CREATING SCALABLE NEURAL NETWORKS WITH MAXIMAL FPGA RESOURCES

By
EVAN WILLIAMS
Bachelor of Science in Electrical Engineering
Oklahoma State University
Stillwater, OK, USA
2013

Submitted to the Faculty of the
Graduate College of
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
MAY, 2015

CREATING SCALABLE NEURAL NETWORKS WITH MAXIMAL FPGA RESOURCES

Thesis Approved:

Dr. Martin T. Hagan

Committee Chair and Advisor

Dr. Carl D. Latino

Committe Member

Dr. James E. Stine

Committe Member

ACKNOWLEDGMENTS

I would like to express my gratitude to Dr. Hagan for allowing me to work with him. I greatly value his guidance, patience, and understanding that he has shown. I appreciate the time he has spent helping me through the challenges that this work brings.

I want to thank Dr. Latino and Dr. Stine for being on the committee and encouraging me to pursue a masters degree. Their guidance in my undergraduate studies were very influential in my decisions to continue my education. I greatly value their input and discussion because it has been so important to my understanding.

I also deeply respect Dr. Hagan, Dr. Latino, and Dr. Stine for their commitment to helping their students learn. Attending courses and working closely with them, it's obvious how much they care.

Finally, I would like to thank all of my friends, colleagues, and family for supporting me through my education with many wonderful memories here.

Name: EVAN WILLIAMS

Date of Degree: MAY, 2015

Title of Study: CREATING SCALABLE NEURAL NETWORKS WITH MAXIMAL FPGA RESOURCES

Major Field: ELECTRICAL ENGINEERING

Abstract: Hardware implementations of Artificial Neural Network (ANN) architectures can take advantage of parallelism in the ANN algorithm. Using automated procedures, arbitrary amounts of Field Gate Programmable Array (FPGA) resources can be allocated to calculate arbitrary ANN algorithms. This document analyzes trade-offs between the speed and area required as an arbitrary ANN algorithm is computed over arbitrary hardware sizes. Comparing the calculated number cycles and area as the number of inputs to a neuron and the number of neurons in a layer vary, it is found that there exists an optimal number of inputs and neurons for a given ANN algorithm.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER 1

Introduction

The goal of this thesis is to develop the most reconfigurable and efficient implementation of a neural network on a field programmable gate array (FPGA). The result should allow multilayer networks of arbitrary architecture to be implemented on FPGAs with arbitrary resources. A reconfigurable neural network is useful because it allows a designer to easily make and test trade-offs between area, delay, and power consumption of different sizes of neural network. Hardware is known for its ability to process data and signals in parallel. Parallel processing typically decreases the delay for an output value but comes at a cost of increased logic gates and therefore power consumption. By providing simple variables to control the parallelization, and equations that characterize the cycles and slices required, the optimal amount of hardware needed to implement a given neural network for a given FPGA chip can be calculated. A fully reconfigurable design should use modularized blocks to test different number representation and bit length for changes in size and accuracy. By making careful design decisions, the size and efficiency of the design can be maximized for a given artificial neural network (ANN) algorithm.

The main contribution of this thesis is a fully reconfigurable FPGA implementation of arbitrary multilayer neural networks. This was developed by the following stages: the single-input hardware neuron, the multiple single-input hardware neurons, the multi-input hardware neuron, and the multiple multi-input hardware neurons. The construction of each of these stages is described structurally and behaviorally. Each case is important, because comparing each of the stages allows for verification

of each of the designs and shows the development process along the way to a fully functional and fully reconfigurable network. Equations are also developed in each chapter to make predictions about the speed of computation and hardware space requirements.

The thesis begins with some background of the project, and then the development process leads up from a simple case to the fully scalable and reconfigurable network. In Chapter 2, the problem is presented, and is discussed in the context of previous developments. Next, in Chapter 3, the single-input hardware neuron design is analyzed. In Chapter 4, the multiple single-input hardware neurons case, a hardware layer is created by copying multiple hardware neurons, and the concept of hardware layer iterations is introduced. Then, in Chapter 5, the neuron is modified to accept multiple inputs. This introduces an adder tree to the neuron and the concept of input iterations. Finally, in Chapter 6, the ideas are combined into the multiple multi-input hardware neurons, where both input iterations and hardware layer iterations are combined to form the final equation for the cycle requirements. These equations are coupled with the required slice equations and the limitations of an FPGA chip to see what networks would be most efficient for certain cases. Chapter 7 summarizes the results and provides recommendations for future work.

CHAPTER 2

Literature Review

The objective of this research can be stated as follows. Consider that there are a limited amount of FPGA resources available. Design an automated procedure to make maximum use of these resources in such a way that arbitrary neural networks can be simulated most efficiently. This has not been done before, but there have been FPGA implementations of neural networks that have been proposed previously. This chapter reviews some of that earlier work.

Artificial neural networks (ANN) are known for their ability to be implemented in parallel hardware structures. Three types of parallelism intrinsic to an ANN have been described as spacial parallelism, algorithmic parallelism, and layer parallelism [1]. Spacial parallelism is equivalent to increasing the number of neurons in a layer, while algorithmic parallelism is equivalent to increasing the number of inputs to a hardware neuron. Layer parallelism duplicates the hardware layers, but is not truly parallel in computation, because the layers only allow for pipelining. The work by Zhu [1] focused on the general concept of computing the ANN algorithm in parallel, but did not consider arbitrarily sized networks.

Using a single hardware neuron, and no parallelism, it has been shown that arbitrary multi-layer perceptron (MLP) networks can be calculated by using minimal hardware resources [2]. This design considered the fully parallel structure and found that, when using floating-point, the design would consume significant resources for an increase in speed. Because the hardware consumed such a small area, the design was considered quite useful. Other strategies have implemented ANN algorithms with

some degree of scalability.

Static sizes of software networks have been considered in other works, and they have shown that networks can be computed more quickly by exploiting spacial parallelism over a single hardware layer [3], [4], [5], [6]. Using a single hardware layer is referred to as muxing the inputs to the layer. However, these designs used a single processing element, a neuron that accepts only one input.

Another design has shown that networks can be computed more quickly by exploiting algorithmic parallelism [7]. However, this design did not consider adding multiple neurons to a hardware layer and used layer muxing over a single neuron to run the calculations.

There exist many other ANN hardware designs which have been implemented, but these do not account for arbitrary networks, or use only a single hardware description [8], [9], [9], [10]. These designs found that using fixed-point representation achieves the maximum accuracy with the smallest hardware requirements.

There exists an implementation of ANN hardware which requires less space in floating-point representation than fixed-point [11], but most papers have found that fixed-point consumes significantly less space and is more accurate for general implementations of an ANN [7], [12].

Using an optimized efficient hardware structure for floating-point dot-product calculations [13] could make floating-point more viable for scalable networks. However, more efficient fixed-point dot-product calculations that use a single carry propagate adder (CPA), instead of an adder tree, have been suggested [7]. So, using fixed-point may still be better. Other optimizations, like efficient pipelined matrix multiplication, can calculate various sizes of matrix-matrix multiplications [14], [15]. However, because the calculation for an MLP ANN is a vector-matrix multiplication, the structures for matrix-matrix multiplication are not very useful.

This thesis analyzes the trade-offs between algorithmic parallelism and spacial

parallelism when implementing arbitrary software networks. Changing the number of inputs to a neuron and changing the number of hardware neurons in a layer provides an increase in speed while consuming more hardware resources. Using memory to store the arbitrary network, any amount of hardware will be able to compute any ANN algorithm at varying speeds. These trade-offs are analyzed by developing equations to predict the cycles and slices required by the hardware, then comparing the measurements to the predictions. Instead of using the most efficient floating-point or fixed-point computation methods, a generic multiplier and adder tree structure can allow either fixed-point or floating-point modules to be configured by the user. However, optimizations could be made to the current design to increase speed and decrease area when increasing the number of inputs to a layer.

The research described in this thesis is different than previous work described above, in that the proposed automated procedures can make maximum use of an arbitrary amount of FPGA resources. In addition, arbitrary multilayer neural network architectures can be simulated on the resulting hardware.

CHAPTER 3

Single Input Hardware Neuron

In developing a scalable neural network, of the simplest hardware to analyze is the single-input hardware neuron (SIHN) case. This case allows for easier testing and verification during the beginning of development because the block diagram can be more easily represented, and the algorithms are more easily compared to the speed and size of the network. This limited hardware must calculate any size of network. The neurons in the network to be calculated are called software neurons because they are represented in memory. A hardware neuron is a combination of logic gates used to calculate a single neuron's output. A more clear distinction between software and hardware neurons is covered in a later section of this chapter. Understanding a SIHN in the context of a software neural network is the foundation for describing a scalable hardware network, the goal of the document. In order to understand the context of a SIHN, the software neural network should be described first.

3.1 Computation of a Neural Network

Of the multiple types of neural networks that could be considered, the primary network of discussion will be the feed-forward multi-layer network. This type of network contains multiple neurons stacked in layers that cascade to a final output or set of outputs. To better understand this structure, we should first consider the simplest component of the network, the software neuron.

The neuron shown in Figure 3.1 describes the most basic element of a neural network. The purpose of the neuron is to respond to stimuli in a similar way that

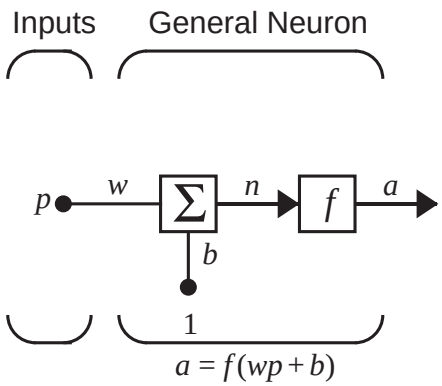

Figure 3.1: Single-Input Software Neuron

nature responds to inputs. Using numbers to represent the activation state of the neuron, the internal sensitivities of the neuron can be configured by setting weights and biases. This neuron takes an input p and multiplies it with a weight w. This product is then summed with a bias b. The transfer function represented by f is used to "activate" the neuron by leveling the output to a range of values, in this case ± 1, similar to how chemicals in nature can increase in concentration in order to trigger the activation of a subsequent neuron. The transfer function that will be used in this document is the *tansig* transfer function found in Equation 3.1. The neuron has an input transition region, the region of input values where the output values noticeably change, between approximately ± 4. A graph of the *tansig* function can be found in Figure 3.2. The importance of this smooth, i.e. differentiable, transfer function function is that it can be used to train the network through backpropagation techniques[16, p. 11-6]. However, backpropagation is not the focus of this document, and so only the forward calculation of the network is analyzed.

$$a = \frac{e^n - e^{-n}}{e^n + e^{-n}} \tag{3.1}$$

Equation 3.1 uses the n variable found in Figure 3.1 and results in the value a, creating the output of one neuron. The behavior of the transfer function is the key component to creating a behavior similar to how some neurons in nature respond

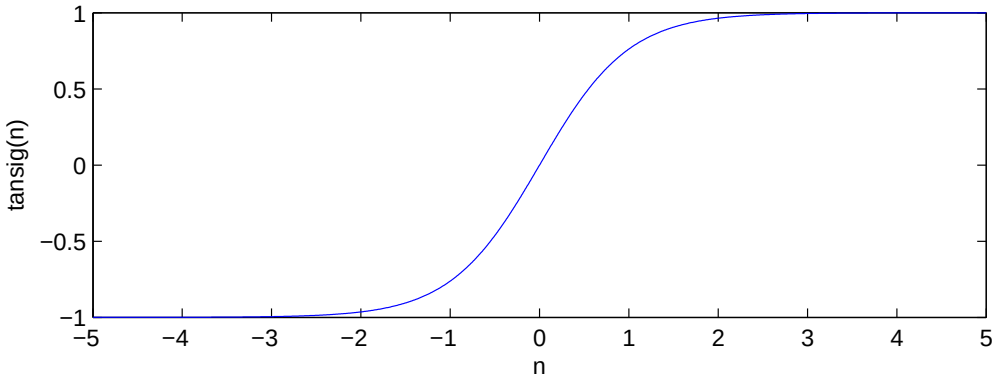

Figure 3.2: Tansig Graph

to stimuli. This is part of the reason why this algorithm is called a neuron. The switching behavior of the transfer function can be seen in Figure 3.2.

From Figure 3.3, in order to accommodate R inputs to a neuron, a vector of inputs and weights, length R, can be used. This multi-input software neuron calculates a dot-product by multiplying each of the inputs with a corresponding weight. This dot-product is the fundamental function of a hardware neuron, so it is important to understand how a dot-product is calculated. Equation 3.2 explicitly shows the calculation of the n intermediate values. In matrix representation, the size of the matrix is described as the number of rows by the number of columns, represented as $row \times column$. When referencing a value within a matrix, it is common to use lower case, and non-bold letters. This is why the values of w used in Figure 3.3 are not bold until they are referenced as a matrix in the function below the figure.

$$n = w_{1,1}p_1 + w_{1,2}p_2 + ... + w_{1,R}p_R + b \tag{3.2}$$

After the dot-product is calculated, only one bias added, no matter the number of inputs, which results in the final value of n passed through the transfer function resulting in the final neuron output a.

Shown in the equation of Figure 3.3, the $\mathbf{p}$ input is now bold in order to represent the value as a vector, length R, and the $\mathbf{W}$ variable is capitalized and made bold

in order to show that the weight is now a matrix of values, 1 row by R columns. Notice how the vectors are multiplied in Figure 3.3 as $\mathbf{Wp}$. This is important for the notation that will be used in describing the hardware further in the document. A key concept here is that $\mathbf{W}$ has only 1 row for 1 neuron. So when multiplying $\mathbf{Wp}$, the multiplication of matrices $1 \times R$ and $R \times 1$ will result in a 1×1, single scalar output value.

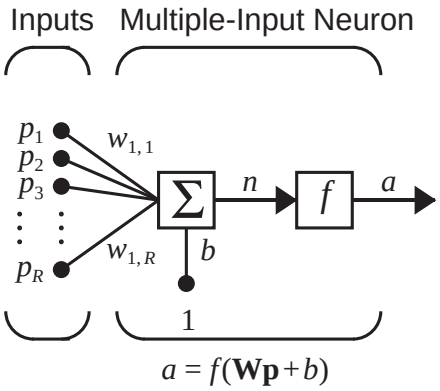

Figure 3.3: Multi-Input Software Neuron

Multiple multi-input neurons can be stacked together in order to create a layer of neurons that provide multiple-layer outputs. Figure 3.4 shows how a layer is constructed and the equation to calculate the layer. Notice that the bias $\mathbf{b}$ has now changed from a single value to a vector of length S, the number of neurons in the layer.

There are R inputs to the network, and S neurons per layer. All inputs are fed into each neuron, meaning that, multiplying $\mathbf{W}$, a matrix now of size $S \times R$, and $\mathbf{p}$, a vector of size $R \times 1$, will result in a vector of the size of $S \times 1$, the number of neuron outputs in the layer. The bias variable $\mathbf{b}$ is also size of $S \times 1$, allowing the addition of each individual bias to the resulting product. Finally, the transfer function is applied to each resulting value, and the final vector $\mathbf{a}$ is also $S \times 1$. Using matrix notation allows for a more elegant description of the network as shown at the bottom of Figure 3.4. These outputs will need to be individually stored in order to be used by another

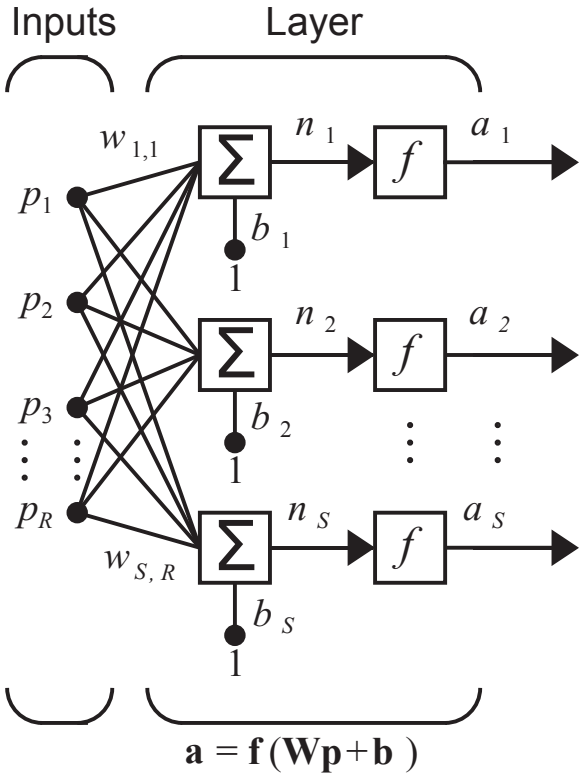

Figure 3.4: Multi-Input Single-Layer Neural Network

layer in a multi-layer network. An example of a multi-layer network can be found in Figure 3.5.

Figure 3.5 represents a general network calculation that should be able to be implemented by any amount of hardware. Notice the notation in Figure 3.5 is similar to the notation found in Figure 3.4. However, there are additional superscripts for each output, transfer function, bias vector, and weight matrix, which corresponds to the layer number. The initial inputs are calculated, then cascaded through the network resulting in the final value or values. Layering outputs is crucial for a neural network to be able to represent any arbitrary function, so the hardware must be able to support the cascade of layers. Notice that each of the layers can have their own independent number of inputs, or neurons, along with different weights, biases, and transfer functions.

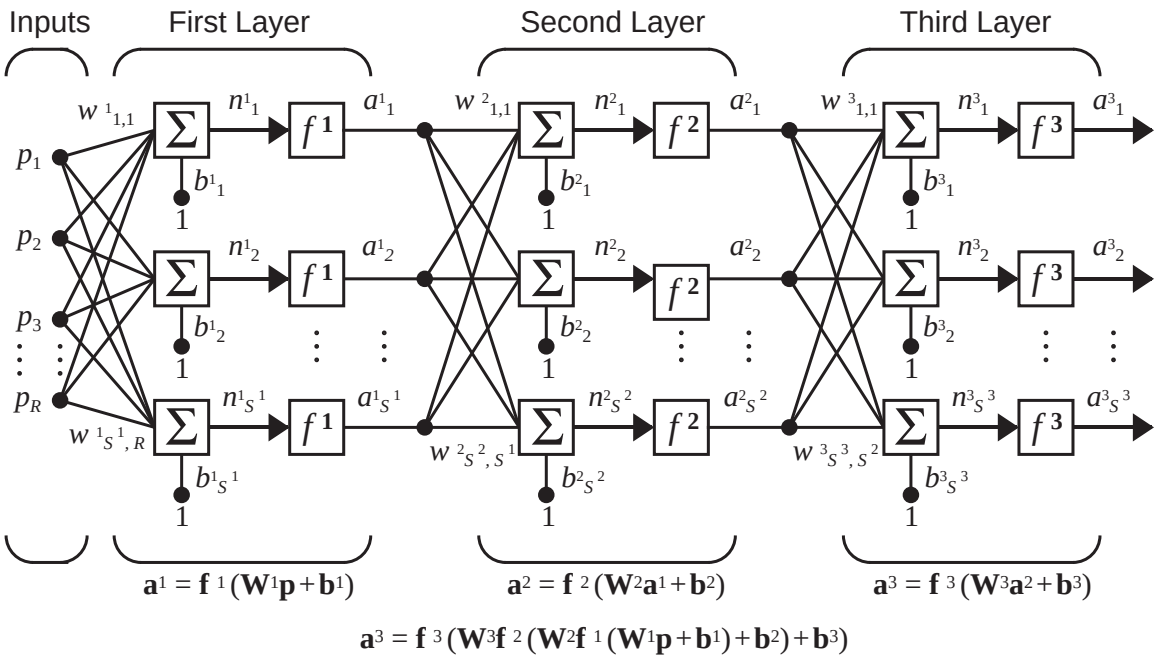

Figure 3.5: Multi-Input Multiple-Layer Neural Network

3.2 Overview of Hardware Implementation of Single-Input Hardware Neuron

The calculation, shown at the bottom of Figure 3.5, can be done with a minimal amount of hardware, as described in a previous paper [2], but can be sped up by using more hardware, exploiting parallelism in the function. The increase in speed obtained through parallelism has been briefly described before [2], however, this document intends to explore the topic in greater detail by implementing different amounts of hardware to perform this calculation. The trade-off is that more hardware uses more resources on the chip and therefore requires more energy and increases design complexity. Although Figure 3.5 shows only three layers, the hardware should be able to calculate any number of inputs and any number of layers no matter what hardware resources are available.

