MONTANA STATE UNIVERSITY

VLSI synthesis of digital application specific neural networks
by Grant Philip Beagles

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering
Montana State University
© Copyright by Grant Philip Beagles (1992)

Abstract:
Most neural net research may be roughly classified into two general categories: software simulations or programmable neural hardware. The major drawback of any software simulation is that, no matter how well written, it is run on a sequential machine. This means that the software must simulate the parallelism of the network. Hardware neural networks are usually general purpose. Depending on the training, a significant percentage of the total hardware resources may be unused.

By defining a software model and then synthesizing a network from that model, all of the silicon area will be utilized. This should result in a significant reduction in chip size when comparing the application specific version to a general purpose neural network capable of being trained to perform the same task.

The purpose of this project is to synthesize an application specific neural network on a chip. This synthesis is accomplished by using parts of both categories described above. The synthesis involves several steps: 1) defining the network application; 2) building and training a software model of the network; and 3) using the model in conjunction with hardware synthesis tools to create a chip that performs the defined task in the expected manner.

The simulation phase of this project used Version 2.01 of NETS written by Paul T. Baffes of the Lyndon B. Johnson Space Center. NETS has a flexible network description format and the weight matrix may be stored in an ASCII file for easy usage in later design steps.

Hardware synthesis is done using the OCT CAD tools from U.C. Berkeley. The weight matrix (obtained from NETS) is translated into the OCT hardware description language (BDS) with several programs written explicitly for this project. OCT has several modules that allow a user to perform logic reduction, test the resulting logic circuit descriptions, and prepare the logic circuits for fabrication.

An application specific neural network, synthesized using the methodology described herein, showed a 75% reduction in total required silicon area when compared to a general purpose chip of the appropriate size. By using a software model to define an application specific neural network before hardware fabrication, significant savings in hardware resources will result.

VLSI SYNTHESIS OF DIGITAL APPLICATION SPECIFIC NEURAL NETWORKS

by

Grant Philip Beagles

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Electrical Engineering

MONTANA STATE UNIVERSITY
Bozeman, Montana

April 1992

© COPYRIGHT

by

Grant Philip Beagles

1992

All Rights Reserved

N378
B357

APPROVAL

of a thesis submitted by

Grant Philip Beagles

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

April 6, 1992
Date

Chairperson, Graduate Committee

Approved for the Major Department

April 9, 1992
Date

Head, Major Department

Approved for the College of Graduate Studies

April 22, 1992
Date

Henry L Parsons
Graduate Dean

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made.

Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted by the copyright holder.

Signature _____

Date April 14, 1992

TABLE OF CONTENTS

	Page
1. INTRODUCTION	1
The Project	1
Introduction to Neural Networks	4
Neural Nets vs. Sequential Computers	4
Neurons	5
The Biological Neuron	5
The Electronic Neuron	7
"Teaching" the Net	9
Interconnection	11
Technological Considerations	14
Conclusion	15
2. NEURAL NETWORK MODELING	16
Neural Network Simulators	16
NETS Neural Network Simulator	16
Defining the Network	17
Back Propagation "Teaching"	18
3. TRANSLATING NETS TO OCT	21
Reducing Network Connectivity	22
Verification of "Trimmed" Network	22
4. HARDWARE SYNTHESIS	26
Hardware Logic Description and Minimization	26
Hardware Description Verification	29
Hardware Realization	30
5. COMPARING APPLICATION SPECIFIC TO GENERAL PURPOSE NETWORK	33
Neuron Cell	33
Comparison	35
6. CONCLUSIONS AND FUTURE DIRECTIONS	36
Looking Back	36
Looking Ahead	37

TABLE OF CONTENTS - Continued

Page

REFERENCES 39

APPENDICES 43
A. NETS FILES 44
B. NETS-TO-OCT TRANSLATOR 53
C. OCT FILES 61

LIST OF TABLES

Table Page

1. Connection Reduction vs. Functionality 23

2. Standard cells required for input layer for given threshold values. 29

3. Comparison between general purpose and application specific realizations 35

LIST OF FIGURES

Figure Page

1. Biological neuron and an electronic analog 6

2. Electronic neuron model [Jackel,87] 9

3. Schematic diagram of abbreviated circuit [Graf,88] 12

4. RAM-controlled interconnection weights [Baird,89] 12

5. Block diagram of network 18

6. Example of training set element 19

7. Sample of Test Set 20

8. Output vector for test vector from Figure 3 23

9. Application specific neural network synthesis process 25

10. Majority logic NETS netlist 27

11. Majority logic BDS description 27

12. Placed and routed numeral recognizer 32

13. Block diagram of a "generic" neuron cell 34

14. Network Description of Numeral Recognizer for NETS 45

15. Training Set 46

16. Test Set For Model Verification 49

17. Network Test Output Vectors 51

18. NETS-to-OCT Translator. 54

LIST OF FIGURES - Continued

Figure Page

19. BDS Description of Input Layer 62

20. BDS Description of Output Layer 74

21. BDNET File to Connect Network Layers 78

22. MUSA Simulation 79

23. CHIPSTATS for Numeral Recognition Neural Chip 80

ACKNOWLEDGEMENTS

This work was supported by an educational grant from the National Science Foundation MOSIS program and equipment donations from the Hewlett-Packard Co., Tektronix, Inc., and Advanced Hardware Architectures, Inc. I would especially like to thank Kel Winters and Paul Cohen of Advanced Hardware Architectures, and Dr. Gary Harkin, Jaye Mathisen, Diane Mathews and Bob Wall of Montana State University for their invaluable assistance.

ABSTRACT

Most neural net research may be roughly classified into two general categories: software simulations or programmable neural hardware. The major drawback of any software simulation is that, no matter how well written, it is run on a sequential machine. This means that the software must simulate the parallelism of the network. Hardware neural networks are usually general purpose. Depending on the training, a significant percentage of the total hardware resources may be unused.

By defining a software model and then synthesizing a network from that model, all of the silicon area will be utilized. This should result in a significant reduction in chip size when comparing the application specific version to a general purpose neural network capable of being trained to perform the same task.

The purpose of this project is to synthesize an application specific neural network on a chip. This synthesis is accomplished by using parts of both categories described above. The synthesis involves several steps: 1) defining the network application; 2) building and training a software model of the network; and 3) using the model in conjunction with hardware synthesis tools to create a chip that performs the defined task in the expected manner.

The simulation phase of this project used Version 2.01 of NETS written by Paul T. Baffes of the Lyndon B. Johnson Space Center. NETS has a flexible network description format and the weight matrix may be stored in an ASCII file for easy usage in later design steps.

Hardware synthesis is done using the OCT CAD tools from U.C. Berkeley. The weight matrix (obtained from NETS) is translated into the OCT hardware description language (BDS) with several programs written explicitly for this project. OCT has several modules that allow a user to perform logic reduction, test the resulting logic circuit descriptions, and prepare the logic circuits for fabrication.

An application specific neural network, synthesized using the methodology described herein, showed a 75% reduction in total required silicon area when compared to a general purpose chip of the appropriate size. By using a software model to define an application specific neural network before hardware fabrication, significant savings in hardware resources will result.

CHAPTER 1

INTRODUCTION

The Project

Most efforts in neural network research are easily classified in to two general categories. The first is software solutions and the second being hardware solutions. Software solutions involve modeling the functionality of a neural network on a sequential computer with simulation programs. The hardware solution entails general purpose neural network chips which are taught to perform a task. This project involves the union of these two categories into a methodology by which simple application specific neural hardware may be made.

The first step in the synthesis process is to define the problem. In the case of this project, a chip capable of recognizing the digits 0 through 9 was specified. The network would output the appropriate ASCII code for each numeral.

Form often follows functionality and this was definitely the case here. A number of factors determined the network configuration. The first was to limit, as much as possible, the number of nodes in the network. The input layer needed to have enough cells to allow reasonably clear definitions of the characters. The output layer is seven neurons since ASCII codewords are seven bits long. The hidden layer was constrained

on the low end by the need for a sufficient number of cells to successfully recognize the characters and on the high end by the need to limit the size of the chip that will eventually be defined.

Defining the problem also involved defining the training set to be used in teaching the model and developing evaluation criteria for the results. The training set used for this project utilized only "ideal" characters. No distortion or corruption of the numerals was considered to simplify the testing of the application specific approach. The simulator that was used simulates an analog neural network. The output values were not the digital "0" or "1" but values ranging from 0 to 1. The decision boundary used in the determination of the bit value was 0.5 with a "1" being 0.5 or greater.

The initial network was fully connected. Many of the connections contributed little or nothing to the solution of the problem. The next step in the process was to eliminate those connections. The approach used for this was to eliminate any connection having a weight that was less than some predetermined level. After each "pruning" of the network connectivity, the simulator was used to verify continued satisfactory functionality of the network. Pruning was continued until the network began to exhibit inconsistent results. The reduction level used was the largest one that still gave adequate separation between ones and zeros.

This is the point where the crossover from the software domain to hardware synthesis occurs. The weight matrix from the model was translated into a hardware description language (HDL) that can be used by the OCT CAD tools which are described in Chapter 4. A translator was written to automate this step.

A driving force behind this method is to reduce the size of the die as far as possible. Silicon area is occupied by either standard cells or routing channels. The neural logic is essentially a sum and compare. If the number of bits used in the sum is reduced, the number of cells experiences a similar fate. In turn, the need for fewer cells will require less routing. Reduction was done by summing the weights and then looking at the two's compliment sign bit. A positive sum causes neural activity (output a 1) while a negative result sets the neural output to zero.

The OCT tools take the HDL description of the network and eventually produce the mask descriptions for hardware fabrication. The process is actually broken down into a number of discrete steps. The first step takes the HDL and converts it into the boolean expressions for the logic. The second step is to optimize the logic and map it into a library of standard cells. Step three is to verify the logic using a simulation tool included in OCT. Following successful verification, the "place and router" is used and the chip is prepared for fabrication. The simulation/verification step is performed again before actually releasing the chip to a foundry.

Introduction to Neural Networks

Neural Nets vs. Sequential Computers

Most everyone is familiar, at least abstractly, with the way in which Von Neumann machines are designed. A set of instructions is stored in memory and executed in a sequential manner.

For many years, these machines have been considered capable of doing anything to near perfection. "Many believe that considerations of speed apart, it is unnecessary to design various new machines to do various computing processes. They can all be done with one digital computer, suitably programmed for each case, the universal machine [Anderson,89]." Recently, however, the limits of this architecture are becoming more recognizable, and a radically different form of processor is again being studied. This new design is the **neural network.**

A neural network is a massive parallel array of simple processing units which simulate (at least in theory) the neurons contained in the human brain. An electronic neural network attempts to mimic the ability of the brain to consider many solutions to a problem simultaneously. It has also been shown to have the ability to successfully process corrupted or incomplete data [Murray,88].

Until recently, existing high speed digital supercomputers have been programmed to simulate the workings of a neural network. With the advent of VLSI technology and the possibility of ULSI (ultra large scale integration) in the near future, it has become possible to realize fairly complex neural networks in silicon.

