Optimization of FPGA Based Neural Network Processor

by

Ivan Teh Fu Sun

Dissertation submitted in partial fulfillment of
the requirements for the
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

JUNE 2004

Universiti Teknologi PETRONAS
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

CERTIFICATION OF APPROVAL

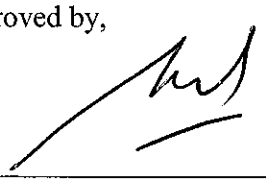
Optimization of FPGA Based Neural Network Processor

by

Ivan Teh Fu Sun

A project dissertation submitted to the
Electrical & Electronics Engineering Programme
Universiti Teknologi PETRONAS
in partial fulfillment of the requirement for the
BACHELOR OF ENGINEERING (Hons)
(ELECTRICAL & ELECTRONICS ENGINEERING)

Approved by,



(Mr. Noohul Basheer bin Zain Ali)

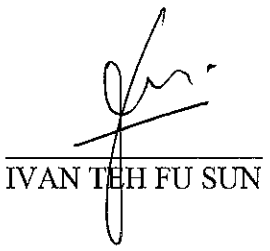
UNIVERSITI TEKNOLOGI PETRONAS

TRONOH, PERAK

June 2004

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.



IVAN TEH FU SUN

ABSTRACT

Neural information processing is an emerging new field, providing an alternative form of computation for demanding tasks such as pattern recognition problems which are usually reserved for human attention. Neural network computation is sought after where classification of input data is difficult to be worked out using equations or sets of rules.

Technological advances in integrated circuits such as Field Programmable Gate Array (FPGA) systems have made it easier to develop and implement hardware devices based on these neural network architectures. The motivation in hardware implementation of neural networks is its fast processing speed and suitability in parallel and pipelined processing.

The project revolves around the design of an optimized neural network processor. The processor design is based on the feedforward network architecture type with BackPropagation trained weights for the Exclusive-OR non-linear problem. Among the highlights of the project is the improvement in neural network architecture through reconfigurable and recursive computation of a single hidden layer for multiple layer applications. Improvements in processor organization were also made which enables the design to parallel process with similar processors. Other improvements include design considerations to reduce the amount of logic required for implementation without much sacrifice of processing speed.

ACKNOWLEDGEMENT

I would like to thank Mr. Noohul Basheer bin Zain Ali for his guidance and support which without this project would not have completed. I would also like to thank Dr. Varun Jeoti for his advice and all the staff in the Electrical and Electronics Engineering faculty for their time and technical expertise.

Special thanks to Professor Mark J. Embrechts from Rensellaer Polytechnic Institute, Troy, New York for the use of his MetaNeural™ software which have been an integral part of the project.

Last but not least, to my family and friends who had been a tremendous source of moral and mental support before and present.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1 INTRODUCTION

1.1 Background of Study

Neural networks is still considered a relatively new area of research, with development picking up speed only in the 1990's since the first mathematical model of the biological neuron was presented by McCulloch and Pitts in the 1940's. Neural networks are not confined to solely an attempt of replicating the human brain, it has as well, wide reaching applicability and its concepts are incorporated into applications such as optical character recognition, machine health monitoring as well as stock market forecasting (Callan, 1999, p.2).

Neural networks can be implemented using software simulations and as fast hardware devices. The former platform has been more often used for the development of neural networks because it is cheaper and is more flexible for research purposes. However, with the rapid advances in Field Programmable Gate Array (FPGA) technology, the option to implement hardware based neural networks now seems more appealing due to its customizability, relatively faster processing speed and more importantly the infrastructure to tap into the neural network's intrinsic property of parallel processing.

1.2 Problem Statement

The focus of most engineering and scientific groups on neural networks is to produce working models of neural networks into hardware designs. These implementations are usually made up of neural processing units known as neurons which take up considerable amounts of logic gates to implement. Thus the complexity of a basic neuron design affects very much the capacity of neurons which can be fitted into a

fixed amount of silicon real estate. On the other hand, not much work has been put into increasing the performance and processing speed of these neural network processors. The successful addressing of these areas would pave the way for cheaper and faster neural network processors.

1.3 Objectives and Scope of Study

The project undertaken will be focused on the optimization of FPGA based neural network chips in terms of logic gate numbers, processing speed, and performance. The project will be basically a study of neural networks and implementation of logic design onto FPGA.

The targeted result of this project would be an FPGA implementation of a neural network architecture which is suited for the supervised learning paradigm. The connectivity of the feedforward neural network architecture would be of a multiple layered type.

The end product would be an Artificial Neural Network FPGA implementation with at least 2 inputs and 1 output. The architecture should be able to implement 2 or more neuron hidden layers. Components inherent to a neuron such as multipliers, adders and activation functions will be part of the overall design. The ultimate goal of the project is to produce an optimized and functioning neural network processor which is able to solve linear and nonlinear problems. The benchmark of the project would be utilising the end product to solve the nonlinear Exclusive-OR (XOR) problem.

Due to time constraint and design considerations, the supervised training phase of the neural network would be performed separately using the MetaNeural™ software. The resultant weights and bias values for each connection between neurons would then be directly programmed into the FPGA design.

CHAPTER 2

LITERATURE REVIEW AND THEORY

2 LITERATURE REVIEW AND THEORY

2.1 Artificial Neural Network

Common to neural network architectures are the simple processing units known as "neurons" as is described in Figure 2.1 below. Neural networks are made of these simple neurons with different architectures dictating how a collection of neurons are interconnected as well as how calculations are made to adjust the weighted inputs such that it can function as is intended.

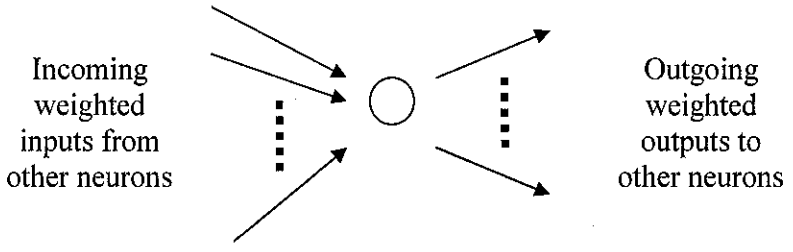


Figure 2.1: A single neuron model

Typical neural network architecture is made up of an input layer, one or more hidden layers, and an output layer. The input layer merely acts as a buffer to external inputs, whereas the output layer functions as a last stage hidden layer with the output buffered and passed on to external outputs. The more interesting section of the neural network would be the hidden layer(s) in which the neurons reside. The number of input and output nodes for their respective layers is dependent on the application in which the neural network was designed for. In example, an application which recognizes alphabets using a 10 x 10 pixelizer (total 100 pixels) would require a neural network which has 100 inputs and 26 outputs. However, there is no

convention in which the number of hidden layers to be used for a particular application can be directly generated. To add to the confusion, the number of neurons per hidden layer is also arbitrary. Thus the most optimized number of hidden layers and neurons for each layer are largely decided using trial and error methods. Figure 2.2 below is a typical neural network architecture used for applications which require only 2 inputs and one output; the number of hidden layers and neurons are chosen arbitrarily. Simple problems such as 2 input AND, OR or XOR can be solved using this network configuration.

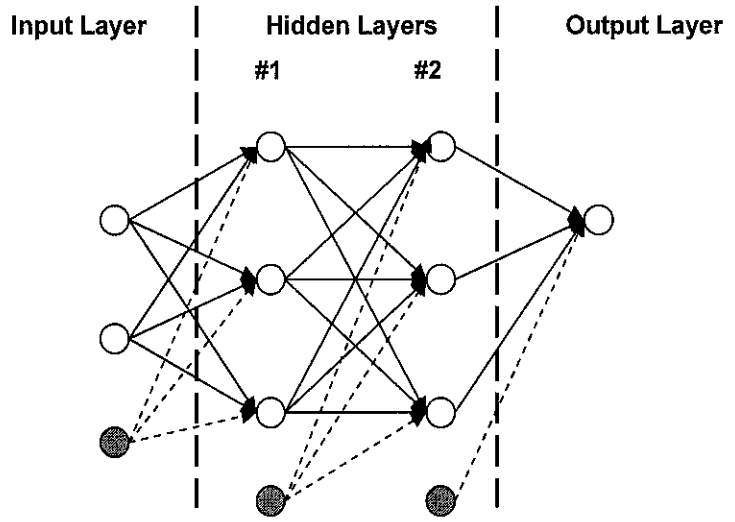


Figure 2.2: A 2-3-3-1 Neural Network Architecture

The connections between two neurons or nodes from adjacent layers have weights which are multiplied to the output of the neuron/node which feeds to the neuron of the subsequent layer. The coloured nodes represent storage for bias values which are predetermined like the weights of each connection. Each neuron for each hidden layer has its own unique bias value.

The combination of these weights and bias values allows a trained neural network which is subjected to a set of inputs to provide a correct categorization of them as an output. The processing of these weights in a neuron is divided into 2 stages: Summation of weighted inputs and the mapping of the summation output to an activation function. Firstly, all the weighted inputs from the preceding layer are

summed up together with a predetermined bias value. The result from this phase is then normalised using an activation function such as the identity function, binary threshold function or the sigmoid function. The normalised result is then fed into the subsequent hidden layer where the cycle continues until the output layer is reached. Figure 2.3 below illustrates the summation and activation function computations of a neuron.

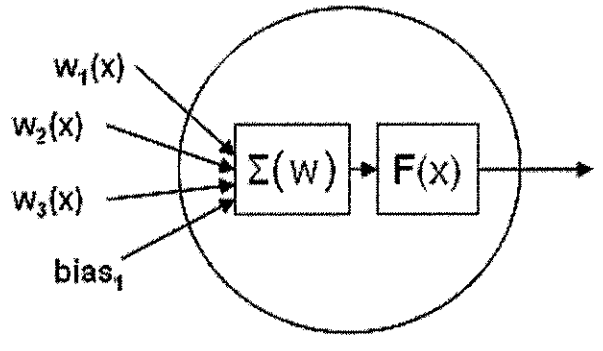


Figure 2.3: Computations within a Neuron

An important attribute of a neural network system based on weights is its capability to learn and generalize variations in a set of inputs (Picton p., 1994 p.4). For example, in character recognition applications, the same character 'A' can be written in a multitude of ways, however these variations would produce the same output when presented to a successfully trained and tested neural network.

2.2 Classification of Artificial Neural Network Models

Neural networks are organized and classified according to 4 attributes. These are: learning paradigm, network architecture, network connectivity and learning algorithm.

2.2.4 Learning Paradigm

The above concerns on the type of learning or training that a neural network is subjected to, be it either supervised, unsupervised or a hybrid of both methods. The learning paradigm determines which type of learning algorithm that can be used when training the neural network.

2.2.5 Network Architecture

Generally there are two types of network architecture, feedforward and recurrent. These architecture types are descriptions of how the interconnections between neurons in a neural network are made. The connection type for the neural network in Figure 2.2 is of the feedforward type.

2.2.6 Network Connectivity

Not to be confused with network architecture, network connectivity describes how neurons are positioned within a neural network. For example, single layer connectivity describes a neural network with only one hidden layer. Other connectivity types are; Self Organizing Map, Multilayer and Hopfield. Connections between hidden layers and neurons are decided by the network architecture.

2.2.7 Learning Algorithm

The learning algorithm deals with training the particular neural network chosen. There are many types of learning algorithm such as BackPropagation, Associative Memory, Madaline and more. The application of a particular learning algorithm depends on its suitability with the three parameters mentioned earlier of a particular neural network. For example, BackPropagation learning algorithm can only be applied to a neural network which is based on supervised learning paradigm and has feedforward architecture.

The taxonomy of Artificial Neural Network is shown in **APPENDIX A**[1]. A table showing the similarities and differences between the Von Neumann computer model and neural networks is available in **APPENDIX B**.

2.3 Field Programmable Gate Array (FPGA)

FPGA technology is given preference over other options such as VLSI (Very Large Scale Integrated) circuit, ASIC (Application Specific Integrated Circuit) and MPGA's (Mask Programmed Gate Arrays) for the implementation of neural networks due to its flexibility to be reconfigured while providing all the advantages inherent to hardware devices such as low sensitivity to electric noise and

[1] Reproduced from [4] pg 6

temperature, memory of weight storage, processing speed and parallel processing. Besides technical advantages, implementations on experimental projects using FPGA's have lower non-recurring engineering costs as well as faster development and implementation processes. With its reprogrammability features, any design defects on FPGA can be easily corrected and tested, thus shortening the time-to-market. Studies on computer architectures provide well documented optimization techniques which can be incorporated into the neural network hardware design.

FPGA technology uses Hardware Description Language (HDL) to design circuits. Among the more popular are VHDL and Verilog. Both HDL's are similarly powerful and usage of either or both is up to preference. There are 3 different levels at which a circuit design can be specified: behavioural, dataflow and structural. Behavioural style of coding is at the highest level in terms of similarity with natural language whereas the most specific is the structural style. Coding structurally would make the design easier to synthesize and also provide more control over the physical assignation of the circuit design. A design can be a specified using one or more coding styles.

Hardware design using FPGAs are usually performed using both combinational and sequential logic. These designs are specified using any of the three hardware description methods; which are the structural, dataflow and behavioral. For design of complex circuits, dataflow and behavioral styles are preferred because it frees the designer from having to fully specify the connections and logic gates to be used.

Design of hardware using the dataflow method is used more often to design combinational circuits. The use of the keyword "assign" in Verilog for the dataflow method refers to a combinational logic assignment. On the other hand, behavioral design method uses the "always" block for the design of sequential logic. Specification of conditions such as "posedge" and "negedge" for certain signals dictate when the procedural block is executed or triggered.

The Verilog hardware description language (HDL) is a very powerful simulation language. However, only about 10% of its instructions are synthesizable [3]. Synthesis refers to the ability to successfully convert from codes to actual hardware

logic implementation. Thus the hardware designer has to be aware of which instructions are synthesizable friendly as well as some designing rules that may cause the design to not being able to be implemented in hardware. Instructions that are not synthesizable are those which deal with timing parameters such as "time, wait, initial, delay" and others such as "fork, join, defparam, UDP".

There are also other restrictions such as those concerning signals which are either wires or registers. For example, it is illegal to connect two registers together inside a module, which would be a common mistake for new designers who are usually comfortable with software programming style of liberally using variables.

2.4 Optimizing the Design

To produce an optimized design, there are two things to consider, first would be the algorithm of the design architecture, and second, the optimization of the implementation itself. This section would be dedicated to the second kind of optimization. Optimization of logic implementation would require the knowledge of how FPGAs work. Xilinx FPGAs are made of arrays of Configurable Logic Blocks which normally houses two 4-input LUTs feeding a pair of flip flops [2]. To produce an optimized implementation, the designer has to constantly think of how the codes will be implemented into hardware given the amount and type of available resources.

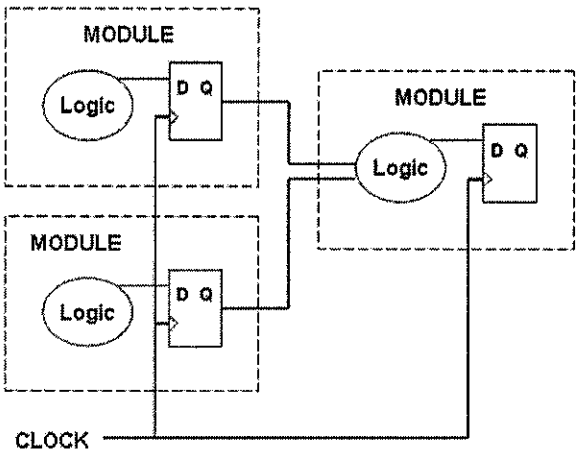


Figure 2.4: Suggested Module Boundary Selection and Register Assignment

Hardware designs are usually segregated into several modules which are then linked to each other and controlled using a control unit which is made a state machine. Each module is best designed to have its module boundaries as specified in Figure 2.4 which basically means that all outputs are to be buffered using registers. Using this standardization would mean that synchronizing flip flops on inputs are not necessary.

The synthesizer optimizes combinational logic within modules. Therefore it would be advisable that related combinational input and output of related signals to be grouped together in the same module so that there would not be any redundancy which is harder to detect if specified in separate modules. In hardware design, sequential logic or state machines are preferably synchronous. This is to reduce propagation delay as well as chances of timing problems occurring.

2.5 Completing the design

The next steps of the design process after the completion of the RTL (Register Transfer Logic) codes would be the synthesis and the "place and route" process. What the synthesis process does is to generate a netlist which is also a Verilog file that represents the higher level Verilog codes presented to it into lower level structural/gate level format. Information that is provided to the synthesis tool are for the FPGA target chip model which will be used to invoke the appropriate library file containing all physical and timing parameter information and implementation of Verilog constructs into gates.

From synthesis, the designer would be able to know how fast the design could run, which depends on many factors such as the longest length of combinatorial propagation delay. For FPGA, problems usually encountered in ASIC design such as clock skew and signal strengths do not have to be worried as this are already accounted for by the FPGA library file by appropriately assigning clock and signal buffers which are available in the FPGA chip itself. Thus the synthesis process is rather straightforward for FPGA based designs, with design constraint usually dependant on the achievable processing speed.

After successfully synthesizing, the netlist file is passed to the place and route portion of the application to simulate the physical implementation of the design. This process ensures that the design does not violate the boundary parameters of the target

FPGA chip. It is also possible that manual routing maybe required to produce an optimized design. The output of this process is a gate.v file which is also a Verilog file which can be fed back into the RTL simulator for simulation. The process of RTL coding synthesis and place and route is continuous and is repeated until the specifications of the design are met.

CHAPTER 3

METHODOLOGY AND PROJECT WORK

3 METHODOLOGY AND PROJECT WORK

3.1 Project Methodology

The project involves the completion of the following four stages; literature research, specification of neural network architecture design, Field Programmable Gate Array (FPGA) training and experimentation, FPGA implementation and training of neural network, and optimization of FPGA design. The project design approach would be based on the "waterfall" process model as shown in Figure 3.1 below.

