state;
   signal clk, rst, e_sum : qsim_state;
   signal ucl, vcl, csum : qsim_state_vector(3 downto 0);
   signal cl : qsim_state_vector(1 downto 0);
begin
   clock : process
   begin
      clk <= '0';
      wait for 5 ns;
      clk <= '1';
      wait for 5 ns;
   end process;

   test : process
   begin
      e_sum <= '0';
      rst <= '1';
      rst <= transport '0' after 10 ns;
      en0 <= '1' after 10 ns;
      en1 <= '1' after 10 ns;
      for i in 0 to 8 loop
         wait for 10 ns;
         ucl <= data4(i);
         cl  <= data2(i);
      end loop;

      wait for 10 ns;
      en0 <= '0';
      en1 <= '0';

      e_sum <= '1';      -- End of summation; divide by csum
      wait for 10 ns;
      csum <= "0010";
      en2 <= '1';
      wait for 10 ns;
      en2 <= '0';
      wait for 10 ns;
   end process test;

   -- apply stimulus to entity under test
   D1 : vc_cell port map(rst, clk, en0, en1, en2, e_sum, cl,
ucl, csum, vcl);
end rtl;
```

```
-------------------------------------------------------------
-- Test bench model the S-cell VHDL component               --
-------------------------------------------------------------

library mgc_portable;
use mgc_portable.qsim_logic.all;
use work.all;

entity usTB is
end usTB;

architecture rtl of usTB is
   component s_cell
      port(rst    : in qsim_state;
           clk    : in qsim_state;
           en0    : in qsim_state;
           en1    : in qsim_state;
           en2    : in qsim_state;
           en3    : in qsim_state;
           add    : in qsim_state;
           e_sum1 : in qsim_state;
           e_sum2 : in qsim_state;
           vcl    : in qsim_state_vector(3 downto 0);
           ucl    : in qsim_state_vector(3 downto 0);
           wgtfac : in qsim_state_vector(2 downto 0);
           al     : in qsim_state_vector(2 downto 0);
           rl     : in qsim_state_vector(2 downto 0);
           usl    : out qsim_state_vector(6 downto 0));
   end component;

   type data4_type is array(0 to 8) of qsim_state_vector(3
downto 0);
   type data3_type is array(0 to 8) of qsim_state_vector(2
downto 0);
   type data2_type is array(0 to 1) of qsim_state_vector(2
downto 0);

   constant ucl_in : data4_type := ("0000", "0000", "0000",
                                    "1111", "1111", "1111",
                                    "0000", "0000", "0000");

   constant al_in  : data3_type := ("100", "100", "100",
                                    "010", "011", "010",
                                    "100", "100", "100");

   constant wgt_in : data2_type := ("100", "000");

   signal clk, add, e_sum1, e_sum2 : qsim_state;
   signal rst, en0, en1, en2, en3  : qsim_state;
   signal vcl, ucl                 : qsim_state_vector(3 downto 0);
   signal usl                      : qsim_state_vector(6 downto 0);
   signal al, rl, wgtfac           : qsim_state_vector(2 downto 0);
begin
   clock : process
   begin
      clk <= '0';
      wait for 5 ns;
      clk <= '1';
      wait for 5 ns;
   end process clock;

   test : process
      variable cnt : integer;
   begin
      cnt := 0;
      e_sum1 <= '0';
      e_sum2 <= '0';
      rst <= '1';
      rst <= transport '0' after 10 ns;
      en0 <= '1' after 10 ns;
      en1 <= '1' after 10 ns;
      add <= '1' after 10 ns;
      add <= transport '0' after 30 ns;
      en0 <= transport '0' after 40 ns;

      for i in 0 to 8 loop
         wait for 10 ns;
         ucl <= ucl_in(i);
         al  <= al_in(i);

         -- calculates 2-term approximation
         if (cnt <  2) then
            vcl <= "1011";
            wgtfac <= wgt_in(cnt);
            cnt := cnt + 1;
         end if;
      end loop;

      wait for 10 ns;
      add <= '1';
      en1 <= '0';

      -- End of summation; divide by IC2 then multiply by rl
      e_sum1 <= '1';
      wait for 10 ns;
      e_sum1 <= '0';
      e_sum2 <= '1';
      wait for 10 ns;
      en2 <= '1';
      add <= '0';
      e_sum2 <= '0';
      wait for 10 ns;
```

```
      en2 <= '0';
      en3 <= '1';
      rl <= "001";      -- shift by 1
      wait for 10 ns;
      en3 <= '0';
      wait for 10 ns;
   end process test;