A clear distinction needs to be made between the hardware and software neurons. The network described in the previous section is a software neural network, sometimes

referred to as the artificial neural network (ANN) algorithm [17, p. 351]. This software network is described in random access memory (RAM) and read only memory (ROM) and should be able to be computed for any size of network. The hardware neuron consists of digital logic circuits, which will be used to implement the software network. The capabilities of the hardware to process in parallel a neuron or layer is what we refer to as a hardware neuron or hardware layer. The cases that are considered will look at multiple-hardware-inputs, and multiple-hardware-neurons. These will be covered in greater detail later.

Multiple-hardware-layers could be created, but because outputs of one layer must be calculated before they can be used as inputs to the next layer, the operation is serial. This serial operation can only be solved efficiently in hardware when a limitation is put on the number of outputs of a network. If the maximum number of neurons for a network were known, then multiple hardware layers could be beneficial. Parallelism could be exploited when saving each output value of each layer, but in order for the hardware to calculate any size of arbitrary network, the values must be saved to memory. So, the speedup would be proportional to the cycles required to store a single value and the number of gates required for an entire hardware layer. Because the number of cycles to store a value is minimal, and limitations would need to be placed on the number of software neurons for a layer, the multiple-hardware-layer case was not considered. Therefore, all hardware networks created will be have a single layer.

The amount of hardware can be smaller than the software network desired to be represented, so the hardware must use RAM and ROM to represent the larger software neural network. Then, the limited hardware can iterate over ROMs and RAMs in order to compute each input, each neuron, and each layer. For this section we consider the general calculation using only one SIHN. The diagram in Figure 3.6, similar to a previous paper [2], shows the general idea for creating a SIHN.

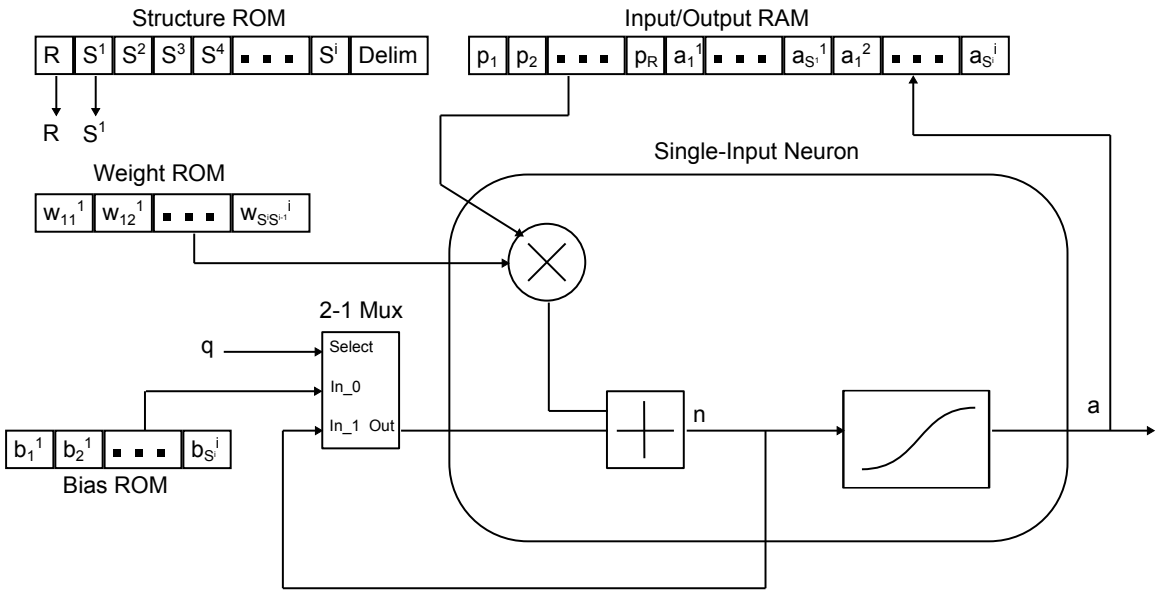

Figure 3.6: General Single-Input Hardware Neuron Diagram

The neural network algorithm can be computed in hardware using memory modules, a multiplier, an adder, a lookup table for the transfer function, and a combination of simple state machines. The memory modules are needed to store the inputs, outputs, and values representing the network architecture. The values of the software network required for the calculation are shown in the RAM and ROM blocks of Figure 3.6.

The Structure ROM contains the number of inputs for each layer. These are the values that will need to be loaded each time a new layer begins. The last layer is signaled when a delimiter is reached. The variables R and S come from the notation in Figure 3.5. Notice that because the inputs for layer $i+1$ are the outputs of the layer i, the Structure ROM can be implemented as a shift register.

For the most basic neuron hardware, a single multiplication module can be used along with a single adder in order to sum a series of inputs to calculate n, as in Equation 3.2. Each of the values can be loaded serially from the Weight ROM and Bias ROM. The 2-1 mux will load the bias on the first iteration in the loop, then will load the accumulated output n during the dot-product calculation. Values from

the weight matrix are loaded for each corresponding input, while the bias values are updated only after one hardware neuron completes. After iterating over one set of inputs, the n value can then be passed through a transfer function to calculate a single neuron's output. This neuron's output value, $a^i_{S^i}$, where i is the layer number, can be stored in the Input/Output RAM to be accessed later for each subsequent layer's inputs. The Input/Output RAM contains the inputs to the network, and then the outputs for each neuron. The inputs are loaded one at a time until the number of inputs in the layer is reached, then a check is made to see if the number of neurons is reached. If not, then reload the inputs again for the next neuron. Otherwise, begin the next layer. The previous layer's outputs are then loaded as the new inputs.

3.3 Hardware Considerations

There are multiple ways to describe hardware structures, and understanding these methods can aid in understanding the trade-offs in hardware space and cycle requirements. Some of the basic decisions are whether a Hardware Descriptive Language (HDL) should be used or a purely schematic approach. A typical approach is to use an HDL, creating a combination of schematics (implemented as structural modules alongside behavioral modules), in order to provide understandable abstractions while still providing the flexibility of an HDL. The structural modules are implemented by instantiating the behavioral modules shown in the schematic and then wiring them together with no behavioral code. The behavioral modules are typically shown with state machines and flow charts and are described in code with procedural blocks that allow a compiler to generate gates for the logic described. A field programmable gate array (FPGA) can be used to test and use the hardware described in the HDL. Being able to synthesize the hardware would imply that the device could also be turned into an application specific integrated circuit (ASIC), which typically are run at faster speeds within a smaller footprint than an FPGA.

3.3.1 Hardware Descriptive Language Considerations

Multiple HDLs exist, and there are multiple trade-offs between the languages. Choosing a language is important because the language changes how the behavioral simulation software simulates the timing between modules. Popular languages considered are Verilog, System Verilog, or VHSIC (Very High Speed Integrated Circuit) Hardware Descriptive Language (VHDL). The main benefit for using VHDL is it's strong typing scheme[18], but this trade-off is that the code becomes longer and takes more time to read through and understand. A strong typing scheme means that the compiler will make less assumptions about what the code is attempting to generate, which can lead to less errors later on in the project. Another benefit is that the timing and cycles simulation in VHDL is deterministic, meaning that the timing found for given code is well defined for all computers running the same simulation code, so results will always be the same. This is not necessarily true for other languages. The longer VHDL code can be less intuitive to a new user, meaning that modifying the code will take more time [18]. The main benefit of using Verilog would be decreased design time and simplicity of code [18], but the behavioral simulation is not guaranteed to be deterministic for every version of Verilog. Because of the existing framework, and the strong typing scheme, VHDL was used in order to build on previous work and avoid errors in the long run.

3.3.2 FPGA Selection

In order to test the network in hardware, an FPGA can be used. Two of the most popular manufacturers of FPGA's are Xilinx and Altera. The chips they produce use different hardware and utilization schemes, but both could implement the hardware described in this document. Understanding the components on the selected FPGA can help clarify the size requirements and limitations of the hardware that will be described. The manufacturers provide software packages in order to simulate hardware

signals and test hardware on their chips. So choosing an FPGA also implies choosing the simulation software you will be using. In order to support newer chips, the project was moved from Xilinx's Interactive System Environment (ISE) into Xilinx's Vivado. Vivado has essentially the same functionality as ISE, but with a different graphical user interface (GUI). Even though the project was not implemented with Altera software, it should be possible to export the majority of the code into Altera's editor to test on Altera boards. The Xilinx IP cores would need to be replaced, but the connections between the cores would be synthasizable afterward. The code should also be able to be backwards compatible with older boards and chips too because of the old language standard.

The FPGA that was used was the Artix®-7 100t, part number XC7A100T. A student learning board, the Nexys 4, comes with the XC7A100T. This board, created by Digilent, allows for quickly implementing and testing hardware described in HDLs. Some of the statistics of the Artix®-7 100t can be found in the Table 3.1. The part number for the chip is XC7A100T. Notice that the number of DSP slices is 240. DSP slices can be used to create many multipliers so that more neurons can fit on the board.

Table 3.1: Artix®-7 FPGAs[19]

Part Number	XC7A100T	XC7A200T
Logic Cells	101,440	215,360
Slices	15,850	33,650
CLB Flip-Flops	126,800	269,200
Max Distributed RAM(Kb)	1,188	2,888
Block RAM/FIFO (36 Kb each)	135	365
Total Block RAM (Kb)	4,860	13,140
DSP Slices	240	740

A logic cell contains a Look-up Table (LUT), a flip-flop, and wires to adjacent cells. A 4 input LUT produces a single bit output, meaning that the LUT can be configured for gates like an and, or, xor, etc. A flip-flop is a basic memory component used to save a bit. In this architecture, there are 8 flip-flops and 4 LUTs per slice. Block RAM is special to Xilinx FPGAs and is located in strips across the chip. Distributed RAM can be single cycle access memory, making it more convenient to use. In large sizes of block RAMs, an exceptionally long delay path can be formed when one section of block RAM is connected to another set of block RAM across the chip. However, because the block RAM is dedicated memory, unlike a slice, it can be used in order to free slices for more neurons. The highest range chip in the Artix®-7 family, the XC7A200T, can more than double the capacity of the current chip. This is significant in case the hardware limitations of the chip are limiting the desired network.

3.3.3 Number Representation

Bit length and representation of numbers is critical to the size, speed, and capability of hardware. However, the goal is to describe and analyze a scalable network, so these trade-offs could be analyzed in greater detail in future work. Two basic representations that could be considered are fixed-point and floating-point. Accuracy is lost when a value cannot be represented in the number of bits available. This means increasing the bit length of values being multiplied could resolve accuracy issues. The hardware described should be able to easily accommodate different bit lengths and bit representations. Appendix A contains an analysis between the areas of floating-point and fixed-point multiplication and addition Intellectual Property (IP) cores. Although fixed-point is significantly smaller, floating-point is used in order to avoid overflow [17, p. 56] during the calculation of the dot-product.

Testing later in the chapters uses the floating-point values and represents them in hex format. It may be useful to describe how to calculate the base 10 value from the

floating-point hex representation. The floating-point values use 5 bits of exponent, with 11 bits of mantissa, for a total of 16 bits. The signed bit is included in the exponent, so the calculation for converting the binary value is given in Equation 3.3. The equation uses a signed bit to make the number negative and an exponent with an offset to allow for a wider range of value representation. While fixing the exponent at a range of 1 to 2, the equivalent accuracy for signed fixed-point would be equal to the number of bits in the mantissa plus 2 bits (one for the constant 1 and another for the sign), meaning that an equivalent fixed-point representation for this range would use 11 plus 2 bits. This means that for an equivalent fixed-point bit length of 16 bits, the accuracy could be increased by 2 raised to the power of the number of additional bits, or 2^3. The result is 8 times more accurate over the range of 1 to 2.

$$-1^{\text{bit}_{15}} \times 2^{\text{bits}_{14:11}-7} \times (1 + \sum_{i=1}^{11} \text{bit}_{11-i} \times 2^{-i}) \tag{3.3}$$

3.3.4 Xilinx Intellectual Property Cores

IP cores are provided by Xilinx and Altera in order to speed up development times. The design described in this thesis uses memory cores and floating-point cores in order to store information and do the computation. Using cores is not always a good idea, because they require regeneration every time a parameter describing the core changes. Regeneration rebuilds the core as a new project, and then synthesizes it for hardware. For a scalable neural network that has changing memory sizes, it can be cumbersome to regenerate cores for each new memory size that needs to be tested.

As mentioned earlier, Xilinx provides block RAMs in order to more efficiently store information on the chip. However, long delay paths can be created using the block rams. Block RAMs can also have multiple cycle access times. This means that using distributed RAM is typically faster and easier to analyze. In order to fill up the entire FPGA with neurons, both the distributed RAM and the block RAM would

need to be used.

One of the options when creating the floating-point cores is the number of pipeline cycles. These can determine the number of delays per cycle to calculate the final output from a multiplier or adder. The simulation tests are behavioral simulations, meaning that they do not actually consider real timing values. However, the behavioral simulation is still useful to ensure that logic is correct for simulation, and to see the delay chain between signals that require a clock. Because the floating-point components do not require a clock, results could be modified to appear to have infinite speed when doing a calculation. However, this would be incorrect because longer and more complicated chains of logic require more time. By using the pipelined versions of the cores, the reduction in cycles from exploited parallelism can be seen in later chapters.

3.4 Implementation in VHDL

To describe the network, the neuron can be abstracted so that it is in its own block, and then a network_control block can be coupled with ROMs and RAMs to feed in and read out data. Using this structure will be useful for when the neuron needs to be copied multiple times for the multiple neuron case. Using the abstractions means that there will be multiple schematics, each moving from the most simple level in the neuron, outward to the entire network structure.

3.4.1 The Single-Input Hardware Neuron Schematic

The fundamental block for the network is the SIHN. This can be described structurally with schematic shown in Figure 3.7. There are five elementary blocks, the neuron_ctrl block, the multiplier block, the b_adder block, the b_reg block, and the transfer_function block, which can be combined to form the SIHN block called the neuron. This is significant because with only a single input, weight, and bias at a time

into the hardware, the algorithm for a multi-input software neuron can be calculated. The primary connections into the neuron are the clock, enable pin, the weight, the input value, the bias, and a request to send signal (RTS). The output of the neuron is a clear to send signal(CTS), neuron done signal, and the output a. When the neuron_ctrl block sees that new inputs are available, it enables the multiplier block.

The multiplier block in Figure 3.7 takes in the weight and the input value, 16-bit floating-point, and then outputs a 16-bit value. The ready value of the multiplier is attached to the enable pin of the adder. So, when the output of the multiplier is done, it triggers the b adder to add. The b_adder module, when enabled by the ready signal of the multiplier, will accept and add the multiplied values and then the initialized bias. After the first iteration, it will take in the intermediate dot-product value from the b_reg module. When it is complete, the b register latches the value, and then signals the neuron control block that it is ready for the next set of inputs. If the b_reg value was latched on the last input for the layer, then the transfer function is enabled, converting the n value into the a output. This is the calculation shown in Equation 3.2.

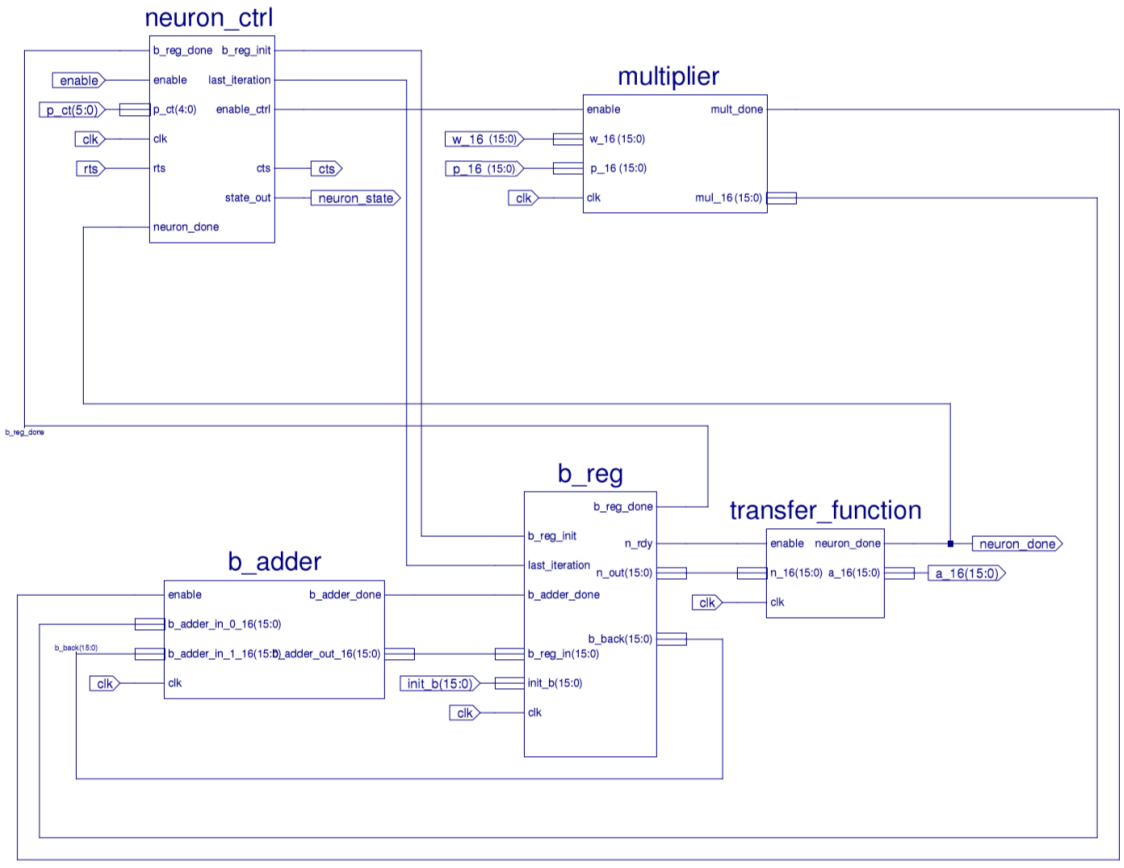

Figure 3.7: Neuron Schematic

Table 3.2: Neuron Schematic Signals

Signal	Description
enable	Enables only the neuron_ctrl block.
clk	Clocks signals for timing.
p_ct	Number of software inputs to process.
rts	External module sets high to request to change inputs.
cts	Neuron control sets high to allow change of inputs.
neuron_state	Allows for easy test bench access to states.
w_16	The weight value.
p_16	The input value.
a_16	The output value.
init_b	The bias value.
neuron_done	Set high when output is ready.
b_reg_done	Set high when b_reg is finished saving.
b_reg_init	Set high when input bias needs loaded into b_reg.
mult_done	Set high when multipliers are complete.
b_adder_done	Set high when the adder is complete.
n_rdy	Set high when n is computed.
n_out	The value of n.
b_back	Value used to accumulate values, initialized as bias.

Because the neuron is controlled by the control block using state machines, a flow chart, seen in Figure 3.8, can describe the states the neuron traverses. The control states begin by initializing the neuron, this state is entered every time the neuron is disabled. Upon enabling, the neuron loads a single input and weight multiple times, until it reaches the number of inputs for the current layer. A counter keeps track of the number of inputs and compares the value to the p_ct wire. The b_reg is set with the input bias when loading the first set of inputs. The module will latch only when the output values are appropriate in order to prevent an infinite loop of additions occurring. It is critical that only each partial value is added only once in order for the calculation to be correct. The control block then waits on the first partial dot product to be completed. This is signaled after the b register latches the output value. When all input values to the dot product are calculated, then the transfer_function block signals the b register to turn on the transfer function. The transfer function in Figure 3.7, will take in the final value from the b register, and then use rounding logic with a lookup table in order to find the corresponding transfer function value. After the value is ready in the transfer function block, the neuron will signal to modules above it that its value is ready. When the transfer function is done, then the control block signals to upper modules outside of the neuron that the value is ready. The state that is used as an output is intended for debugging purposes and was not used for logic in upper modules. Each control block has a state output for this debugging.