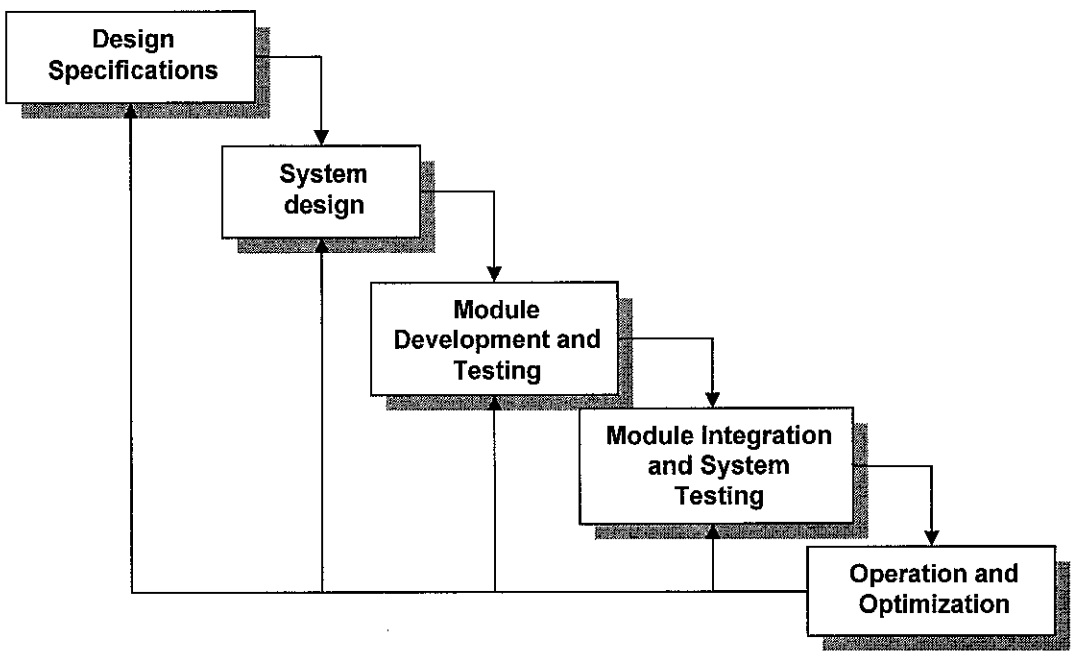


Figure 3.1: ANN System Process Model for FPGA Implementation

3.1.1 Literature Research

Early stages of the literature review were performed to enhance and widen the author's knowledge on the different types of neural networks architecture. Later on, research on FPGA technology and also a study on Hardware Description Language (HDL) follow.

3.1.2 Specification of Neural Network Design

The next stage of the project was to specify the characteristic of the neural network architecture to be designed and implemented onto FPGA. The specification of the neural network architecture is important as it would serve as a guide and reference for later design stages. Samples of specification to be included would be the number of inputs and outputs, algorithms and architecture for logic blocks, functional specification of the control logic as well as the expected results of the design.

3.1.3 FPGA Training and Experimentation

FPGA training and experimentation would be conducted in parallel with the preceding stage to get familiarized with the devices and related HDL programming software. Part of the experiments or exercises on FPGA would be to design modules such as multipliers and fast adders and also to learn to write testbenches that will be used to validate the functionality of the design. Aldec's Active HDL software would be used to develop and simulate the functionality of the HDL codes.

3.1.4 FPGA Implementation and Testing of Neural Network

RTL implementation of neural network followed by optimization was performed after sufficient skill is gained. Specified modules would be developed separately and tested. These modules would then be integrated into a complete design. Testing at both the Register Transfer Level (RTL) would be performed to ensure that the design performs reliably. The supervised learning portion using the back propagation algorithm of the design will be performed off line using a shareware program known as MetaNeural™.

3.2 Equipments and Software Applications Used

3.2.1 Xilinx Virtex-II XC2V1000 Reference Board

Based on research on previous implementations of neural networks onto FPGA, the designs documented usually take up a sizeable portion of the target FPGA chip used. Some of the designs are even implemented using multiple FPGA chips. From this it seems appropriate that the largest FPGA in terms of gate numbers available in our labs be selected for the purpose of this project. Due to the experimental nature of the project, a larger gated FPGA would easily allow for variations of neural networks to be designed and implemented. Restrictions on the design would then be minimal. More features of the Virtex FPGA chip is discussed below.

"The highest performance designs are tailored for the target FPGA" as mentioned by Coffman (pg 221 [3]). FPGA vendors such as Altera and Xilinx build their devices differently from each other; their line of products which may very well vary between themselves, differing in pin counts, routing density, logic block construct, availability of RAM blocks and more. Therefore it is of much importance for an HDL FPGA designer to find out about the limitations and advantages of the devices they are dealing with.

The Virtex-II FPGA series are the latest devices offered by Xilinx. It is built using 0.15 micron lithography and 8-layer metal technology. Different from conventional FPGA devices, the Virtex series is of a hardwired version. A hardwired FPGA version improves the performance of conventional FPGA as well as contributing towards a smaller silicon die footprint.

The FPGA chip that is used for the project is the Xilinx Virtex II XC2V1000-4FG256C. This chip has 256 pins of which 172 are usable as input/output pins. As like all Xilinx devices, the Virtex II chips are made up of an array of Configurable Logic Blocks (CLB). Each of these CLB's contain function generators, carry logic, arithmetic logic gates, wide function multiplexers and storage elements which will be appropriately interconnected according to design. Unique to the Virtex board, is built in resources for Look-Up Table (LUT) of up to 8 inputs, which is useful for implementing mathematical functions such as a sigmoid function. The chip used also

has 40 blocks of 18kbits Select RAM resources which can be used to store connection weights for the neural network. Another feature built into the device is 18 by 18 bit multipliers. Usage of these built in multipliers would considerably speed up the design as compared to manually customized ones.

3.2.2 MetaNeural™

MetaNeural™ is a shareware program originally developed by Professor Mark J. Embrechts of Rensselaer Polytechnic Institute, New York (USA) in 1988 as a demonstration package for a lecture course. The application has since evolved and given a user interface as seen in Figure 3.2. The application is used to train feedforward neural network architecture using the BackPropagation learning algorithm. The program is able to work with network architectures which have up to 3 hidden layers while allowing an arbitrary number of neurons per hidden layer.

The list below summarizes the core features and user customization allowed by the MetaNeural™ application. Items listed are illustrated in Figure 3.3.

- Specification of neural network architecture up to 3 hidden layers.
- Specification of number of training epochs.
- Error threshold to stop training
- Training rate which affects the amount of weight adjustment for each training cycle.
- Selection of activation function type
- Easy text format for training pattern and test pattern input files.

The MetaNeural™ software application will be used in this project to supply the values for connection weights and neuron biases to be designed into the FPGA implementation of the neural network. The neural network for an intended application can be trained given a set of training pattern which for this case would be the truth table for the XOR logic.

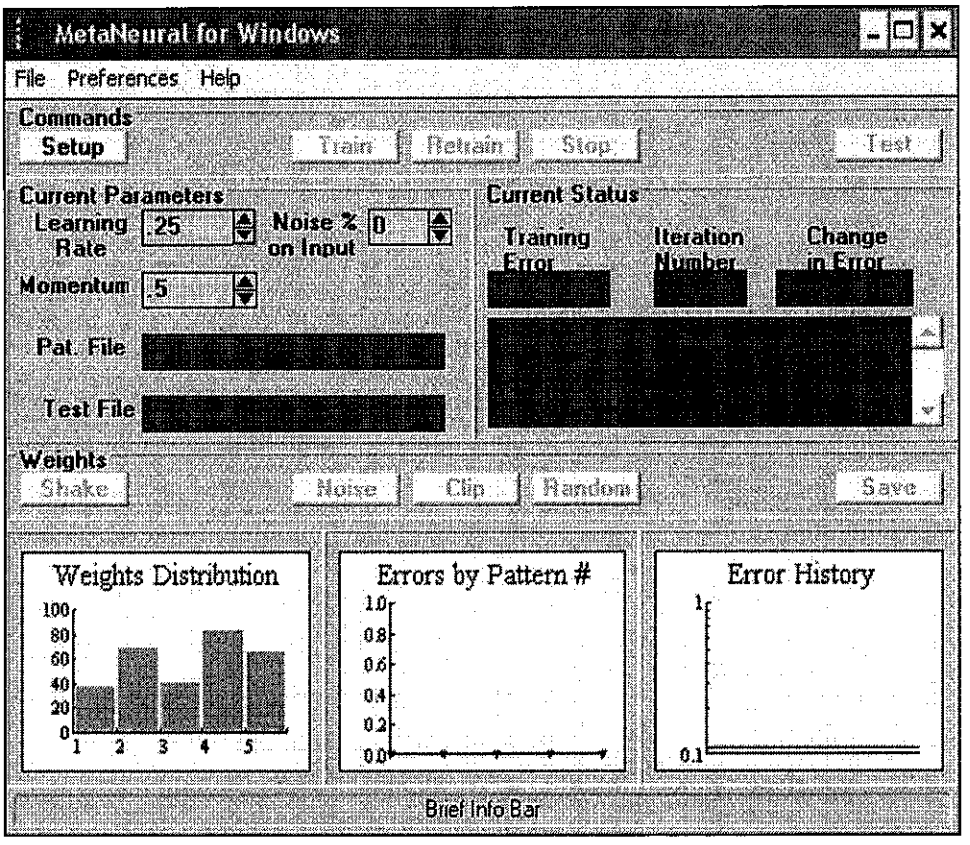


Figure 3.2: Screenshot of MetaNeural™ Main Interface

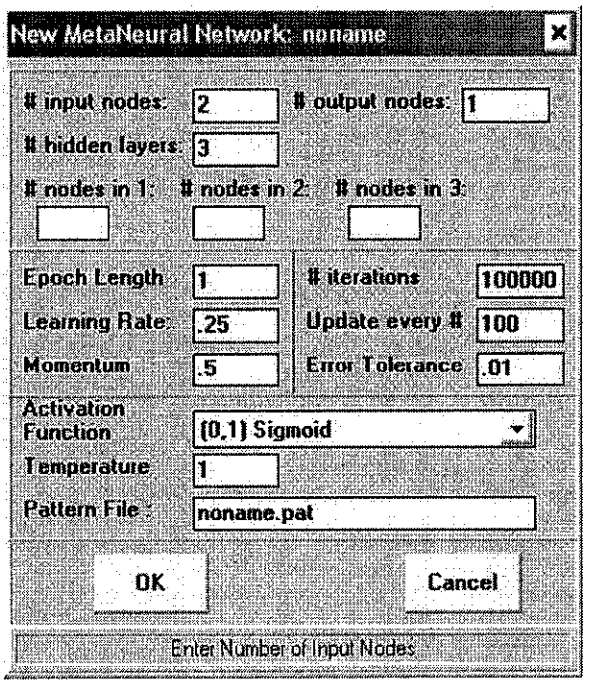


Figure 3.3: Screenshot of MetaNeural™ Network Setup Interface

3.3 Hardware Design Flow

The hardware implementation of the RFNNA processor is divided into several tasks. The approach taken as seen in Figure 3.4 is to first define the modules that will be present in the design such as the neuron module and activation function LUT.

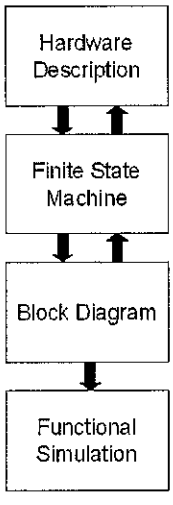


Figure 3.4: Design Flow for Verilog based Register Transfer Logic

The second step would be to design a control unit which is basically a state machine. The state machine would provide control signals to all modules so that processing is performed whenever it is intended. The block diagram editor provided with the Aldec Active HDL compiler allows the designer to have an overview of the entire design. This is also the place where the inputs and outputs of modules and state machines are connected to each other.

The process of functional simulation and verification takes place in intermediate module design stages and also for structural testing when the design is complete. The verification process is performed by applying simulators either by using readily available tools or by using a testbench. A testbench provides the designer with more control regarding the stimulus which is subjected to the module unit under test. Stimuli are essentially generated input signals inclusive of the clock which triggers the module. The output of the module and intermediate net and register status are checked for any discrepancies.

3.4 Functional Simulation Using Testbenches

Testbenches for this project are written using Verilog; however they are not subjected to instruction and optimization constraints as imposed for synthesizable modules. Timing parameters such as delays are applicable.

Modules are usually tested by asserting inputs and clocks going into the module. The assertions of inputs are chosen to reflect actual operating as well as timing conditions.

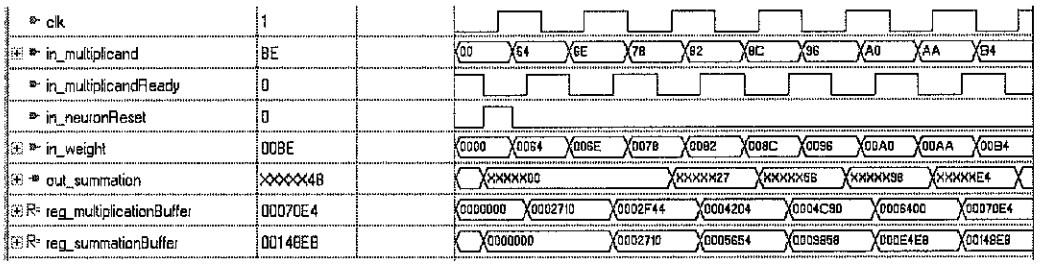


Figure 3.5: Screenshot of Simulated Input Signals and Output Registers

The simulation signals are as shown in Figure 3.5, all status of inputs, outputs and declared registers can be added into the watch window and monitored.

CHAPTER 4

RESULTS AND DISCUSSION

4 RESULTS AND DISCUSSION

Development of the optimized neural network processor is divided into two equally important parts. The initial part was to comprehensively describe an optimized architecture as well as specifications of the neural network that is to be implemented into hardware, whereas the second part was to design the neural network processor modules and optimize the organization of the neural network processor which is based on the architecture specified in the first part.

Below is a description of the subtopics that will be discussed in this chapter:
Research and analysis performed on existing neural network implementation into Field Programmable Gate Array (FPGA) and arithmetic computations in binary circuits led to the discovery of an innovative hardware implementation architecture. This implementation is termed and referred to from here onwards as the Reconfigurable Feedforward Neural Network Architecture (RFNNA). More discussion on this architecture is available under the heading **Architecture Optimization (4.1).** A unique numbering convention to optimize the information processing of a neural network has also been worked out with more discussion available under **Number Convention (4.2).** A Visual Basic application was produced to help model and analyze the implementation of both findings. A description of the application is shown in the subtopic **RFNNA Simulator (4.3).** With a comprehensive specification of the desired neural network, the project proceeded with the actual hardware design of the neural network processor. Details of the modules and how they are optimized in terms of organization are available under **Neural Network Processor Modules (4.4) and Multiplier Bus (4.5).**

4.1 Architecture Optimization

4.1.1 Neural Network Architecture and Learning Algorithms

There are two types of neural network architectures: Feedforward and Recurrent [13]. The former has the advantage of being less complex in its connections between layers of neurons and there are no connections between neurons within the same layer. The recurrent architecture on the other hand necessitates connections between adjacent layers as well as connections intra layer.

Implementation wise, the feedforward architecture would be more appealing due to the considerable amount of logic gates that would be saved without implementing these extra connections. It may be argued that since a neural network application may perform better in a particular type of architecture than another, one cannot just take simplicity of implementation to choose between which architecture to be used. However, the back propagation supervised learning algorithm which is widely used in most applications works in feedforward neural network architecture. Having a choice on which architecture to be used for a general purpose neural network FPGA implementation, the feedforward architecture would inarguably be selected for its simpler implementation and wide applicability.

Thus it is decided that a neural network architecture which is based on the feedforward architecture be used for the FPGA design implementation in this project. Since the training portion of the neural network function will be performed separately, the architecture would allow all types of learning algorithms which fall under the feedforward neural network architecture branch (refer **APPENDIX A**). Besides the mentioned back propagation algorithm, other learning algorithms such as the Adaptive Linear Network (Adaline), Multiple Adaptive Linear Networks (Madaline) and Perceptron can be used.

4.1.2 RFNNA

Following the decision to use the feedforward neural network architecture as the basis for implementing the design, more thought has been given to optimizing the logic gate area in which this can be implemented. Review of other works [8][9] on neural network implementation into FPGA mentions about time multiplexing the

resources for connections between neurons. This however would slow down the processing speed of the design.

Instead of time multiplexing the resources within a layer by reusing the same neuron module for each neuron in a hidden layer, the resources for a hidden layer can be time multiplexed so that a full parallel implementation of a neuron layer is reused by subsequent hidden layers. This will overcome the disadvantage of reducing the processing speed. The ability to pipeline the initialization of weights and other resources while other stages are processing would make the design more efficient. Taking the idea further by adding controls to determine the number of hidden layer iterations and connectivity patterns, we would then have a multiple connection architecture neural network at hand. This architecture is given the name Reconfigurable Feedforward Neural Network Architecture (RFNNA).

4.1.3 RFNNA for XOR Problem

The RFNNA example shown in Figure 4.1 is developed to be able to solve the nonlinear XOR problem. The circuit is designed to have only two inputs and one output. The hidden layers have a maximum number of 2 neurons. The number of hidden layers and neurons for each layer to be used is arbitrary and selectable. The resources are designed to be fully utilized at each layer where the same resources are used recursively for each hidden and final output layer computation.

Pipelining is used to update the value of weights for the next hidden layer's multiplier during initialization as well as when summation and mapping of activation function for the current layer is being performed. Not included in the diagram is the control logic, which controls the uploading of weights and activation of logic switches to ensure proper network connections between registers. The logic switches connects and disconnects accordingly to ensure proper network connectivity. The control logic also determines the number of computation iterations which in turn depends on the number of hidden layers selected. This logic switch shows the connection path for the output layer.

