8-2004

Development of an FPGA-Based Hardware Evaluation System for Use with GA-Designed Artificial Neural Networks

Dennis Duncan Earl
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a dissertation written by Dennis Duncan Earl entitled "Development of an FPGA-Based Hardware Evaluation System for Use with GA-Designed Artificial Neural Networks." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Electrical Engineering.

Donald Wayne Bouldin, Major Professor

We have read this dissertation and recommend its acceptance:

Dr. Mongi Abidi, Dr. Seong-Gon Kong, Dr. Bruce MacLennan

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a dissertation written by Dennis Duncan Earl entitled "Development of an FPGA-Based Hardware Evaluation System for Use with GA-Designed Artificial Neural Networks." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Electrical Engineering.

Donald Wayne Bouldin
Major Professor

We have read this dissertation
and recommend its acceptance:

Dr. Mongi Abidi

Dr. Seong-Gon Kong

Dr. Bruce MacLennan

Accepted for the Council:

Anne Mayhew
Vice Chancellor and
Dean of Graduate Studies

(Original signatures are on file with official student records.)

DEVELOPMENT OF AN FPGA-BASED HARDWARE EVALUATION SYSTEM FOR USE WITH GA-DESIGNED ARTIFICIAL NEURAL NETWORKS

A Dissertation

Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Dennis Duncan Earl

August 2004

DEDICATION

This dissertation is dedicated to my grandfather, Duncan C. Earl, and my father, Dennis D. Earl, whose personal evolution made my progression possible.

ACKNOWLEDGEMENTS

I would like to thank my mentoring professor, Dr. Bouldin, for his support and guidance throughout this research. I would like to thank my committee members, Dr. Abidi, Dr. Kong, Dr. MacLennan, and Dr. Smith for their participation, encouragement, and advice. I would also like to thank Oak Ridge National Laboratory for providing me with the paid, two-year, educational sabbatical that made this endeavor financially possible. And lastly, I would like to thank my wife, son, and all of my immediate family for the countless shared hours they forfeited in support of this effort.

ABSTRACT

The Hardware-Evolved Digital Artificial Neural Network (HEDANN) design platform is a circuit design platform built to evolve complex architecture ANN circuits using re-configurable hardware. By using genetic algorithms to evolve complex architecture ANN designs in field programmable gate arrays, this system is the first design system to evolve physical ANN circuits with unconstrained network architectures. With the HEDANN design system, the evolution of ANNs with recursive, non-layered, complex architectural connections is made possible. In addition, the HEDANN design system is capable of evolving device-independent circuit designs that can operate properly across a wide range of operating temperatures. This system is presented as a powerful new tool for researchers working to develop both artificially intelligent systems and complex evolvable hardware. To demonstrate the potential benefits of this unique design platform, the details of two trial experiments are presented and the results discussed.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 Introduction

1.1 Overview

In the 1940s, developed nations were enjoying a multitude of scientific successes, ranging from the development of new plastic materials to the harnessing of atomic energy. At that time in history, these breakthroughs must have appeared to be the pinnacle of an unsustainable technological surge. However, nothing could have been further from the truth. With the benefit of hindsight, we now know that people of the 1940s were witnessing the dawn of an unprecedented technological revolution. A sustained revolution that would dramatically affect the lives of people throughout the world. Often called the "Digital Revolution", these coming technological advancements would begin with the introduction of the transistor and eventually result in many of today's common technologies, such as the laptop computer, global positioning systems, and the cellular telephone — technologies that would likely have been considered science fiction by most in the 1940s.

Today, the Digital Revolution has demonstrated just how dramatically technology can affect society. Millions of desktop computers connected to the World Wide Web are redefining business practices, opening new markets, and permitting an unprecedented level of worldwide communication and cooperation. Militarily and politically, the use of digital microchips for guidance and navigation in precision "smart" missiles has redefined the way wars are fought and won by industrialized countries and has provided a strategic capability that has affected the foreign policy of developed nations. Socially,

digital microchips have strengthened the public's use of, integration with, and dependence on digital devices such as cellular telephones, pacemakers, Automated Teller Machines (ATMs), and personal digital assistants (PDAs).

Despite the impressive achievements of the digital microchip, just as in the 1940's, today's society may only be witnessing a glimpse of what is soon to come. The capabilities and impact of the digital microchip could one day be dwarfed by the introduction of an "intelligent" electronic processor, a processor capable of mimicking simple human skills. Intelligent processors that could be reliably used to navigate a vehicle, understand human commands, mimic gross human motor skills, translate languages, recognize faces, identify common objects, etc. would dramatically affect the way humans interact with, work with, and rely upon machines. The development of a truly "intelligent" electronic processor could usher in one of the greatest technological revolutions to date, blurring the boundaries between humans and electronics and altering life and society as we know it.

Just as researchers of the 1940s struggled to understand the physics required to develop primitive transistors, many researchers today strive to understand the fundamental processes required in creating a truly intelligent processor. Of this research, the most successful efforts are looking to nature for inspiration, believing that to develop a technology that mimics human processing one must duplicate, at least partially or in whole, the natural methods and mechanisms used to develop the biological processor, or brain. Of the many approaches explored, three design methods have demonstrated continued success in artificial intelligence (AI) applications. These approaches include: artificial neural networks (ANN), genetic algorithms (GA), and evolvable hardware

(EHW). As will be discussed in proceeding chapters, each of these techniques has demonstrated some success at solving simple AI problems. However, for more challenging and interesting problems, these techniques have produced disappointing results and the cause of their limitations is currently the source of much debate in the AI community.

It will be argued in this dissertation that current AI techniques are limited because they lack one or more of three "essential components" of natural design. To produce artificial processors with complexity and capabilities rivaling those of biological processors, it is necessary to acknowledge and preserve the synergistic relationship between these three components. It is proposed that these essential components include:

- **The Neuron** - In Nature, the neuron is the fundamental building block of all biological intelligence on earth.
- **Genetic Encoding** – In Nature, complex indirect encoding of genetic information allows high-level traits/features of an individual to be efficiently transferred from one generation to the next. Successful genetic encoding and genetic operators play a vital role in Nature's efficient creation of new populations from past generations.
- **Evolution in the "Physical-World"** – In Nature, populations are evolved using Darwinian principles of evolution, executed in, and constrained only by, the physical world.

As will be discussed in proceeding chapters, attempts to design intelligent processors using only one or two of these three essential components have resulted in limited success. Although the importance of each component is separately acknowledged in existing AI design methods (such as ANN, GA, and EHW research), no design methodology has yet to combine all three of these components within a singular approach. Consequently, the synergistic benefits of these three components have yet to be demonstrated.

1.2 Goals and Expected Contributions

The goal of this dissertation is to develop and evaluate a design methodology that captures all three of the essential components of natural design into a single system. Using Field Programmable Gate Array (FPGA) technology to implement digital artificial neurons in hardware, the Hardware-Evolvable Digital Artificial Neural Network (HEDANN) design platform will be presented. This system is expected to be the first AI design platform to permit ANN networks, of any architecture, to be realized and evaluated in the physical world. By using a user-configurable genetic algorithm to evolve complex-architecture ANNs in hardware, this system resembles a conglomerate of past ANN, GA, and EHW research. However, due to synergistic relationships, it is expected that this approach will have the potential to solve significantly more complex AI problems.

In addition to being a powerful new design methodology, it is expected that this platform will also be valuable to researchers investigating and evaluating various new techniques related to evolutionary design. In this way, areas currently being researched

and developed, such as complex indirect genetic encoding schemes, could be more effectively explored using this new flexible platform. In some instances, this platform is even expected to eliminate the limitations of past evolutionary techniques, such as the unwanted environmental sensitivity of past evolvable hardware design techniques.

To familiarize the reader with relevant background material, an abbreviated review of ANN, GA, and EHW theory and research will be presented in Chapter 2. In Chapters 3 and 4, the details of the HEDANN design platform will be described. In Chapters 5 and 6, the details and results of two experiments demonstrating the potential of the HEDANN design platform will be described while Chapter 7 identifies some of the unique results/benefits attributed to this new system. Chapter 8 provides concluding remarks and a recommended direction for future research.

2.0 Background

2.1 Introduction

The HEDANN design platform presented in this dissertation allows artificial neural networks, of any architecture, to be realized and evaluated in the physical world. The foundation of this platform is an amalgam of artificial neural network, genetic algorithm, and evolvable hardware theory. Consequently, before discussing the specifics of the HEDANN design platform, a brief review of ANN, GA, and EHW research is in order. This chapter contains three sections to address each of these topics individually and with consideration of their interdependencies. Although this chapter provides many of the fundamental theories of ANN, GA, and EHW, it is not meant to be an exhaustive review of any of these topics. As such, the reader is directed to various sources throughout the text for further information.

2.2 Artificial Neural Networks

The field of artificial neural networks covers a vast and diverse number of applications and research efforts. This section, however, provides only an abbreviated review of ANN theory so that the reader is familiarized with the most common ANN terms and models. In addition, the limitations and failures of past and current ANN research are presented along with theories speculating as to the cause of these limitations. For a more thorough review of ANN theory and research, the reader is encouraged to explore References 1-3.

2.2a The Biological Neural Network

The human brain is a biological neural network containing approximately 10 billion neurons connected via 100 trillion interconnections. A simplified diagram of a biological neuron is shown in Figure 1. Although there are many classes of neurons in the human nervous system, the simplified neuron illustrated in Figure 1 is generally accepted as a simplified, but adequate, representation of the typical brain neuron. The cell body contains the axon hillock, which is responsible for summing input signals that are transmitted from neighboring neural cells. Neighboring signals are communicated to the neuron shown in Figure 1 through the neuron's dendrite branches. In the human brain, a single neuron's dendrites might connect the neuron to millions of other neurons from various parts of the brain, or it might connect it to only a handful of neighboring neurons. From these connected neurons, inputs are received. The inputs are weighted and summed, and if the sum is above a certain threshold value, the neuron's cell body transmits an electrical signal down the neuron's axon.

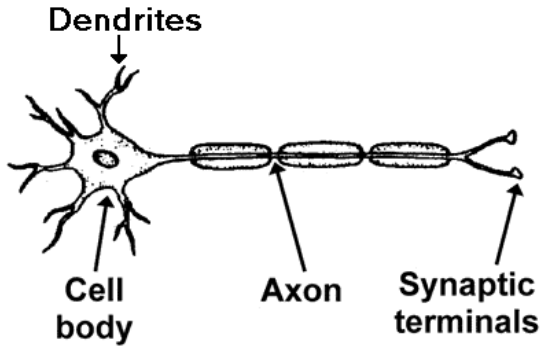


Figure 1. Simplified Biological Neuron Structure

This output signal propagates down the axon and is split into many outputs connected to multiple neighboring neurons through branches of the axon. Branches of the axon connect to the dendrites of other neurons, allowing one neuron's output to become another neuron's input. The axon-to-dendrite communication occurs across a synaptic junction. The synaptic junction, or synapse, is an electro-chemical boundary that transmits and modifies the value of an incoming signal. Therefore, the signal transmitted between two neurons is weighted at the synaptic junction. The value of this weight may be either positive (excitatory) or negative (inhibitory) and the absolute value of this weight (but not the sign) will typically change over time as the neural network learns.

Billions of neurons repeat this process of receiving inputs, weighting, summing, thresholding, and firing, all in parallel. Although, the mathematical operations performed by each neuron may be limited, the function of the network as a whole can be extremely complex. It is generally believed that the power of this type of computational system lies in its large number of independent computational units and their complex interconnectivity.

2.2b The Simple Architecture Artificial Neural Network

In the late 1950's, a biologically-inspired parallel processing device, known as the Perceptron, was invented by psychologist Frank Rosenblatt. This computational device was modeled after the workings of a simplified biological neuron and its development constituted the birth of ANN research. The Perceptron used an artificial neuron model similar to the one illustrated in Figure 2.

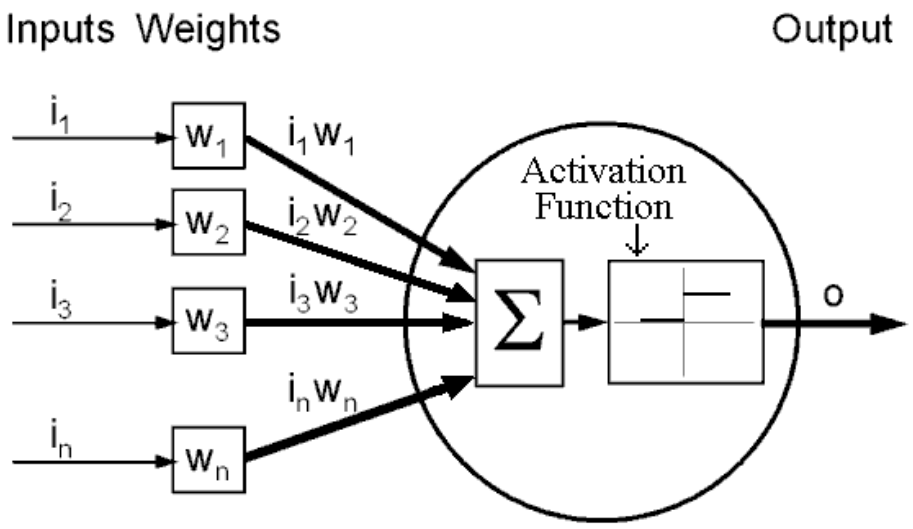


Figure 2. Rosenblatt's Artificial Neuron Model

Multiple inputs (i_1, i_2, i_3, …) are multiplied by a weighting factor (w_1, w_2, w_3, …) and are summed together. Given the value of the summed inputs, an activation function f(net) is used to calculate the output value of the neuron. For Rosenblatt's Perceptron, the activation function used was a simple step-function, as shown in Equation 1 and 2:

$$output = f(net) = \begin{cases} 0 & net < \theta \\ 1 & net \geq \theta \end{cases} \qquad \text{(Equation 1)}$$

$$net = \sum_{j=1}^{N} i_j w_j \qquad \text{(Equation 2)}$$

where θ is defined as the threshold value. Although a Perceptron network can contain as many neurons as desired, a "trainable" Perceptron network required neurons to be

arranged into two, and only two, layers. This two-layer limitation was imposed because the mathematical methods for optimizing network weights more than one layer deep were not known at the time.

In its simplest form, the two-layered Perceptron would be as illustrated in Figure 3. The Perceptron shown in Figure 3 is capable of performing simple logic computations. For example, using a threshold value of 0.5, the weights w_1 and w_2 can be adjusted such that the output of the network can correctly predict the output of an OR gate, whose truth table is shown in Table 1. Adjustment of the weights w_1 and w_2 is accomplished through a simple training procedure whereby the weight of an input is either decreased or increased depending on whether the associated input contributed to the correct or incorrect output response of the Perceptron[1-2].

In 1969, Marvin Minsky and Seymour Papert published a book entitled, *Perceptrons: An Introduction to Computational Geometry,* where they detailed the severe limitations of the two-layer Perceptron. It was shown that the Perceptron was capable of solving only the most trivial problems (specifically those requiring only the differentiation between linearly separable input patterns). As Minsky and Papert correctly pointed out, even a simple problem like predicting the output of an XOR gate (truth table shown in Table 2) is beyond the capabilities of the Perceptron regardless of how many neurons are added to the two layers.

Fortunately, in the 1980s, a new mathematical method for training neural networks with multiple layers was developed. It was discovered, as AI researchers had long suspected, that the addition of multiple layers of neurons in between the input and output

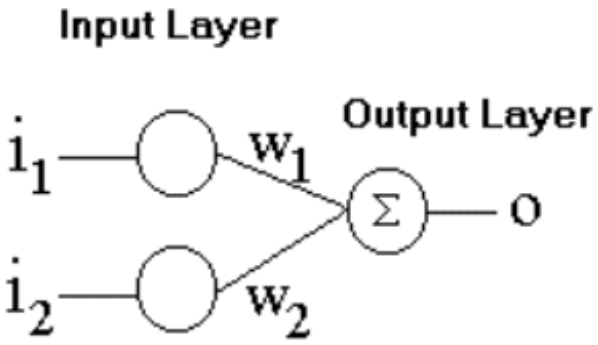


Figure 3. Two-Input, Two-Layer Perceptron

Table 1. OR Truth Table

Input 1 (i_1)	Input 2 (i_2)	Output
0	0	0
1	0	1
0	1	1
1	1	1

Table 2. XOR Truth Table

Input 1 (i_1)	Input 2 (i_2)	Output
0	0	0
1	0	1
0	1	1
1	1	0

layers, called hidden layers, would result in a network capable of classifying more complex patterns. This improvement to the Perceptron model, called the Feed-Forward Back-Propagating Neural Network or "backprop", is now frequently used to create networks capable of solving simple pattern recognition problems. An example of a Feed-Forward Backprop ANN is shown in Figure 4. To train this network, the errors experienced at each network output are computationally back-propagated to the associated hidden nodes/weights within the network using a training technique called the generalized delta rule[4] (GDR). Appropriate weight adjustments to hidden weights can then be made to optimize network performance. For the generalized delta rule to work successfully, the activation function of each neuron must be defined by a mathematically differentiable function.

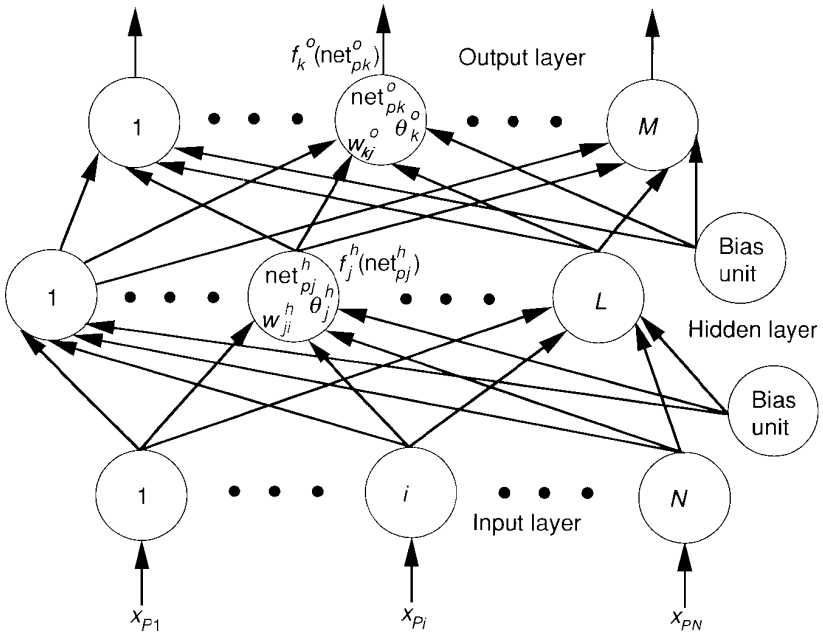


Figure 4. Common Feed-Forward Network With Hidden Layers

Therefore, the sigmoid function, shown in Figure 5, is often used as the neuron's activation function. Training with the generalized delta rule requires that the "flow" of data always proceed sequentially from first layer to last layer, with no recursive connections. Data is not permitted to flow in the opposition direction or to skip a layer. Nonetheless, working within these architectural constraints, researchers have successfully trained Feed-Forward ANNs to perform a wide variety of simple tasks currently in use today.

Although the Feed-Forward BackProp ANN is by far the most commonly applied ANN today, many, many other network architectures, neuron models, and associated training techniques have also been developed. Unfortunately, a description of these various approaches is beyond the scope of this dissertation and, therefore, will not be discussed (see Ref. 4 for a good overview).

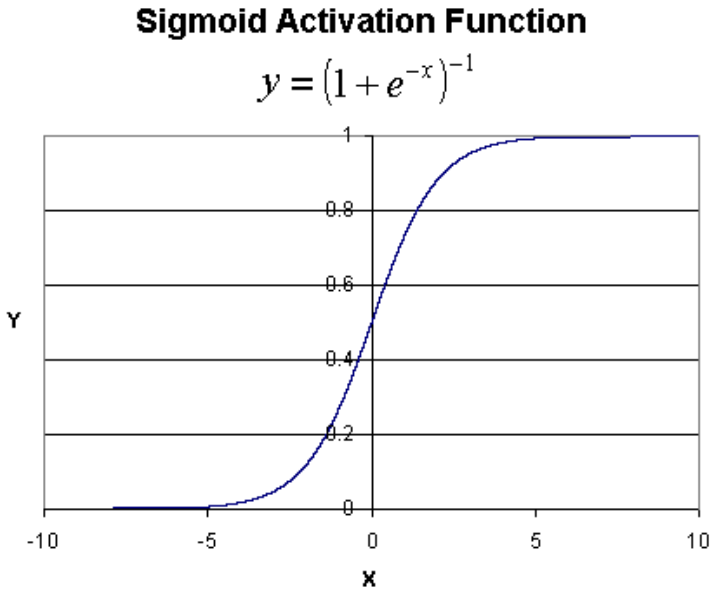


Figure 5. Sigmoid Function

2.2c The Successes and Limitations of ANN Research

Today, artificial neural networks are used in a wide variety of applications. They have been successfully applied to problems where conventional methods have been unsuccessful, such as voice recognition, optical character recognition, weather forecasting, financial modeling, etc. It has been known for some time that the great benefit of an ANN is its ability to learn to recognize a known set of training data and then generalize and apply that information in categorizing future data. Consequently, ANNs are used almost exclusively for pattern recognition problems. In many ways, ANNs today function as a type of "fuzzy" memory. Once a known training set of data has been "memorized", and as long as future data does not deviate significantly from the original training set, an ANN is capable of associating future data with past data. For many applications, the acquisition of data is carefully controlled to minimize deviations, allowing the ANN to perform with a high rate of accuracy. For example, with ANN voice recognition systems, a user typically trains an ANN by repeatedly speaking the words to be learned by the ANN into an input device. After training, the ANN can recognize, with high accuracy, the user's speech, even if the tone or speed of the speech is altered slightly. However, if another user were speaking, the ANN would be unable to recognize their speech without repeating the training process (with the new user). This limited generalization capability, although inferior to biological neural networks, is still valuable to many applications and has resulted in widespread use of ANNs.

As mentioned, ANNs are almost exclusively used today with simple pattern recognition problems. If a problem requires a high degree of generality, traditional ANN designs are often incapable of satisfying them. Only just recently have ANNs been

explored for use as simple controllers[5], though success in these applications is typically limited and the experiments overly-simplified. Ironically, biological systems use ANNs extensively to control complex motor skills and routinely perform sophisticated pattern recognition with a high degree of generality. This difference has caused researchers to ask the question, "What is the feature of a biological neural network that enables it to perform so much more effectively than current ANNs?"

Initially, many researchers believed that the sheer size of biological neural networks explained their increased capabilities. Therefore, it was believed that the more neurons in a network, the greater its capability. Early on, this generally proved to be true, as larger networks were demonstrated to have improved generality over smaller networks. Naturally, researchers working with feed-forward backprop ANNs attempted to solve increasingly complex problems by developing ANNs with more and more neurons. Although, initially a reasonable assumption, there are now many instances that contradict this hypothesis. A perfect example was demonstrated by the CAM-BRAIN project[6-7]. Based on the assumption that larger is better, a 75 million neuron, hardware-implemented, feed-forward ANN, called the CAM-Brain, was built in an attempt to demonstrate advanced ANN intelligence. In the late 1990's the development of the CAM-Brain, with its network size approximating that of a mouse's brain, invited much speculation about the potential capabilities of this massive network. Its supporters envisioned revolutionary speech recognition, image processing, and human interaction capabilities, and its designers even expressed fears that the potential intelligence of the network might quickly become difficult to control. However, ten years after its development, this massive ANN has not demonstrated any unique behaviors beyond

small ANNs, and the potential for demonstrating revolutionary ANN intelligence appears to be unlikely. The CAM-Brain, and other similar experiments, has demonstrated that ANN intelligence may not be determined by size alone. Therefore, there must be some other property of the biological neural network that enables powerful processing. Today, a growing number of ANN researchers believe that network architecture, or the interconnectivity between neurons, is responsible[8].

As discussed in Section 2.2a, a biological neural network exhibits connections between neighboring neurons that are very complex. Far from the simple layered structure of a feed-forward ANN, biological neural networks have complex network architectures that include recursive connections and rarely display a layered structure. If the architectural complexity of biological neural networks could be recreated in an artificial neural network, many researchers believe that the processing capabilities of the ANN could also be recreated. Giving credence to this hypothesis, the development of the Neocognitron, in the early 1980's, demonstrated that a small ANN with a complex architecture could outperform large feed-forward ANNs for certain challenging problems. The network architecture of the Neocognitron was copied from the neural connections of the human retina and it was shown to be capable of unprecedented accuracy in discriminating between alphanumeric characters. It is still used today in many optical character recognition (OCR) systems. Although the network architecture of the Neocognitron was not exceedingly complex, its success nonetheless demonstrated the influence of network architecture on ANN intelligence.

Additional research highlighting the influence of network architecture on ANN intelligence can be found throughout the literature. Although the precise impact of

network architecture on ANN intelligence is not yet completely understood, for the purposes of this dissertation, its importance is recognized and it is assumed to be one of the significant factors influencing ANN intelligence. Given those assumptions, any AI discussion quickly turns to how best to develop the complex architectures that are sought. Reverse-engineering biological networks, as was done with the human retina for the Neocognitron, has proven to be exceedingly difficult. Therefore, ANN researchers are looking to more practical methods for developing network architecture. The following section discusses one such option – Genetic Algorithms.

2.3 Genetic Algorithms

In the previous section, it was argued that ANNs with complex architectures most accurately reflect the organization of the biological brain. Many AI researchers believe that complex architecture ANNs are essential to the development of a truly intelligent processor. However, despite a growing consensus, very little research is currently focused on developing ANNs with complex architectures. To help explain this curiosity, this section introduces the reader to the massive problems encountered when allowing ANN network architecture to be unconstrained. In addition, it introduces the reader to genetic algorithms (GAs), one of the few techniques for partially handling these difficulties. Because genetic algorithms are a cornerstone of the HEDANN design system, a general introduction to GAs is provided so that the reader will be familiar with basic theory and terminology. This section also discusses the specific successes and limitations of applying genetic algorithms to ANNs. As in the previous section, this

introduction is not meant to be a complete discourse on the topic of genetic algorithms. Therefore, the reader is directed to References 9-16 for a more thorough review of genetic algorithm theory and research.

2.3a Challenges of Exploring Complex Architecture ANNs

If complex architecture ANNs most closely resembles the biological brain, why are researchers not exploring them more? The reason for the limited exploration can be quickly understood once the scope of the problem is examined: If given a very simple ANN with only two neurons, call them neuron A and neuron B, one could describe the architecture of the network with a simple "connectivity matrix" as shown in Figure 6. Where a "1" in column X and row Y, indicates that neuron X is connected to neuron Y (a "0" would indicate the opposite). Now if both neurons in the network are treated as *unique* nodes (e.g. neuron X connected to neuron Y is unique from neuron Y connected to neuron X), and if we allow at most one recursive connection (e.g. neuron X connected to neuron X), then the total possible number of network architectures is shown in Figure 7. To find the optimal two-neuron network architecture for a particular problem, it might be necessary to form and evaluate all sixteen of the unique network architectures shown in Figure 7.

The total number of unique network architectures can also be computed by realizing that the connectivity matrix shown in Figure 6 has four elements. Each element can be one of two values (either a 0 or a 1), so the total number of unique connectivity matrices is the product of the four element choices, hence 2 x 2 x 2 x 2 = 16.

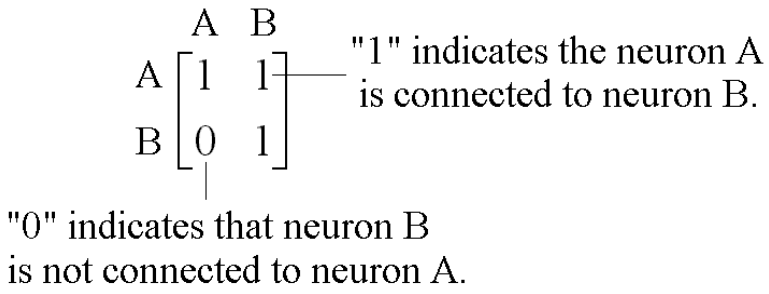


Figure 6. Sample ANN Connectivity Matrix

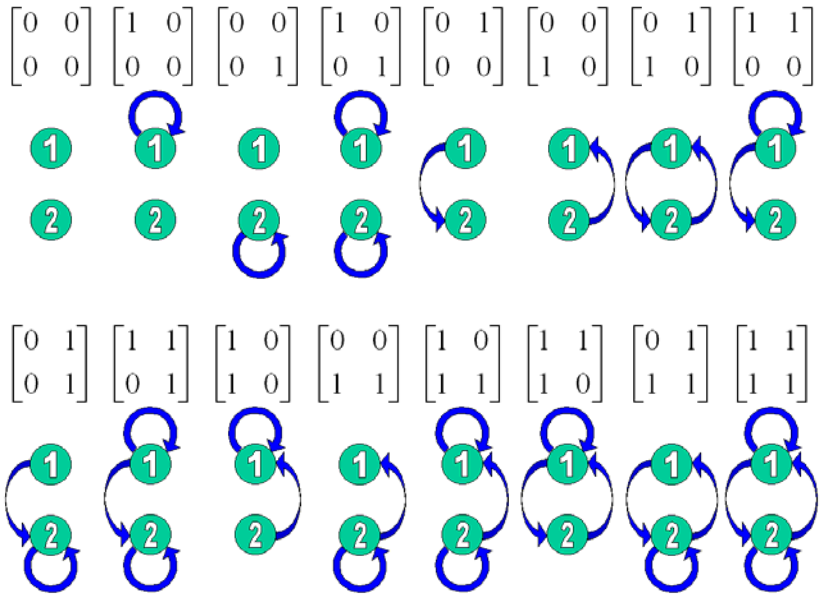


Figure 7. Total Possible Network Architectures for a Two Neuron ANN

Therefore, for an N neuron network, the number of unique network architectures is given by Equation 3:

$$\text{Number of Unique Network Architectures} = 2^{N^2} \qquad \text{(Equation 3)}$$

For a simple ANN with only five neurons, there would be more than 30 million different network architectures to be explored! For the human brain, which has approximately 10 billion neurons, the number of unique network architectures is roughly 10^{542}.