   -- apply stimulus to entity under test
   D1 : s_cell port map(rst, clk, en0, en1, en2, en3, add,
                e_sum1, e_sum2, vcl, ucl, wgtfac, al, rl, usl);
end rt1;
```

```
------------------------------------------------------------------
-- Test bench model the Vs-cell VHDL component                  --
------------------------------------------------------------------

library mgc_portable;
use mgc_portable.qsim_logic.all;
use work.all;

entity vsTB is
end vsTB;

architecture rtl of vsTB is
   component vs_cell
      port(rst   : in qsim_state;
           clk   : in qsim_state;
           en0   : in qsim_state;
           en1   : in qsim_state;
           en2   : in qsim_state;
           e_sum : in qsim_state;
           usl   : in qsim_state_vector(6 downto 0);
           dl    : in qsim_state_vector(1 downto 0);
           ksl   : in qsim_state_vector(2 downto 0);
           vsl   : out qsim_state_vector(15 downto 0);
           is2   : out qsim_state_vector(2 downto 0));
   end component;

   type data4_type is array(0 to 8) of qsim_state_vector(6
downto 0);
   type data2_type is array(0 to 8) of qsim_state_vector(1
downto 0);

   constant data5 : data4_type :=
              ("0000000", "0000000", "0000000",
               "0001000", "0000110", "0000110",
               "0000000", "0000000", "0000000");

   constant data2 : data2_type := ("01", "01", "01",
                                   "01", "00", "01",
                                   "01", "01", "01");

   signal rst, clk, en0   : qsim_state;
   signal en1, en2, e_sum : qsim_state;
   signal usl : qsim_state_vector(6 downto 0);
   signal dl  : qsim_state_vector(1 downto 0);
   signal ksl : qsim_state_vector(2 downto 0);
   signal vsl : qsim_state_vector(15 downto 0);
   signal is2 : qsim_state_vector(2 downto 0);
begin
   clock : process
   begin
      clk <= '0';
      wait for 5 ns;
      clk <= '1';
      wait for 5 ns;
   end process;

   test : process
   begin
      e_sum <= '0';
      rst <= '1';
      rst <= transport '0' after 10 ns;
      en0 <= '1' after 10 ns;
      ksl <= "100" after 10 ns;
      en1 <= '1' after 10 ns;

      for i in 0 to 8 loop
         wait for 10 ns;
         usl <= data5(i);
         dl  <= data2(i);
      end loop;

      wait for 10 ns;
      en0 <= '0';
      en1 <= '0';
      e_sum <= '1';
      wait for 10 ns;
      en2 <= '1';
      wait for 10 ns;
      en2 <= '0';
      wait for 10 ns;
   end process test;

   -- apply stimulus to entity under test
   D1 : vs_cell port map(rst, clk, en0, en1, en2,
                         e_sum, usl, dl, ksl, vsl, is2);
end rtl;
```

```
------------------------------------------------------------
-- Test bench model the C-cell VHDL component              --
------------------------------------------------------------

library mgc_portable;
use mgc_portable.qsim_logic.all;
use work.all;

entity ucTB is
end ucTB;

architecture rtl of ucTB is
   component c_cell
      port(rst   : in qsim_state;
           clk   : in qsim_state;
           en0   : in qsim_state;
           en1   : in qsim_state;
           en2   : in qsim_state;
           e_sum : in qsim_state;
           usl   : in qsim_state_vector(6 downto 0);
           dl    : in qsim_state_vector(1 downto 0);
           vsl   : in qsim_state_vector(15 downto 0);
           is2   : in qsim_state_vector(2 downto 0);
           alpha : in qsim_state_vector(2 downto 0);
           ucl   : out qsim_state_vector(3 downto 0));
   end component;

   type data4_type is array(0 to 24) of qsim_state_vector(6
downto 0);
   type data2_type is array(0 to 24) of qsim_state_vector(1
downto 0);
   constant data5 : data4_type :=
      ("0000000", "0000000", "0000000", "0000000", "0000000",
      "0000000", "0000000", "0000000", "0000000", "0000000",
      "0001000", "0000110", "0000110", "0000110", "0001000",
      "0000000", "0000000", "0000000", "0000000", "0000000",
      "0000000", "0000000", "0000000", "0000000", "0000000");

   constant data2 : data2_type :=
        ("10", "10", "10", "10", "10",
         "10", "01", "01", "01", "10",
         "10", "01", "00", "01", "10",
         "10", "01", "01", "01", "10",
         "10", "10", "10", "10", "10");