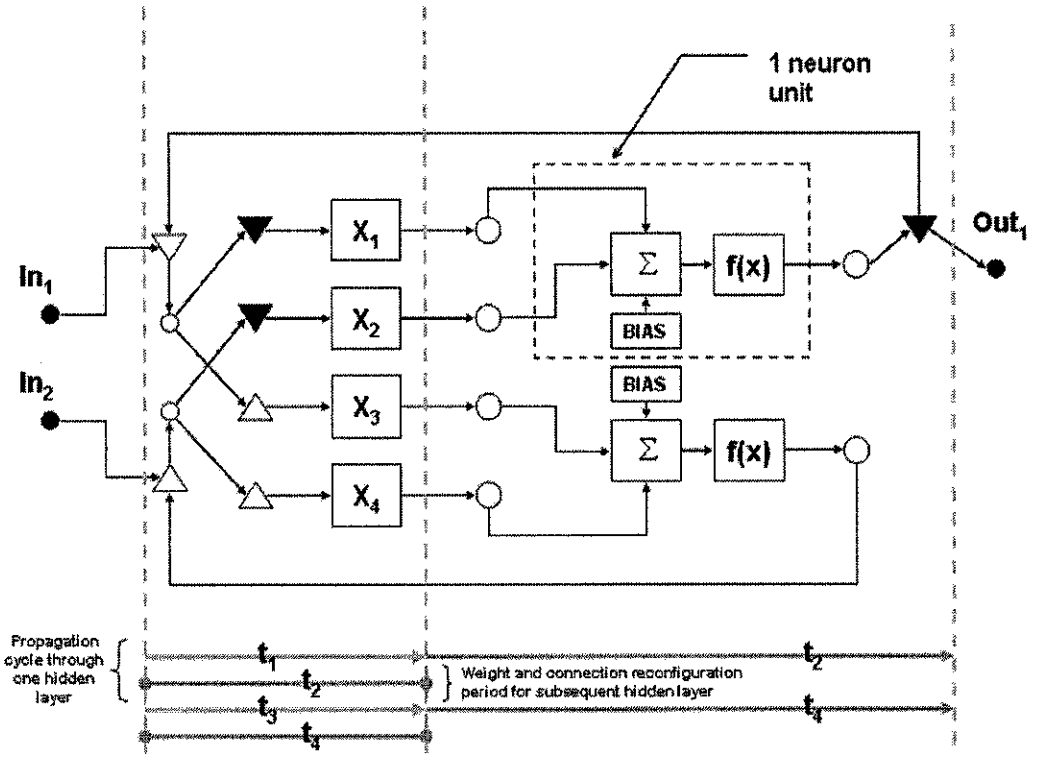


Figure 4.1: Reconfigurable Feedforward Neural Network Architecture and Execution Phase Diagram for XOR Problem

Of particular interest is the way the multiplier constants are updated. As seen in Figure 4.1, the weights for subsequent hidden layer which are stored in memory would be fetched and stored as the multiplier constants while the current layer is still executing its summation and activation function transformation. This pipelined method reduces the processing speed difference between this architecture from a full cascaded parallel implementation.

4.2 Number Convention

There are 3 arithmetic operations that a neural network computes as is shown in Figure 2.3. These are multiplication, summation and conversion of inputs using an activation function.

Generated weights of a trained neural network are not integers rather they are represented using decimal numbers. As computation of arithmetic functions using

logic circuits are in binary format, these decimal values would have to be substituted using a suitable numbering convention in which the circuit is able to manipulate with. The usual representation of decimal numbers in binary format are the sign-magnitude representation, twos complement representation and the floating point representation.

4.2.1 Twos Complement and Floating Point Representation

Twos complement is a variation of number representation for integers. It is similar to the sign magnitude representation for positive values but is different when representing negative values. Unlike the sign magnitude representation, twos complement does not use an extra MSB as a sign bit but is automatically represented when translated to binary format. The main advantage of twos complement over sign magnitude is that it simplifies mathematical operations such as addition and subtraction, no extra logic is required to test for the polarity of the sign bit. Two's complement and sign-magnitude numbering convention are described as follows.

Sign Magnitude

$$N = \sum_{i=0}^{n-2} 2^i a_i \qquad \text{if } a_{n-1} = 0 \qquad \textit{(equation 4.1)}$$

$$N = -\sum_{i=0}^{n-2} 2^i a_i \qquad \text{if } a_{n-1} = 1 \qquad \textit{(equation 4.2)}$$

Twos Complement

$$N = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i \qquad \textit{(equation 4.3)}$$

Floating point numbers are used when the numbers to be represented are spread across a wide range. It is the binary equivalent of the decimals scientific notation. A floating point number is divided into 3 sections (Figure 4.2); the sign bit, significand or mantissa and the exponent.

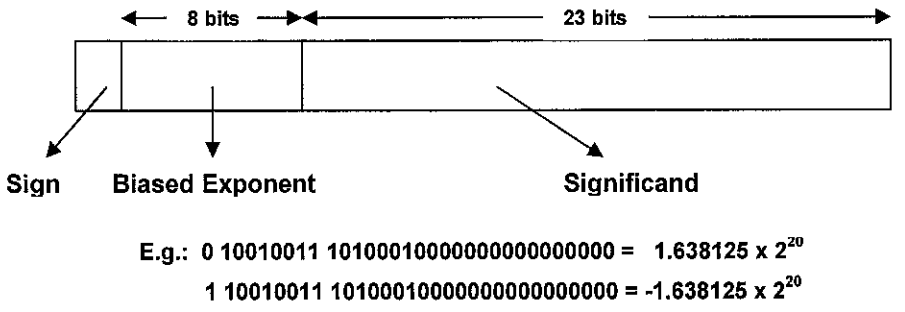


Figure 4.2: 32-Bit Floating Point Format [7]

Mathematical operations performed using floating point representation is generally more complex and require substantially more logic to implement when compared to twos complement. Comparing multiplication operations between twos complement and floating point representation, the whole multiplication operation of a twos complement representation is performed only in the significand section of the floating point representation. Extra logic is required to manipulate the exponent bits to reflect the changes made by multiplying the significand of two floating point numbers.

4.2.2 Fixed Point with Fractional Component

From findings, it is best that a floating point representation convention be avoided. Besides being complex in its implementation, its significand component cannot be directly used to address the LUT for which the activation function is to be implemented. The storage of a number would also use much more memory.

Although the computation within a neural network contains decimal numbers which may seem more suited for floating point implementation, usage of twos complement or sign magnitude is possible due to the numbering range which is always limited after each hidden layer by the sigmoid function (Figure 4.3) to between 0 and 1.

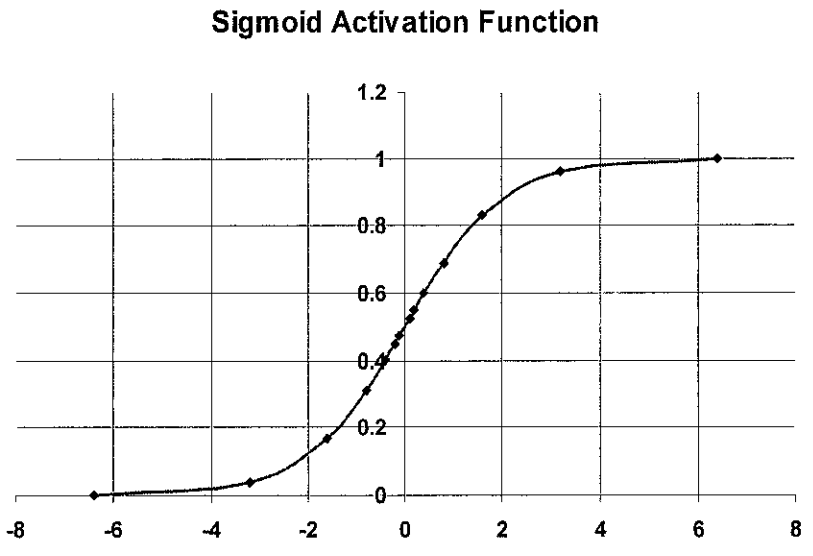


Figure 4.3: Graphical Plot of a Sigmoid Function

From equation 4.4, the sigmoid function involves division and the natural exponential function. Direct mathematical implementation of this function in hardware would be difficult due to the number of mathematical steps involved when the function is breakdown into fundamental operations such as add, subtract, divide and multiply. Thus a Look-Up-Table (LUT) is used to plot out the function with the value of "*net*" as its pointer.

Sigmoid Function

$$f(net) = \frac{1}{1+\exp(-net)} \qquad (equation\ 4.4)$$

This small and confined range of operation eases the use of twos complement representation. Besides representing integers, twos complement representation can also be used to represent fractions.

The size of each fraction equivalent to 1 binary value is determined by the accuracy of the computation as well as hardware implementation constraints. As was mentioned before in Chapter 3.2, the maximum implementable LUT size for the Virtex 2 FPGA chips is at 8 bits. This gives $2^8 = 256$ memory words which are 8 bits wide. From observation of Figure 4.3, the sigmoid function provides distinct values

between the ranges on the x-axis of -6.4 to 6.4. Corresponding values outside of this range is clipped off to either 1 or 0. With the implementation of a sigmoid LUT and 8 bit addressing, the resolution for the entire range would be 6.4/256 = 0.025. An extra bit is used to store the sign information.

This addressing step size of the LUT table is not to be confused with the twos complement fraction size. This twos complement fraction size is dependent on the multiplicand and multiplicand size for multiplication. Figure 4.4 below describes the operation of neural network computation following the numbering convention suggested.

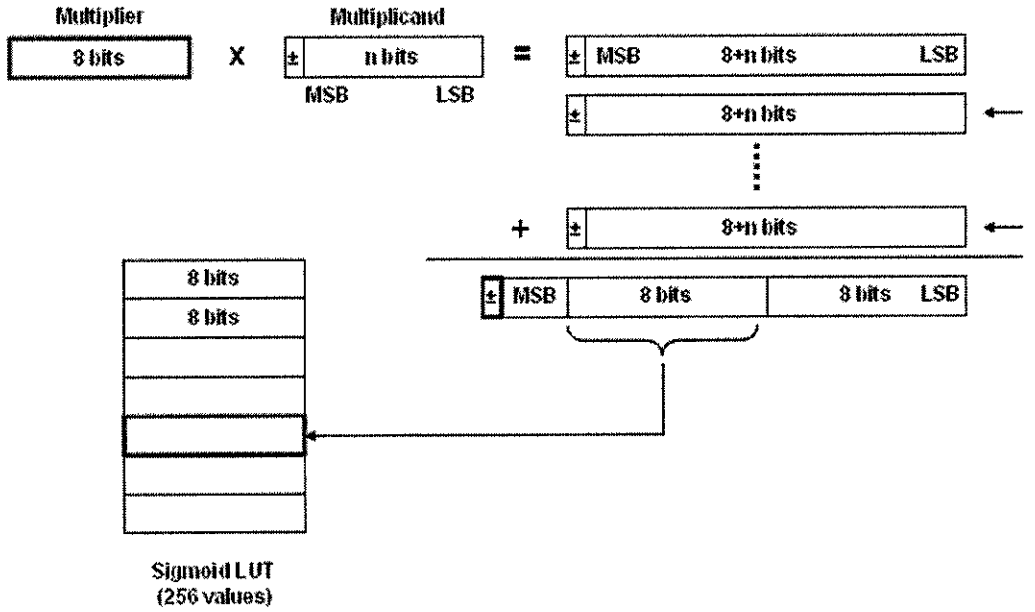


Figure 4.4: Description of Neural Network Computation

The multiplier represents the weights of the connection where as the multiplicand is the value of the presented input. The value of the multiplicand never exceeds 1_{10} and is represented by an 8 bit wide word whereas the multiplier is represented by an n-bit wide word depending on the value of the weight. Notice that the multiplicand has no sign bit as it is always positive. Multiplication of the multiplier and multiplicand would yield a (8 + n) bit + 1 sign bit output. This output is then summed up with other similar operations including the bias value in the neuron before being fed to the

sigmoid function. Bits 9 to 16 of the summation result are used to address the sigmoid LUT. The value associated with the address is the sigmoid function equivalent value of the pointer. To fully address 256 items of the sigmoid LUT, bits 9 to 16 of the summation result must fully vary between for decimal values 0 to 6.528 (value chosen instead of 6.4 for more straightforward correlation). For example, to match the value of 6.528 the summation result would be 1111111100000000_2 or 65280_{10}. From this, the twos complement fraction size can be derived.

Max value	=	65280_{10}
Range width	=	6.528_{10}
Fraction Size (N)	=	$(6.528/65280)^{0.5} = 0.01$

E.g.:

Using twos complement fractional representation

Multiplicand $=1_{10} = 1/0.01 = 100_{steps} = 01100100_2$

Multiplier $= 6.528_{10} = 6.528/0.01 = 653_{steps} = 1010001101_2$

By multiplying the above binary values, the result is as shown below.

$$\begin{array}{r} 1010001101_2 \\ \times \quad 1100100_2 \\ \hline 1111111100101000_2 \\ \hline \end{array}$$

With the bits 9 to16 all set to1, the highest value of the sigmoid LUT is accessed and a corresponding result of 1 is provided. For binary values exceeding 1111111100000000_2, the output would be automatically set to 1 whereas binary values below 0000000011111111_2, the output will be set to 0. The value of weights and neuron biases generated by the MetaNeural™ program and the value of sigmoid outputs would be converted using this representation.

4.3 RFNNA Simulator

This application was initially written by the author to help simulate the neural network for the XOR problem using the RFNNA architecture. The simulation of neural networks for up to 3 hidden layers with a maximum of 3 neurons per layer is possible using this application. The weights used for the program utilizes the WGT weight file as generated by the MetaNeural™ application. The simulator simulates neural network operation by showing all internal mathematical computation and result for a chosen network. Additional features such as accuracy adjustor for adjusting the number of decimal places for mathematical operation can be used to analyze how well a network performs and the margin of error caused by rounding and truncating the numbers.

Figure 4.5 below shows the software simulating the first hidden layer of a 2-3-2-1 architecture for the XOR problem. The operation manual for the application is included in **APPENDIX C**. More screenshots is available in **APPENDIX D**.

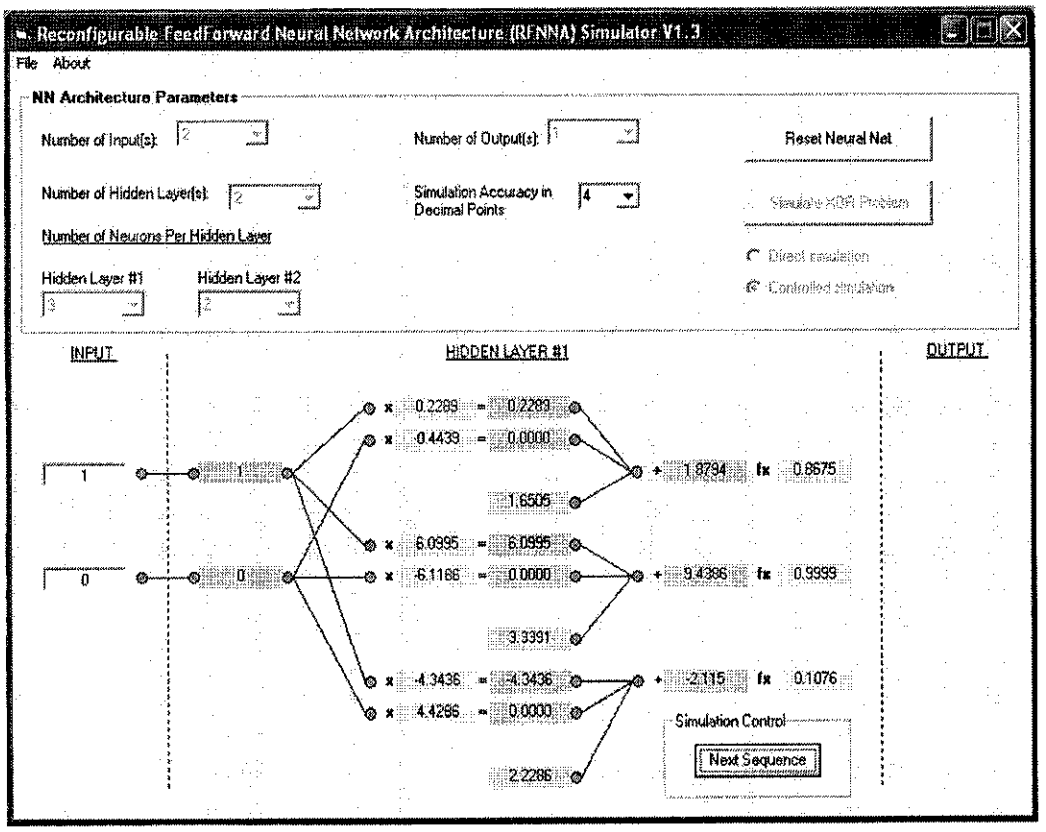


Figure 4.5: Screenshot of RFNNA Simulator v1.3

4.4 Neural Network Processor Modules

The RFNNA processor was designed using the modular method as was mentioned in Chapter 3 Methodology. Figure 4.6 shows a basic block diagram for the hardware implementation of the RFNNA processor. A more detailed and accurate block diagram is available in **APPENDIX E**. All modules are individually designed based on the specifications mentioned earlier and validated using testbenches. However some adjustments were made to better suit the architecture for hardware implementation. The adjustments made to a module will be discussed the module's subtopic respectively

The final validation was performed after the control unit of the processor was completed. The RTL simulation of the RFNNA processor for a 2-3-3-3-1 was successful. The discussion in this section will be divided into the respective modules available in the hardware implementation as shown in **APPENDIX E**.