2.3b Use of Genetic Algorithms with ANNs

To explore the vast and complex search space associated with ANNs of variable architecture, an optimization algorithm is needed which is well suited to this type of problem. The genetic algorithm is one such optimization technique. The genetic algorithm is an optimization technique based on the principles of natural selection and genetics. GAs exploit historical information to direct a search into regions of better performance and, as such, are well suited to search spaces that are too large to be exhaustively searched (which is exactly the case when seeking to optimize ANN architectures[17]). GAs were introduced more than three decades ago and they have seen impressive growth and application in the ANN field over the past few years. A number of excellent reviews describe the current state of the art[18-19].

To implement a genetic algorithm, a population of individuals is first constructed, with each member of the population typically represented by a finite string of symbols,

known as the genome, which encodes a possible solution in a given problem space. This space, referred to as the search space, comprises all possible solutions to the problem. The standard genetic algorithm proceeds as follows:

Step 1: An initial population of varying genomes is generated at random.

Step 2: The genome of each individual is transformed into its associated phenotype (a phenotype is what the genome codes for, i.e. a human genome can be transformed into the phenotype of a human, an ANN genome can be transformed into a working ANN design).

Step 3: Each member of the population is evaluated for fitness. Fitness is measured by a fitness function and an associated score. The fitness function is defined by the designer and represents the suitability of a population member at performing a certain task or exhibiting a desired property.

Step 4: Once all members of the population have been evaluated (e.g. one generation), a new population is constructed from the fittest members of the current population. The genomes of the top N-number of fittest members may be cloned, bred, and mutated to construct a new population. The methods for cloning, reproducing, and mutating the current population may be varied.

Step 5: Steps 2-5 are repeated until an acceptable solution has been found.

To understand how genetic algorithms can be applied to designing neural networks, it is helpful to examine a simple example. Suppose the simple neural network in Figure 8 is to be evolved to recognize some pattern of inputs. For certain input patterns the output needs to respond with a one, while for all other input patterns the output should be zero. For the moment, the design of the network architecture will be ignored, and the question of how a genetic algorithm can be used to train the weights of this particular network will be examined.

First, a genome structure is needed that adequately represents the network being trained/evolved. The genome structure will obviously need to describe which neurons are interconnected and what weight values are associated with each connection. The neurons in Figure 8 have been labeled to help simplify this task. If we require that the network in Figure 8 be a feed-forward, fully-connected ANN, then one approach to encoding the neural network might be as shown in Figure 9. The genome specified in Figure 9 is simply a "weight list" vector. As is common in ANN design, the weight values can be any value between –1 and +1. The "sample genome" provides an example of what one member of a population might look like. Obviously, a population of members could easily be generated by creating multiple genomes each filled with randomly assigning values between –1 and +1 for each of the entries in the weight list vector. Although it is highly unlikely that the randomly generated weights would represent a neural network that performed as desired, the randomly generated population does provide a starting point in the search for a suitable solution.

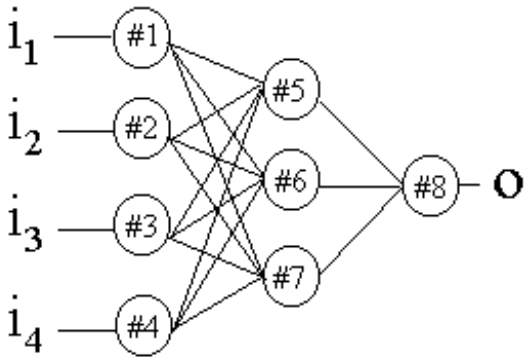


Figure 8: Simple 3-Layer FeedForward ANN

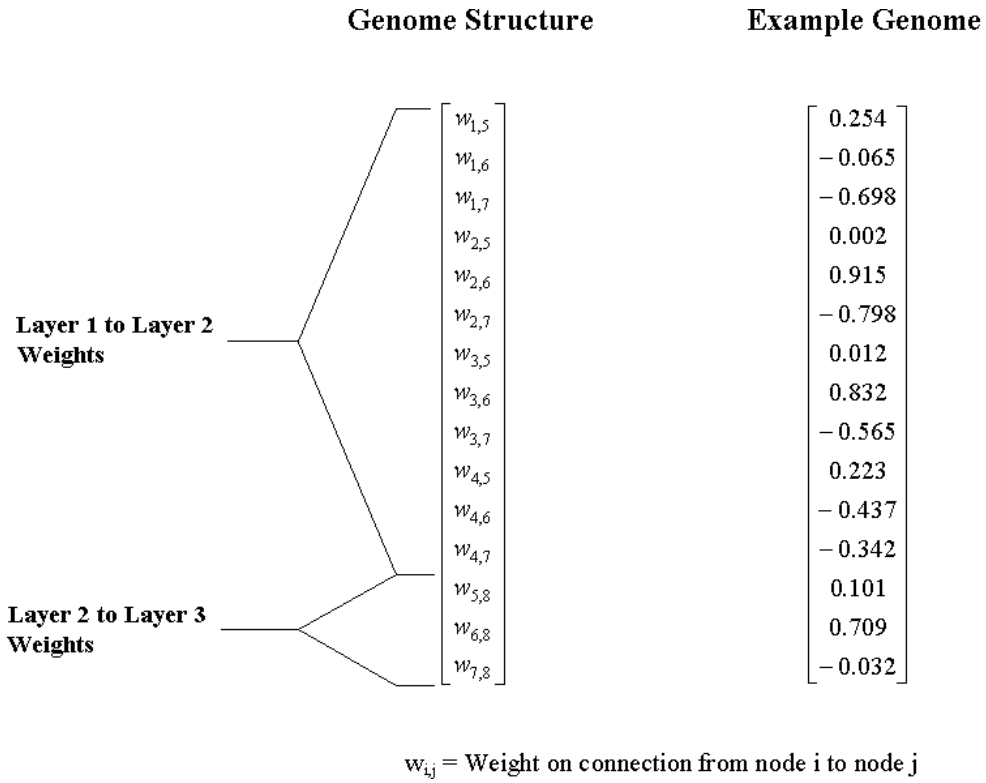


Figure 9: Example of Direct Encoding of ANN Design

Given the starting population, each member's weight list vector would be converted into a neural network (the phenotype). A set of trial input patterns would be applied to each ANN in the population and the output for various patterns simulated. If an ANN generated the correct output for a given input, its fitness score would be increased by some value. If an ANN generated an incorrect output, its fitness score would be decreased by some value.

After exposure to all test inputs, the fitness, or score, of an ANN design could be calculated. Once all members of a population had been evaluated, their scores would be compared and the population sorted from highest to lowest score. Only those genomes that produced a high score would be worth keeping. Therefore, the top ten percent of the population might be selected for future progression while the remaining ninety percent would be discarded.

A new population would be constructed from the top designs by cloning, mutating, and breeding the fittest members. Cloning merely involves copying the current genome into the next population. Cloning is useful because it ensures that "good" solutions are not lost due to non-progressive breeding. However, cloning does not produce any improvements in a population and, therefore, mutations must be introduced to add variation to the population. A mutation operation takes an original genome and slightly alters one or more of its components. For example, a single weight in the genome shown in Figure 9 might be randomly re-generated, or a random amount might be added to one of the various weights. A "rate of mutation" would determine how frequently the mutations occurred within the genome and within the population. A proper mutation rate is essential for timely convergence to an adequate solution.

Breeding provides another method of improving the population of solutions. Breeding takes two genomes and splices them together as illustrated in Figure 10. One or more crossover points are selected at random. Breeding between the two parents produces two offspring, a son and a daughter, where one is the complement of the other. For the simple case of our neural network, this amounts to "weight swapping" about the crossover point.

With a newly constructed population, the process of evaluation, selection, and cloning/mutation/breeding begins again. With each subsequent, generation, the average score of the top N-number of fittest members should increase. After many generations, a solution that fulfills the designer's requirements may be met.

It is not necessarily true that a solution will be found for all problems. If no solution is found within some defined time interval, the genetic algorithm is said to not converge. The factors effecting convergence is a broad and important topic and is covered in many general books and papers on genetic algorithms used for training neural networks[33-37].

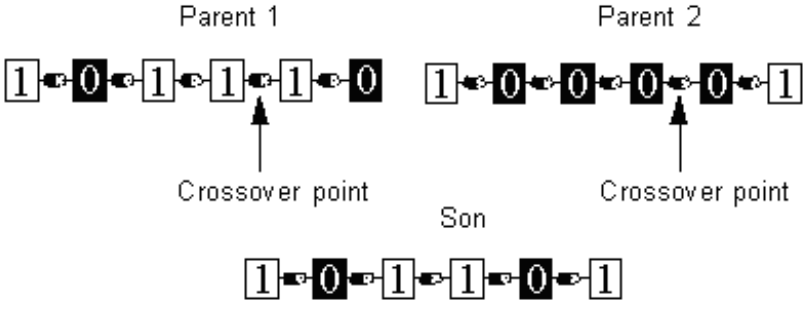


Figure 10: Illustration of GA Breeding Operation

Many more levels of sophistication can be added to the basic GA concept presented here. In particular, the use of dual genome structures (diploid representations), dominant/recessive gene interactions, variable mutation rates, etc. can result in added complexity and improved performance when utilized[20-21].

2.3c Successes and Limitations of Genetic Algorithms

Although a genetic algorithm can be used to evolve an ANN, many problems have been encountered which reduce the efficiency and success of this method. Primarily, these problems relate to how the ANN is genetically "encoded". Just for the simple case of the ANN illustrated in Figure 8, we can see that two symmetrically opposite network structures would have two different genomes, but would function identically. Unfortunately, the offspring resulting from the breeding of these two genomes would likely be non-functioning and would exhibit a much lower fitness value.

Likewise, we can glimpse the much larger problems that occur when we attempt to alter or modify the architecture of an ANN. The encoding scheme, shown previously in Figure 9, required the neurons to be in some defined order and the network to be fully interconnected. If we suddenly do not want to connect neuron #1 to neuron #5, what do we do with the weight value $w_{1,5}$? If we simply remove it, how will we know which weights represent which connections? What if we wanted one member of our population to have only seven neurons instead of eight? How would we breed this seven-neuron network with an eight-neuron network?

A number of schemes have been proposed to deal with these complex encoding problems[22]. Many proposed representations continue to involve the direct encoding of

both the network architecture and the network weight information into a single genome. Unfortunately, most of these representations fail to provide the architectural flexibility needed to fully cover a given search space. The complexity of these proposed solutions can also make implementation tedious. Additional solutions have been proposed which include growing or pruning fully interconnected networks to remove unnecessary neurons. These proposed solutions have shown some success but, unfortunately, also limit architectural flexibility, do not adequately cover the search space, and have complex implementations.

Recently, there have been efforts to create an "indirect" encoding scheme that allows the network architecture of the ANN to be represented at a higher, more flexible, symbolic level[24-28] . Indirect encoding schemes more closely mimic the processes believed to occur with biological DNA. Research has shown that complex network architectures can be represented and successfully bred using a variety of indirect encoding schemes. The potential to maintain favorable "traits" when breeding designs with differing architectures, as well as the potential to easily create redundant network structures, illustrates that indirect encoding may one day be a powerful design tool. However, currently, there are many indirect encoding schemes, each with its own advantages and disadvantages, and no consensus as to which scheme is the best. Furthermore, because these indirect encoding schemes can produce such complex architecture ANNs, and these ANNs can be tedious to realize and evaluate in code, research into indirect encoding of ANNs is progressing very slowly. Before genetic algorithms using indirect encoding can be effectively used to evolve optimal ANN architectures, the various problems associated with representing architecture and neuron

weight values must be remedied. Future tools, such as the HEDANN design platform presented in later chapters, may help GA researchers to evaluate the complex architecture designs that result from indirect encoding and, thereby, help to accelerate the investigation of indirect encoding schemes. In the meantime, this dissertation relies on a traditional direct encoding scheme for representing complex-architecture ANNs.

2.4 Evolvable Hardware

Field Programmable Gate Arrays, or FPGAs, constitute the core hardware of the HEDANN design platform. The HEDANN design platform uses FPGAs to implement and evaluate ANN circuits in a physical medium. Therefore, an introduction to FPGA devices, as well as past efforts that have utilized this technology for similar evolutionary research, is presented in this section.

2.4a Introduction to Field Programmable Gate Arrays

FPGAs are re-programmable digital ICs that were developed in the mid 1980's. FPGAs contain an internal array of logic elements, surrounded by a ring of programmable input/output blocks, all connected together via programmable interconnects. A personal computer can be used to design a digital circuit which is then compiled into a special programming file that will realize the circuit once it is downloaded to the FPGA. High density FPGAs (offering millions of logic gates) are currently available, with the gate density increasing every year. Multiple manufacturers of FPGAs offer a wide variety of FPGA device families.

FPGAs available from Altera, one of the leading FPGA manufacturers, currently consist of thousands of re-configurable logic elements. Each Altera logic element typically contains some number of D-flip-flops, look-up tables, NAND gates, and NOR gates. The wiring to connect these components, as well as the wiring that connects neighboring logic elements, is electrically re-configurable. Therefore, components can be wired together to create one design, erased, and then wired together differently to create a new design. Digital designs can be implemented in FPGAs using tools provided by various chip and software providers. For example, Altera offers the Quartus II software, shown in Figure 11, to help users design circuits specifically targeting Altera FPGAs. FPGA development tools typically follow the same basic design flow. First, a design is created in a high-level language such as VLSI Hardware Description Language (VHDL), or some other alternative (AHDL, graphical schematics, etc.). The high-level code is then converted, in a process called "synthesis," into lower-level logic equations. These logic equations can be expressed in terms of the simple components that make up a device's logic element (i.e. DFF, NAND, NOR and LUTs), and a wiring list (or netlist). Given the synthesized netlist, the FPGA development tools then place and route the appropriate components and wires within the FPGA architecture. Optimally placing and routing a design to ensure efficient use of area and optimized processing speed requires sophisticated algorithms. Various FPGA development tool manufacturers use a variety of algorithms. All are computationally intensive and require a significant amount of time to complete. Once placed-and-routed, designs can be simulated in software or converted into a programming file for download to an FPGA.

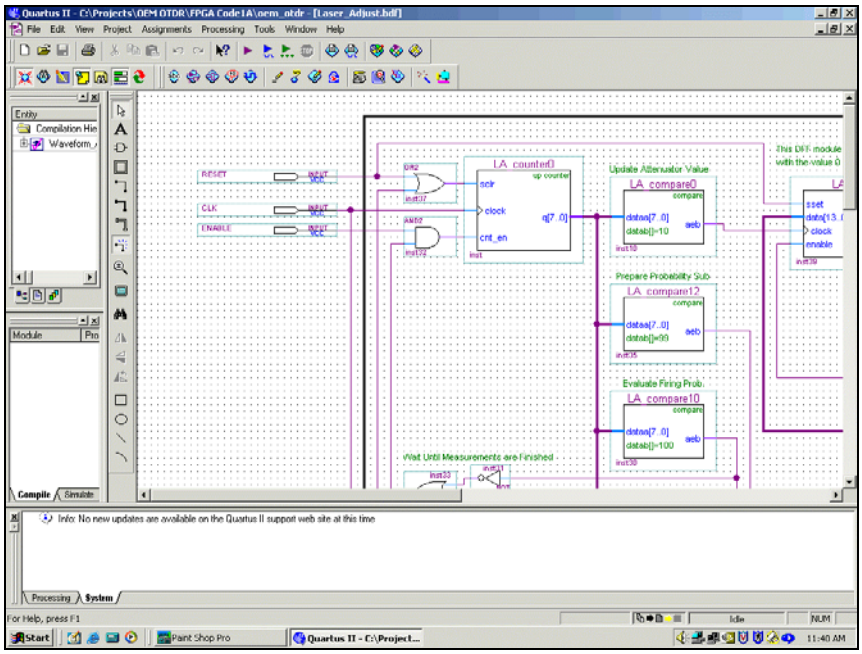


Figure 11. Altera Quartus II Software

The reader should know that this description is a very abbreviated and over-simplified description of the FPGA design process, but it does provide a basic picture of how an FPGA is programmed. For a more detailed description of FPGA design, the reader is directed to Reference 29.

Traditionally, FPGAs have been used as a low-cost, low-risk, method of evaluating digital circuit designs being prepared for fabrication in a foundry. Although not fully equivalent in speed, FPGA implemented designs allow basic design functions to be evaluated with minimal cost and improved ease of debugging. However, with recent increases in FPGA speed, decreased cost, and increased gate density, FPGAs are now used often as an end-product. FPGAs are offered in a wide range of device families and provide powerful hardware acceleration to many applications. In addition, they can

permit the remote reprogramming of hardware, a valuable feature in rapidly developing markets such as the telecommunication industry.

2.4b The Use of FPGAs in Evolvable Hardware

In the mid-1990s, an exciting circuit design technique was studied at the University of Sussex and reported by Adrian Thompson. Thompson's research into evolvable hardware used computer simulations of Darwinian evolution as a way of designing electronic circuits automatically. Working with FPGAs, Thompson showed that a genetic algorithm could be used in wiring simple NAND and NOR gates together to form "bizarre but useful"[30] electronic circuits. His early experiments applied genetic algorithms to the design of FPGA devices for the purpose of speech recognition. Through his combination of genetic algorithms and hardware implementation/evaluation, Thompson showed that it was possible to evolve a two-word speech recognition circuit that performed effectively with less circuitry than conventional design techniques. How the final FPGA design worked was actually unknown, but its ability to differentiate between the words "Stop" and "Go" (spoken only by Thompson) was verified.

Since the FPGA circuit had been designed purely by a GA, it was unclear whether the circuit functioned as a digital, analog, or mixed signal circuit. Thompson observed that the evolved circuit was non-transferable to other FPGA devices and would not function properly if internal connection lengths or operating conditions (primarily temperature) were modified. This observation led Thompson to speculate that the evolved FPGA circuit was actually functioning as a mixed signal system. He proposed that interference effects between nearby logic elements and connection lines, and the inherent, but rarely

exploited, analogue nature of digital components, were being used to the advantage of the FPGA circuit. Because Thompson had given the GA complete design flexibility, the final design was free to utilize all physical phenomena available and, thereby, evolved a more space efficient design than had been previously developed by man.

Thompson's research was "the first reported direct evolution of an evolvable FPGA configuration"[30] and demonstrated the value of including hardware implementation and evaluation *within* a genetic algorithm. If Thompson had tried to merely simulate the performance of his evolving circuit, he would most likely have developed a circuit that did not exploit the useful analog features of FPGA technology and/or he may have failed to develop a working circuit altogether. However, although Thompson's GA evolved circuit was observed to work, its design was not transferable to other FPGA devices, thereby limiting the design's application and inviting questions concerning the circuit's ability to meet operational specifications. Thompson believed that these shortcomings could be overcome by repeating the evolution of the design with a variety of parallel FPGA devices under a variety of parallel operating conditions. Doing so would require additional design time for downloading and evaluating multiple FPGAs. Research into this area is currently ongoing and the field is expanding to include other researchers interested in the problems of high speed FPGA reconfiguration[31-32].

2.4c Successes and Limitations of Evolvable Hardware

Since Thompson's groundbreaking work, additional researchers have published similar examples of "evolution on a chip". This growing field, now called Evolvable Hardware (EHW), encompasses a number of researchers seeking to evolve powerful

circuits using evolutionary design methods with re-configurable hardware. EHW has diversified considerably since Thompson's early work and now encompasses various forms of re-configurable hardware, most notably Field Programmable Analog Arrays (the analog equivalent of digital FPGAs).

Currently, EHW research has produced only marginally interesting designs. Many of these designs continue to suffer the same device-dependent, environmentally sensitive problems of Thompson's original work. In addition, the evolution of complex behaviors is proving to be exceedingly difficult. There are believed to be a number of reasons for these limitations, but this dissertation advocates that the major shortcoming of past EHW research is that it utilizes simple NAND and NOR gates as the "basic building blocks" of a design (or in the case of some FPAA research, individual op-amps/capacitors/resistors). Such primitive computational units offer extreme flexibility in circuit design but also significantly expands the search space of potential solutions and permits the undesirable development of device-dependent designs. Perhaps a slightly more sophisticated, yet still flexible, "building block" can be chosen that can more quickly evolve into a powerful processor exhibiting device-independent, environmentally robust operation. For Nature, the "basic building blocks" of biological processors has always been the neuron. Therefore, if one follows Nature's lead, an artificial neuron model may provide a better "basic building block" for applications seeking to develop robust, intelligent processors. With that said, the research that is described later in this document could be viewed as a reformation of Thompson's original research, substituting artificial neurons instead of NAND/NOR gates as the "basic building blocks" of an evolvable FPGA circuit.

2.4d Using Evolvable Hardware to Develop ANNs

In general, there has been very little published EHW research related to the use of artificial neurons as the fundamental component of an evolvable hardware system. However, there are some notable exceptions[33-34]. In July 2003, research at Ecole Polytechnique Federale de Lausanne (EPFL) was published describing an evolvable hardware system that used spiking neural networks, implemented in an Altera FPGA (with 200,000 gates), to create a run-time re-configurable ANN. The EPFL research sought to evolve an ANN's network architecture for the purpose of creating an intelligent processor that could control the motion of a wheeled robot. Specifically, the re-configurable ANN was being evolved as an obstacle avoidance controller. Obstacle avoidance was possible if the ANN could intelligently interpret sensor data originating from two IR sensors located at strategic positions on the robot. The 2003 publication presented a complex architecture, 64-neuron, FPGA-implemented ANN circuit that successfully achieved obstacle avoidance under certain controlled conditions. Although different in many important ways, EPFL's research echoes the basic themes of this dissertation. Namely, that developing evolvable hardware with higher-level primitives can lead to powerful processing.

However, the EPFL research does also differ significantly from the research presented in this dissertation in several important ways. First, the EPFL research did not permit truly unconstrained evolution of ANN network architecture. To the contrary, the possible connectivity patterns allowed for a neuron were extremely limited. The EPFL researchers allowed a neuron to be connected to other neurons in the network in only one of six possible patterns. Because their network featured 64 neurons, there were actually

10^{19} possible connectivity patterns available to a single neuron within the network. However, to limit resource usage and increase the rate of evolution, the EPFL researchers decided to permit only six of these possible connections. For more challenging problems, it is likely that this decision would have significantly affected the system's potential for success.

In addition, the EPFL research did not actually evaluate the ANN designs in physical hardware. Rather, the ANN designs were implemented in an FPGA, but their success in controlling the robot was simulated on a computer, using measured I/O from the FPGA. Once a suitable design had been evolved, according to the simulator, the FPGA was then installed in the robot and its performance verified. From the 2003 publication, it is evident that some differences were encountered between the simulated performance and the measured performance. However, these differences were not elaborated on. The research presented in this dissertation asserts that hardware evolution should be performed "in-system" whenever possible. This eliminates the need to develop an exhaustive model and provides the truest measure of a design's performance. This improved accuracy allows the evolving processor to exploit all features of the physical world, features that a model may have excluded. Past research related to evolvable hardware and robotics have reached similar conclusions[35].

As will be shown in future chapters, the HEDANN design platform presented in this dissertation provides an evolvable ANN system that is truly unconstrained in its evolution of network architecture. However, it utilizes fundamental components that are more complex, and in many ways more constrained (i.e. the analog properties of the gates are not exploited), than the primitive components used by most EHW researchers. This

choice between constrained and unconstrained evolution is critical to evolving robust solutions that are applicable to today's problems and represents one of the major differences between the HEDANN design platform and other design approaches.

2.4e Efficient ANN Implementation in FPGAs

It is likely that EHW researchers are leery of using artificial neurons as the basic building blocks of their circuits due to the considerable resources consumed by an artificial neuron, if not cleverly designed. Fortunately, stochastic arithmetic techniques can be used to significantly reduce the resources required to implement an artificial neuron in digital logic. Several researchers have been investigating and publishing incremental improvements to this basic technique for the past decade[36-40]. One of the most important of these research efforts, particularly to this dissertation, was performed and reported[41] by Stephen Bade of Brigham Young University in 1994. Bade developed an ingenious lookup table technique that combined stochastic arithmetic bit-stream generation with bit-stream multiplication to create a highly condensed digital artificial neuron. Bade's research exploited the nature of established stochastic arithmetic techniques while providing improvements targeted specifically at FPGA hardware.

Like many other researchers, the main thrust of Bade's research was the space efficient construction of artificial neurons in FPGA devices. As was described in Section 2.2b, artificial neurons must multiply input signals by associated weight values and then add the results. The principal factor affecting the size of FPGA-based ANNs is the high space cost associated with implementing digital multiplication. High-speed, i.e. fully-parallel, ANNs require a tremendous number of multipliers because each synaptic

connection in an ANN requires a single multiplier and the number of synaptic connections typically grows as the square of the number of neurons. Practical implementation of ANNs is only possible if the amount of circuitry devoted to multiplication can be significantly reduced.

At the start of Bade's research, it was known that a highly efficient method for reducing multiplication circuitry was the use of bit-serial *stochastic* computing techniques. Stochastic computing uses relatively long, probabilistic, bit-streams where the numeric value is proportional to the density of "1"s in the bit-stream. The main advantage of stochastic computing is that the multiplication of two probabilistic bit-streams can be accomplished with a single two-input AND gate. Reducing complex multiplication circuits down to a single AND gate makes it much more feasible to implement large, dense, fully parallel ANNs with digital components.

To understand Bade's advancement of the stochastic arithmetic technique, it is necessary to further understand the details of stochastic arithmetic. In stochastic computing, signals are represented as relatively long bit-streams where the actual signal value is encoded as a statistical distribution of "1"s in the bit-stream. The bit-stream representation of values and the basic principles of multiplication and addition are illustrated in Figure 12. In Figure 12, we see how two bit-streams can be added or multiplied together by using either a single OR gate (addition) or a single AND gate (multiplication). Because of the probabilistic nature of the bit-stream representation, it is possible for errors to exist in either of these calculations.

The arithmetic functions illustrated in Figure 12 can be more fully formulated if we define a variable A to represent a binary random variable (i.e. randomly 0 or 1).

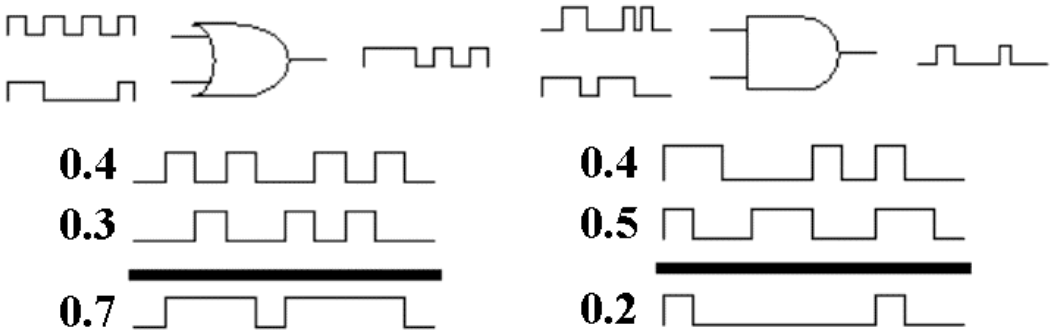


Figure 12. Basic Arithmetic Operations of Stochastic Signals

We'll let the value A(t) represent the value of A at time t. Thereby, the sequence of values, [A(0), A(1),….,A(N-1)], constitutes a stochastic bit-stream of length N. The numeric value that the stochastic bit-stream represents is determined by the number of times A(t) = 1 and the particular number representation chosen. The bit-stream value can be represented using one of two possible number representations[42]:

Representation 1: Unipolar:

The unipolar representation can be used to represent numbers between 0 and 1. The value of a bit-stream is calculated as shown in Equation 4.

$$value = \frac{\sum_{t=0}^{N} A(t)}{N} \qquad \text{(Equation 4)}$$

Thus the maximum value (1) would obviously occur when all N bits of the bit-stream are "1", and the minimum value (0) when all N bits of the bit-stream are "0". The potential error for this representation technique is given by Equation 5.

$$Variance = \frac{p(1-p)}{N} \qquad \text{(Equation 5)}$$

Where p is the number to be represented. The maximum representation error occurs for the number 0.5.

Representation 2: Bipolar:

The bipolar representation can be used to represent numbers between -1 and 1. The value of a bit-stream is calculated using Equation 6.

$$value = \frac{2\sum_{t=0}^{N} A(t)}{N} - 1 \qquad \text{(Equation 6)}$$

Thus the maximum value (1) would occur when all N bits of the bit-stream are "1", and the minimum value (-1) when all N bits of the bit-stream are "0". The potential error for this representation technique is given by Equation 7, with the maximum error occurring for a representation of the number 0.

$$Variance = \frac{4p(1-p)}{N} \qquad \text{(Equation 7)}$$

We can see from Equations 5 and 7, that it is important for the length N of a bit-stream to be sufficiently long. Otherwise errors in number representation can become quite large. For most FPGA applications, a bit-stream length of 256 will keep the representation errors low and represents an arithmetic precision of approximately 8 bits.

