25098580

THESIS

This is to certify that the

thesis entitled

Modeling Artificial Neural Networks
Using VHDL

presented by

Keshavachandra, C.K.

has been accepted towards fulfillment
of the requirements for

Master's degree in Electrical Engineering

[illegible]
Major professor

Date Aug. 9, 1990

# MODELING ARTIFICIAL NEURAL NETWORKS USING VHDL

By

Keshavachandra, C.K.

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Electrical Engineering

1990

# ABSTRACT

# MODELING ARTIFICIAL NEURAL NETWORKS USING VHDL

By

Keshavachandra, C.K.

This thesis describes an Artificial Neural Network (ANN) coprocessor modeled behaviorally using VHSIC Hardware Description Language (VHDL).

There has been renewed interest in the area of ANN of late. Everyday new ANN models for new domains are being suggested. There is however a dearth of equal progress in this field as far as hardware implementation is concerned. This is partially due to limitations posed by the current technology and it's orientation towards traditional VonNeuman architectures. It is a logical step to adopt VHDL as the design test bench for behavioral modeling of Artificial Neural Networks considering the fact that VHDL is gaining ground as the standard design test bench for hardware design. It is fast becoming the industry standard for hardware design, simulation and exchange.

One such system modeled in VHDL is described in this report. Even though the system is aimed at solving dynamic programming problems, it is designed such that any similar ANN can be modeled using it. The design structure facilitates easy modification in order to incorporate new features, such as learning, and different models for a neuron. This coprocessor can be used to test any ANN model and the corresponding energy function.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# Chapter 1

# Introduction

## 1.1 Introduction

The past few years have witnessed an increased interest in the area of Artificial Neural Networks (ANN) [1]. Research in this field, although started in early 60s, could not proceed at the desired pace due to the limitations posed by available technology. The phenomenal advances in the fields of computer engineering and IC engineering have given a fresh impetus to this field. These technological leaps have also opened up a myriad applications for Artificial Neural Nets [2, 3].

In spite of these factors, research in this area is still impeded by the fact that there is no standard hardware testbench to test new designs. Simulation and verification of most of the networks suggested are done on traditional computers in software. Designing and fabricating hardware that can be used for research is still prohibitively expensive. This problem, when considered with an academic setup in perspective, appears to be one of the bottlenecks in ANN research.

Another notable development of the past few years is the emergence of Design Automation (DA) tools, and Hardware Description Languages [4]. VHSIC Hardware Description Language (VHDL) is fast becoming an industry standard for hardware design and exchange [5]. The merits of using such a language for hardware design and verification are explained in Chapter 2. Apart from the immediate advantage of using an emerging standard for hardware design, VHDL provides an economical alternative for hardware design and verification. A premise of this work is that ANN research, impeded by the huge cost required for physical implementation, will find VHDL ideal for modeling and verification. Using this approach the actual implementation of these networks can be separated from much of the research effort.

## 1.2 Objective

This thesis is an effort in demonstrating VHDL as a viable design test bench for ANN research. A conscious decision was taken to develop a general purpose ANN coprocessor in VHDL, even though the primary objective was to model a neural network in VHDL capable of solving dynamic programming problems. The system is designed as a "coprocessor" similar to a math coprocessor in the sense that it receives data from the CPU and returns the solution or sends a signal that it could not converge to a solution within the predefined accuracy. This system can be used to test any similar network without major changes by describing the size of the network (in terms of the number of stages and the number of states in each stage) and providing the initial conditions (the weights on the links in the network). The design is also kept flexible enough for incorporation of additional features such as learning. One can use subcomponents of the present system in an alternative design without much effort. Essentially, this is a skeletal system that can be enhanced and used for simulation and verification of many types of Artificial Neural Networks.

# *Chapter 2*

# *VHDL*

## 2.1 Introduction

Early in the Department of Defence (DoD) Very High Speed Integrated Circuits (VHSIC) program, a need was felt for a standard medium of expression to communicate the massive amounts of design data associated with device designs of the desired scale and complexity. VHSIC Hardware Description Language (VHDL) is the outcome of efforts in this direction [6]. It is fast becoming industry standard for hardware design and exchange [5].

In the present day hardware design environment, where DA tools are used virtually in all phases of design cycle, hardware description languages are crucial to the design and test of hardware. They allow incremental development of designs, store design data, and communicate those data between various design activities. VHDL marks the first coordinated effort to develop a common hardware description language, that is being recognized as an industry standard [7].

## 2.2 Factors Influencing the Development of VHDL

Improving the documentation of electronic systems: Government electronic systems require stringent documentation because they have long life cycles and are deployed around the world. Maintaining and upgrading electronic systems while they are an active part of the inventory requires detailed, up-to-date, and accurate documentation. Hence there was a need for a precise hardware description language.

Since VHDL can serve as a design automation tool interface, it can document the electronic system during (instead of after) the design process. Therefore, VHDL more accurately reflects a system's true properties and characteristics.

Decreasing system design time and cost: There is a need for a significant number of custom ICs to meet performance, reliability, and classification requirements that off-the-shelf ICs won't satisfy. Already in the $2 to $5 million range, development costs of advanced ICs must be reduced to economically meet future IC demands[6]. VHDL can reduce IC development time and expense by promoting repeated use of previous design investments, and by providing a vehicle for more efficient management of the design process among individual designers or organizations.

When considering the development of large electronic systems, the paradigm of design as an iterative process building new designs upon past designs is quite powerful. Similarities between this process and human learning provide a conceptual basis for knowledge-based design tools that improve with use. In addition, some business analysts forecast that a "redesign era" will emerge to fuel the next major semiconductor market, and that equipment manufacturers will upgrade their products to take advantage of VLSI technology[6].

By allowing for parameterized generic design components, VHDL simplifies the reuse of designs. Once a generic component has been designed, it can be reused by instantiating its parameters with values meeting given application requirements - a feature significantly reducing resources expended in complex electronic system development.

By providing many features to assist in design management and documentation by configuration control, VHDL helps to establish more structured policies and procedures for developing electronic systems. Similar to the specification and body concept in Ada, VHDL allows designers to define specifications representing design component interfaces separately from several associated bodies representing alternative component implementations. Use of the interface and associated bodies enables VHDL to support configuration management of top-down and bottom-up design methodologies. In addition, VHDL supports packages, this allows managers to establish common naming conventions, data types and convenient functions among designers by encapsulating descriptions with VHDL packages.

## 2.3 The Language

VHDL provides a standard textual means of description for hardware components at abstraction levels ranging from the logic gate level to the digital system level. It provides precise syntax and semantics for these hardware components, enabling design transfer both within and among organizations. The language is designed to be efficiently simulated and natural for hardware designers. In addition, it allows designers to represent information outside the primary range of language coverage.

Some of the building blocks and abstractions of the language are explained below. This section has been written with extensive reference to the paper by J.D. Nash and L.F. Saunders [7].

**2.3.1 Design Entities:** A design entity models hardware of any complexity. For example, it may model a logic gate, a flip-flop, a RAM or a computer system. A design entity is composed of an interface and one or more alternative bodies. The interface contains a set of definitions common to alternative bodies. The hardware entity's external view and specify communication channels between the design entity and the outside world are captured in such definitions. The entity's operating characteristics and conditions may also be described as part of its interface's definition. Each alternative body describes an alternative view of the hardware entity. For example, one body may describe a hardware entity's behavior while another body may describe its structure, decomposing the entity in terms of its subcomponent interconnections; a third body may model the entity's operations in terms of register-transfer microoperatons. There are no restrictions on the number of ways designers can view hardware entities. Alternative structural implementations of the same hardware entity can be modeled so as to enable evaluation of cost and speed factors. Similarly, both functional and physical structures of a hardware entity can be modeled. Each alternative body

is associated with the same interface and can make use of all definitions supplied in the interface.

**2.3.2 Interface Description:** A design entity's interface contains information common to its alternative bodies. A subset of this information (namely, the specification of ports and generics) is externally visible. When a design entity is used as a subcomponent in a higher level design entity, its interface must conform to that of the subcomponent. Externally visible interface information is used for such consistency checks.

*Ports* define communication channels between design entities and the outside world. A port definition involves description of its mode and type. The port's mode specifies the direction of information flow through the port. A port can be of mode in, out, inout or buffer. A port type specifies the set of values a port may assume. Port values may be represented by voltage levels, truth values, binary digits, or multiple logic values. Each of these sets is a type, and each may be an abstraction of the same underlying electrical phenomenon.

A design entity interface may also define *generics*. Such a design entity defines a class of components. When used, a generic design entity is particularized to select one component in the class. To particularize a generic design entity, desired values are supplied for corresponding generics. Generics increase a design entity's reusability. For example, technology dependencies such as noise margins or power consumption may be captured in generics. When design entities are used, a particular technology may be specified by supplying the necessary generic values. An example is given in Figure 2.1.

**2.2.3 Body Descriptions:** VHDL provides two body description types: architectural bodies and configuration bodies. Architectural bodies describe how the input and output ports of a design entity relate, either by expressing the involved input/output data transformation or by connecting those ports to subcomponents. Such predominantly local information pertains to one design hierarchy level. On the other hand, a configuration body contains global information such as which design entities model subcomponents used in an architectural body or how global signals are distributed.

```
entity Full_Adder is
   generic ( Time_Delay : TIME );
   port ( X,Y,Cin :IN Bit; Cout,Sum: OUT Bit );
end Full_Adder;
```

Figure 2.1. Interface description of an entity.

There are three styles of description within an architectural body: *structural*, *dataflow*, and *behavioral*. Structural descriptions capture the schematic view of hardware and consist primarily of interconnected components. Dataflow descriptions, a little more abstract, specify data transforms being performed in terms of concurrently executing RTL statements. Behavioral descriptions, the most abstract, specify data transforms in terms of algorithms for computing output responses to input changes.

A given architectural body may make use of any combination of these styles of descriptions, for they are all defined under a common set of semantics. Together, these features support most hardware design styles. An example is given in Figure 2.2.

Component instances in structural descriptions are placeholders for behavioral information specified as separately described design entities. In the absence of contrary information, we assume a separate design entity to have the same characteristics as the component being instantiated (the same name, for example, plus ports and generics with the same names, types, and compatible modes). Thus, if a design is being created bottom-up and the user wants to declare a component whose instances exhibit a given behavior, he need only copy the name, port declarations, and generic declarations of an existing design entity exhibiting the required behavior. Similarly, if a design is being created top-down and the user wants to define a design entity that implements the behavior required for a given component, he need only copy the name, port declarations, and generic declarations from that component's declaration to create the design entity's interface description.

```
-- Behavioral model of a Full Adder
architecture behavior of Full_Adder is
begin
 Process(A, B, Cin)
 begin
  Sum <= ( A xor B xor Cin ) after Time_Delay;
  Cout <= ( A and B ) or ( B and Cin ) or
            ( Cin and A) after Time_Delay;
 end process;
end behavior;
```

```
-- Structural model of a Full Adder
architecture structure of Full_Adder is

   Component Half_Adder
       port (I1,I2:in Bit;S,C:out Bit);
   end component;

   Component Or_Gate
       port (I1,I2:in Bit;O:out Bit);
   end component;

Signal S1,C1,C2:Bit;

begin
   X1: Half_Adder port map (X,Y,S1,C1);
   X2: Half_Adder port map (S1,Cin,Sum,C2);
   X3: Or_Gate port map (C1,C2,Cout);
end My_Full_Adder;
```

Figure 2.2. Different architectural bodies for the same entity Full Adder.

*Configuration specifications* provide the ability to override default association rules so that an architectural body's component instances may be bound to similar but not identical design entities. The names and port types may be different, in which case the configuration specification must identify appropriate type conversion functions. Moreover, additional signals may be connected to formal design entity ports that do not correspond to ports of the component.

Although configuration specifications may appear in either architectural or configuration bodies, they become most useful in the latter. A configuration body of a given design entity relates to an architectural body of the same entity; it's configuration specifications relate to it's component instances.

## 2.4 Impact of VHDL

The steadily increasing level of integration has motivated a growing emphasis on design automation and semicustom/custom ICs. The dependency of continued growth of the semiconductor industry and the nature of the IC market on the maturation rate of design automation and semicustom/custom technology, which in turn depend mainly on standardization and legal copyright protection, indicate that not everything can be or should be standardized. However, the lack of appropriate standards to guide and focus the growth of a technology can foster costly and burdensome diversity. Hence, there is a need for design, test, and manufacturing standards to establish interoperability and required interfaces. VHDL aims at filling this void.

As the electronic design process becomes increasingly dependent on automation tools, IC designing firms will develop proprietary tools to maintain a competitive edge. Many companies won't depend completely on closed and inflexible vendor design automation systems. On the other hand, most companies cannot attract the expertise or afford the sizable resources required to develop their own custom design automation systems.

While existing CAE environments provide excellent capabilities in specialized areas, in general they do not contribute to custom design automation system integration. By making these CAE environments provide interfaces with a standard such as VHDL, design exchange and CAE environment interoperability can be realized.