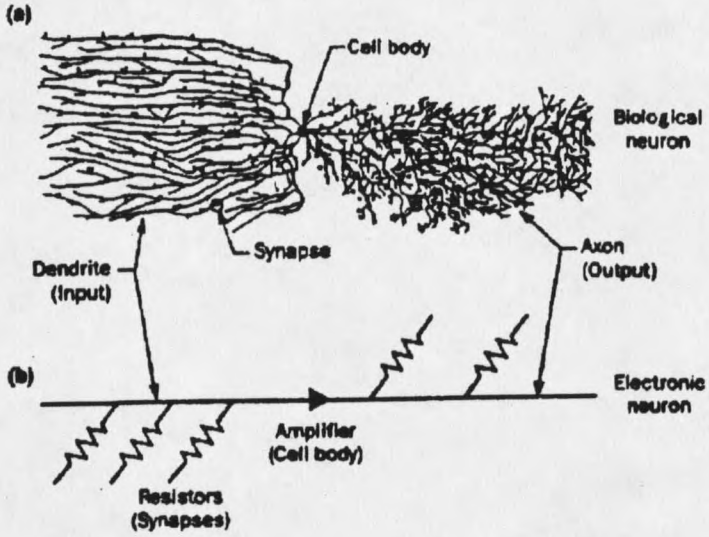
To date, electronic neural networks are still quite limited. Existing neural network hardware falls roughly into two categories: 1) one of a kind experimental projects; or 2) general purpose neural network chips that must be "taught".

Two examples that fall into the first category come from Bell Labs and Bellcore. A chip having 54 "neurons" manufactured by Bell Laboratories in Holmdel, New Jersey was "taught" to recognize key features in handwritten characters. This simple network was able to perform this task about 1,000 times faster than a suitably-programmed VAX 11/750 [Howard,88]. Bellcore has produced a neural network chip which they claim is 100,000 times faster than a neural network simulated on a general purpose computer [Cortese,88].

Neurons

The Biological Neuron A biological neuron and a simple electronic model of it taken from Richard E. Howard, et. al. are shown in Figure 1. The human brain contains about 10^{11} of these neurons, each with approximately 10^4 input and output connections. Both the input and output structures are a branching tree of fibers. The input structures called ***dendrites*** receive incoming signals from other neurons through adjustable

connections known as ***synapses***. The output is sent through the ***axons*** to the synapses of other neurons.



Biological neuron and an electronic analog.
Figure 1

Intercellular communication is accomplished through streams of electric pulses of varying rates sent out along a cell's axons to the synapses of other cells. Some synapses are excitatory, that is, input along these channels tend to make the neuron increase its activity, while others are inhibitory and cause the cell to slow its output pulse rate [Howard,88].

The real heart of the scheme is that the synaptic connections are "adjustable," meaning that the signal received at each connection is weighted and the neural response

is based on a weighted sum of all inputs. This weighing is the end result of learning and is being constantly retuned.

The Electronic Neuron The electronic neuron, like its biological cousin, has a great number of weighted interconnections. In the electronic model, however, there are at best, thousands of connections, which is several orders of magnitude less than the biological network. Each neural response is based on a weighted sum of its inputs combined with a neuron threshold value to determine the neuron's level of activity.

Ideally, the neural network should be asynchronous, unlike the bulk of present day computers. There are valid arguments that point to an analog processor as being superior to a digital model.

In their paper, Murray and Smith present a brief contrast of the strengths and weaknesses of digital and analog networks which is shown below [Murray,88].

A. *Digital Neural Networks*

The strengths of a digital approach are:

- design techniques are advanced, automated, and well understood;
- noise immunity is high;
- computational speed can be very high; and
- learning networks (i.e., those with programmable weights can be implemented readily.

For neural networks there are several drawbacks:

- digital circuits of this complexity must be synchronous, while real neural nets are asynchronous;
- all states, activities, etc. in a digital network are quantized; and
- digital multipliers, essential to the neural weighting function, occupy large silicon area.

B. *Analog Neural Networks*

The benefits of analog networks are:

- asynchronous behavior is automatic;
- smooth neural activation is automatic; and
- circuit elements can be small.

The disadvantages include:

- noise immunity low;
- arbitrarily high precision is not possible; and
- worst of all, no reliable analog nonvolatile memory technology exists.

The weights which are set through a learning process must be stored somewhere. As stated above, digital networks can easily use memory to hold the weight values, and auxiliary support logic can be designed to "reteach" the network every time it is powered up. This also allows a single general purpose neural network to be used for more than one application. Analog technology falls somewhat short of the mark.

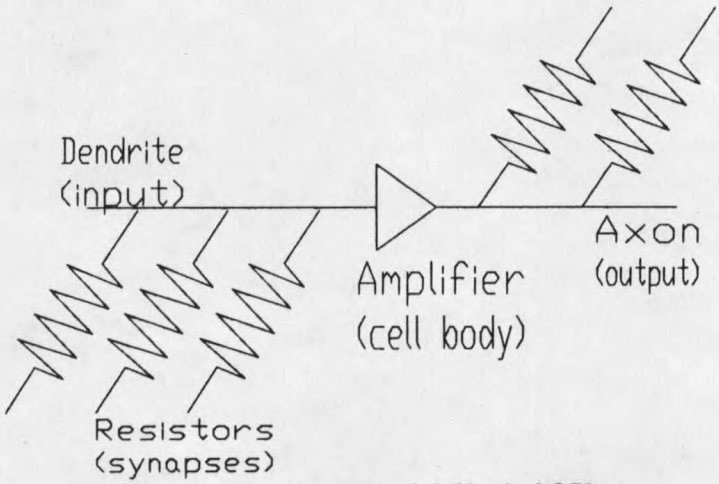
Figure 1 contains a representation of an electronic neuron which is shown in Figure 2. The neuron body is composed of an amplifier. Weighting of the synaptic connections is represented by resistors on both the input (dendrite) path and output (axon).

This model appears overly simplified, but it is really quite complete Several working neuron designs are nearly this simple.

The goal is not to identically imitate the biological neuron. Clearly, the models are grossly simplified compared to the genuine article, yet even these relatively simple networks have complex dynamics and show interesting collective computing properties. The model which is used to exploit these collective properties is referred to as a ***connectionist model*** [Graf,88]. In a connectionist model individual neurons do very little individual computations. It is the multiple parallel paths that allow the neural network to render astonishing results.

The abbreviated schematic in Figure 3 (page 12) is a portion of a circuit implemented by Bell Labs in Holmdel, New Jersey. The entire network consisted of 54

amplifiers with their inputs and outputs interconnected through a matrix of resistive coupling elements. All of the coupling elements are programmable; *i.e.*, a resistive connection can be turned on or off [Baird,89].



Electronic neuron model [Jackel,87].
Figure 2

"Teaching" the Net

All of the garden variety digital microprocessors in use today have one major trait in common; they need a built-in instruction set so that their users may program them. These instruction sets may be as succinct as a few tens of instructions or be composed of hundreds of complex instructions.

Neural networks have no built-in instruction sets. Programming a neural network is accomplished through a teaching process. Presently, there are three main methods of teaching neural networks, which are shown below.

1. Explicit programming of connectivity and weight assignments.
2. Learning algorithms that can be used to train a network starting from some (generally random) initial configuration.
3. Compiling approaches that take an abstract specification and translate it into a network description [Jones,88].

Explicit programming of connectivity and weight assignments requires that the problem be modeled and the initial weights be assigned so that the network exhibits the desired properties. Use of this method requires that some educated guessing takes place with regard to network configuration and connection weighting.

The major drawback to the learning algorithm method is the difficulty of designing a good training set. For example, the United States Postal Service has a set that is used to train optical readers that sort mail. This set, although limited to alpha-numeric characters, requires over ten thousand samples.

The teaching method most often used for this type of network training is back propagation. An input vector is applied to the input layer of a neural network and the network processes the vector. The network output is compared to the desired resultant vector and the error is propagated backwards through the network and the synaptic weights are adjusted. By using an iterative process, these learning algorithms can be used to adjust the neural weighting scheme until the error is within an acceptable range.

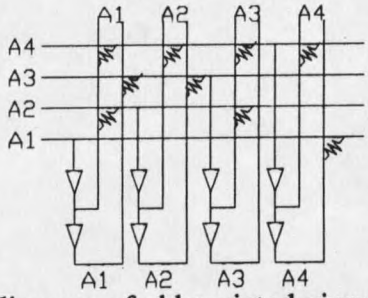
This method has the advantage of allowing greater flexibility in training the network. The network learns a task in the manner that is best suited to its design. Preconceived notions of how the network "should" function do not skew results. It should be noted that this is the technique employed in this project.

The third method involves programming the processor in somewhat the same way that a conventional machine is programmed. A neural network compiler would be necessary to convert a programmer's instructions into a language that the network could understand. The difficulties inherent in this method are the traditional notions of sequential operation.

Interconnection

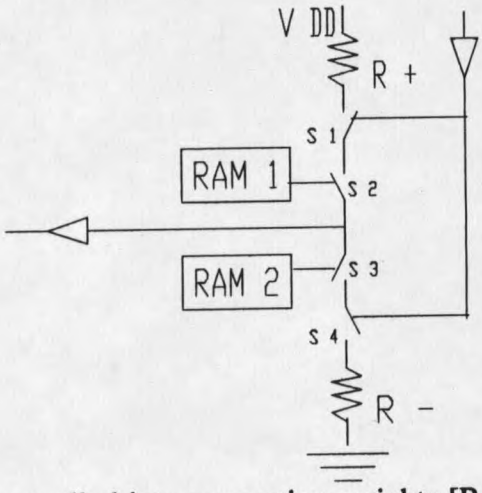
In many designs, the resistive interconnection values are precalculated and put right into the silicon when the chip is made. Bell Labs has designed a network that has 256 individual neurons on a single chip, a portion of which is shown schematically in Figure 3. The connections are provided by amorphous-silicon resistors which are placed on the chip in the last fabrication step [Graf,86]. The end product of this type of chip synthesis is application specific; it will only perform a single task. Most chips of this type are one-of-a-kind experimental devices.

This method is by far the easiest to make and use, although it does not allow for a great deal of flexibility. However, since the weights are nonvolatile, reprogramming is not necessary every time the chip is powered off and on.



Schematic diagram of abbreviated circuit [Graf,88].
Figure 3

The resistors in Figure 2 were actually realized using the circuit in Figure 4. The output lines do not actually feed current into the input lines but instead control switches S1 and S4. The two memory cells determine the type of connection. One of three connections can be selected: 1) an excitatory (S2 enabled): 2) an inhibitory (S3 enabled); or 3) an open (both disabled) state [Graf,88].



RAM-controlled interconnection weights [Baird,89].
Figure 4

By employing the RAM cells into each neuron, it becomes a simple task to "teach" the network to perform a task, download the weighting information to nonvolatile storage, and reload it the next time the network is powered up. This method also allows the network to have numerous predetermined weighting patterns loaded for specific assignments. Although this general purpose approach may seem advantageous, the application specific approach may be more desirable.

A fascinating development in neuron theory comes from a model of the human immune response which in turn is modeled on Darwin's Theory of Evolution. The theory is that a neural network with hysteresis at the single cell level will more closely model a brain. The neuron is slightly more complicated than the model previously discussed. "Learning [in this type of network] occurs as a natural consequence of interactions between the network and its environment, with environmental stimuli moving the system around in an N-dimensional phase space until [an appropriate equilibrium is reached in response to the stimuli] [Hoffmann,86]."

In the connectionist model, the weighting is necessarily fixed, or at best, modifiable within strict limitations (usually -1, 0, +1). The hysteresis model does not have such restrictions. It should be able not only to learn, but to readily adapt to changing operational conditions. Experiments carried out at Los Alamos National Laboratories with this class of neuron have had promising results.