Notice that in the schematic of the neuron, Figure 3.8, there is an RTS input, and a CTS output. These handshaking pins are used by each module with their states in order to ensure that each value is latched at the correct time. Because each module will use this convention, it is useful to describe the process in general. A description of typical handshaking schemes can be found in Appendix B. The modules that send the data to the neuron have RTS signals, because they send data into the neuron,

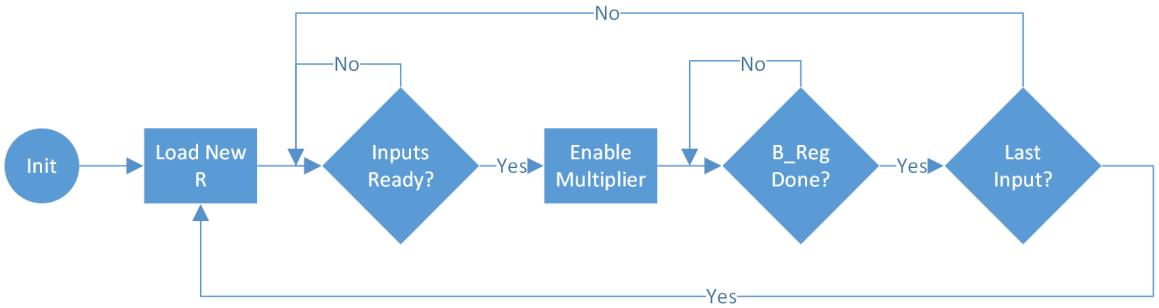

Figure 3.8: Neuron Control Block Flow Diagram

while the neuron uses a CTS signal in order to tell the modules declared above not to change values.

3.4.2 Adding the Weight ROM to the Neuron

An abstraction of the neuron is useful to enable easy debugging and understanding of the project. Before adding all of the ROMs and RAMs to the project, the neuron can be combined with only the weight ROM to ensure that the weights are loaded into the neuron correctly. In order to differentiate this block from the neuron block, it can be named the w_hardware_neuron.

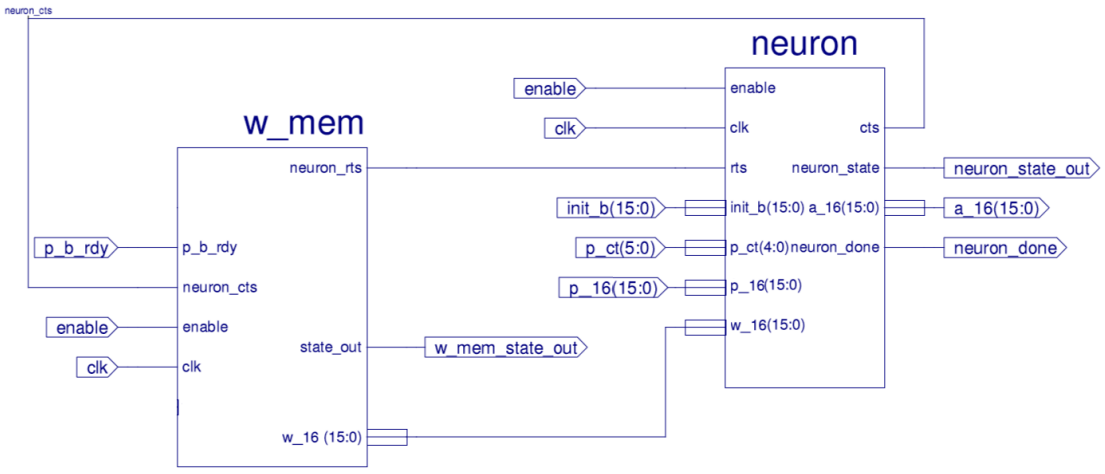

Figure 3.9: The w_hardware_neuron Schematic

Table 3.3: Weight ROM Schematic Signals

Signal	Description
enable	Enables both the w_mem and neuron.
clk	The clock for the modules.
p_b_rdy	Set high when input and bias are ready.
neuron_rts	Set high when inputs, bias, and weights are ready.
neuron_cts	Set high when neuron allows values to change.
w_16	The weight value.
w_mem_state_out	The state of the w_mem for the test bench.

In Figure 3.9, two modules can be seen. The w_mem contains a state machine for loading the values into the neuron and the Weight ROM from Figure 3.6. Although this may seem to add unnecessary complexity at this development stage, it will make expanding the number of Weight ROMs easier for later in the project. The only connections between the w_mem block and the Neuron are the RTS signal and the actual weight value at this time.

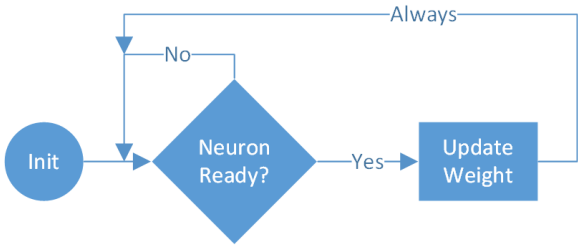

Figure 3.10: The w_mem Flow Diagram

The states of the w_mem module, shown in Figure 3.12, are very simple for the single input, single neuron case. It simply loads a new value for each time the neuron requests a new value using the CTS signal. The p_b_rdy signal is anded with an internal w_mem_rdy signal in order to generate the RTS signal for the neuron.

3.4.3 Adding the Bias ROM to the Neuron

Combining the w_hardware_neuron with a bias ROM uses another abstraction to isolate modules into smaller components. In Figure 3.11, the bias ROM is connected to the w_hardware_neuron with the b_rdy signal, the bias input, and the hardware neuron done signal.

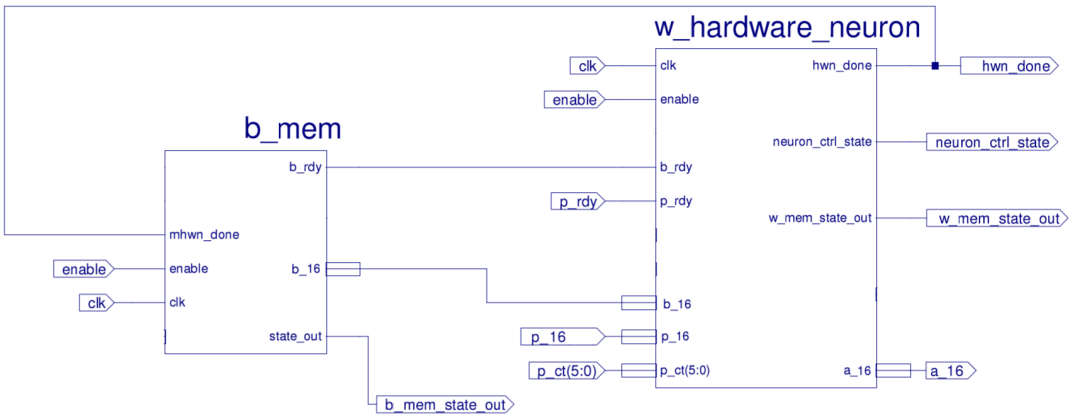

Figure 3.11: The bw_hardware_neuron Schematic

Table 3.4: Bias ROM Schematic Signals

Signal	Description
enable	Enables b_mem and w_hardware_neuron.
clk	Clock for the modules.
p_rdy	Set high when the input value is ready.
b_rdy	Set high when the bias value is ready.
b_16	The bias value.
hwn_done	Set high when the output of the neuron is ready.

The two modules in Figure 3.11 are the w_hardware_neuron and the b_mem module. The b_mem module contains the bias ROM from Figure 3.6 and control logic, in

order to determine when the next value should be loaded. Again, these states seem simple now but will simplify the design process when adding more parallel hardware.

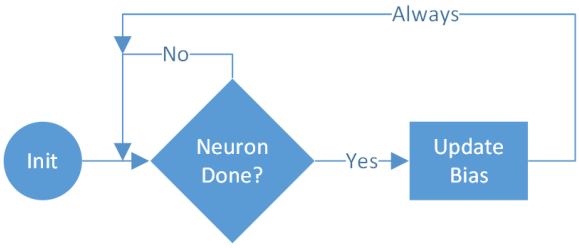

Figure 3.12: The b_mem Flow Diagram

The b_mem module will load new values for each time that the neuron is complete. This is because there is one new bias value for each software neuron.

3.4.4 The Network Module

The network block contains the a_loader_store block, which contains the Input/Output RAM from Figure 3.6, the network_control block, which contains the Structure ROM, and the bw_hardware_neuron, which contains the Bias ROM, Weight ROM and SIHN. A more detailed diagram is located in Appendix E, Figure E.1. The neuron waits for data from the bias, weight, and input memory, and begins calculating the first dot-product. The neuron handshakes a CTS signal to let the memory modules know when the input, weight, and bias values can be changed. After the neuron is done, the network_control block updates it's state based on the remaining number of software neurons in the layer. If the number of software neurons in the layer is complete, then the next layer information is loaded from the structure ROM. If the structure ROM contains a delimiter then the network calculation will be complete. This is shown in the Figure 3.14 flow diagram.

From Figure 3.13, the a_loader_store RTS and CTS are connected from network_control and to the a_loader store block. The a_ct and p_ct values are connected into the a_loader_store block from the network_control block. This means that the

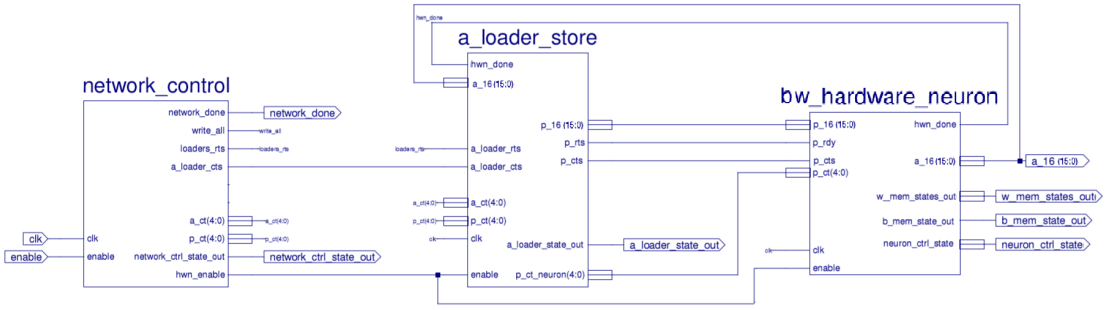

Figure 3.13: Network Schematic

bw_hardware_neuron block does not see the p_ct from the network_control, but rather from the a_loader_store module. This separation is to ensure that a good p_ct value is held for the neuron during computation, allowing the network_control block to change at any time. The CTS and RTS signals between the a_loader_store and the bw_hardware_neuron block handshake the layer's input values from the Input/Output RAM located within the a_loader_store module.

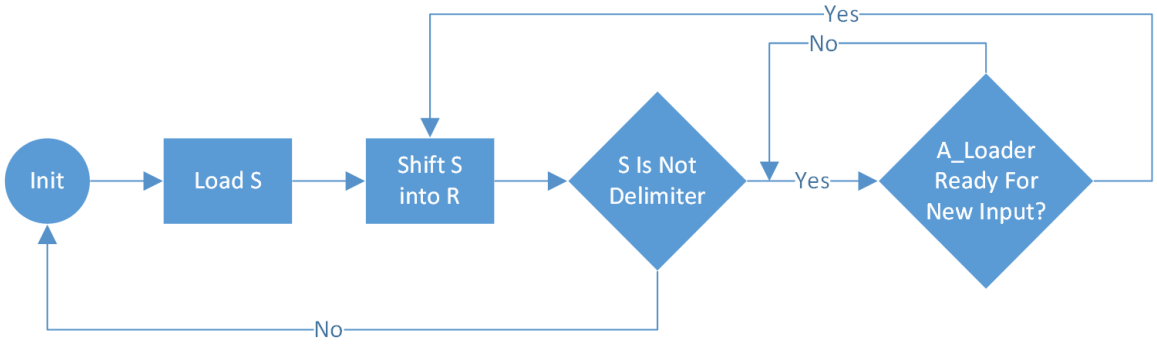

Figure 3.14: The Network Control Flow Diagram

The Network Control Block traverses the states in the flow diagram in Figure 3.14. Initialization resets the entire network. The first S value, the layer size, needs to be loaded because the Structure ROM is used as a shift register. Shifting S into R, the input vector size, causes two values to be loaded into the a_ct and p_ct wires. This occurs each time one layer has been completed and the next layer is started.

Table 3.5: Network Schematic Signals

Signal	Description
enable	Enables only the network_control.
clk	Clock for the modules.
network_done	Set high when software network has been calculated.
loaders_rts	Set high when a_ct and p_ct requested to change.
a_loader_cts	Set high when a_ct and p_ct safe to change.
a_ct	Software neurons for layer.
p_ct	Software inputs for layer.
network_ctrl_state_out	State of network_control for test bench.
hwn_enable	Enables the calculation of the hardware layer.
p_16	The input value.
p_rts	Set high when new input value requested to change.
p_cts	Set high when new input value safe to change.
a_loader_state_out	The state of a_loader_store for test bench.
p_ct_neuron	The number of software inputs for the layer.

The a_ct wire stands for output count, while the p_ct wire stands for input count. Therefore the R, total number of inputs for a layer, is stored into p_ct, while the S, total number of neurons for a layer, is stored into a_ct. A comparison is made to see if the new S value is the delimiter. If it is not the delimiter, then the block waits for the a_loader_store block to signal that it is ready for new values. If it is ready, then the old S is shifted into R, and the new S is loaded and compared to the delimiter.

The a_loader_store block uses a state machine to control when it loads and stores inputs. In order to pass inputs into the neuron, it handshakes RTS and CTS pins in order to load the number of inputs, R, and the number of outputs, S. These are stored in the variables p_ct, and a_ct, respectively. The values are loaded from

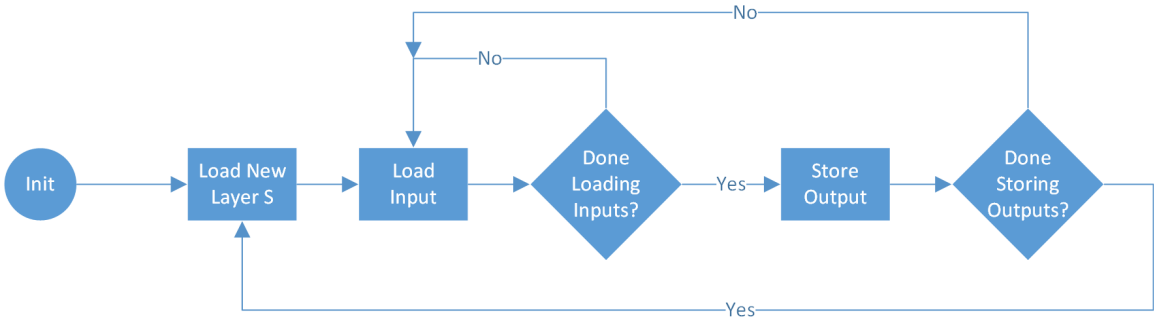

Figure 3.15: The A Loader Store Flow Diagram

the network_control block. Then the a_loader_store block begins loading inputs to the neuron. The a_loader_store block handshakes a different set of RTS and CTS pins from the neuron to know when to update to a new input. If it is finished loading inputs, then it will store outputs, and then check if more neurons need to be calculated. If more software neurons exist, then the a_loader_store block will begin loading the same layer inputs again. If it is finished storing the software neuron outputs, then it will signal to the network_control block to begin the next layer.

3.5 Clock Cycle Requirements

An equation to predict the time for the computation of a software network can help to predict speed up when increasing the amount of parallelism in the hardware. Although the multipliers and adders could be implemented combinatorially without any delay, the maximum delay was used for each module in order to decrease the delay paths within the multipliers and adders. Because behavioral simulation was used, the timing of the combinational logic would not be visible. The variables in Table 3.15 explain the values that are needed to characterize the cycles required for this hardware structure. Equation 3.4 explains the equations for calculating the number of cycles per layer, where $cycles^1$ is the number of cycles required to compute the output of the first layer, and the $cycles^n$ is the number of cycles required to compute the output of the

n^{th} layer.

Table 3.6: Cycle Variables

Variable	Description
R^n	Number of inputs to the layer n
S^n	Number of neurons of the layer n
C_A	Cycles required for one adder to complete
C_M	Cycles required for one multiplier to complete
C_{LNC}	Cycles of the neuron control logic per input
C_{TF}	Cycles required for the transfer function
C_{SA}	Cycles required for storing a single output
C_{LS}	Cycles required to load a new structure

$$Cycles^1 = R^1 S^1 (C_M + C_A + C_{LNC}) + S^1 (C_{TF} + C_{SA}) + C_{LS} \tag{3.4}$$

$$Cycles^n = S^{n-1} S^n (C_M + C_A + C_{LNC}) + S^n (C_{TF} + C_{SA}) + C_{LS} \tag{3.5}$$

Equation 3.4 says that for each layer, the number of cycles for that layer can be computed by by summing each stage of the neuron. The first stage is inside the neuron. Each input will need to be multiplied and then added with the bias or the partial dot product. This is done for as many times as there are outputs, so it is multiplied by the number of neurons. The transfer function is only used once per output, and the same for the number of times the values are stored. Finally, the next layer structure needs to be loaded.

3.6 Slice Requirements

The determination of the number of gates for the SIHN case is simple, because the sizes of the hardware do not change. This means that the number of slices per module

are constant, because we are keeping the number of neurons, and the number of inputs per neuron, constant. Table 3.7 shows the descriptions of the variables, and Equation 3.6 shows the slices being added together.

Table 3.7: Slice Variables

Variable	Description
S_M	Slices per multiplier
S_A	Slices per adder
S_{TF}	Slices per transfer function
S_W	Slices per weight ROM
S_{WL}	Slices per weight logic
S_{NC}	Slices per neuron control block
S_B	Slices per bias ROM
S_{BL}	Slices per bias logic
S_{PA}	Slices per a_loader_store block
S_{NET}	Slices per network_control block

$$Slices = S_M + S_A + S_{TF} + S_W + S_{WL} + S_{NC} + S_B + S_{BL} + S_{PA} + S_{NET} \quad (3.6)$$

From the equation, simply add the size of each block in order to find the total size. This will change when the hardware neurons, or the number of inputs, are increased. Another issue will be the wires connecting the modules, which will cause discrepancies in the measurements later on.

3.7 Simulation Testing and Verification in Vivado

In order to verify that the device works as expected, a test bench can be used to simulate inputs into the network. Because the network works primarily off of ROMs,

the ROMs can be generated with preset values using a coefficient file. The last letters of a coefficient file, the codec, is coe. These files provide Vivado with the values that the ROMs should be set to upon hardware creation. Because the software network is described using the coe files, the network will only need to be enabled using the enable pin for the network to start functioning and outputting values. Matlab can be used to create these coe files and can be used to run calculations on the values stored in the ROMs to compare the results with the Vivado simulation.

3.7.1 Testing the Output Values of the Network

Using the *rand* function of Matlab, and then a conversion function, random values can be generated for a coe file. The *rand* function attempts to generate numbers with an equal probability between 0 and 1. This can be mapped to the range -1 to 1 by multiplying by 2 and then subtracting 1. Because Matlab uses double precision, these will need to be converted down to 16-bit floating precision in order to compare the values that the network outputs.

Multiple test cases have been used to verify the network, but only one test case will be used in this document. The memory cores that are generated use the same structure as specified in Figure 3.6. The test case covered will be a software network consisting of 4 input values, 10 hidden neurons, and then 1 output neuron. This is specified by changing the values in the Structure ROM. A table of the values in the Structure ROM is given in Table 3.8. Notice that the values in the structure ROM are offset by -1. So an S^2 value of 0 means that there is one neuron in layer 2. Also, take note of the delimiter in the 4th memory address. The memory was generated in order to test a maximum case of 32 inputs or neurons, which is why the delimiter is 5 bits, because the maximum value represented in 5 bits is 31. These are fixed-point integer values, as opposed to the floating-point values in the Weight, Bias and Input/Output ROMs and RAM. Different cases will be used in later sections to show the increase

in speed of the network. However, this case will be used to verify that the output values are the same or similar to predicted Matlab results.