As the sophistication of the DA Tools being used is increasing, and as VHDL is fast becoming an industry standard for hardware design and exchange, many CAD vendors are coming out with compilers which translate the structural design in VHDL to an intermediate format, such as CIF or EDIF, with which an IC can be fabricated. Although the current versions can accomplish this only when the design is atleast at the Register Transfer Level (RTL), there are signs of VHDL growing into a full fledged *Silicon Compiler* - any hardware designer's dream.

As IC complexity increases, circuits become more specialized and their broad applicability decreases. It is estimated that about half the total IC market would be custom and semicustom ICs, by 1991 [6]. In this scenario, VHDL can play a major role in providing a clearly defined interface to customers of varying experience and sophistication to shorten development cycles, reduce costs, and avoid expensive legal proceedings resulting from design specification misunderstandings between vendor and customer. It can provide an elegant user documentation method for the difficult task of documenting custom/ semicustom designs. Another interesting offshoot is the phenomenon of design second sourcing.

Historically, design has been an art rather than a science. Starting with sometimes vague and incomplete specifications, designers go through an iterative series of transformations until systems can be built within given technologies - or until it is clear that intended functional behavior, performance goals, or design constraints are not feasible. There is a need for a top-down design approach, with all the specifications available at the outset and then trying to implement it physically. It translates to having a behavioral description of a system at the beginning and then implementing it structurally, in the VHDL design paradigm. Thus, VHDL can play in important role in advancing electronic system design the above stages to form a science of design by providing a economical vehicle to do the same.

VHDL has a crucial role to play in an academic environment in terms of educational value, propagating a science of design, and as an economical hardware design test bench. VHDL serves as a vehicle for investigating new approaches to design techniques, models, and automation in areas such as test, synthesis, and simulation. Knowledge about hardware properties and characteristics applicable to design is the very essence of language constructs comprising VHDL. A system can be first modeled behaviorally, verifying the correctness of the design, then modeled structurally testing the feasibility of hardware realization by incorporating the current technology constraints into the design.

## 2.5 Suitability of VHDL for ANN Implementations

In order to test various theories and hypotheses propounded in the field of Artificial Neural Networks, a "true" neural network is needed rather than a software simulation. ICs must be fabricated with "neurons" and interconnections built-in. (In the cases of programmable interconnections, the size of the network is dictated by the connectivity). Some of the major impediments to this are: 1) It is still with an empirical knowledge that a neuron has to be modeled; 2) Costs involved are enormous which is especially critical, considering that major portion of current research is being done in Universities; and 3) These ICs cannot be used to model different networks of realistic size, since a generic network would require complete connectivity with ways of programming the inter-neuron links.

VHDL merits a closer examination as an alternative for ANN implementations just by the fact that it is an economical and viable alternative. There is no incremental cost involved in modeling different networks; it can be made as close to hardware implementation as desired as against software approach. Limitations imposed by the existing technology can be circumvented by using VHDL. One need not get lost in problems such as connectivity, die-size, etc. Research effort will be directed at solving the real problem on hand. These limitations can be looked into when the system is designed, tested and is ready to be used.

When VHDL is used as the implementation tool for ANN implementations, the research will have a cumulative effect as the design can be exchanged with out any problem. Earlier designs can be modified to suit the current needs, a bigger system can be built upon those developed earlier. This provides a language to the research community in which theories can be propounded, tested, challenged and verified.

With the idea of VHDL growing into a *silicon compiler* gaining currency, the ANN community would be one of the prime beneficiaries by using VHDL for their designs. It does not appear too far-fetched to imagine having a ANN chip fabricated directly after it is modeled and tested in VHDL.

# Chapter 3

# Dynamic Programming

## 3.1 Introduction

Dynamic programming is a useful mathematical technique for making a sequence of interrelated decisions. It provides a systematic procedure for determining the combination of decisions that maximizes overall effectiveness.

Dynamic programming is a general type of approach to problem solving; the particular equations used must be developed to fit each individual situation. There does not exist a standard mathematical formulation of "the" dynamic programming problem.Therefore, a certain degree of ingenuity and insight into the general structure of dynamic programming problems is required to recognize when problem can be solved by dynamic programming procedures and how it can be done.

## 3.2 Stagecoach Problem

This chapter is with extensive reference to the book by Hillier and Lieberman [8]

The stagecoach problem is an example especially constructed to illustrate the features and to introduce the terminology of dynamic programming. It concerns a mythical salesman who had to travel west by stagecoach about 125 years ago when there was a serious danger of attack by

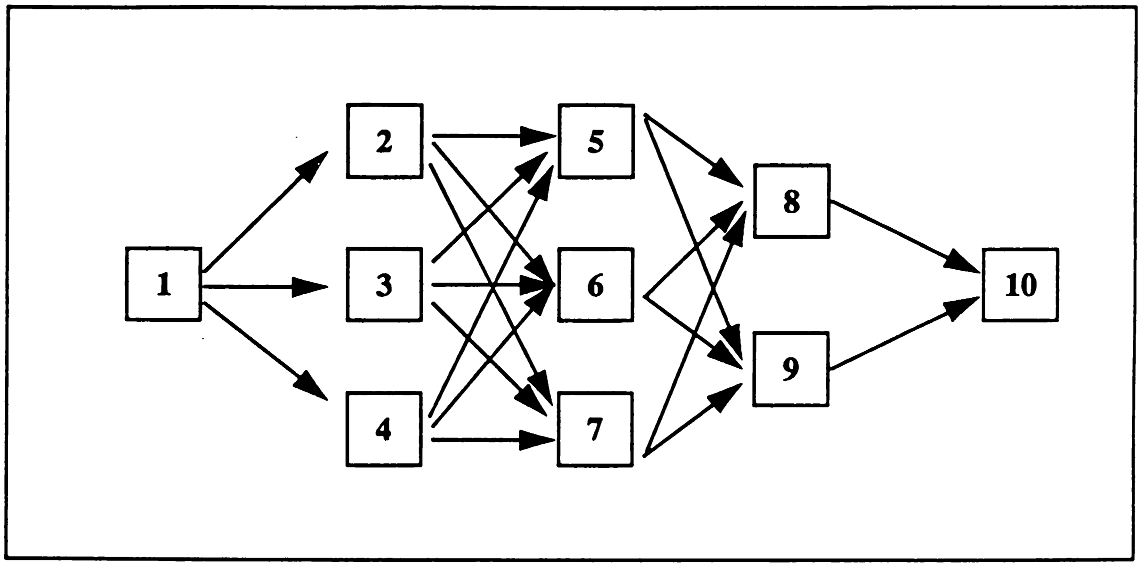


Figure 3.1. The road system for the stagecoach problem.

marauders. Although his starting point and destination were fixed, he had considerable choice as to which state (or territories that subsequently became states) to travel through en route. The possible routes are shown in Figure 3.1, where each state is represented by a numbered block. Thus four stagecoach runs(stages) were required to travel from his point of embarkation in state 1 to his destination in state 10.

This salesman was a prudent man who was quite concerned about his safety. After some thought, he came up with a rather clever way of determining the safest route. Life insurance policies were offered to stagecoach passengers. Because the cost of the policy for taking any given stagecoach run was based on a careful evaluation of the safety of that run, the safest route should be the one with the cheapest total life insurance policy. The cost for the standard policy on the stagecoach run from state *i* to state *j*, which will be denoted by $c_{ij}$, is as shown in Table 3.1.The objective now is to find the route that minimizes the total cost of the policy.

## 3.3 Solution to the Stagecoach Problem

First note that the shortsighted approach of selecting the cheapest run offered by each successive stage may not yield an overall optimal decision. Following this strategy would give the

Table 3.1. The cost for the standard policy on the stagecoach run from state *i* to state *j*.

| | 2 | 3 | 4 |
|---|---|---|---|
| 1 | *2* | *4* | *3* |

| | 5 | 6 | 7 |
|---|---|---|---|
| 2 | *7* | *4* | *6* |
| 3 | *3* | *2* | *4* |
| 4 | *4* | *1* | *5* |

| | 8 | 9 |
|---|---|---|
| 5 | *1* | *4* |
| 6 | *6* | *3* |
| 7 | *3* | *3* |

| | 10 |
|---|---|
| 8 | *3* |
| 9 | *4* |

route 1 - 2 - 6 - 9 - 10 at a total cost of 13. However, sacrificing a little on one stage may permit greater savings thereafter. For example, 1 - 4 - 6 is cheaper overall than 1 - 2 - 6. One possible approach to solving this problem is to use trial and error. However, the number of possible routes is large and having to calculate the total cost for each route is not an appealing task.

Dynamic programming provides a solution with much less effort than exhaustive enumeration. The computational savings are enormous for larger versions of this problem. Dynamic programming starts with a small portion of the original problem and finds the optimal solution for this smaller problem. It then gradually enlarges the problem, finding the current optimal solution from the preceding one, until the original problem is solved in its entirety. For the stagecoach problem, we start with the smaller problem where the salesman has nearly completed his journey and has only one more stage (stagecoach run) to go. The obvious optimal solution for this smaller problem is to go from his current state (whatever it is) to his ultimate destination (state 10). At each subsequent iteration, the problem is enlarged by increasing by one the number of stages left to go to complete the journey. For this enlarged problem, the optimal solution for where to go next from each possible state can be found relatively easily from the results obtained at the preceding iteration.

Let the decision variables $x_n$ where n = 1,2,3,4 be the immediate destination on stage n (the $n^{th}$ stagecoach run to be taken). Thus the route selected is 1 - $x_1$ - $x_2$ - $x_3$ - $x_4$ where $x_4 = 10$. Let $f_n(s,x_n)$ be the total cost of the best overall policy for the remaining stages, given that the salesman is in state s ready to start stage n and selects $x_n$ as the immediate destination. Given s and n, let $x_n^*$ denote the value of $x_n$ that minimizes $f_n(s,x_n)$, and let $f_n^*(s)$ be the corresponding minimum value.

Thus

$$f_n^*(s) = \min f_n(s,x_n) = f_n(s,x_n^*),$$

$$= c_{sxn} + f_{n+1}^*(x_n),$$

where the value of $c_{sxn}$ is given by the preceding tables for $c_{ij}$ by setting $i = s$ (the current state) and $j = x_n$ (the immediate destination). Because the ultimate destination (state 10) is reached at the end of stage 4, $f_5^*(10) = 0$. The objective is to find $f_1^*(1)$ and the corresponding route. Dynamic programming finds it by successively finding $f_4^*(s)$, $f_3^*(s)$, $f_2^*(s)$ for each of the possible states s and then using $f_2^*(s)$ to solve for $f_1^*(s)$.

By solving the stagecoach problem using the above algorithm, the optimal routes are found to be

1 - 3 - 5 - 8 - 10

1 - 4 - 5 - 8 - 10

1 - 4 - 6 - 9 - 10

They all yield a total cost of $f_1^*(1) = 11$.

Chapter 4

# Artificial Neural Networks

## 4.1 Introduction

Conventional digital computers are extremely good at executing sequences of instructions that have been precisely formulated for them, with the "stored program" representing the processing steps that need to be done. The human brain, on the other hand, performs well at such tasks as vision, speech, information retrieval, and complex spatial and temporal pattern recognition in the presence of noisy and distorted data - tasks that are very difficult for sequential digital computers to do. The brain accomplishes this, even though its "processing elements" (neurons) are significantly slower than the processing elements of contemporary supercomputers. In fact neurons, which are electrochemical devices, can respond in milliseconds, whereas current, off-the-shelf electronic technology can switch states in nanoseconds.

Current estimates place the number of neurons in the human brain at $10^{11}$[9]. They are organized in a complex, unknown interconnection structure, and an individual neuron may be connected to several thousand other neurons. There has been considerable research going on for quite some time to understand how such a network (biological neural network) is capable of storing data like images, smell, sensations and thoughts, allowing us to represent, retrieve and manipulate these data. There has been a concerted effort to duplicate such a network at different abstraction levels, creating what has come to be known as an Artificial Neural Network (ANN).

## 4.2 Hopfield - Tank Networks

Many logical problems arising from real world situations can be formulated as optimization problems. It can be described as a qualitative search for the best solution. In their landmark paper, J.J.Hopfield and D. W. Tank proposed a network topology, that has come to be known as Hopfield-Tank network. The Hopfield - Tank network consists highly-interconnected nonlinear analog neurons that can be used for solving optimization problems[1]. These networks can rapidly provide a collectively-computed solution (a digital output) to a problem on the basis of analog input information. The problems to be solved must be formulated in terms of desired optima, often subject to constraints.