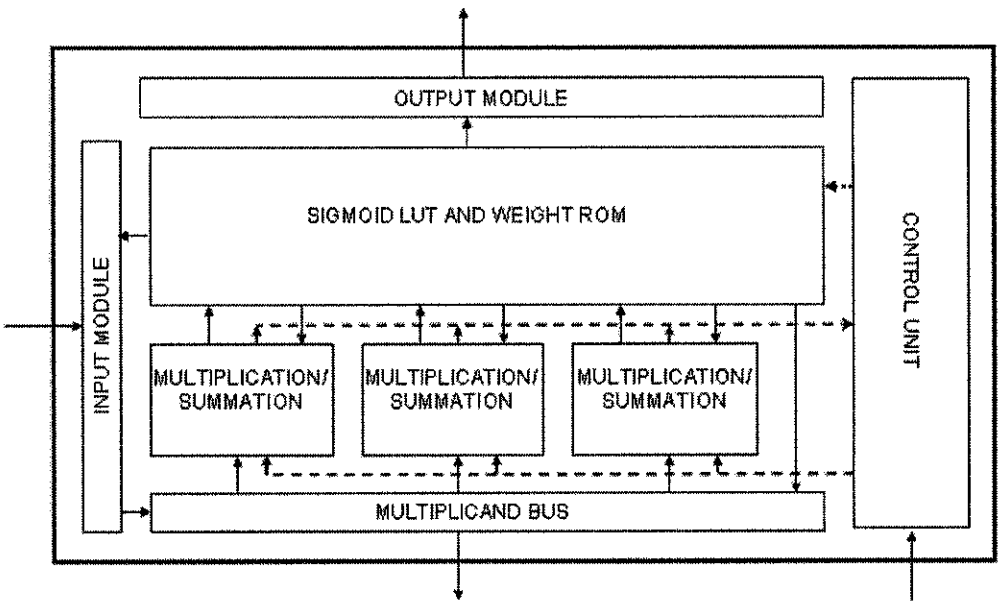


Figure 4.6: Layout of 3 Neuron RFNNA Processor HDL Modules

There are total 11 unique modules within the neural network processor designed. The modules are as follows (labels in brackets refer to the actual module name in the design schematic):

1) Input Module (mux1)
2) Input Module Counter (mux1_cnt)
3) Bias and Weight ROM Module (values)
4) Bias and Weight Counter (values_cnt)
5) Neuron Module (neuron)
6) Neuron Output Multiplexer Module (mux2)
7) Neuron Output Multiplexer Counter (mux2_cnt)
8) Number Representation Converter Module (interface)
9) Activation Function LUT Module (activ_function)
10) Output Threshold Module (threshold)
11) Control Unit (Fub2)

The simulated 2-3-3-3-1 RFNNA processor is able to process a given set of XOR inputs in 265 to 270 clock cycles depending on the combination of inputs provided. This means that if the processor is running at a conservative speed of 10 MHz, the number of XOR computations possible per second would be around 37000 to 37700.

4.4.1 Input Module (mux1)

The Input Module serves as buffer and multiplexer to external inputs as well as outputs from the activation function. External inputs are first converted into its equivalent value for arithmetic computation. For example, if the input is logic 1 the equivalent value stored in the buffer (mult_reg) for multiplier values would be 01100100_2. An all 0 word is provided if the input is logic 0.

The input module accepts and assigns values to external inputs to be stored into the multiplier buffers concurrently. However the same buffers are written to sequentially when data is passed from the activation function. No conversion is required for data from the activation function. The control signals provided to the inputs sel_reg[1:0] addresses the appropriate buffer in which the incoming data is supposed to be written to. Besides receiving and storing data, the Input Module is also required to correctly broadcast the values in its buffers onto the multiplier bus (refer to section **4.4.12**

Multiplier Bus). Again, the sel_reg[1:0] is used to properly select which buffer is being broadcast. To switch between broadcast and write mode, control signal from the control unit to the input sel_inout is used. However the sel_inout control is nulled when the sel_reg is in 2'b00 mode, which is when the external inputs are being read, converted and stored into buffers. Figure 4.7 shows the waveform of module being simulated.

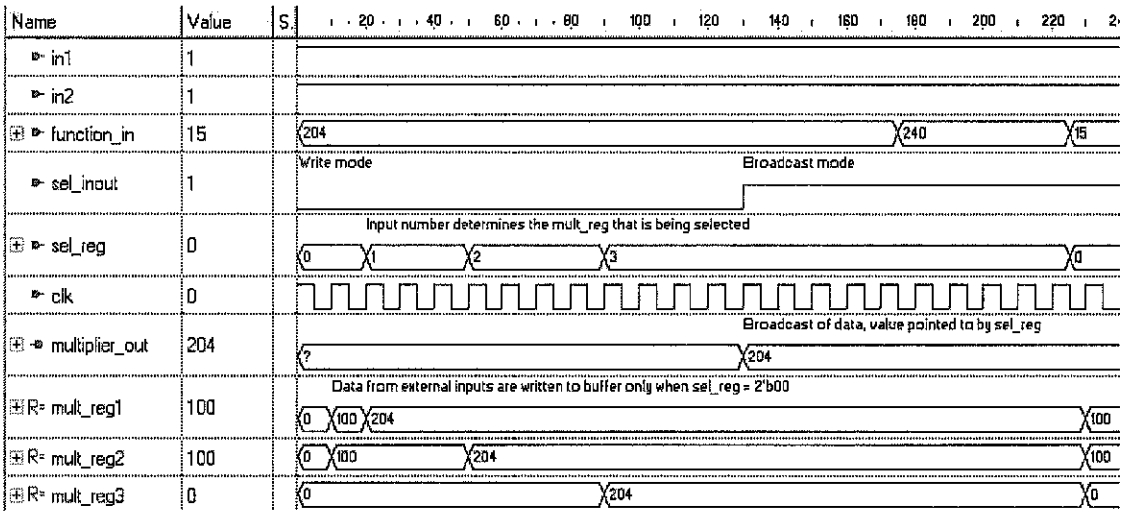


Figure 4.7: Simulation Result for Input Module

The Verilog code for the Input Module is available in **APPENDIX G**. The testbench for the simulation in Figure 4.7 is available in **APPENDIX H**.

4.4.2 Input Module Counter (mux1_cnt)

The Input Module Counter provides the control signals to the Input module's sel_reg[1:0] to select between the multiplier buffers. The counter counts from 0 to 3, and loops back to 1, whenever there is a reset, the counter goes back to 0.

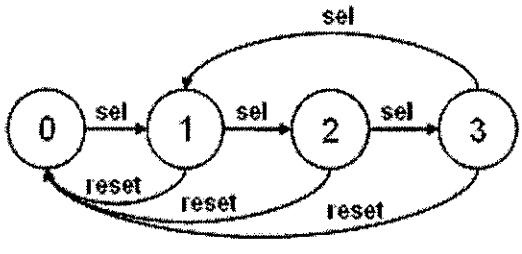


Figure 4.8: Mealy Machine for Input Module Counter

The sel input signal comes from the Control Unit of the neural network processor. The reset signal is provided by the external reset. The Verilog code for the Input Module Counter is available in **APPENDIX I.**

4.4.3 Bias and Weight ROM Module (values)

There are three Bias and Weight ROM modules in the designed processor. Each module is similar to the other, differing only in the weight and bias values they carry. Each module is dedicated to one neuron and stores bias and weight values for three hidden layers and one output layer.

As in the Input Module, weight values are sequentially passed to the Neuron Module. Each module has an internal counter which tells the module which weight value is to be passed on. External signals to the modules layer[1:0] input tells which layer the bias and weights value it can select from. The bias values stored in the modules are in twos complement format while weight values are stored in sign magnitude integer. The fraction size for the weight and bias values are different. The fraction size for weight values is 0.01 whereas the fraction size for bias values is 0.0001.

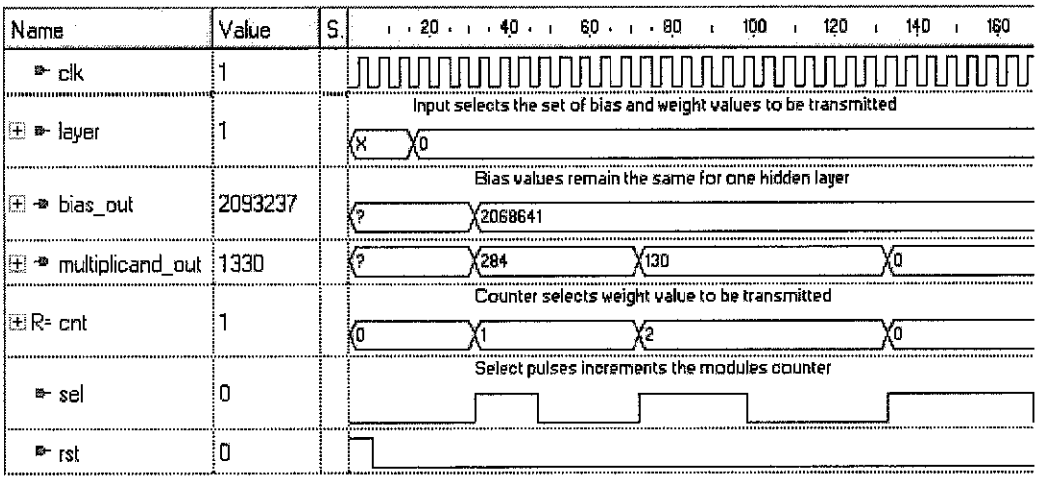


Figure 4.9: Simulation Result for Bias and Weight ROM Module

The Verilog code for the Bias and Weight ROM Module is available in **APPENDIX J**. The testbench for the simulation in Figure 4.9 is available in **APPENDIX K.**

4.4.4 Bias and Weight Counter (values_cnt)

The Bias and Weight Counter provides input to the layer[1:0] input of the Bias and Weight Module. This counter keeps track and of which hidden layer or output layer the processor is in. Figure 4.10 shows the state machine for the counter.

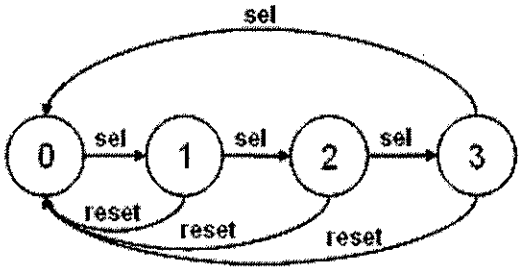


Figure 4.10: Mealy Machine for Bias and Weight Counter

The counter is incremented by the sel signal from the Control Unit of the neural network processor. The reset signal is provided by the external reset. The Verilog code for the Input Module Counter is available in **APPENDIX L**.

4.4.5 Neuron Module (neuron)

Instead of just using the "*" arithmetic operator to perform multiplication, there is a need for more control from the process so that control bits can be incorporated into the multiplication process.

```
add_out <= multiplier_in* multiplicand_in;
```

The control bits are important so that feedback such as when the multiplier, multiplicand and bias values has been recorded or when the multiplication and summation has completed can be provided back to the control unit. The control unit would then determine the appropriate action required for the next operation in the neuron module for the hidden layer iteration.

Multiplication for the neuron module uses the Add Shift Right (ASR) algorithm. This algorithm is suited for unsigned binary multiplication which is the type of data presented to it. Figure 4.11 below shows the flowchart for the ASR algorithm.

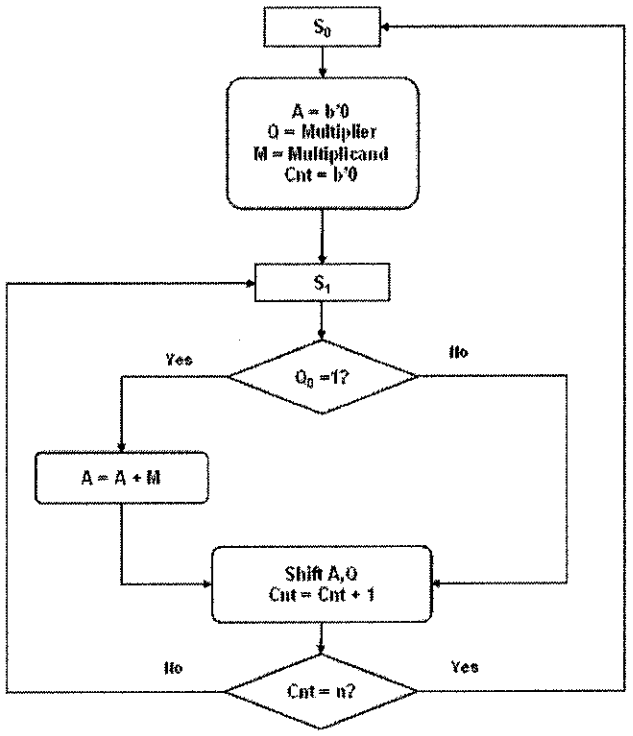


Figure 4.11: Flowchart for Unsigned Binary Multiplication

While the multiplication algorithm involves unsigned binary integer, the register mult_sign stores the sign bit of the multiplicand. The sign bit will be a flag as to whether the multiplication result need to be complemented before it is summed up with the value stored in add_out.

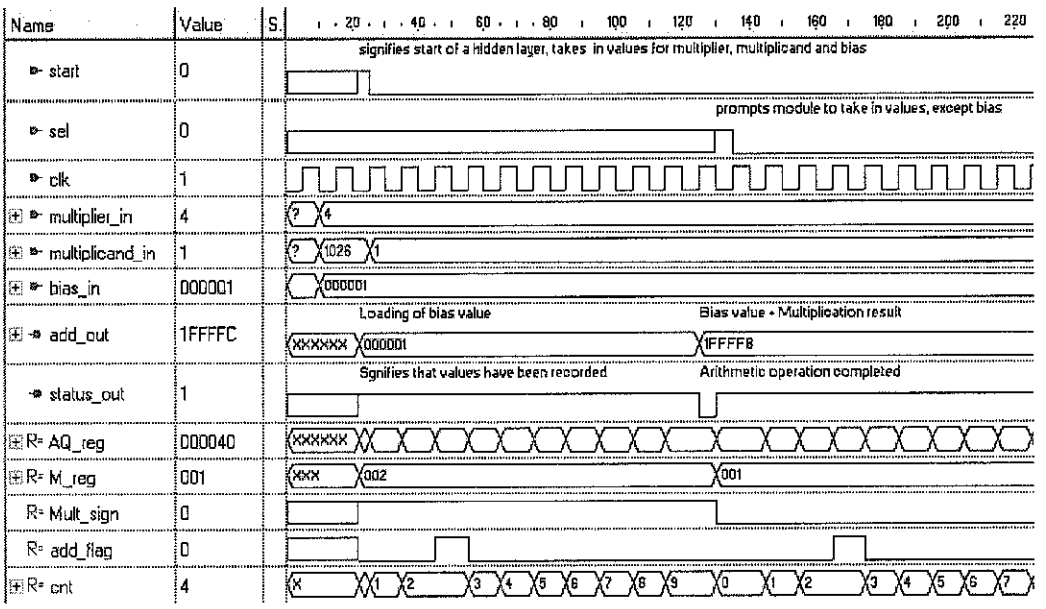


Figure 4.12: Simulation Result for Neuron Module

Figure 4.12 shows the waveform showing all internal operations of the Neuron module. There is only one multiplier per neuron, therefore multiplier and multiplicand values provided by the Input Module and the Bias and Weight ROM Module respectively are multiplied and summed up sequentially with the bias value which is directly stored in the add_out register. The loading of values is different for the starting of a hidden layer and the loading of values there after. Bias values are only loaded once for every hidden layer into the Neuron Module whereas multiplier and multiplicand values are loaded at every iteration. To differentiate between starting of a hidden layer iteration and a normal iteration, two different stimulus signals are provided, start and sel. The status_out register provides output that signifies that values has been loaded into the module and when arithmetic operations for a particular iteration have completed.

The Verilog code for the Neuron Module is available in **APPENDIX M**. The testbench for the simulation in Figure 4.12 is available in **APPENDIX N**.

4.4.6 Neuron Output Multiplexer Module (mux2)

The purpose of the Neuron Output Multiplexer Module is to multiplex between the outputs of the neurons such that only one output is passed through the activation function. There is only one activation function in the processor and all neuron has to share its use. This is because the implementation of an 8 bit wide LUT requires a lot logic gates, thus it is not feasible to have dedicated activation function LUT's for each neuron.

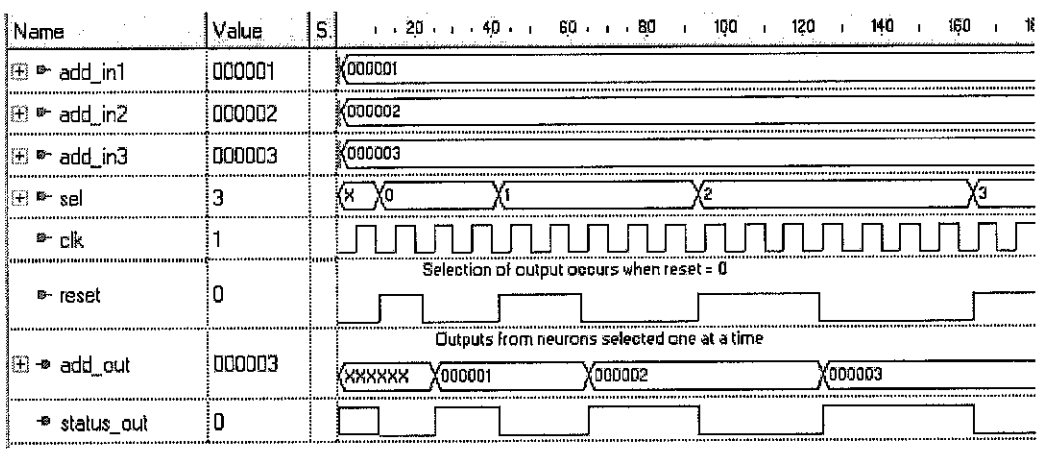


Figure 4.13: Simulation Result for Neuron Output Multiplexer Module

The module does not buffer the output from the Neuron Modules. It simultaneously reads in the values and passes only one which is chosen by the sel[1:0] input to be output.

The Verilog code for the Neuron Output Multiplexer Module is available in **APPENDIX O**. The testbench for the simulation in Figure 4.12 is available in **APPENDIX P**.

4.4.7 Neuron Output Multiplexer Counter (mux2_cnt)

The Neuron Output Multiplexer Counter's design and function is the same as that for the Bias and Weight Counter. The output of the counter however is now used to provide selection for the Neuron Output Multiplexer Module's sel[1:0].

The counter is incremented by the sel signal from the Control Unit of the neural network processor. The reset signal is provided by the external reset. The Verilog code for the Input Module Counter is available in **APPENDIX Q**.

4.4.8 Number Representation Converter Module (interface)

The purpose of the interface is to convert the twos complement result from the neuron module back to unsigned binary integer. This is because the twos complement result for negative results would not be able to be used with the activation function unless modifications were made to it. The other reason why the interface is required is because it relieves each neuron modules from having extra logic to perform the conversion. A centralized interface would reduce the number of logic used, because the activation function is only accessed one at a time by each neuron. The output of the interface would be an 8 bit wide word with a separate sign bit and also a status flag bit.