Table 3.8: Structure ROM

R^1	S^1	S^2	$Delim$
03	09	00	$1F$

Using Equation 3.7, the total number of weights can be calculated for the network. This means that the total number of weights for this network will be 50. The weights for this case can be found in Table 3.11. Because there will be one bias value per neuron, there will be 11 bias values. The bias values can be found in Table 3.10. And finally, the Inputs/Output RAM will need to have the first four values initialized, found in Table 3.9, because that is the number of inputs to the network, as specified by the structure ROM.

$$NumberofWeights = R^1S^1 + S^1S^2 \tag{3.7}$$

Table 3.9: Input/Output RAM

4800	4400	4000	$3C00$

Table 3.10: Bias ROM

3668	$358B$	$AEAC$	$B54A$	$AC73$	$AD5C$	3352	$ADFC$	$373A$	3697	3738

Values that should be expected from the network can be calculated using Matlab. Comparing results from more accurate 64 bit precision calculations and an accurate

Table 3.11: Weight ROM

$A831$	$B191$	$B6D0$	$AD15$	$B55C$	3677	$B7F0$	$B5A4$	2459	$356D$	...
3787	$B283$	$B205$	$B2EA$	3622	$302A$	$21E7$	3470	$B412$	$37B8$	...
3575	$2B76$	3652	$B73A$	3153	$37BA$	$31B5$	$AE75$	$B4E6$	$300B$	...
$B6BE$	$B6EB$	$355B$	$B47E$	3590	$2D73$	$2C35$	$365F$	$B68B$	$B519$	...
$B432$	$B6E5$	$B7F6$	$B42C$	3433	3573	$A583$	$B22C$	$B035$	$B608$	

transfer function can show the error of the network. Because the hardware is in 16-bit floating-point precision, instead of 64 bit floating-point, some discrepancies between ideal and hardware calculations can be expected. The hardware transfer function uses a lookup table and therefore has non-continuous values, unlike the nearly continuous values Matlab can generate. Calculating the first output, the partial sum value, n, will be initialized to the bias. Each input will be multiplied by a corresponding weight, and added to n. Because this is a single-input neuron, n will need to accumulate the sum over multiple cycles. Finally, the n value will be passed through the transfer function lookup table. Table 3.12 shows the value of n at each step of the process and the final resulting value when calculated in Matlab.

Comparing the values calculated in Matlab from Table 3.12, they can be compared to the hardware values found in Table 3.13. Notice that because Matlab converts the values to 64 bit precision, there is a discrepancy, which is underlined, between the partial dot-product calculation in the column under Input 3. The n value's mantissa is incremented by 1 compared to what the Matlab calculation expects. This carries through to the final value. But because of the error in the transfer function lookup table, the final value is the same. The error in the transfer function table could result in final values being much more different than the expected results in Matlab, causing a large error to propagate through the network.

Table 3.12: Calculation of the First Neuron in Matlab

	Varible	Init	Input 1	Input 2	Input 3	Input 4	Output
	b	3668					
Inputs	p		4800	4400	4000	$3C00$	
	w		$A831$	$B191$	$B6D0$	$AD15$	
	wp		$B431$	$B991$	$BAD0$	$AD15$	
Outputs	n	3668	$306E$	$B875$	$BDA2$	$BDF3$	
	a						BB38

Table 3.13: Calculation of the First Neuron in Behavioral Simulation

	Varible	Init	Input 1	Input 2	Input 3	Input 4	Output
	b	3668					
Inputs	p		4800	4400	4000	$3C00$	
	w		$A831$	$B191$	$B6D0$	$AD15$	
	wp		$B431$	$B991$	$BAD0$	$AD15$	
Outputs	n	3668	$306E$	$\underline{B876}$	$\underline{BDA3}$	$\underline{BDF4}$	
	a						BB38

After comparing the values for the first neuron, each a output can be calculated in Matlab in order to compare the results to the resulting values from the behavioral simulation. Table 3.14 shows the expected Matlab results side by side with the results of the behavioral simulation. Differences between the calculations are underlined.

Over longer calculations, and larger numbers of layers, and because of the non-linear nature of neural networks, this error could cause problems in some systems. Another approach could be used, like using fixed-point, and also increasing the resolution of the transfer function lookup table in order to decrease the error through the network. Notice that the output on the second layer is no longer only one bit of

Table 3.14: Comparison of Layer Outputs Between Matlab and Simulation

Variable	a_1^1	a_2^1	a_3^1	a_4^1	a_5^1	a_6^1
a_{Matlab}	$BB38$	$BBBE$	$3BC6$	$BB93$	$3A6B$	$3BF9$
$a_{Simulation}$	$BB38$	$\underline{BBBF}$	$3BC6$	$BB93$	$\underline{3A6A}$	$3BF9$
Variable	a_7^1	a_8^1	a_9^1	a_{10}^1	a_1^2	
a_{Matlab}	$3BFD$	$BBFA$	$3BF0$	$3B11$	$3A76$	
$a_{Simulation}$	$3BFD$	$BBFA$	$3BF0$	$\underline{3B12}$	$\underline{3A7B}$	

error, but rather four. With such a small bit range, this error may be significant. The accuracy of the system should be analyzed in greater detail in order to support larger networks, even more for recurrent networks, as the error on a recurrent network could cause the network to become unbounded, forever incrementing values until saturation. However, this is not the focus of this document as the primary goal is to design a scalable network.

3.7.2 Cycle Analysis

Other than the values the network outputs, other important characteristics are the size of the hardware and the time it takes to compute outputs. Using values from the previous sections, we can analyze if the equations are effective at describing the network.

Plugging in the values found in Table 3.15 into Equation 3.4, the number of cycles required for each layer and the total number of cycles are found in Table 3.16. This matches what measured by the simulation. Notice that the logic required for controlling the neuron is significant compared to the number of cycles that the floating-point adder and multiplier consume.

Each set of cycles was measured through simulation. In order to simplify the measurement and calculation, some of the cycles between modules were lumped

Table 3.15: Cycle Variables

Variable	Value	Description
R^1	4	Number of inputs to the layer 1
S^1	10	Number of neurons of the layer 1
S^2	1	Number of neurons of the layer 2
C_A	8	Cycles required for one adder to complete
C_M	6	Cycles required for one multiplier to complete
C_{LNC}	9	Cycles of the neuron control logic per input
C_{TF}	3	Cycles required for the transfer function
C_{SA}	5	Cycles required for storing a single output
C_{LS}	1	Cycles required to load a new structure

Table 3.16: Cycles Required for Each Layer

Layer	Calculated Cycles	Measured Cycles
Layer 1	1001	1001
Layer 2	239	239
Total	1240	1240

into the logic cycles. For example, the C_{LNC} variable includes delays between the a_loader_module, and the w_mem module. This is because the neuron handshakes backwards when to load new values, but the modules take a cycle to register the value, and then need to load values and handshake to the neuron that it is ready.

3.7.3 Slice Analysis

For the single neuron case, the size of the structure is the sum of each module. They are simply added together and put in Table 3.17. The changes in these values will be critical in later sections when the size of the hardware is changing. It will be

interesting to see the non-linear aspects of the hardware in those cases. However, in this case, only one size of hardware is considered, and therefore there is nothing interesting to see. The Weight ROM is much larger than needed for this case, and so uses much more LUTs than it would require for 50 values. This is because it is cumbersome to regenerate the Weight ROM for each new network size. With a maximum of 32 inputs and outputs for the first two layers, and then 5 outputs for the output layer, the total number of weights possible would be 5120. A script could be used to generate the cores automatically to simplify the process. Alternatively, memory cores could be created to simplify scalability.

Table 3.17: Slice Requirements

Variable	Slice LUTs	Slice Registers	Description
S_M	74	110	Slices per multiplier
S_A	178	251	Slices per adder
S_{TF}	77	37	Slices per transfer function
S_W	1399	16	Slices per Weight ROM
S_{WL}	30	22	Slices per weight logic
S_{NC}	126	54	Slices per neuron control block
S_B	64	16	Slices per Bias ROM
S_{BL}	18	16	Slices per bias logic
S_{PA}	179	148	Slices per a_loader_store block
S_{NET}	49	31	Slices per network_control block
Total	2194	701	

3.8 Single-Input Hardware Neuron Summary

After testing and verification, along with the timing analysis, the SIHN case can be used to compare against the other hardware size cases. This means that by adding inputs, or by adding neurons, the hardware can be analyzed for decrease in cycles for a given network as parallelism is exploited, and the increase in hardware resources as the speed increases. Using Vivado with behavioral tests, the outputs of the network can be compared with Matlab in order to see the error caused by hardware limitation. In the next chapters, similar tests can be used to verify that the modified hardware still provides the same outputs.

CHAPTER 4

Multiple Single-Input Hardware Neurons

Increasing the number of hardware neurons for a hardware layer increases the computation speed of a hardware neural network by taking advantage of parallelism in the ANN algorithm. This chapter will, first, outline the general change in structure to the hardware, and then will verify correctness and analyze computation time and size requirements. A new variable for the number of hardware neurons is introduced, the variable HWN. The key concept for increasing the number of hardware neurons, is that the single Bias ROM and single Weight ROM, from Figure 3.6, will need to be split into parallel memory modules based on the number of hardware neurons in the network. The parallel memory modules maximize the amount of parallelism in the network. In order to load hardware neurons with their portion of the weight and bias matrices, special loader modules are developed. More justification and a detailed discussion of these modules is given in a later section. Finally, the chapter will analyze the increase of speed and size of the hardware network by comparing the results to the size and speed of the hardware network introduced in Chapter 3.

4.1 Overview of Hardware Implementation of Multiple Single-Input Hardware Neurons

By adding in multiple single-input hardware neurons to the hardware network, a software layer can be computed in less time. However, there will only be an increase in speed during certain conditions. In order to provide parallel computation, each hardware neuron will require each of the components that make up a neuron, and

then an additional Bias ROM and Weight ROM. The diagram in Figure 4.1 shows a specific case of the scalable multiple single-input hardware neuron (MSIHN) neural network where there are two hardware neurons. The Weight and Bias ROM of the first neuron contain the odd rows of the bias vector and weight matrix from the equation in Figure 3.5. The Weight and Bias ROMs corresponding with the second hardware neuron use the even rows of the weight matrix and bias. This will make the logic for the hardware more simple and will reduce the number of connections between modules. Because the weight matrix is the size $S \times R$ and the number of software neurons to be calculated is S, the number of Hardware Neurons will correspond to the value of S. This means the Weight ROMs will need to be split across the weight matrix rows as the number of hardware neurons changes.

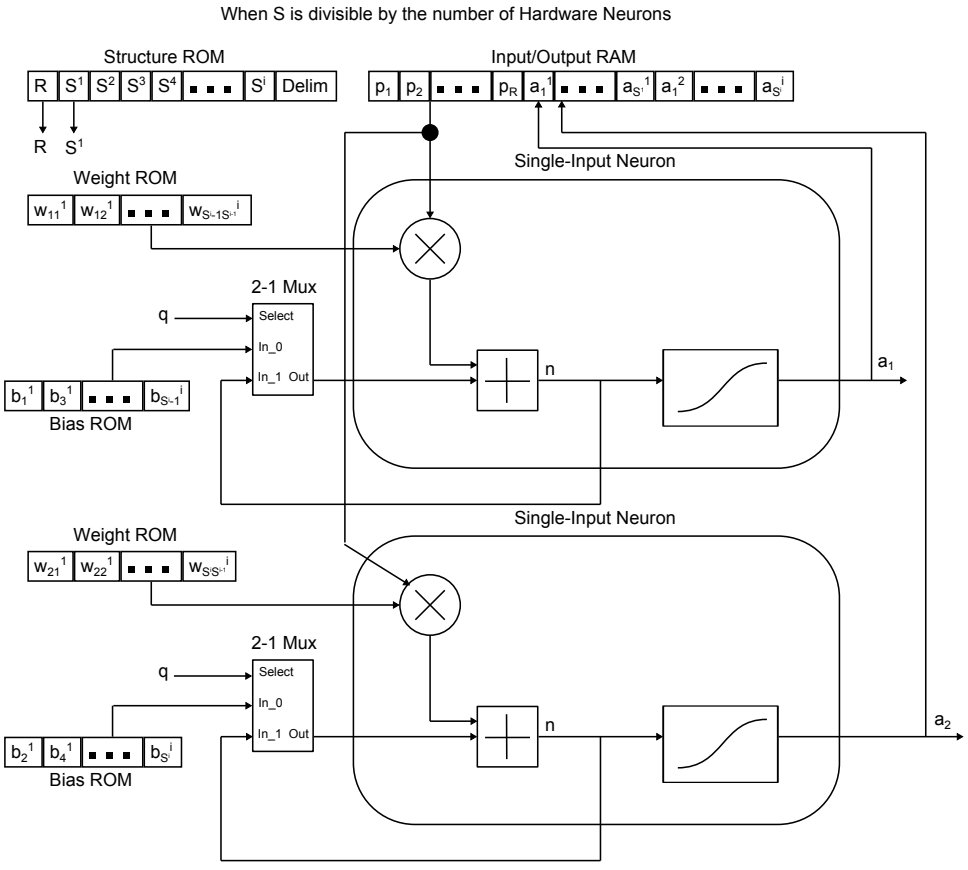

Figure 4.1: General Multiple Single-Input Hardware Neurons

This hardware has inefficiencies when the desired software layer is not divisible

by the number of hardware neurons. As shown in Figure 4.1, the values for b_1^1 are in the same Bias ROM address as the b_2^1 value. A single address will be used to control these ROMs in order to reduce signals, so, if the variable for b_2^1 does not exist for the software network, then a zero should be stored in that memory location. This is wasteful, so the memory required could be reduced in a later work by changing the logic that controls how the Weight and Bias ROMs are fed into the neuron. Perhaps by using an address for each ROM, but only incrementing when the condition for loading the next value, and therefore incrementing the address, is met.

Another important aspect of this diagram is that there are multiple outputs being fed back from the hardware layer to the Input/Output RAM. The Input/Output RAM is only a single input storage device. This means that additional logic will be required in order to save the multiple values into their destination addresses. The Input/Output RAM will need to serially increment the address for each hardware neuron until the number of hardware neurons are loaded, then after checking if the layer was completed, load a new input into the hardware layer starting from a new address. If the software layer has not finished computation, then the input values for the current layer will need to be loaded by resetting the the address counter variables.

4.2 Implementation of Multiple Single-Input Hardware Neurons in VHDL

One of the goals of the VHDL code is to test different ranges of sizes for software and hardware neurons efficiently. Because the contents of the Weight ROMs and Bias ROMs will change based on the number of hardware neurons in a hardware layer, ways to mitigate compile time should be considered. At a fundamental level, the memory modules could be optimized to contain only necessary values and be the smallest size for the desired software network. A scripting tool would need to be used to generate the core's coe files and IP core parameters for each test, as described

in the previous chapter. Alternatively, the coe files could be made for each case, but this is not scalable, because each test case would require a new coe for each Weight and Bias ROM. At 3-6 minutes of generation for each generation, creating each individual memory core for each case could take a significant period of time. For example, to test 10 hardware neurons, each coe file would need to be created, then each core generated, resulting in 10 generations for Weight ROMs, 10 generations for Bias ROMs, and a final regeneration for the project, resulting in 21 generation cycles. For only one test case, this process would require, at minimum, an hour of time. This is an unreasonable amount of required generation time for gathering information about the scalable network. In order to avoid the regeneration process when changing the number of hardware neurons per hardware layer, the memory modules can use additional logic to load and store values from a Weight ROM and a Bias ROM that match the specifications in Chapter 3 to allow for only a single regeneration cycle of the entire project after changing the HWN variable. The modules that contain the Weight ROM and Bias ROM that match the previous chapter's specification will be instantiated inside the loader modules and be called the top ROMs. The memory modules that are loaded with values and are duplicated HWN number of times will be called the nested ROMs.

Because the hardware neuron is still a single-input neuron, as in Chapter 3, there are no changes to be made to the neuron schematic or control flow chart. So, when building up in abstraction levels from the hardware neuron to the total hardware network, the first change can be seen when adding the Weight ROM to the hardware neuron.

4.2.1 Weight ROM Changes

The w_mem module, shown in Figure 4.2, displays the neuron connected with the w_mem module. The w_mem module contains the Weight ROM from the general

diagram in Figure 4.1. Compared to the schematic from Chapter 3 Figure 3.9, this schematic has new RTS, CTS, w_write, and w_data signals in preparation to clock in data from modules above it. These signals follow the same scheme as described in Appendix B, loading a new value when RTS is high, then toggling CTS. This module is encapsulated into a module called the w_neuron. A VHDL generate statement can be used to copy the module multiple times, providing abstraction that makes the VHDL description easier to understand in code. At this level, the other signals are the same as for the single input neuron from the last chapter. However, the states of the w_mem module will need to be changed.

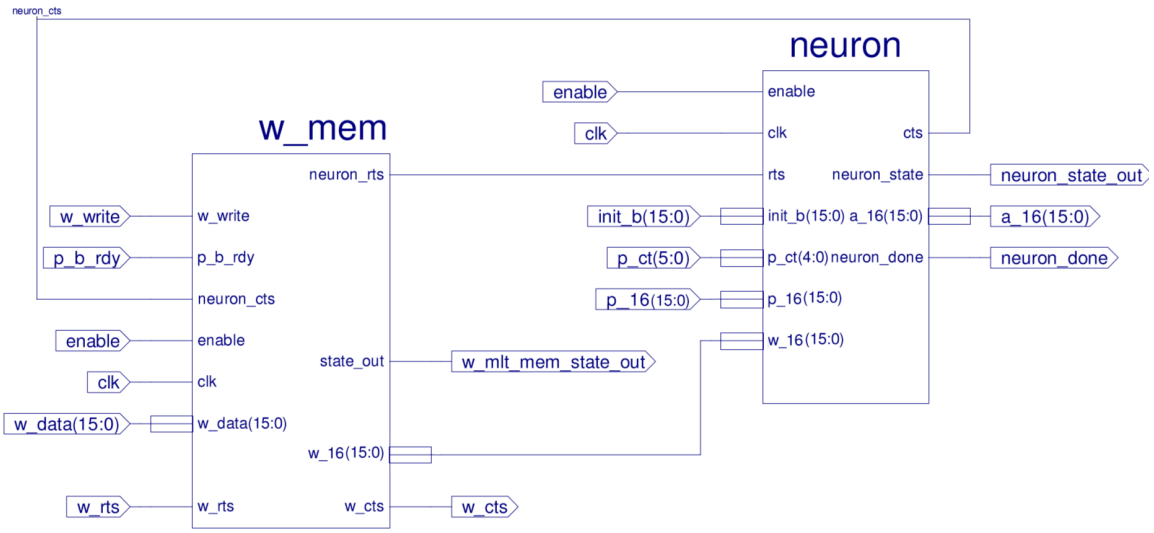

Figure 4.2: Adding the Weight ROM Schematic

The w_mem module needs new states to accept the values from the w_loader module, described in a later section at a higher level of abstraction. The new states, which can be seen in Figure 4.3, check if a new value is present and then store the value and increment the address. The w_write pin is what switches the module between writing new values and loading old values. When the module is enabled and the w_write pin is high, values will be clocked into the Weight ROM until the module is reset by setting the enable signal and w_write signals low. When clocking in the

Table 4.1: Weight ROM Schematic Signals

Signal	Description
enable	Enables both the w_mem and neuron.
clk	The clock for the modules.
p_b_rdy	Set high when input and bias are ready.
neuron_rts	Set high when inputs, bias, and weights are ready.
neuron_cts	Set high when neuron allows values to change.
w_16	The weight value.
w_mem_state_out	The state of the w_mem for the test bench.
w_write	Set high when the Weight ROMs are being written to.
w_data	The data being stored into the Weight ROMs.
w_rts	Set high when w_data is requested to change.
w_cts	Set high when w_data is safe to change.

data, the address is always increased by 1 for each data stored. The data loaded from the module above will be shown to each hardware neuron, but only the neuron that is supposed to receive the value will receive an RTS signal that is high. This is how each hardware neuron can be selected by an upper module. Because there is an infinite loop for the storage of values, the module will need to be disabled by setting the enable pin low, then back high in order to start loading values. Moving up a layer in abstraction, we can analyze the changes when adding the Bias ROM.