In order to use stochastic computing techniques, traditional multi-bit digital signals must be converted into bit-stream representation. To generate a bit-stream requires the use of a bit-stream generator circuit. There are numerous ways to design a bit-stream generator, but all designs first require a random input signal called a *carrier signal*. For FPGA designs, a pseudo-random carrier signal can be generated using a maximal Linear Feed-Back Shift Register (LFSR). A maximal LFSR produces a pseudo-random output that repeats after n^2 clock pulses, where n is the number of D-flipflops in the LFSR design. As an example, an 8-bit LFSR is shown in Figure 13.

To design a bit-stream generator to convert 8-bit digital values into continuous bit-streams requires a series connection of eight *modulator* sub-circuits (as shown in Figure 14). A modulator sub-circuit consists of only four NAND gates and a single D-flipflop. The output from one sub-circuit is fed to the "previous stage" input of the next modulator sub-circuit (except for sub-circuit k-1, which receives a "0"). All modulator sub-circuits also receive the same randomly generated carrier signal from the LFSR previously described in Figure 13. Each of the eight modulator sub-circuits have a *modulation bit*

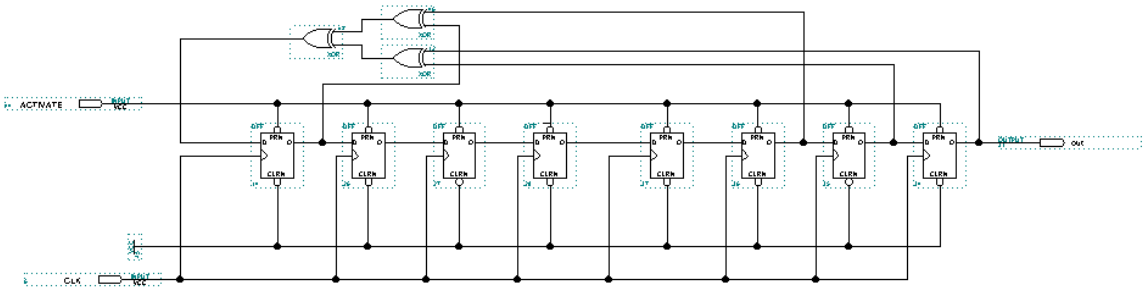


Figure 13. Maximal 8-Bit LFSR Design

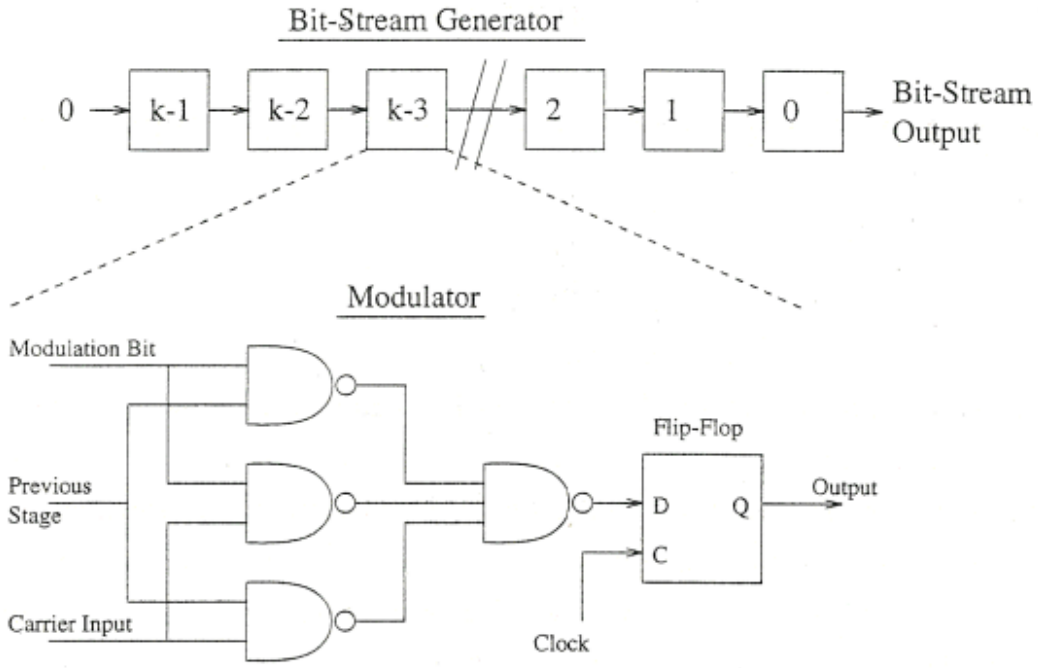


Figure 14. Bit-Stream Generator

associated with them. To convert an 8-bit number into a stochastic bit-stream requires each of these modulation bits to be set appropriately. For example, to represent the unipolar number, $(0.625)_{10} = (.10100000)_2$ would require setting the modulation bit values to: modulation bit(7) and modulation bit(5) equal to 1, all other modulation bits equal to zero.

This bit stream generator design allows each modulator to add a single bit of precision to the final bit-stream. Therefore, to represent an 8-bit number requires eight modulator sub-circuits. To represent a 10-bit number requires ten modulator sub-circuits, and so on.

For a traditional ANN, multiple k-bit digital input values would first be converted into multiple stochastic bit-streams. These inputs would be connected to artificial neurons and each of the inputs multiplied by an associated connection weight. However, to permit stochastic computing techniques, the connection weights would also need to be represented as bit-streams and, consequently, one bit-stream generator would be required for each connection in the ANN. Therefore, a traditional ANN design would require a single LFSR circuit for generating carrier signals, multiple bit-stream generators for converting inputs and weight into bit-streams, and AND/OR gates for multiplying and summing bit-streams.

In addition, as was described in Section 2.2b, a neuron activation function is required to map the sum of the neuron's weighted inputs to a defined output value. A sigmoid activation function is often used to provide a continuous output that approximates a threshold activation function. Fortunately, for FPGA ANN designs, no additional circuitry is required to implement this sigmoid activation function. Because of the nature of stochastic addition, the OR gate of a stochastic artificial neuron circuit does more than

merely add the neuron's weighted inputs. As the number of inputs increase, an OR gate actually generates a non-linear output that approximates a sigmoid activation function. Figure 15 shows the output bit-stream value that results from different summed values. When there are only a few neuron inputs, the sum and the OR-gate output are nearly identical. As the number of inputs increase, the properties of the sigmoid activation function become more predominant. Therefore, a stochastic artificial neuron circuit does not require additional circuitry to implement a pseudo-sigmoid activation function.

Although stochastic computing techniques require much less space than digital circuits using multi-bit digital multipliers, the numerous bit-stream generators employed still require a significant amount of design space. Stephen Bade's improvement to this field was the combination of weight bit-stream generation and stochastic multiplication. Using an innovative approach, Bade was able to combine the functions of bit-stream multiplication and weight bit-stream generation into a single three-bit lookup table.

To generate a weight bit-stream, Bade used an 8-bit lookup table with select lines A_0, A_1, and A_2, as shown in Figure 16. The A_0, A_1, and A_2 address lines are each connected to stochastic bit-streams. Consequently, the output weight bit-stream consists of the stream of bits selected by the ever fluctuating A_0, A_1, and A_2 address lines. If each bit in the lookup table has an equal probability of being addressed, then eight possible weights can be generated by filling the lookup table with more or fewer "1" values. For example, if $(b_0 , \ldots , b_7) = (00000000)$ then the unipolar output weight bit-stream, W, is obviously going to be zero. If $(b_0 , \ldots , b_7) = (10000000)$ or $(b_0 , \ldots , b_7) = (01000000)$ or $(b_0 , \ldots , b_7) = (00100000)$, etc., then the output weight bit-stream, W, is going to be 1/8.

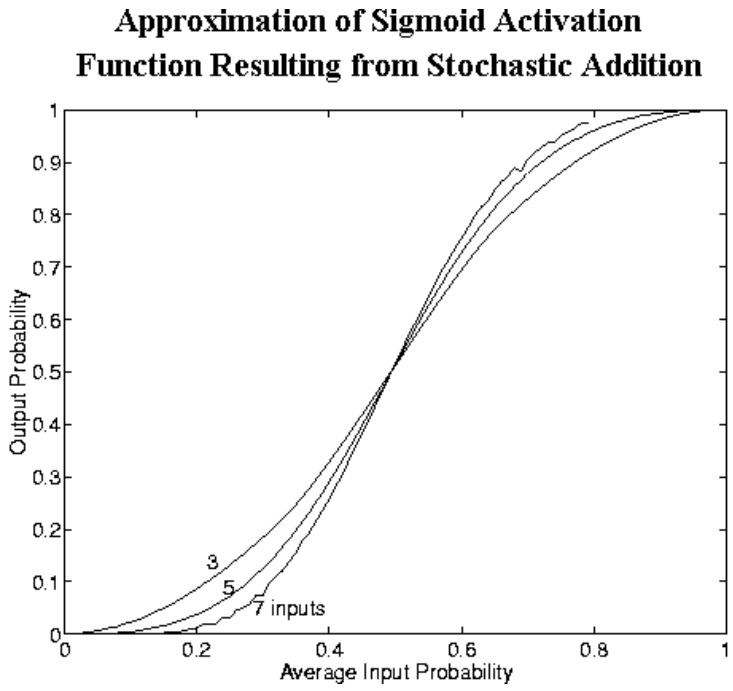


Figure 15. Activation Function Resulting from OR-Gate Stochastic Addition

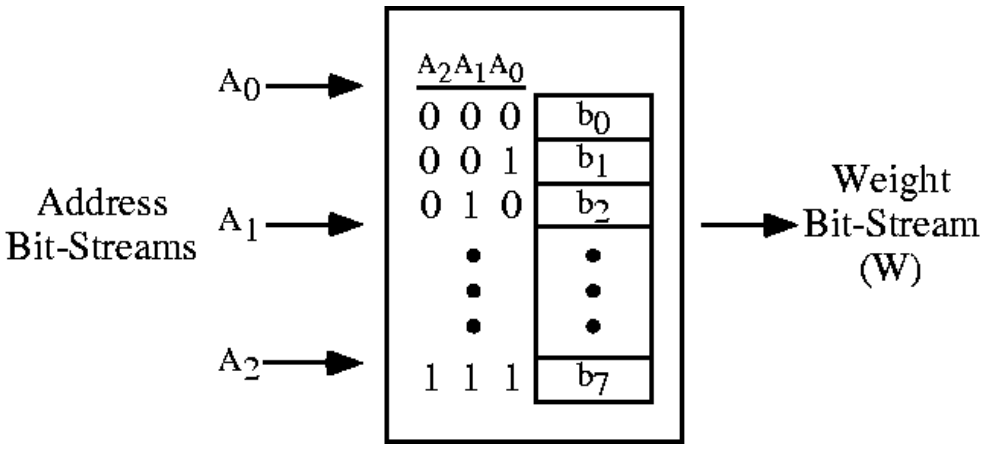


Figure 16. LookUp Table for Generating Weight Bit-Stream

However, if the address bit-streams are chosen such that A_0 is connected to a bit-stream representing the "special" value 0.668, and A_1 to the value 0.802, and A_2 to the value 0.944, then the probability of the table's n-th bit being addressed is given by Equation 8.

$$P = \frac{2^n}{2^8 - 1} \qquad \text{(Equation 8)}$$

Under these special conditions, a lookup table containing the values (b_0 , … , b_7) will produce a weight bit-stream that actually represents the binary value $(b_0b_1b_2b_3b_4b_5b_6b_7)_2$. For example, lookup values (b_0 , … , b_7) = (00000000) results in a unipolar output weight bit-stream of 0, (b_0 , … , b_7) = (10000000) results in a unipolar output weight bit-stream of 1/255, and (b_0 , … , b_7) = (01000000) results in a unipolar output weight bit-stream of 2/255, etc. Using this technique, Bade was able to realize the generation of 256 unique weights values using a compact notation that locally stored the weight values within the 3-bit lookup table and did not require the use of the traditional weight bit-stream generators.

Bade took the process one step further by extending the 3-bit lookup table to 4-bits, with the additional eight table bits populated with zeros. The select lines were then the I, A_2, A_1, A_0 address lines, where A_2, A_1, A_0 were as before and the address line I was now an input bit-stream to be multiplied. This eliminated the need for an AND gate to perform the multiplication of an input signal with a connection weight. It was particularly well suited to many FPGA architectures, where 4-bit lookup tables are

present in every logic element. Therefore, using a 4-bit LUT as opposed to a 3-bit LUT did not actually require any additional FPGA resources.

Using this lookup table technique, the space requirement of a stochastic, digital, artificial neural network design can be significantly reduced. Address bit-stream generators and LFSR circuits are only required once in a single design, so the percent resources devoted to these components is minimal. It should be noted that, to avoid correlation errors, Bade did utilize in-line delays of the address bit-streams and carrier signals at every weight generator. However, even with these additions, Bade's design improvements result in an incredibly efficient method for implementing ANNs in digital logic.

2.4f Re-configurable ANNs using FPGAs – Design-Time vs. Run-Time

Bades' research permitted ANNs to be efficiently implemented in digital logic, but it did little to address how those ANNs could be re-configured (a property necessary for evolutionary design of ANNs). On this problem, there appear to be two solutions: Run-Time Reconfiguration and Design-Time Reconfiguration.

Design-time reconfiguration involves designing an ANN with the proper connections and weights and then implementing that design in FPGA hardware. If the architecture of the network should need to be re-configured, the ANN must be redesigned from scratch and reloaded into the FPGA. Obviously, this method eliminates the need for additional logic to permit external reprogramming of connection weights and makes for a very efficient use of the device resources. A device resource is not utilized unless it plays an active role in the operation of the ANN at design time (e.g. connections between neurons

are only implemented if they have a non-zero weight). A design-time reconfiguration ANN would be capable of realizing any possible network architecture and would be capable of implementing ANNs with large numbers of neurons. Unfortunately, reconfiguring this type of circuit would require the design to be regenerated and recompiled. This means that the high-level language describing the ANN would have to be re-generated, re-synthesized, and placed-and-routed again. As mentioned in Section 2.4a, placement and routing can be a very time-consuming process due to the highly computational nature of the algorithms. Therefore, a design-time re-configurable ANN could require a significant amount of time, depending on processor speed, to evolve a population of designs.

The alternative to design-time reconfiguration is run-time reconfiguration. In a run-time re-configurable ANN, any and all possible network connections are physically created when implementing an ANN network. Even if a network connection is not initially utilized, the logic for that connection must exist in case that connection should eventually be needed. Therefore, if a network is going to contain ten neurons, the run-time re-configurable ANN must contain a hundred connections, each with its own programmable weight value. To allow the individual weight values to be programmed during run-time, the network's LUTs, which contain the weight values as described in the previous section, must be externally addressable and re-writable. This requires a significant amount of additional logic, required for communication and control, to be added to the stochastic ANN design presented in Section 2.4e. However, once complete, a run-time re-configurable ANN could realize an artificial network of any architecture and could be re-configured with considerable speed. Consequently, the amount of time

required to evaluate a generation of designs would be minimal. Unfortunately, the additional logic required by the run-time re-configurable ANN would severely limit the size ANN that could be implemented.

The trade-offs between design-time and run-time re-configuration make it difficult to identify one approach as being superior to the other. In fact, which approach works best for a given application depends on the computational hardware and the FPGA resources available. Therefore, this dissertation presents a re-configurable ANN design platform that actually has two separate embodiments: a design-time re-configurable and a run-time re-configurable embodiment. An overview of the general system is provided in the following chapter.

3.0 A Hardware-Evolved Digital ANN Design Platform: An Overview of the HEDANN Design Platform

3.1 Introduction

In the previous chapter, the fundamental concepts of ANN, GA, and EHW theory were presented. The failure of past efforts to develop powerful artificial intelligence systems, using each of these approaches, was also discussed. In tandem, an argument was being formulated to suggest that the limitations of each of these techniques are interconnected. Specifically, the previous chapter asserted that ANN researchers have been unsuccessful in developing intelligent processors because they have yet to adequately explore ANNs with complex architecture. Although complex architecture ANNs are challenging to investigate, it is possible using GAs and indirect encoding. GAs, however, cannot be effectively utilized without a flexible and accurate tool for evaluating the performance of the encoded ANN designs. Although evolvable hardware (FPGAs) could be used to accurately evaluate the performance of a complex architecture ANN design, the EHW community is currently focused on its own problems related to the unwanted evolution of environmentally sensitive circuits based on primitive logic components. Ironically, many of the problems currently encountered with EHW could potentially be eliminated by evolving higher-level components such as artificial neurons.

Therefore, in this chapter, a new design methodology is presented which attempts to combine aspects of ANN, GA, and EHW research into a single approach that captures the benefits of each technique while eliminating the limitations of each. This approach is

based on the hypothesis that Nature requires three "essential components" to successfully evolve intelligent processors. All three of these essential components must be recreated if an evolutionary technique is to create a synergistic environment suitable for developing powerful AI systems. The author argues that these three "essential components" include:

- **The Neuron** - In Nature, the neuron is the fundamental building block of all biological intelligence on earth.
- **Genetic Encoding** – In Nature, complex indirect encoding of genetic information allows high-level traits/features of an individual to be efficiently transferred from one generation to the next. Successful genetic encoding and genetic operators play a vital role in Nature's efficient creation of new populations from past generations.
- **Evolution in the "Physical-World"** – In Nature, populations are evolved using Darwinian principles of evolution, executed in, and constrained only by, the physical world.

Interestingly, many past ANN, GA, and EHW techniques have included as many as two of these essential components. However, never have all three components been combined into a single synergistic system. This chapter will describe an evolutionary design platform to do just that.

The Hardware-Evolved Digital ANN (HEDANN) design platform is a circuit design platform built to evolve complex architecture ANN circuits using FPGA hardware. The

development and evaluation of this unique design system is the primary contribution and focus of this dissertation. By using genetic algorithms to evolve complex-architecture ANN designs in FPGA hardware, this system is, as far as the author knows, the first AI design system to explore and evolve physical ANN circuits with unconstrained network architectures. As such, it is the first system to incorporate all three of the "essential components" of natural design stated previously. With the HEDANN design system, the development and evaluation of ANNs with recursive, non-layered, complex architectural connections is made possible. Because of the inherent digital nature of the ANN circuitry, the HEDANN design system is also one of the few evolvable hardware design methodologies to produce device-independent circuit designs capable of operating properly throughout the environmental ranges typically required of digital circuits. In this chapter, the components of the HEDAAN design system are presented along with a description of the control and flow of data through the system.

3.2 Overview of the HEDANN Design Platform

The basic hardware and software components of the HEDANN design platform are shown in Figure 17. This system consists of a desktop computer running a genetic algorithm that serves as the central controller for the system. The system's genetic algorithm creates multiple ANN circuit designs that are individually downloaded to an FPGA development board, where the circuits are realized in hardware and tested for fitness. The FPGA-implemented designs are exposed to various test inputs and their resulting outputs are monitored through a computer-based data acquisition system.

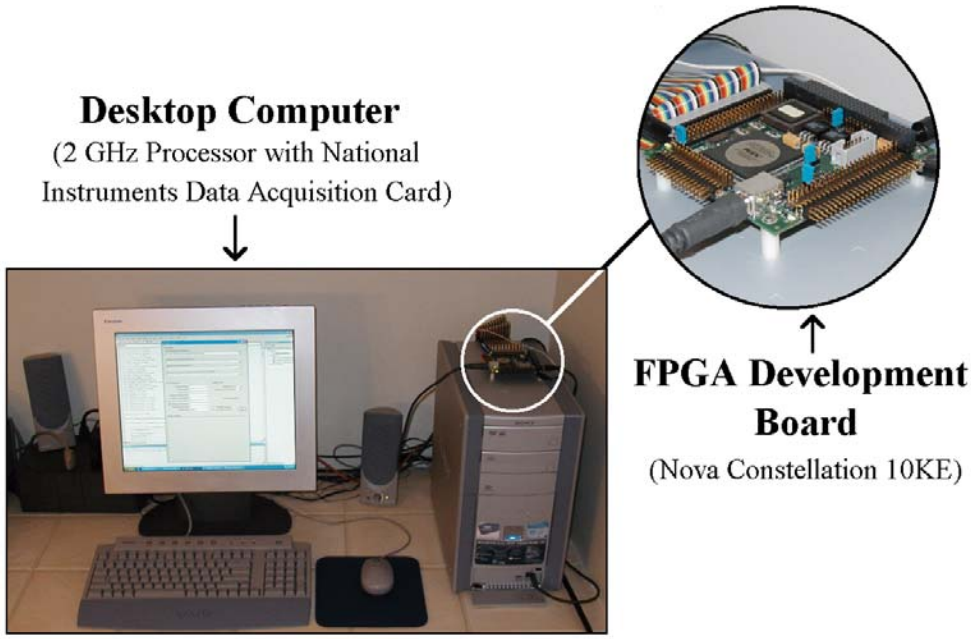


Figure 17. Hardware Evolved Digital ANN System (Design-Time Re-configurable)

Based on this fitness information, the GA constructs a new generation of ANN circuits through cloning and mutation. This next generation, and future generations, of circuit designs are exposed to the same procedures until a suitable solution is found.

To describe the HEDANN design platform in more detail, the major hardware and software components of the system can be further reduced to four sub-systems: the ANN circuit design software, the Genetic Algorithm software, the evolvable FPGA hardware, and the fitness evaluation hardware.

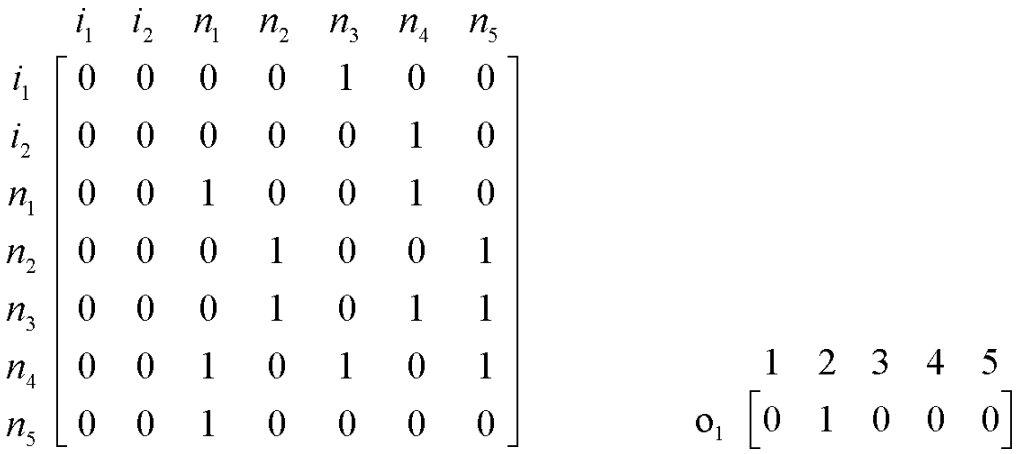
3.3 The ANN Circuit Design Software

The HEDANN design platform aims to evolve complex-architecture ANNs in FPGA *hardware*. Therefore, the system must have software that converts theoretical ANN

designs (usually contained in arrays) into digital circuits that can be downloaded into an FPGA. The first step in accomplishing this task is to establish a standard format by which to represent all theoretical ANN designs.

For the research described in this dissertation, any neural network design can be described as consisting of primary inputs (denoted as i_n), primary outputs (o_n), and neurons (n_n). A primary I/O is an input or output that connects to the outside world (e.g. connects to an I/O pin on the FPGA). Whereas, local I/O are inputs and outputs that connect within the FPGA itself (i.e. from one neuron to another). The connections between all inputs, outputs, and neurons can be fully described by a weight matrix and output matrix, an example of which is shown in Figure 18. The weight matrix is a square matrix of dimension N, where N is the total number of neurons plus the total number of inputs to the network. The elements of the weight matrix can be any value between 0 and 1. Although the weight matrix can be constructed to have both positive and negative weight values, for circuit simplicity, only positive value weights were used in this dissertation. As was discussed in Chapter 2, any non-zero value represents a weighted connection between the column index neuron and the row index neuron, where the data flow is from column index neuron to row index neuron. A 0 value represents no connection.

The output matrix, shown in Figure 18b, identifies which local node outputs correspond to primary network outputs. Unlike the weight matrix, the elements of the output matrix can be one of only two values, either a 0 or a 1. A "1" value indicates that the row index primary output is connected to the column index local neuron output.



a) Weight matrix **b) Output Matrix**

Figure 18. Matrix Representation of an ANN Design with (a) Weight Matrix and (b) Output Matrix

A "0" value represents no connection. For the simple network represented in Figure 18, there is only one output, corresponding to the output from neuron 2.

To construct ANN designs that can be physically implemented in an FPGA, certain rules must be followed to generate viable weight and output matrices. These rules are as follows:

- No primary input can be connected to another primary input (including itself).
- No primary output can be connected to another primary output (including itself).
- A primary input cannot be connected directly to a primary output, and vice-versa.
- A primary output can be connected to one, and only one, neuron output.
- There must exist at least one input, one output, and one neuron in a design.
- At least one primary output must be connected indirectly to one primary input.

The HEDANN software follows these rules whenever creating or evolving a population of theoretical ANN designs. Any matrix that follows all six of these rules is permitted to exist, even if it possesses network sub-structures that are non-influential (i.e. structure that have no direct or indirect connection to the output). Once a design's matrices (weight and output matrices) have been constructed, they are converted into a circuit design, expressed in the FPGA hardware description language VHDL. An algorithm was written to automatically construct the VHDL code needed to describe a stochastic ANN circuit corresponding to a given weight and output matrix. The high-level VHDL code, along with appropriate I/O pin assignments for a given FPGA, are converted by commercial FPGA development tools (Altera's Quartus II) into lower-level, device-dependent FPGA programming files. These programming files can then be downloaded to an FPGA to physically realize the theoretical ANN design. Custom-written Visual Basic code was used to automatically create the described VHDL code based on the circuit designs to be presented in Chapter 4. Although the stochastic neuron model has been used to construct the physical ANN circuits presented in this dissertation, it should be noted that the HEDANN system is flexible enough to incorporate alternative neuron models that could eventually be incorporated into the system. This would simply be accomplished by re-writing the VHDL code generating algorithm to reflect the newly desired ANN circuitry.

Depending on how the HEDANN design platform is configured (i.e. run-time versus design-time), the FPGA implemented design will either be immediately ready to test or it will require some additional configuration data before being ready to test. The details of

this additional configuration data will be discussed later in Chapter 6. The metamorphosis of the ANN design is shown graphically in Figure 19.

As intended, the HEDANN software allows the creation of ANNs with extremely complex architectures. Other than I/O pin requirements, very few limits have been placed on the design of the ANN's network architecture. Because of the extreme architectural flexibility, an intelligible method for visualizing the complex networks was needed to aid in monitoring the circuits' evolution. Figure 20, shows a graphical scheme that was developed to represent an individual neuron within a complex ANN network. Under this graphical scheme, input connections to the node are always made to the neuron's left side port. Outputs connections from the neuron are always made from the right side port. Primary network connections, such as primary inputs or outputs, are identified by a green solid-fill of either the input or output port. Using this strategy, a typical network might look as shown in Figure 21. Notice that the connection lines between inputs, neurons, and outputs vary in intensity. These variations represent the weights of the connections and provide a qualitative description for the user (the darker the connection line, the higher the weight value). In addition, the placement of the nodes in Figure 21 has been arranged in a grid fashion, even though the neuron position is in fact arbitrary. Since there are no "layers" in a complex architecture ANN, it would be meaningless to try and impose some sort of systematic positioning scheme. Therefore, the only other placement rules (in addition to the grid-like structure) concern the location of the primary inputs and outputs. Notice that the primary inputs all lie to the left of the network in Figure 21. Similarly, all of the primary outputs lie to the right of the network in Figure 21.