The Verilog code for the Neuron Output Multiplexer Module is available in **APPENDIX R**. The testbench for the simulation in Figure 4.14 is available in **APPENDIX S**.

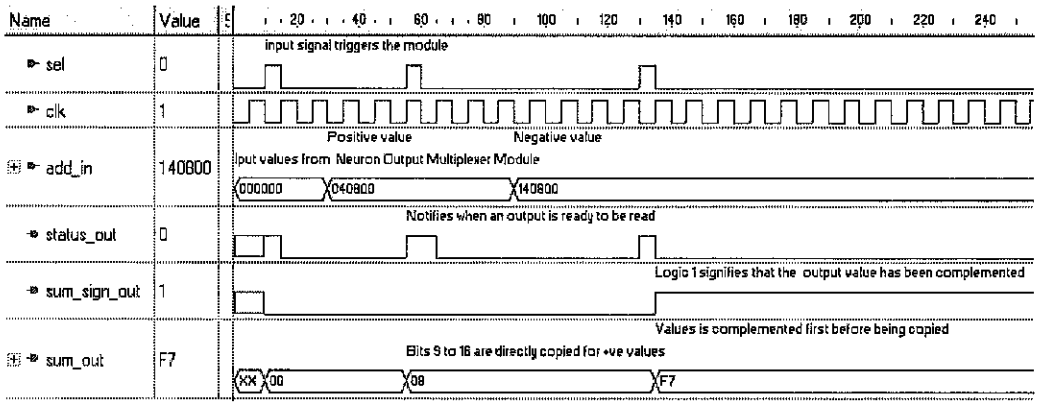


Figure 4.14: Simulation Result for Neuron Representation Converter Module

4.4.9 Activation Function LUT Module (activ_function)

The LUT which is to be implemented and declared as a ROM block in the FPGA device itself contains many redundant entries. Addressing the LUT is an 8 bit input which has 256 entries. Using mathematical analysis, it is possible to reduce the ROM usage from 256 x 8 bit word entries to 68 x 8 bit word entries. This is because there are only 50 unique data that is being accessed in the LUT ranging from decimal equivalent of 50 to 99.

All input combinations are accounted for with the help of mathematical analysis as shown below. Several of the combinations can be grouped together for an entry thus reducing the need for individual access for equivalent results.

Input	LUT values
0	110010
1	110010
10	110011
11	110011
100	110100
101	110101
110	110101
111	110110
1000	110111
1001	110111
1010	111000
1011	111000
1100	111001
1101	111010
1110	111010
1111	111011

Figure 4.15: Mathematical Analysis on Sigmoid LUT Values

Inputs in each colored band in Figure 4.15 are grouped together to address a similar LUT equivalent.

The inputs from the Number Representation Converter Module are used to address the activation function LUT as well as to note the sign of the input argument. As was mentioned before, the LUT tale can only address values from 50 to 99 corresponding to the sigmoid range of 0.50 to 0.99. These values are only valid for positive arguments. If the sign of the argument is negative, some manipulation of the LUT result has to be performed. The LUT equivalent would be subtracted from 100 to produce the correct answer.

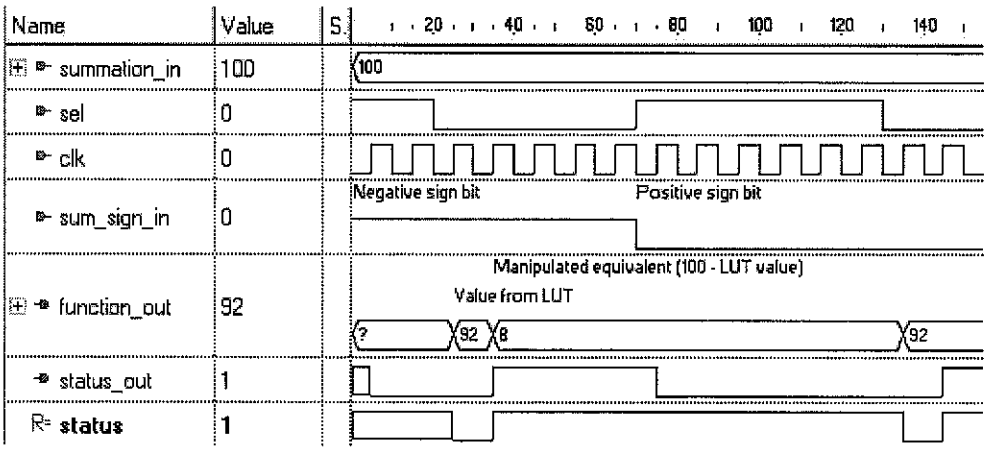


Figure 4.16: Simulation Result for Activation Function LUT Module

The Verilog code for the Activation Function LUT Module is available in **APPENDIX T**. The testbench for the simulation in Figure 4.16 is available in **APPENDIX U**.

4.4.10 Output Threshold Module (threshold)

The output Threshold Module is used to provide a categorization of the output to whether it is a logic 1, logic 0 or in the indeterminate state. Output from the Activation Function ROM Module is continuously processed to provide classification. For logic 1 the output from the activation function must be in the range of 0.9 to 1.0. For logic 0, the output must be in the range of 0.0 to 0.1. Any other values would produce a high impedance output.

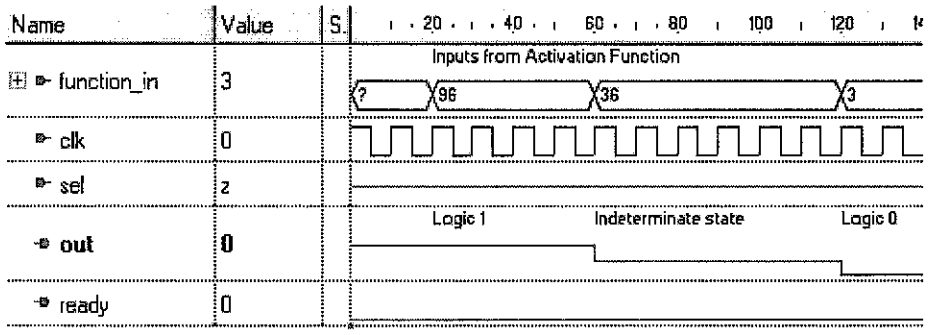


Figure 4.17: Simulation Result for Output Threshold Module

The Verilog code for the Output Threshold Module is available in **APPENDIX V**. The testbench for the simulation in Figure 4.16 is available in **APPENDIX W**.

4.4.11 Control Unit (Fub2)

The Control Unit for this neural network processor has 10 distinct inputs and 10 distinct outputs. The Control unit is able to guide the rest of the modules to function as intended as the right time. The sequence of operations is as stated in **APPENDIX F**. The control unit was designed directly in the finite state machine editor and converted to Verilog.

The Control Unit provides control signals to all modules except for the Number Representation Converter Module and the Activation Function LUT Module. The system is designed such that in case of a reset, the whole processor can be set back to its initial state.

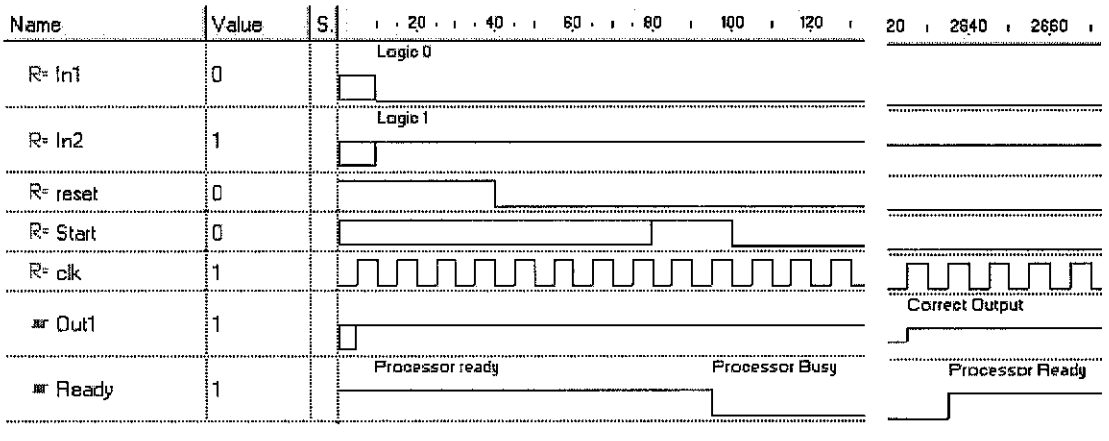


Figure 4.18: Simulation Result for RFNNA Processor

Figure 4.17 shows the simulation for the whole neural network processor as one working entity of the modules discussed earlier. The Ready flag, notifies the user when the processor is available or when processing is complete.

The Verilog code for the Control Unit is available in **APPENDIX X**. The testbench for the simulation in Figure 4.18 is available in **APPENDIX Y**.

4.5 Multiplier Bus

An additional improvement to the original RFNNA architecture would be the inclusion of the multiplier bus for transmission of inputs for multiplication with connection weights. The bus works by time multiplexing its resources for the transmission of multiplier to each neuron available in the architecture.

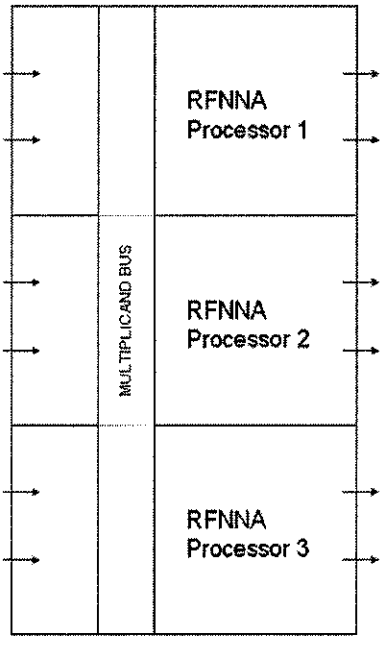


Figure 4.19: RFNNA Processors Paralleled

This implementation of the multiplier bus would reduce the number of multipliers in a neuron unit exponentially. Comparing with the previous architecture, the number of multipliers required for N number neuron architecture would be N^2 in total, whereas the number of multipliers required now is only N. This would save a significant amount of logic resources.

The implementation of the multiplicand bus would also open up the possibility of similar RFNNA processors to be paralleled as in Figure 4.19, thus increasing the number of inputs, outputs and neurons. This ability would allow more application to neural network processed as the number of inputs and outputs are no longer a limitation

5 CONCLUSION AND FUTURE IMPROVEMENT

5.1 Conclusion

The project can be rated as successful with the completion of the RTL design for the neural network processor for the XOR problem. The objectives of optimizing the implementation have been performed on two fronts: architecture and organization.

Through research and analysis the RFNNA architecture and the optimized numbering convention have been specified. The RFNNA architecture is a space efficient implementation for neural network onto hardware, where all hidden layers use the same logic resources without sacrificing implementation speed. The twos complement fixed point fraction on the other hand increases processing speed by simplifying the computation of binary values and addressing of the sigmoid activation function's LUT. In the process of analyzing the proposed architecture, an application was also developed to help simulate and justify neural networks based on the RFNNA architecture. Organization improvements through the implementation of the multiplicand bus enable the processor to process in parallel. The decision to use a single activation function ROM module helps reduce implementation size so does the recursive use of a single multiplier within each neuron.

All of the mentioned findings and implementation have contributed towards achieving the objectives of the project in which an optimized FPGA hardware implementation of neural networks in terms of size, speed and performance is desired.

5.2 Future Improvement

The project provides a strong foundation in which more sophisticated neural network processors can be built upon. The FeedForward architecture utilising the BackPropagation learning algorithm caters for a variety of applications in which fast processing is a must. With this motivation in hand, it is viable for processors to be designed catering for these needs.

Utilising the RFNNA design and hardware organization design methodologies mentioned in this report will provide any beginner in neural network hardware design much useful analysis. However there are still areas which can be improved on in terms of design and implementation. The author would suggest that future designs would have a general purpose neural network processor design in mind.

This general purpose neural network processor ideally can be used for any applications and can be parallel processed with similar processors so that the number of inputs, outputs and neurons per layer will not be a constraint. The processor would only require the weight and bias values to be reprogrammed. These values can be stored on external memory so that hardware reprogramming is not required. The possibility of an ASIC implementation would be more plausible then. The processor would also have external inputs which controls the number of hidden layers it can process. For this, the designer must do away with a hardwired implementation of the control unit.

With a general purpose neural network processor, implementations in ASIC technology would provide faster processing and at lower prices, opening up the possibility of neural network processing to a multitude of applications.

REFERENCES

[1] Callan R. 1999, *Essence of Neural Networks,* Hertfosrdshire, Prentice Hall Europe.

[2] Bose N.K., Liang P. 1996, *Neural Network Fundamentals with Graphs, Algorithms and Applications*, McGraw-Hill, Singapore.

[3] Coffman, K. 1999, *Real World FPGA Design with Verilog*, Prentice Hall, New Jersey.

[4] Picton, Phil 1994, *Introduction to Neural Networks*, MacMillan Press LTD, London.

[5] Sundararajan N., Saratchandran P. 1998, *Parallel Architectures for Artificial Neural Networks*, IEEE Computer Society, Los Alamitos, California.

[6] Vaughn B., Jonathan R., Alexander M. 1999, *Architecture and CAD for Deep-Submicron FPGA's*, Kluwer Academic Publishers, Massachusetts.

[7] William Stallings 2003, *Computer Organization and Architecture*, Prentice Hall, New Jersey.

[8] J. Zhu, G.J. Milne and B.K. Gunther, *Towards An FPGA Based Reconfigurable Computing Environment for Neural Network Implementations*, in Proceedings of IEE Conference on Artificial Neural Networks, 1999: p. 661-666

[9] Eldredge, J.G. and B.L Hutchings, *Design Methodologies for Partially Reconfigured Systems*, in Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines, 1994: p. 78-84.

[10] Brown, S. and Vranesic, Z. 2003, *Fundamentals of Digital Logic with Verilog Design*, McGraw Hill.

[11] Ciletti, M. 2002, *Advanced Digital Design with the Verilog HDL*, Prentice Hall, New Jersey.

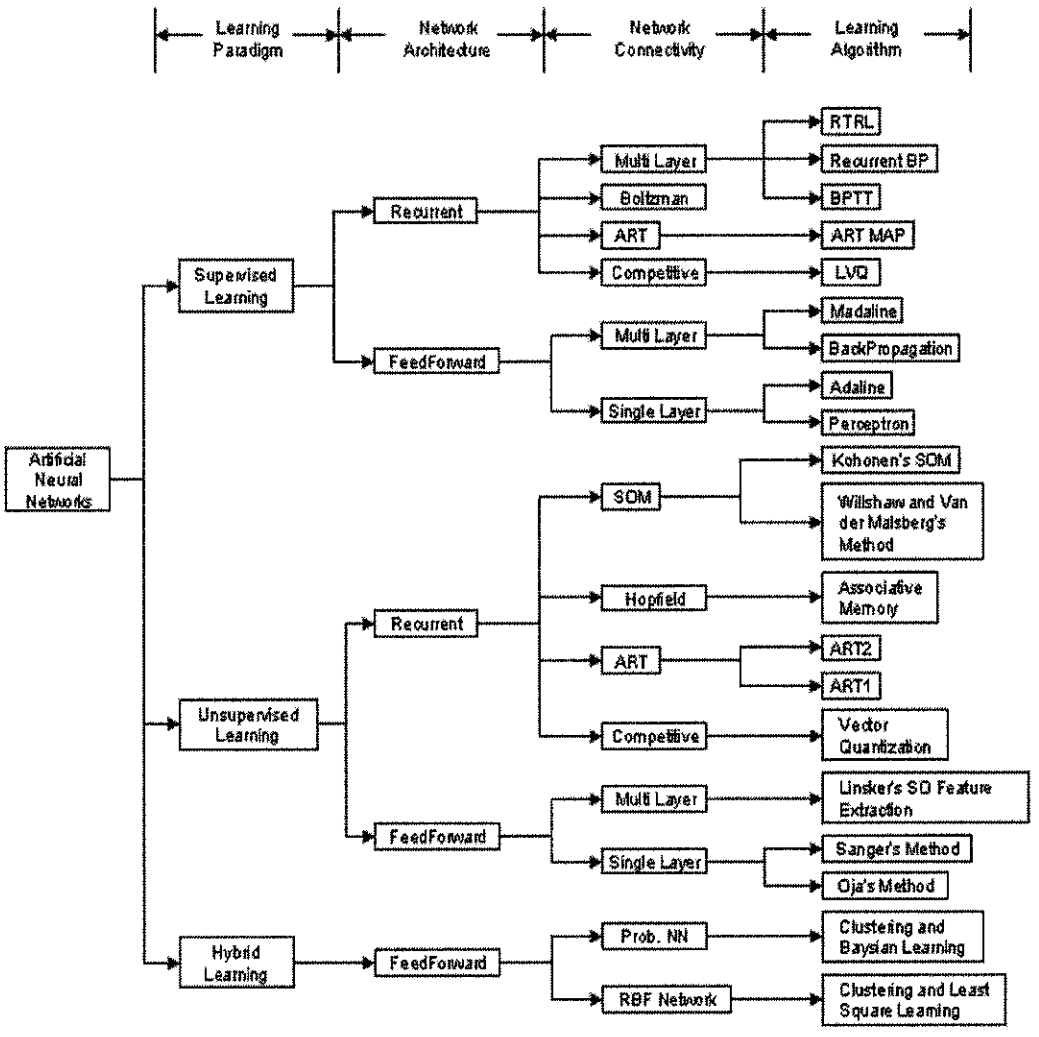
[12] Szabo T., Feher B. and Horvath G. 1998, *Neural Network Implementation Using Distributed Arithmetic*, in Proceedings of the 2nd International Conference on Knowledge Based Intelligent Electronic Systems

[13] Haykin S. 1999, Neural *Networks – A Comprehensive Foundation*, Prentice Hall New Jersey, p 23

[14] Gschwind, M. V. Salapura. and O. Maischberger, *Space Efficient Neural Network Implementation*, in Proceedings of the 2nd ACM Workshop on Field Programmable Gate Arrays, 1994. p. 23-28

APPENDIX A

Taxonomy of ANN Models



APPENDIX B

Similarities and Differences between Neural Net and Von Neumann Computer.