Technological Considerations

Several categories of electronic neurons are presently being employed in functional neural networks. The largest networks-on-a-chip existing today weigh in at around five hundred neurons. As feature sizes continue to shrink below the 1 μm range, 1000+ neuron networks on a single chip will become feasible [Sivilotti,86]. Such chips will be valuable tools in the direct modeling and evaluation of neural networks.

VLSI and ULSI chip design will allow for network processors containing a modest number of neurons. Silicon has an obvious drawback in realizing a network. Two dimensionality makes it impossible to build really large nets. Three dimensional biological material are more suited to this type of architecture, although the days of "growing" reliable processors is a long way off.

Neural network processing technology is in its infancy and no specific application for these processors has yet been clearly defined. It has even been shown that the optimal solution for many problems is in existing conventional equipment.

Neural networks have the ability to work with corrupted or incomplete data sets and to look at many solutions to a problem simultaneously. Due to this unique capability, network processors seem to be aptly suited to pattern recognition applications.

Conclusion

A biological neuron is a complex entity whose operation is only partially understood. Simple electronic models of a neuron can be built to mimic the basal operation of their biological counterpart. These simple processors are linked into connectionist networks which are capable of accomplishing tasks that are nearly impossible for most computers to do.

One thousand neuron networks are on the horizon. Although a quantum step backward from the 100 billion or so neurons of the human "network," the speed capability of a neural network chip far surpasses that of our brains. This differential will partially make up for the diminutive size and allow neural network processors to execute assignments with great dispatch. The day of the true "thinking machine" is far in the future, but the ***idiot savant*** is already here.

CHAPTER 2

NEURAL NETWORK MODELING

Neural Network Simulators

Many neural network simulators exist. It is even possible to simulate a network using any good spreadsheet program. Most neural net simulators that are available perform quite well. However, since only a simulation is involved, there are some drawbacks. Simulators are generally run on a sequential operation computer. A neural network by its very definition is a "highly parallel array of simple processors." The parallism must be modeled, which requires an iterative process. If the network being modeled has more than a few tens of nodes, or more than a few thousand connections, the number of iterations becomes astronomical.

NETS Neural Network Simulator

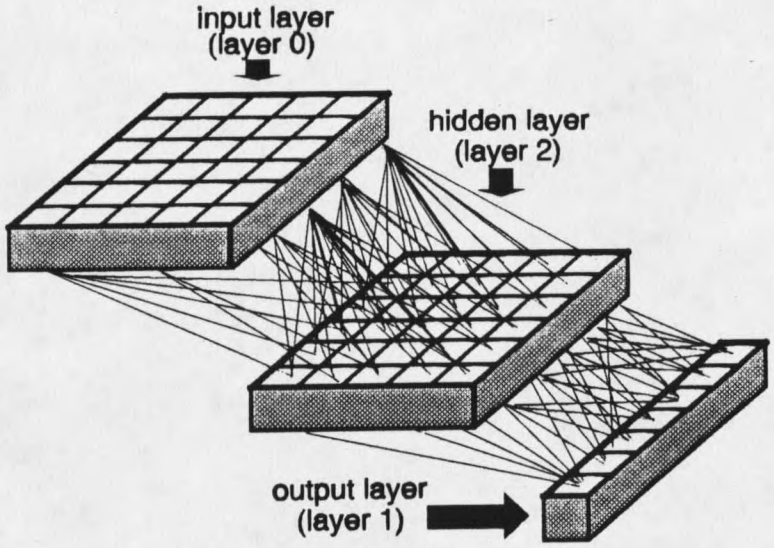
The simulator that is being used for this project is Version 2.01 of NETS written by Paul T. Baffes of the Software Technology Branch of the Lyndon B. Johnson Space Center. This simulator was chosen for several reasons. NETS has a flexible network description format which allowed easy modifications to a network. The source code was

available and well documented; this allows modifications to be made directly on the simulator. Finally, the weight matrix generated during the teaching phase may be stored in an ASCII file for easy usage in later design steps.

Defining the Network

The function that a neural network will be used for will place constraints on its configuration. As a first design effort, a simple network with three layers and 67 neurons was defined. The network consists of a 5 by 6 node input layer, one hidden layer that is also 5 by 6 nodes , and a 1 by 7 node output layer. This geometry was driven by three major factors: 1) the simulations were done on a PC which slows to a near standstill if the network is too large; 2) the network was trained to recognize the ten numerals and output the appropriate ASCII code (hence, the 1 by 7 node output layer); and 3) the network, mapped into a standard cell library, needed to be small enough to be fabricated on a reasonable size die.

The network geometry is fully connected, that is, every node in a layer is connected to every node in the layer below it. For the network in this example there are 1110 connections. The NETS network description for this network is contained in Appendix A. Figure 5 shows a stylized block diagram of the network.



Block diagram of network
Figure 5

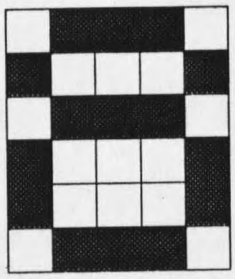
Back Propagation "Teaching"

Once the network geometry has been defined, it must be "taught"to perform its task. The training method employed was the classic back propagation algorithm. An input vector was input at layer 0 (the uppermost layer). The network was allowed to process this vector and produce a resultant vector at the output layer. The resultant is compared to a desired vector and the error calculated. This error is propagated upward through the network. The weights of the connections within the network are adjusted. This cycle is repeated until the calculated error falls within an acceptable boundary.

A training set consisting of ten digits (0 through 9) and the corresponding ASCII values was used to build the network weighting matrix. Figure 6 illustrates a typical

character representation and its corresponding input/output vector. The entire training set is shown in Appendix A. Although neural networks have been shown to do well at processing noisy or corrupted data, this training set does not include any noisy or corrupted data. This was done to simplify the model and to reduce the number of unknowns inherent in the project.

Training the network required one hundred iterations and was completed in about six minutes. As was stated earlier, the fully connected network has 1110 connections. The magnitude of the weights in the weight matrix range between ±1.7 following training.



011 1000 b

Character
w/ASCII code

(.1 .9 .9 .9 .1
.9 .1 .1 .1 .9
.1 .9 .9 .9 .1
.9 .1 .1 .1 .9
.9 .1 .1 .1 .9
.1 .9 .9 .9 .1
.1 .9 .9 .9 .1 .1 .1)

Input/output vector

Example of training set element.
Figure 6.

After the training of a network is complete, the resulting network's functionality must be verified. This was done within the NETS program. A test set was given to the network and the resulting output examined. A sample of the test set with its resultant is shown in Figure 7. Since NETS simulates an analog network, a boundary between a digital "1" and "0" must be determined. Because the output values ranged from near 0 to 1, the obvious decision boundary was 0.5. Appendix A contains the test set and the corresponding resultant vectors. A quick examination of the resultants show that the network will indeed recognize all 10 numerals.

```
-- test set for min.net

(.1 .9 .9 .9 .1          -- "8"
 .9 .1 .1 .1 .9
 .1 .9 .9 .9 .1
 .9 .1 .1 .1 .9
 .9 .1 .1 .1 .9
 .1 .9 .9 .9 .1)

Outputs for Input 9:
(0.002  0.846  0.994  0.865  0.036
0.257  0.164)
```

Sample of Test Set
Figure 7

CHAPTER 3

TRANSLATING NETS TO OCT

The software simulation portion of the application specific neural network synthesis used the NETS simulator and is described in Chapter 2. The hardware synthesis is accomplished using the OCT CAD tools, written at The University of California at Berkeley.

The OCT tools are a suite of software tools which aid in the synthesis of VLSI hardware. The package includes programs that; 1) reduce a hardware description to its representative boolean expressions; 2) optimize the equations and map them into a standard cell library; 3) do place and route for optimal silicon usage and; 4) perform pad place and route. A simulator is also included which allows the circuits to be verified and a design rule checker verifies the layout. A more detailed description of the hardware synthesis is found in Chapter 4.

After a suitable NETS model is constructed, the synaptic weight matrix is translated from the NETS weight file to an OCT hardware description (BDS). A number of short utility programs were written to accomplish this task as well as modify the weight matrix.

Reducing Network Connectivity

The next step in the synthesis of an application specific neural network is to reduce the number of connections in the network. This is done by setting all weights having an absolute value less than a specified value to zero (no connect). This process is easily automated allowing various cut-off values to be evaluated. Appendix B has a listing of a program written in "C" that will trim the connections.

Verification of "Trimmed" Network

NETS allows a weight matrix to be reloaded. The trimming program created a file that is compatible with NETS so that the modified weight matrix can be used with the basic network description. The test set used to evaluate the fully connected network is again used to test the reduced connection network. Table 1 summarizes the results of reducing the "numeral to ASCII" neural network. The cut-off value is relative to a maximum weight of 1.

The actual cut-off values tested ranged up to 1, however, all results with a cut-off above 0.55 were inconsistent with the desired results. Figure 8 is the test vector for the character shown in Figure 6 with its associated output vector. (The cut-off is 0.55 and the threshold is 0.5.) With the threshold value taken into consideration, the output is 011 1000 which is the ASCII code for "8".

CUT-OFF VALUE	NUMBER OF CONNECTIONS	SATISFACTORY PERFORMANCE	THRESHOLD[1]
0.0	1110	yes	0.5
0.3	688	yes	0.5
0.4	530	yes	0.5
0.5	413	yes	0.5
0.55	344	yes	0.5
0.6	288	no	----