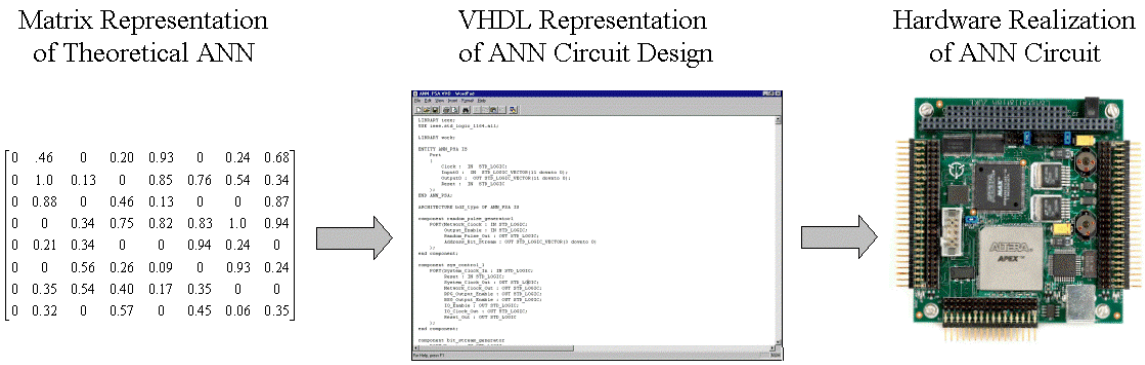


Figure 19. Conversion of Theoretical ANN Design into FPGA Circuitry

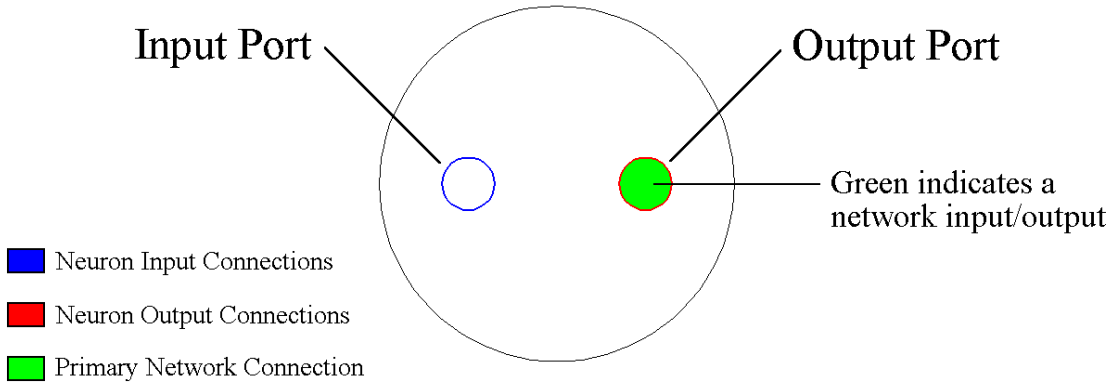


Figure 20. Neuron Color Convention Used by the HEDANN Design Platform

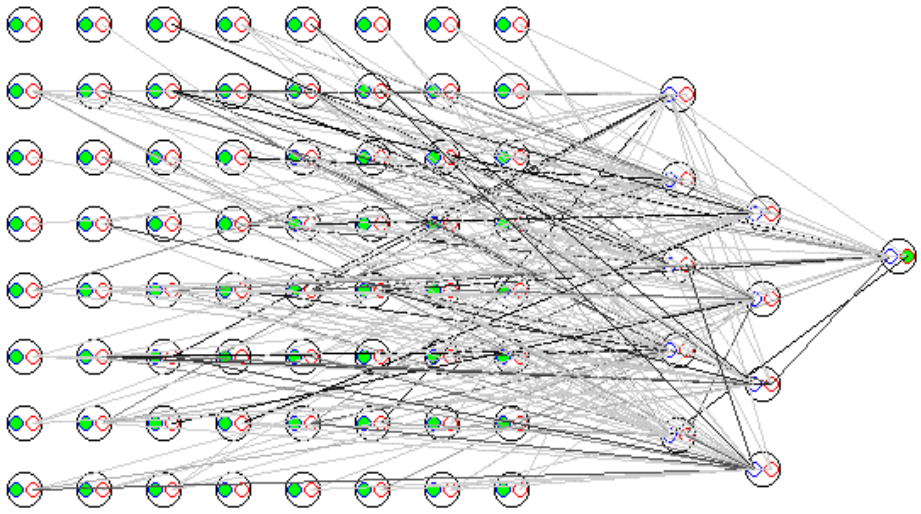


Figure 21. HEDANN Graphical Display of ANN Design with 64 Inputs, 10 Neurons, and 1 Output

Given this graphical scheme, a complex ANN's connections can be displayed representing connection weights, direction of data flow (i.e. from neuron a's output to neuron b's input), recursive connections, etc. Although this graphical data provides some useful qualitative information, and will be used later in this dissertation to depict the change in evolved ANN architecture, a thorough and complete understanding of the ANN can only be discerned through a careful evaluation of the ANN's weight matrix and output matrix.

3.4 Genetic Algorithm Software

Ideally, the HEDANN design system should evolve ANN designs using a genetic algorithm with an indirect encoding scheme well suited to handling the generalities of

complex architecture ANNs. Unfortunately, as was discussed in Chapter 2, no such genetic algorithm currently exists. Therefore, the traditional, non-efficient, genetic algorithms associated with direct encoding have been used with the HEDANN system, but modified, to accommodate the unique requirements of complex architecture ANN designs. These modifications include the exclusive use of asexual operators, which will be described in greater detail in this section.

The HEDANN's genetic algorithm software directly encodes ANN designs as weight matrices and output matrices. Since very few limitations are placed on the architecture of the ANN designs, an evolving population will likely contain ANNs with varying numbers of neurons. Consequently, the dimensions of one ANN's weight matrix might be 10 x 10, while another ANN's weight matrix, in the same population, could be 3 x 3. Unfortunately, with a direct encoding scheme, it is not clear that a crossover operator (sexual operator) is available that would allow these two designs to be bred and result in an intelligible offspring. Therefore, the production of a new generation of designs from existing designs was accomplished, in this dissertation, through purely asexual operators. The following list identifies the asexual operators used by the platform's GA:

Cloning – An existing design is copied flawlessly into a new population

Weight Mutation – A connection's weight value is modified.

Connection Mutation – A connection within the network is reversed (if it exists, it is eliminated / if it is does not exist, it is added)

Expansion – A neuron is added to an existing design

Contraction – A neuron is subtracted from an existing design.

With each of these operators, conditional statements were included in the software to ensure that modified ANN designs did not violate the rules described in Section 3.3.

Equipped with these asexual operators, the HEDANN design platform's genetic algorithm has the necessary, albeit inefficient, tools needed to begin an evolutionary run. To initiate and monitor that process, the software provides a user interface that allows key design parameters to be entered and monitored. Figure 22 shows a screenshot of the HEDANN design platform's graphical user interface (GUI).

This GUI allows the user to select the total number of designs in a population (e.g. the population size) as well as the directory in which to store the current generation's design files, the last generation's design files, and the top ten performing design files from each generation. Additional settings are available which help the user to configure and monitor the GA's progress, but a description of these settings is non-essential to understanding the operation of the system.

As with any GA, the population size selected will directly affect the amount of time required to evaluate one generation. The larger the population, the longer the evaluation time per generation and, consequently, the fewer number of generations which can be evaluated in a day. Choosing a smaller population size will increase the number of generations that can be created and evaluated in a day, but this may not necessarily translate into a faster evolutionary process. If a population is too small, the initial designs created may lack any interesting or evolvable qualities. Depending on the nature of the problem, it may take a considerably number of generations to evolve these designs into suitable solutions.

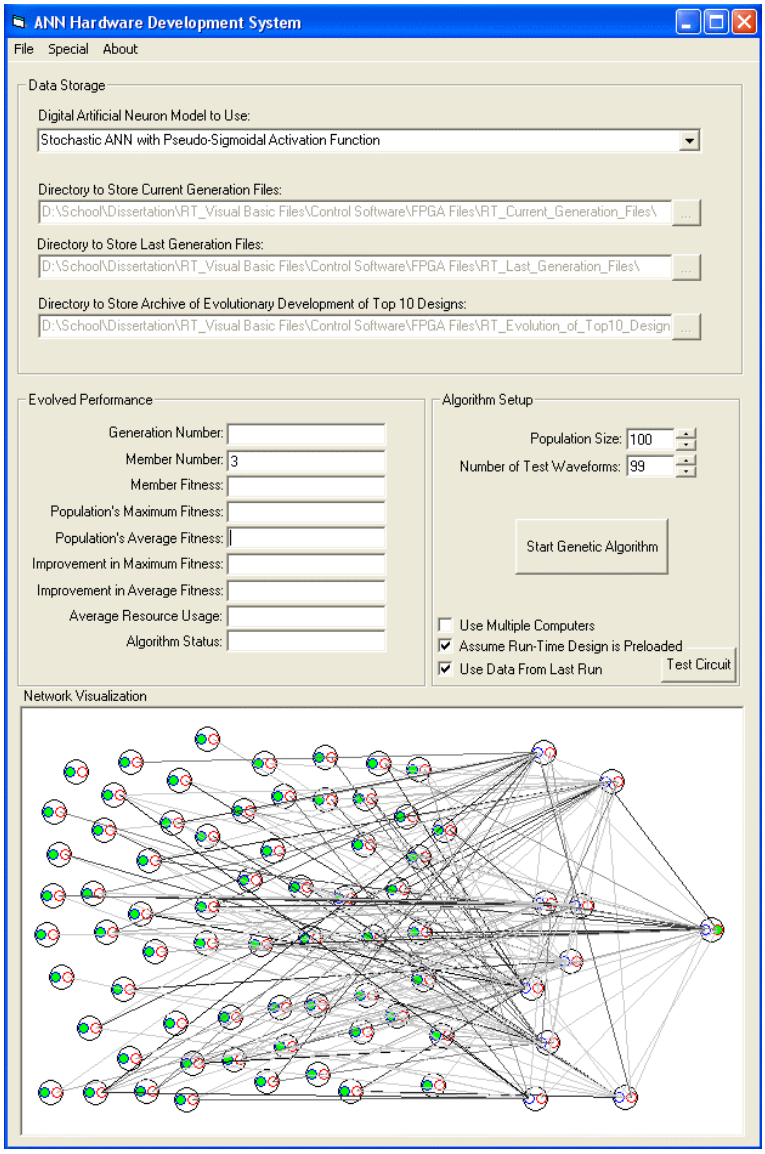
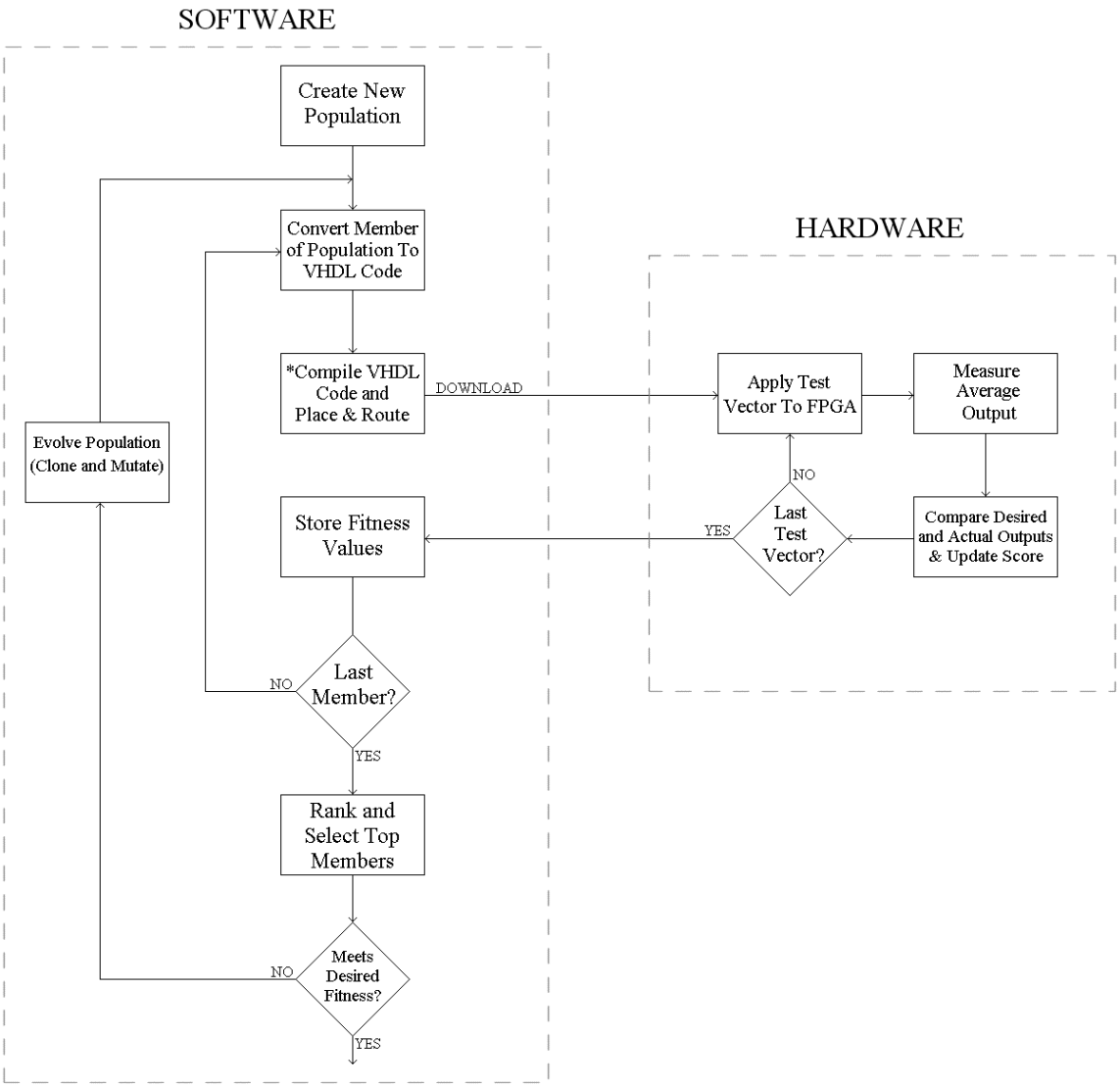


Figure 22. HEDANN Graphical User Interface

An extremely large population is more likely to generate better initial designs that can be evolved into suitable solutions within a relatively few number of generations. However, one generation of such a large population will take a considerable amount of time to evaluate.

To balance the advantages and disadvantages of population size, the HEDANN design system actually uses an expanded population size for its first two generations. For the first two generations only, the user selected population size is multiplied by a factor of ten. Therefore, if a user were to select a population of 100, the first and second generations would actually have 1000 members. For the third generation and after, the population would be reduced to 100. Because of the time limitations imposed by hardware evaluation (to be discussed further in Chapter 7), this procedure is critical to the success of the HEDANN design platform.

With an initial population created, the HEDANN design platform's GA converts each member of the population into a programming file that can be downloaded into an FPGA. Once downloaded to the FPGA, the circuit is evaluated for a set of known inputs. The response of the circuit to these known inputs is compared against a desired response. Through a user defined fitness function, how closely the circuit's actual response matches the desired response is measured and recorded as the design's fitness value (where a higher fitness value indicates that the actual response is close to the desired response). Following the evaluation of all members of a population, designs are ranked according to fitness value. The highest-ranking 10% of the designs are cloned to create a new generation of solutions (this complete process is illustrated graphically in Figure 23).



*The compile step is not required for the HEDANN Design Platform using a run-time reconfigurable ANN circuit.

Figure 23. Flowchart Describing HEDANN Genetic Algorithm

Weight mutation, connectivity mutation, expansion, and contraction operations are performed on the cloned designs from a previous generation. The mutation operations are performed randomly to create the remaining 90% of a new population. Once a new population has been generated, the evolutionary process continues until a suitable solution is eventually found.

3.5 Evolvable FPGA Hardware

As described in Section 3.3, the HEDANN design platform creates ANN design files that are converted into device-dependent programming files for eventual downloading into an FPGA device. Depending on the complexity of the ANN design, there may be various FPGA device families that are suitable for implementing a particular design. For practical reasons, the HEDANN design system presented in this dissertation uses only a single type of FPGA device for all design files in a population. Since very complicated designs might utilize nearly all of the resources of the FPGA device, while a less complicated design in the same population might use only 1% of the FPGA resources, a device was chosen which provided the resources required to implement the largest ANN design expected.

It should be noted that choosing merely the largest FPGA available is not necessarily recommended. Quite to the contrary, the FPGA chosen for use with the HEDANN system should be only as dense as absolutely necessary. Doing so, especially for the design-time re-configurable approach, ensures that ANN designs will be placed-and-routed as quickly as possible into the targeted FPGA.

Because of the differences between a design-time and a run-time re-configurable approach, two different FPGA development boards were used to conduct the research presented in this dissertation. Fortunately, a number of FPGA development boards are available from various FPGA manufacturers featuring a variety of device families. These development boards offer various I/O configurations and peripheral chip support (such as D/A converters, memory, Ethernet converters, etc.), allowing a variety of signals to be easily connected to a test board.

The following two prototype boards were selected for use in this research:

1. Constellation 10KE Prototype Board from Nova Engineering
2. APEX DSP development Kit (Professional Version) from Altera

The Constellation 10KE Development board, shown in Figure 24, was used primarily with the design-time re-configurable system described later in Chapter 5. This board utilizes the Altera Flex10K FPGA (EPF10K200SRC240). The Constellation Development board provides a 40 MHz clock, easy access to roughly 132 FPGA I/O pins, and allows high-speed programming of the FPGA via a USB port. The 10K chip offers approximately 9,984 logic elements, or roughly 200,000 gates.

The APEX DSP Development board, shown in Figure 25, was used primarily with the run-time re-configurable system described later in Chapter 6. This board utilizes the Altera APEX20K FPGA (EP20K1500EBC652-1X). The APEX DSP development board provides a 40 MHz clock with easy access to over 200 FPGA I/O pins.

Figure 24. Nova Constellation 10KE Development Board

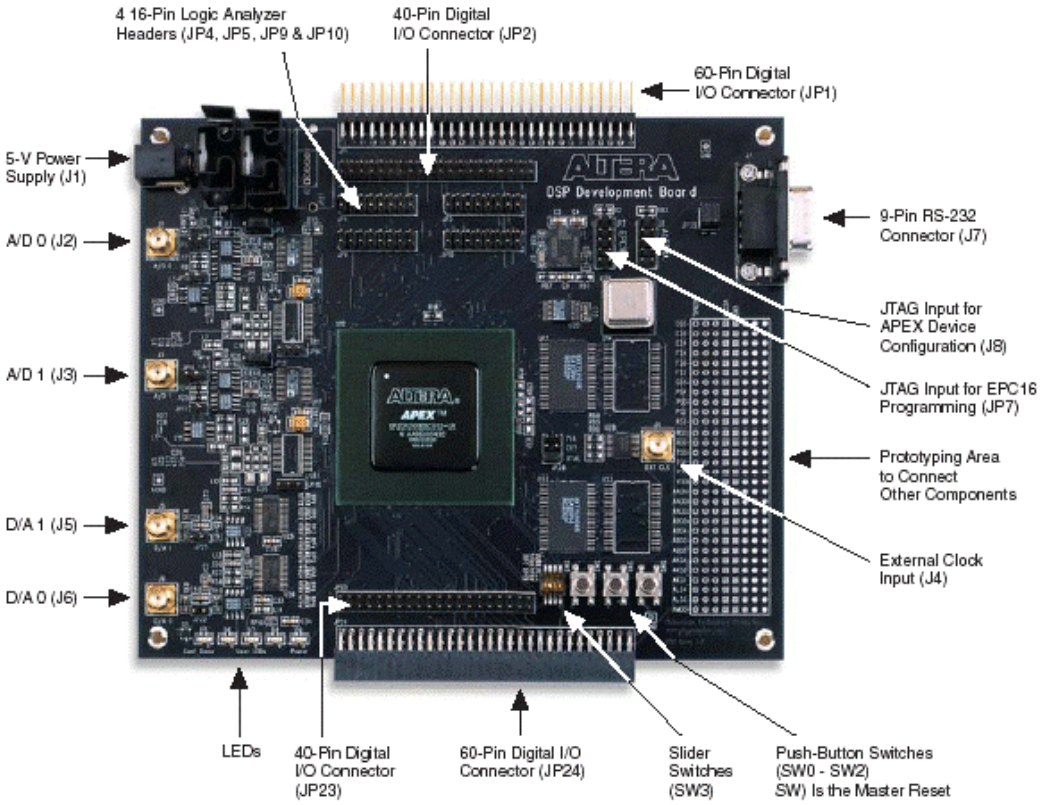


Figure 25. Altera APEX DSP Development Board

In addition, it allows fixed programming of the FPGA via an EPC storage chip programmed via a parallel port connection. The APEX chip offers approximately 51,840 logic elements (1.5 million gates), and is one of Altera's highest density FPGAs.

3.6 Fitness Evaluation Hardware

To evaluate the fitness of an ANN design realized on an FPGA, test signals are sent to the FPGA development board and the FPGA's response measured. The hardware required to generate these test signals will vary considerably depending on the type of circuit being designed. For the experiments described in this document, a multifunction Data Acquisition board, manufactured by National Instruments, provided a flexible platform for generating and recording the various test signals. The NI DAQCard-1200 board is equipped with 24 digital I/O lines, two 12-bit analog outputs, and eight 12-bit analog input channels (20KHz sample rate). This board provides access to analog and digital I/O through a 50-pin header, is easily programmed by third party software, and is relatively inexpensive. To connect the DAQCard-1200's 50-pin connector to the appropriate pins of the Nova Constellation and APEX DSP Development Board, two PCB boards were fabricated.

Prior to initiation of the GA, a test vector table must be generated by the user and stored in a program directory. The test vector provides the inputs and outputs desired of the circuit under test. This data is used by the GA program in evaluating the fitness of all implemented circuits and is the responsibility of the user.

This general description of the HEDANN design platform provides a good overview of the system's basic functions. However, the specific details of the circuitry required to

create a design-time or run-time re-configurable circuit would be valuable to anyone wishing to recreate this approach. Therefore, the following chapter provides an in-depth description of the circuitry involved with each design approach.

4.0 Re-Configurable ANN Circuit Designs

4.1 Introduction

As mentioned in previous chapters, the goal of the HEDANN design system is to evolve complex architecture ANNs that can solve challenging AI problems. Depending on the AI problem being solved, the best solution may require an ANN with hundreds to thousands of neurons and may require many thousands of generations to evolve. Therefore, it is essential that any ANN circuitry developed make efficient use of the FPGA resources available, such that ANNs with large numbers of neurons can be realized. In addition, the high-speed re-configurability of the ANN circuits must be ensured to allow multiple generations of designs to be implemented and evolved quickly. With the HEDANN design platform, these two factors, the speed of reconfiguration and the maximum achievable ANN size, are, unfortunately, competing variables. Therefore, as was discussed in Chapter 3, this has led to the creation of two alternate strategies for implementing a re-configurable ANN circuit: a design-time and a run-time re-configurable ANN circuit. Both of these re-configurable strategies optimize for a different resource (i.e. speed of reconfiguration vs. network size). This chapter describes the specific circuitry associated with each approach, the resources required, and the advantages and disadvantages of each system.

4.2 Design-Time Re-Configurable ANN Circuitry

The design-time re-configurable ANN circuit consists of the following five top-level modules:

- System Controller Module
- Random Pulse Generator Module
- Bit-Stream Generator Module
- Bit-Stream Converter Module
- Neuron Module

The overall layout and interconnection of the five top-level modules is shown in Figure 26.

The System Controller Module is responsible for resetting and initializing all of the modules during power-up or reset and for controlling all timing signals. There is only one System Controller module for any given design. The schematic of this module is shown in Figure 27.

For the prototype boards targeted in this research, there is typically a 40MHz on-board crystal oscillator available for clocking. To relax the timing requirements on the ANN circuitry, the on-board clock is divided down by 32 (via a 5-bit counter) to produce a 1.25 MHz "system" clock. The 1.25 MHz system clock serves as the global clock for the circuit. Upon power up, the system clock feeds an 8-bit counter whose output feeds three comparators used for system initialization. After 100 pulses of the system clock, one comparator sends an enable signal to the Random Pulse Generator module. After 200 pulses, one comparator sends an enable signal to the Bit-Stream Generator modules.

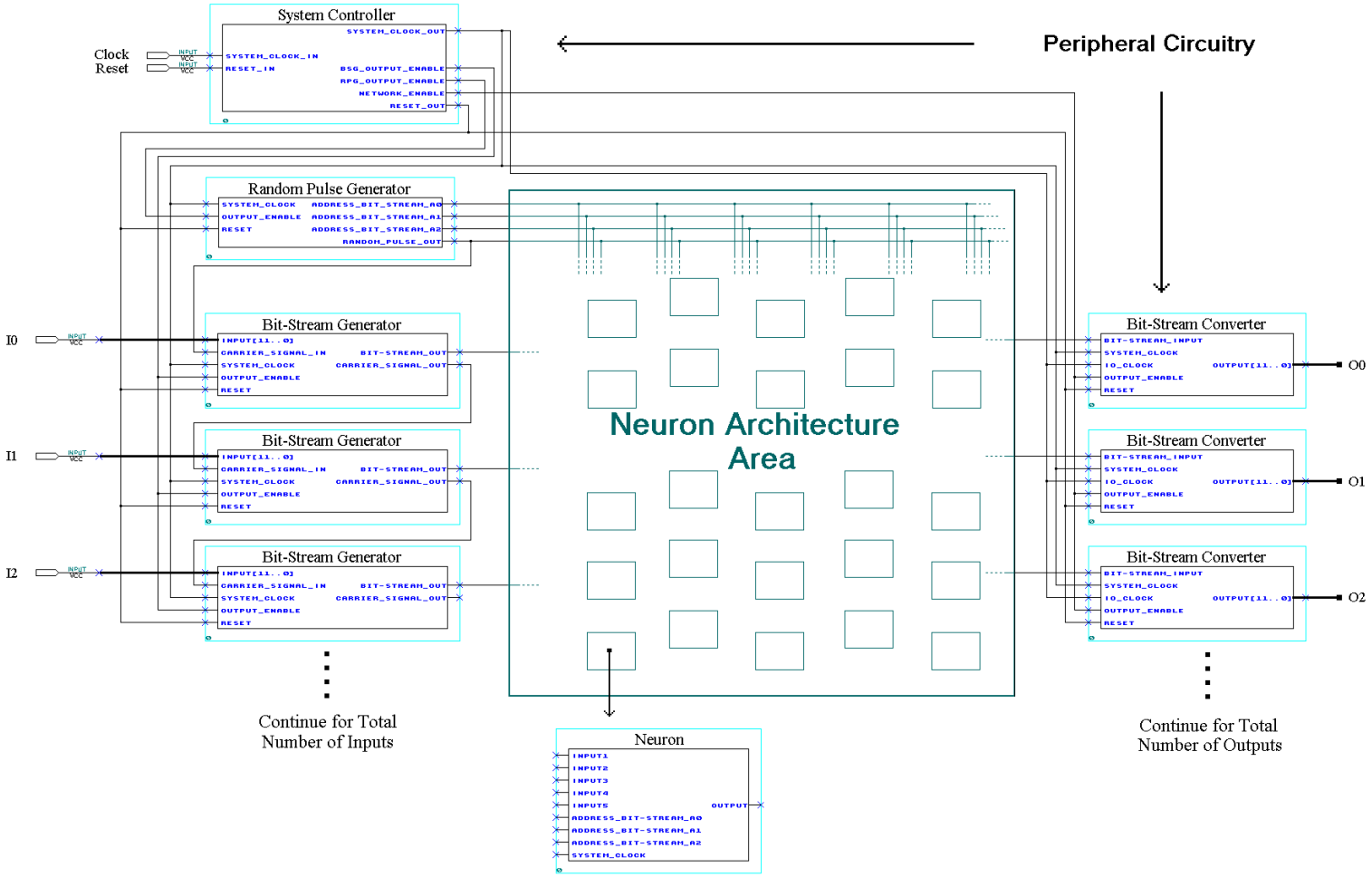


Figure 26. Top-Level Schematic of Design-Time Re-configurable ANN Circuit

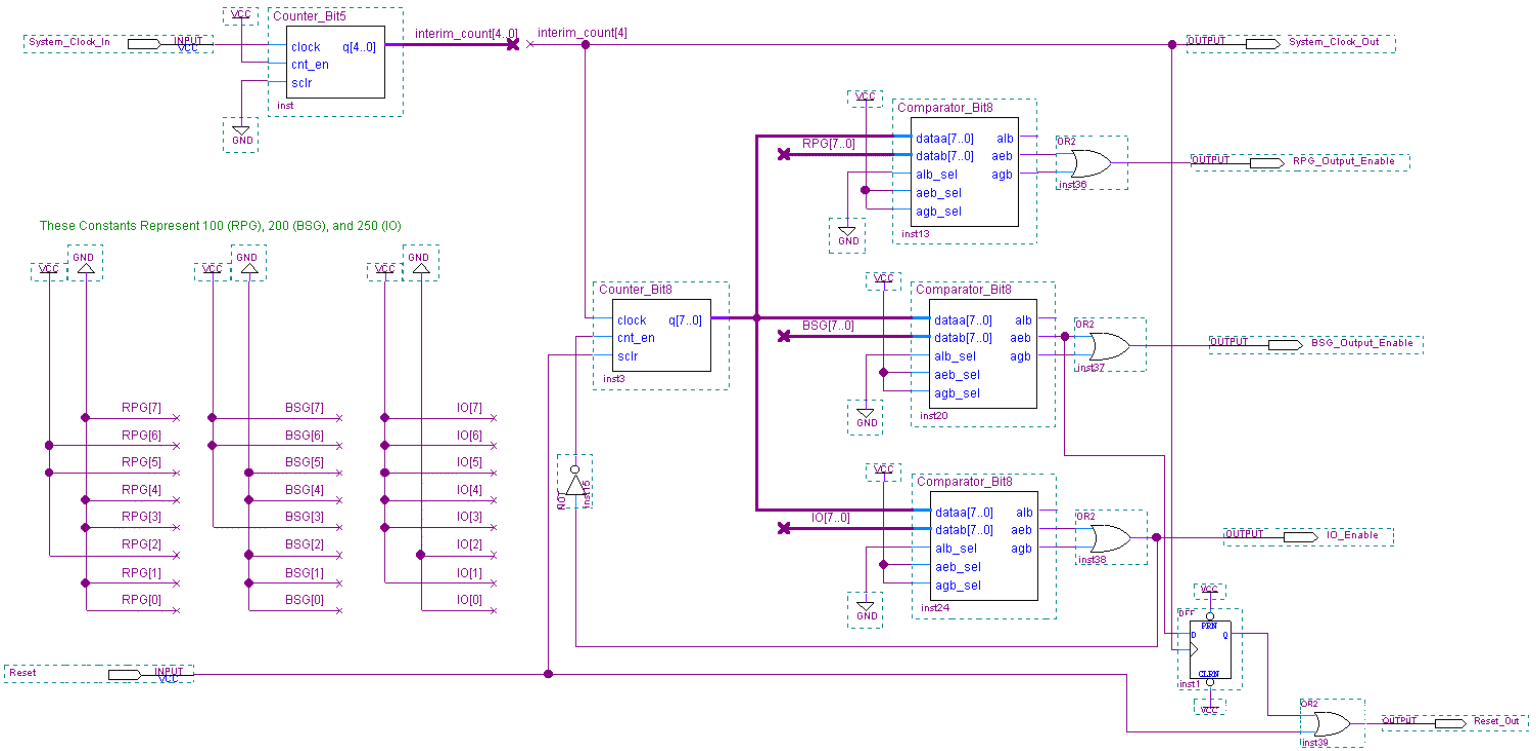


Figure 27. Schematic of System Controller Module

After 250 pulses, one comparator sends an enable signal to the Bit-Stream Converter modules. This last comparator's signal is routed back to the 8-bit counter's enable pin allowing the comparator signals to remain static (until the reset pin to the module is triggered). After 250 pulses of the system clock (200 μsec), following power-up or reset, the entire ANN circuit is initialized and ready to receive inputs.