	Neural Net	Von Neumann Computer
1.	Trained (learning by example) by adjusting the connection strengths , thresholds and structure	Programmed with instructions (if-then analysis based on logic)
2.	Memory and processing elements are collocated	Memory and processing separate
3.	Parallel (discrete and continuous), and asynchronous	Sequential or serial, digital, synchronous (with a clock)
4.	May be fault tolerant because of distributed representation and large scale redundancy	Not fault tolerant
5.	Self organization during learning	Software dependant
6.	Knowledge stored is adaptable; information is stored in the interconnection between neurons	Knowledge stored in an addressed memory location is strictly replaceable

APPENDIX C

Operation of the RFNNA Simulator v1.3 Software Application

The RFNNA Simulator works in 2 simulation modes:

a) Direct Simulation
- In this mode, the user would be able to obtain instant results for the XOR problem for any given input when the simulation button is pressed.

b) Controlled Simulation
- An extra command button "Next Sequence" will appear. In this mode the simulation will pause after computation is performed on every hidden layer. To proceed to the next hidden layer, press on the "Next Sequence" command button.

NOTICE:

Pattern files can be manually generated, examples of neural network architecture and corresponding weight files are listed below:

Architecture	WGT File
2-3-1	W2131.wgt
2-3-3-1	W22331.wgt

The wgt files are directly generated from the metaNeural software. However to be able to use it using this software, the user has to use an appropriate naming convention.

Example:

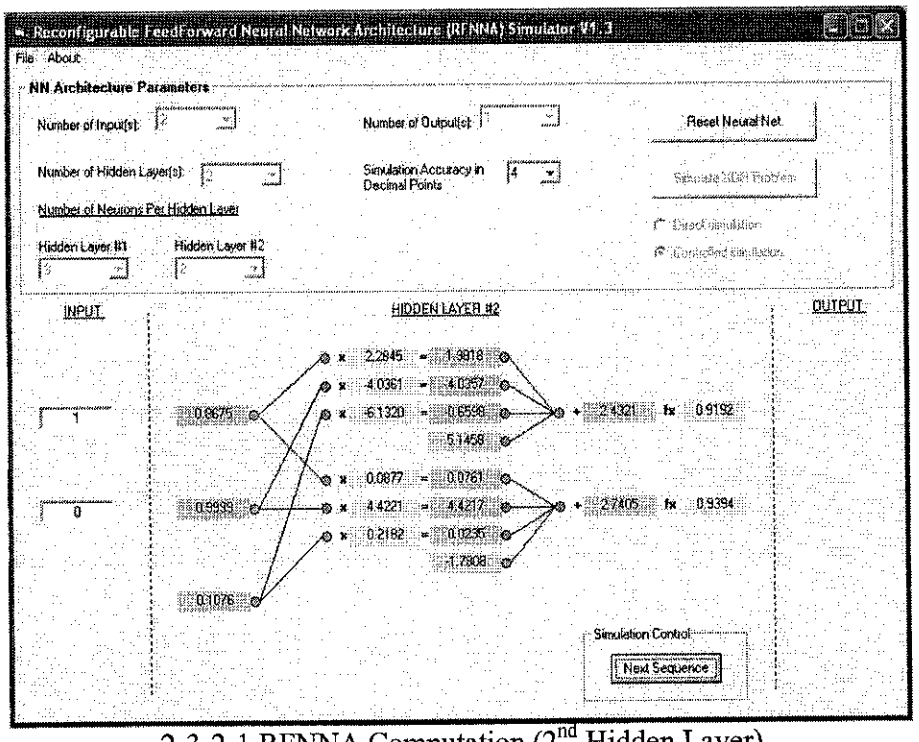
W2131.wgt

W = All files must start with "W"
2 = Indicates the number of inputs
1 = Indicates the number of hidden layer
3 = Indicates the number of neurons for 1st hidden layer and so on
1 = Indicates the number of outputs

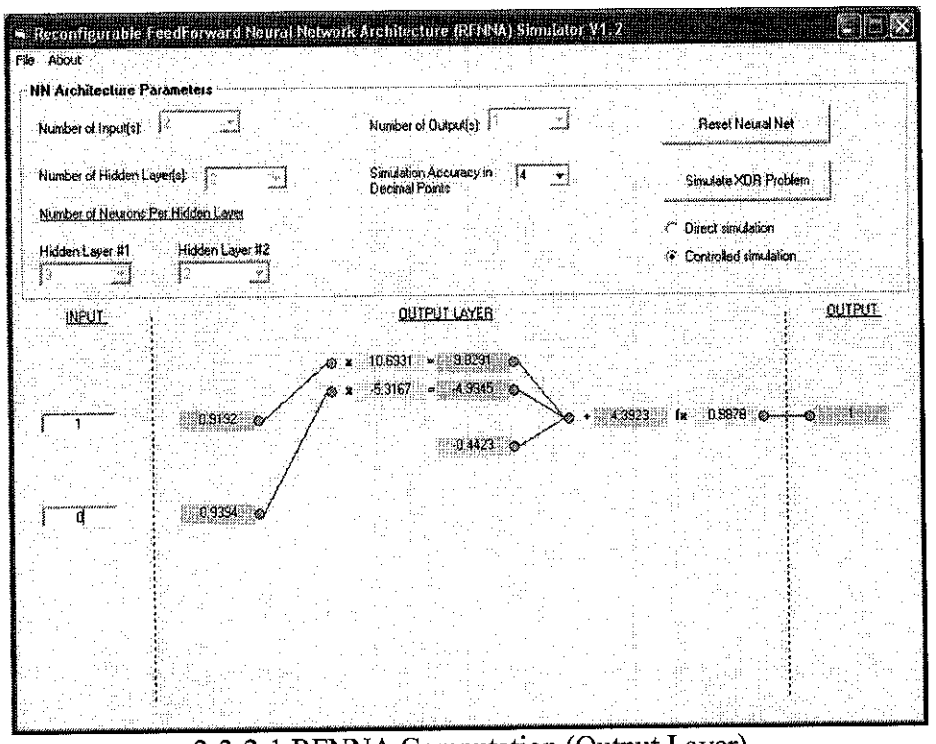
The WGT files will be automatically retrieved from c:\METACTRL\. Please ensure that the wgt files provided are located into this directory. All WGT files generated using MetaNeural will be saved in the same directory.

APPENDIX D

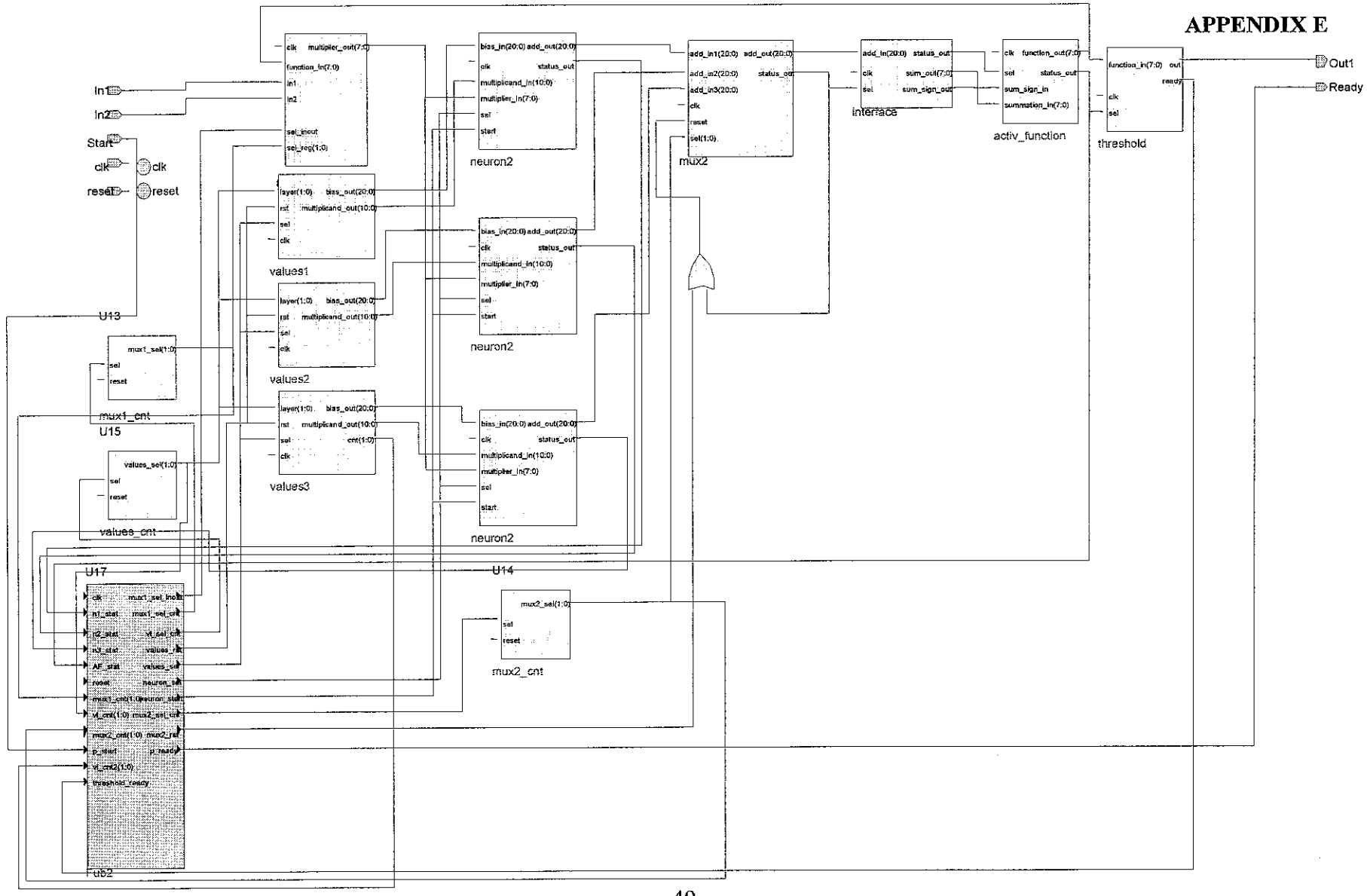
Screenshots of RFNNA Simulator Performing XOR operation for 2-3-2-1 Neural Network Architecture.



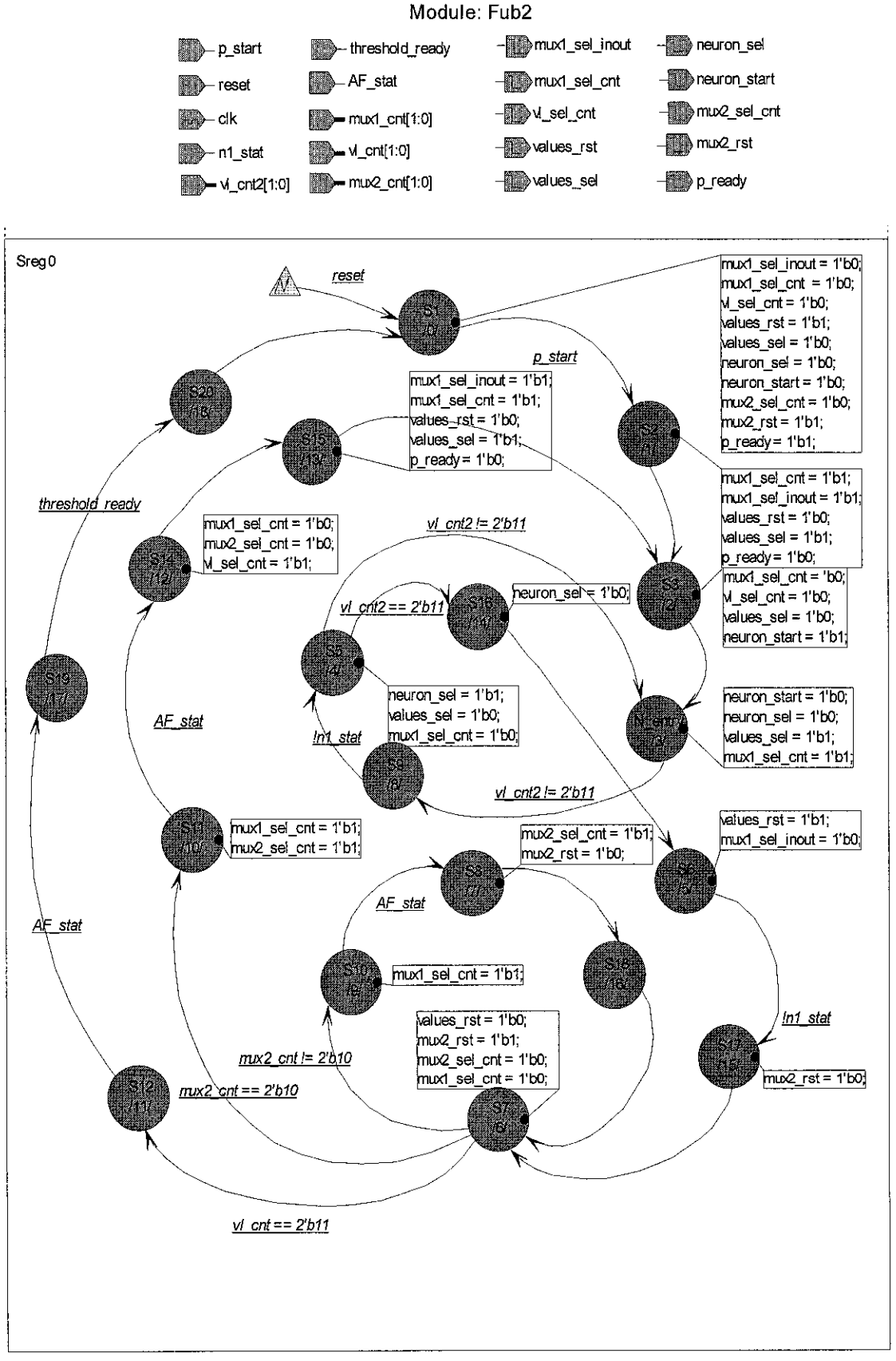
2-3-2-1 RFNNA Computation (2nd Hidden Layer)



2-3-2-1 RFNNA Computation (Output Layer)


In1
In2
Start
clk
reset
clk
reset
U13
mux1_sel(1:0)
sel
reset
mux1_cnt
U15
values_sel(1:0)
sel
reset
values_cnt
U17
values1
values2
values3
neuron2
neuron2
neuron2
U14
mux2_sel(1:0)
sel
reset
mux2_cnt
mux2
interface
activ_function
threshold
Out1
Ready

APPENDIX F


Module: Fub2
p_start
reset
clk
n1_stat
vl_cnt2[1:0]
threshold_ready
AF_stat
mux1_cnt[1:0]
vl_cnt[1:0]
mux2_cnt[1:0]
mux1_sel_inout
mux1_sel_cnt
vl_sel_cnt
values_rst
values_sel
neuron_sel
neuron_start
mux2_sel_cnt
mux2_rst
p_ready
Sreg 0
reset
mux1_sel_inout = 1'b0;
mux1_sel_cnt = 1'b0;
vl_sel_cnt = 1'b0;
values_rst = 1'b1;
values_sel = 1'b0;
neuron_sel = 1'b0;
neuron_start = 1'b0;
mux2_sel_cnt = 1'b0;
mux2_rst = 1'b1;
p_ready = 1'b1;
p_start
mux1_sel_inout = 1'b1;
mux1_sel_cnt = 1'b1;
values_rst = 1'b0;
values_sel = 1'b1;
p_ready = 1'b0;
threshold_ready
mux1_sel_cnt = 1'b0;
mux2_sel_cnt = 1'b0;
vl_sel_cnt = 1'b1;
vl_cnt2 != 2'b11
vl_cnt2 == 2'b11
neuron_sel = 1'b0;
neuron_sel = 1'b1;
values_sel = 1'b0;
mux1_sel_cnt = 1'b0;
AF_stat
In1_stat
vl_cnt2 != 2'b11
mux1_sel_cnt = 1'b1;
mux2_sel_cnt = 1'b1;
mux2_sel_cnt = 1'b1;
mux2_rst = 1'b0;
values_rst = 1'b1;
mux1_sel_inout = 1'b0;
AF_stat
mux1_sel_cnt = 1'b1;
values_rst = 1'b0;
mux2_rst = 1'b1;
mux2_sel_cnt = 1'b0;
mux1_sel_cnt = 1'b0;
In1_stat
mux2_rst = 1'b0;
mux2_cnt != 2'b10
mux2_cnt == 2'b10
vl_cnt == 2'b11

APPENDIX G

Input Module (mux1)

```
//-------------------------------------------------------------------------------
//
// Title       : mux_1.v
// Author      : Ivan Teh Fu Sun
// University  : University Teknologi Petronas
//
//-------------------------------------------------------------------------------
//
// Description : Interfaces btw inputs/LUT and neurons
//
//-------------------------------------------------------------------------------

module mux1 (in1, in2, function_in, sel_inout, sel_reg, clk, multiplier_out);

input   [7:0]   function_in;
input           in1,in2;
input           sel_inout;
input   [1:0]   sel_reg;
input           clk;
output  [7:0]   multiplier_out;
reg             [7:0]   multiplier_out;
reg             [7:0]   mult_reg1;
reg             [7:0]   mult_reg2;
reg             [7:0]   mult_reg3;

always @ (posedge clk)
begin

        if (sel_reg == 2'b00)
                begin
                if(in1)
                        begin
                                mult_reg1       <=      8'b01100100;
                        end
                else
                        begin
                        mult_reg1       <=      8'b00000000;
                        end

                if(in2)
                        begin
                                mult_reg2       <=      8'b01100100;
                        end
                else
                        begin
                                mult_reg2       <=      8'b00000000;
                        end

                mult_reg3       <= 8'b00000000;

                end

// outputs contents of buffers sequentially
        if (sel_inout)
                begin
                case(sel_reg)
                2'b01:  multiplier_out <=       mult_reg1;
                2'b10:  multiplier_out <=       mult_reg2;
                2'b11:  multiplier_out <=       mult_reg3;
                endcase
                end

// stores output from activation function into respective buffers sequentially
        else
                begin
                if (sel_reg == 2'b01)
                        begin
                        mult_reg1       <=       function_in;
                        end
                if (sel_reg == 2'b10)
```

```
                begin
                mult_reg2       <=      function_in;
                end
        if (sel_reg == 2'b11)
                begin
                mult_reg3       <=      function_in;
                end
        end
end

endmodule
```