1 Any value ≥ threshold is a "one" otherwise "zero".

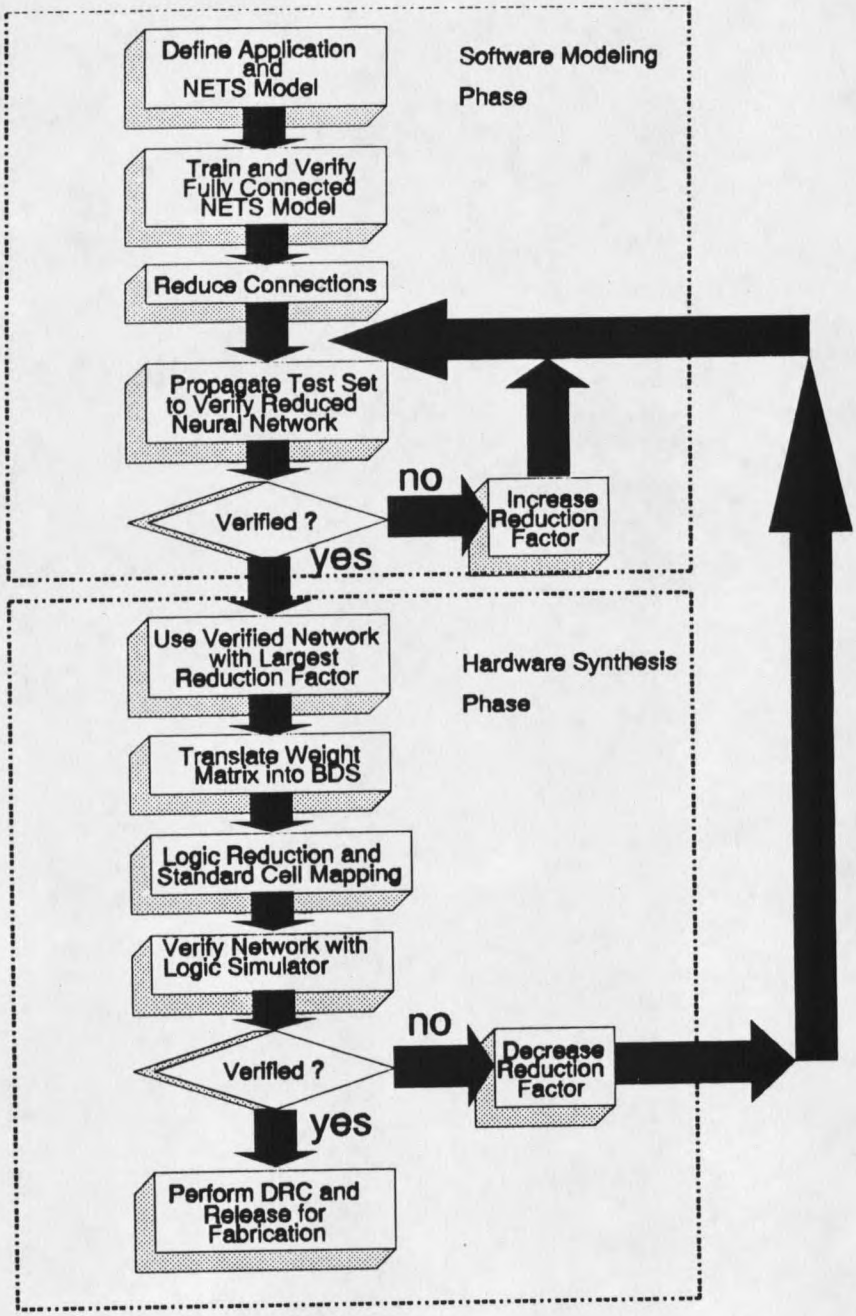
Connection Reduction vs. Functionality
Table 1.

```
-- test set for min.net

(.1 .9 .9 .9 .1          -- "8"
 .9 .1 .1 .1 .9
 .1 .9 .9 .9 .1
 .9 .1 .1 .1 .9
 .9 .1 .1 .1 .9
 .1 .9 .9 .9 .1)

Outputs for Input 9:
(0.002  0.846  0.994  0.865  0.036
0.257  0.164)
```

Output vector for test vector from Figure 3
Figure 8

Figure 9, found at the end of the chapter, is a flow chart of the synthesis process used in this project. The figure shows that the next step following the reduction of connectivity is to translate the network parameters from the NETS domain into the OCT tools domain. The OCT tools set from the University of California, Berkeley is a set of CAD tools which includes logic optimization, technology mapping, standard-cell place-and-route, and composite artwork assembly and verification [Beagles,91].

This step is accomplished using another program which can be found in Appendix B. This program takes the NETS weight matrix and user interactive input to produce a BDS file (BDS is the OCT tools hardware description language).



Application specific neural network synthesis process.
Figure 9.

CHAPTER 4

HARDWARE SYNTHESIS

The intent of the neural network synthesis process is to provide a fully automatic path to silicon realization once a network model has been constructed and verified in the NETS environment. The entire synthesis process is schematically shown in Figure 9 in the last chapter.

Hardware Logic Description and Minimization

At this point, the NETS weight matrix has been converted into a BDS file. A simple example of a single neuron in NETS netlist is shown in Figure 10 and the corresponding BDS description in figure 11. The NETS description of the numeral recognition network is in Appendix A, and the corresponding BDS file is contained in Appendix C.

The primary goal of the application specific method being used in this project is to reduce the silicon area needed to realize a neural network trained for a specific task. Closer examination of the BDS file in Appendix C will show another method used beyond the application specific philosophy to reduce the die size. Since the neural logic is essentially a sum and compare function, logic is needed to sum the weights and then

compare this sum with a threshold to determine neural activity. This same operation was done using two's compliment arithmetic and examining only the sign bit of the sum. A positive sum causes the neuron to output a "1" while a negative result induces a "0"

```
LAYER : 0          --INPUT LAYER
      NODES : 5
      TARGET : 1
LAYER : 1          --OUTPUT LAYER
      NODES : 1
```

Majority logic NETS netlist
Figure 10

```
MODEL dumb
      out<0>,sum0<19:0>=in<4:0>;

      CONSTANT THRESHOLD = 0;
ROUTINE dumbnet;
!     target layer # 0  node # 0

sum0<19:0> =        (in0<0> AND 12903) +
            (in0<1> AND 9234) +
            (in0<2> AND 12389) +
            (in0<3> AND 11115) +
            (in0<4> AND 11151) -
            21832;
IF sum0<19:0> GEQ THRESHOLD
      THEN out<0> = 1
```

Majority logic BDS description
Figure 11

The BDS file is compiled into unminimized logic functions by the OCT tool, BDSYN. The output file generated by BDSYN is simply the Boolean equations for the logic described in the BDS file. These logic functions are mapped into a standard cell library by misII. Currently, the SCMOS2.2 standard-cell library from Mississippi State University is used, implemented in the SCMOS6 N-Well CMOS process available from the National Science Foundation MOSIS program. This process has a minimum feature size of two microns.

MisII is an n-level logic optimizer which creates a realization of a logic function from a given cell library minimizing both worst-case propagation delay and the number of cells required. The relative priority of area versus speed is user selectable.

Another feature of misII is the ability to collapse a node into its fanouts using the "eliminate" option. A node is eliminated if its value is less than a specified threshold [Casotto,91]. A node's value is approximately based on the number of times that the node is used in the factored form from each of its fanouts. Table 2 shows the results of using this option on the input layer of the neural recognition neural network.

"ELIMINATE" THRESHOLD	STANDARD CELLS REQUIRED
not used	1574
10	1145
25	1125
35	1122
40	1077
45	1141
60	1147

Standard cells required for input layer for given threshold values.
Table 2.

Hardware Description Verification

MUSA is the OCT simulation module. It takes the cell view generated by misII and simulates the functionality of the circuits. Appendix C holds the simulation parameters and the MUSA results. The data in Appendix C verifies that the desired ASCII code is generated for all ten numerals. Although this neural network was trained with a small teaching set which contained only clean undistorted characters, it is capable of correctly recognizing slight deviations from the "ideal" input.

As was mentioned in Chapter 2, the use of distorted or noisy data was avoided to simplify the problem. The ability shown by the network to correctly identify distortions

of the model digits clearly shows that the model is behaving in the manner expected of neural networks.

The testing done at this point in the synthesis process evaluates the standard cell logic circuits. If the network did not perform as expected, another iteration of the synthesis process would be done.

Hardware Realization

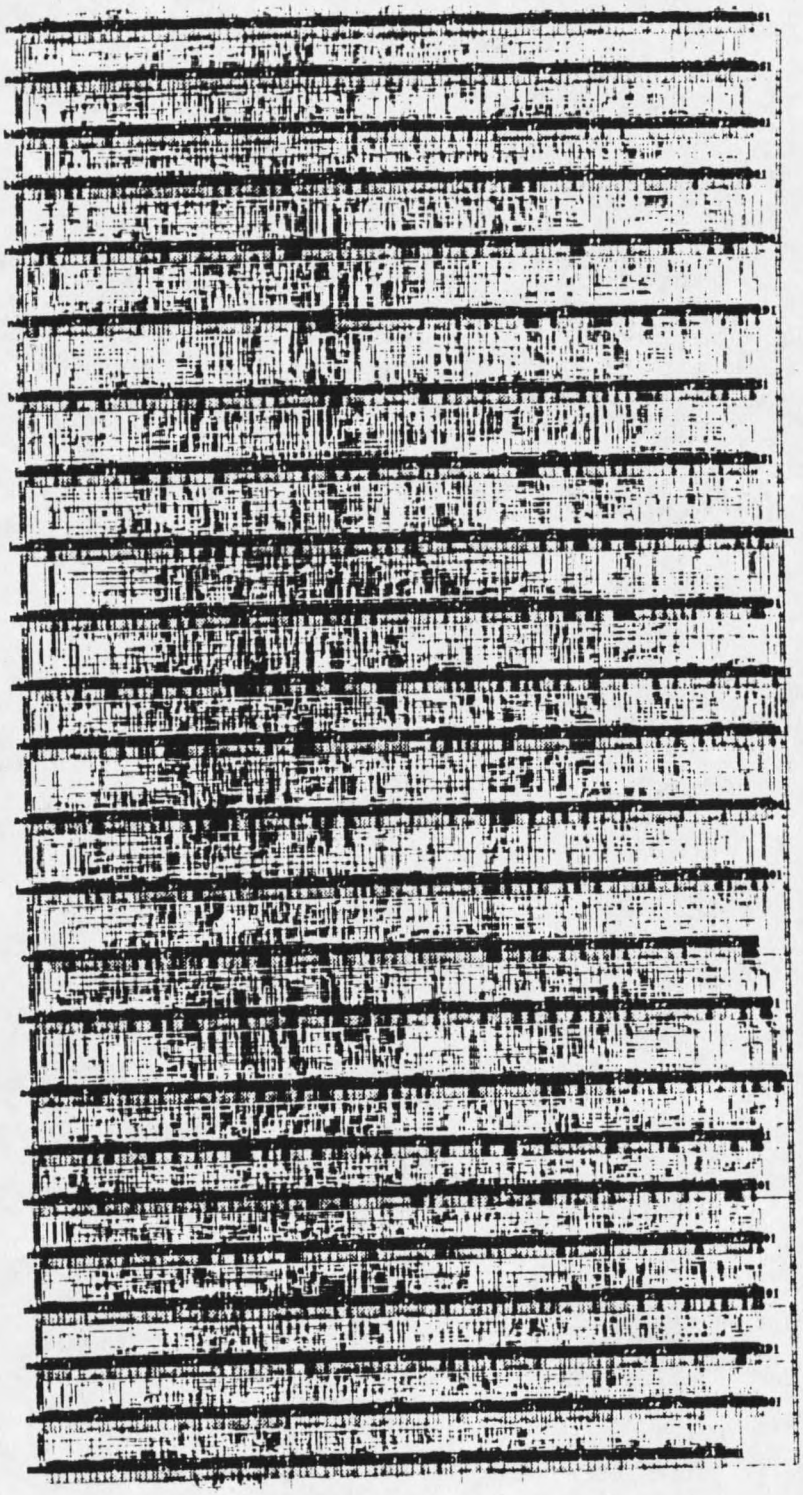
In order to maximize the reduction in the number of standard cells used in the numeral recognizer, each layer of the network was actually run through the misII step individually. After each layer's description was mapped, the layers were combined into the complete network. This combination is accomplished using the OCT tool called BDNET. Appendix C contains the BDNET parameter file used to connect the network layers.

After the verification, the logic circuits must be prepared for fabrication. The first step in the preparation process is to reduce the hierarchy to a single level. The OCT module used in this step is octflatten. Once past the flattening stage, the place and route tools are used to optimize the placement of the standard cells and minimize the routing for maximum usage of silicon resources.

A number of additional OCT tools are available for padring composition, composite placement and channel routing, power distribution routing, and artwork

verification. Artwork may be generated from the OCT database in Caltech Intermediate Format (CIF) for release to MOSIS or other foundry services.

The standard-cell realization of the digit recognizer described previously is shown in Figure 12. This figure only shows the standard cell placement and routing. The padring is not included. Its 67 neurons require 1851 standard cells. The cells and associated routing require almost 15 square millimeters of silicon.