The Random Pulse Generator Module is responsible for generating a random 4-bit address value to be used by the Neuron modules. In actuality, the output is only pseudo-random since it repeats after approximately a billion pulses. However, this level of randomness is more than suitable for the ANN designs explored in this dissertation. An overview of the Random Pulse Generator module is shown in Figure 28. The clock input triggers an LFSR_32 sub-module and four bs_generator12 sub-modules. The LFSR_32 sub-module is a 32-bit Maximal Length Linear Feedback Shift Register (LFSR) that generates a pseudo-random stream of pulses (repeat length = 4,294,967,295 pulses). The LFSR_32 sub-module (consisting mainly of D-flip-flops) is shown in Figure 29. A 2-bit counter is used to initially pre-set each flip-flop high. This ensures that the LFSR is not "stuck" at a null output. On average, the probability of receiving a high output from the LFSR_32 is 0.5.

In Figure 28, we can see that the LFSR output feeds four bs_generator12 sub-modules to produce four unique random pulse streams (note that the carrier out in the bs_generator12 sub modules is identical to the carrier in). The bs_generator12 sub-modules allow the "firing" probability of the original LFSR output (original probability = 0.5) to be altered. The details of the bs_generator12 sub-module are shown in Figure 30.

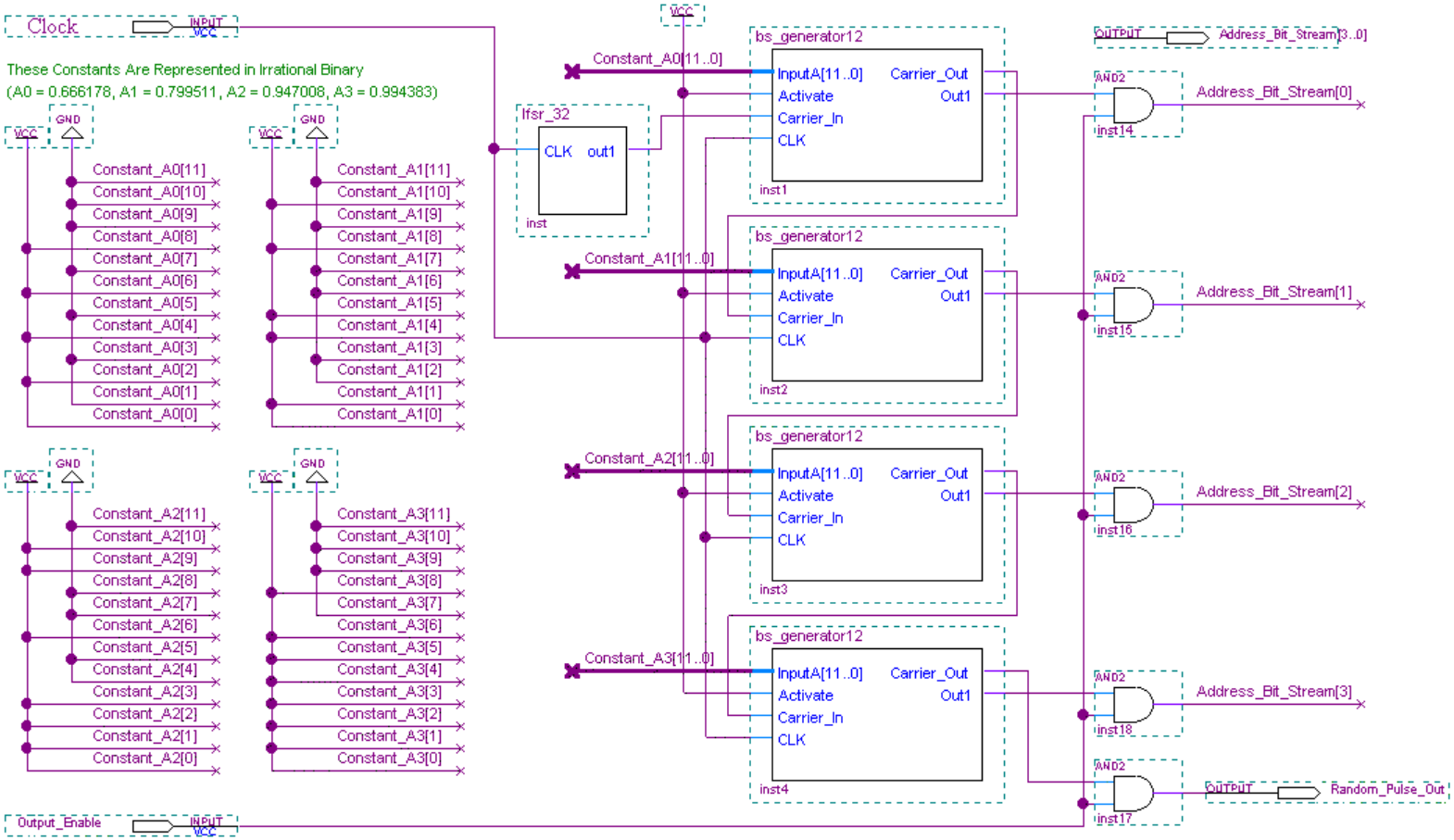


Figure 28. Schematic of Random Pulse Generator Module

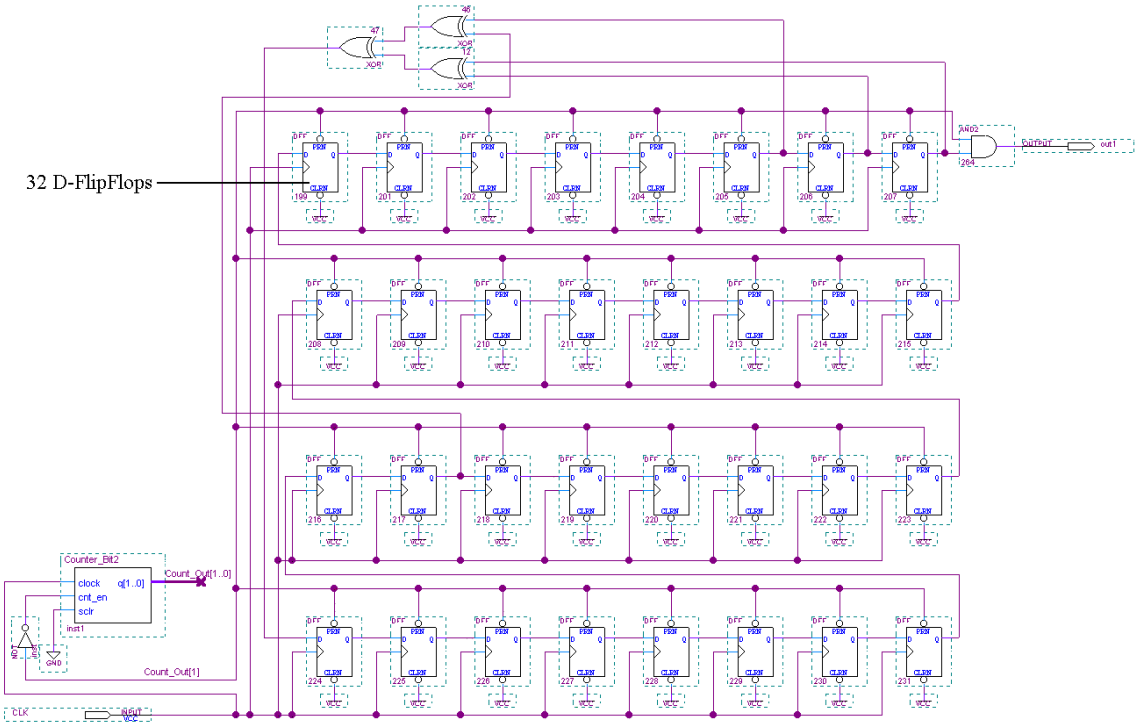


Figure 29. Schematic of "LFSR_32" Sub-Module

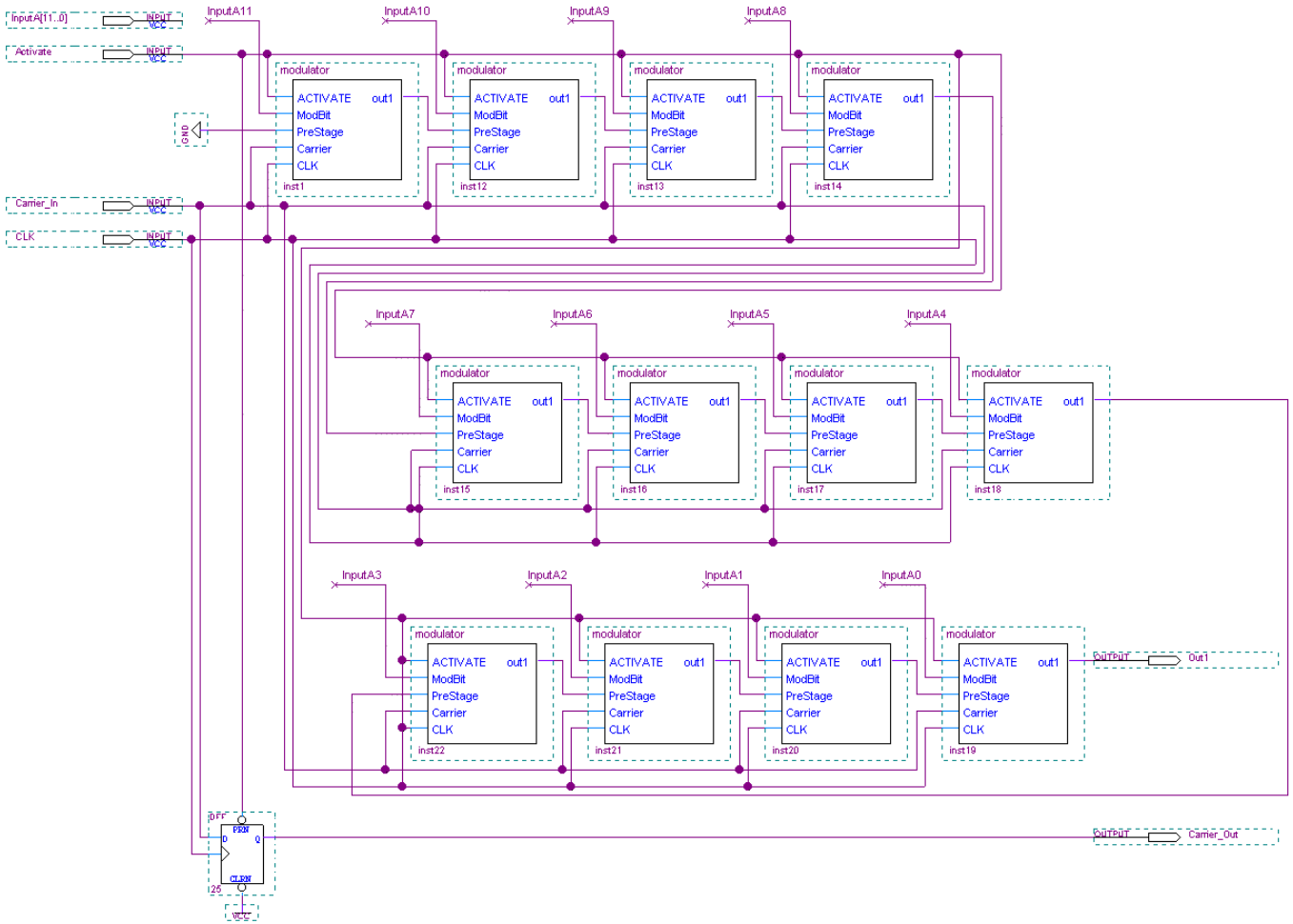


Figure 30. Schematic of "bs_generator12" Sub-Module

The bs_generator12 sub-module modifies the input firing probability using the stochastic arithmetic techniques discussed in Section 5.2. Although the theory behind this technique will not be presented again, the circuitry at the heart of the sub-module, the modulator, is shown in Figure 31. The four bs_generator12 sub-modules, contained within the Random Pulse Generator module, allow the LFSR output to be modified such that it creates four outputs having a firing probability of 0.666178, 0.799511, 0.947008, and 0.994383. These outputs are assigned the labels Address_Bit_Stream[0], Address_Bit_Stream[1], Address_Bit_Stream[2], and Address_Bit_Stream[3], respectively (notice that this is an expansion on Bade's three Address bit technique presented in Section 2.4e). The result is a randomly generated 4-bit address value that can take on any value between 0 and 15. As was discussed in Section 2.4e, when a 4-bit address value (with the proper firing probabilities) triggers a 4-bit lookup table, the output can represent a 16-bit stochastic value (0 – 65535). For an ANN circuit design, only one Random Pulse Generator module is required per design.

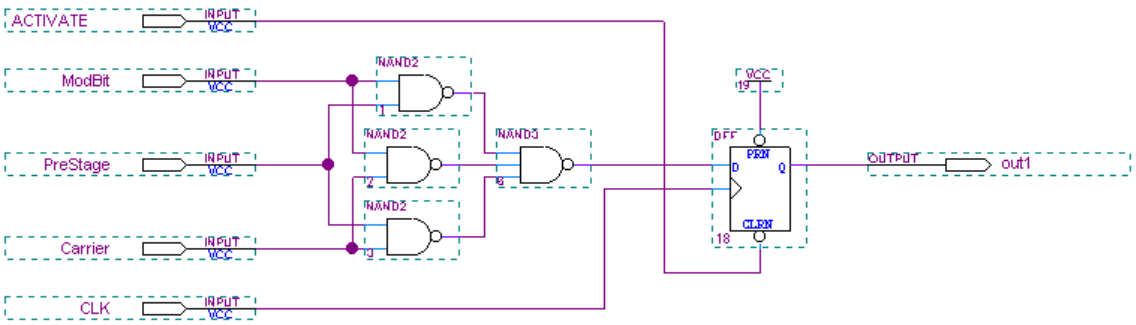


Figure 31. Schematic of "modulator" Sub-Module

The Bit-Stream Generator module is responsible for taking 12-bit input values and converting them into stochastic bit-streams. The number of Bit-Stream Generator modules required in an ANN design depends on the number of inputs (i.e. a 6-input ANN requires six Bit-Stream Generator modules). The circuitry for the Bit-Stream Generator is very similar to the bs_generator12 sub-modules shown previously in Figure 30. The only difference is that the input to the Bit-Stream Generator module is an external 12-bit pin on the FPGA chip, and these inputs are AND-ed with an "Enable" pin.

The Neuron module is responsible for multiplying chosen inputs by static weight values, summing the results, and then producing a non-linear (pseudo-sigmoid) output. These are the fundamental operations of the neuron model described in Section 2.2b. Because these operations are performed through stochastic arithmetic techniques, the circuitry for the Neuron module is actually quite small. Figure 32 illustrates the circuitry of a 3-input Neuron module.

This 3-input Neuron shown in Figure 32 consists of three lookup tables (labeled ann_rom0), some AND and OR gates, and two D-flip-flops. Each lookup table has a 4-bit Address line (which is fed by the Random Pulse Generator module discussed previously), a 16-bit weight value input, and a clock input. The 16-bit weight value fills the lookup table with data to be accessed by the address lines resulting in a final output, q, which is a stochastic representation of a 16-bit weight value. Based on the weight matrix for the ANN design, these weight values are hardwired into the circuit at design time. During operation, the resulting weight values are multiplied (stochastically) with the inputs (inputted as bit-streams from the Bit-Stream Generator modules) using AND gates.

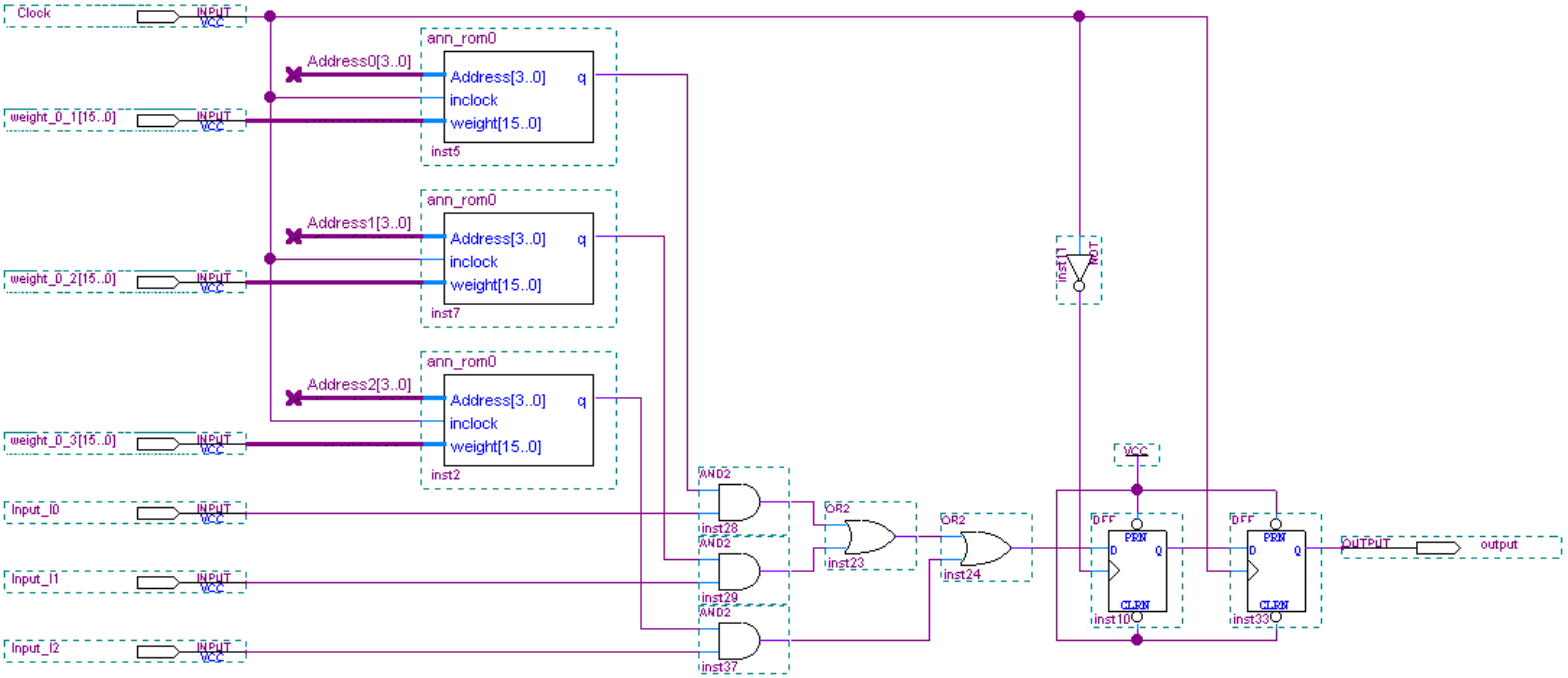


Figure 32. Schematic of "Neuron" Module (with 3 inputs)

The result of each multiplication is summed and threshold using OR gates. The output is a bit-stream of data representing the neuron's final output. This output can then be fed to other neurons based on the connectivity matrix specified in the ANN design. The number of Neuron modules in a circuit will depend on the number of neurons specified in an ANN design. The number of outputs from any neuron module is always one, but the number of inputs can vary. Thus, for neurons requiring more or less than three inputs (as shown in Figure 32), the number of lookup tables, AND gates, and OR gates will increase or decrease accordingly.

The Bit-Stream Converter module is responsible for converting bit-stream signals into 12-bit output signals. To do this, it is necessary to define a fixed bit-stream pulse length over which to calculate the average probability of firing. The longer the bit-stream pulse length, the less statistical noise will be present in each signal. However, a longer bit-stream pulse length will also require more time to process signals. Therefore, a compromise between the two constraints was chosen and 4096 pulses was chosen as the bit-stream pulse length. The circuitry of the Bit-Stream Converter module is shown in Figure 33. Every 4096 pulses (or 3.3 msec), the outputs from the ANN circuit are updated. Interestingly, this is on a timescale that is comparable to biological neurons. For an ANN design with N number of primary outputs, N number of Bit-Stream Converter modules are required. The circuitry describing the design-time re-configurable ANN circuit is actually implemented in VHDL code, as opposed to the graphical representation shown in this section. In the course of its operation, the HEDANN design platform generates this VHDL code automatically.

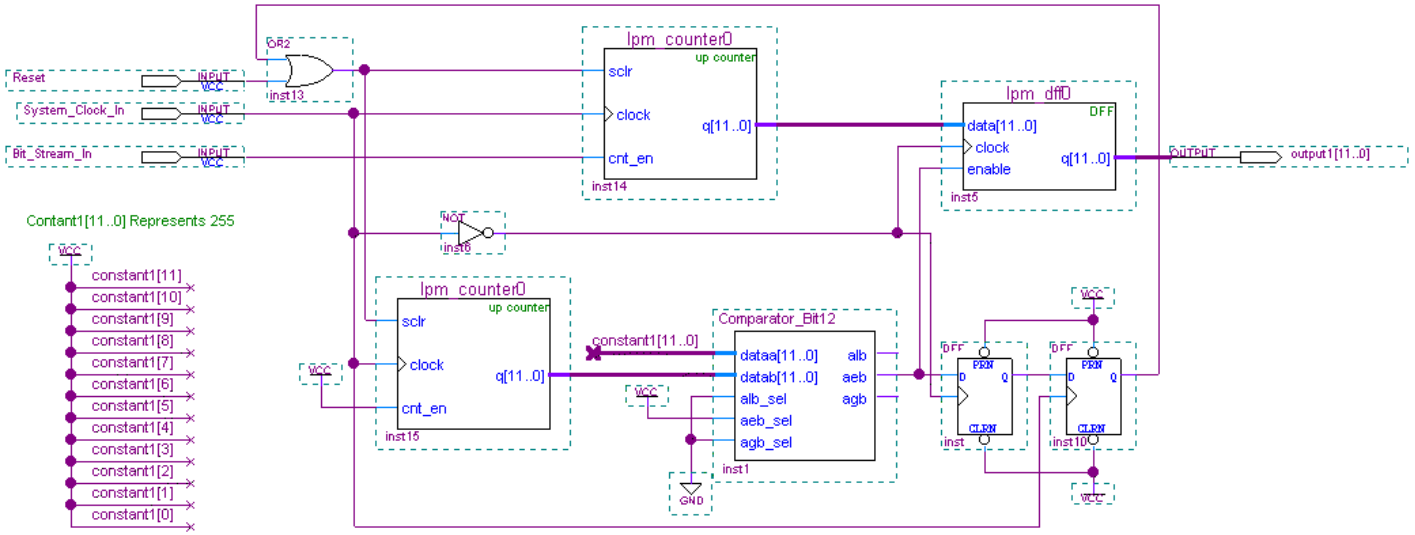


Figure 33. Schematic of "Bit-Stream Converter" Module

4.3 Run-Time Re-Configurable ANN Circuitry

The run-time re-configurable ANN circuit consists of the following six high-level modules:

- System Controller Module
- Random Pulse Generator Module
- Bit-Stream Generator Module
- Bit-Stream Converter Module
- Neuron Module
- Node Communications Module

The overall layout and interconnection of the six top-level modules is shown in Figure 34.

The architecture of the run-time re-configurable circuit is nearly identical to the design-time re-configurable circuit. However, there is the addition of one module that significantly changes the performance of the run-time re-configurable ANN circuit. The "Node Communication" module (or Node_Comm module) is responsible for allowing the weights and connections within the circuit to be changed during run-time. Using a 2-wire serial communication, the weights on individual connections can be manipulated without the need to re-route and re-program the FPGA. The details of the Node_Comm module are shown in Figure 35. This simple circuit allows serial data to be sent to a Node_Comm module when the proper node address is selected. The unique node address for each Node_Comm module is determined at design-time and is hardwired into the circuit.

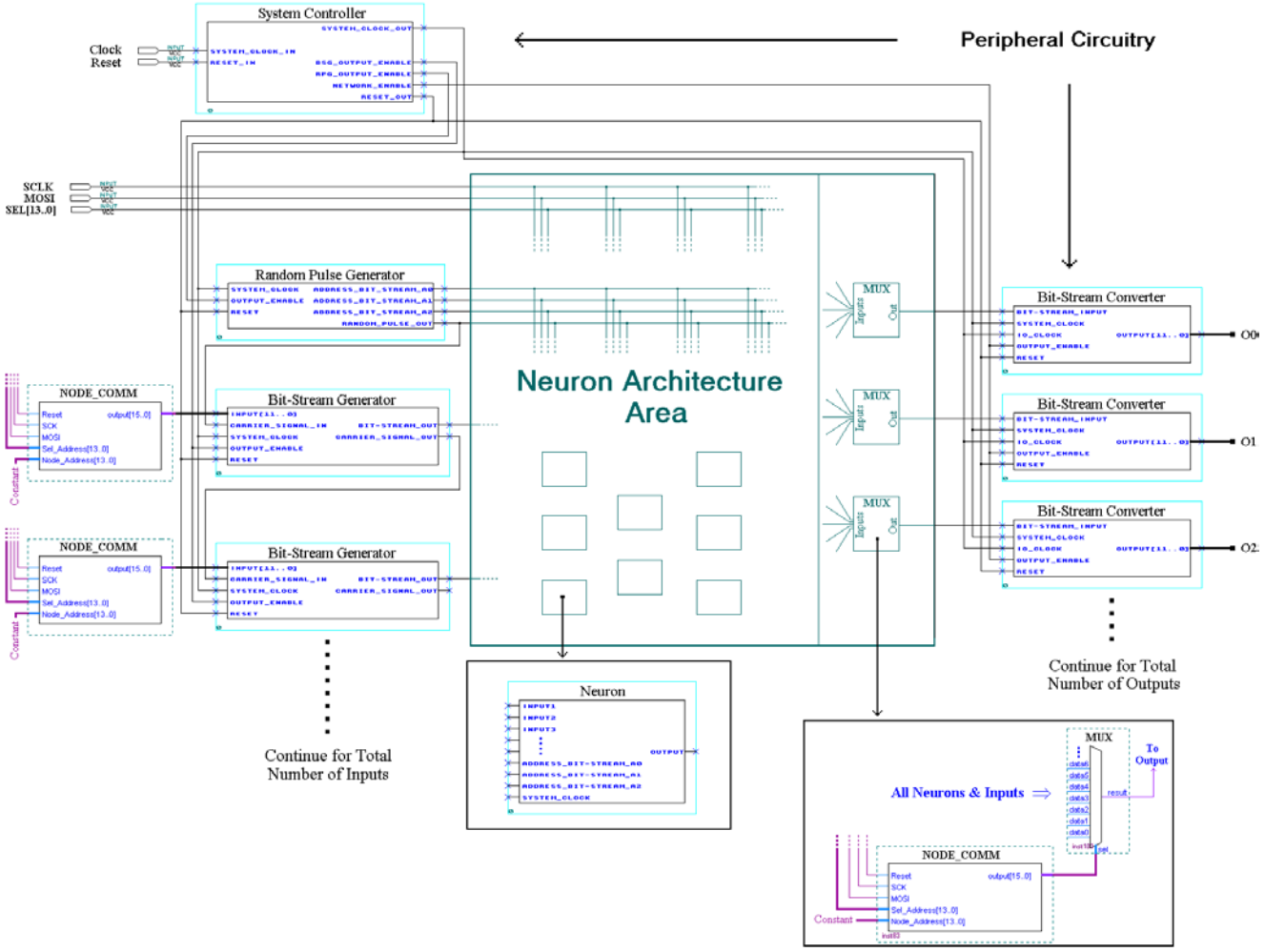


Figure 34. Top-Level Schematic of Run-Time Re-configurable ANN Circuit

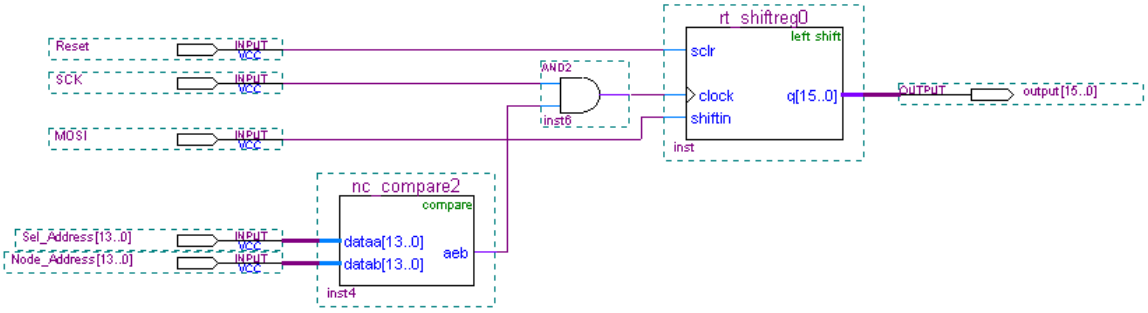


Figure 35. Schematic of Node_Comm Module

The circuitry describing the run-time re-configurable ANN circuit is actually implemented in VHDL code, as opposed to the graphical representation shown in this section.