APPENDIX H

Testbench for Input Module

```
`timescale 1ps / 1ps

module TB_mux1;

reg     [7:0]   function_in;
reg             in1,in2;
reg             sel_inout;
reg     [1:0]   sel_reg;
reg             clk;
wire    [7:0]   multiplier_out;

mux1 UUT (in1, in2, function_in, sel_inout, sel_reg, clk, multiplier_out);

initial
        begin

        #0;
        clk =1'b1;
        sel_inout = 1'b0;
        sel_reg = 2'b00;
        function_in= 8'b11001100;
        in1 <= 1'b1;
        in2 <= 1'b1;

        #20000;
        sel_reg=2'b01;

        #30000;
        sel_reg=2'b10;

        #40000;
        sel_reg=2'b11;

        #40000;
        sel_inout = 1'b1;

        #45000;
        function_in = 8'b11110000;

        #50000;
        function_in = 8'b00001111;
        sel_reg = 2'b00;

        end

always

        begin

                #5000
                clk = !clk;

        end
endmodule
```

APPENDIX I

Input Module Counter (mux1_cnt)

```
//-------------------------------------------------------------------------------
//
// Title       : mux1_cnt.v
// Author      : Ivan Teh Fu Sun
// University  : University Teknologi Petronas
//
//-------------------------------------------------------------------------------
//
// Description : provides selection for mux_1 module
//
//-------------------------------------------------------------------------------

module mux1_cnt (sel, reset, mux1_sel);

input                   sel;
input                   reset;
output  [1:0]   mux1_sel;
reg             [1:0]   mux1_sel;

always @ (posedge reset or posedge sel)
begin

        if(reset)
                mux1_sel <=2'b00;
        else
                begin
                        case(mux1_sel)
                        2'b00:  mux1_sel <= 2'b01;
                        2'b01:  mux1_sel <= 2'b10;
                        2'b10:  mux1_sel <= 2'b11;
                        2'b11:  mux1_sel <= 2'b01;
                        endcase
                end
end
endmodule
```

APPENDIX J

Bias and Weight ROM Module (values)

```
//-----------------------------------------------------------------------------
//
// Title       : neuron1_values.v
// Author      : Ivan Teh Fu Sun
// University  : University Teknologi Petronas
//
//-----------------------------------------------------------------------------
//
// Description : ROM block for bias and multiplicand values
//               Bias values are initial twos complement
//               Multiplicand values are initial unsigned binary
//               integer. MSB of multiplicand is the sign bit
//
//-----------------------------------------------------------------------------

module values1 (sel, rst, clk, layer, bias_out, multiplicand_out);

input                 sel;
input                 rst;
input                 clk;
input   [1:0]   layer;
output  [20:0]  bias_out;
output  [10:0]  multiplicand_out;
reg             [20:0]  bias_out;
reg             [10:0]  multiplicand_out;
reg             [1:0]   cnt;

always @ (posedge rst or posedge clk)
begin

        if(rst)
                begin
                cnt     <=      2'b00;
                end
end

always @ (posedge sel)
begin
if(layer == 2'b00)
        begin
        if(cnt == 2'b00)
                begin
                multiplicand_out <= 11'b00100011100;                //2.84
                bias_out         <= 21'b111111001000010100001;      //-2.84
                cnt              <= cnt + 1;
                end

        if(cnt == 2'b01)
                begin
                multiplicand_out <= 11'b00010000010;                //1.30
                cnt              <= cnt + 1;
                end

        if(cnt == 2'b10)
                begin
                multiplicand_out <= 11'b00000000000;                 //0.0
                cnt              <= 2'b00;
                end

        end

if(layer == 2'b01)
        begin
        if(cnt == 2'b00)
                begin
                multiplicand_out <= 11'b10100110010;                //-3.06
                bias_out         <= 21'b111111111000010110101;      //-0.39
```

```
                cnt                 <= cnt + 1;
                end

        if(cnt == 2'b01)
                begin
                multiplicand_out <= 11'b10011110000;                    //-2.40
                cnt                 <= cnt + 1;
                end

        if(cnt == 2'b10)
                begin
                multiplicand_out <= 11'b00110010011;                    //4.03
                cnt                 <= 2'b00;
                end
        end

if(layer == 2'b10)
        begin
        if(cnt == 2'b00)
                begin
                multiplicand_out <= 11'b00101111111;                    //3.83
                bias_out         <= 21'b000000000011101110011;          //0.19
                cnt                 <= cnt + 1;
                end

        if(cnt == 2'b01)
                begin
                multiplicand_out <= 11'b10101111110;                    //-3.82
                cnt                 <= cnt + 1;
                end

        if(cnt == 2'b10)
                begin
                multiplicand_out <= 11'b10001001000;                    //-0.72
                cnt                 <= 2'b00;
                end
        end

if(layer == 2'b11)
        begin
        if(cnt == 2'b00)
                begin
                multiplicand_out <= 11'b01001001101;                    //5.89
                bias_out         <= 21'b111111111100001101010;          //-0.18
                cnt                 <= cnt + 1;
                end

        if(cnt == 2'b01)
                begin
                multiplicand_out <= 11'b000011110111;                   //1.19
                cnt                 <= cnt + 1;
                end

        if(cnt == 2'b10)
                begin
                multiplicand_out <= 11'b11010011010;                    //-6.66
                cnt                 <= 2'b00;
                end
        end

end

endmodule
```

APPENDIX K

Testbench for Bias and Weight ROM Module

```
`timescale 1ps / 1ps

module  TB_values1;

reg             sel;
reg             rst;
reg             clk;
reg     [1:0]   layer;
wire    [20:0]  bias_out;
wire    [10:0]  multiplicand_out;


values1 UUT (sel, rst, clk, layer, bias_out, multiplicand_out);

        initial
                begin
                        #0;
                        clk = 1'b0;
                        rst = 1'b1;
                        sel = 1'b0;

                        #6000
                        rst = 1'b0;

                        #10000                  // first layer
                        layer = 2'b00;

                        #15000
                        sel = 1'b1;

                        #16000
                        sel = 1'b0;

                        #25000
                        sel = 1'b1;

                        #26000
                        sel = 1'b0;

                        #35000
                        sel = 1'b1;

                        #36000
                        sel = 1'b0;

                        #40000                  // second layer
                        layer = 2'b01;

                        #45000
                        sel = 1'b1;

                        #46000
                        sel = 1'b0;

                        #55000
                        sel = 1'b1;

                        #56000
                        sel = 1'b0;

                        #65000
                        sel = 1'b1;

                        #66000
                        sel = 1'b0;
```

```
                    #70000
                    layer = 2'b10;                  // third layer

                    #75000
                    sel = 1'b1;

                    #76000
                    sel = 1'b0;

                    #85000
                    sel = 1'b1;

                    #86000
                    sel = 1'b0;

                    #95000
                    sel = 1'b1;

                    #96000
                    sel = 1'b0;

                    #100000
                    rst = 1'b1;

                    #105000
                    rst = 1'b0;
                    end
always
        begin
                #2500
                clk = !clk;
        end

endmodule
```

APPENDIX L

Bias and Weight Counter (values_cnt)

```
//-------------------------------------------------------------------------------
//
// Title       : values_cnt.v
// Author      : Ivan Teh Fu Sun
// University  : University Teknologi Petronas
//
//-------------------------------------------------------------------------------
//
// Description : provides selection for neuron_values module
//
//-------------------------------------------------------------------------------

module values_cnt (sel, reset, values_sel);

input                   sel;
input                   reset;
output  [1:0]   values_sel;
reg             [1:0]   values_sel;

always @ (posedge reset or posedge sel)
begin

        if(reset)
                values_sel <=2'b00;
        else
                begin
                        case(values_sel)
                        2'b00:  values_sel <= 2'b01;
                        2'b01:  values_sel <= 2'b10;
                        2'b10:  values_sel <= 2'b11;
                        2'b11:  values_sel <= 2'b00;
                        endcase
                end
end
endmodule
```

APPENDIX M

Neuron Module (neuron)

```
//-----------------------------------------------------------------------------
//
// Title       : neuron_mult_sum2.v
// Author      : Ivan Teh Fu Sun
// University  : University Teknologi Petronas
//
//-----------------------------------------------------------------------------
//
// Description : Neuron block containing multiplication and summation blocks
//               Multiplication is using the AddShiftRight (ASR)
//               algorithm for unsigned binary.
//               Value of bias is initial twos complement
//               Value of multiplier and multiplicand is initial unsigned binary
//               integer.
//
//-----------------------------------------------------------------------------

module  neuron2 (start, sel, clk, multiplier_in, multiplicand_in, bias_in,
add_out,status_out) ;

input           start;
input           sel;
input           clk;
input   [7:0]   multiplier_in;   //no sign bit, multiplier value is always positive
input   [10:0]  multiplicand_in;
input   [20:0]  bias_in;
output  [20:0]  add_out;
output          status_out;
reg             status_out;
reg     [20:0]  add_out;         //output in twos complement
reg     [20:0]  AQ_reg;          //Overflow bit included
reg     [9:0]   M_reg;
reg             Mult_sign;       //1 means negative
reg             add_flag;
reg     [3:0]   cnt;


always @ (posedge clk or posedge sel or posedge start)
        begin
        //loop1
        if(start)
                begin
                AQ_reg[20:8]<= 13'b0;
                AQ_reg[7:0]     <= multiplier_in[7:0];
                M_reg           <= multiplicand_in[9:0];
                cnt             <= 4'b0;
                Mult_sign       <= multiplicand_in[10] ;
                //XOR the sign bit, 1 means -ve
                add_out         <= bias_in[20:0];
                status_out      <= 1'b1;
                // tells I/O module that values have been loaded
                add_flag        <= 1'b0;
                end
        //loop1
        else if(sel)
                begin
                AQ_reg[20:8]<= 13'b0;
                AQ_reg[7:0]     <= multiplier_in[7:0];
                M_reg           <= multiplicand_in[9:0];

                cnt             <= 4'b0;
                Mult_sign       <= multiplicand_in[10];
                status_out      <= 1'b1;
                 // tells I/O module that values have been loaded
                add_flag        <= 1'b0;
                end
        //loop1
```

```
        else if(cnt == 4'b1001)
                begin
                if(status_out)
                        begin
                        add_out         <= add_out + AQ_reg[20:0];
                        status_out      <= 1'b0;
                        end
                end
        //loop1
        else if(cnt != 4'b1000)
                begin
                //loop2
                if(cnt != 4'b1001)
                        begin
                        //loop3
                        if(add_flag)            //When Q = 1
                                begin
                                AQ_reg[18:0]    <= {1'b0,AQ_reg[18:1]};
                                cnt                     <= cnt+1;
                                add_flag                <= 1'b0;
                                end

                        else if(!add_flag)
                                begin
                                //loop4
                                if(AQ_reg[0])
                                        begin
                                        AQ_reg[18:8] <= AQ_reg[18:8] + M_reg;
                                        add_flag <= 1'b1;
// status bit notifies that shift needs to take place
// clk not incremented because shift right has not occur
                                        end

                                else if(!AQ_reg[0])
                                        begin
                                        AQ_reg[18:0]    <= {1'b0,AQ_reg[18:1]};
                                        cnt             <= cnt+1;
                                        end
                                end
                        end
                end
        //loop1
        else if(cnt == 4'b1000)
                begin

                        if (Mult_sign)
//only multiplicand value can begin negative
                                begin
//thus changes initial sign depends solely on multiplicand's sign
                                if (AQ_reg != 0)
                                        begin
                                        AQ_reg <= {3'b111,~AQ_reg[17:0]};
//twos complement inversion, +1 not required
                                        end
                                end

                cnt <= cnt +1;
                end
        end
endmodule
```

APPENDIX N

Testbench for Neuron Module

```
`timescale 1ps / 1ps

module  TB_neuron_mult_sum2;

reg             start;
reg             sel;
reg             clk;
reg     [8:0]   multiplier_in;   //MSB bit denotes sign
reg     [10:0]  multiplicand_in;
reg     [20:0]  bias_in;
wire    [20:0]  add_out;
wire            status_out;

neuron2 UUT (start, sel, clk, multiplier_in, multiplicand_in, bias_in,
add_out,status_out);

        initial
                begin
                        #0;
                        clk = 0;

                        #10000;
                        multiplier_in   =  9'b100000100;          //Q
                        multiplicand_in = 11'b10000000010;        //M
                        bias_in                 = 21'h1;

                        #12000;
                        start = 1;

                end

                always
                begin
                        if(status_out)
                        begin
                                start=0;
                                multiplicand_in = 11'b000000001;
                        end

                        if(sel)
                        sel <= 0;

                        if(!status_out)
                        sel <= 1;

                        #5000;
                        clk = !clk;

                end


endmodule
```

APPENDIX O

Neuron Output Multiplexer Module (mux2)

```
//-------------------------------------------------------------------------------
//
// Title       : mux_2.v
// Author      : Ivan Teh Fu Sun
// University  : University Teknologi Petronas
//
//-------------------------------------------------------------------------------
//
// Description : Interface between neuron_mult_sum2 and activation_function's
//               interface.
//               Selects between the outputs of neurons initial hidden layer to begin
//               presented to the neuron_AF_interface
//
//-------------------------------------------------------------------------------

module mux2 (add_in1, add_in2, add_in3, sel, clk, reset, add_out, status_out);

input   [20:0]  add_in1;
input   [20:0]  add_in2;
input   [20:0]  add_in3;
input   [1:0]   sel;
input           clk;
input           reset;
output  [20:0]  add_out;
output          status_out;
reg     [20:0]  add_out;
reg             status_out;

always @(posedge clk or posedge reset)
        begin

        if(reset)
                begin
                status_out <= 1'b0;
                end

        else
                begin
                if(sel == 2'b00)
                        begin
                        add_out         <= add_in1;
                        status_out      <= 1'b1;
                        end

                if(sel == 2'b01)
                        begin
                        add_out         <= add_in2;
                        status_out      <= 1'b1;
                        end

                if(sel == 2'b10)
                        begin
                        add_out         <= add_in3;
                        status_out      <= 1'b1;
                        end

           end

        end
endmodule
```

APPENDIX P

Testbench for Neuron Output Multiplexer Module

```
`timescale 1ps / 1ps

module TB_mux2;

reg             [20:0]  add_in1;
reg             [20:0]  add_in2;
reg             [20:0]  add_in3;
reg     [1:0]   sel;
reg                             clk;
reg                             reset;
wire    [20:0]  add_out;
wire                    status_out;

mux2 UUT (add_in1,add_in2,add_in3,sel,clk,reset,add_out,status_out);

initial
        begin

        #0
        clk = 0;
        reset = 0;

        #1000
        add_in1 = 21'b1;
        add_in2 = 21'b10;
        add_in3 = 21'b11;

        #10000
        sel = 2'b00;
        reset = 1'b1;
        #11000
        reset = 1'b0;

        #20000
        sel = 2'b01;
        reset = 1'b1;
        #21000
        reset = 1'b0;

        #30000
        sel = 2'b10;
        reset = 1'b1;
        #31000
        reset = 1'b0;

        #40000
        sel = 2'b11;
        reset = 1'b1;
        #41000
        reset = 1'b0;

        end

always
        begin

                #5000
                clk = !clk;

        end
endmodule
```

APPENDIX Q

Neuron Output Multiplexer Counter (mux2_cnt)

```
//-------------------------------------------------------------------------------
//
// Title       : values_cnt.v
// Author      : Ivan Teh Fu Sun
// University  : University Teknologi Petronas
//
//-------------------------------------------------------------------------------
//
// Description : provides selection for neuron_values module
//
//-------------------------------------------------------------------------------

module values_cnt (sel, reset, values_sel);

input                   sel;
input                   reset;
output  [1:0]   values_sel;
reg             [1:0]   values_sel;

always @ (posedge reset or posedge sel)
begin

        if(reset)
                values_sel <=2'b00;
        else
                begin
                        case(values_sel)
                        2'b00:  values_sel <= 2'b01;
                        2'b01:  values_sel <= 2'b10;
                        2'b10:  values_sel <= 2'b11;
                        2'b11:  values_sel <= 2'b00;
                        endcase
                end
end
endmodule
```

APPENDIX R

Number Representation Converter Module (interface)

```
//-------------------------------------------------------------------------------
//
// Title       : neuron_AF_interface.v
// Author      : Ivan Teh Fu Sun
// University  : University Teknologi Petronas
//
//-------------------------------------------------------------------------------
//
// Description : Interface between neuron_mult_sum2 and activation_function.
//               Converts 23 bit twos complement number into signed integer.
//               Selects bits 9 to 16 and sign bit for output
//
//-------------------------------------------------------------------------------

module  interface (sel, clk, add_in, status_out, sum_sign_out, sum_out) ;

input                   sel;
input                   clk;
input   [20:0]  add_in;
output                  status_out;
output                  sum_sign_out;
output  [7:0]   sum_out;
reg             [7:0]   sum_out;
reg                             sum_sign_out;
reg                             status_out;


always @ (posedge  sel or posedge clk)
        begin

        if(sel)
                begin
                sum_out <= add_in[15:8];
                sum_sign_out <= 0;
                status_out <= 1 ;  //used to indicate module processing, sel = 0
                end

        if(status_out)
                begin
                if(add_in[20])
                        begin
                        sum_sign_out <= 1;
                        sum_out <= ~sum_out;
                        end
                        status_out <= 0;
                end

        end

endmodule
```

APPENDIX S

Testbench for Number Representation Converter Module

```
`timescale 1ps / 1ps

module TB_neuron_AF_interface;

reg             sel,clk;
reg     [18:0]  add_in;
wire            sum_sign_out;
wire    [7:0]   sum_out;
wire            status_out;

interface UUT (sel, clk,  add_in, status_out, sum_sign_out, sum_out);

initial
        begin

                #0
                sel = 0;
                clk = 0;
                add_in = 19'h00000;

                #10000
                sel = 1;

                #20000
                add_in = 19'h40800;

                #50000
                sel = 1;
        end
always
        begin
                #5000;
                clk = !clk;

                if(status_out)
                        sel = 0;
                end
endmodule
```