The general structure of the analog computational networks which can solve optimization problems, as suggested by Hopfield and Tank is shown in Figure 4.1. These networks have the three major forms of parallel organization found in neural systems: parallel input channels, parallel output channels, and a large interconnectivity between the neural processing elements. The processing elements (neurons) are modeled as amplifiers in conjunction with feedback circuits

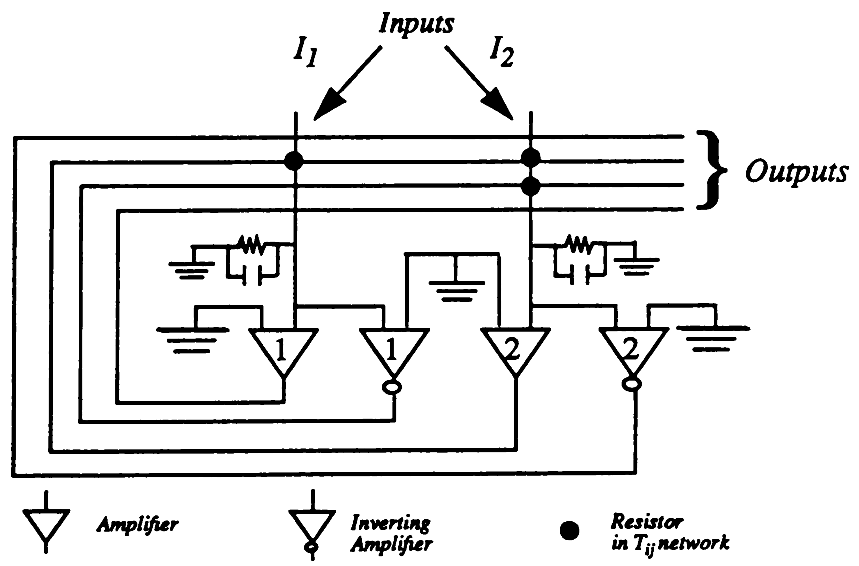


Figure 4.1. Hopfield - Tank Network.

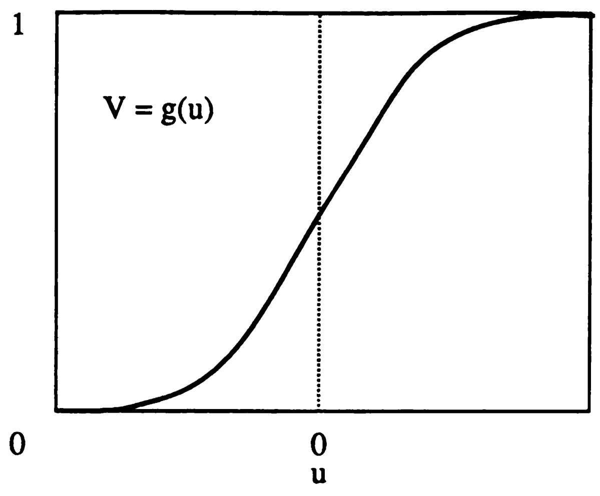


Figure 4.2. Transfer function of a neuron.

comprised of wires, resistors and capacitors organized so as to model the most basic computational features of neurons, namely axons, dendrites, and synapses connecting the different neurons.

The amplifiers have sigmoid monotonic input-output relations, as shown in Figure 4.2. The function $V_j = g_j(u_j)$ which characterizes this input-output relation describes the output voltage of amplifier $V_j$ due to an input voltage $u_j$. The time constants of the amplifiers are assumed negligible. However, like the input impedance caused by the cell membrane in a biological neuron, each amplifier $j$ has an input resistor $\rho_j$ leading to a reference ground and an input capacitor $C_j$. These components partially define the time constants of the neurons and provide for integrative analog summation of the synaptic input currents from other neurons in the network. In order to facilitate both excitatory and inhibitory synaptic connections between neurons while using conventional electrical components, each amplifier is given two outputs, a normal (+) output and inverted (-) output. The minimum and maximum outputs of the normal amplifier are taken as 0 and 1, while the inverted output has corresponding values of 0 and -1.

A synapse between two neurons is defined by a conductance $T_{ij}$ which connects one of the two outputs of amplifier $j$ to the input of amplifier $i$. This connection is made with a resistor of value $R_{ij} = 1/|T_{ij}|$. If the synapse is excitatory ($T_{ij} > 0$), this resistor is connected to the normal (+) output of amplifier $j$. For an inhibitory synapse ($T_{ij} < 0$), it is connected to the inverted (-) output of amplifier $j$. The matrix $T_{ij}$ defines the connectivity among the neurons. The net input current to any neuron $i$ (and hence the input voltage $u_i$) is the sum of the currents flowing through the set of resistors connecting its input to the outputs of the other neurons. Thus the normal and inverted output for each neuron allow for the construction of both excitatory and inhibitory connections using normal (positive valued) resistors; biological neurons do not require a normal and inverted output since exicitatory and inhibitory synapses are defined by use of different receptor/ion channel combinations.

As indicated in Figure 4.1, these circuits include an externally supplied input current $I_i$ for each neuron. These inputs can be used to set the general level of excitability of the network through constant biases, which effectively shift the input-output relation along the $u_i$ axis, or to provide direct parallel input to drive specific neurons.

Although this "neural" computational circuit is described here in terms of amplifiers, resistors, capacitors, etc., it has been shown that networks of neurons whose output consists of action potentials and with connections modeled after biological excitatory and inhibitory synapses could compute in a similar fashion to this conventional electronic hardware [1].

The equation of motion describing the time evolution of this circuit is

$$C_i( du_i / dt ) = \Sigma T_{ij} V_j - u_i / R_i + I_i ,$$

$$1 / R_i = 1 / \rho_i + \Sigma T_{ij} ,$$

and

$$V_i = g_i(u_i),$$

where $g_i$ is commonly a monotonically increasing sigmoid function.

The main task in solving a problem using an ANN is finding an energy function corresponding to the problem at hand, whose minima correspond to the solution to the problem. The

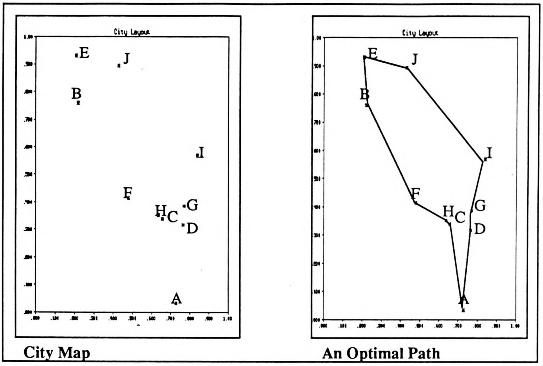


Figure 4.3. An example of a 10 - city TSP.

minima of the energy function can be found by using the above network.

## 4.3 Traveling Salesman Problem

In order to explain the ideas developed in the previous section, the Traveling Salesman Problem (TSP) is discussed below, explaining how it can be solved using a network similar to the one described in the previous section[1].

The TSP is a classic example of a difficult optimization problem. A set of n cities A, B, C, ... have (pairwise) distances of separation $d_{AB}$, $d_{AC}$, ......, $d_{BC}$, .... The problem is to find a closed tour which visits each city once, returns to the starting city, and has a short (or minimum) total path length. A tour defines some sequence B, F, E, G, .... , W in which the cities are visited, and the total path length d of this tour is

$$d = d_{BF} + d_{FE} + d_{EG} + ..... + d_{WB} .$$

The actual best solution to a TSP problem is computationally very hard - the problem is npcomplete, and the time required to solve this problem on any given computer grows exponentially with number of cities. An example of a 10-city TSP is given in Figure 4.3.

The solution to the n-city TSP problem consists of an ordered list of n cities, To "map" this problem onto the computational network, a representation scheme which allows the digital output states of the neurons operating in the high - gain limit to be decoded into this list, is needed. Hopfield and Tank have chosen a representation scheme in which the final location of any individual city is specified by the output states of a set of n neurons. For example, for a 10-city problem, if city A is in position 6 of the tour which is the solution to the problem, then this is represented by the sixth neuron out of a set of ten having an output with all other outputs at 0.

This representation scheme is natural, since any individual city can be in any one of the n positions in the tour list. For n cities, a total of n independent sets of n neurons are needed to represent a complete tour. This is a total of $N=n^2$ neurons. The output state of these $n^2$ neurons which we will use in the TSP computational network is most conveniently displayed as an n x n square array. Thus, for a 5-city problem using a total of 25 neurons, the neuronal state is shown in Figure 4.3 would represent a tour in which city C is the first city to be visited, A the second, E the third, etc. (The total length of the 5-city path is $d_{CA} + d_{AE} + d_{EB} + d_{BD} + d_{DC}$). Each such final state of the array of outputs describes a particular tour of the cities. Any city cannot be in more than one position in a valid tour (solution) and also there can be only one city at any position. In the n x n "square" representation this means that in an output state describing a valid tour there can be only one "1" output in each row and each column, all other entries being zero. Likewise, any such array of output values, called a permutation matrix can be decoded to obtain a tour (solution).

To enable the N neurons in the TSP network to compute a solution to the problem, the network must be described by an energy function in which the lowest energy state (the most stable state of the network) corresponds to the best path. This can be separated into two requirements. First, the energy function must favor strongly stable states of the form of a permutation matrix, rather than more general states. Second, of the n! such solutions, all of which correspond to valid tours, it must favor those representing short paths. An appropriate form for this function can be

found by considering the high gain limit, in which all final normal (+) outputs will be 0 or 1. The space over which the energy function is minimized in this limit is the $2^N$corners of the N-dimensional hypercube defined by $V_i = 0$ or 1. A suitable energy function would be

$$E = A/2 \sum_X \sum_i \sum_{j \neq i} V_{Xi} V_{Xj} +$$

$$B/2 \sum_i \sum_X \sum_{X \neq Y} V_{Xi} V_{Yi} +$$

$$C/2 ( \sum_X \sum_i V_{Xi} - n )^2 +$$

$$D/2 \sum_X \sum_{Y \neq X} \sum_i d_{XY} V_{Xi} ( V_{Y,i+1} + V_{Y,i-1} )$$

where A, B, C and D are positive.

The first triple sum is zero if and only if each city row X contains no more than one "1", the rest of the entries being zero. The second triple sum is zero if and only if each "position in tour" column contains no more than one "1" the rest of the entries being zero. The third term is zero if and only if there are n entries of "1" in the entire matrix. Thus, this energy function evaluated on the domain of the corners of the hypercube has minima with E=0 for all state matrices with one "1" in each row and column. All other states have higher energy. Hence, including these terms in an energy function describing a TSP network strongly favors stable states which are at least valid tours in the TSP problem and, and fulfills the first requirement for E. The last term in the above equation fulfills the second requirement, that E favor valid tours representing short paths. This term contains information about the length of the path corresponding to a given tour.

From the above energy function, one can deduce the implicitly defined connection matrix, which is given by:

$$\begin{aligned} T_{Xi,Yj} = & - A\, \delta_{XY} ( 1 - \delta_{ij} ) \\ & - B\, \delta_{ij} ( 1 - \delta_{XY} ) \\ & - C \\ & - D\, d_{XY} ( \delta_{j,i+1} + \delta_{j,i-1} ) \end{aligned}$$

where $\delta_{ij} = 1$ if i = j and is 0 otherwise .

This model of TSP has been simulated and verified to yield "reasonably optimal" solution to the TSP[1].

## 4.4 An ANN for Solving Dynamic Programming Problems

As discussed in the previous chapter, traditional dynamic programming is a computational technique which makes a sequence of decisions to define an optimal policy and path based on the principle of optimality. The conventional algorithm begins by finding the optimal path for the last stage and moves backward stage by stage until the optimal path starting at the source node is found. An ANN model to solve this is discussed below. This section has been written with extensive reference to the paper by Chui, Maa and Shanblatt [10].

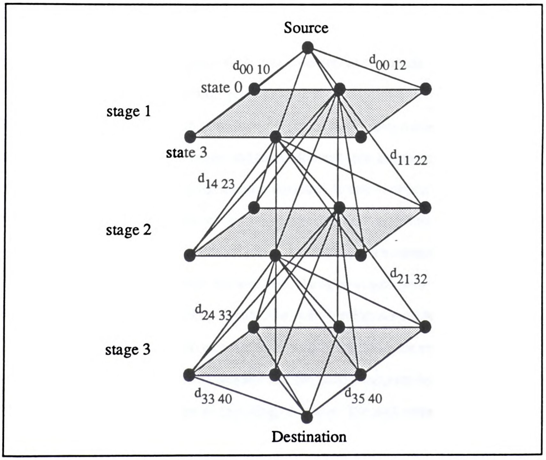


Figure 4.4. A 3x6 dynamic programming problem.