4.2.2 Bias ROM Changes

Shown in Figure 4.4, the b_mem module changes from Chapter 3 Figure 3.11, to the b_mhwn_mem module. This is because the Bias ROM inside the module is duplicated HWN times. Notice also that there are now RTS and CTS pins that talk to the loader modules, along with a data line and the b_write signal to accept the

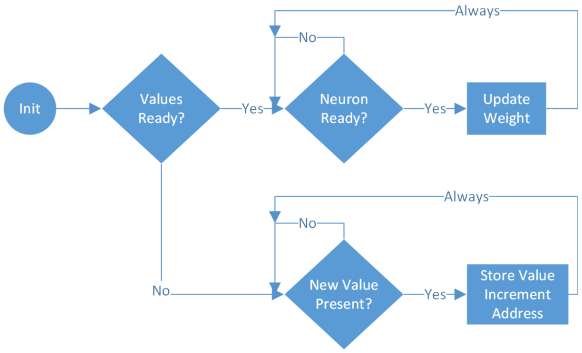

Figure 4.3: The w_mem Flow Diagram

loader module data. The goal of duplicating the Bias ROM in the module is to produce the b_hwn_16 signal, which is attached from the b_mhwn_mem module to the multi_hardware_neuron.

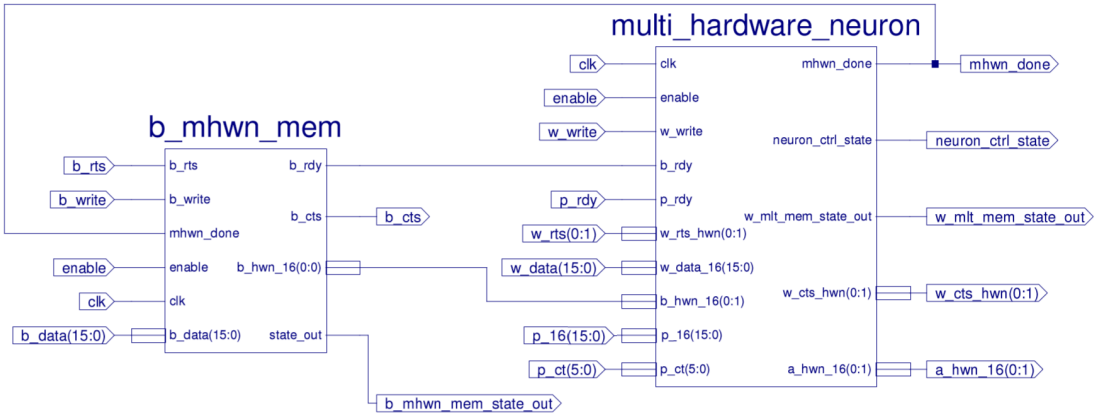

Figure 4.4: Adding the Bias ROM Schematic

The schematic in Figure 4.4 also shows the multi_hardware_neuron module. This block contains a VHDL generate statement that duplicates the w_neuron HWN number of times. The RTS and CTS signals going into the w_mem module are an HWN size array of signals to connect all of the w_neurons that were duplicated. This also means that this module after encapsulation, called the mhwn_b_mem module, will have an output that is an array of 16-bit wires of length HWN.

The b_mhwn_mem module needs to have additional states added to handle multi-

Table 4.2: Bias ROM Schematic Signals

Signal	Description
enable	Enables b_mem and w_hardware_neuron.
clk	Clock for the modules.
p_rdy	Set high when the input value is ready.
b_rdy	Set high when the bias value is ready.
b_16	The bias value.
hwn_done	Set high when the output of the neuron is ready.
b_rts	Set high when b_data is requested to change.
b_cts	Set high when b_data is safe to change.
b_write	Set high when values are being written to Bias ROMs.
b_data	The value being stored into the Bias ROMs.

ple numbers of neurons. These states are only for loading values because the outputing values are still exactly same as the prior case. As seen in Figure 4.5, a comparison is made to test whether the values have been initialized. This is done with the b_write signal shown in Figure 4.4. If the signal is high, then the values will be written to the memory, if the signal is low, then values will be loaded. Notice that the states do not go back to the initialization state. This means that the module will need to be disabled before the values can be loaded into the neurons. A key difference between the w_mem module and the b_mhwn_mem module is that when loading values, a HWN counter will keep track of which nested Bias ROM is supposed to receive the new data value. Because the data is loaded serially, only one Bias ROM is ever enabled at one time. The b_mhwn_mem module alternates between the ROMs using the counter, and then resets to the first ROM when the HWN counter overflows.

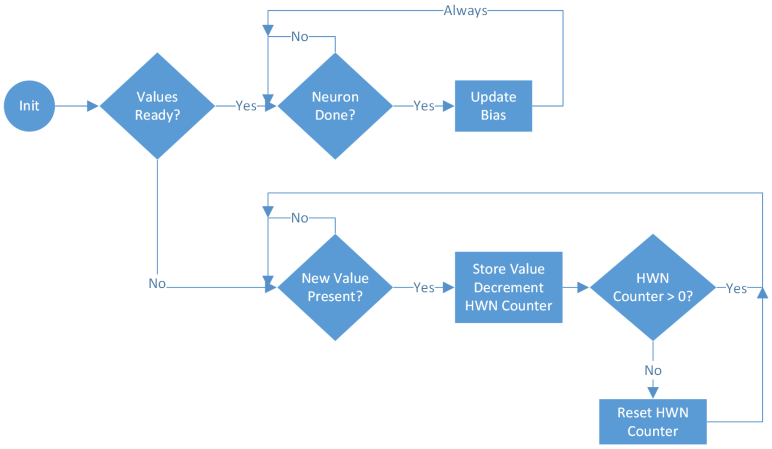

Figure 4.5: The b_mem Flow Diagram

4.2.3 Changes in the Network Block

Key differences to the Network Block, see Figure 3.13 and Figure 4.6, will be that there are new loader modules that interface with a module named the mhwn_b_mem module. A more detailed schematic is located in Appendix E, Figure E.2. In order to handle an initialization step for the network, the network_control module will need to modified. As shown in Figure 4.6, multiple signals must be added between the network_control module and the b_loader and the w_loader. The main changes in signals for this module are the CTS signals that are sent backwards from the loader modules. These CTS signals must be high in order for the a_ct and p_ct values to change. The write_all signal acts as an enable pin for the b_loader and w_loader modules. So, when the values from the top memory modules are being sent to the nested memory modules, the write_all signals will need to be set high. The b_loader and w_loader modules each connect RTS and CTS pins that follow the communication scheme outlined in Appendix B. One of the key changes in the schematic is that the a_hwn_16 signal is now an array of 16 bit signals. This is because there will be HWN 16-bit signals. The naming scheme was chosen because the variable a represents the output of the neurons from Equation 3.1, and there are HWN 16-bit signals, creating the name a_hwn_16. Using an array that can be defined in code allows for the number

of outputs for this one block to be easily scalable.

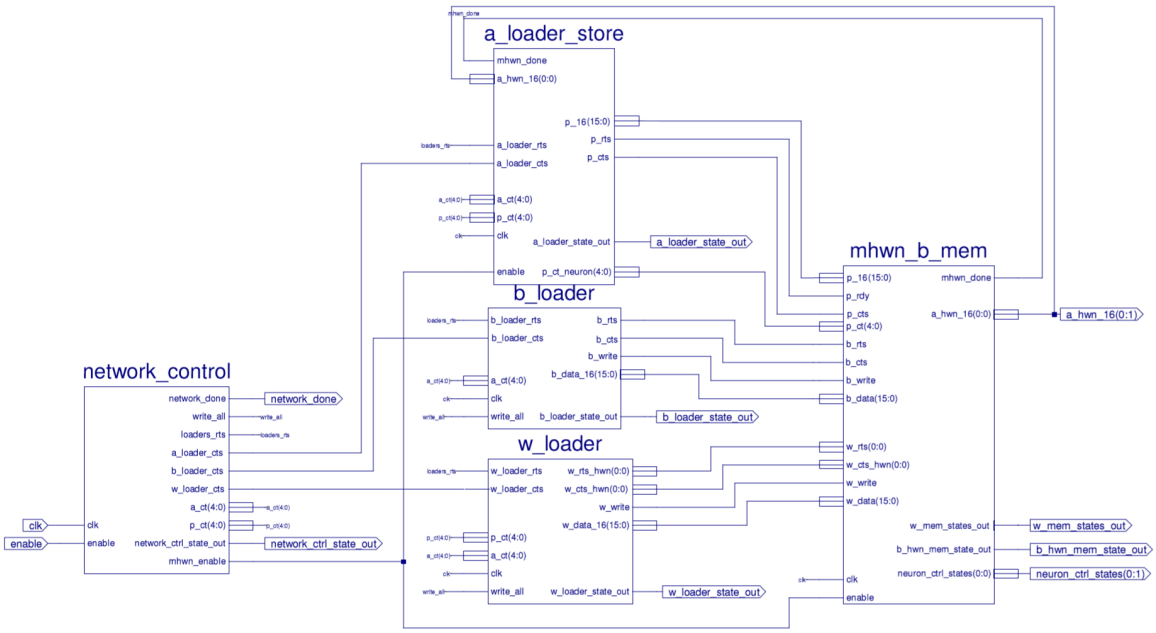

Figure 4.6: Neural Network Multiple Single-Input Hardware Neurons Schematic

A set of new states, shown in Figure 4.7, will need to be added to handle the initialization of values into the nested memory modules. The change is a comparison on an internal flag to see if the values have been initialized. If they have not, then the values are initialized, otherwise the network is calculated as normal. Notice that during these states, the values are loaded for the loader only once, and then the network repeats the computation.

The top ROMs are stored in the w_loader and b_loader modules, as seen in Figure 4.6. These modules communicate using an RTS and CTS scheme, as seen in Appendix B. The primary goal of the loaders is to filter the values being sent to the nested memory modules, setting the values to zero for hardware neurons that do not need to be calculated. A key difference between the b_loader and the w_loader is that the b_loader only has one set of RTS and CTS lines attached to the mhwn_b_mem block, while the w_loader has an array the size of the number of hardware neurons. The number of abstraction levels causes the w_mem blocks to be copied the same number

Table 4.3: Network Schematic Signals

Signal	Description
enable	Enables only the network_control.
clk	Clock for the modules.
network_done	Set high when software network has been calculated.
loaders_rts	Set high when a_ct and p_ct requested to change.
a_loader_cts	Set high when a_ct and p_ct safe to change.
a_ct	Software neurons for layer.
p_ct	Software inputs for layer.
network_ctrl_state_out	State of network_control for test bench.
hwn_enable	Enables the calculation of the hardware layer.
p_16	The input value.
p_rts	Set high when new input value requested to change.
p_cts	Set high when new input value safe to change.
a_loader_state_out	The state of a_loader_store for test bench.
p_ct_neuron	The number of software inputs for the layer.

of times as the neurons, while the b_mem block is only created once.

The b_loader is more simple than the w_loader, so it is easier to look at the states of the b_loader first. From the b_loader flow diagram in Figure 4.8, it can be seen that there are two counters which load each Bias ROM with the correct corresponding bias or zero value. First, the current layer's software neuron count is loaded, and then another counter is set to the value of HWN, the number of hardware neurons. Notice that the loop will always repeat for the number of hardware neurons, even if there are no longer any remaining software neurons for the layer. This will occur when the number of software neurons is not divisible by the number of hardware neurons. Because of this, zero values are loaded into the Bias ROMs.

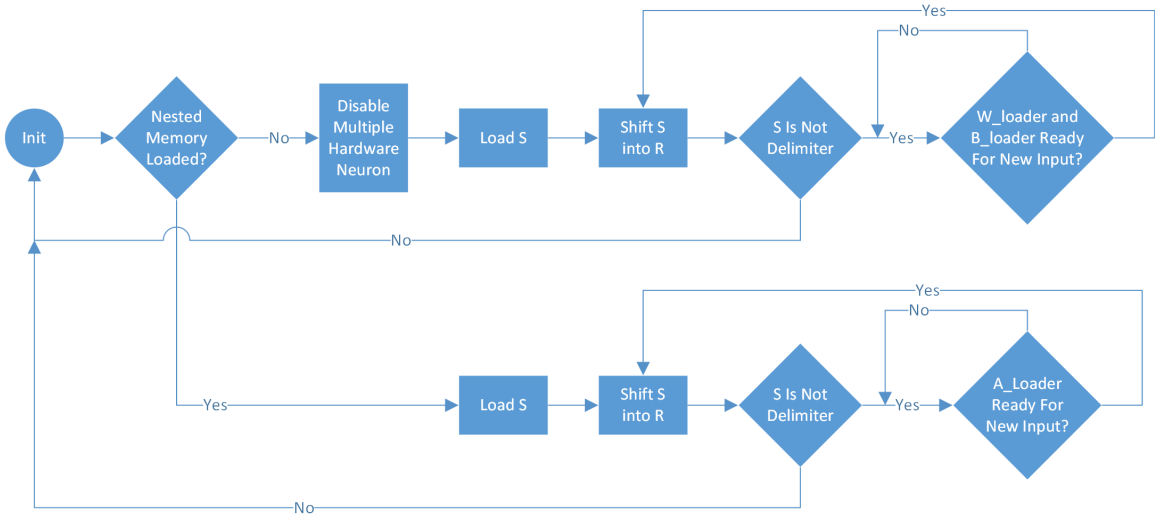

Figure 4.7: Flow Chart for Network Control Multiple Hardware Neuron

Looking at Figure 4.9, the w_loader flow diagram performs a similar operation as the b_loader but has states for loading the number of inputs for the current software layer. The idea is that if a neuron needs to be skipped, then all of the inputs will need to be set to zero. So, if all software neurons have been loaded for this layer, then it will send p_ct number of zeros into each remaining hardware neuron's Weight ROM until all Weight ROMs of that software layer have zeros stored for nonexistent variables in the software network. However, if the counter for the number of HWNs has been reached, the module will continue storing values by reseting the HWN counter.

Compared to the a_loader_store module from Chapter 3 (see Figure 3.13), the a_loader_store module of this chapter will require some modification in order to handle multiple outputs. This is done by using a counter which is set to the number of HWNs, and then storing HWN values, but only until the number of outputs for the current software layer have been stored. It will not store values from the hardware neurons that do not contain useful data values. Notice in Figure 4.10 that there is a conditional block added to the flow diagram compared to the a_loader_store flow diagram in the single-input single-hardware-neuron case. If the a_loader_store module has completed the storage of the hardware neurons' outputs, then the module will begin loading new

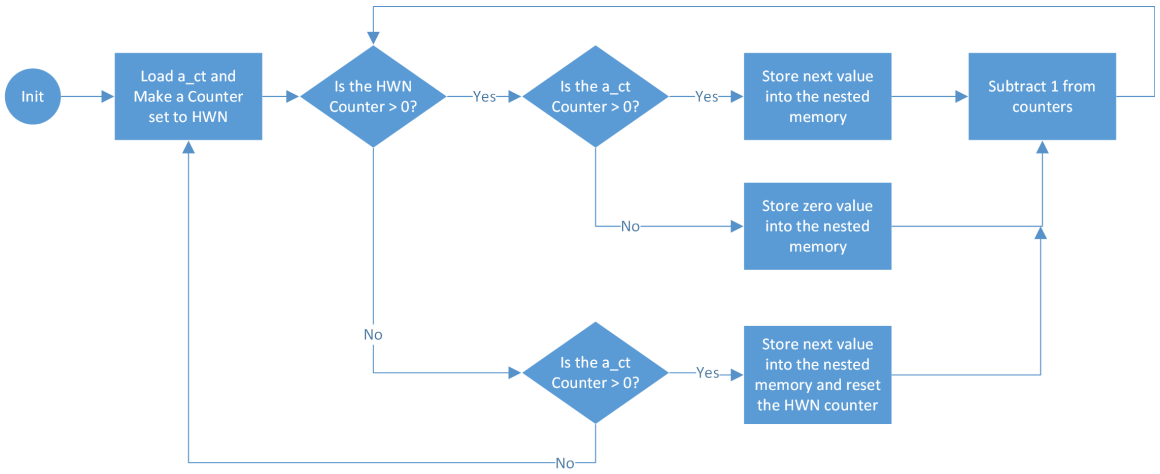

Figure 4.8: The b_loader Flow Diagram

inputs to calculate another set of neurons. However, if the calculation of software neurons are complete for that layer, then the module will load a new S value. This is significant, because it means that not all hardware neurons will be stored when the software neurons are not divisible by the hardware neurons.

4.3 Clock Cycle Requirements

During the computation of a software layer, the hardware layer will only be able to produce the number of hardware neuron outputs that exist in the hardware layer, the variable HWN. This means that if the number of software neurons desired, the variable S, is not divisible by the number of hardware neurons, there will be some inefficiency in the calculation. In order to keep the hardware logic simple, the hardware network can always calculate all hardware neurons. This means that the values of the weights and biases for the wasted software neurons can be set to zero, and then the outputs of the hardware neurons can be ignored when saving values. As shown in Equation 4.1 the number of iterations over the hardware layer can be calculated by taking the ceiling of the number of software neurons in the desired software layer and dividing by the number of hardware neurons in the network.

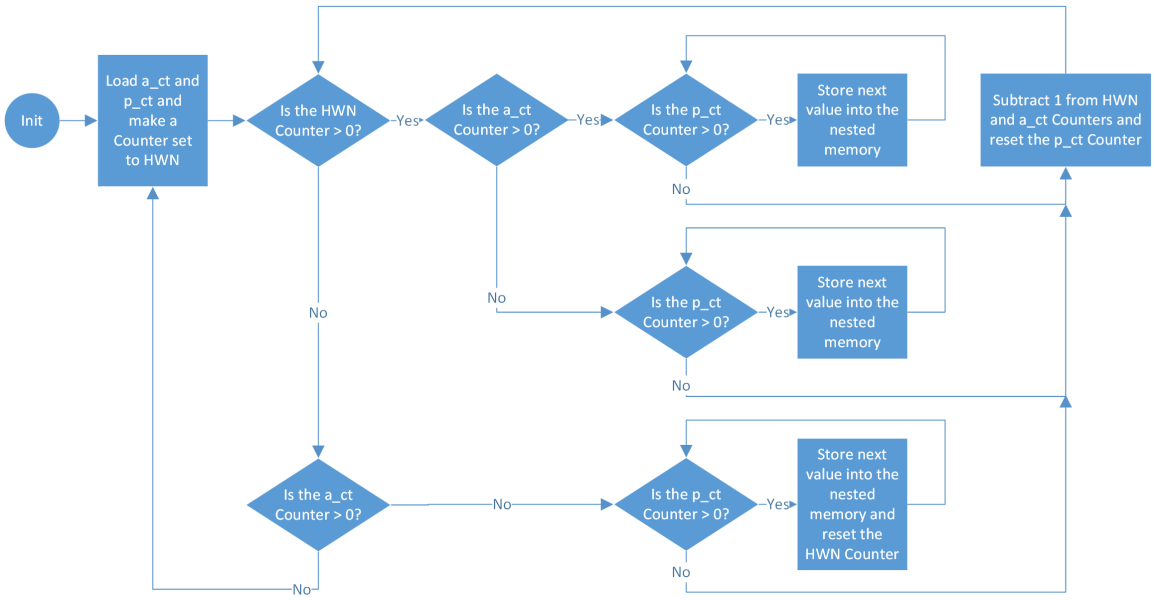

Figure 4.9: The w_loader Flow Diagram

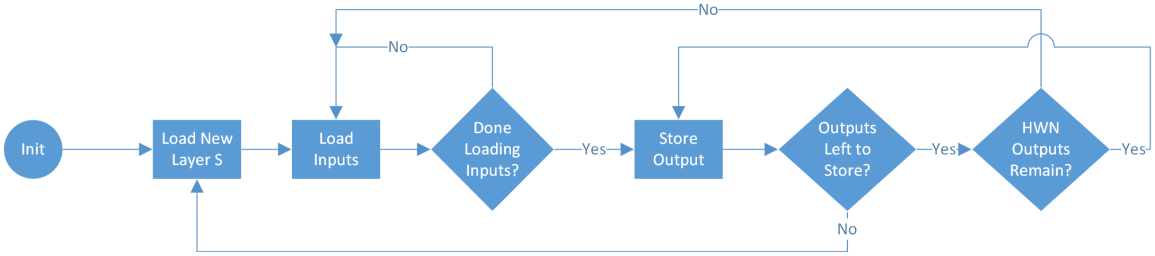

Figure 4.10: The a_loader_store Flow Diagram

$$HardwareLayerIterations = \left\lceil \frac{S}{HWN} \right\rceil \tag{4.1}$$

The number of layer iterations calculated in Equation 4.1 can be substituted into Equations 3.4 and 3.5 (repeated here in Equation 4.2 and Equation 4.3). The description of the variables can be found in Table 5.5. The result is significant because it means that we will see a somewhat linear trend of the increase in speed when looking at a layer with a large number of outputs. Notice that there are two terms that are multiplied by the number of hardware layer iterations in Equation 4.2 and Equation 4.3. The cycles required by the logic to store the outputs from the neurons, C_{SAL}, needs to be separated from the cycles required to store the outputs, C_{SA}, because the

variables were added together and combined in the previous chapter as C_{SA}. Because multiple values are stored in a repetitive state, the logic to reach the storing state is used only once per hardware layer calculation. This is why C_{SAL} is multiplied by the number of hardware layer iterations, while C_{SA} is multiplied by the number of software neurons to be stored. Table 5.5 shows the new variable underlined with description.