Placed and routed numeral recognizer.
Figure 12

CHAPTER 5

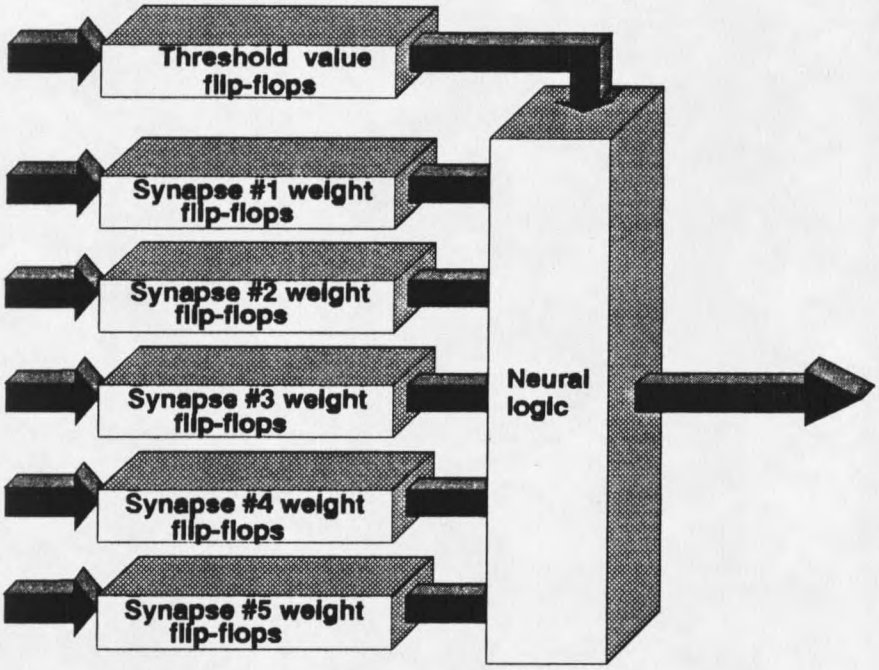
COMPARING APPLICATION SPECIFIC TO GENERAL PURPOSE NETWORK

The basic premise of this project is that an application specific neural network will utilize less silicon resources than a general purpose neural network that is capable of performing the same task. To provide some basis for this premise, a general purpose neuron which was developed by a senior design group at Montana State University was used as a "cell" for building the generic network [Wandler,91].

Neuron Cell

A block diagram of the generic neuron cell is shown in Figure 13. The weights must be stored in the synapse weight flip-flops. This does allow the network to be programmed for various tasks, but requires additional routing and internal complexity when compared to the application specific model. The threshold flip-flops hold a value which is used by the neural logic. The neural logic is a simple sum and compare circuit. The sum of the enabled weights is compared to the threshold. The enabling is accomplished by "ANDing" the synaptic input with the weight value, thus an input one causes the weight value to contribute to the sum, while a zero causes the weight to be

ignored. If this sum meets or exceeds the threshold the neuron outputs a one. If the sum is less than the threshold the neuron output is a zero.



Block diagram of a "generic" neuron cell.
Figure 13.

Comparison

The realization of the numeral recognition chip requires 1851 standard cells. Each one of the generic neuron cells require 184 standard cells. To build a network of generic neuron cells that will perform the numeral recognition assignment satisfactorily requires approximately 45 neurons. This general purpose realization would use a grand total of nearly 8300 standard cells and would occupy nearly 141 square millimeters of silicon. In contrast, the 1851 cells of the application specific network only require 15 square millimeters of silicon. These results are summarized in Table 3.

	Standard Cells Required	Die Size (mm^2)
General Purpose	8300	141
Application Specific	1851	15
Savings	78%	87%

Comparison between general purpose and application specific realizations.
Table 3.

The savings inherent in the application specific methodology are quite obvious. The number of cells saved by utilizing this technique is nearly 6500 cells or a 70% savings. The 87% savings in silicon requirements is even more significant since the cost of fabricating is related to the square of the die size.

CHAPTER 6

CONCLUSIONS AND FUTURE DIRECTIONS

This project has not proposed any new technology or application concerning neural networks. The goal of this research is a new methodology for fabricating application specific neural networks.

Looking Back

A number of points brought forth in the course of this thesis deserve repeating. First, the basic premise upon which the entire project is based; a neural network that is application specific will be less costly to produce when compared to a suitably programmed general purpose neural network chip. The previous chapter shows the comparison made between an application specific numeral recognition neural net and a similar network made up of general purpose neuron cell. The application specific approach produces a significantly less costly result.

The chip designed in the course of this research is not commercially viable. The training set used to teach it the numeral set contained only one idealized example of each digit. However, even with this basic training, the network is capable of recognizing slightly distorted versions of the numerals.

The NETS-to-OCT translation programs written in the course of this project are fairly narrow in scope. They are designed to produce a minimal connected network that will still perform the desired task.

The process used in the hardware description is the SCMOS6 N-Well CMOS process available from the National Science Foundation MOSIS Program. This is a 2 micron process.

Looking Ahead

To make this neural network a commercially viable product several improvements in the design would be necessary. The training set used in the modeling phase of the design would need to be more robust. A comprehensive training set would require the inclusion of many varied examples of each character. The training examples should also include some noise, that is, not be easily discernable from background clutter.

The NETS-to-OCT translation tools written for this project might be expanded to allow more options than are presently available. The focused nature of the current software could be softened so that currently unavailable network strategies would be possible.

Chip size could be reduced significantly by using a smaller feature size. Further, this realization was created using the Mississippi State University standard cell library that is provided with the OCT tools. The use of more compact standard cells might also enable further reduction of the required die size.

The ultimate goal of this research was to produce an application specific neural network that was trained to recognize the alphabet (both upper and lower case), the ten numeral characters, and more commonly used punctuation marks. Early on it was projected that this network would require well over 1000 nodes to be of any value. This drastic increase is due to two factors.

The first culprit in the node explosion is found in the input layer. The layer in the network used in the project is 5 by 6 neurons. For a network to be selective enough when "looking" at a character, the input layer should have many more nodes allowing finer detail to enter into the network's decision. An input layer of 20 by 30 neurons would likely deliver adequate resolution.

The second factor adding to the number of nodes in the network is the increased size of the character set that the network would recognize. The hidden layer needs to have, at minimum, twice the number of nodes as characters to be recognized [Firebaugh,88]. The minimum hidden layer would then be about 150 neurons.

This ambitious design was necessarily downsized to simplify the project (since the project was to provide a method not a commercial product). Although more research is necessary, early results show the method to be promising.

REFERENCES

Allman, W.F., "Designing Computers That Think the Way We Do," *Technology Review*, Vol. 90, pp. 58-65, May/June 1987.

Almasi, G.S. and Gottlieb, A., *Highly Parallel Computing*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1989.

Anderson, H.C, "Neural Network Machines," *IEEE Potentials*, Vol. 8 no. 1, pp. 13-16, Feb 1989.

J.A. Anderson,J.A., Hammerstrom, D., and Jackel, L.D., "*Neural Network Applications for the 90's*," IEEE Videoconference, May 23, 1991.

Baird, H.S., Graf, H.P., Jackel, L.D., and Hubbard, W.E., "A VLSI Architecture for Binary Image Classification," Information provided by AT&T Bell Labs, Holmdel, New Jersey.

Baffes, P.T., "*NETS User's Guide*," Software Technology Branch, Lyndon B. Johnson Space Center, Houston, TX, 1990.

Beagles G.P. and Winters, K.D., "VLSI Synthesis of Digital Application Specific Neural Networks", *Proceedings - Third Annual NASA SERC VLSI Design Symposium*, Moscow, Idaho, 1991.

Burkley, J.T., "*MPP VLSI Multiprocessor Integrated Circuit Design*," The Massively Parallel Processor (Research Reports & Notes, MIT Press, pp. 205-215, 1985.

Casotto, A., ed., *OCTTOOLS Revision 5.1 User Guide*, University of California Berkeley, Electronics Research Laboratory, Berkeley, Ca, 1991

Cortese, A., "Bellcore puts Neural Nets on a Chip," *Computerworld*, Vol. 22 no. 38, pp. 23, Sept 19,1988.

Denker, J.S., "Neural Network Refinements and Extensions," *Proceedings - Neural Networks for Computing*, Snowbird, Utah, 1986.

Firebaugh, M.W., "*Artificial Intelligence*," Ch. 18, PWS-KENT, Boston MA, 1988.

Fox, G.C.and Messina, P.C., "Advanced Computer Architectures," *Scientific American*, Vol. 257, pp. 66-74, October 1987.

Graf, H.P., Jackel, L.D., Hubbard, W.E., "VLSI Implementation of a Neural Network Model," *IEEE Computer*, pp. 41-49, March 1988.

Graf, H.P., Jackel, L.D., Howard, R.E., Straughn, H.B., Denker, J.S., Hubbard, W., Tennant, D.M., and Schwartz, D., "VLSI Implementation of a Neural Network Memory with Several Hundreds of Neurons," *Proceedings - Neural Networks for Computing*, Snowbird, Utah, 1986.

Gorman, C. "Putting Brainpower in a Box; Neural Networks Could Change the way Computers Work.," *Time*, Vol. 132, pp.59, Aug 8, 1988.

Harvey, R.E., "Darpa Poised to Sink Funds into Neural Net Computer R&D for Military End Users. (Defense Advanced Research Projects Agency)," *Metalworking News*, Vol. 15, pp.19, Oct 10,1988.

Harvey, R.E., "Neurocomputers find new Adherents, "*Metalworking News*, Vol. 15, pp.23, May 23,1988.

Hoffmann, G.W. and Benson, M.W., "Neurons with Hysteresis Form a Network that can Learn Without any Changes in Synaptic Connection Strengths," *Proceedings - Neural Networks for Computing*, Snowbird, Utah, 1986.

Hopfield, J.J. and Tank, D.W., "Computing with Neural Circuits: A Model," *Science*, Vol. 233, pp. 625-633, August 8, 1986.

Howard, R.E., Jackel, L.D., and Graf,H.P., "Electronic Neural Networks," *AT&T Technical Journal*, Vol. 67, pp. 58-64, Jan/Feb 1988.

Hubbard, W., Graf, H.P., Jackel, L.D., Howard, R.E., Straughn, H.B., Denker, J.S., Tennant, D.M., and Schwartz, D., "Electronic Neural Networks," *Proceedings - Neural Networks for Computing*, Snowbird, Utah, 1986.

Jackel, L.D., Graf, H.P., and Howard, R.E., "Electronic Neural Network Chips," *Applied Optics*, Vol. 26, pp. 5077-80, December 1,1987.

Jones, M.A., "Programming Connectionist Architectures," *AT&T Technical Journal*, Vol. 67, pp. 65-68, Jan/Feb 1988.

Mackie, S., Graf, H.P., Schwartz, D.B., and Denker, J.S., "*Microelectronic Implementations of Connectionist Neural Networks*," Denver, Colorado, November 8-12, 1987.

Mueller P. and Lazzaro, J., "A Machine for Neural Computation of Acoustical Patterns with Application to Real Time Speech Recognition," *Proceedings - Neural Networks for Computing*, Snowbird, Utah, 1986

Murray A.F. and Smith, A.V.W., "Asynchronous VLSI Neural Networks Using Pulse-Stream Arithmetic," *IEEE Journal of Solid State Circuits*, Vol. 23 no. 3, pp. 688-697, June 1988.

Rumelhart D.E. and McClelland, J.L., "*Parallel Distributed Processing*," Vol 1 and 2, MIT Press, Cambridge, MA, 1986.