   signal rst, clk, en0, en1 : qsim_state;
   signal en2, add, e_sum : qsim_state;
   signal usl : qsim_state_vector(6 downto 0);
   signal dl  : qsim_state_vector(1 downto 0);
   signal vsl : qsim_state_vector(15 downto 0);
   signal is2, alpha : qsim_state_vector(2 downto 0);
   signal ucl : qsim_state_vector(3 downto 0);
begin
   clock_PROCESS : process
   begin
      clk <= '0';
      wait for 5 ns;
      clk <= '1';
      wait for 5 ns;
   end process clock_PROCESS;

   test_PROCESS : process
   begin
      e_sum <= '0';
      rst <= '1';
      rst <= transport '0' after 10 ns;
      en0 <= '1' after 10 ns;
      for i in 0 to 24 loop
         wait for 10 ns;
         usl <= data5(i);
         dl  <= data2(i);
      end loop;

      wait for 10 ns;
      en0 <= '0';
      vsl <= "0000000000000000";
      e_sum <= '1';
      wait for 10 ns;
      en1 <= '1';
      is2 <= "000";
      wait for 10 ns;
      en1 <= '0';
      alpha <= "000";
      en2 <= '1';
      wait for 10 ns;
      en2 <= '0';
      wait for 10 ns;
   end process test_PROCESS;

   -- apply stimulus to entity under test
   D1 : c_cell port map(rst, clk, en0, en1, en2, e_sum,
                        usl, dl, vsl, is2, alpha, ucl);
end rtl;
```

REFERENCES

[1] P Treleaven et al., "VLSI Architectures for Neural Networks", IEEE MICRO Magazine, pp. 8-27, Dec 1989.

[2] T. Khanna, "Foundations of Neural Networks", Addison-Wesley Publishing Company, 1990.

[3] K. Fukushima, "Cognitron: A Self-organizing Multilayered Neural Network", Biological Cybergenics 20, pp 121-136, 1975.

[4] K. Fukushima, "Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shifts in Position", Biological Cybergenics, 36, pp 193-202, 1980.

[5] L. D. Jackel et al., "Hardware Requirements for Neural-Net Optical Character Recognition", Proc. IEEE IJCNN, pp. 855-861, 1990.

[6] M. S. Tomlinson Jr. et al., " A Digital Neural Network Architecture for VLSI", Proc. IEEE IJCNN, pp. 2433-2436, 1990.

[7] A. Masaki et al., " Neural Networks in CMOS: A Case Study", Circuits and Devices, pp. 12-17, July 1990.

[8] B. A. White and M. I. Elmasry, "The Digi-Neocognitron: A Digital Neocognitron Neural Network Model for VLSI", IEEE Transactions on Neural Networks, Vol. 3 No. 1, pp. 73-85, January 1992.

[9] D.H. Hubel and T.N. Wiesel, "Receptive field, binocular interaction and functional architecture in cat's visual cortex", J. Physiol., (London) 160, pp.106-154, 1962.

[10] D.H. Hubel and T.N. Wiesel, "Receptive field and functional architecture in two nonstriate visual area (18 and 19) of the cat", J. Neurophysiol., 28, pp.229-289, 1965.

[11] D.H. Hubel and T.N. Wiesel, "Functional architecture of a macaque monkey visual cortex", Proc. R. Soc. London, Ser. B 198, pp.1-59, 1977.

[12] K. Fukushima and S. Miyake, "Neocognitron: A New Algorithm for Pattern Recognition Tolerant of Deformations and Shifts in Position", Pattern Recognition, Vol. 15 No. 6, pp. 455-469, 1982.

[13] K. Fukushima, S. Miyake and T. Ito, "Neocognitron: A neural network model for a mechanism of visual pattern recognition," IEEE Trans. Syst. Man, Cybern., vol. SMC-13, pp. 826-834, Sept. 1983.

[14] Y. C. Lim, et al., "VLSI circuits for decomposing binary integers into signed powers-of-two terms," Proc. IEEE ISCAS, 1990, pp. 2304-2307.

[15] K. Fukushima and N. Wake: "Handwritten Alphanumeric Character Recognition by the Neocognitron", IEEE Transactions on Neural Networks, Vol. 2 No. 3, pp. 355-365, May 1991.

[16] Jayaram Bhasker, "A VHDL Primer," Printice Hall, Inc., 1992.

[17] John P. Uyemura, "Circuit Design for CMOS VLSI," Kluwer Academic Publishers, 1992.

[18] N. H. E. Weste and K. Eshraghian, "Principles of CMOS VLSI Design: A Systems Perspective," Addison-Wesley Publishing Company