Table 4.4: Cycle Variables

Variable	Description
R^n	Number of inputs to the layer n
S^n	Number of neurons of the layer n
C_A	Cycles required for one adder to complete
C_M	Cycles required for one multiplier to complete
C_{LNC}	Cycles of the neuron control logic per input
C_{TF}	Cycles required for the transfer function
C_{SA}	Cycles required for storing a single output
C_{LS}	Cycles required to load a new structure
$\underline{C_{SAL}}$	Cycles required for the logic to begin storing values

$$Cycles^1 = R^1 \left\lceil \frac{S^1}{HWN} \right\rceil (C_M + C_A + C_{LNC}) + \left\lceil \frac{S^1}{HWN} \right\rceil (C_{TF} + C_{SAL}) + S^1 C_{SA} + C_{LS} \tag{4.2}$$

$$Cycles^n = S^{n-1} \left\lceil \frac{S^n}{HWN} \right\rceil (C_M + C_A + C_{LNC}) + \left\lceil \frac{S^n}{HWN} \right\rceil (C_{TF} + C_{SAL}) + S^n C_{SA} + C_{LS} \tag{4.3}$$

4.4 Slice Requirements

Space requirements for this hardware network can be expected to expand by multiplying the size of the neurons and the memory modules by the HWN value. As the network fills the FPGA, the actual required number of slices and cells can be expected to change non-linearly. Variables that make up the new slice equation, found in Table 4.5, now include the size of the b_loader and a_loader blocks. These two variables are underlined for convenience. Equation 4.4 is modified in order to multiply the variables that are duplicated by the number of hardware neurons.

Table 4.5: Slice Variables

Variable	Description
S_M	Slices per multiplier
S_A	Slices per adder
S_{TF}	Slices per transfer function
S_W	Slices per weight ROM
S_{WL}	Slices per weight logic
S_{NC}	Slices per neuron control block
S_B	Slices per bias ROM
S_{BL}	Slices per bias logic
S_{PA}	Slices per a_loader_store block
$\underline{S_{BLD}}$	Slices per b_loader block
$\underline{S_{WLD}}$	Slices per w_loader block
S_{NET}	Slices per network_control block

$$Slices = HWN(S_M+S_A+S_{TF}+S_W+S_{WL}+S_{NC}+S_B)+S_{BL}+S_{PA}+S_{BLD}+S_{WLD}+S_{NET} \quad (4.4)$$

4.5 Simulation Testing and Verification in Vivado

To test if the hardware structure works correctly, the same test case from Chapter 3 verification can be used in order to see that the hardware calculates the same values. The error compared to Matlab does not need to be considered here, because it should be the same as the previous test case. The increase in speed can be verified using the same test case, but should be analyzed under more controlled conditions to see the maximum increase in speed. This can be done by setting the number of software inputs to 1, and the number of software neurons to a large value. By increasing the number of hardware neurons for each test, the increase in speed can be seen clearly.

4.5.1 Verification Comparing Output Values

Using the behavioral simulation from Chapter 3, the weight values will be loaded into the Weight ROMs in a well ordered manner, as shown in Figure 4.1, along with zero values for neurons that do not matter. The values that are calculated will be stored into the Input/Output RAM serially after they are all calculated. As seen in Table 4.6, there are no discrepancies between the calculated values of multiple sizes of HWN. This means that the network is able to scale the amount of hardware and perform the same calculation.

4.5.2 Cycles and Gate Analysis after Synthesis

The number of cycles required for each layer should be verified for the test case. However, to see the increase in speed more clearly, a software layer should be set to 1 input and to a large number of neurons. The maximum number of software and hardware neurons in this example will be 23. Using Equation 4.2 and Equation 4.3, a table of expected values can be constructed and used to anticipate the increase in speed. In order to calculate the increase in speed, the total number of cycles can be divided by the number of cycles required when the HWN count is equal to 1. This

Table 4.6: Comparison of Layer Outputs During Simulation at Different HWN Sizes

Variable	a_1^1	a_2^1	a_3^1	a_4^1	a_5^1	a_6^1
$a_{HWN=1}$	$BB38$	$BBBF$	$3BC6$	$BB93$	$3A6A$	$3BF9$
$a_{HWN=2}$	$BB38$	$BBBF$	$3BC6$	$BB93$	$3A6A$	$3BF9$
$a_{HWN=3}$	$BB38$	$BBBF$	$3BC6$	$BB93$	$3A6A$	$3BF9$
$a_{HWN=4}$	$BB38$	$BBBF$	$3BC6$	$BB93$	$3A6A$	$3BF9$
Variable	a_7^1	a_8^1	a_9^1	a_{10}^1	a_1^2	
$a_{HWN=1}$	$3BFD$	$BBFA$	$3BF0$	$3B12$	$3A7B$	
$a_{HWN=2}$	$3BFD$	$BBFA$	$3BF0$	$3B12$	$3A7B$	
$a_{HWN=3}$	$3BFD$	$BBFA$	$3BF0$	$3B12$	$3A7B$	
$a_{HWN=4}$	$3BFD$	$BBFA$	$3BF0$	$3B12$	$3A7B$	

will result in a ratio of required cycles that we can call the speedup. These values can be compared with a line equal to the HWN value for an ideal reference point. Table 4.8 shows the expected and the measured speedup found in simulation. This is significant, because it means that the equation can accurately predict the number of cycles for any software network. The values that changed from the previous chapter are shown in Table 4.7, and the equations for calculating the number of cycles are shown again for convenience in Equation 4.5 and Equation 4.6.

$$Cycles^1 = R^1 \left\lceil \frac{S^1}{HWN} \right\rceil (C_M + C_A + C_{LNC}) + \left\lceil \frac{S^1}{HWN} \right\rceil (C_{TF} + C_{SAL}) + S^1 C_{SA} + C_{LS} \tag{4.5}$$

$$Cycles^n = S^{n-1} \left\lceil \frac{S^n}{HWN} \right\rceil (C_M + C_A + C_{LNC}) + \left\lceil \frac{S^n}{HWN} \right\rceil (C_{TF} + C_{SAL}) + S^n C_{SA} + C_{LS} \tag{4.6}$$

The resulting values from Table 4.8 match the calculation, and therefore the equations for the cycles have been verified. The equation can be used to extrapolate over

Table 4.7: Modified Values of Cycle Variables

Variable	Value	Description
R^1	4	Number of inputs to the layer 1
S^1	10	Number of neurons of the layer 1
S^2	1	Number of neurons of the layer 2
C_A	8	Cycles required for one adder to complete
C_M	6	Cycles required for one multiplier to complete
C_{LNC}	9	Cycles of the neuron control logic per input
C_{TF}	3	Cycles required for the transfer function
C_{SA}	1	Cycles required to store a single value
C_{SAL}	4	Cycles required for the logic to begin storing values
C_{LS}	1	Cycles required to load a new structure

new cases. The maximum ratio for speedup will occur when there is one input and a large number of outputs. Appendix C shows the values calculated for the number of cycles in these cases, and Figure 4.11 shows the speedup in a graph. In order to calculate speedup, Equation 4.7 is used. Notice the speedup will be equal to 1 when HWN is equal to 1. The ideal speedup is plotted with the speedup of the network for better clarity.

$$Speedup = \frac{TotalCycles^1}{TotalCycles^{HWN}}d \tag{4.7}$$

Notice that the graph in Figure 4.11 is not smooth. This is because of the ceiling function, which causes discontinuities. Just by doubling the number of neurons, the computation time for this layer is nearly doubled. However, as the number of hardware neurons approach the number of software neurons, the benefit of the investment in hardware resources diminishes. Coupling this with the equation for the size re-

Table 4.8: Expected and Measured Cycles for varying Number of Hardware Neurons

Layer	$HWN = 1$	$HWN = 2$	$HWN = 3$	$HWN = 4$
Layer 1	1001	506	407	308
Layer 2	239	239	239	239
Total	1240	745	646	547

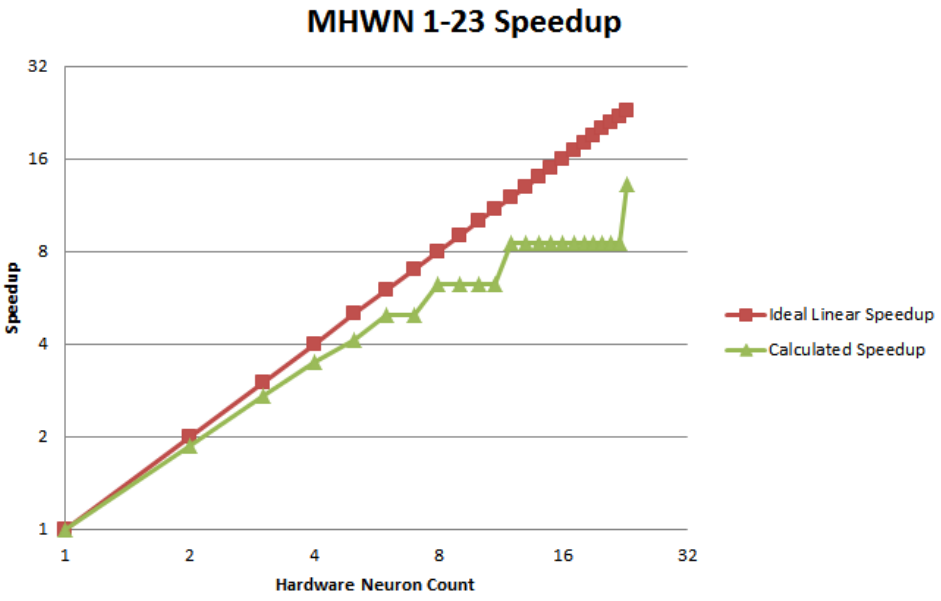

Figure 4.11: Speedup as Hardware Neurons Increase with 1 Input and 23 Neurons

quirements, a heuristic could be used to determine the appropriate size of hardware network for certain sizes of software networks.

Using the equation for the size of the hardware network, the estimated size of the network can be determined. Other slice requirements are considered in Appendix D. A key thing to note is that as the number of hardware neurons increase, the logic that controls them changes size. This logic is created using a compiler that does optimizations, and so the slices required will not only vary in size as the HWN value increases, but will vary within the same project as modules are duplicated. The slices required for a certain module may be slightly different in size when the

Table 4.9: Slice LUT Requirements by Hardware Neuron Count

Variable	HWN=1	HWN=2	HWN=3	HWN=4	HWN=5
S_M	74	74	74	74	74
S_A	178	178	178	178	178
S_{TF}	77	77	77	77	77
S_W	1399	1399	1399	1399	1399
S_{WL}	30	29	30	28	28
S_{NC}	126	127	125	123	123
S_B	64	64	64	64	64
S_{BL}	18	47	48	46	48
S_{PA}	179	170	192	215	195
S_{BLD}	81	83	83	85	87
S_{WLD}	246	253	289	265	274
S_{NET}	49	49	58	53	54
$Total_{Measured}$	2521	4514	6519	8563	10402
$Total_{Calculated}$	2521	4498	6511	8436	10373

module is duplicated, due to overlapping logic or space limitations on the FPGA. This means that the size can only be approximated. The size of these modules vary greatly, depending on the size of network required, and so, in order to form a good approximation, the parameters of the project, like the FPGA being used or the number of bits required for accuracy, may be critical in order to determine an optimal hardware network size.

4.6 Multiple Single-Input Hardware Neuron Summary

This chapter has shown that MSIHNs can be used in order to speedup the calculation of the ANN algorithm on an FPGA. Using a modular approach, the design can be abstracted so that VHDL generate statements, along with arrays of signals, can be used to create dynamically scaling hardware to exploit parallelism in the ANN algorithm. Using equations that model the structure of the hardware and the number of required cycles for a software network, the size and required cycles can be predicted. The exact size of the hardware cannot be easily predicted, because compilation of VHDL code changes the size of the control logic. Other ways to increase speed of the network should be considered in order to find an optimal area and cycle requirements for a given software network. One way is to increase the number of inputs to a hardware neuron, which will be discussed in the next chapter.

CHAPTER 5

Multi-Input Hardware Neuron

This chapter will describe the process of adding multiple inputs to a single hardware neuron. Processing multiple inputs to a neuron will allow for an increase in speed by increasing the slices used and power consumption. After describing the hardware architecture, equations for the increase in speed and the area will be analyzed. Finally, tests will be performed on a sample problem to verify correct network calculations and to verify the theoretical equations for speedup and area requirements.

5.1 Overview of Hardware Implementation of Multi-Input Hardware Neuron

Computing a software neuron requires multiplying multiple software inputs with corresponding weights. These values are added together to produce a sum, previously described in Chapter 3 as the variable n (see Equation 3.1). A single input neuron (see Figure 3.6) contains a single multiplier module which accepts both one input value and one weight value. This process can be parallelized by instantiating multiple multipliers within the neuron. Because of this fundamental idea, the variable for the number of inputs may sometimes be referenced as the MLT count. This variable is used in the schematics when creating arrays of signals. So, a hardware neuron that accepts two inputs will contain two multipliers, meaning that the neuron will be able to multiply two input values and two weights at the same time. In Figure 5.1, the general neuron diagram shows the hardware neuron containing multiple multipliers. Notice that there is an additional adder that is now required. When using more than

two multipliers, an adder tree will need to be used in order to sum the multiplications in parallel. A description of adder trees is given later in this section. Notice that the Weight ROM will need to be divided along the columns, which contrasts with the previous chapter, where the weight matrix was divided along the rows. This is because the weight matrix is an $S \times R$ matrix. Notice that the outputs are distributed along the rows, and the inputs are distributed along the columns. Because in Figure 5.1 there are two inputs to the neuron, the first Weight ROM contains the even columns, while the second Weight ROM contains the odd columns. Because the weights are controlled by a single address signal, the number of columns in the Weight ROMs will need to be divisible by the number of inputs. This means that values for the weight matrix that do not exist should be set to zero. Because the Input/Output RAM can still only be serially accessed, additional logic will be required to load multiple inputs into the neuron.

Within the hardware neuron, an adder tree is required to compute the sum of multiple multipliers in parallel. The adder tree will generate a single sum that can be input into the accumulating adder which calculates the sum of n over multiple input iterations based on the number of software inputs. The adder tree, shown in Figure 5.2, shows the basic structure of an adder tree. This is similar to a structure used in computer science called a complete binary tree. The value of Inputs is the number of inputs to the neuron. Notice that when there are 3 inputs, the top group of adders will contain 2 adders being fed down into a final adder. This is not the most efficient way to make the adder tree, because the adders could be cascaded in a way that only two adders are used. However, the code to generate this regular structure is much more understandable than the alternative. The timing and size requirements of this structure are more easily calculated too. The depth of the tree can be calculated by taking the log of the number of inputs. This is critical for counting the number of cycles and for generating the hardware of the adder tree, because each depth will

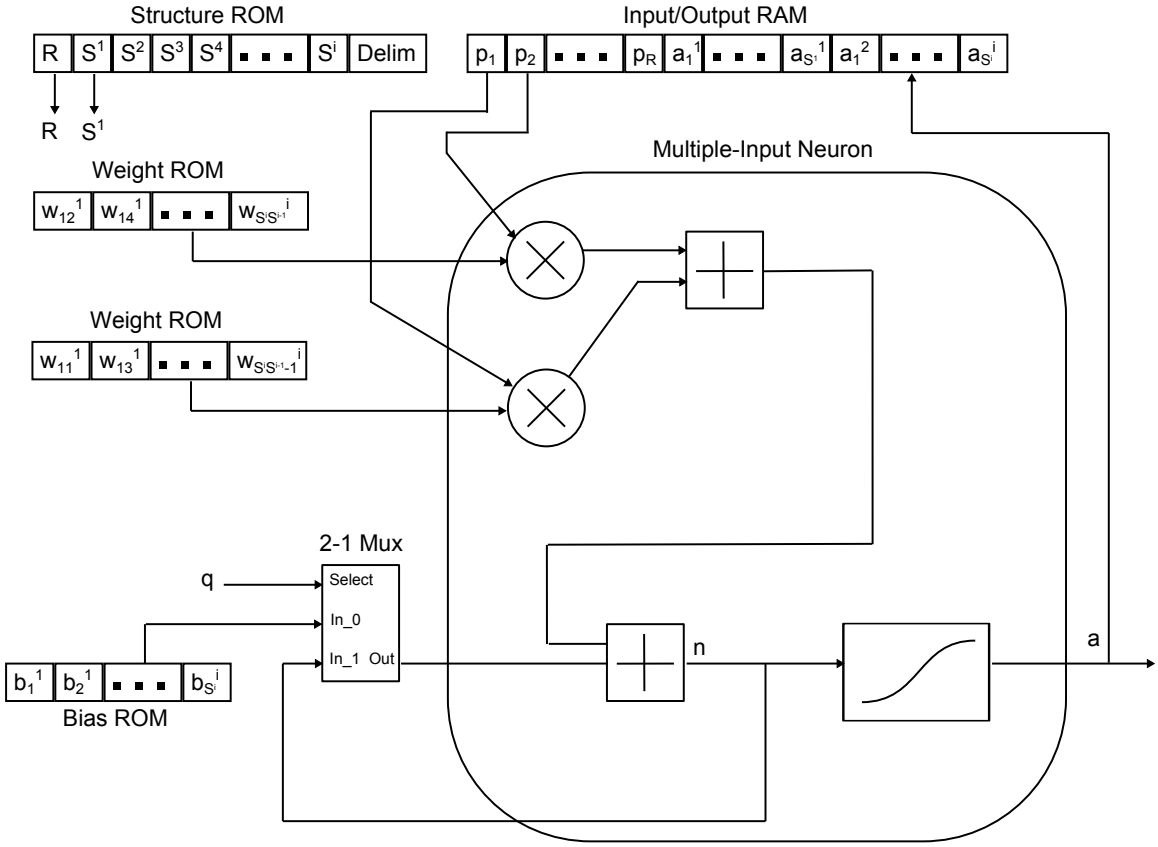

Figure 5.1: The General Multi-Input Hardware Neuron Diagram

contribute an additional delay of an adder module.