Sivilotti, M.A., Emerling, M.R., and Mead, C.A., "VLSI Architectures for Implementation of Neural Networks," *Proceedings - Neural Networks for Computing*, Snowbird, Utah, 1986.

Wandler, S. and Metcalf, D., "*Development of a Neural Network Integrated Circuit*," Senior Project Report, Montana State University, Department of Electrical Engineering, 1991.

APPENDICES

APPENDIX A

NETS FILES

Figure 14.
Network Description of Numeral Recognizer for NETS

```
LAYER : 0          --INPUT LAYER
    NODES : 30
        X-DIMENSION : 5
        Y-DIMENSION : 6
    TARGET : 2

LAYER : 1          --OUTPUT LAYER
    NODES : 7
        X-DIMENSION : 1
        Y-DIMENSION : 7

LAYER : 2          --FIRST HIDDEN LAYER
    NODES : 30
        X-DIMENSION : 5
        Y-DIMENSION : 6
    TARGET : 1
```

Figure 15.
Training Set

```
-- training set for min.net

(.1 .9 .9 .9 .1          -- "0"
 .9 .1 .1 .1 .9
 .9 .1 .1 .1 .9
 .9 .1 .1 .1 .9
 .9 .1 .1 .1 .9
 .1 .9 .9 .9 .1
 .1 .9 .9 .1 .1 .1 .1)

(.1 .1 .9 .1 .1          -- "1"
 .1 .9 .9 .1 .1
 .1 .1 .9 .1 .1
 .1 .1 .9 .1 .1
 .1 .1 .9 .1 .1
 .1 .9 .9 .9 .1
 .1 .9 .9 .1 .1 .1 .9)

(.1 .9 .9 .9 .1          -- "2"
 .9 .1 .1 .1 .9
 .1 .1 .1 .9 .1
 .1 .1 .9 .1 .1
 .1 .9 .1 .1 .1
 .9 .9 .9 .9 .9
 .1 .9 .9 .1 .1 .9 .1)

(.1 .9 .9 .9 .1          -- "3"
 .9 .1 .1 .1 .9
 .1 .1 .9 .9 .1
 .1 .1 .1 .1 .9
 .9 .1 .1 .1 .9
 .1 .9 .9 .9 .1
 .1 .9 .9 .1 .1 .9 .9)
```

```
(.9 .1 .1 .9 .1          -- "4"
 .9 .1 .1 .9 .1
 .9 .1 .1 .9 .1
 .9 .9 .9 .9 .9
 .1 .1 .1 .9 .1
 .1 .1 .1 .9 .1
 .1 .9 .9 .1 .9 .1 .1)

(.9 .9 .9 .9 .9          -- "5"
 .9 .1 .1 .1 .1
 .9 .9 .9 .9 .1
 .1 .1 .1 .1 .9
 .1 .1 .1 .1 .9
 .9 .9 .9 .9 .1
 .1 .9 .9 .1 .9 .1 .9)

(.1 .1 .9 .1 .1          -- "6"
 .1 .9 .1 .1 .1
 .9 .1 .9 .9 .1
 .9 .9 .1 .1 .9
 .9 .1 .1 .1 .9
 .1 .9 .9 .9 .1
 .1 .9 .9 .1 .9 .9 .1)

(.9 .9 .9 .9 .9          -- "7"
 .1 .1 .1 .1 .9
 .1 .1 .1 .9 .1
 .1 .1 .9 .1 .1
 .1 .9 .1. 1. 1
 .9 .1 .1 .1 .1
 .1 .9 .9 .1 .9 .9 .9)

(.1 .9 .9 .9 .1          -- "8"
 .9 .1 .1 .1 .9
 .1 .9 .9 .9 .1
 .9 .1 .1 .1 .9
 .9 .1 .1 .1 .9
 .1 .9 .9 .9 .1
 .1 .9 .9 .9 .1 .1 .1)
```

```
(.1 .9 .9 .9 .1
 .9 .1 .1 .1 .9
 .9 .1 .1 .9 .9
 .1 .9 .9 .1 .9
 .1 .1 .1 .1 .9
 .1 .1 .1 .1 .9
 .1 .9 .9 .9 .1 .1 .9)
```

Figure 16.
Test Set For Model Verification

```
-- test set for min.net

(.1 .9 .9 .9 .1          -- "0"
 .9 .1 .1 .1 .9
 .9 .1 .1 .1 .9
 .9 .1 .1 .1 .9
 .9 .1 .1 .1 .9
 .1 .9 .9 .9 .1)

(.1 .1 .9 .1 .1          -- "1"
 .1 .9 .9 .1 .1
 .1 .1 .9 .1 .1
 .1 .1 .9 .1 .1
 .1 .1 .9 .1 .1
 .1 .9 .9 .9 .1)

(.1 .9 .9 .9 .1          -- "2"
 .9 .1 .1 .1 .9
 .1 .1 .1 .9 .1
 .1 .1 .9 .1 .1
 .1 .9 .1 .1 .1
 .9 .9 .9 .9 .9)

(.1 .9 .9 .9 .1          -- "3"
 .9 .1 .1 .1 .9
 .1 .1 .9 .9 .1
 .1 .1 .1 .1 .9
 .9 .1 .1 .1 .9
 .1 .9 .9 .9 .1)

(.9 .1 .1 .9 .1          -- "4"
 .9 .1 .1 .9 .1
 .9 .1 .1 .9 .1
 .9 .9 .9 .9 .9
 .1 .1 .1 .9 .1
 .1 .1 .1 .9 .1)
```

```
(.9 .9 .9 .9 .9          -- "5"
 .9 .1 .1 .1 .1
 .9 .9 .9 .9 .1
 .1 .1 .1 .1 .9
 .1 .1 .1 .1 .9
 .9 .9 .9 .9 .1)

(.1 .1 .9 .1 .1          -- "6"
 .1 .9 .1 .1 .1
 .9 .1 .9 .9 .1
 .9 .9 .1 .1 .9
 .9 .1 .1 .1 .9
 .1 .9 .9 .9 .1)

(.9 .9 .9 .9 .9          -- "7"
 .1 .1 .1 .1 .9
 .1 .1 .1 .9 .1
 .1 .1 .9 .1 .1
 .1 .9 .1. 1. 1
 .9 .1 .1 .1 .1)

(.1 .9 .9 .9 .1          -- "8"
 .9 .1 .1 .1 .9
 .1 .9 .9 .9 .1
 .9 .1 .1 .1 .9
 .9 .1 .1 .1 .9
 .1 .9 .9 .9 .1)

(.1 .9 .9 .9 .1
 .9 .1 .1 .1 .9
 .9 .1 .1 .9 .9
 .1 .9 .9 .1 .9
 .1 .1 .1 .1 .9
 .1 .1 .1 .1 .9)
```

Figure 17.
Network Test Output Vectors

Outputs for Input 1:
(0.002 0.890 0.995 0.217 0.095 0.050
0.139
)

Outputs for Input 2:
(0.002 0.872 0.996 0.013 0.151 0.149
0.828
)

Outputs for Input 3:
(0.004 0.938 0.995 0.017 0.185 0.917
0.419
)

Outputs for Input 4:
(0.003 0.858 0.992 0.137 0.125 0.729
0.871
)

Outputs for Input 5:
(0.001 0.899 0.999 0.143 0.771 0.032
0.135
)

Outputs for Input 6:
(0.002 0.787 0.996 0.180 0.714 0.087
0.879
)

Outputs for Input 7:
(0.002 0.872 0.988 0.056 0.690 0.609
0.204
)

Outputs for Input 8:
(0.004 0.904 0.993 0.021 0.780 0.924
0.723
)

Outputs for Input 9:
(0.002 0.819 0.993 0.720 0.106 0.105
0.183
)

Outputs for Input 10:
(0.001 0.802 0.996 0.668 0.211 0.175
0.943
)

The file shown above is the output vectors from the network that was translated to OCT. Input 1 is the character "0" and input 10 is the character "9" with the others in sequential order.

APPENDIX B

NETS-TO-OCT TRANSLATOR

Figure 18.
NETS-to-OCT Translator.

```
/*****************************************************************************
*****************************************************************************

        Program:        NETS WEIGHT MATRIX TO BDS
        Programmer:     Grant P. Beagles
        Date:           August 21, 1991
        Revision:       1

        Description:    This program reads a NETS portable weight
                        matrix file (pwt) and generates the BDS
                        description of the neural network.

*****************************************************************************
*****************************************************************************/

#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <ctype.h>

main()
{
void    build_BDS();            /* function to build BDS code */
int     build_header();         /* function to build BDS file header */

FILE *in,*out;

char input_filename[31],
    output_filename[31];

int number_layers,              /* number of layers in the network */
   i,j,x,y,
   layer_nodes[19],             /* This array of nodes is used to          */
                                /* determine the number of connections     */
                                /* between the layers                      */
   sum_size,

   scale,                       /* weight scaling factor */
   current_upper_layer;         /* current upper layer */
```

```
clrscr();
i=0;
j=0;

do {
        printf("\nInput file path : ");
        scanf("%s",input_filename);

        printf("\nOutput file path : ");
        scanf("%s",output_filename);

        printf("\nNumber of layers : ");
        scanf("%d",&number_layers);

        printf("\nInput weight scaling factor : ");
        scanf("%d",&scale);

        printf("\nIs this correct (y/n)?");

    } while((tolower(getch())) != 'y');

in = fopen(input_filename,"rt");
out = fopen(output_filename,"wt");

clrscr();

while (j <= number_layers-1)
        {
        printf("\nLayer number %d dimensions (x y): ",j+1);
        scanf("%d%d",&x,&y);
        layer_nodes[j] = x*y;
        j++;
        }       /** end of while **/

sum_size =build_header(out,layer_nodes,number_layers);

while (i <= number_layers-2)
        {
        current_upper_layer = i;

build_BDS(in,out,current_upper_layer,layer_nodes[i],layer_nodes[i+1],scale,sum_size,n
umber_layers);
        i++;
        }       /** end of while **/
```

```
        fprintf(out,"\nENDROUTINE;\nENDMODEL;");

        fclose(in);
        fclose(out);

        printf("\n\nBDS file construction complete.\n\n\n\n\n");
}       /** end of main **/
```

```
/*********************************************************************
*********************************************************************

        Function:       build BDS header
        Programmer:     Grant P. Beagles
        Date:           August 21, 1991
        Revision:       0

        Description:    This function uses user supplied information
                        matrix file *   (pwt) and generates the BDS
                        to build the header of a BDS file.

*********************************************************************
*********************************************************************/

int build_header(outfile,node_array,layers)

FILE    *outfile;
int     node_array[19],
        layers;


{

        char    string[70];             /* general character string
                                           for informational input */
        int i,j,k,                      /* general purpose integer variables */
           total_nodes;

        i=0;
```

```
clrscr();
printf("Input title for BDS file :");
gets(string);
gets(string);
fprintf(outfile,"!\n!\n!\t%s\n!\n",string);
printf("\nDesigner's name : ");
gets(string);
fprintf(outfile,"!\tdesigned by:\t%s\n",string);
printf("\nrevision date : ");
gets(string);
fprintf(outfile,"!\trevision date:\t%s\n",string);
printf("\nrevision number : ");
scanf("%s",&string);
fprintf(outfile,"!\trevision no.:\t%s\n!\n!\n\n",string);
printf("\nModel name : ");
scanf("%s",&string);
fprintf(outfile,"MODEL %s\n",string);

fprintf(outfile,"\tout<%d:0>=in<%d:0>;\n",node_array[layers-1],node_array[0]-1);

printf("\nEnter number of bits to be used in summing : ");
scanf("%d",&k);
k--;
i = 0;
fprintf(outfile,"\tSTATE\tinner%d<%d:0>,\n",i,node_array[i]-1);
i++;

while(i <= layers-3)
        {
        fprintf(outfile,"\t\tinner%d<%d:0>,\n",i,node_array[i]-1);
        i++;
        }
i = 1;          /* total nodes reflects the total nodes in the
                   network minus the input layer.  */
total_nodes = 0;
while (i <= layers-1)
        {
        total_nodes = total_nodes + node_array[i];
        i++;
        }

i = 0;
while(i <= total_nodes-2)
        {
```

```
            fprintf(outfile,"\t\tsum%d<%d:0>,\n",i,k);
            i++;
            }

        fprintf(outfile,"\t\tsum%d<%d:0>;\n\n",i,k);

        printf("\nThreshold constant : ");
        scanf("%s",&string);
        fprintf(outfile,"\tCONSTANT THRESHOLD = %s;\n",string);
        printf("\nRoutine name : ");
        scanf("%s",&string);
        fprintf(outfile,"ROUTINE %s;\n\n",string);
        return(k);
}


/*****************************************************************************
*****************************************************************************

        Function:       NETS to BDS converter
        Programmer:     Grant P. Beagles
        Date:           August 5, 1991
        Revision:       0

        Description:    This function reads a NETS portable weight
                        matrix file (pwt) and generates the BDS
                        description of the neural network.

*****************************************************************************
*****************************************************************************/


void build_BDS(input,output,current,up,low,scale,sum_size,layers)

int     up,low,current,scale,sum_size;
FILE    *input,*output;


{


        int     i,j,                    /* general purpose counters */
                weight,                 /* current value being processed */
                in_layer;
```

```
static int sum_counter,head_counter;   /* sum counter */


j = 0;                  /* node counter */

while (j <= low-1)
        {
        fprintf(output,"!\ttarget layer # %d  node # %d\n",current,j);
                fprintf(output,"sum%d<%d:0> =",head_counter,sum_size);
        i=0;            /* upper layer counter */
        while (i <= up-1)
                {

                fscanf(input,"%d",&weight);
                if (weight != 0)
                        {
                        if (i<= up-2)
                                {
                                fprintf(output,"\t\t(in%d<%d>   *   %d)   +
\n",current,i,weight/scale);
                                }
                        else
                                {
                                fprintf(output,"\t\t(in%d<%d>   *
%d);\n",current,i,weight/scale);
                                }
                        }
                        i++;
                } /* end if */

        if (current <= layers)
                {
                fprintf(output,"IF   sum%d<%d:0>   GEQ
THRESHOLD\n",sum_counter,sum_size);
                fprintf(output,"\tTHEN inner%d<%d>=1\n",current,sum_counter);
                fprintf(output,"\tELSE inner%d<%d>=0;\n",current,sum_counter);
                }
        else
                {
                fprintf(output,"IF   sum%d<%d:0>   GEQ
THRESHOLD\n",sum_counter,sum_size);
                fprintf(output,"\tTHEN out<%d> = 1\n",current,j);
                fprintf(output,"\tELSE out<%d> = 0;\n",current,j);
                }
```

```
                    sum_counter++;
                    fprintf(output,"\n\n");
                    j++;
                    head_counter++;
            }
    }
```