Based on network design parameters provided to the HEDANN design platform, a single VHDL file is automatically generated for all future configurations of the run-time re-configurable ANN circuit.

4.4 Resource Requirements for Re-Configurable ANN Circuits

The design-time re-configurable ANN circuit presented in Section 4.2 is a relatively efficient way of creating ANNs with large numbers of neurons. Especially when the interconnections between neurons are kept to a minimum. If we define the connectivity of a network to be:

$$\text{Connectivity} = \frac{\text{Number of Actual Connections}}{\text{Number of Possible Connections}} = \frac{\text{Number of Actual Connections}}{(\text{Number of Neurons})^2}$$

(Equation 9)

Then Figure 36 describes the device resources required (in logic elements) to implement a design-time re-configurable ANN circuit of various size and connectivity.

The data shown in Figure 36 was acquired for actual designs placed-and-routed using Altera's Quartus II software with fast-fit option. To a first approximation, the resource requirements for a design-time re-configurable circuit can be approximated as:

$$\text{Resource Usage} = 2.74 \cdot C \cdot N^2 + 147 \qquad \text{(Equation 10)}$$

Where C = Connectivity and N = Number of Neurons. From Equation 10, we can see that, if the connectivity is fixed at 0.1, it is possible to create a design-time re-configurable ANN containing nearly 500 neurons using commercially available FPGAs.

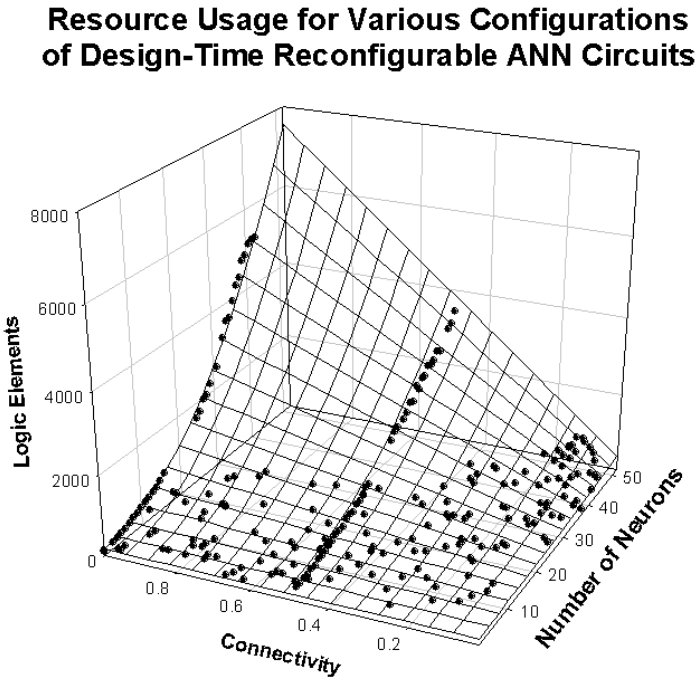


Figure 36. Graph of Resource Usage by Design-Time Re-configurable ANN Circuit

Unfortunately, as mentioned in Chapter 2, to implement a design-time re-configurable ANN circuit requires that each design in a population be synthesized, placed-and-routed, and downloaded to an FPGA. This process can take a considerable amount of time depending on the processor speed of the computer running the FPGA development tools. Although the actual time required to perform these tasks will vary considerably depending on the FPGA device family chosen, the FPGA development tool and optimization options selected, the number of neurons in a network, and the network connectivity, most designs containing between 10 and 1000 neurons can be placed in 1 to 10 minutes respectively (assuming a 2 GHz processor). Therefore, to evaluate a population of a hundred design-time re-configurable ANN circuits would require anywhere from 100 minutes to 16 hours (assuming again a 2 GHz processor). As processor speeds increase and FPGA development tools are improved, the compile times required to implement a population of design-time re-configurable ANN circuits will naturally reduce. However, this lengthy implementation time currently limits the number of design generations that can be evaluated and evolved in a reasonable period.

On the contrary, the run-time re-configurable ANN circuit presented in Section 4.3 is a relatively high-speed way of configuring ANN designs. Because a run-time re-configurable ANN circuit connects all neurons within a network, its connectivity is always 1.0. To re-configure a run-time re-configurable ANN circuit merely requires that all connection weights be rewritten via a two-wire serial interface and an 8-bit address line. Given a serial clock of 1 MHz, a network with ten to a hundred neurons can usually

be configured in less than 1 msec. Therefore, a generation of a hundred run-time re-configurable ANN circuits can typically be realized in less than one second.

The disadvantage of a run-time re-configurable ANN circuit is that it requires a significant amount of additional logic to enable run-time re-configurability. This additional circuitry taxes the available device resources and significantly limits the number of neurons that can be created on an FPGA. As a general rule of thumb, the resources required (logic elements) to implement a run-time re-configurable ANN circuit will increase as the square of the number of neurons in the network. Therefore, a ten-neuron run-time re-configurable ANN circuit will require roughly 25,000 logic elements to implement. A twenty-neuron run-time re-configurable ANN circuit will require close to 100,000 logic elements. As the size of the network becomes larger, the circuit quickly exceeds the resources available in today's commercial FPGAs. Currently, a 15-neuron run-time re-configurable ANN circuit is the largest ANN network that can be realized with commercially-available FPGA devices. As FPGA densities increase, the ability to implement larger run-time re-configurable ANN circuits should become more feasible.

This quick examination of resource requirements clearly identifies the need to explore both approaches of creating re-configurable ANN circuitry. Depending on the type of problem to be solved, one approach will likely be better suited than the other. If a problem is selected that is expected to require large ANN networks to solve, it might be best solved with the slower design-time re-configurable ANN circuit. However, if the problem is less complex and requires fewer neurons, it might be best solved with a run-time re-configurable ANN circuit.

The following two chapters will present a HEDANN design platform that is based on either a design-time or a run-time re-configurable ANN circuit. Each chapter will demonstrate the effectiveness of either system at solving non-trivial problems. Chapter 5 describes the evolution of a frequency recognition solution created from a design-time re-configurable ANN circuit, while Chapter 6 describes the evolution of a shape recognition solution created from a run-time re-configurable ANN circuit.

5.0 Demonstration of the HEDANN Design Platform Utilizing a Design-Time Re-Configurable ANN Circuit

5.1 Introduction

This chapter describes an experiment aimed at solving a frequency recognition problem by evolving a complex-architecture ANN circuit using the HEDANN design platform. In this experiment, the platform was configured such that it utilized the design-time re-configurable ANN circuitry described in Section 4.2. Because this configuration of the platform can require a considerable amount of time to evolve, a problem was chosen that was not likely to require large numbers of evolutionary generations to solve. The solution of the chosen problem required that a circuit be developed that responded linearly to square wave waveforms of increasing frequency, between 10 Hz and 70 Hz. A circuit with a linear frequency response (see Figure 37) was desired.

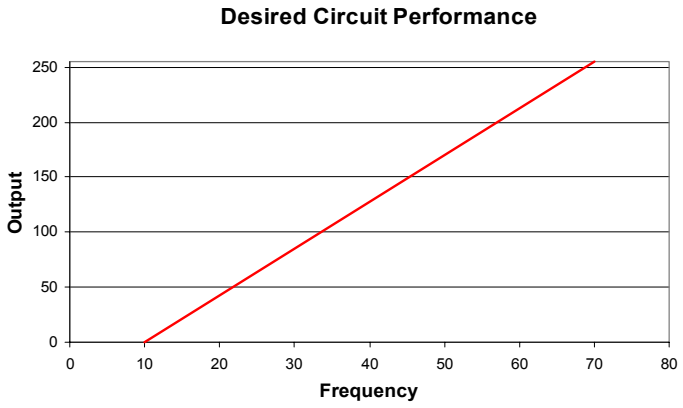


Figure 37. Desired Circuit Response for Design-Time Re-configurable ANN Circuit

Ergo, for a square wave input frequency of 10 Hz, the 8-bit output value of the ideal circuit would be 0. For a square wave input frequency of 70 Hz, the output would be 255. Such a problem was considered challenging because the clock speed of the evolving circuit is approximately 10,000 to 50,000 times faster than the input data frequency. In addition, the output of the circuit is not a simple binary output but rather represents a continuously increasing digital output (represented in 8-bit digital form as 0 to 255).

Interestingly, there is actually an analog of this experiment found in Nature. It has long been known that some ostracode crustaceans (commonly called seed shrimpt), like the one shown in Figure 38, are luminescent and use their luminescence in attracting mates of the same species. However, because many species of ostracodes are luminescent and live in close vicinity, each individual species has had to develop a unique, fluctuating luminescent display.

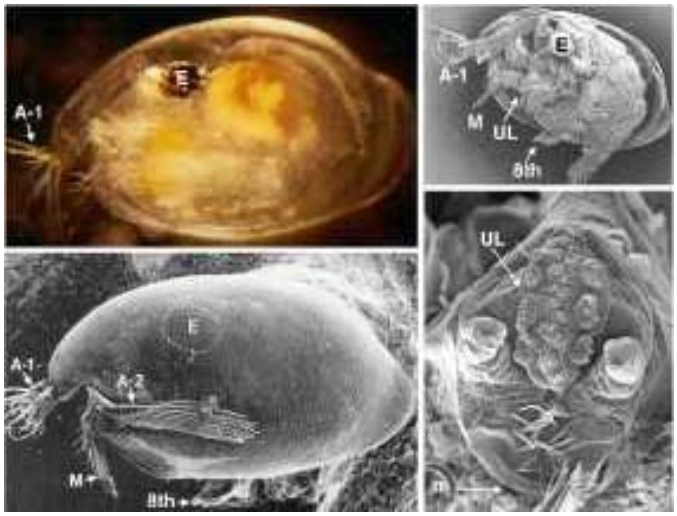


Figure 38. Microscopic Images of a Luminescent Ostracode

Equipped with a simple eye, the various species of ostracodes have, presumably, evolved to identify the unique fluctuation rate of their own species, thereby avoiding unsuccessful mating with ostracodes of a different species. The fluctuation frequencies are quite low and the neural networks present in the ostracode are fairly limited (with the ostracodes' protocerebral encompassing only a couple hundred to thousands of neurons). However, research seems to indicate that this frequency recognition capability has been evolved numerous times in various ostracode species, and in possibly a relatively short number of generations. These findings would appear to indicate that the proposed frequency recognition problem is well suited to the capabilities of the design-time re-configurable ANN circuit.

5.2 Experiment

As described in Chapter 3, the HEDANN design platform consists of a 2 GHz desktop computer equipped with a National Instruments DAQCard-1200 multifunction data acquisition board. For this experiment, the DAQCard-1200 was used to create square waveforms of various frequencies (ranging from 10 to 70 Hz) and to monitor the responding output signal of an evolving circuit. The digital I/O pins of the DAQCard-1200 were connected to the breakout pins of a Constellation 10KE Prototype Board from Nova Engineering, which featured a 200,000 gate Altera Flex10K FPGA device (EPF10K200SRC240). Design-time re-configurable ANN circuits could be quickly downloaded to the FPGA development board, through a USB cable, using Nova's Constellation Programming software.

An initial population of 1000 designs were randomly generated and evolved for two generations before the population size was reduced to 100 members. Throughout the evolutionary process, a mutation probability rate of 0.15 was used in creating future generations. Networks of any size and any architecture were permitted as long as they met they criteria stated in Section 3.3 and they did not exceed the available resources of the FPGA.

Compilation of each design was achieved using the HEDANN design platform in concert with Altera's Quartus II FPGA Development software. The software's place-and-route algorithms were configured for circuit speed optimization and the tools "fast-fit" option was employed. All compiled designs were required to meet timing requirements imposed to eliminate the occurrence of race conditions.

As mentioned, the HEDANN design platform utilized the DAQCard-1200 to generate the various frequency square waveforms used to evaluate each implemented circuit. However, during testing, due to limitations in multi-tasking, these waveforms were not always perfectly accurate. For example, a 30 Hz signal might actually exhibit fluctuations of +/-2 Hz over time. Therefore, the testing inputs to the evolving circuits contained some degree of random and uncharacterized noise. To fully evaluate each implemented design, the HEDANN design platform generated 120 various frequency waveforms (between 10 and 70 hertz) that were applied to each circuit design under test. For each waveform, the FPGA-implemented ANN circuit was tested for roughly 300 msec. The average output from the circuit, over that time, was measured and the fitness calculated based on the difference between the circuit's output and the desired output. All testing was performed at room temperature (22°C).

5.3 Results and Analysis

The HEDANN design platform was allowed to evolve unattended for roughly two months. In that time, the fitness of the design improved from an initial fitness of 0.5 to 0.86 in 300 generations. The fitness of each design was calculated using Equation 11 and 12:

$$fitness_i = 1 - \frac{\left| O_i - \left(\frac{255}{60} \times f_i - \frac{255}{6} \right) \right|}{255 \times \left(\frac{|f_i - 40|}{60} + 0.5 \right)} \qquad \text{(Equation 11)}$$

$$fitness = \sum_{i=1}^{N} \frac{fitness_i}{N} \qquad \text{(Equation 12)}$$

Where f_i is the input frequency in Hertz, O_i is the digital circuit output (0 - 255), and N is the total number of test vectors. The evolution of the highest fitness design-time re-configurable ANN circuit for each generation is shown in Figure 39. The output of the evolving design was observed to be steadily improving throughout the evolutionary process. Figure 40 illustrates the population's best circuit performance after the initial 1st generation.

Obviously, the initial response of the ANN circuit is essentially random, regardless of frequency. However, after approximately 150 generations, the ANN circuit response had evolved into a more structured output that responded differently for higher frequency inputs versus lower frequency inputs, as shown in Figure 41.

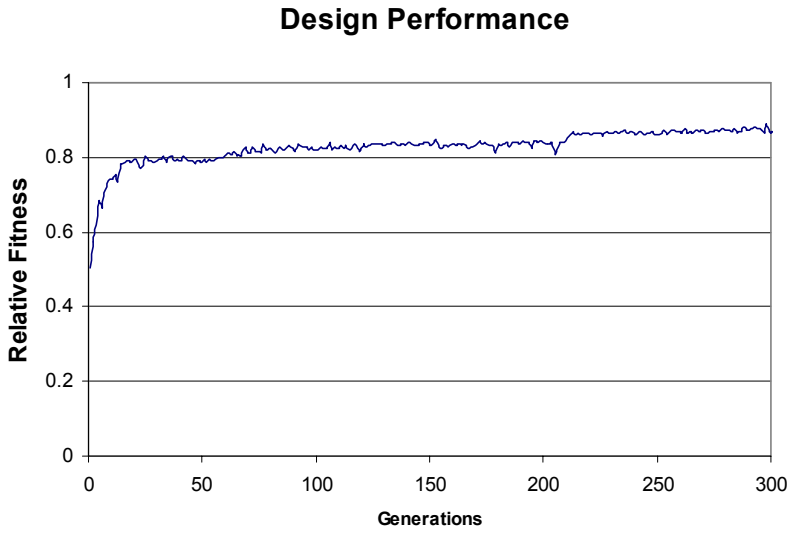


Figure 39. Evolutionary Performance of Design-Time Re-configurable ANN with Increasing Generations

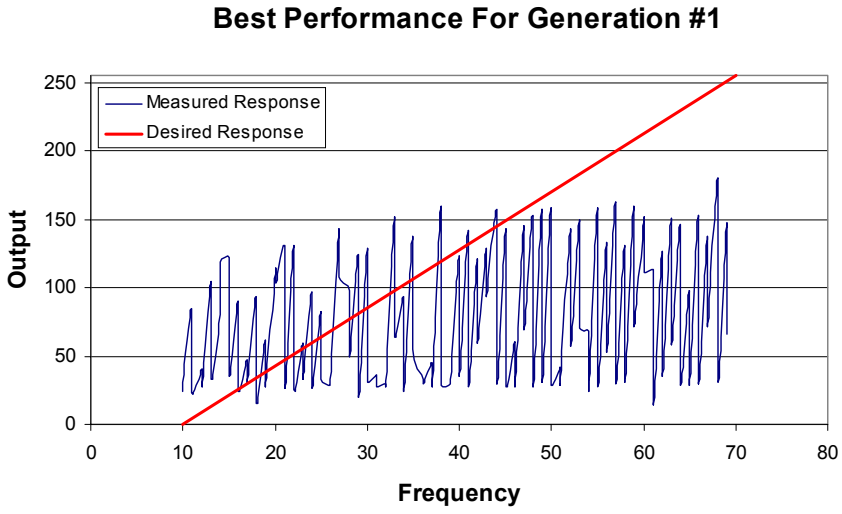


Figure 40. Performance of Highest Fitness Design-Time Re-configurable ANN After 1st Generation

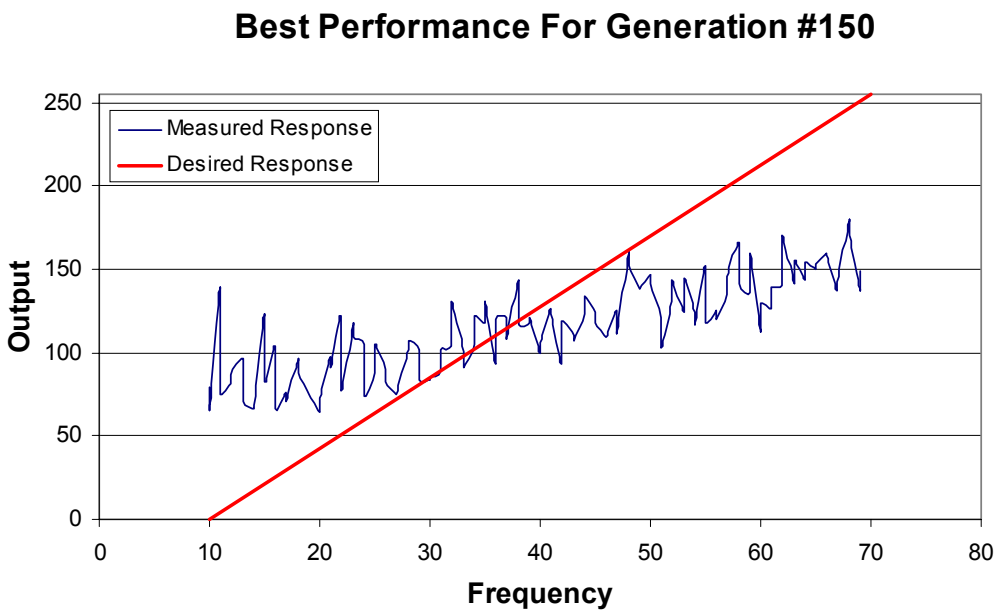


Figure 41. Performance of Highest Fitness Design-Time Re-configurable ANN After 150 Generations

Finally, after 300 generations, the output of the evolved ANN circuit closely approximated the desired linear frequency response, as shown in Figure 42. The change in network architecture throughout these 300 generations is qualitatively represented in Figures 43 – 45.

Although the number of neurons was constrained to 100 neurons or less, the final evolved network size actually appears to be quite small. In fact, within the first few generations the average member of a population exhibited twenty neurons or less. The final evolved ANN is also sparsely interconnected, with each neuron connected to only 3 neighboring neurons (on average). As with most evolutionary designs, how the final design achieves its frequency recognition capabilities is unclear.

Because the population fitness shown in Figure 39 does not appear to be approaching an asymptotic value, it is likely that continued evolution would have produced designs with improved performance. However, due to time constraints, continued evolution was not pursued and the evolutionary process was terminated after only 300 generations.

On average, the amount of time required by the Design-Time Reconfigurable HEDANN design platform to evaluate a single generation of designs was 4.8 hour. For the design problem presented in this chapter, the factors influencing this resource usage are summarized in Table 3. The values in Table 3 are average values and would likely vary from design to design based on ANN network size and complexity. However, from Table 3, one can clearly see that the design compilation time easily represents the most significant consumer of time.

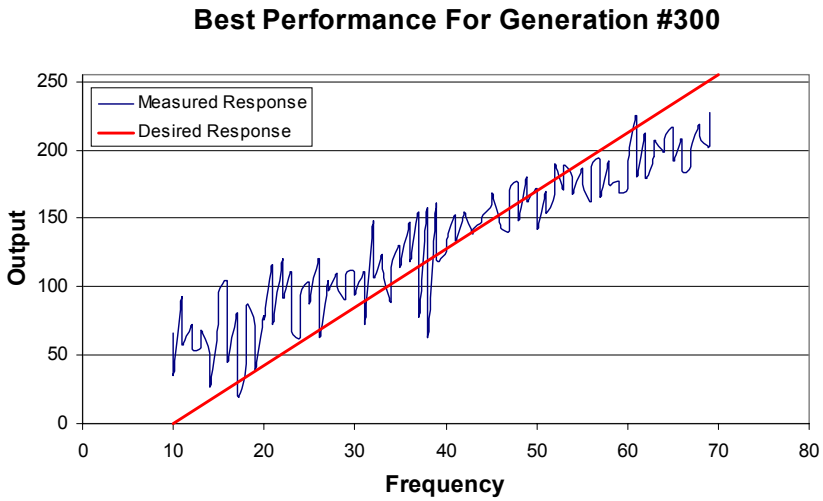


Figure 42. Performance of Highest Fitness Design-Time Re-configurable ANN After 300 Generations

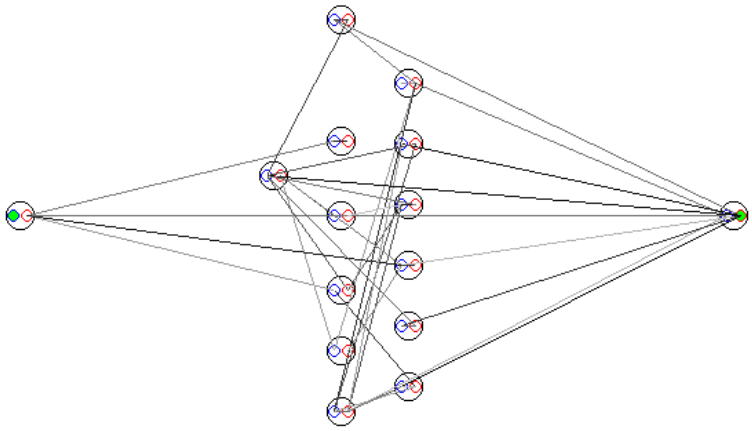


Figure 43. Best Design-Time Re-configurable ANN's Network Architecture after 1st Generation

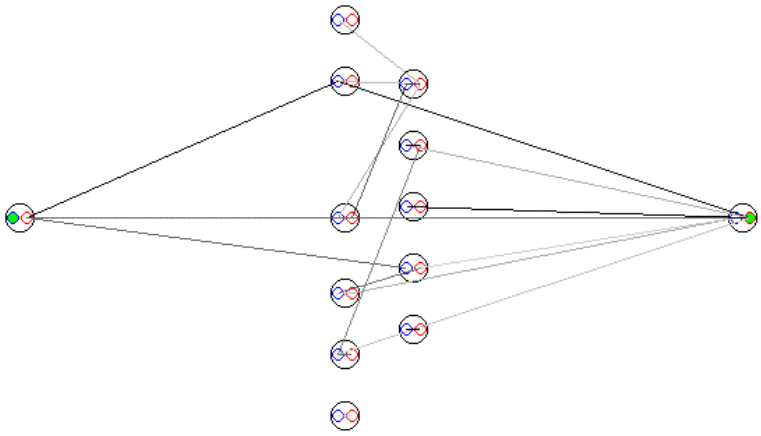


Figure 44. Best Design-Time Re-configurable ANN's Network Architecture after 150 Generations

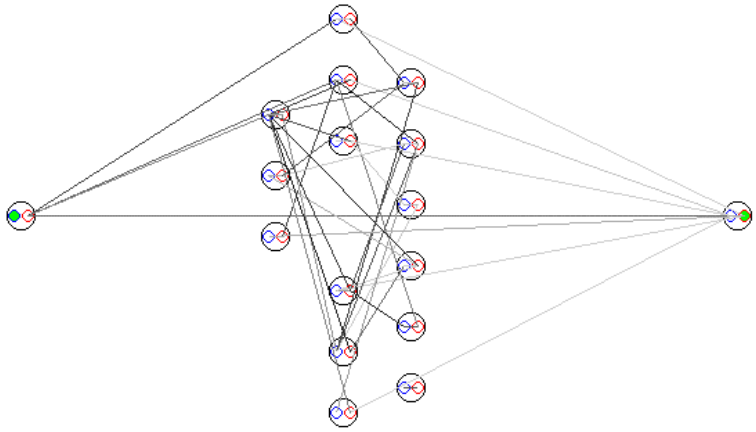


Figure 45. Best Design-Time Re-configurable ANN's Network Architecture after 300 Generations

Table 3: Time Usage in Design-Time Re-configurable Approach

Operation	Percentage of Required Time
Design Compilation	90%
Uploading of Design File	6%
Testing	3%
Genetic Algorithm Routines (breeding, file backup, etc.)	>1%
Other	>1%

5.4 Conclusions

The experiment presented in this chapter illustrates that the HEDANN design system, utilizing a design-time re-configurable ANN circuit, can be used to evolve complex-architecture ANN designs capable of exhibiting simple behaviors similar to those found in Nature. The final evolved ANN circuit required 286 logic elements, contained 16 neurons, exhibited a network connectivity of 0.19, and achieved a final fitness of 0.86. Although this trial problem required two months to evolve a solution, the majority of this time can be traced to design compilation operations. Approximately 90% of the evolutionary time was spent synthesizing and placing-and-routing the population designs. The remaining 10% of the evolutionary time was required for downloading design files to the FPGA and testing. The time required for testing was dependent on the input frequencies of the square waves.

6.0 Demonstration of the HEDANN Design Platform Utilizing a Run-Time Re-Configurable ANN Circuit

6.1 Introduction

To evaluate the performance of the HEDANN design platform, utilizing run-time re-configurable ANN circuitry, a suitable test problem is again needed. As mentioned in Chapter 3, the run-time re-configurable HEDANN system requires a significant amount of FPGA device resources. At the writing of this dissertation, this requirement permitted the development of ANNs with no more than 10 to 15 neurons. Therefore, a problem was desired that could demonstrate how a small number of neurons, if connected in a complex architecture, could evolve powerful processing capabilities.

Fortunately, Nature has provided some guidance in this area, since many animals exhibit neural sub-networks, featuring a relatively small number of neurons doing a considerable amount of processing. Specifically, the retina is known to act as a complex pre-processor of images acquired by the eye. Images projected on the retina are significantly de-composed and evaluated prior to being transmitted to the brain. Motion measurements, shape decomposition, segmentation, and other processes are believed to occur initially in the retina. One of the basic functions of the retina is to decompose complex shapes into simple line segments. For example, when looking at a square, the brain doesn't identify that shape by comparing the full image of the square to past images of known squares. Rather, the retina decomposes the square into four straight lines, with adjacent lines joined at right angles. This "higher level" information is then transmitted

to the brain via the optic nerve. Whether future squares are large or small, bright or dark, straight or rotated doesn't matter. The retina still recognizes a square as merely containing four lines joined at right angles. Likewise, a similar argument can be made for triangles and other shapes.

To evolve an ANN capable of generalized, space-invariant, rotationally invariant, shape recognition, is non-trivial even for simple shapes such as squares and triangles. Consequently, to create an accurate, or even partially-accurate, shape recognition circuit using an ANN design with only ten neurons would represent a significant achievement. As such, a shape-recognition problem was chosen that required an ANN to be evolved that could effectively discriminate between squares shapes, triangular shapes, and random noise of any orientation and size. The input to the system was an 8 pixel by 8 pixel image that contained 8-bit pixel intensities ranging from 0 to 255. When fully evolved, the final system was expected to return a single output equal to 250 ± 5 for square shapes, 128 ± 5 for triangular shapes, and 5 ± 5 for random noise.

6.2 Experiment

As described in Chapter 3, the Run-Time Re-configurable HEDANN System consists of a 2 GHz desktop computer equipped with a National Instruments DAQCard-1200 multifunction data acquisition board. For this experiment, the digital I/O pins of the DAQCard-1200 were connected to the breakout pins on an Altera APEX DSP Development board featuring a 1.5 million gate FPGA (EP20K1500EBC652-1X). The FPGA development board was configured such that, on power-up, a previously compiled

design file was downloaded automatically to the FPGA from an on-board memory device. That design file was the run-time re-configurable ANN circuit discussed in Section 4.3, configured such that it featured 64 addressable network inputs, 1 network output, and 10 stochastic neurons. Because the run-time re-configurable HEDANN system permits dynamic weights for each possible network connection, the final compiled design required >75% of the FPGA's resources. However, unlike the design-time re-configurable HEDANN system, the connectivity of the network could be completely modified without the need to recompile higher-level design files. Therefore, complete reconfiguration of the network architecture was theoretically possible in less than five microseconds, with the appropriate hardware. However, because of bottlenecks in the DAQCard-1200 hardware used in this experiment (which had a sample rate of only 20kHz), the actual reconfiguration time was closer to 150 milliseconds.

To evolve an ANN design targeting generalized shape recognition, a training set of images needed to first be generated. A Visual Basic program was written to generate an equal number of squares, isosceles triangles, and random noise within an 8 pixel by 8 pixel image array. The rotation, translation, and intensity of the generated images were allowed to vary. In all, 99 training images were created that contained the full shapes (i.e. no portion of the shape was allowed to be "out of view" or obscured). Figures 46-48 show examples of the generated images. In addition to the 99 training set images, 300 similar images were generated for use in evaluating the final evolved system.