A typical dynamic programming problem is shown in Figure 4.5. A performance measure is defined as the total length of a valid path from the source node to the destination node. Given the source and destination nodes, the number of stages m, the number of states in each stage n, and the metric data $d_{xi,(x+1)j}$, where x is the index of stages, and i and j are the indices of states in each stage, the problem is to find an optimal path from source to destination. This optimal path is measured with respect to a performance criterion. The conventional approach uses the principle of optimality. It requires intensive calculations and a huge amount of memory to determine the optimal solution. In many dynamic programming applications where a real-time solution is required, the rapid calculation of near-optimal solutions is more attractive than a slowly computed globally optimal solution. For example, robot trajectory planning problems, aircraft altitude control problems, and optimal control problems that must respond quickly to radically changing environmental conditions are of this type. Following is a dynamic programming ANN that can provide a near-optimal solution in an elapsed time of only a few characteristic time constants of the circuit.

Consider again the 3x6 dynamic programming problem shown in Figure 4.4. The goal of the DPP is to find a valid path which starts from the source node, visits one and only one state node in each stage, reaches thedestination node, and has a minimum total length among all possible paths. To ensure that the ANN dynamic programming algorithm is able to obtain at least a near-optimal solution, the network must be defined by an energy function in which the optimal solution corresponds to the lowest energy state of the network. Looking at the characteristics of the optimal path carefully, two constraints become evident. First, the optimal path must visit one and only one state in each stage (structure constraint). Second, the optimal solution must have the minimum total cost based on the given performance measure (cost constraint). Thus, the energy function has two requirements. The structural constraint implies that the energy function must converge to stable states where one and only one state in each stage is active. The cost constraint dictates that the energy function must converge to stable states representing an optimal path.

Each state node is considered as an individual neuron. To develop an appropriate energy function for the dynamic programming network, take $V_{xi}$ as the output of a neuron of the *i*th state in the *x*th stage, where n is the number of stages and a and b are positive numbers. The following formal constraints are thus defined.

1. To ensure that one and only one neuron is active in any stage and the number of active processing elements is equal to the number of stages,

$$E_1 = a/2 \left( \sum_x \sum_i \sum_{j \neq i} V_{xi} V_{xj} + \left( \sum_x \sum_i V_{xi} - n \right)^2 \right).$$

2. To ensure that the total length of a valid path is minimum,

$$E_2 = b/4 \left( \sum_x \sum_i \sum_j \left( d_{xi\,(x+1)j} V_{xi} V_{(x+1)j} + d_{(x-1)j\,xi} V_{xi} V_{(x-1)j} \right) \right).$$

$E_1$ comes from the structure constraint and $E_2$ comes from the cost constraint. For a valid path, $E_1$ will vanish. For a minimum length path, $E_2$ has the minimum value. Therefore, to retain the characteristic of a gradient system, the energy function for the dynamic programming network can be written as

$$E = a/2 \left( \sum_x \sum_i \sum_{j \neq i} V_{xi} V_{xj} + \left( \sum_x \sum_i V_{xi} - n \right)^2 \right) +$$

$$b/4 \left( \sum_x \sum_i \sum_j \left( d_{xi\,(x+1)j} V_{xi} V_{(x+1)j} + d_{(x-1)j\,xi} V_{xi} V_{(x-1)j} \right) \right) +$$

$$\sum_x \sum_i (1/R_{xi}) \int_{0.5}^{V_{xi}} g_i^{-1}(\zeta)\, d\zeta.$$

$$= E^* + \sum_x \sum_i (1/R_{xi}) \int_{0.5}^{V_{xi}} g_i^{-1}(\zeta)\, d\zeta.$$

The quadratic terms in the above equations define the connection weight matrix T and the linear term defines the bias current vector I of the dynamic programming network.

Thus, the weight of the connection linking the *i*th neuron of stage x with the *j*th neuron of stage y is

$$T_{xi\,yj} = - a * \delta_{xy} * ( 1-\delta_{ij} ) - a - b/2 * d_{xiyj} * ( \delta_{(x+1)y} + \delta_{(x-1)y} )$$

where

$a * \delta_{xy} * ( 1-\delta_{ij} )$ is the inhibitory connection within each stage,

a is the global inhibition,

$b/2 * d_{xiyj} * ( \delta_{(x+1)y} + \delta_{(x-1)y} )$ is the strength of the metric distance,

$$\delta_{ij} \text{ ( or } \delta_{xy} \text{ )} = \begin{cases} 1 \text{ if } i = j \ ( x = y ), \\ 0 \text{ otherwise,} \end{cases}$$

and the input bias current of *i*th neuron of stage *x* is

$$I_{xi} = a\ n.$$

It is apparent that $T_{xiyj}$ is equal to $T_{yjxi}$ for all *x,y,i* and *j*. Moreover, E is positive-definite. Thus the dynamic programming ANN is a gradient system and the equilibria are bounded in the V space. With the high-gain limit, the stable states will be close to the minimum states of E* since the integral term can be neglected.

## 4.5 An ANN Model for the Stagecoach Problem

Using the above algorithm, an ANN model for the stagecoach problem is constructed as shown in Figure 4.5.Values chosen for different parameters are as follows:

a = 5

b = 5.

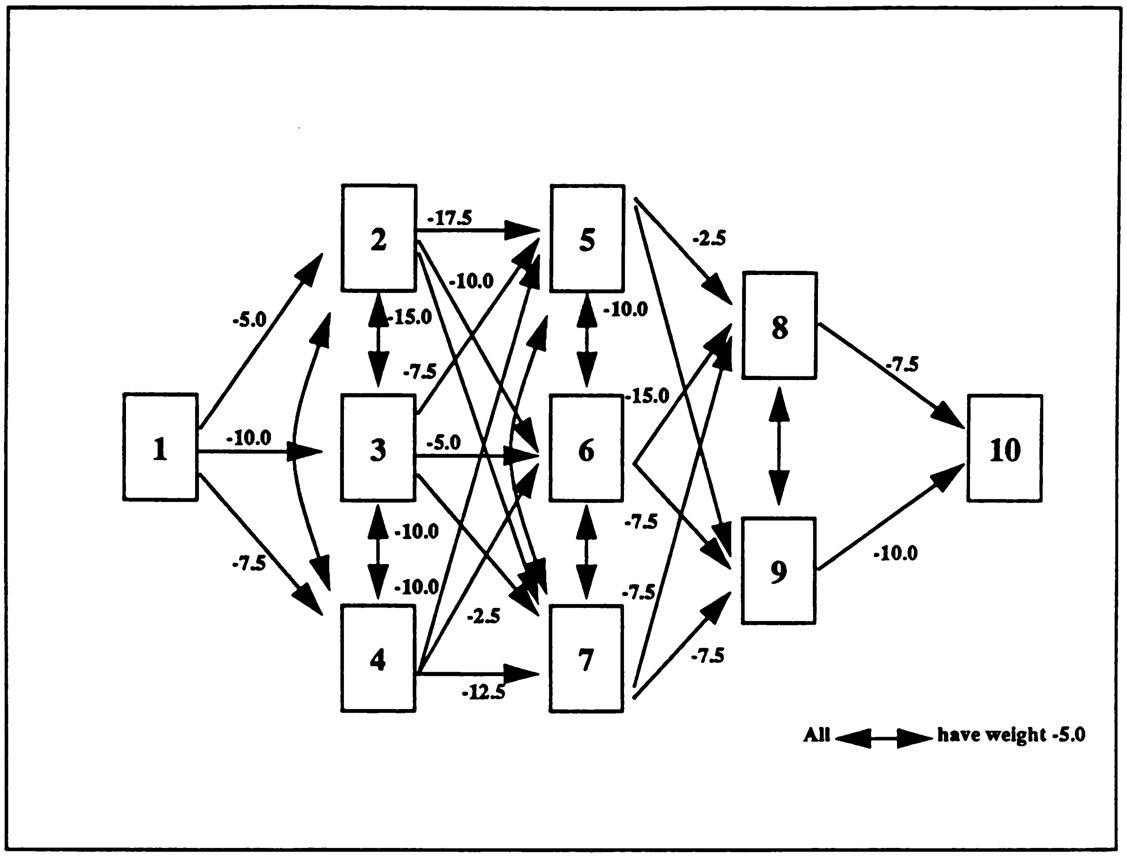


Figure 4.5. An ANN model for the stagecoach problem.

# Chapter 5

# Neural Coprocessor: A Behavioral Model

## 5.1 Introduction

The suitability and advantages of using VHDL for ANN modeling and simulation have been discussed in the preceding chapters. We shall see one such system modeled using VHDL.

The immediate objective of this thesis is to model an ANN for solving dynamic programming problems. The stagecoach problem explained in Chapter 3 is used as a test example for running the simulation. The ANN model to solve such problems proposed by Chiu, Maa and Shanblatt[10] has been used for implementing this network.

The larger objective of this research effort, however, is to demonstrate the suitability and various advantages one accrues by using VHDL as the vehicle for modeling ANNs. These advantages include flexibility in design, modularity, and ease of information exchange, and testing of designs. Hence, the network has been modeled as a general-purpose ANN coprocessor, much like a math coprocessor. This system can be configured to model any network with very little extra effort (none in many cases). Hence, this network is not limited to solving this particular example or this specific kind of problem. It can be used as a testbench to simulate and verify different ANN models for various domains without significant modification.

## 5.2 Design Methodology

In confirmation with the objective of developing a general-purpose ANN coprocessor, the design has been made as flexible and as general as possible. At the highest level of the hierarchy, the whole system can be viewed as a coprocessor which has an input port to get initial conditions for the ANN, a mechanism to enter the interconnection weight-matrix, and an output port to return the output values of the neurons. This is shown in Figure 5.1.

This coprocessor is built using various components as explained in the next section. The design methodology adopted across the system is to make different components independent of the application. The whole network as applied can be described in a package right at the beginning. In order to do this, one has to assign values for certain variables such as the number of stages in the network, number of states in each stage, and connectivity between different neurons in the network. Finally, an appropriate test bench can be created to test the system.

This structured design scheme facilitates easy modification as necessary. New features like learning, or changing the neuron model, for example, can be accomplished without much effort.

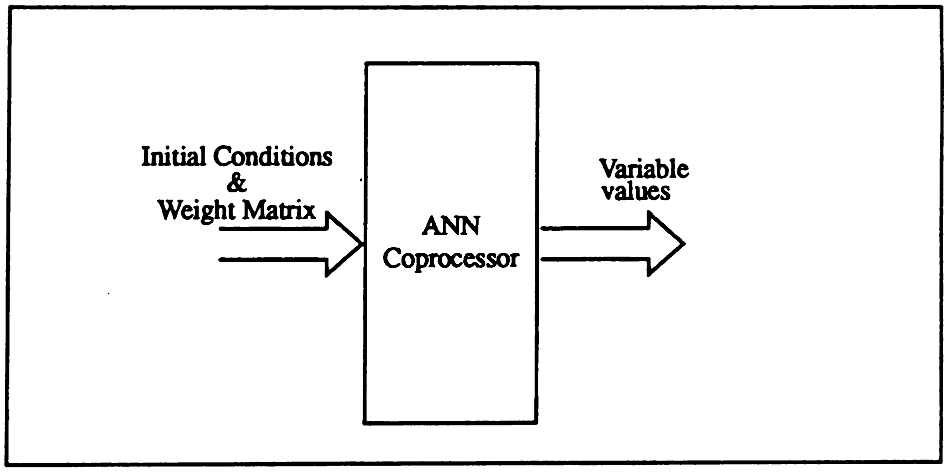


Figure 5.1. The ANN coprocessor.

## 5.3 Design

The top level design of the neural coprocessor is shown in Figure 5.2. The system is comprised of an ANN at the core, a memory to hold the interconnection weight - matrix, a set of registers to hold the value of the neurons, and a convergence sensor.