Calculating the area of the adder tree requires knowing the total number of nodes in the tree. Instead of calculating the summation for each of the rows, a closed form solution can be used. The closed form equation for the number of nodes in a complete binary tree is given by Equation 5.1. The variable N is the number of nodes on the top group of adders in the tree, sometimes called the leaf nodes. Therefore, as shown in Equation 5.2, by substituting in the number of inputs to the tree, we can relate the number of nodes in the adder tree to the number of inputs to the neuron.

$$Nodes = \sum_{n=1}^{N} n = \frac{N(N+1)}{2} \tag{5.1}$$

$$Nodes = \frac{\left\lceil \frac{Inputs}{2} \right\rceil \left(\left\lceil \frac{Inputs}{2} \right\rceil + 1 \right)}{2} \tag{5.2}$$

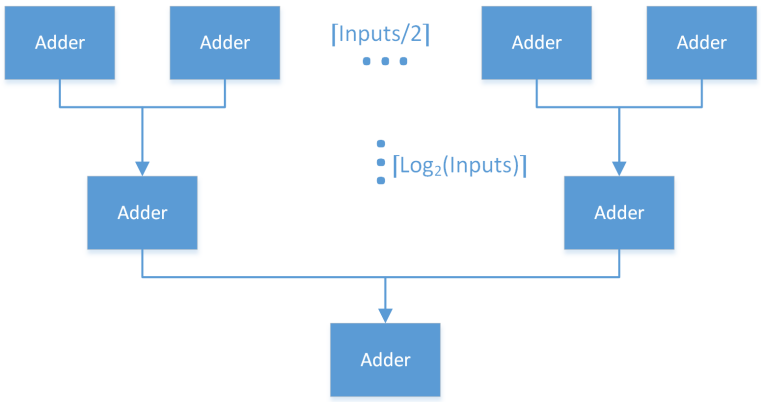

Figure 5.2: Adder Tree Structure

5.2 Implementation in VHDL

Abstraction levels of the previous designs will need to be modified in order to add multiple inputs to the hardware neuron. Similar to the last chapter, a loader module can be used in order to store values into a new module that contains the Weight ROMs, both described later in this section. Modifications to the state machines that control how values are stored and loaded into the Weight ROMs are described too. Arrays of signals will need to be added to each abstraction layer to make the design scalable based on the number of inputs. The first module that should be analyzed is at the lowest level of abstraction - the hardware neuron.

5.2.1 Modification to the Neuron

In order to accommodate the additional inputs, the neuron will need to have a module that creates multiple multipliers, and another module to create the adder tree. These two modules are the key differences between the neurons described in previous chapters (see Figures 3.6 and 4.1) and the new neuron schematic, shown in Figure 5.3. The control signals are the same as those in the previous chapters, but the input and weight signals are now updated to arrays of 16-bit signals, length MLT. Notice the signals going into the multipliers block have the new notation w_mlt_16 and p_mlt_16.

This means that these signals are an array of 16-bit signals, length MLT. The control for the neuron is the same as those in the previous chapters, and will not be described here. The multipliers module uses the variable MLT in order to generate the number of multipliers, and outputs all of results as an array of values, size $\lceil \frac{Inputs}{2} \rceil$. The adders module generates the adder tree using loops of VHDL generate statements that are based on the total number of nodes in the adder tree. Combining each of the modules together, the inputs and outputs can be abstracted into a smaller module, the neuron module.

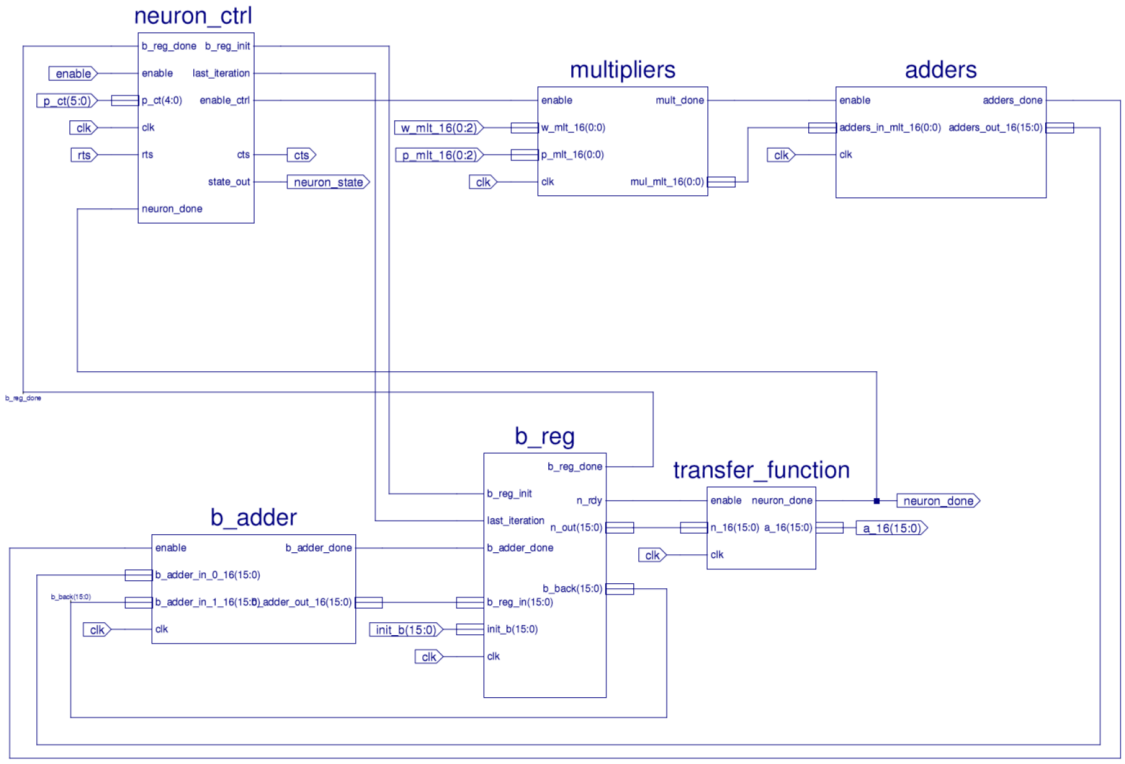

Figure 5.3: The Multi-Input Neuron Schematic

Table 5.1: Neuron Schematic Signals

Signal	Description
enable	Enables only the neuron_ctrl block.
clk	Clocks signals for timing.
p_ct	Number of software inputs to process.
rts	External module sets high to request to change inputs.
cts	Neuron control sets high to allow change of inputs.
neuron_state	Allows for easy test bench access to states.
w_mlt_16	Array of MLT 16-bit weight values.
p_mlt_16	Array of MLT 16-bit input values.
a_16	The 16-bit output value.
init_b	The bias value.
mul_mlt_16	$\lceil \frac{MLT}{2} \rceil$ 16-bit multiply results.
mult_done	Enables the adder tree when multipliers values are computed.
adders_done	Enables the b_adder when the adder tree output value is computed.
neuron_done	Set high when transfer function has been computed.
b_reg_done	Set high when b_reg is finished saving b_adder output.
b_reg_init	Set high when input bias needs loaded into b_reg.
b_adder_done	Enables b_reg when the b_adder is complete.
n_rdy	Set high when n is computed.
n_out	The value of n.
b_back	Value used to accumulate values, initialized as bias.

5.2.2 Adding the Weight ROMs

The Weight ROM will need to be broken up into multiple ROMs to produce multiple inputs for the neuron. Figure 5.5 shows the new w_mlt_mem module, which contains a VHDL generate statement producing MLT Weight ROMs. The output of each Weight ROM is clustered into the w_mlt_16 signal, which is connected to the neuron. The logic controlling the w_mlt_mem module changes from the w_mem modules in the previous chapters by including logic for storing values from a w_loader module. The w_write signal is used to tell the module whether or not values are being written into the module, or if the values are being clocked out of the ROMs into the neuron. The RTS and CTS pins are used to signal to the loader when the neuron is ready to load more data into the ROMs.

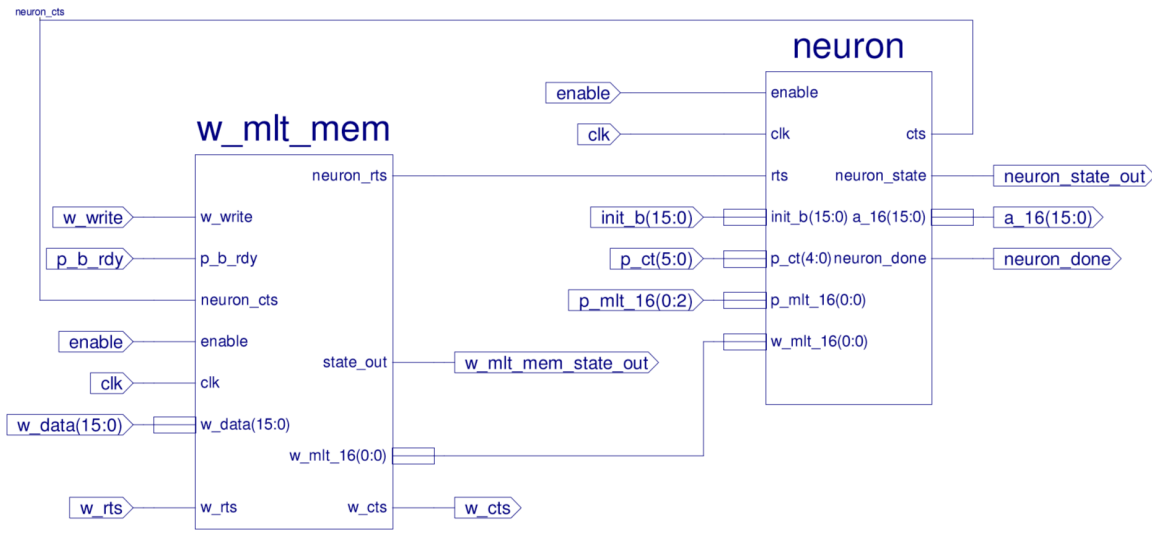

Figure 5.4: Adding the Weight ROMs Schematic

The flow diagram, shown in Figure 5.5, shows the new states for the module. If the values have been loaded, then the values are clocked into the neuron as usual. An additional counter is required in order to enable and disable specific Weight ROMs.

Table 5.2: Weight ROM New or Important Schematic Signals

Signal	Description
enable	Enables both the w_mem and neuron.
clk	The clock for the modules.
p_b_rdy	Set high when input and bias are ready.
neuron_rts	Set high when inputs, bias, and weights are ready.
neuron_cts	Set high when neuron allows values to change.
w_mlt_16	Array of MLT 16-bit weight values.
w_mem_state_out	The state of the w_mem for the test bench.
w_write	Set high when the Weight ROMs are being written to.
w_data	The data being stored into the Weight ROMs.
w_rts	Set high when w_data is requested to change.
w_cts	Set high when w_data is safe to change.

This means that the address is not incremented until the total number of MLT values have been stored into each of their respective memories. The combination of blocks and signals can be encapsulated into the hardware_neuron module.

5.2.3 Adding the Bias ROM

Other than the connections for the hardware neuron, the control signals and the states of the bias_loader are exactly the same as the single-input single-hardware neuron case because there is only one neuron in the hardware_neuron block. The b_mem module does not need to be modified to handle multiple hardware neurons. The b_mem module is attached to the hardware_neuron module by the b_16 wire, the value of the bias for a given software neuron. The b_mem contains only one Bias ROM, and therefore does not require a loader module. Figure 5.6 shows the schematic of the hardware_neuron with the Bias ROM attached.

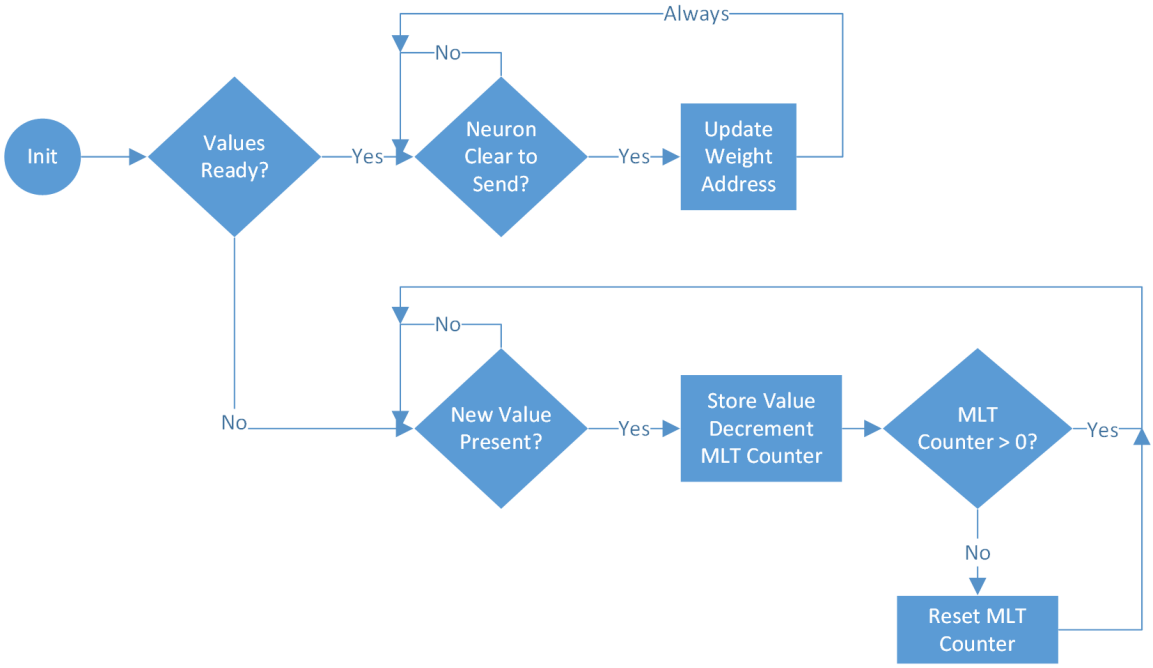

Figure 5.5: The w_mlt_mem Flow Diagram

The states of the b_mem module are strictly for loading the bias values into the neuron and making them simple. If the hwn_done flag is set, then the bias value will update by increasing the address value to load the new input. Figure 5.7 shows the states the module traverses.

5.2.4 Modification to the Network Block

The primary changes to the network for the multi-input hardware neuron (MIHN) are the introduction of a w_loader module and the modification of the input signal p_16 to an array of 16-bit signals, p_mlt_16. The schematic in Figure 5.8 shows the changes included in the schematic from Chapter 3. A more detailed schematic is located in Appendix E, Figure E.3. The network_control module contains the same flow diagram from Chapter 4, because it uses the same logic for all of the loader modules.

The states of the network_control module, shown in Figure 5.9, describe how the

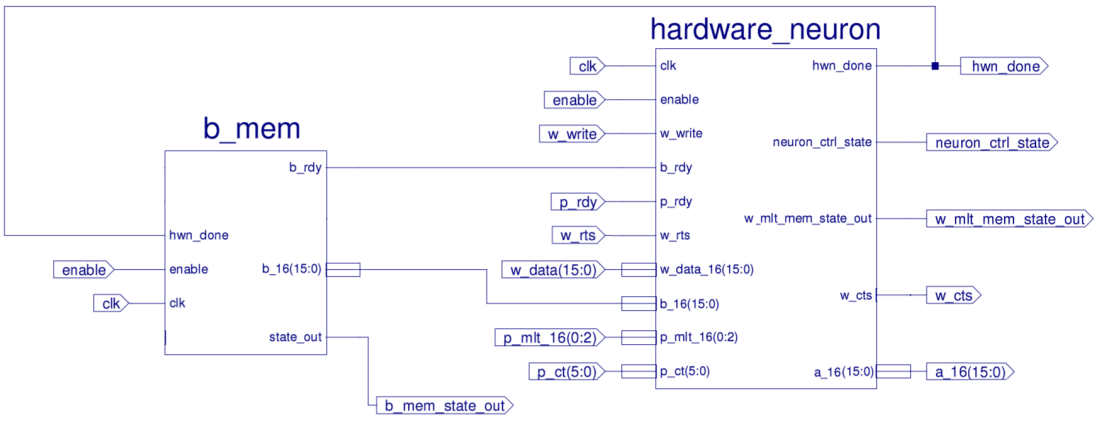

Figure 5.6: Adding the Bias ROM Schematic

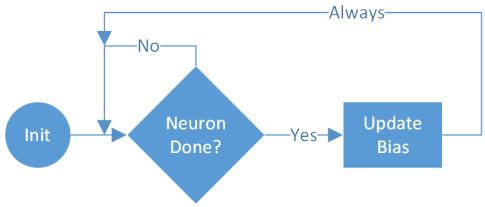

Figure 5.7: The b_mem Flow Diagram

memory is clocked out of the Structure ROM. The values first check a flag to see if the memory is loaded, if not, then the loaders will need to write all of the values into the hardware neuron. After they have been loaded, the hardware neuron is reset, and the values are loaded again for the hardware neuron and loader modules. To signal the memory modules below that values are stored, the write_all flag will be set to low.

Splitting the weight matrix with the loader modules ensures that testing multiple cases in the behavioral simulation can be done quickly. The w_loader needs to use the number of inputs to the neuron and the corresponding layer inputs in order to decide when to load zeros into the weight ROMs inside the neuron. Shown in Figure 5.10, a counter is used to load MLT weights, but if the p_ct value, the number of software inputs, is zero, then no more weights exist for this hardware layer, meaning

Table 5.3: Bias ROM New or Important Schematic Signals

Signal	Description
enable	Enables b_mem and w_hardware_neuron.
clk	Clock for the modules.
p_rdy	Set high when the input value is ready.
b_rdy	Set high when the bias value is ready.
b_16	16-bit bias values.
a_16	16-bit output values.
hwn_done	Set high when the output of the hardware_neuron is ready.
w_rts	RTS values for w_hwn_mem.
w_cts	CTS values for w_hwn_mem.

that a zero value should be stored. If the values have been stored, then the w_loader module will stay in the initialization state, so no values are clocked in.

To increase the number of inputs to the hwn_b_mem module, the a_loader_store states will need to be changed. Shown in Figure 5.11, whenever values are being loaded into the neuron, instead of loading a single value, MLT values are loaded. These will need to be done serially, because the Input/Output RAM is a single output module. So to get multiple modules, multiple accesses will need to be made into the memory. After the values have been loaded, the a_loader_store module will wait for the CTS flag from the neuron, signaling that new input values are ready to be loaded. If the counter for the software inputs within the a_loader_store module is equal to zero, then all inputs have been loaded. If all inputs have been loaded, then the hardware neuron done signal will be set, so the output of the neuron can be stored. After all outputs of the software layer have been stored, new numbers of inputs and outputs are loaded for the next layer.