APPENDIX C

OCT FILES

Figure 19.
BDS Description of Input Layer

```
!
!
!       Unitary net
!
!       designed by:  GPB
!       revision date: 9/9/91
!       revision no.:  0
!
!       Revised by Paul Cohen and Kel Winters to bias to eliminate
!               negative products and reduce compare to single bit.
!       Revised by Kel Winters to trim neuron word width to an
!               appropriate value.
!
!       Revised 9-22-91 KW to readjust bias to just about threshold
!       Revised 9-26 KW to fix bad bit number substitutions
!
!       Revised 1-14-91 by Grant P. Beagles to describe only the upper
!               layer.


MODEL inlayer
        inner0<29:0> = in0<29:0>;

STATE           sum0<3:0>,
        sum1<3:0>,
        sum2<2:0>,
        sum3<3:0>,
        sum4<3:0>,
        sum5<3:0>,
        sum6<3:0>,
        sum7<2:0>,
        sum8<2:0>,
        sum9<3:0>,
        sum10<3:0>,
        sum11<3:0>,
        sum12<3:0>,
        sum13<3:0>,
```

```
        sum14<3:0>,
        sum15<2:0>,
        sum16<3:0>,
        sum17<3:0>,
        sum18<3:0>,
        sum19<3:0>,
        sum20<3:0>,
        sum21<3:0>,
        sum22<3:0>,
        sum23<3:0>,
        sum24<1:0>,
        sum25<4:0>,
        sum26<3:0>,
        sum27<4:0>,
        sum28<3:0>,
        sum29<3:0>;

ROUTINE upper_layer;

!       target layer # 0  node # 0
sum0<3:0> = 8
                + in0<8>
                + in0<10>
                - in0<16>
                - in0<19>
                + in0<21>
                - in0<22>
                - in0<24>
                + in0<27>
                + in0<28>;
IF sum0<3> EQL 1
        THEN inner0<0> = 1
        ELSE inner0<0> = 0;


!       target layer # 0  node # 1
sum1<3:0> =  8
                - in0<1>
                - in0<3>
                + in0<4>
                - in0<8>
                + in0<9>
                - in0<10>
                - in0<11>
```

```
                - in0<12>
                + in0<13>
                - in0<19>
                + in0<20>
                + in0<21>
                + in0<25>;
IF sum1<3> EQL 1
        THEN inner0<1> = 1
        ELSE inner0<1> = 0;


!       target layer # 0  node # 2
sum2<2:0> = 4
                + in0<1>
                + in0<7>
                - in0<15>
                - in0<21>;
IF sum2<2> EQL 1
        THEN inner0<2> = 1
        ELSE inner0<2> = 0;


!       target layer # 0  node # 3
sum3<3:0> = 8
                - in0<2>
                + in0<3>
                + in0<5>
                - in0<6>
                - in0<9>
                - in0<12>
                + in0<15>
                + in0<16>
                + in0<19>
                - in0<26>;
IF sum3<3> EQL 1
        THEN inner0<3> = 1
        ELSE inner0<3> = 0;


!       target layer # 0  node # 4
sum4<3:0> = 8
                + in0<2>
                + in0<4>
                - in0<10>
```

```
                - in0<11>
                - in0<15>
                + in0<18>
                - in0<21>
                + in0<25>
                + in0<28>;
IF sum4<3> EQL 1
        THEN inner0<4> = 1
        ELSE inner0<4> = 0;


!       target layer # 0  node # 5
sum5<3:0> = 8
                + in0<3>
                + in0<4>
                + in0<6>
                + in0<14>
                + in0<19>
                - in0<25>
                + in0<26>;
IF sum5<3> EQL 1
        THEN inner0<5> = 1
        ELSE inner0<5> = 0;


!       target layer # 0  node # 6
sum6<3:0> = 8
                - in0<3>
                - in0<5>
                + in0<9>
                - in0<17>
                - in0<24>;
IF sum6<3> EQL 1
        THEN inner0<6> = 1
        ELSE inner0<6> = 0;


!       target layer # 0  node # 8
sum7<2:0> = 4
                - in0<0>
                + in0<2>
                + in0<5>
                - in0<21>
                - in0<27>;
```

```
IF sum7<2> EQL 1
        THEN inner0<7> = 1
        ELSE inner0<7> = 0;


!       target layer # 0  node # 8
sum8<2:0> = 4
                + in0<1>
                - in0<5>
                - in0<13>
                - in0<21>;
IF sum8<2> EQL 1
        THEN inner0<8> = 1
        ELSE inner0<8> = 0;


!       target layer # 0  node # 9
sum9<3:0> = 8
                - in0<4>
                + in0<8>
                + in0<11>
                + in0<13>
                + in0<15>
                + in0<18>
                + in0<19>
                - in0<20>
                - in0<27>;
IF sum9<3> EQL 1
        THEN inner0<9> = 1
        ELSE inner0<9> = 0;


!       target layer # 0  node # 10
sum10<3:0> = 8
                - in0<0>
                + in0<4>
                - in0<5>
                - in0<23>
                - in0<27>;
IF sum10<3> EQL 1
        THEN inner0<10> = 1
        ELSE inner0<10> = 0;
```

```
!       target layer # 0  node # 11
sum11<3:0> = 8
                - in0<0>
                + in0<1>
                + in0<9>
                + in0<11>
                - in0<16>
                + in0<17>
                - in0<20>
                - in0<21>
                + in0<22>
                - in0<25>
                - in0<28>;
IF sum11<3> EQL 1
        THEN inner0<11> = 1
        ELSE inner0<11> = 0;


!       target layer # 0  node # 12
sum12<3:0> = 8
                - in0<0>
                - in0<7>
                - in0<9>
                - in0<10>
                + in0<11>
                + in0<15>
                + in0<21>;
IF sum12<3> EQL 1
        THEN inner0<12> = 1
        ELSE inner0<12> = 0;


!       target layer # 0  node # 13
sum13<3:0> = 8
                - in0<2>
                + in0<3>
                + in0<14>
                + in0<15>
                - in0<17>
                - in0<26>
                - in0<27>;
IF sum13<3> EQL 1
        THEN inner0<13> = 1
        ELSE inner0<13> = 0;
```

```
!       target layer # 0  node # 14
sum14<3:0> = 8
                - in0<0>
                - in0<2>
                - in0<4>
                + in0<5>
                - in0<7>
                - in0<12>
                - in0<13>
                + in0<15>
                - in0<17>
                + in0<22>
                + in0<23>
                + in0<28>;
IF sum14<3> EQL 1
        THEN inner0<14> = 1
        ELSE inner0<14> = 0;


!       target layer # 0  node # 8
sum15<2:0> = 4
                - in0<11>
                - in0<15>
                + in0<16>
                + in0<19>
                - in0<23>;
IF sum15<2> EQL 1
        THEN inner0<15> = 1
        ELSE inner0<15> = 0;


!       target layer # 0  node # 16
sum16<3:0> = 8
                - in0<0>
                - in0<4>
                - in0<12>
                + in0<21>
                - in0<25>
                - in0<26>;
IF sum16<3> EQL 1
        THEN inner0<16> = 1
        ELSE inner0<16> = 0;
```

```
!       target layer # 0  node # 17
sum17<3:0> = 8
                + in0<1>
                - in0<2>
                - in0<3>
                + in0<9>
                - in0<10>
                + in0<14>
                - in0<25>
                + in0<27>;
IF sum17<3> EQL 1
        THEN inner0<17> = 1
        ELSE inner0<17> = 0;


!       target layer # 0  node # 18
sum18<3:0> = 8
                + in0<1>
                - in0<17>
                - in0<19>
                - in0<22>
                - in0<26>
                - in0<27>;
IF sum18<3> EQL 1
        THEN inner0<18> = 1
        ELSE inner0<18> = 0;


!       target layer # 0  node # 19
sum19<3:0> = 8
                - in0<4>
                + in0<5>
                + in0<7>
                + in0<12>
                - in0<15>
                - in0<21>
                + in0<22>
                - in0<27>
                + in0<29>;
IF sum19<3> EQL 1
        THEN inner0<19> = 1
        ELSE inner0<19> = 0;
```

```
!       target layer # 0  node # 20
sum20<3:0> = 8
                + in0<0>
                + in0<3>
                + in0<4>
                - in0<9>
                - in0<15>
                - in0<20>
                - in0<26>;
IF sum20<3> EQL 1
        THEN inner0<20> = 1
        ELSE inner0<20> = 0;


!       target layer # 0  node # 21
sum21<3:0> = 8
                - in0<2>
                - in0<9>
                - in0<10>
                - in0<18>
                + in0<21>
                - in0<24>;
IF sum21<3> EQL 1
        THEN inner0<21> = 1
        ELSE inner0<21> = 0;


!       target layer # 0  node # 22
sum22<3:0> = 8
                - in0<3>
                + in0<4>
                - in0<6>
                - in0<9>
                + in0<11>
                - in0<14>
                + in0<18>
                - in0<21>
                + in0<24>
                - in0<26>
                - in0<27>;
IF sum22<3> EQL 1
        THEN inner0<22> = 1
        ELSE inner0<22> = 0;
```

```
!       target layer # 0  node # 23
sum23<3:0> = 8
                - in0<4>
                + in0<6>
                + in0<7>
                + in0<10>
                + in0<15>
                - in0<19>
                - in0<20>
                + in0<28>;
IF sum23<3> EQL 1
        THEN inner0<23> = 1
        ELSE inner0<23> = 0;


!       target layer # 0  node # 24
sum24<1:0> = 2
                - in0<29>;
IF sum24<1> EQL 1
        THEN inner0<24> = 1
        ELSE inner0<24> = 0;


!       target layer # 0  node # 25
sum25<4:0> = 16
                + in0<0>
                - in0<1>
                - in0<3>
                - in0<5>
                + in0<6>
                + in0<8>
                - in0<11>
                + in0<13>
                + in0<18>
                - in0<19>
                + in0<20>
                + in0<21>
                + in0<25>
                + in0<28>;
IF sum25<4> EQL 1
        THEN inner0<25> = 1
        ELSE inner0<25> = 0;
```

```
!       target layer # 0  node # 26
sum26<3:0> = 8
                - in0<3>
                + in0<6>
                - in0<7>
                - in0<10>
                - in0<11>
                + in0<13>
                - in0<19>
                + in0<20>
                + in0<21>
                - in0<22>
                + in0<27>;
IF sum26<3> EQL 1
        THEN inner0<26> = 1
        ELSE inner0<26> = 0;


!       target layer # 0  node # 27
sum27<4:0> = 16
                + in0<0>
                + in0<3>
                - in0<4>
                - in0<7>
                - in0<12>
                + in0<14>
                + in0<15>
                - in0<19>
                - in0<20>
                - in0<21>
                - in0<22>
                - in0<23>
                - in0<24>
                - in0<25>
                + in0<28>;
IF sum27<4> EQL 1
        THEN inner0<27> = 1
        ELSE inner0<27> = 0;


!       target layer # 0  node # 28
sum28<3:0> = 8
                + in0<5>
                - in0<9>
```

```
                - in0<13>
                + in0<15>
                - in0<16>
                + in0<22>
                - in0<24>
                + in0<27>;
IF sum28<3> EQL 1
        THEN inner0<28> = 1
        ELSE inner0<28> = 0;


!       target layer # 0  node # 29
sum29<3:0> = 8
                - in0<3>
                + in0<4>
                + in0<6>
                - in0<9>
                + in0<15>
                + in0<16>
                - in0<23>
                - in0<29>;
IF sum29<3> EQL 1
        THEN inner0<29> = 1
        ELSE inner0<29> = 0;

ENDROUTINE;
ENDMODEL;
```