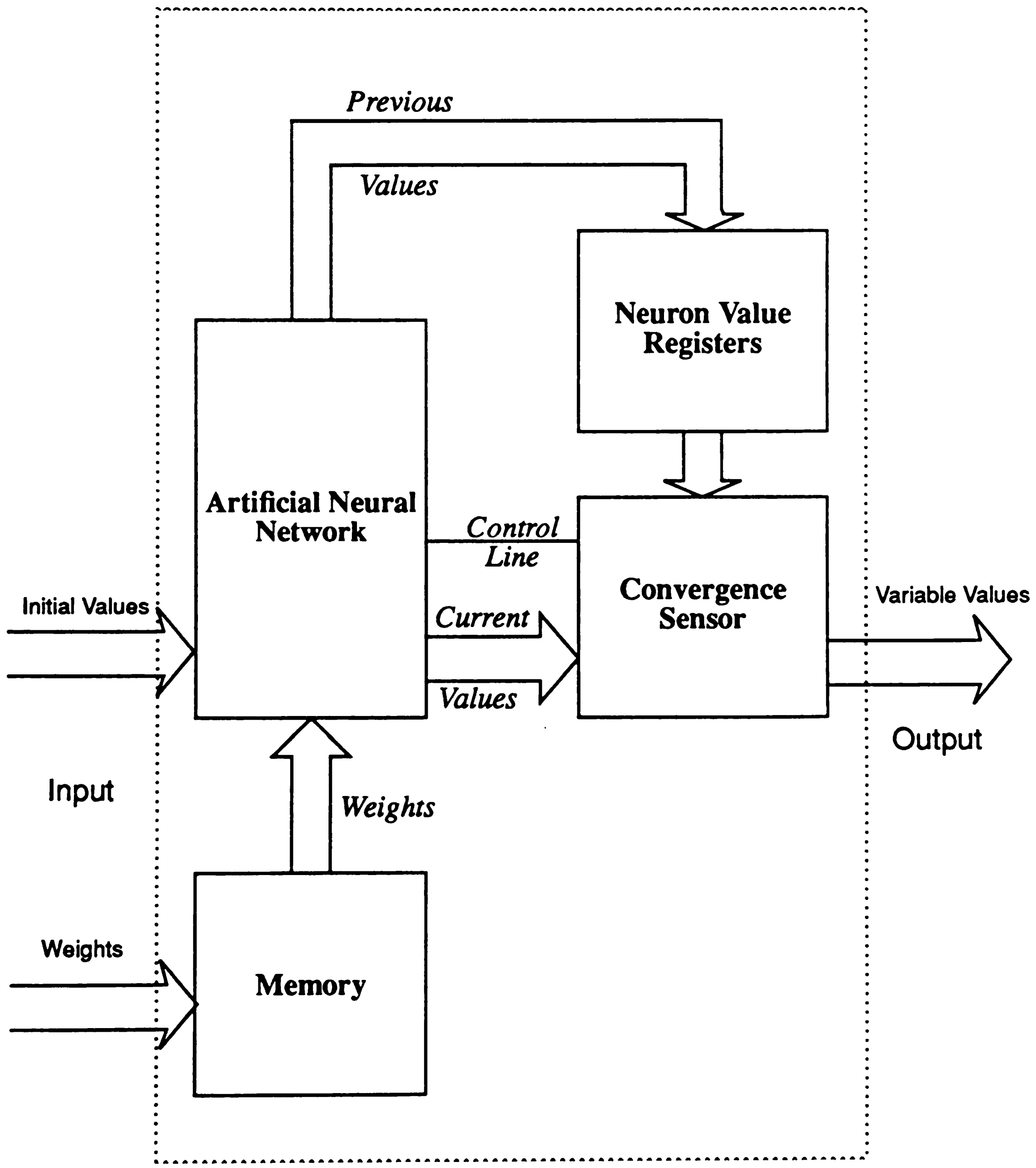


Figure 5.2. Top level schematic diagram of the system.

**5.3.1 Network:** A schematic of the composition of the network is shown in Figure 5.3. The network is built using neurons, where every neuron in the network is connected to all the neurons in the previous, current and the next stage. The very first stage is connected to the very last stage in the network, making it a ring structure.

As stated earlier, the design is such that the network can be of any size and can be described by declaring some parameters such as number of stages and number of states per stage. This description is to be given in the VHDL package declaration called Neural_Package.

Every neuron in the network has available to it the weights and the corresponding stimuli for it's links with neurons in the preceding stage and neurons in the same stage (including itself). Complete connectivity (where every neuron is connected to every other neuron in the network) is not implemented as it would put unnecessary load on the system. Most of the present day models are not completely connected. Nevertheless, the system design can be modified to make the network completely connected if desired . The VHDL code to accomplish this is listed in Figure 5.4.

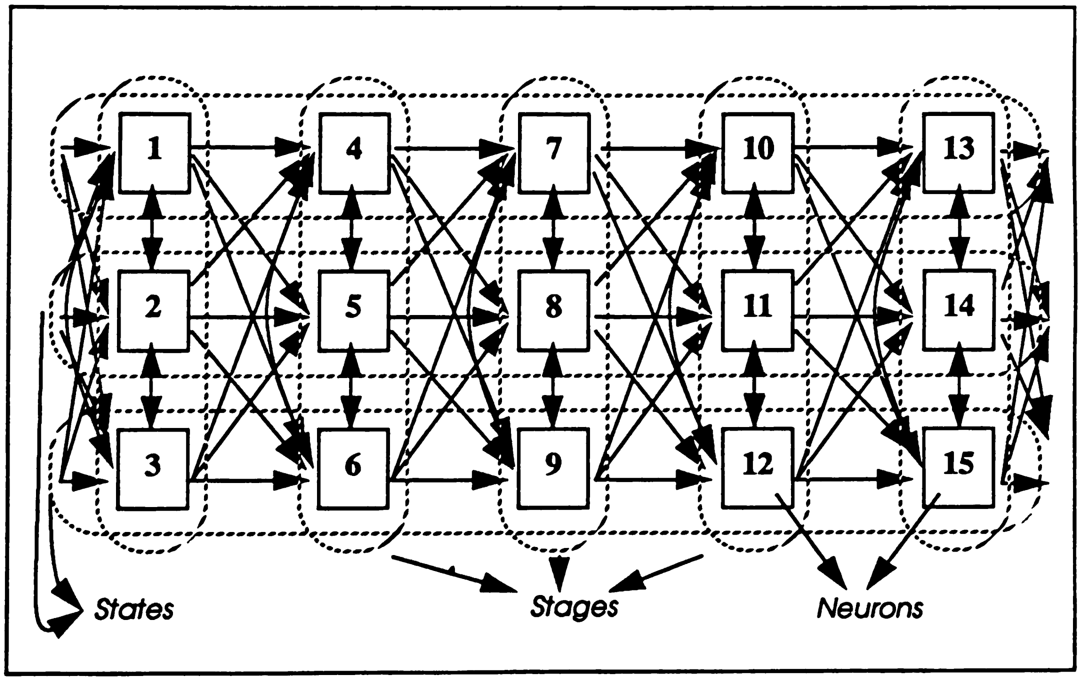


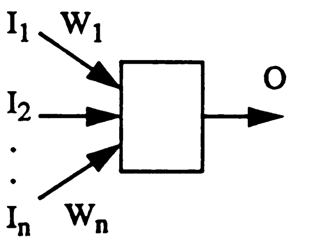
Figure 5.3. Composition of the network for a case of 5 stages with 3 states per stage.

```
– A network of Neural Elements
use work.Neural_Package.all;
entity network is
 port (
                Network_Stimulus : in Stimulus_Matrix;
                Network_Weights : in Weights_Matrix;
                Network_Output : out Neural_Array);
end network;

architecture network_structure of network is

component Neural_Node
 port (
                Stimulus : in Unit_Array;
                Weights : in Unit_Array;
                Output : out Real := 0.0);
 end component;


– Instantiation of all Neural_Nodes to the Neural_Element design
unit

for all : Neural_Node use entity work.Neural_element(behavior);

begin

    element_generate:
        for i in 1 to N_Units generate
          Nodes : Neural_node
                   port map (Network_Stimulus(i),
                   Network_Weights(i), Network_Output(i) );
    end generate;

end network_structure;
```

Figure 5.4. VHDL code implementing the network.

**5.3.2 Neurons:** Each neuron in the network is essentially a summation unit. It calculates the inner product of the weight and stimulus vectors and provides an output value based on this inner product. The size of these vectors depends on the size of the network. The parameters are read in from the package declaration Neural_Package. The function CalculateSum is described in the package body of Neural_Package.

In the present implementation, the sigmoid input - output relation of a neuron as described in Chapter 4 has been approximated by a step function. This is due to the non-availability of certain mathematical functions in VHDL (such as $\tanh^{-1}$ which is the usual approximation for the sigmoid function). Approximating the sigmoid with "stair-case" function or a ramp function is is also possible. A schematic of a neuron and the corresponding VHDL code is shown in Figure 5.5.



Neuron
Inputs -> $I_i$ and $W_i$
Neural Function
Output O
-> 1 if $\Sigma I_i W_i$+ bias > threshold
0 otherwise
after delta delay

```
– Model of a Neuron Element

use work.Neural_Package.all;
entity Neural_element is
 port (
      Stimulus : in Unit_Array;
      Weights : in Unit_Array;
      Output : out Real);
end Neural_Element;

architecture behavior of neural_element
is
begin
 NeuralProcess:
  process(Stimulus'Transaction)
    variable Sum : Real;

  begin

  Sum := CalculateSum(Stimulus,
Weights) + 10;
  if Sum > (0.0) then
      Output <= 0.0 after 4 ns;
  else
      Output <= 0.5 after 4 ns;
  end if;
  end process;
```

Figure 5.5. Schematic of a neuron and the corresponding VHDL code.

**5.3.3 Memory:** The memory component holds the weights to be used in the network. Weights can be read from this memory only once at the beginning, or iteratively, if a learning algorithm is incorporated. Initially, it was desired to read in interconnection weight - matrix values into the memory from a file, making it possible to run different networks without having to rebuild the network (as long as their dimension is same). This could not be accomplished as the file I/O functions available with VHDL are not capable of handling such data transfers in the latest version. Hence, the current implementation has the interconnection weight-matrix built into the memory (essentially a ROM). If a different system is to be simulated, the corresponding weight matrix is to be entered into the architecture of the memory and the system has to be rebuilt.

The VHDL code to model this memory is shown in Figure 5.6.

```
-- This module is meant to be used as a memory to hold the weights. A centralized
memory is visualized as the change required for a different network would be min-
imal this way.

use work.Neural_Package.all;
entity memory is
port (
        memory_output : out Weights_Matrix);
end memory;

architecture memory_arch of memory is

begin
        memory_output <= (
                (0.0, 0.0, 0.0, -100.0, -100.0, -100.0), (0.0, 0.0, 0.0, 0.0, 100.0, 0.0), (0.0, 0.0, 0.0, -100.0, -100.0, -
100.0),
                (0.0, -5.0, 0.0,0.0, -5.0,-5.0), (0.0, -10.0, 0.0, -5.0,0.0, -5.0), (0.0, -7.5, 0.0,-5.0,-5.0, 0.0),
                (-17.5, -7.5, -10.0, 0.0, -5.0,-5.0), (-10.0, -5.0, -2.5,-5.0,0.0, -5.0),( -15.0, -10.0, -12.5,-5.0,-5.0, 0.0),
                (-2.5, -15.0, -7.5,0.0, -5.0,-5.0),(-10.0, -7.5, -7.5,-5.0, 0.0, -5.0), (0.0, 0.0, 0.0,-100.0,-100.0, 0.0),
                (0.0, 0.0, 0.0, 0.0, -100.0,0.0),(-7.5, -10.0, 0.0, 0.0, 100.0, 0.0), (0.0, 0.0, 0.0, 0.0,-100.0, 0.0));
end memory_arch;
```

Figure 5.6. VHDL code implementing the memory.

**5.3.4 Convergence Sensor:** The convergence sensor is designed to detect convergence of the network to a set of values. It compares the present neuron-outputs with the previous set of output values and sends out a signal if they are identical. This component is more of a hardware abstraction and has not been used in actual implementation as the VHDL simulator itself senses convergence if it is run in the interactive mode, and as the neuron model adopted is a step function.

The schematic and the equivalent VHDL code is shown in Figure 5.7.

```
– This unit is used to sense the convergence of the network
to a solution

use work.Neural_Package.all;
entity conv_sensor is
 port (
        Old_Outputs : in Neural_Array;
        New_Outputs : in Neural_Array;
        Sensor_Out : out integer);
end conv_sensor;

architecture sensor_behavior of conv_sensor is

    signal difference : Real := 0.0;
begin
   loop_process:
     process
     begin
       Sensor_Out <= 1 after 0 ns;
       loop1: for i in 1 to N_Units loop

         difference <= abs (Old_Outputs(i) - New_Outputs(i));
         if difference > Tolerence then
              Sensor_Out <= 0;
         end if;
       end loop loop1;
   end process;
end sensor_behavior;
```

Figure 5.7. VHDL code to implement the convergence sensor.