To evaluate a network design, the run-time re-configurable HEDANN system would first re-configure the weights of the FPGA-implemented ANN circuit to reflect the weights/architecture of a desired design.

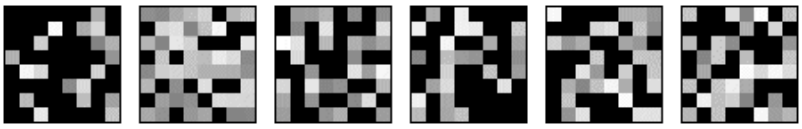


Figure 46. Example of "Random" Trial Image

Figure 47. Example of "Triangle" Trial Image

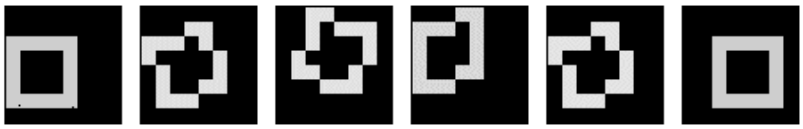


Figure 48. Example of "Square" Trial Image

As mentioned previously, this was accomplished through a two-wire serial communication scheme that allowed each weight to be individually addressed and modified. Once the network weights had been properly modified, the inputs to the network were loaded with the pixel data for a given trial image. The FPGA-implemented ANN had 64 addressable inputs, one for each pixel in the 8x8 array. Again, a two-wire serial communication scheme was used to address and load each input with its proper pixel value.

Once all weights and inputs had been configured, the output from the FPGA-implemented ANN circuit was evaluated. The 8-bit output from the ANN circuit was connected to eight external pins on the FPGA prototype board, allowing the HEDANN system's DAQCard-1200 data acquisition board to monitor the network's output. The

output was monitored at 20,000 samples/sec for roughly 300 milliseconds. After 300 msec, the average output value for the circuit was calculated and used to evaluate the circuit's performance. This process was repeated for all 99 trial images before a new design could be implemented and evaluated.

Initially, a population size of 1000 was used by the HEDANN design platform's genetic algorithm. This population size, however, was reduced to 100 after the second generation to increase the speed of development. The reasons for this change are discussed in greater detail in Chapter 7.

6.3 Results and Analysis

The HEDANN design platform was allowed to evolve unattended for roughly two and a half months. In that time, the fitness of the design improved from an initial fitness of 0.34 to 0.48 in 1500 generations. The fitness of each design was calculated using Equations 13-16:

$$fitness_{Random_i} = \begin{cases} 1 - \dfrac{O_i}{127.5} & \text{if } O_i < 127.5 \\ 0 & \text{if } O_i > 127.5 \end{cases} \qquad \text{(Equation 13)}$$

$$fitness_{Triangle_i} = \begin{cases} 0 & \text{if } O_i < 127.5 \\ \dfrac{O_i}{127.5} - 1 & \text{if } O_i > 127.5 \end{cases} \qquad \text{(Equation 14)}$$

$$fitness_{Square_i} = 1 - \frac{|127.5 - O_i|}{127.5} \qquad \text{(Equation 15)}$$

$$fitness = \sum_{i=1}^{N} \frac{fitness_i}{N} \qquad \text{(Equation 16)}$$

Where O_i is the digital circuit output (0 - 255) and N is the total number of test vectors. The evolution of the run-time re-configurable ANN circuit is shown in Figure 49.

The performance of the evolving designs appeared to be slowly approaching an asymptotic value of approximately 0.5. Therefore, the evolution of the designs was halted around 1500 generations after no significant improvement in fitness had been observed for the past 250 generations. Figure 50 illustrates the population's best circuit performance after the first generation.

After the first generation, the highest-fitness circuit would output a proper response 33% of the time when viewing a square. Likewise, the proper response value was output 2% of the time for a triangle, and 5% of the time for a random image. This made the cumulative accuracy of the circuit roughly 13.3% accurate at properly identifying the difference between a square, triangle, or random noise shape.

After 500 generations, the evolved designs were capable of accurately identifying one of the three possible shapes (square, triangle, noise) very well, and were having growing success at detecting other shapes. Figure 51, shows the accuracy with which the highest fitness design in generation 500 was able to identify the various trial images. In Figure 51 we see that the highest-fitness circuit output a proper response 100% of the time when

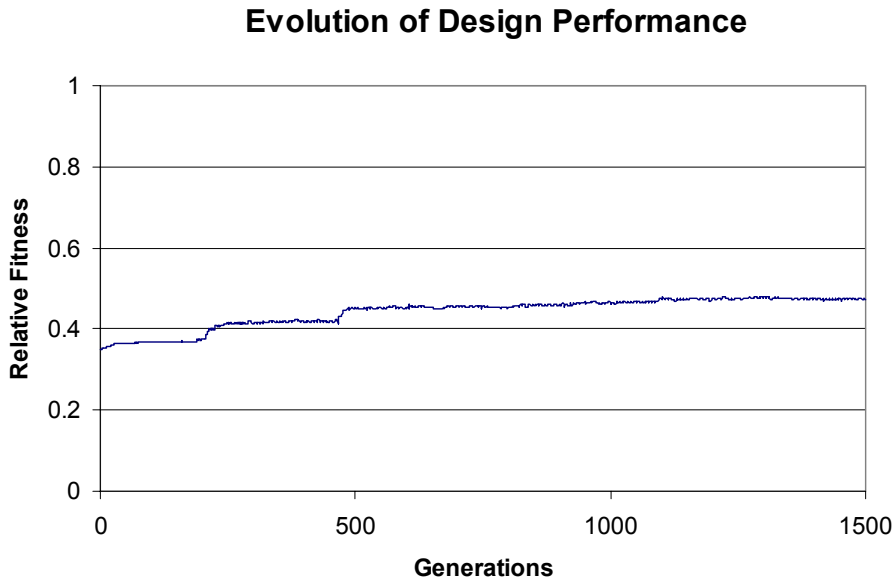


Figure 49. Performance of Run-Time Re-configurable ANN with Increasing Generations

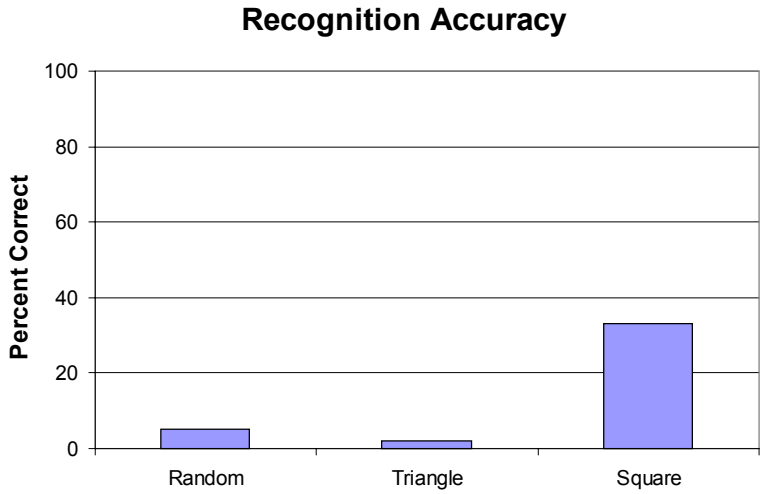


Figure 50. Performance of Highest Fitness Run-Time Re-configurable ANN After 1st Generation

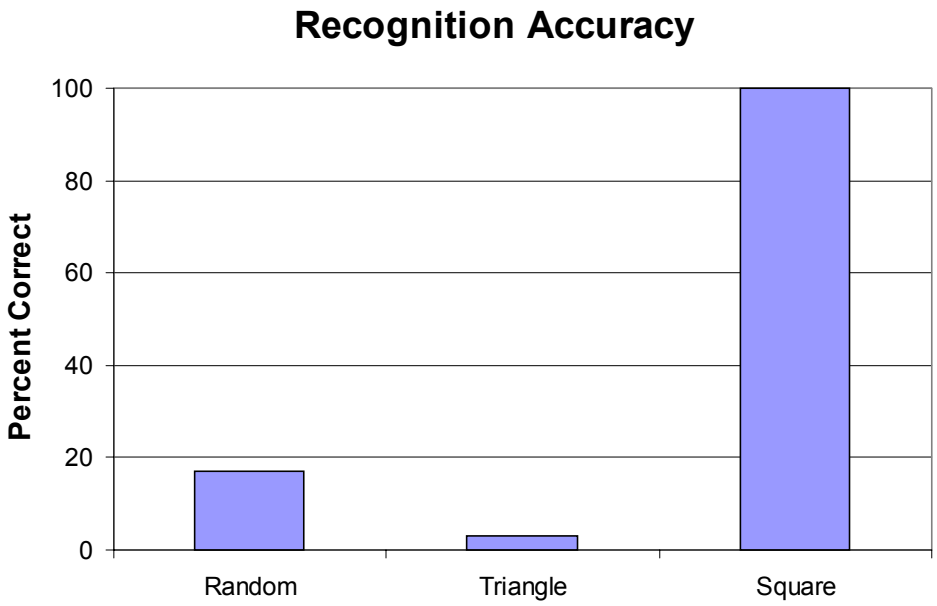


Figure 51. Performance of Highest Fitness Run-Time Re-configurable ANN After 500 Generations

viewing a square, 3% of the time for a triangle, and 17% of the time for a random image. Therefore, the cumulative accuracy of the circuit was roughly 40% at identifying the difference between a square, triangle, or random noise shape.

After 1500 generations, the highest fitness design exhibited a fitness of 0.48 and was capable of properly identifying a square shape with 91% accurate, a triangular shape with 19% accuracy, and a random noise image with 37% accuracy. Therefore, the cumulative accuracy of the circuit was nearly 49%. The performance of the final evolved design is shown in Figure 52. At 1500 generations, the evolutionary process was stopped due to time constraints and apparent fitness convergence. It is possible that continued evolution of the designs would have produced ANNs with improved performance. However, it is more likely, in the author's opinion, that the small network size of the run-time re-configurable circuit is limiting the maximum achievable performance. If correct, then continuing the evolutionary process would have failed to generate an improved solution and a larger FPGA would have to be targeted to permit larger ANNs to be explored.

The change in network architecture throughout these 1500 generations is qualitatively represented in Figures 53-55. From Figures 53-55, it appears that a lightly interconnected network, with relatively small connection weights, produces the most favorable configuration. As in Chapter 5, given the complexity of the interconnections it was not possible to determine how the ANN achieved its shape recognition capability.

On average, the amount of time required by the Run-Time Reconfigurable HEDANN design platform to evaluate a single generation of designs was 1.2 hours. For the design problem presented in this chapter, the factors influencing this resource usage are summarized in Table 4.

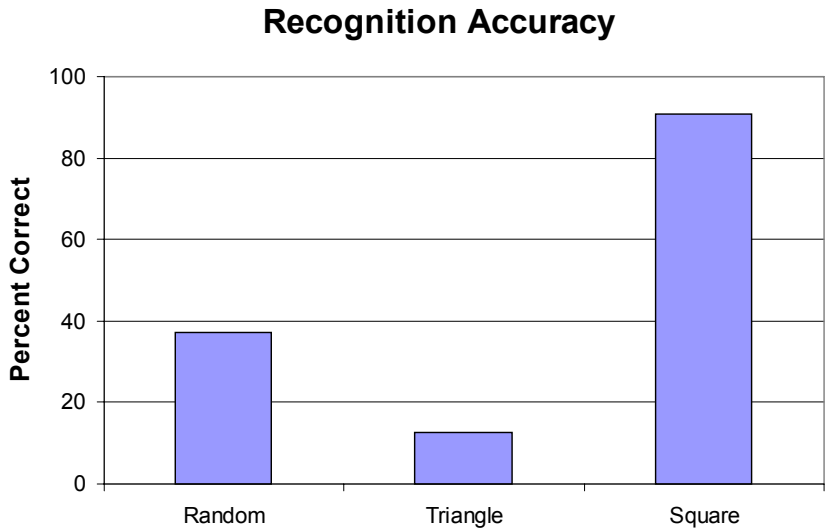


Figure 52. Performance of Highest Fitness Run-Time Re-configurable ANN After 1500 Generations

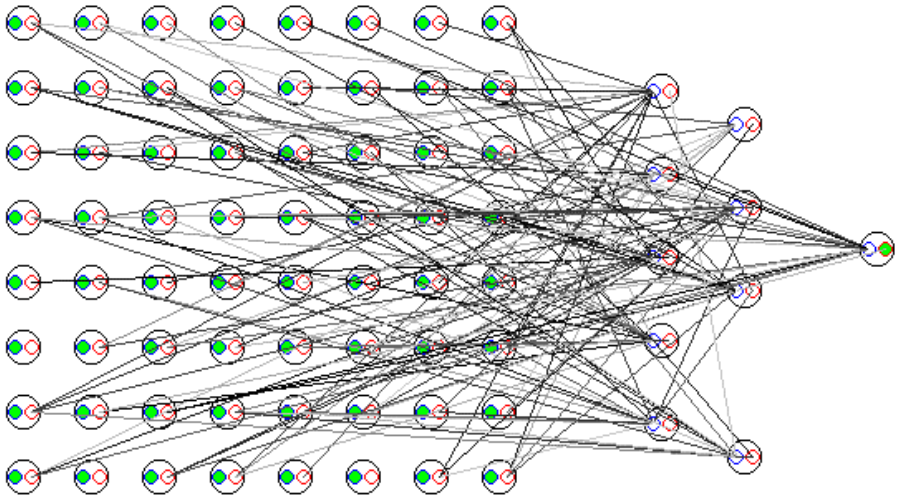


Figure 53. Leading Run-Time Re-configurable ANN's Network Architecture after 1st Generation

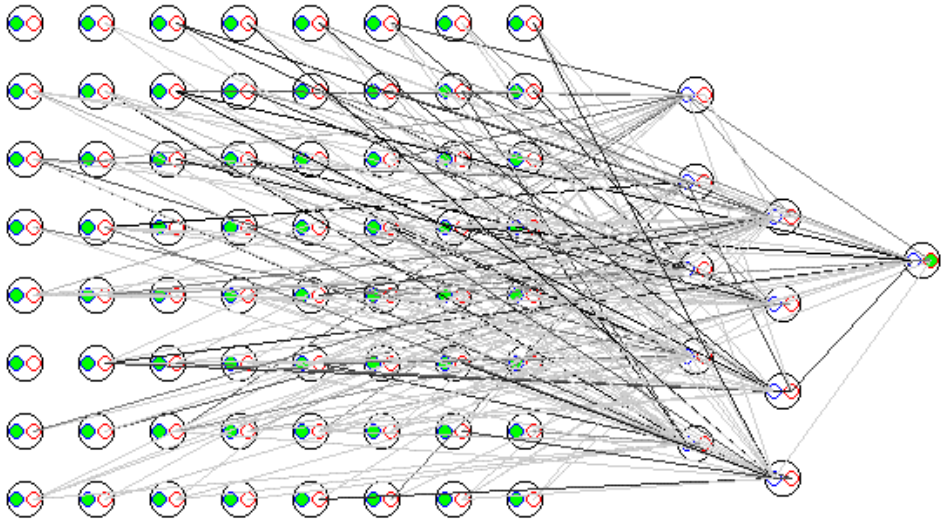


Figure 54. Leading Run-Time Re-configurable ANN's Network Architecture after 500 Generations

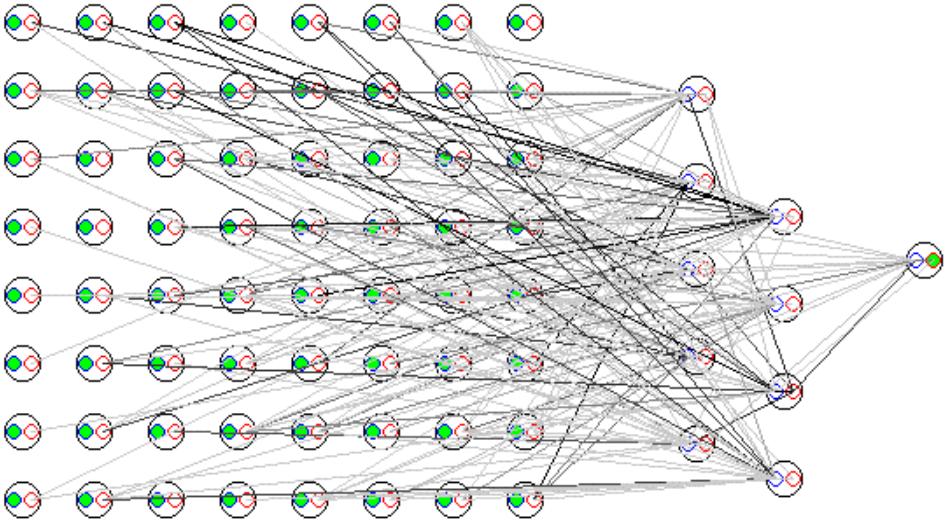


Figure 55. Leading Run-Time Re-configurable ANN's Network Architecture after 1500 Generations

Table 4: Time Usage in Run-Time Reconfigurable Approach

Operation	Percentage of Required Time
Testing	75%
Reconfiguration of FPGA	24%
Genetic Algorithm Routines (breeding, file backup, etc.)	>1%
Other	>1%

From Table 4, it is clear that the testing and reconfiguration of the FPGA circuit consumed the majority of the required evaluation time. Both of these processes required the uploading of either weight or input data through a serial data transfer line. Unfortunately, the serial data transfer line was capable of a transmission rate of only 20 kbits/second, which contributed significantly to the lengthy reconfiguration and testing process.

6.4 Application-Specific Optimization

To ensure the application of the HEDANN design platform to a wide diversity of AI problems, the platform's software and associated hardware were designed to be highly flexible and re-configurable. As such, the platform has not been optimized for any one particular set of problems or input patterns. The solution of a time-dependent, single input, frequency recognition problem (presented in the previous chapter) and a time-independent, 64 input, shape recognition problem (presented in this chapter) demonstrate the versatile nature of this platform. However, the price for this flexibility is the significant reduction in evolutionary speed (e.g. the 2-3 months of evolutionary design time for the simple problems presented in this and the previous chapter). However, if an application is to be studied in depth, it is often possible and worthwhile to optimize the platform for the specific requirements of that particular problem. Often, increases in circuit clock frequency and circuit I/O transfer methods will be likely ways for improving the performance of the platform. Therefore, this section provides an example of how this optimization could be accomplished for the shape recognition problem presented in Section 6.2.

As was presented in the previous section, a position, rotation, and intensity invariant shape recognition solution was sought that could recognize the differences between square shapes, triangular shapes, and random noise. Initially, the platform required 1500 generations (evaluated over 75 days) before eventually converging to a recognition accuracy of only 48% (fitness value = 0.48). Because the test patterns associated with this problem are not time-dependent, the speed at which the test patterns are sent to and interpreted by the FPGA is non-critical. As such, it should be possible to dramatically improve the speed of the HEDANN design platform by increasing the serial data transfer rate associated with the transmission of input test data to the FPGA. Likewise, the clock frequency of the FPGA can also be increased to reduce the amount of time required by the FPGA in evaluating input data. As mentioned previously, a National Instrument, DAQCard-1200, general purpose, data acquisition card is used to serially transmit and receive all weight and input data to the evolving FPGA circuit. Unfortunately, the digital I/O on the DAQCard-1200 is static/non-latched I/O. This means that the digital outputs from the card can only be updated through programmable, interrupt-driven communications with the PC's main processor. For the PC used with this dissertation research, this resulted in a serial data transfer rate of only 20 kbits/second. By modifying the system hardware to permit serial communications with the FPGA prototype board using the computer's RS232 COM serial port, the transfer rate was easily increased to nearly 200 kbits/second. This simple modification, which required some alteration of the connection circuitry between the PC and the FPGA prototype board and some changes to the software, resulted in a 10 times improvement in serial data transfer rate. In addition, the clock speed of the FPGA circuitry was initially only 1.25 MHz. This constraint was

initially imposed on the system to avoid timing conflicts that could arise as more complicated ANN designs were explored. However, for the small 10-neuron network being employed for this problem, it should have been possible to operate at a much higher clock frequency. Therefore, the clock frequency of the ANN circuitry was increased to 40MHz (the fastest available clock speed on the FPGA Prototype board) and past, evolved circuits were evaluated to verify that the circuitry operated properly at this higher clock frequency.

Together, the improvement in clock speed and serial data transfer rate resulted in an optimized HEDANN design platform that was roughly 25 times faster than the original platform. Therefore, 1500 generations of evolutionary design could be accomplished in only three days.

Four experiments were performed to evaluate the HEDANN design platform's performance given various platform settings. These experiments included:

Experiment #1: The original experiment described in Section 6.3 was repeated at the higher 40 MHz clock speed and higher serial data transfer rate. As shown in Figure 56, its final performance (fitness value = 0.47) is relatively the same as in the previous section (fitness value = 0.48).

Experiment #2: As recommended in Section 6.3, the complexity of the problem was relaxed to better suit the small network size implemented on the FPGA. As before, a set of test patterns were generated which produced square shapes, triangular shapes, and random noise with varying levels of intensity and positional placement (notice

that the rotational variation of the shapes was eliminated from this new test set). The results of this simplified experiment are also shown in Figure 56 and demonstrate the improved fitness that resulted within the same evolutionary period (fitness value = 0.58 for 1500 generations).

Experiment #3: The position and intensity-invariant shape recognition problem of Experiment #2 was repeated with the mutation rate doubled (mutation rate = 0.30). This change in the genetic algorithm's mutation rate resulted in a slight improvement in final performance (fitness value = 0.60).

Experiment #4: The conditions of Experiment #3 were repeated but the precision of the network's weight values was reduced from 16-bits to 4-bits. This resulted in a reduction of the search space but, unexpectedly, did not produce an improved fitness value within the same evolutionary period (fitness value = 0.5). Potentially, the high precision of the weight values may be required to adequately evaluate the varying intensity values of the 12-bit data inputs.

Each experiment required approximately 72 hours of run time, which corresponded to the evaluation of a full generation of designs every 173 seconds. As in Section 6.3, the population consisted of 100 members, with the first two generations featuring a population size of 1000. The highest fitness value occurred for Experiment #3, in which a final fitness value of 0.60 ± 0.023 was reached. The performance and architecture of the network are shown in Figures 57 and 58.

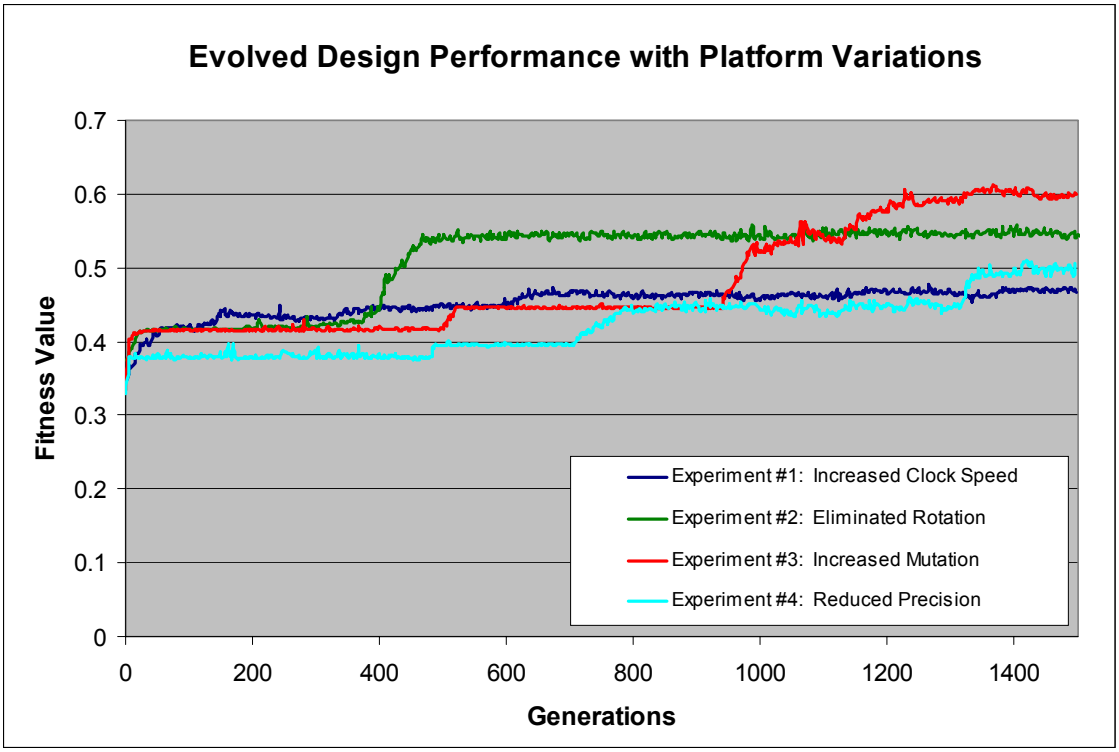


Figure 56. Evolutionary Trends with Platform Variations

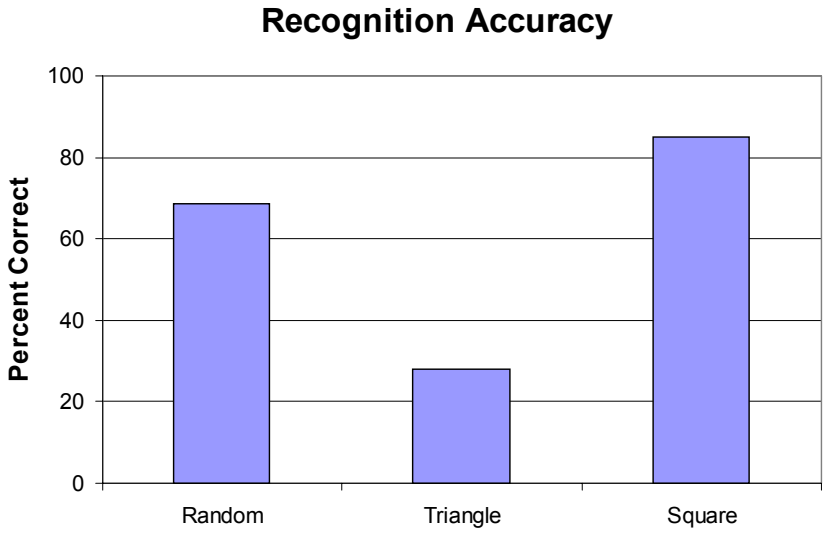


Figure 57. Best Performance for Experiment #3

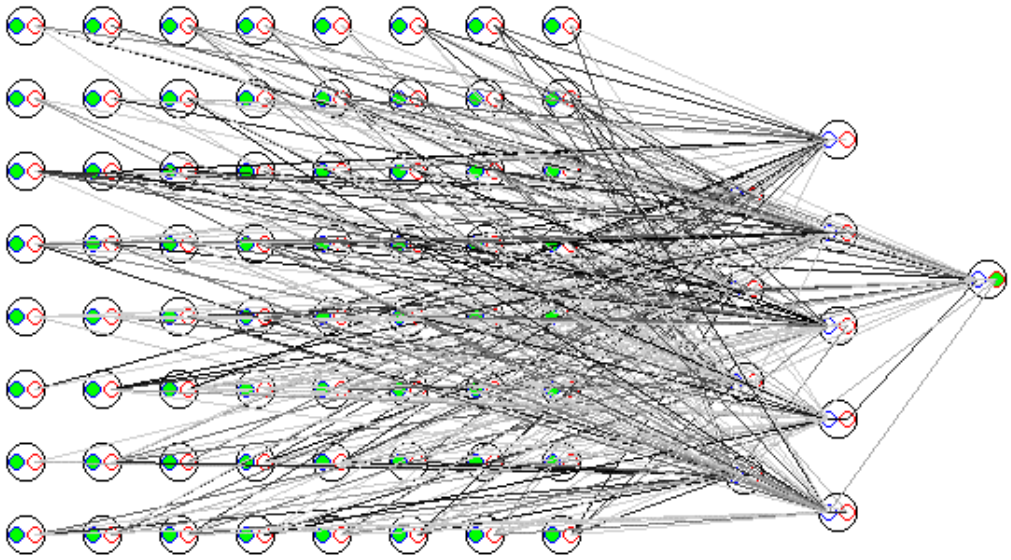


Figure 58. Best Performing ANN for Experiment #3

Although trends in evolutionary data are notoriously difficult to substantiate, it can be observed that changes to the various hardware/software parameters of the HEDANN design platform do produce noticeably differing results. With continued exploration, it is likely that an optimal platform setting could be found that resulted in even further improvements in evolved circuit performance. It is precisely this feature of the HEDANN design platform that makes it such a valuable tool to researchers studying various aspects of complex-architecture artificial neural network design. By providing a flexible software/hardware platform that can be easily re-configured to explore various applications and design variables, the HEDANN design platform provides an accessible and valuable tool to ANN researchers currently constrained by slow, code-intensive, computer-based design systems.

6.5 Conclusions

The experiments presented in this chapter demonstrate that the HEDANN design platform, utilizing a run-time re-configurable ANN circuit, can be used to evolve complex-architecture ANN designs with the potential to exhibit sophisticated processing capabilities. Through application-specific optimization, this platform evolved a circuit that was able to achieve 60% accuracy at discriminating between three shapes. Given the small network size, this performance is quite impressive. The final evolved ANN circuit contained only 10 neurons, exhibited a network connectivity of 0.149, and achieved a final fitness of greater than 0.60. The optimized system required three days to evolve the final network circuitry and could likely have shown fitness improvements with increased optimization of the various platform settings. Aspects of the application-specific optimization could also be targeted for use with the more generalized implementation of the HEDANN design platform. Particularly, the increase in FPGA clock frequency and serial data transmission rate would be possible with newer, faster, FPGAs and higher speed, programmable, digital I/O cards. It is likely that these improvements would result in a flexible, highly applicable, HEDANN design system that would be 50 times faster than the embodiment described in section 6.2.