Figure 20.
BDS Description of Output Layer

```
!
!       This file is nn4e.bds
!       Unitary net
!
!       designed by:  GPB
!       revision date: 9/9/91
!       revision no.:  0
!
!       Revised by Paul Cohen and Kel Winters to bias to eliminate
!               negative products and reduce compare to single bit.
!       Revised by Kel Winters to trim neuron word width to an
!               appropriate value.
!
!       Revised 9-22-91 KW to readjust bias to just about threshold
!       Revised 9-26 KW to fix bad bit number substitutions
!       revised 11-1 KW to output inner0 state vars
!               so they can be compressed in misII
!
!       revised 1/20/92 to remove upper layer.  This file represents only
!               the output layer.


MODEL output_layer
        ascii_out<6:0> = inner0<29:0>;
STATE           sum30<4:0>,
        sum31<4:0>,
        sum32<4:0>,
        sum33<4:0>,
        sum34<4:0>,
        sum35<4:0>,
        sum36<4:0>;

ROUTINE network;

!       target layer # 1  node # 0
sum30<4:0> = 16
                - inner0<2>
```

```
                - inner0<3>
                - inner0<5>
                - inner0<6>
                - inner0<8>
                - inner0<9>
                - inner0<13>
                - inner0<16>
                - inner0<19>
                - inner0<22>
                - inner0<24>
                - inner0<25>
                - inner0<27>
                - inner0<29>;
IF sum30<4> EQL 1
        THEN ascii_out<0> = 1
        ELSE ascii_out<0> = 0;

!       target layer # 1  node # 1
sum31<4:0> = 16
                + inner0<1>
                + inner0<4>
                + inner0<12>
                + inner0<13>
                + inner0<14>
                - inner0<17>
                + inner0<21>
                + inner0<26>
                + inner0<27>
                + inner0<28>;
IF sum31<4> EQL 1
        THEN ascii_out<1> = 1
        ELSE ascii_out<1> = 0;


!       target layer # 1  node # 2
sum32<4:0> = 16
                + inner0<0>
                + inner0<3>
                + inner0<4>
                + inner0<5>
                + inner0<6>
                + inner0<9>
                + inner0<16>
                + inner0<17>
```

```
                + inner0<19>
                + inner0<20>
                + inner0<21>
                + inner0<22>
                + inner0<27>
                + inner0<28>;
IF sum32<4> EQL 1
        THEN ascii_out<2> = 1
        ELSE ascii_out<2> = 0;


!       target layer # 1  node # 3
sum33<4:0> = 16
                - inner0<0>
                - inner0<1>
                - inner0<4>
                + inner0<6>
                + inner0<7>
                - inner0<8>
                + inner0<9>
                + inner0<11>
                + inner0<12>
                + inner0<13>
                - inner0<21>
                - inner0<23>
                - inner0<24>
                - inner0<25>
                - inner0<28>;
IF sum33<4> EQL 1
        THEN ascii_out<3> = 1
        ELSE ascii_out<3> = 0;


!       target layer # 1  node # 4
sum34<4:0> = 16
                - inner0<0>
                + inner0<3>
                - inner0<6>
                + inner0<10>
                - inner0<11>
                - inner0<14>
                - inner0<16>
                - inner0<17>
                - inner0<19>
```

```
                + inner0<20>
                - inner0<21>
                + inner0<22>
                + inner0<24>
                + inner0<25>
                - inner0<28>
                + inner0<29>;
IF sum34<4> EQL 1
        THEN ascii_out<4> = 1
        ELSE ascii_out<4> = 0;


!       target layer # 1  node # 5
sum35<4:0> = 16
                + inner0<1>
                - inner0<3>
                + inner0<4>
                - inner0<11>
                - inner0<13>
                - inner0<14>
                + inner0<15>
                + inner0<18>
                - inner0<20>
                - inner0<22>
                - inner0<23>
                - inner0<24>
                + inner0<25>
                + inner0<26>
                - inner0<27>
                - inner0<28>
                - inner0<29>;
IF sum35<4> EQL 1
        THEN ascii_out<5> = 1
        ELSE ascii_out<5> = 0;


!       target layer # 1  node # 6
sum36<4:0> = 16
                - inner0<0>
                + inner0<2>
                - inner0<3>
                + inner0<4>
                - inner0<6>
                + inner0<7>
```

```
                - inner0<9>
                + inner0<10>
                - inner0<12>
                - inner0<14>
                + inner0<15>
                + inner0<19>
                + inner0<20>
                + inner0<22>
                - inner0<24>
                - inner0<25>
                - inner0<26>
                - inner0<27>
                - inner0<28>;
IF sum36<4> EQL 1
        THEN ascii_out<6> = 1
        ELSE ascii_out<6> = 0;


ENDROUTINE;
ENDMODEL;
```

Figure 21.
BDNET File to Connect Network Layers

```
MODEL Numrec:unplaced;
TECHNOLOGY scmos;
VIEWTYPE SYMBOLIC;
EDITSTYLE SYMBOLIC;

INPUT in0<29:0>;
OUTPUT ascii_out<6:0>;


        INSTANCE In40:logic PROMOTE;

        INSTANCE Out40:logic PROMOTE;

ENDMODEL;
```

Figure 22.
MUSA Simulation

Note that the output is in hex.

```
musa: Warning: There are no SUPPLY nodes
musa: Warning: There are no GROUND nodes
Source Flags: F   Depth 1
MUSA(1)> mv in in0<29:0>
MUSA(1)> mv out ascii_out<0:6>
MUSA(1)> se in b01110100011000110001000101110
MUSA(1)> ev
MUSA(1)> sh out
 out:H30
MUSA(1)> se in b01110001000010000100001000100
MUSA(1)> ev
MUSA(1)> sh out
 out:H31
MUSA(1)> se in b11111000100010001000100101110
MUSA(1)> ev
MUSA(1)> sh out
 out:H32
MUSA(1)> se in b01111100001000001100100101110
MUSA(1)> ev
MUSA(1)> sh out
 out:H33
MUSA(1)> se in b01000010001111101001010010100 1
MUSA(1)> ev
MUSA(1)> sh out
 out:H34
MUSA(1)> se in b01111100001000001111000011111 1
MUSA(1)> ev
MUSA(1)> sh out
 out:H35
MUSA(1)> se in b01110100011001101101000100010 0
MUSA(1)> ev
MUSA(1)> sh out
 out:H36
MUSA(1)> se in b00001000100010001000100001111 1
```

```
MUSA(1)> ev
MUSA(1)> sh out
 out:H37
MUSA(1)> se in b01110100011000111111000101110
MUSA(1)> ev
MUSA(1)> sh out
 out:H38
MUSA(1)> se in n10000100001011011001100010111 0
MUSA(1)> ev
MUSA(1)> sh out
 out:H39
```

Figure 23.
CHIPSTATS for Numeral Recognition Neural Chip

```
chipstats: Version 1.0 : Chip  Numrec:placed

        Number of instances     = 1851
        Number of nets          = 1883
        Number of connectors    = 14224
        Number of vias          = 9674

        Average instance area  =        2747.41
        St.dev. instance area  =         751.66
        Chip Area W x H        =   2776 x  5355
        Chip Area              =   14865480.00 (100.00%)
        Total Instance Area    =    5085464.00 ( 34.21%)
        Routing + Empty Area   =    9780016.00 ( 65.79%)
        Total   net area       =    6148405.00
        Total   net length     =    1758595.00
        Average net length     =        933.93
        St.dev. net length     =       2963.64

***** 6 LONGEST NETS
(---) NAME              LENGTH (lambda)  AREA (lambda^2)
(  1) GROUND                 70476.00         651868.00
(  2) SUPPLY                 68824.00         638652.00
(  3) in0<21>                18155.50          54466.50
(  4) in0<13>                17136.00          51408.00
(  5) [5609]                 15794.50          47383.50
(  6) in0<4>                 14870.50          44611.50
```