**5.3.5 Register Set:** The Register set is an abstraction with no exact equivalent in the VHDL model.

**5.3.6 Auxiliary Components:** Some auxiliary components required to implement this system in VHDL, namely a package definitions called Neural_Package, a neural processor, where different components are integrated into one entity, and a test bench to test the system are shown in Figures 5.8, 5.9 and 5.10.

```
package Neural_Package is
    constant N_Stages : natural := 5;
    constant N_States : natural := 6;
    constant States_per_Stage : natural := 3;
    constant N_Units : natural := 15;
    type Neural_Array is array (Natural range 1 to N_Units ) of Real;
    type Unit_Array is array (Natural range 1 to N_States ) of Real;
    type Weights_Matrix is array (Natural range 1 to N_Units) of Unit_Array;
    type Stimulus_Matrix is array (Natural range 1 to N_Units) of Unit_Array;

   function CalculateSum (
                Stimulus : Unit_Array;
                Weights : Unit_Array)
                return Real;
   end Neural_Package;

package body Neural_Package is

   function CalculateSum (
                Stimulus : Unit_Array;
                Weights : Unit_Array)
                return Real;
   is
        variable Sum : Real := 0.0;
   begin
      for I in 1 to N_States loop
         Sum := Sum + Stimulus(I) * Weights(I);
      end loop;
      return Sum;
   end CalculateSum;

end Neural_Package;
```

Figure 5.8. VHDL code for the Neural_Package.

```
-- This is the top level assembly of all the sub-components. Different signals
are generated and fed to different parts.

use work.Neural_Package.all;

entity ann_processor is
 port (
        Stimulus : in Neural_Array;
        Output : out Neural_Array);
end ann_processor;

architecture processor_structure of ann_processor is

    component memory
    port (
          memory_output : Weights_Matrix);
    end component;

    component network
    port (
          Network_Stimulus : in Stimulus_Matrix;
          Network_Weights : in Weights_Matrix;
          Network_Output : out Neural_Array);
     end component;

     signal Matrix_Weights : Weights_Matrix :=

          ((0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0),
           (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0),
           (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0),
           (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0),
           (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0) );

     signal Matrix_Stimulus : Stimulus_Matrix :=

          ((0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0),
           (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0),
           (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0),
           (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0),
           (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0) );

for all : network use entity work.network(network_structure);
for all : memory use entity work.memory(memory_arch);
```

Figure 5.8. VHDL code implementing the neural coprocessor ( *Cont'd. on next page* ).

```
begin

     process(Stimulus'Transaction)
          variable tmp1 : integer := 1;
          variable tmp2 : integer := 1;
     begin
          element_loop1:
               for ii in 1 to N_Units loop
               element_loop2 :

                    for k in 1 to States_per_Stage loop

                         tmp1 := (ii mod States_per_Stage);
                         if tmp1 = 0 then
                                   tmp1 := N_Stages -1;
                         end if;
                         tmp2 := ((tmp1 -1) * States_per_Stage ) + k;
                         Matrix_Stimulus(ii)(k) <= Stimulus(tmp2);

               end loop element_loop2;

               element_loop3 :

                    for l in 1 to (N_States - States_per_Stage ) loop

                         tmp1 := (ii mod States_per_Stage);
                         if tmp1 = N_Stages then
                                   tmp1 := N_Stages - 1;
                         end if;

                         tmp2 := ((tmp1) * States_per_Stage ) + l ;
                         Matrix_Stimulus(ii)(l + States_per_Stage) <= Stimulus(tmp2);

               end loop element_loop3;

          end loop element_loop1;

     end process;

read_memory : memory port map ( Matrix_Weights);

-- feed the Neural Network with these Weights and the Stimulus and obtain the Output

run_network : network port map ( Matrix_Stimulus, Matrix_Weights, Output );


end processor_structure;
```

Figure 5.8. VHDL code implementing the neural coprocessor ( *Cont'd. from previous page* ).

```
-- This is the test bench for testing the whole system. Whole system is integrated in ann_processor.

use work.Neural_package.all;
entity test_bench is
end test_bench;

architecture test_bench_arch of test_bench is

 component ann_processor
port (
            Stimulus : in Neural_Array;
            Output : out Neural_Array);
 end component;


 signal kIn : Neural_Array := (0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0);
 signal kOut : Neural_Array;

 for all : ann_processor use entity work.ann_processor(processor_structure);

begin

            tester : ann_processor port map ( kIn , kOut );

            Process(kOut)
            begin

                        kIn <= kOut after 10 ns;
            end process;
end test_bench_arch;
```

Figure 5.9. VHDL code implementing the test bench.

## 5.4 Simulating the Stagecoach Problem

As has explained earlier, the neural coprocessor is a general one and has to be customized to run a particular example. Following are the steps to be taken to run the network to solve the stagecoach problem that was discussed in Chapter 3.

1. Since there are 5 stages ( counting the point of origin and the destination ) and there are 3 states in each stage, the following parameters have to be set in the package declaration Neural_Package.

```
constant N_Stages : natural := 5;
constant States_per_Stage : natural := 3;
constant N_States : natural := 6;
constant N_Units : natural := 15;
```

Note that N_States specifies the number of links into a neuron ( including the one to itself) in the network. States_per_Stage is the number of states in each stage.

2. Since the network modeled in VHDL is a symmetric rectangular network, some neurons may have to be "deactivated" to represent an assymmetric network. The network used to solve dynamic programming problems is not rectangular. Therefore, some of the neurons in the network described in the package declaration have to be isolated from the network. This can be achieved by making the weights in the interconnection weight matrix corresponding to the inputs of these neurons totally inhibitory. Weights corresponding to their output is set to 0. The modified network is shown in Figure 5.11.

3. The interconnection weight - matrix, after incorporating the above changes would be as shown in Figure 5.10. This is to be incorporated in the memory's architectural description.

This matrix is structured in the form of an array of arrays, where each array represents the weights for different links for every neuron in the network. The first array corresponds to the first neuron ( top left) and the last one corresponds to the last neuron ( bottom right). A weight of -100 is found to be inhibitory enough to isolate unwanted neurons from the network.

```
(

(0.0, 0.0, 0.0, -100.0, -100.0, -100.0), (0.0, 0.0, 0.0, 0.0, 100.0, 0.0), (0.0, 0.0, 0.0, -100.0, -100.0, -100.0),

(0.0, -5.0, 0.0, 0.0, -5.0, -5.0), (0.0, -10.0, 0.0, -5.0, 0.0, -5.0), (0.0, -7.5, 0.0, -5.0, -5.0, 0.0),

(-17.5, -7.5, -10.0, 0.0, -5.0, -5.0), (-10.0, -5.0, -2.5, -5.0, 0.0, -5.0),( -15.0, -10.0, -12.5, -5.0, -5.0, 0.0),

(-2.5, -15.0, -7.5, 0.0, -5.0,-5.0),(-10.0, -7.5, -7.5,-5.0, 0.0, -5.0), (0.0, 0.0, 0.0,-100.0,-100.0, 0.0),

                                                                                                              )
```

Figure 5.10. The interconnection weight - matrix for solving the stagecoach problem.

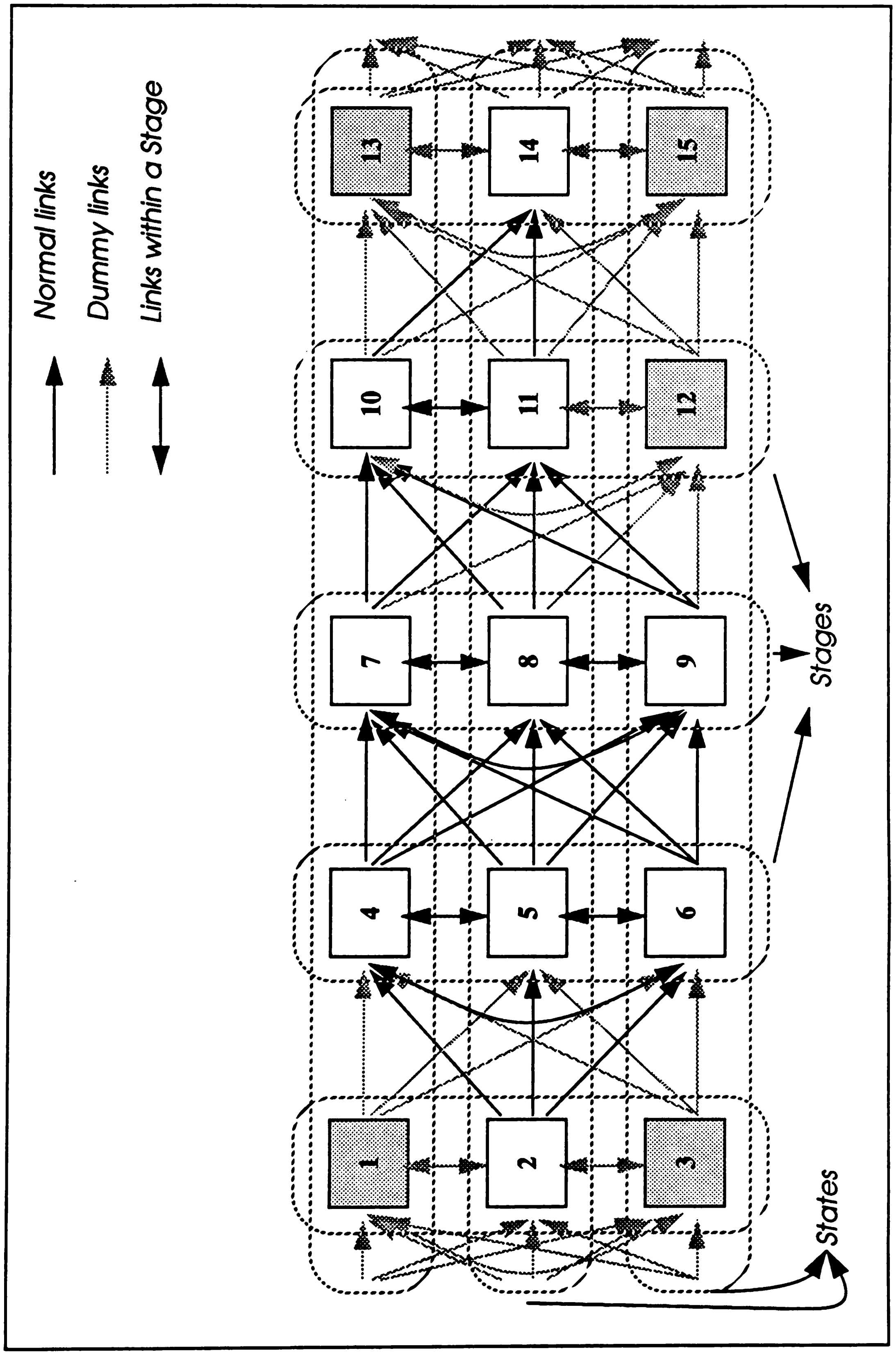


Figure 5.11. The ANN for the stagecoach problem.

*Chapter 6*

# *Post-Simulation Analysis*

## 6.1 Introduction

The system modeled in VHDL as explained in Chapter 5 was simulated to solve the stagecoach problem. The initial stimulus values used for these simulations were chosen such that all possible cases were covered. These simulation runs were monitored for all the transactions for about 1us. The characteristic delay of the whole network was set to about 4 ns and the stimulus values- output values of the previous iteration, were fed in at the intervals of 10 ns. Outcome of these simulation runs and an analysis of these results follow.

This system was also used to simulate a different problem to verify that the system designed is not anecdotal to the stagecoach problem. This was accomplished by adding another stage to the network. The outcome of these simulation runs are analyzed in Section 6.4.

## 6.2 Results and Analysis

The simulation results for the stagecoach problem are tabulated in Table 6.1. The complete simulation results can be seen in Appendix C.

Some general observations on the simulation run data in Table 6.1 follow. All the vectors shown indicate the values for all the neurons in the complete network discussed in Chapter 5.

Table 6.1. Simulation results for the stagecoach problem.

| Run | Initial Stimulus | Final Outcome | # of cycles |
|---|---|---|---|
| *a* | ( 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0 ) | ( 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0 ) | 6 |
| *b* | ( 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0 ) | ( 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0 ) | 5 |
| *c* | ( 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0 ) | ( 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0 ) | 7 |
| *d* | ( 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0 ) | ( 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0 ) | 7 |
| *e* | ( 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0 ) | ( 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0 ) | 1 |
| *f* | ( 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0 ) | ( 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0 ) | 4 |
| *g* | ( 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ) | Does not converge ( Toggles ) | - |

1. Runs *a* and *b* have an intial stimulus that is a valid solution to the problem, though not optimal. Both converge to one of the optimal solution in less than 6 cycles.

2. Run *c* has an input stimulus which has one of the deactivated neurons active. But still the network converges to an optimal solution in 7 cycles.

3. Run *d* has an invalid neuron state as the input stimulus (two neurons in the same stage are active). The network converges to an optimal solution in 7 cycles.

4. Run *e* has one of the optimal solution itself as the input stimulus. It remains in the optimal state.

5. Run *f* has another optimal state as the input stimulus, and it is observed that the network converges to a different optimal state.

6. Run *g* has a zero input stimulus for all the neurons, and the network toggles between all the neurons being at zero and all the neurons being at one. The same case is observed when all the neurons are initially set to one.

Hence, it can be seen that the network converges to an optimal solution, even when the input stimulus is a different optimal state, in all cases except when all the neurons are initially set to zero or one.

## 6.3 Possible Reasons for the Observed Behavior

Some of the assumptions and approximations made in the present implementation that might be responsible for the observed behavior of the network are listed below:

1. The algorithm proposed by Chiu, Maa and Shanblatt claims that optimization done pairwise will lead to a global optimization[10], i.e., every stage in the network need be connected to only the preceding and succeeding stages, and total connectivity is unnecessary. But this does not guarantee that this algorithm will lead to all the optimal solutions. It may lead to only one of the optimal solutions which happens to be pairwise optimal too. This could be the reason for the network converging to the same optimal solution in all the cases (even when the input stimulus is a different optimal state).