APPENDIX T

Activation Function LUT Module (active_function)

```
//-------------------------------------------------------------------------------
//
// Title       : activation_function.v
// Author      : Ivan Teh Fu Sun
// University  : University Teknologi Petronas
//
//-------------------------------------------------------------------------------
//
// Description : ROM block for sigmoid activation function
//
//-------------------------------------------------------------------------------

module  activ_function (summation_in, sel, clk, sum_sign_in, function_out,
status_out);

input   [7:0]   summation_in;
input           sel, clk, sum_sign_in;
output  [7:0]   function_out;
output          status_out;
reg     [7:0]   function_out;
reg             status_out;
reg             status;

always @ (posedge clk )
begin
       if (!sel)
              begin
              if(!status_out)
                     begin
                     casex (summation_in)
                     8'b0000000x:   function_out    <=          8'b00110010;
                     8'b0000001x:   function_out    <=          8'b00110011;
                     8'b0000010x:   function_out    <=          8'b00110100;
                     8'b00000110:   function_out    <=          8'b00110101;
                     8'b00000111:   function_out    <=          8'b00110110;
                     8'b0000100x:   function_out    <=          8'b00110111;
                     8'b0000101x:   function_out    <=          8'b00111000;
                     8'b00001100:   function_out    <=          8'b00111001;
                     8'b00001101:   function_out    <=          8'b00111010;
                     8'b0000111x:   function_out    <=          8'b00111011;
                     8'b0001000x:   function_out    <=          8'b00111100;
                     8'b0001001x:   function_out    <=          8'b00111101;
                     8'b00010100:   function_out    <=          8'b00111110;
                     8'b00010101:   function_out    <=          8'b00111111;
                     8'b0001011x:   function_out    <=          8'b01000000;
                     8'b0001100x:   function_out    <=          8'b01000001;
                     8'b00011010:   function_out    <=          8'b01000001;
                     8'b00011011:   function_out    <=          8'b01000010;
                     8'b0001110x:   function_out    <=          8'b01000011;
                     8'b0001111x:   function_out    <=          8'b01000100;
                     8'b0010000x:   function_out    <=          8'b01000101;
                     8'b0010001x:   function_out    <=          8'b01000110;
                     8'b0010010x:   function_out    <=          8'b01000111;
                     8'b0010011x:   function_out    <=          8'b01001000;
                     8'b0010100x:   function_out    <=          8'b01001001;
                     8'b0010101x:   function_out    <=          8'b01001010;
                     8'b0010110x:   function_out    <=          8'b01001011;
                     8'b0010111x:   function_out    <=          8'b01001100;
                     8'b0011000x:   function_out    <=          8'b01001101;
                     8'b0011001x:   function_out    <=          8'b01001110;
                     8'b0011010x:   function_out    <=          8'b01001111;
                     8'b0011011x:   function_out    <=          8'b01010000;
                     8'b0011100x:   function_out    <=          8'b01010001;
                     8'b0011101x:   function_out    <=          8'b01010001;
                     8'b0011110x:   function_out    <=          8'b01010010;
                     8'b0011111x:   function_out    <=          8'b01010011;
                     8'b0100000x:   function_out    <=          8'b01010011;
```

```
                8'b0100001x:   function_out    <=            8'b01010100;
                8'b0100010x:   function_out    <=            8'b01010101;
                8'b0100011x:   function_out    <=            8'b01010101;
                8'b0100100x:   function_out    <=            8'b01010110;
                8'b0100101x:   function_out    <=            8'b01010111;
                8'b0100110x:   function_out    <=            8'b01010111;
                8'b0100111x:   function_out    <=            8'b01011000;
                8'b0101000x:   function_out    <=            8'b01011000;
                8'b0101001x:   function_out    <=            8'b01011001;
                8'b0101010x:   function_out    <=            8'b01011001;
                8'b01010110:   function_out    <=            8'b01011001;
                8'b01010111:   function_out    <=            8'b01011010;
                8'b010110xx:   function_out    <=            8'b01011010;
                8'b010111xx:   function_out    <=            8'b01011011;
                8'b011000xx:   function_out    <=            8'b01011100;
                8'b0110010x:   function_out    <=            8'b01011100;
                8'b0110011x:   function_out    <=            8'b01011101;
                8'b011010xx:   function_out    <=            8'b01011101;
                8'b011011xx:   function_out    <=            8'b01011110;
                8'b011100xx:   function_out    <=            8'b01011110;
                8'b011101xx:   function_out    <=            8'b01011111;
                8'b011110xx:   function_out    <=            8'b01011111;
                8'b011111xx:   function_out    <=            8'b01100000;
                8'b10000xxx:   function_out    <=            8'b01100000;
                8'b10001xxx:   function_out    <=            8'b01100001;
                8'b10010xxx:   function_out    <=            8'b01100001;
                8'b10011xxx:   function_out    <=            8'b01100010;
                8'b1010xxxx:   function_out    <=            8'b01100010;
                8'b101100xx:   function_out    <=            8'b01100010;
                8'b101101xx:   function_out    <=            8'b01100011;
                8'b10111xxx:   function_out    <=            8'b01100011;
                8'b11xxxxxx:   function_out    <=            8'b01100011;
                // no default required because all conditions are defined
                endcase
                status <= 1'b0;
                end
        end
    else
        begin
        status_out <=1'b0;
        end

        if(!status)
            begin
            if(sum_sign_in)
                begin
                function_out <= 8'b01100100 - function_out;
                end

                status_out <= 1'b1;
                status <=1'b1;
            end
end

endmodule
```

APPENDIX U

Testbench for Activation Function LUT Module

```
`timescale 1ps / 1ps

module  TB_activ_function;

reg     [7:0]   summation_in;
reg             sel;
reg             clk;
reg             sum_sign_in;
wire    [7:0]   function_out;
wire            status_out;


activ_function UUT (summation_in, sel, clk, sum_sign_in, function_out, reset) ;

        initial
                begin
                #0
                clk = 1'b0;
                sel = 1;
                summation_in = 8'b01100100;
                sum_sign_in = 1'b1;

                #20000
                sel = 0;

                #50000
                sel = 1;
                summation_in = 8'b01100100;
                sum_sign_in = 1'b0;

                #60000
                sel = 0;

                end

                always
                begin
                #5000
                clk <= !clk;
                end

endmodule
```

APPENDIX V

Output Threshold Module (threshold)

```
//-------------------------------------------------------------------------------
//
// Title       : threshold.v
// Author      : Ivan Teh Fu Sun
// University  : University Teknologi Petronas
//
//-------------------------------------------------------------------------------
//
// Description : Provides threshold values for the RFNNA output
//
//-------------------------------------------------------------------------------

module threshold (function_in, clk, sel, out, ready);

input   [7:0]   function_in;
input           clk;
input           sel;
output          out;
output          ready;
reg             ready;
reg             out;

always @ (posedge clk or posedge sel)
begin

        if(!sel)
                begin
                ready <= 1'b1;
                end
        else
                begin
                casex(function_in)
                8'b01011001:    out <= 1'b1;
                8'b0101101x:    out     <= 1'b1;
                8'b010111xx:    out <= 1'b1;
                8'b011xxxxx:    out <= 1'b1;
                8'b0000100x:    out     <= 1'b0;
                8'b000010x0:    out <= 1'b0;
                8'b00000xxx:    out <= 1'b0;
                default:                out <= 1'bZ;
                endcase
                ready <= 1'b0;
                end
end
endmodule
```

APPENDIX W

Testbench for Output Threshold Module

```
`timescale 1ps / 1ps

module TB_threshold;

reg             [7:0]   function_in;
reg                             clk;
wire                    out;

threshold UUT (function_in, clk, sel, out, ready);

initial
        begin

        #0
        clk = 1'b1;

        #20000
        function_in = 8'b01100000;

        #40000
        function_in = 8'b00100100;

        #60000
        function_in = 8'b00000011;

        end
always

        begin

        #5000
        clk = !clk;

        end
endmodule
```

APPENDIX X

Control Unit

```
// File         : e:\fypfpgaprojectfolder\Neural Network 3 Neuron\compile\Fub2.v
// Generated    : 05/06/04 09:05:23
// From         : e:\fypfpgaprojectfolder\Neural Network 3 Neuron\src\Fub2.asf
// By           : FSM2VHDL ver. 3.0.4.1

`timescale 1ns / 1ps

module Fub2 (AF_stat, clk, mux1_cnt, mux1_sel_cnt, mux1_sel_inout, mux2_cnt,
mux2_rst, mux2_sel_cnt, n1_stat, neuron_sel, neuron_start, p_ready, p_start, reset,
threshold_ready, values_rst, values_sel, vl_cnt2, vl_cnt, vl_sel_cnt);

input   AF_stat;
input   clk;
input   [1:0]mux1_cnt;
input   [1:0]mux2_cnt;
input   n1_stat;
input   p_start;
input   reset;
input   threshold_ready;
input   [1:0]vl_cnt2;
input   [1:0]vl_cnt;
output  mux1_sel_cnt;
output  mux1_sel_inout;
output  mux2_rst;
output  mux2_sel_cnt;
output  neuron_sel;
output  neuron_start;
output  p_ready;
output  values_rst;
output  values_sel;
output  vl_sel_cnt;

wire    AF_stat;
wire    clk;
wire    [1:0]mux1_cnt;
reg     mux1_sel_cnt, next_mux1_sel_cnt;
reg     mux1_sel_inout, next_mux1_sel_inout;
wire    [1:0]mux2_cnt;
reg     mux2_rst, next_mux2_rst;
reg     mux2_sel_cnt, next_mux2_sel_cnt;
wire    n1_stat;
reg     neuron_sel, next_neuron_sel;
reg     neuron_start, next_neuron_start;
reg     p_ready, next_p_ready;
wire    p_start;
wire    reset;
wire    threshold_ready;
reg     values_rst, next_values_rst;
reg     values_sel, next_values_sel;
wire    [1:0]vl_cnt2;
wire    [1:0]vl_cnt;
reg     vl_sel_cnt, next_vl_sel_cnt;

// BINARY ENCODED state machine: Sreg0
// State codes definitions:
`define S1 5'b00000
`define S2 5'b00001
`define S3 5'b00010
`define N_entry 5'b00011
`define S5 5'b00100
`define S6 5'b00101
`define S7 5'b00110
`define S8 5'b00111
`define S9 5'b01000
`define S10 5'b01001
`define S11 5'b01010
`define S12 5'b01011
`define S14 5'b01100
```

```
`define S15 5'b01101
`define S16 5'b01110
`define S17 5'b01111
`define S18 5'b10000
`define S19 5'b10001
`define S20 5'b10010

reg [4:0]CurrState_Sreg0, NextState_Sreg0;

// Diagram actions (continuous assignments allowed only: assign ...)
// diagram ACTION


//-------------------------------------------------------------------
// Machine: Sreg0
//-------------------------------------------------------------------
//----------------------------------
// NextState logic (combinatorial)
//----------------------------------
always @ (p_start or vl_cnt2 or n1_stat or mux2_cnt or vl_cnt or AF_stat or
threshold_ready or mux1_sel_inout or mux1_sel_cnt or vl_sel_cnt or values_rst or
values_sel or neuron_sel or neuron_start or mux2_sel_cnt or mux2_rst or p_ready or
CurrState_Sreg0)
begin : Sreg0_NextState
        NextState_Sreg0 <= CurrState_Sreg0;
        // Set default values for outputs and signals
        next_mux1_sel_inout = mux1_sel_inout;
        next_mux1_sel_cnt = mux1_sel_cnt;
        next_vl_sel_cnt = vl_sel_cnt;
        next_values_rst = values_rst;
        next_values_sel = values_sel;
        next_neuron_sel = neuron_sel;
        next_neuron_start = neuron_start;
        next_mux2_sel_cnt = mux2_sel_cnt;
        next_mux2_rst = mux2_rst;
        next_p_ready = p_ready;
        case (CurrState_Sreg0) // synopsys parallel_case full_case
                `S1:
                begin
                        next_mux1_sel_inout = 1'b0;
                        next_mux1_sel_cnt = 1'b0;
                        next_vl_sel_cnt = 1'b0;
                        next_values_rst = 1'b1;
                        next_values_sel = 1'b0;
                        next_neuron_sel = 1'b0;
                        next_neuron_start = 1'b0;
                        next_mux2_sel_cnt = 1'b0;
                        next_mux2_rst = 1'b1;
                        next_p_ready = 1'b1;
                        if (p_start)
                                NextState_Sreg0 <= `S2;
                end
                `S2:
                begin
                        next_mux1_sel_cnt = 1'b1;
                        next_mux1_sel_inout = 1'b1;
                        next_values_rst = 1'b0;
                        next_values_sel = 1'b1;
                        next_p_ready = 1'b0;
                        NextState_Sreg0 <= `S3;
                end
                `S3:
                begin
                        next_mux1_sel_cnt = 'b0;
                        next_vl_sel_cnt = 1'b0;
                        next_values_sel = 1'b0;
                        next_neuron_start = 1'b1;
                        NextState_Sreg0 <= `N_entry;
                end
                `N_entry:
                begin
                        next_neuron_start = 1'b0;
                        next_neuron_sel = 1'b0;
                        next_values_sel = 1'b1;
                        next_mux1_sel_cnt = 1'b1;
                        if (vl_cnt2 != 2'b11)
```

```
                NextState_Sreg0 <= `S9;
end
`S5:
begin
        next_neuron_sel = 1'b1;
        next_values_sel = 1'b0;
        next_mux1_sel_cnt = 1'b0;
        if (vl_cnt2 == 2'b11)
                NextState_Sreg0 <= `S16;
        else if (vl_cnt2 != 2'b11)
                NextState_Sreg0 <= `N_entry;
end
`S6:
begin
        next_values_rst = 1'b1;
        next_mux1_sel_inout = 1'b0;
        if (!n1_stat)
                NextState_Sreg0 <= `S17;
end
`S7:
begin
        next_values_rst = 1'b0;
        next_mux2_rst = 1'b1;
        next_mux2_sel_cnt = 1'b0;
        next_mux1_sel_cnt = 1'b0;
        if (mux2_cnt == 2'b10)
                NextState_Sreg0 <= `S11;
        else if (vl_cnt == 2'b11)
                NextState_Sreg0 <= `S12;
        else if (mux2_cnt != 2'b10)
                NextState_Sreg0 <= `S10;
end
`S8:
begin
        next_mux2_sel_cnt = 1'b1;
        next_mux2_rst = 1'b0;
        NextState_Sreg0 <= `S18;
end
`S9:
        if (!n1_stat)
                NextState_Sreg0 <= `S5;
`S10:
begin
        next_mux1_sel_cnt = 1'b1;
        if (AF_stat)
                NextState_Sreg0 <= `S8;
end
`S11:
begin
        next_mux1_sel_cnt = 1'b1;
        next_mux2_sel_cnt = 1'b1;
        if (AF_stat)
                NextState_Sreg0 <= `S14;
end
`S12:
        if (AF_stat)
                NextState_Sreg0 <= `S19;
`S14:
begin
        next_mux1_sel_cnt = 1'b0;
        next_mux2_sel_cnt = 1'b0;
        next_vl_sel_cnt = 1'b1;
        NextState_Sreg0 <= `S15;
end
`S15:
begin
        next_mux1_sel_inout = 1'b1;
        next_mux1_sel_cnt = 1'b1;
        next_values_rst = 1'b0;
        next_values_sel = 1'b1;
        next_p_ready = 1'b0;
        NextState_Sreg0 <= `S3;
end
`S16:
begin
        next_neuron_sel = 1'b0;
```

```
                        NextState_Sreg0 <= `S6;
                end
                `S17:
                begin
                        next_mux2_rst = 1'b0;
                        NextState_Sreg0 <= `S7;
                end
                `S18:
                        NextState_Sreg0 <= `S7;
                `S19:
                        if (threshold_ready)
                                NextState_Sreg0 <= `S20;
                `S20:
                begin
                        next_p_ready = 1'b1;
                        NextState_Sreg0 <= `S1;
                end
        endcase
end

//---------------------------------
// Current State Logic (sequential)
//---------------------------------
always @ (posedge clk or posedge reset)
begin : Sreg0_CurrentState
        if (reset)
                CurrState_Sreg0 <= `S1;
        else
                CurrState_Sreg0 <= NextState_Sreg0;
end

//---------------------------------
// Registered outputs logic
//---------------------------------
always @ (posedge clk or posedge reset)
begin : Sreg0_RegOutput
        if (reset)
        begin
                mux1_sel_inout <= 1'b0;
                mux1_sel_cnt <= 1'b0;
                vl_sel_cnt <= 1'b0;
                values_rst <= 1'b1;
                values_sel <= 1'b0;
                neuron_sel <= 1'b0;
                neuron_start <= 1'b0;
                mux2_sel_cnt <= 1'b0;
                mux2_rst <= 1'b1;
                p_ready <= 1'b1;
        end
        else
        begin
                mux1_sel_inout <= next_mux1_sel_inout;
                mux1_sel_cnt <= next_mux1_sel_cnt;
                vl_sel_cnt <= next_vl_sel_cnt;
                values_rst <= next_values_rst;
                values_sel <= next_values_sel;
                neuron_sel <= next_neuron_sel;
                neuron_start <= next_neuron_start;
                mux2_sel_cnt <= next_mux2_sel_cnt;
                mux2_rst <= next_mux2_rst;
                p_ready <= next_p_ready;
        end
end

endmodule
```

APPENDIX Y

Testbench for RFNNA Processor

```
`timescale 1ps / 1ps

module TB_RFNNA_processor;


reg In1;
reg In2;
reg reset;
reg Start;
reg clk;
wire Out1;
wire Ready;

RFNNA_Processor UUT (In1,In2,Start,clk,reset,Out1,Ready) ;

initial
        begin

                #0
                clk = 0;
                reset = 1;

                #10000
                In1 = 1'b0;
                In2 = 1'b1;


                #30000
                reset = 0;

                #40000
                Start = 1'b1;

                #2750000
                reset = 1;

                #2800000
                Start = 1'b1;
                reset = 0;
        end

always
        begin
                if (!Ready)
                        Start = 1'b0;

                #5000
                clk = !clk;

        end

        endmodule
```