7.0 Unique Characteristics of the HEDANN Design Platform

7.1 Introduction

The HEDANN design platform is possibly the first evolvable hardware platform to provide unconstrained evolution of complex architecture ANN circuits in physical hardware. As such, there are a number of unique properties associated with this system that are not encountered with other evolutionary design systems. This chapter discusses the unique properties of the HEDANN design platform and highlights challenges associated with this new approach.

7.2 Noise

In traditional, computer-based, genetic algorithms, the evolution of an ANN design begins with the creation of a population of designs. The response of each design to a series of test vectors is typically simulated, and each design's fitness is calculated based on the simulated response. Under these conditions, a design's fitness can be a very precisely defined value. For example, the top five designs in a population might have fitness values as shown in Table 5.

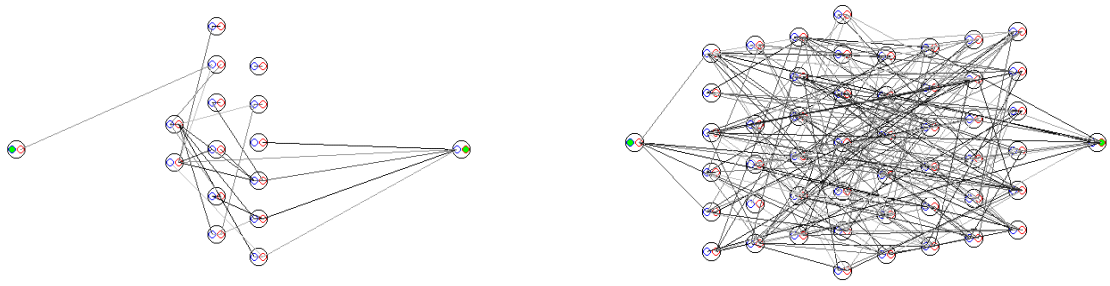
With each of the values in Table 5, the error, or uncertainty, is zero. This means that if the simulations were repeated, the exact same fitness values would be recorded for each corresponding design. Because the computational conditions of the original simulation can be exactly duplicated, the simulated ANNs can repeatedly produce a precise and predicable response to a set of defined input vectors.

Table 5. Typical Fitness Values in Computer Evaluated GAs

Design Number	Fitness Value
1	0.6981
2	0.6980
3	0.6960
4	0.6855
5	0.6743

When using actual hardware to implement and evaluate ANN designs, this level of certainty is not always obtainable. Because physical hardware designs can be dependent on environmental conditions, stochastic circuitry, differences in random number generation, starting clock cycles, etc., it is often impossible to exactly recreate the conditions of a prior test. Consequently, the same set of test vectors can often result in differing outputs, thereby producing a non-precise fitness value measurement. In addition, the degree of uncertainty associated with a fitness evaluation, and the magnitude of this uncertainty, may vary from design to design.

As an example of this effect, we can look at two randomly generated designs, called design A and B. The graphical representation of each design is shown in Figure 59. Both designs have one input and one output. Because the HEDANN design platform's genetic algorithm does not penalize an ANN design based on size, there is no pre-established preference on network size. Consequently, it is not uncommon to have networks, such as Design A (a 13-neuron network) and Design B (a 43-neuron network), competing to perform the same task.



a. **b.**

Figure 59. Sample ANNs for Noise Evaluation (a) Design A and (b) Design B

Both designs in Figure 59 were exposed to a series of sixty test vectors. The response of each design to these sixty test vectors established some arbitrary fitness values. Using these same sixty test vectors, the fitness evaluation was repeated 200 times, giving 200 different fitness measurements. From these 200 fitness measurements, a histogram showing the distribution (and thereby the uncertainty) of the design's fitness value was created. As is shown in Figures 60 and 61, both designs have approximately the same mean fitness. However, it is immediately obvious that the uncertainty of the fitness value for design B is considerable lower than for design A. An examination of the outputs of each design would have shown that design B had a more consistent response to all of the input test vectors, whereas design A appears to respond more randomly from test vector to test vector. Such inconsistencies are the result of a particular design's circuitry (i.e. number of recursive connections, near-zero weight values, etc.). Understanding why one ANN design displays more or less uncertainty than another can be extremely difficult to determine with a complex architecture ANN.

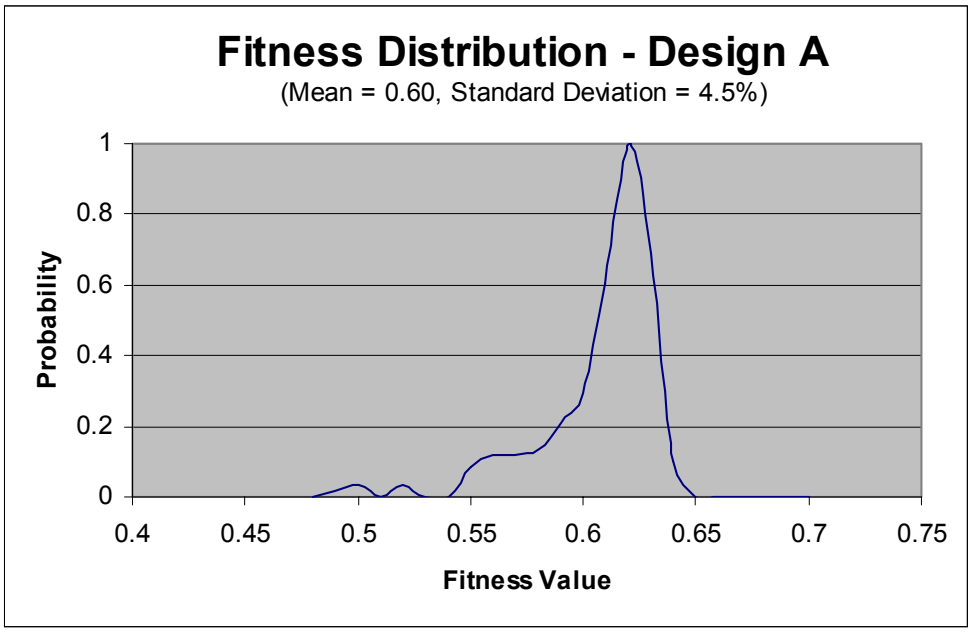


Figure 60. Measured Fitness Uncertainty in ANN Design A (sigma = 4.5%)

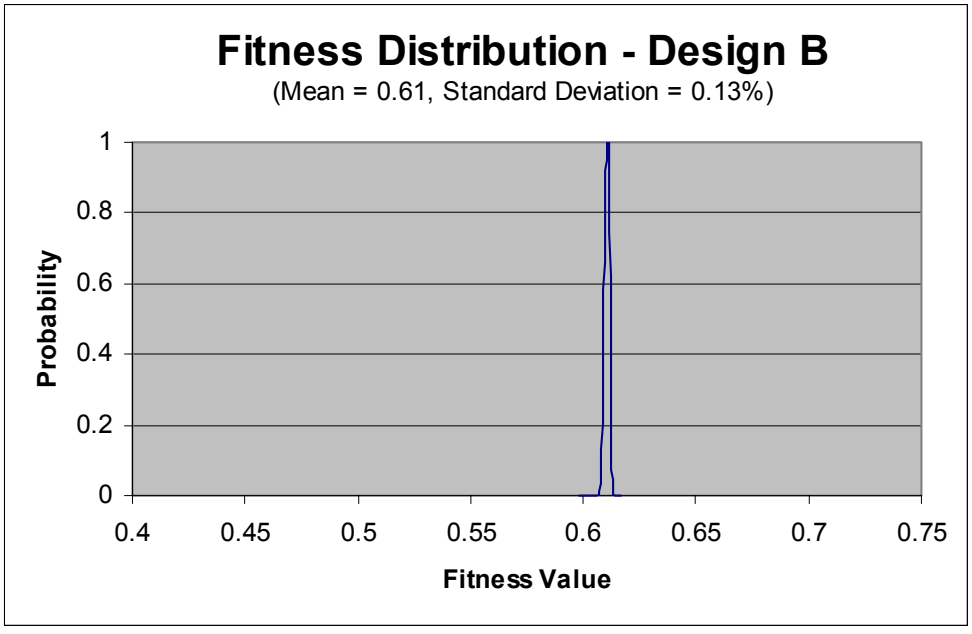


Figure 61. Measured Fitness Uncertainty in ANN Design B (sigma = 0.13%)

Consequently, it is difficult to develop a method for preventing the uncertainty variations from design to design without overly constraining the architecture of the ANN. As such, the research presented in the dissertation considered these variations to be an unavoidable feature of the HEDANN design platform and made no attempt to prevent their occurrence.

However, to reduce the uncertainty of an individual design's fitness measurement, it was possible to increase the number of test vectors supplied to a circuit under test. Unfortunately, the uncertainty only decreases as the square root of the number of test vectors. Therefore, if the number of test vectors is increased by a factor of 100, the fitness deviation would only be reduced by a factor of 10. Since increasing the number of test vectors lengthens the time required to evaluate a generation of designs, this solution produce only minimal improvements.

As a result of fitness uncertainty associated with the HEDANN design platform, the maximum fitness of an evolving population will not increase steadily as with most evolutionary techniques. Figure 62 shows a typical evolutionary trend encountered with computer-based and simulated GAs. The maximum fitness of an evolving population is always increasing, where the derivative of the fitness curve (or rate of evolution) is always positive.

However, the maximum fitness of a population evolved with the HEDANN design platform will more closely resemble Figure 63. In Figure 63, the rate of evolution is observed to have both positive and negative slopes.

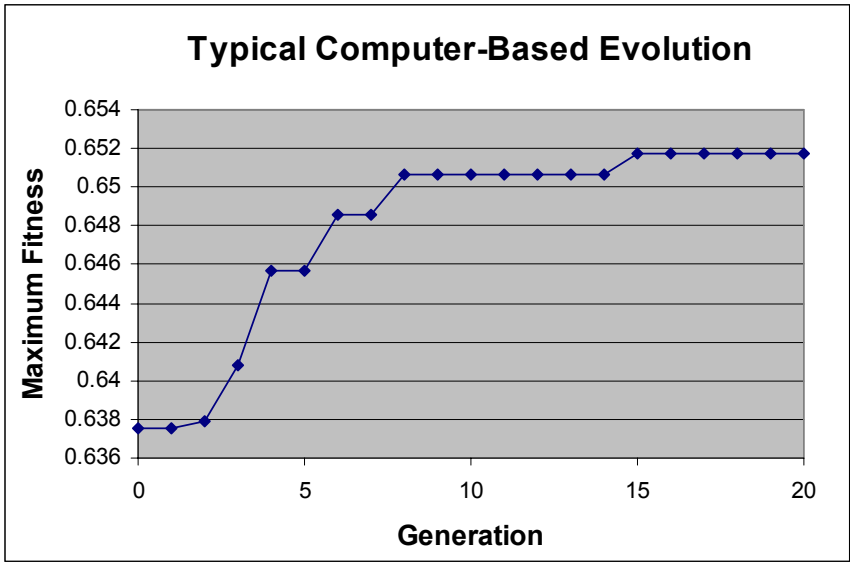


Figure 62. Traditional Improvements in Population Fitness with Increasing Generations

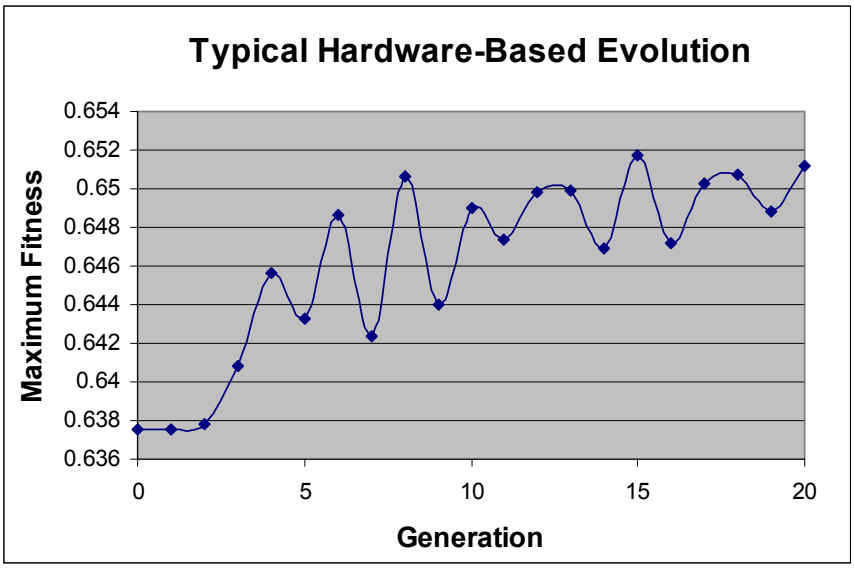


Figure 63. HEDANN's Characteristic Fluctuating Improvement in Population Fitness with Increasing Generations

Despite these fluctuations, the maximum fitness of the evolving population in Figure 63 does improve and will eventually allow for an acceptable design to be evolved. However, the fluctuations do create some very interesting effects. First, the uncertainty in fitness measurement can easily obscure small improvements in design performance. Therefore, if a design improvement creates an increase in design fitness that is smaller than the fitness noise margin, it is very possible that this improvement will not be recognized by the HEDANN design platform and the improved design will not propagate to future generations. As a rule, if the incremental improvements in design performance are below the fitness noise margin, the rate at which the population evolves will be zero. Therefore, evolutionary trends associated with the HEDANN design platform often exhibit periods of stalled evolution (i.e. zero rate of increased fitness) followed by periods of "punctuated" evolution, where large increases in design fitness result from substantial mutations within a population.

Interestingly, according to fossil evidence, Nature is known to exhibit similar evolutionary trends. Obviously, Nature's process for evaluating the fitness of organisms is also inherently noisy. Therefore, the punctuated evolution observed in Nature may result from a similar noise-related effect. The one major difference is that Nature typically evolves populations contain large numbers of members. A large population size can allow subtle design improvements, expressed throughout a population, to be more precisely recognized and selected. Likewise, increasing population size in the HEDANN design system was observed to help in reducing the fluctuations shown in Figure 63.

7.3 Initial Population Size

Accidentally selecting and propagating a poor design, which might appear to be a high performer due to favorable noise effects, can generally slow an evolutionary process. However, making a poor selection in the first generation can have a devastating effect on the success of an evolutionary run. If poor designs are selected during the first generation, the entire evolutionary process will be constrained to improvements upon these poor designs. Consequently, it may take a considerable amount of time to evolve these complex and inconsistent designs into high fitness solutions. Due to the noise effects discussed previously, and given the fact that an initial population of randomly generated designs often possesses fitness values that are closely ranked, it can be difficult for a GA to discriminate between good designs and bad designs in early generations. Therefore, if the population size is small, there is a higher probability that many poor designs will be chosen for propagation into the next generation, eventually resulting in inefficient and often stagnated evolution.

To avoid this problem, it is helpful to start with a large initial population. A large initial population improves the chances of creating randomly generated designs that perform observably well, exhibiting a fitness value greater than the fitness noise margin. In addition, a large population allows poor performers to be more easily absorbed. Potentially, if the fitness noise were well characterized for an initial population, a rule could be developed that would ensure only top-performing designs were selected. However, for the research presented in this dissertation, no such rule was imposed. Rather an initial population size of 1000 was chosen for early generations, and this

population size was eventually reduced to 100 after the second generation. Results appear to indicate that this approach was generally effective.

7.4 Sensitivity to Environmental Factors and Device Dependency

The evolved design-time and run-time re-configurable ANN circuits presented in Chapters 8 and 9 achieved maximum fitness values of 0.86 and 0.48 respectively. To be more precise, the fitness values of each design were measured to be 0.86 ± 0.014 and 0.48 +/- 0.008 respectively, at room temperature. It was argued, in Chapter 3, that designs evolved using the HEDANN design platform would perform reliably throughout the typical digital logic operating temperature rangee and would perform properly regardless of which FPGA device was used for implementation.

To test the environmental sensitivity of the evolved designs, each of the final evolved designs were downloaded to their corresponding FPGA prototype board and exposed near the limits of the board's operating temperatures. For both boards, the ambient temperature limits were 0° to 50° C. The boards were tested at 5° C and 40° C. Table 6 shows the performance of the evolved designs at these temperature extremes. For the design-time re-configurable circuit, which functioned as a frequency recognition circuit, all environmental testing was done with an external clock located outside the environmental test chamber (i.e. the clock was kept at room temperature). This allowed the temperature sensitivity of the FPGA circuitry to be isolated from the rest of the prototype board components. With the clock at room temperature, the performance of the circuit was consistent regardless of FPGA board temperature.

Table 6. Environmental Sensitivity of Evolved Design Performance

Evolved Design Name	Room Temperature	5° C	40° C
Design-Time Re-configurable ANN	0.863 ± 0.014	0.845 ± 0.016	0.854 ± 0.011
Run-Time Re-configurable ANN	0.484 ± 0.008	0.484 ± 0.009	0.484 ± 0.013

Interestingly, if the clock was allowed to vary with FPGA temperature, the performance of the evolved frequency recognition circuit declined as the temperature deviated from room temperature (22°C). A change in average fitness of approximately 0.2% for every 1°C deviation from room temperature was observed. Although not terribly significant, this trend does indicate a slight dependency of the evolved circuit on clock stability. No such dependency was measured for the run-time re-configurable circuit, which functioned as a static shape recognition circuit.

To test the device dependency of each evolved design, each design's VHDL file was recompiled to target an Altera STRATIX FPGA (EP1S25F780) containing approximately 25,000 logic elements. The STRATIX device family has a device architecture that is considerably different from the FLEK 10K and APEX 20K device families used to initially evolve each design. Therefore, the performance of each evolved circuit when implemented on this device would represent a good test of the evolved design's device dependency. Table 7 describes the performance of the evolved designs when implemented on the STRATIX device.

Table 7. Device Dependency of Evolved Design Performance

Evolved Design Name	FLEX 10K Device	APEX Device	STRATIX Device
Design-Time Re-configurable ANN	0.863 ± 0.014	N/A	0.858 ± 0.025*
Run-Time Re-configurable ANN	N/A	0.484 ± 0.008	0.479 ± 0.010**

*External Clock Used
**Fitting Was Optimized to Reduce Area Usage

As is evident in Tables 6 and 7, the designs evolved with the HEDANN design platform perform independently of device family and ambient temperature. Other environmental factors and device families could have been tested but it should be apparent from the circuitry presented in Chapter 4 that these designs are constrained to the same rules as traditional digital designs targeting FPGAs. Therefore, design portability and operational reliability throughout the traditional industrial temperature range should be likely.

8.0 Conclusions

8.1 Contributions

This dissertation has demonstrated a design methodology that captures all three of the "essential components" of natural design. Using FPGA technology to implement digital artificial neurons in hardware, the HEDANN design platform has demonstrated the ability to physically evolve complex architecture ANNs for the purpose of solving challenging problems. The solution of two AI related problems demonstrated how, in a relatively small number of generations and with a relatively small number of neurons, the HEDANN design platform could evolve useful designs using one of two re-configurable ANN circuits (design-time or run-time re-configurable). Reminiscent of natural design, the evolved designs exhibited efficient use of network resources and, generation-wise, evolved relatively quickly. In addition, the evolved circuits were demonstrated to exhibit device independent, environmentally robust performances. This insensitivity to environmental variations is essential to the development of circuit designs suitable for commercial use. Given further improvements in software and hardware, this approach could one day be used for developing solutions to significantly more challenging and complex AI problems.

As was discussed and demonstrated in this dissertation, the HEDANN design platform not only has the potential to create ANN circuitry for AI applications in general, but also is a tool that can be used to:

- Assist ANN researchers exploring complex architecture ANNs using advanced neuron models.
- Assist GA researchers exploring indirect encoding strategies for ANNs.
- Assist EHW researchers in creating environmentally robust, device independent, complex behavior, evolved circuitry.

8.2 Future Research

A number of technical hurdles still exist in implementing this platform that must be addressed in future research. The major technical hurdle limiting the application of the HEDANN design platform is related to preserving the flexibility of the platform while also reducing the amount of time required in evaluating a generation of designs. As was mentioned previously, the trial problems posed in this dissertation required two to three months to solve. Obviously, this amount of time is unacceptable for general use and must be improved considerably, especially when considering that more complex problems will almost certainly require the evaluation of many more generations. In the near term, optimization of the run-time re-configurable HEDANN design platform, for certain amenable applications, could be used to significantly reduce evaluation times (as demonstrated in Section 6.4).

Future research on the HEDANN design platform, featuring the run-time re-configurable ANN circuit, should focus on increasing the bandwidth between the FPGA and the computer responsible for serial programming of the FPGA implemented designs, and on increasing the ANN system clock from 1.25 MHz to 80 MHz or greater. These improvements would be expected to increase the speed of the HEDANN design platform,

with run-time re-configurable ANN circuitry, by a factor of at least fifty and without compromising platform flexibility. In addition, new, higher density FPGAs have become available since the start of this research and could be targeted to permit the evolution of larger-sized, run-time re-configurable ANN circuits.

To improve the speed of the HEDANN design platform, featuring a design-time re-configurable ANN circuit, a more profound change would be required. Because the design-time approach is computationally intensive, the speed at which designs can be synthesized and placed-and-routed must be improved. Realistically, the speed of these processes can only be improved through increased computing speeds or parallel processing. Therefore, development of an FPGA design tool that can be executed on a massively multi-processor computer, such as Oak Ridge National Laboratory's Phoenix super-computer, should be sought. It is estimated that the speed of the HEDANN design platform, using design-time re-configurable ANN circuitry, could be increased by approximately 100 to 200 times if the ORNL Phoenix super-computer and required FPGA design tools were utilized.

At the time of this writing, computing speeds and gate densities of commercial microchips continue to enjoy geometric improvements, as they have for the past 30 years. Moore's Law, which states that digital technology will double in speed or density every 18 months, continues to be a viable trend and is expected to continue for the near future. If this trend continues throughout the century, it would be possible by 2050 to implement a complex-architecture ANN circuit in an FPGA, using the design-time re-configurable ANN circuitry, containing the same number of neurons as the human brain. Likewise, by 2090, it would be possible to realize an ANN circuit, using the run-time re-configurable

ANN circuitry, containing the same number of neurons as the human brain. Unlike past efforts, these FPGA-implemented devices would truly be capable of evolving artificial neural networks with architectures as complex as biological processors. As has been argued throughout this dissertation, it is the physical evolution of these complex architecture ANNs that may eventually lead to an "intelligent" processor with the skills needed to once again revolutionize our world.

REFERENCES

[1]R. Beale and T. Jackson, *Neural Computing, an Introduction*. Bristol: Adam Hilger, IOP Publishing Ltd, 1990.

[2]R. Hecht-Nielsen, *Neurocomputing*. Redwood City: Addison Wesley, 1990.

[3]J. Hertz, J., A. Krogh, and R. Palmer, *Introduction to the Theory of Neural Computation*. Redwood City: Addison-Wesley, 1991.

[4]J. A. Freeman and D. M. Skapura, *Neural Networks – Algorithms, Applications, and Programming Techniques*. Massachusetts: Addison-Wesley, 1992.

[5]X. Yao, "Evolutionary artificial neural networks," *International Journal of Neural Systems*, vol. 4, no. 3, pp. 203-222, 1993.

[6]H. de Garis, "An Artificial Brain: ATR's Cam-Brain Project Aims to Build/Evolve An Artificial Brain with a Million Neural Net Modules Inside a Trillion Cell Cellular Automata Machine," *New Generation Computing Journal*, vol. 12, no. 2, July 1994.

[7]N. Nawa, M. Korkin, and H. de Garis, "ATR's CAM-Brain Project: The Evolution of Large-Scale Recurrent Neural Network Modules," in *1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98),* 1998.

[8]K. Balakrishnan, and V. Honavar, "Evolutionary Design of Neural Architectures -- A Preliminary Taxonomy and Guide to Literature," Department of Computer Science, Iowa State University, Iowa, USA, Tech. Rep. CS TR95-01, 1995.

[9]M. Mitchell, *An Introduction to Genetic Algorithms*. Cambridge: MIT Press, 1996.

[10]M. D. Vose, *The Simple Genetic Algorithm: Foundations and Theory*. Cambridge, MIT Press, 1999.

[11]D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Redwood: Addison-Wesley, 1989.

[12]J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor: The University of Michigan Press, 1975.

[13]Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*. Berline: Springer-Verlag, 1996.

[14]W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic Programming -- An Introduction: On the Automatic Evolution of Computer Programs and its Applications*. San Francisco: Morgan Kaufmann Publishers, 1997.

[15]D. B. Fogel, *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. Piscataway: IEEE Press, 1999.

[16]J. R. Koza, F. H. Bennett, D. Andre, and M. A. Keane, *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco: Morgan Kaufmann, 1999.

[17]H. P. Schwefel, *Evolution and Optimum Seeking*. New York: John Wiley & Sons, 1995.

[18]J. R. Koza, *Genetic Programming*. Cambridge: The MIT Press, 1992.

[19]T. Bäck, *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. New York: Oxford University Press, 1996.

[20]D. Whitley, "The GENITOR Algorithm and Selective Pressure: Why Rank-based Allocation of Reproductive Trials is Best," in *3rd ICGA*, J. D. Schaffer, Ed. San Mateo: Morgan Kaufmann, 1989.

[21]K. Vekaria and C. Clack, "Genetic Programming with Gene Dominance," in *Late Breaking Papers at the Genetic Programming 1997 Conference*, J. Koza Ed. Stanford: Stanford University Bookstore, 1997.

[22]P. Koehn, "Combining genetic algorithms and neural networks: The encoding problem," M.S. thesis, The University of Tennessee, Knoxville, TN, USA, 1994.

[23]F. Gruau, "Automatic definition of modular neural networks," *Adaptive Behavior*, vol. 3, no. 2, 1994.

[24]L. Marti, "Genetically Generated Neural Networks II: Search for an Optimal Representation," in *IJCNN'92*, 1992.

[25]J. R. Koza and J. P. Rice, "Genetic Generation of both the Weights and Architecture for a Neural Network," in IJCNN-91, 1991.

[26]D. B. Fogel, L. J. Fogel, and V. W. Porto, "Evolving Neural Networks," *Biological Cybernetics*, vol. 63, pg. 487-493, 1990

[27]M. Srinivas and L. M. Patnaik, "Learning Neural Network Weights using Genetic Algorithms - Improving Performance by Search-Space Reduction," in *International Joint Conference on Neural Networks*, 1991.

[28]G. F. Miller, P. M. Todd, and S. U. Hedge, "Designing neural networks using genetic algorithms," in *3rd ICGA*, J. D. Schaffer, Ed. San Mateo: Morgan Kaufmann, 1989.

[29]R. Zeidman, *Designing with FPGAs and CPLDs*. Kansas: CMP Books, 2002.

[30]A. Thompson, 1996. *Silicon evolution*. in *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Ed. Cambridge: MIT Press, 1996.

[31]J. F. Miller, P. Thomson, and T. C. Fogarty, "Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study," in *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*, D. Quagliarella, J. Periaux, C. Poloni, and G. Winter, Ed. New York: John Wiley & Sons, 1997.

[32]T. C. Fogarty, J. F. Miller, and P. Thomson, "Evolving Digital Logic Circuits on Xilinx 6000 Family FPGAs," in *Soft Computing in Engineering Design and Manufacturing*, P.K. Chawdhry, R. Roy, and R. K. Pant, ED. London: Springer-Verlag, pages 299-305, 1998.

[33]D. Roggen, S. Hofmann, Y. Thoma, and D. Floreano, "Hardware Spiking Neural Network with Run-Time Reconfigurable Connectivity in an Autonomous Robot," in *Proceedings 2003 NASA/DoD Conference on Evolvable Hardware*, 2003, pp 189-198.

[34]S. W. Moon and S. G. Kong, "Block-based Neural Networks," in *IEEE Transactions on Neural Networks*, vol. 12, no. 2, pp. 307-317, 2001.

[35]H. Lipson and J. B. Pollack, "Automatic design and Manufacture of Robotic Lifeforms," *Nature*, vol. 406, pp. 974-978, 2000.

[36]V. Salapura, M. Gschwind, and O. Maischberger, "A fast FPGA implementation of a general purpose neuron," in *Proceedings of the Fourth International Workshop on Field Programmable Logic and Applications*, 1994.

[37]M. Van Daalen, J. Zhao, and J. Shawe-Taylor, "Real Time Output Derivatives for On Chip Learning using Digital Stochastic Bit Stream Neurons," *Electronics Letters*, vol. 30, no. 21, pp. 1775-1777, 1994.

[38]M. Van Daalen, T. Kosel, P. Jeavons, and J. Shawe-Taylor, "Emergent activation functions from a stochastic bit stream neuron," *Electronics Letters*, vol. 30, no. 4, pp. 331-333, February 1994.

[39]M. Van Daalen, P. Jevons, and J. ShaweTaylor, "A stochastic neural architecture that exploits dynamically re-configurable FPGAs," in *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, 1993, pp. 202-211.

[40]J. G. Eldredge and B. L. Hutchings, "Density enhancement of a neural network using FPGAs and run-time reconfiguration," in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1994, pp. 180-188.

[41]S. Bade and B. L. Hutchings, "FPGA based stochastic neural network implementation," *Proc. IEEE workshop on fpgas for custom computing machines*, 1994, pp.189-198.

VITA

The author received a Bachelor or Science Degree in Engineering Physics in 1993 and a Master's of Science Degree in 1997, both from the University of Tennessee. He currently works in the Engineering and Science Technology Division at Oak Ridge National Laboratory.