2. The neuron input-output relation was modeled by a step function in place of a sigmoid function (due to the unavailability of trigonometric functions in VHDL, at this time). This could be the reason for the network being not able to move towards a solution when all the neurons are set to zeros or ones as much of the information gathered in the previous cycle is lost. It behaves like a memoryless system. Future work is intended to approximate the sigmoid function by a ramp function, a staircase - like function, etc.

## 6.4 An Extended Problem

In order to verify that the network designed is not anecdotal to the particular stagecoach problem and to highlight the ease with which it can be modified to fit a different problem, an extended stagecoach problem was developed and was solved using the network modeled in VHDL. It was seen that the network converges to the optimal solution in less than 4 - 5 cycles as expected.A schematic of the extended system is shown in Figure 6.1. The simulation results are tabulated in Table 6.2. The changes in the VHDL code to effect the desired changes are listed in Figures 6.2, 6.3 and 6.4.

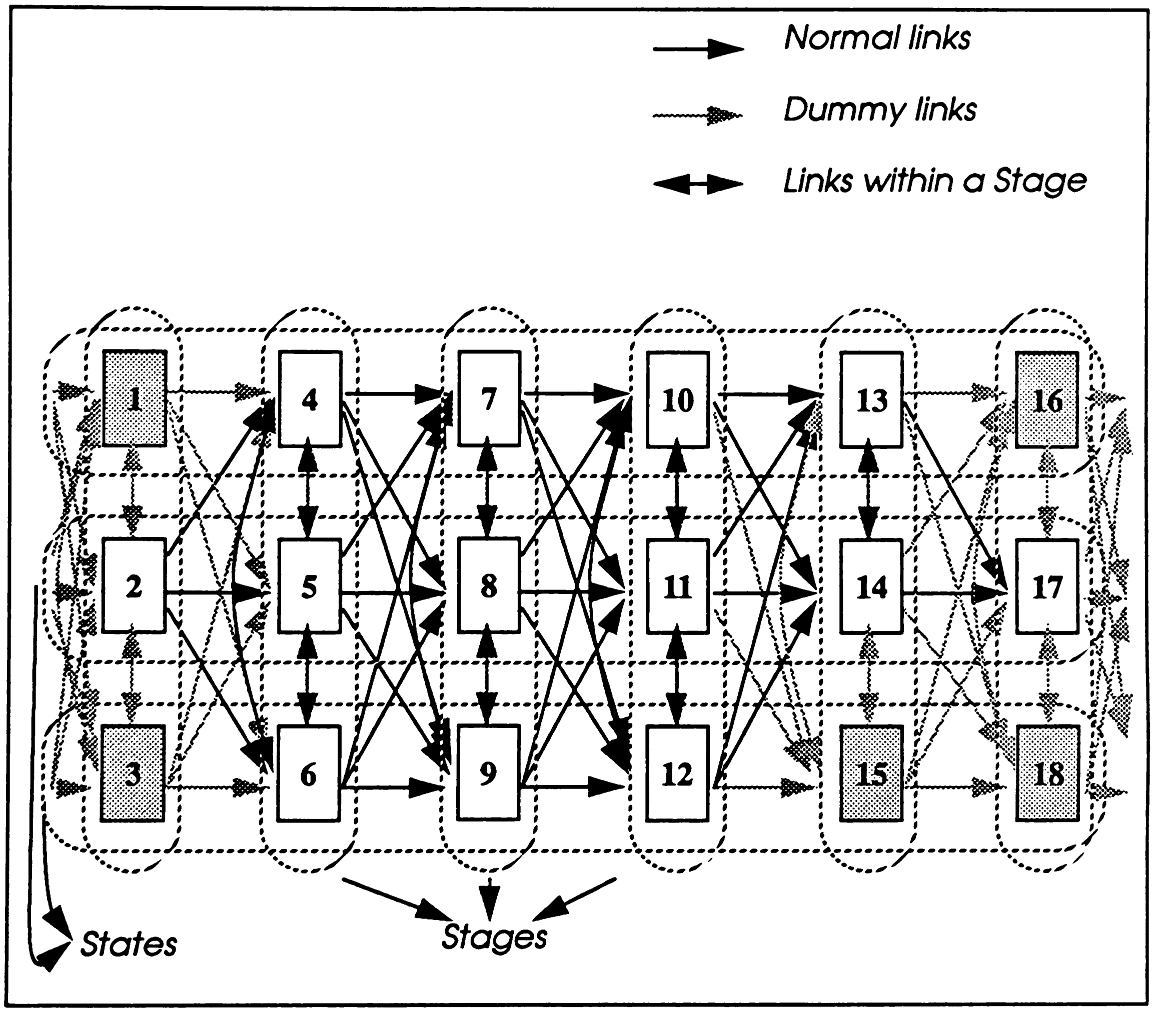


Figure 6.1. The ANN for the extended stagecoach problem.

```
                              •
                              •
package Neural_Package is
 constant N_Stages : natural := 6;
 constant N_States : natural := 6;
 constant States_per_Stage : natural := 3;
 constant N_Units : natural := 18;
                              •
                              •
```

Figure 6.2. The modified package declaration for solving the extended stagecoach problem.

```
                              •
                              •
signal kIn : Neural_Array := (0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0
);
signal kOut : Neural_Array;

for all : ann_processor use entity work.ann_processor(processor_structure);
                              •
                              •
```

Figure 6.3. The modified test bench for solving the extended stagecoach problem.

```
(

      (0.0, 0.0, 0.0, -100.0, -100.0, -100.0), (0.0, 0.0, 0.0, 0.0, 100.0, 0.0), (0.0, 0.0, 0.0, -100.0, -100.0, -100.0),

      (0.0, -5.0, 0.0, 0.0, -5.0, -5.0), (0.0, -10.0, 0.0, -5.0, 0.0, -5.0), (0.0, -7.5, 0.0, -5.0, -5.0, 0.0),

      (-17.5, -7.5, -10.0, 0.0, -5.0, -5.0), (-10.0, -5.0, -2.5, -5.0, 0.0, -5.0),( -15.0, -10.0, -12.5, -5.0, -5.0, 0.0),

      (-2.5, -15.0, -7.5, 0.0, -5.0, -5.0), (-10.0, -7.5, -7.5, -5.0, 0.0, -5.0),( -12.5, -12.5, -2.5, -5.0, -5.0, 0.0),

      (-17.5, -7.5, -10.0, 0.0, -5.0, -5.0), (-12.5, -15.0, -10.0, -5.0, 0.0, -5.0),(0.0, 0.0, 0.0, -100.0, -100.0, 0.0),

      (0.0, 0.0, 0.0, 0.0, -100.0, 0.0),(-7.5, -10.0, 0.0, 0.0, 100.0, 0.0), (0.0, 0.0, 0.0, 0.0,-100.0, 0.0)

                                                                                                                )
```

Figure 6.4. The interconnection weight - matrix for solving the extended stagecoach problem.

Table 6.2. Simulation results for the extended stagecoach problem.

| Run | Initial Stimulus | Final Outcome | # of cycles |
|---|---|---|---|
| *a* | ( 0, 1, 0, 0, 0, 1, 0, 0, 1,0, 1, 0, 1, 0, 0, 0, 1, 0 ) | ( 0, 1, 0, 0, 0, 1, 0, 1, 0,0, 0, 1, 1, 0, 0, 0, 1, 0 ) | 6 |
| *b* | ( 0, 1, 0, 1, 0, 0, 0, 0, 1,0, 0, 1, 1, 0, 0, 0, 1, 0 ) | ( 0, 1, 0, 0, 0, 1, 0, 1, 0,0, 0, 1, 1, 0, 0, 0, 1, 0 ) | 4 |
| *c* | ( 0, 1, 0, 0, 0, 1, 0, 0, 1,0, 0, 1, 0, 0, 1, 0, 1, 0 ) | ( 0, 1, 0, 0, 0, 1, 0, 1, 0,0, 0, 1, 1, 0, 0, 0, 1, 0 ) | 4 |
| *d* | ( 0, 1, 0, 0, 0, 1, 0, 0, 1,1, 0, 1, 1, 0, 0, 0, 1, 0 ) | ( 0, 1, 0, 0, 0, 1, 0, 1, 0,0, 0, 1, 1, 0, 0, 0, 1, 0 ) | 4 |
| *e* | ( 0, 1, 0, 0, 0, 1, 0, 1, 0,0, 0, 1, 1, 0, 0, 0, 1, 0 ) | ( 0, 1, 0, 0, 0, 1, 0, 1, 0,0, 0, 1, 1, 0, 0, 0, 1, 0 ) | 1 |
| *f* | ( 0, 1, 0, 0, 0, 1, 0, 0, 1,0, 0, 1, 1, 0, 0, 0, 1, 0 ) | ( 0, 1, 0, 0, 0, 1, 0, 1, 0,0, 0, 1, 1, 0, 0, 0, 1, 0 ) | 4 |
| *g* | ( 0, 0, 0, 0, 0, 0, 0, 0, 0,0, 0, 0, 0, 0, 0, 0, 0, 0 ) | Does not converge ( Toggles ) | - |

# Chapter 7

# Conclusion

## 7.1 Conclusion

The primary objective of this research effort was to prove the suitability of VHDL for ANN modeling and simulation. This has been achieved by modeling a general purpose ANN coprocessor in VHDL. This system was tested by simulating a dynamic programming problem, namely the stagecoach problem. This system was simulated with different initial conditions. The system modeled behaved according to expectations and the results are encouraging.

This research effort's contribution has been in establishing the suitability of VHDL for ANN modeling. The general purpose ANN coprocessor developed can be used to model different systems with little extra effort.

## 7.2 Future Research

As has been explained earlier, the objective of this thesis effort was to prove the suitability of VHDL for ANN modeling, with dynamic programming as a sample domain. Hence, although the system developed is general in nature and flexible enough to model any problem, the particular example solved is a simple case with very few features. The future research should be directed to

enhancing the capabilities of the network by incorporating new features such as learning, different models for the neuron, and complete connectivity between different neurons in the network. A much larger problem, with a larger solution space is yet to be modeled. These form a framework and a direction for future research efforts in this area.

# APPENDIX I

# VHDL CODE LISTING

## VHDL CODE LISTING

```
-- Package declaration : Neural Package

package Neural_Package is
    constant N_Stages : natural := 5;
    constant N_States : natural := 6;
    constant States_per_Stage : natural := 3;
    constant N_Units : natural := 15;
    type Neural_Array is array (Natural range 1 to N_Units ) of Real;
    type Unit_Array is array (Natural range 1 to N_States ) of Real;
    type Weights_Matrix is array (Natural range 1 to N_Units) of Unit_Array;
    type Stimulus_Matrix is array (Natural range 1 to N_Units) of Unit_Array;

   function CalculateSum (
        Stimulus : Unit_Array;
        Weights : Unit_Array)
        return Real;

end Neural_Package;



package body Neural_Package is

    function CalculateSum (
        Stimulus : Unit_Array;
        Weights : Unit_Array)
        return Real
   is
   variable Sum : Real := 0.0;
   begin
    for I in 1 to N_States loop
        Sum := Sum + Stimulus(I) * Weights(I);
    end loop;
    return Sum;
   end CalculateSum;

end Neural_Package;
```

```
-- Model of a Neuron Element

use work.Neural_Package.all;
entity Neural_element is
    port (
        Stimulus : in Unit_Array;
        Weights : in Unit_Array;
        Output : out Real := 0);
end Neural_Element;


architecture behavior of neural_element is
begin
    NeuralProcess:
        process(Stimulus'Transaction)
        variable Sum : Real;

        begin

        Sum := CalculateSum(Stimulus, Weights) + 10.0;
        if Sum > (0.0) then
            Output <= 1.0 after 4 ns;
        else
            Output <= 0.0 after 4 ns;
        end if;
    end process;
end behavior;




-- This module is meant to be used as a memory to hold the input values
-- ( Weights ). A centralized memory is visualised as the change required
-- for a different network would be minimal this way.


use work.Neural_Package.all;
entity memory is
    port (
        memory_output : out Weights_Matrix);
end memory;
```

```
architecture memory_arch of memory is
begin
    memory_output <= (
        (0.0, 0.0, 0.0, -100.0, -100.0, -100.0), (0.0, 0.0, 0.0, 0.0, 100.0, 0.0), (0.0, 0.0, 0.0, -100.0, -100.0, -100.0),
        (0.0, -5.0, 0.0,0.0, -5.0,-5.0), (0.0, -10.0, 0.0, -5.0,0.0, -5.0), (0.0, -7.5, 0.0,-5.0,-5.0, 0.0),
        (-17.5, -7.5, -10.0, 0.0, -5.0,-5.0), (-10.0, -5.0, -2.5,-5.0,0.0, -5.0),( -15.0, -10.0, -12.5,-5.0,-5.0, 0.0),
        (-2.5, -15.0, -7.5,0.0, -5.0,-5.0),(-10.0, -7.5, -7.5,-5.0, 0.0, -5.0), (0.0, 0.0, 0.0,-100.0,-100.0, 0.0),
        (0.0, 0.0, 0.0, 0.0, -100.0,0.0),(-7.5, -10.0, 0.0, 0.0, 100.0, 0.0), (0.0, 0.0, 0.0, 0.0,-100.0, 0.0));
end memory_arch;


-- A network of Neural Elements

use work.Neural_Package.all;
entity network is
    port (
        Network_Stimulus : in Stimulus_Matrix;
        Network_Weights : in Weights_Matrix;
        Network_Output : out Neural_Array);
end network;

architecture network_structure of network is

    component Neural_Node
        port (
            Stimulus : in Unit_Array;
            Weights : in Unit_Array;
            Output : out Real := 0.0);
    end component;

-- Instantiation of all Neural_Nodes to the Neural_Element design unit

for all : Neural_Node use entity work.Neural_element(behavior);

begin

    element_generate:
      for i in 1 to N_Units generate
        Nodes : Neural_node
                    port map ( Network_Stimulus(i), Network_Weights(i),
                               Network_Output(i) );
    end generate;

end network_structure;
```

```
-- This is the top level assembly of all the sub-components. Different signals are
generated and fed to different parts. In essence, it is the Test-bench for the whole
system

use work.Neural_Package.all;
entity ann_processor is
    port (
        Stimulus : in Neural_Array;
        Output : out Neural_Array);
end ann_processor;



architecture processor_structure of ann_processor is

    component memory
      port (
           memory_output : out Weights_Matrix);
    end component;

    component network
      port (
          Network_Stimulus : in Stimulus_Matrix;
          Network_Weights : in Weights_Matrix;
          Network_Output : out Neural_Array);
    end component;

    signal Matrix_Weights : Weights_Matrix := (
      (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0),
      (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0),
      (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0),
      (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0),
      (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0) );

    signal Matrix_Stimulus : Stimulus_Matrix := (
      (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0),
      (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0),
      (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0),
      (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0),
      (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0), (0.0, 0.0, 0.0, 0.0, 0.0, 0.0) );

    for all : network use entity work.network(network_structure);
    for all : memory use entity work.memory(memory_arch);
```

```
begin

    process(Stimulus'Transaction)
       variable tmp1 : integer := 1;
       variable tmp2 : integer := 1;
       variable tmp3 : integer := 1;
    begin
    tmp3 := (N_States - States_per_Stage );
    element_loop1:
       for ii in 1 to N_Units loop

          element_loop2 :
             for k in 1 to States_per_Stage loop
                tmp1 := (ii / States_per_Stage);
                if tmp1 = 0 then
                   tmp1 := N_Stages -1;
                end if;
                tmp2 := ((tmp1 -1) * States_per_Stage ) + k;
                Matrix_Stimulus(ii)(k) <= Stimulus(tmp2);
          end loop element_loop2;

          element_loop3 :
             for l in 1 to tmp3 loop
                tmp1 := (ii / States_per_Stage);
                if tmp1 = N_Stages then
                   tmp1 := N_Stages -1;
                end if;
                tmp2 := ((tmp1) * States_per_Stage ) + l;
                Matrix_Stimulus(ii)(l + States_per_Stage) <= Stimulus(tmp2);
          end loop element_loop3;

    end loop element_loop1;
    end process;

-- read the weight values to be used in the network from the memory

    memory_read : memory port map (Matrix_Weights);

-- feed the Neural Network with these Weights and the Stimulus and get the Output

    run_network : network port map ( Matrix_Stimulus, Matrix_Weights, Output );

end processor_structure;
```

```
-- This is the test bench for testing the whole system. Whole system is integrated in
ann_processor.

use work.Neural_package.all;
entity test_bench is
end test_bench;

architecture test_bench_arch of test_bench is

    component ann_processor
      port (
          Stimulus : in Neural_Array;
          Output : out Neural_Array);
    end component;


 signal kIn : Neural_Array := (0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0,
1.0, 0.0);
 signal kOut : Neural_Array;

 for all : ann_processor use entity work.ann_processor(processor_structure);

begin

    tester : ann_processor port map ( kIn , kOut );

    Process(kOut)
    begin

      kIn <= kOut after 10 ns;
    end process;
end test_bench_arch;
```

# APPENDIX II

## SIMULATION RESULTS FOR THE STAGECOACH PROBLEM

(a) Stimulus: (0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0)

```
TIME|--------------------------------------------------------------------------------SIGNALNAMES--------------------------------------------------------------------------------|
|
(NS)|KOUT(1)KOUT(2)KOUT(3)KOUT(4)KOUT(5)KOUT(6)KOUT(7)KOUT(8)KOUT(9)KOUT(10)KOUT(11)KOUT(12)KOUT(13)KOUT(14)KOUT(15)
|
0|0.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+00
4|0.000000E+001.000000E+000.000000E+000.000000E+000.000000E+001.000000E+000.000000E+000.000000E+000.000000E+000.000000E+001.000000E+000.000000E+000.000000E+000.000000E+001.000000E+000.000000E+00
14|0.000000E+001.000000E+001.000000E+001.000000E+000.000000E+001.000000E+001.000000E+001.000000E+001.000000E+001.000000E+001.000000E+001.000000E+000.000000E+000.000000E+001.000000E+000.000000E+00
18|0.000000E+001.000000E+000.000000E+000.000000E+000.000000E+001.000000E+000.000000E+001.000000E+001.000000E+001.000000E+001.000000E+001.000000E+000.000000E+000.000000E+001.000000E+000.000000E+00
28|0.000000E+001.000000E+000.000000E+000.000000E+000.000000E+001.000000E+000.000000E+001.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+001.000000E+000.000000E+00
32|0.000000E+001.000000E+000.000000E+000.000000E+000.000000E+001.000000E+000.000000E+001.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+001.000000E+000.000000E+00
42|0.000000E+001.000000E+000.000000E+000.000000E+000.000000E+001.000000E+000.000000E+001.000000E+000.000000E+000.000000E+001.000000E+000.000000E+000.000000E+000.000000E+001.000000E+000.000000E+00
```

(b) Stimulus: (0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0)

```
TIME |----------------------------------------------------------------------------------------SIGNAL NAMES----------------------------------------------------------------------------------------|
|
(NS) | KOUT(1) KOUT(2) KOUT(3) KOUT(4) KOUT(5) KOUT(6) KOUT(7) KOUT(8) KOUT(9) KOUT(10) KOUT(11) KOUT(12) KOUT(13) KOUT(14) KOUT(15)
|
 0 | 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00
0.000000E+00
 4 | 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
 14 | 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
 18 | 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
 28 | 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
 32 | 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
```

(c) Stimulus: (0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0)

```
TIME |------------------------------------------------------------------------------------SIGNAL NAMES--------------------------------------------------------------------------------------------|
|
(NS) | KOUT(1) KOUT(2) KOUT(3) KOUT(4) KOUT(5) KOUT(6) KOUT(7) KOUT(8) KOUT(9) KOUT(10) KOUT(11) KOUT(12) KOUT(13) KOUT(14) KOUT(15)
|
0 | 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00
0.000000E+00
4 | 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
14 | 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
18 | 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
28 | 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
32 | 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
42 | 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
46 | 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
```

(d) Stimulus: (0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0)

```
TIME|------------------------------------------------------------------------------------------SIGNALNAMES------------------------------------------------------------------------------------------|
|
(NS)|KOUT(1)KOUT(2)KOUT(3)KOUT(4)KOUT(5)KOUT(6)KOUT(7)KOUT(8)KOUT(9)KOUT(10)KOUT(11)KOUT(12)KOUT(13)KOUT(14)KOUT(15)
|
0|0.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+00
4|0.000000E+001.000000E+000.000000E+001.000000E+000.000000E+001.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+001.000000E+000.000000E+00
14|0.000000E+001.000000E+000.000000E+001.000000E+000.000000E+001.000000E+000.000000E+000.000000E+001.000000E+001.000000E+001.000000E+000.000000E+000.000000E+001.000000E+000.000000E+00
18|0.000000E+001.000000E+000.000000E+000.000000E+000.000000E+001.000000E+000.000000E+000.000000E+001.000000E+001.000000E+001.000000E+000.000000E+000.000000E+001.000000E+000.000000E+00
28|0.000000E+001.000000E+000.000000E+000.000000E+000.000000E+001.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+001.000000E+000.000000E+00
32|0.000000E+001.000000E+000.000000E+000.000000E+000.000000E+001.000000E+000.000000E+001.000000E+000.000000E+000.000000E+000.000000E+000.000000E+000.000000E+001.000000E+000.000000E+00
42|0.000000E+001.000000E+000.000000E+000.000000E+000.000000E+001.000000E+000.000000E+001.000000E+001.000000E+001.000000E+001.000000E+000.000000E+000.000000E+001.000000E+000.000000E+00
46|0.000000E+001.000000E+000.000000E+000.000000E+000.000000E+001.000000E+000.000000E+001.000000E+000.000000E+000.000000E+001.000000E+000.000000E+000.000000E+001.000000E+000.000000E+00
```

(e) Stimulus: (0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0)

```
TIME |------------------------------------------------------------------------------------------------SIGNAL NAMES------------------------------------------------------------------------------------------------|
|
(NS) | KOUT(1) KOUT(2) KOUT(3) KOUT(4) KOUT(5) KOUT(6) KOUT(7) KOUT(8) KOUT(9) KOUT(10) KOUT(11) KOUT(12) KOUT(13) KOUT(14) KOUT(15)
|
0 | 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00
0.000000E+00
4 | 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
```

(f) Stimulus: (0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0)

```
TIME |------------------------------------------------------------------------------------------SIGNAL NAMES------------------------------------------------------------------------------------------|
|
(NS) | KOUT(1) KOUT(2) KOUT(3) KOUT(4) KOUT(5) KOUT(6) KOUT(7) KOUT(8) KOUT(9) KOUT(10) KOUT(11) KOUT(12) KOUT(13) KOUT(14) KOUT(15)
|
 0 | 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00
0.000000E+00
 4 | 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
 14 | 0.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
 18 | 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
 28 | 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
```

(f) Stimulus: (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)

```
TIME I--------------------------------------------------------------------------------SIGNAL NAMES--------------------------------------------------------------------------------I
I
(NS) I KOUT(1) KOUT(2) KOUT(3) KOUT(4) KOUT(5) KOUT(6) KOUT(7) KOUT(8) KOUT(9) KOUT(10) KOUT(11) KOUT(12) KOUT(13) KOUT(14) KOUT(15)
I
0 I 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00
0.000000E+00
4 I 1.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00
1.000000E+00
18 I 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
32 I 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
46 I 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
60 I 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
74 I 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
88 I 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 0.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
102 I 0.000000E+00 1.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 0.000000E+00 1.000000E+00
0.000000E+00
```

*

*

*

# BIBLIOGRAPHY

# BIBLIOGRAPHY

[1] J.J. Hopfield and D.W. Tank, "Neural Computation of Decisions in Optimization Problems," *Biological Cybernetics*, Vol. 52 , 1985, pp. 141 - 152.

[2] C. Mead, Analog VLSI and Neural Networks, Addison-Wesley, 1989.

[3] H.P. Graf, "VLSI Implementation of a Neural Network Memory with Several Hundred Neurons," *Proc. of AIP Conf. on Neural Networks for Computing*, No.151, 1986, pp. 182-187.

[4] A. S. Gilman, "VHDL - The Designer Environment," *IEEE Design & Test of Computers*, April 1986, pp. 42 - 47.

[5] V.D. Agrawal, "The Linguistics of Design and Test," *IEEE Design & Test of Computers*, April 1986, page 8.

[6] A. Dewey and A. Gadient, "VHDL Motivation," *IEEE Design & Test of Computers*, April 1986, pp. 12 - 16.

[7] J.D. Nash and L.F. Saunders, "VHDL Critique," *IEEE Design & Test of Computers*, April 1986, pp. 54 - 65.

[8] Hillier and Lieberman, Introduction to Operations Research (Fourth Edition), Holden - Day Inc., Oakland, CA., pp. 332 - 336.

[9] B.D. Shriver, "Artificial Neural Systems,", *Computer*, March 1988, pp. 8 - 9.

[10] C. Chiu, C.Y. Maa and M.A.. Shanblatt, "An Artificial Neural Network Algorithm for Dynamic Programming," to appear, *International Journal of Neural Systems*.

[11] VHDL Language Reference Manual, Version 7.2, Technical Report IR - MD - 045 - 3, Intermetrics, Bethesda, MD., 1 Jan, 1987.

[12] R. Lipsett, C. Schaefer and C. Ussery, VHDL : Hardware Description and Design, Kluwer Academic Publishers, 1989.

[13] D.R. Coelho, The VHDL Handbook, Kluwer Academic Publishers, 1989.

[14] S.S. Leung and M.A. Shanblatt, ASIC System Design with VHDL: A Paradigm, Kluwer Academic Publishers, 1989.