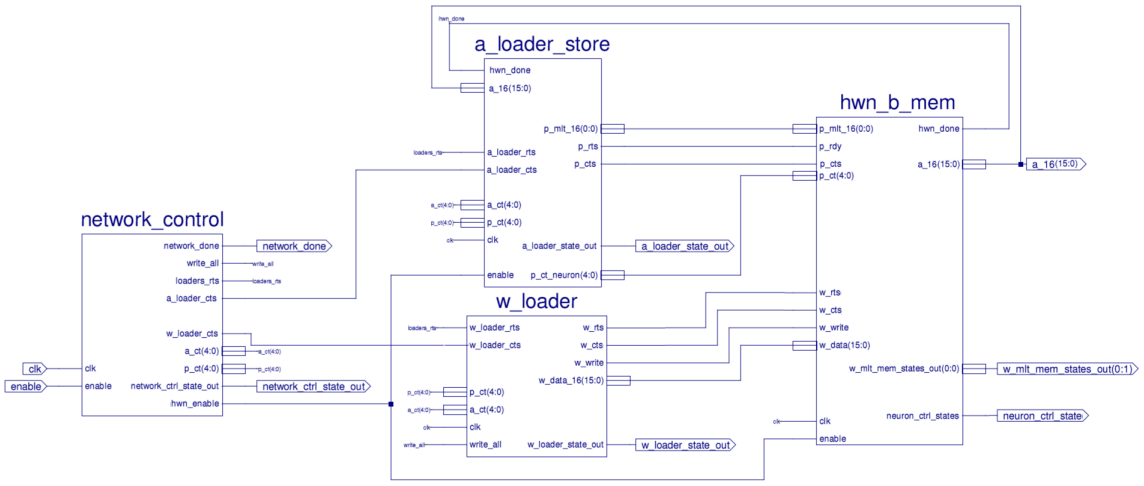

Figure 5.8: Neural Network Multi-Input Single-Hardware Neuron Schematic

5.3 Clock Cycle Requirements

Required clock cycles are calculated by modifying the equation for the single-input single hardware neuron to accommodate the division of software inputs by the inputs to the hardware neuron. To do this, the number of input iterations, shown in 5.3, can be considered. Because there was previously only one hardware input to the neuron, the cycles to load the inputs were lumped into the value for one input iteration. However, because sets of inputs can be computed, multiple inputs can now be handled in a single input iteration. The iterations are based on the number of software inputs. If we divide the number of software inputs by the number of hardware inputs, the remainder of the division represents the fraction of the last iteration that will be used. However, because the number of input iterations must be an integer, the ceiling function is used to obtain the total number of iterations.

$$InputIterations = \left\lceil \frac{R}{MLT} \right\rceil \tag{5.3}$$

New cycle variables will need to be added to those defined in the previous chapters. The new variables are underlined in Table 5.5. They are the number of inputs to the

Table 5.4: Network Schematic New or Important Signals

Signal	Description
enable	Enables only the network_control.
clk	Clock for the modules.
network_done	Set high when software network has been calculated.
write_all	Set high when writing values to nested memory.
loaders_rts	Set high when a_ct and p_ct requested to change.
a_loader_cts	Set high when a_ct and p_ct safe to change.
a_ct	Software neurons for layer.
p_ct	Software inputs for layer.
network_ctrl_state_out	State of network_control for test bench.
hwn_enable	Enables the calculation of the hardware layer.
p_mlt_16	Array of MLT 16-bit input values.
a_16	16-bit output values.
p_rts	Set high when new input value requested to change.
p_cts	Set high when new input value safe to change.
a_loader_state_out	The state of a_loader_store for test bench.
p_ct_neuron	The number of software inputs for the layer.

hardware neuron, MLT, and the number of cycles required for loading a value, C_{LA}.

The cycle equations from previous chapters will need to be modified to accommodate the serial components of the parallel structures. These sets of serial components cause delay. The set of multipliers still only counts for the same number of delays, but the depth of the adder tree has not yet been accounted for. From Equations 3.4 and 3.5 for the single-input neuron, we can derive a new set of equations - Equations 5.4 and 5.5. Originally, the values of the C_{LNC} included the cycles that were required to load a value. Because multiple values must be loaded for each iteration, and the

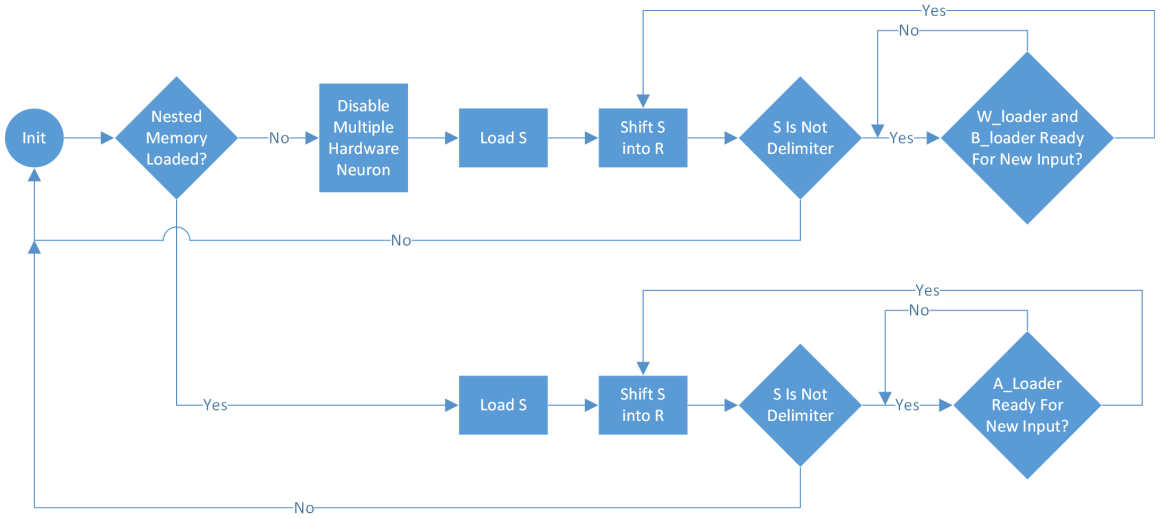

Figure 5.9: The network_control Flow Diagram

number of values change based on MLT, the variable for C_{LNC} can be broken into $MLT \times C_{LA}$. In order to accommodate the increase in delay through the adder tree, the number of adder tree layers can be multiplied by the number of cycles required per adder. Then, all of the other contributing delays can be added or multiplied as usual.

$$Cycles^1 = \left\lceil \frac{R^1}{MLT} \right\rceil S^1(C_M + C_A(\lceil Log_2(MLT) \rceil + 1) + C_{LNC} + MLT(C_{LA})) + S^1(C_{TF} + C_{SAL} + C_{SA}) + C_{LS} \tag{5.4}$$

$$Cycles^n = \left\lceil \frac{S^{n-1}}{MLT} \right\rceil S^n(C_M + C_A(\lceil Log_2(MLT) \rceil + 1) + C_{LNC} + MLT(C_{LA})) + S^n(C_{TF} + C_{SAL} + C_{SA}) + C_{LS} \tag{5.5}$$

5.4 Slice Requirements

The number of slices can be calculated by simply adding up each of the modules that are instantiated. The only variables that need to be added to the equations from

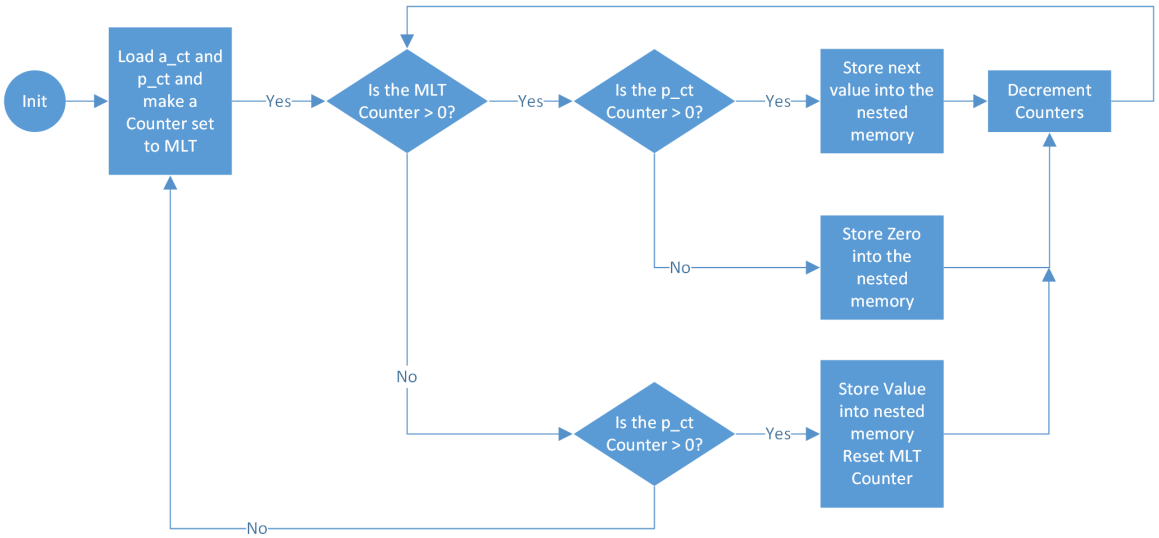

Figure 5.10: The w_loader Flow Diagram

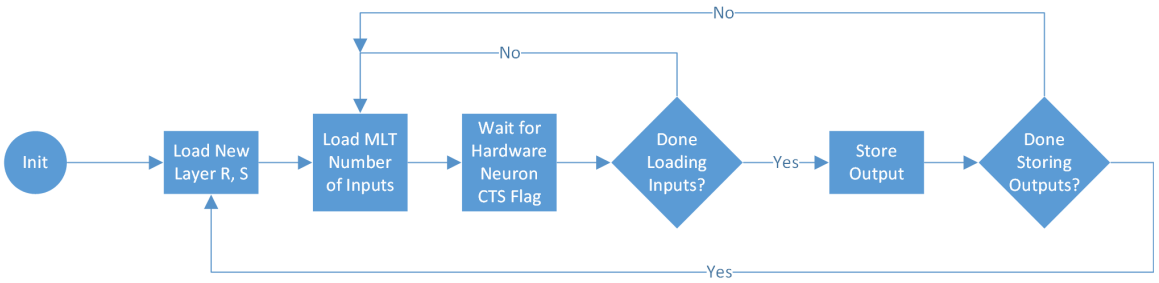

Figure 5.11: The a_loader_store Flow Diagram

the previous chapter are the number of slices per w_loader module and the number of inputs to a neuron. Table 5.6 shows the list of variables with their descriptions. Instead of simply adding up all of the blocks, as was done in Chapter 3, the number of inputs should be multiplied by the size of the Weight ROMs and the size of the multipliers. The number of adders should be counted in order to add them up as well. Using the equation for the number of adder tree nodes, Equation 5.2, the nodes can be multiplied by the amount of slices required for a single adder to get the required number of slices for the adder tree. However, this will not work for the single input case, because the adder tree structure is completely bypassed with a generate statement. This means that this slice equation will calculate one too many adders

Table 5.5: Cycle Variables

Variable	Description
R^n	Number of inputs to the layer n
S^n	Number of neurons of the layer n
$\underline{MLT}$	Number of inputs to the hardware neuron
C_A	Cycles required for one adder to complete
C_M	Cycles required for one multiplier to complete
C_{LNC}	Cycles of the neuron control logic per input
C_{TF}	Cycles required for the transfer function
$\underline{C_{LA}}$	Cycles required for loading a single value
C_{SA}	Cycles required for storing a single output
C_{LS}	Cycles required to load a new structure
C_{SAL}	Cycles required for the logic to begin storing values

when MLT is equal to 1. So, this equation is only valid for MLT values greater than 1.

$$Slices_{MLT>1} = MLT(S_M + S_W) + (\frac{\lceil \frac{MLT}{2} \rceil (\lceil \frac{MLT}{2} \rceil + 1)}{2} + 1)S_A + S_{TF} + S_{WL} + S_{NC} + S_B + S_{BL} + S_{PA} + S_{WLD} + S_{NET} \quad (5.6)$$

5.5 Simulation Testing and Verification in Vivado

To verify that the design works as expected, output values for multiple sizes of input neurons can be tested. This design has been tested extensively using many different software networks and many different MLT values. However, only a small subset of values will be shown here. The equations for the number of cycles and slices can be analyzed in order to ensure that the hardware matches what is expected of the

Table 5.6: Slice Variables

Variable	Description
$\underline{MLT}$	Number of Inputs to a hardware neuron
S_M	Slices per multiplier
S_A	Slices per adder
S_{TF}	Slices per transfer function
S_W	Slices per Weight ROM
S_{WL}	Slices per weight logic
S_{NC}	Slices per neuron control block
S_B	Slices per Bias ROM
S_{BL}	Slices per bias logic
S_{PA}	Slices per a_loader_store block
$\underline{S_{WLD}}$	Slices per w_loader block
S_{NET}	Slices per network_control block

system.

5.5.1 Verification Comparing Output Values

Using the same test case from the previous chapters, the values for each neuron output can be compared as the number of hardware neurons changes. The 16-bit results will still have the same error when compared to more accurate calculations from Matlab. However, the goal is to ensure that the values are all calculated the same, no matter the hardware structure. Table 5.7 shows each of the outputs of each neuron from the test case. Notice that they are all identical. These values show that the w_loader is storing values into the Weight ROMs correctly, and that the adder tree is able to compute the correct values. Because multiple input iterations were used for each case, the storing and loading of inputs are verified.

Table 5.7: Comparison of Layer Outputs During Simulation at Different MLT Values

Variable	a_1^1	a_2^1	a_3^1	a_4^1	a_5^1	a_6^1
$a_{MLT=1}$	$BB38$	$BBBF$	$3BC6$	$BB93$	$3A6A$	$3BF9$
$a_{MLT=2}$	$BB38$	$BBBF$	$3BC6$	$BB93$	$3A6A$	$3BF9$
$a_{MLT=3}$	$BB38$	$BBBF$	$3BC6$	$BB93$	$3A6A$	$3BF9$
$a_{MLT=4}$	$BB38$	$BBBF$	$3BC6$	$BB93$	$3A6A$	$3BF9$
Variable	a_7^1	a_8^1	a_9^1	a_{10}^1	a_1^2	
$a_{MLT=1}$	$3BFD$	$BBFA$	$3BF0$	$3B12$	$3A7B$	
$a_{MLT=2}$	$3BFD$	$BBFA$	$3BF0$	$3B12$	$3A7B$	
$a_{MLT=3}$	$3BFD$	$BBFA$	$3BF0$	$3B12$	$3A7B$	
$a_{MLT=4}$	$3BFD$	$BBFA$	$3BF0$	$3B12$	$3A7B$	

5.5.2 Cycles and Slice Analysis After Synthesis

In order to verify that the design is working as expected, the number of required cycles and slices can be predicted with the equations. First, the individual values can be measured, and then using the values in the equation, the number of cycles for the first hardware iteration can be computed. The variables are defined again in Table 5.8, with measured values. These values are found by measuring the time between states in the simulation. The underlined variables in the table are changed from the previous chapters. The variable is decreased by one for inputs equal to 1, because, for the single-input case, the cycles required to load the value were included in the C_{LNC} value. Now that the equation explicitly loads these values separately, the value is broken out into the C_{LA} variable. However, when the adder tree is instantiated, the neuron_control logic will use one extra cycle in order to make sure that the adder tree has been reset. This means that the logic per input will be one greater than when the adder tree is not instantiated, so, when MLT is greater than one, C_{LNC} is

9.

Table 5.8: Modified Values of Cycle Variables

Variable	Value	Description
R^1	4	Number of inputs to the layer 1
S^1	10	Number of neurons of the layer 1
S^2	1	Number of neurons of the layer 2
C_A	8	Cycles required for one adder to complete
C_M	6	Cycles required for one multiplier to complete
$\underline{C_{LNC}^{MLT=1}}$	8	Cycles of the neuron control logic per input iteration
$\underline{C_{LNC}^{MLT>1}}$	9	Cycles of the neuron control logic per input iteration
C_{TF}	3	Cycles required for the transfer function
C_{LA}	1	Cycles required for loading a single output
C_{SA}	1	Cycles required to store a single value
C_{SAL}	4	Cycles required for the logic to begin storing values
C_{LS}	1	Cycles required to load a new structure

The calculated and expected cycles for the test case can be found in Table 5.9. In order to measure the length of the first iteration, the change of states out of the network_control module can be selected in the simulator, then the times between when new structures are loaded can be subtracted and divided by the frequency of the clock cycles. In Table 5.9, there is no error in the predicted values for the required number of cycles for each of the cases. This equation was tested extensively over many different software networks and hardware configurations. So, the values for this case can be measured and predicted exactly, meaning that the equations can be used to extrapolate over a larger range of values.

In order to see the maximum increase in speed, the number of inputs to the network can be increased to a substantial size, in this case 23. The value 23 was chosen

Table 5.9: Expected and Measured Cycles for varying Number of Hardware Neurons

Layer	$MLT = 1$	$MLT = 2$	$MLT = 3$	$MLT = 4$	$MLT = 5$
$Layer1$	1001	721	901	501	591
$Layer2$	239	174	177	138	113
$Total$	1240	915	1098	649	714

because it is a large prime number. The idea being that if the value is not divisible by another number, the speedup can be characterized more accurately. Figure 5.12 shows the values for the speedup of a 23 input and 1 software neuron network. The graph calculates speedup as described in Chapter 4. The maximum speedup is found when the number of hardware inputs is equal to the number of software inputs. If the number of hardware inputs is larger than the number of software inputs, then there will not be any more speedup, because there is no longer any parallelism to exploit. However, the speedup will actually decrease for the MIHN case, because, as the number of inputs increases, the layers in the adder tree will increase too. This adds cycles to the neuron calculation and would also be very inefficient in slice requirements.

The calculations for the slice requirements are less complex than the cycle requirements. As the total number of slices approaches the size limitations on the board, errors can be expected in the equation. The router will attempt to fill more slices with more logic, but the connections between the slices may become very large, causing errors. Table 5.10 shows the total number of slices measured against the values calculated. These values match very closely. Notice that because the equation being used is intended for when the number of multipliers is greater than one, the slice measurement when MLT is equal to 1 contains an extra adder. The adder tree is bypassed when the number of inputs is one, and so the adder is not generated. Size restraints of the FPGA could be used with these values and equations to predict the

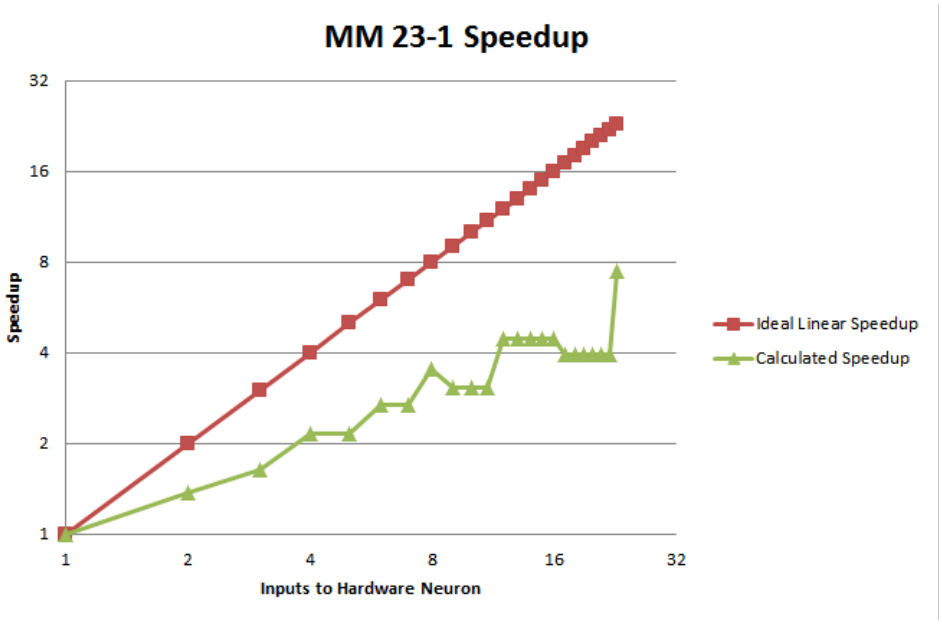

Figure 5.12: Speedup as Hardware Neurons Increase with 23 Inputs and 1 Neuron

maximum number of inputs possible for a single hardware neuron.

5.6 Multi-Input Hardware Neuron Overview

With the development of the MIHN, this chapter has shown that multiple inputs to a hardware neuron can be processed in parallel, decreasing the number of cycles required to obtain the result of an ANN algorithm. Multiple multipliers can be instantiated within the neuron with an adder tree for parallelization. The tree structure can be characterized by the number of nodes and the height, meaning that the area and cycles required can be predicted. These values can be analyzed to understand the trade-offs between required area and speed increase. Modifications could be made to the size of the Weight ROM to decrease overall area. Ideally, the number of weights for a software network should only require a certain amount of memory, no matter the number of inputs to the neuron. So, the area should be able to be fixed with the number of inputs to a neuron. Because the number of inputs and the number of outputs can both be parallelized, they can be combined together so that the trade-offs

Table 5.10: Slice LUT Requirements by Number of Inputs Count

Variable	MLT=1	MLT=2	MLT=3	MLT=4	MLT=5
S_M	74	74	74	74	74
S_A	178	178	178	178	178
S_{TF}	77	77	77	77	77
S_W	1399	1399	1399	1399	1399
S_{WL}	30	57	57	64	62
S_{NC}	126	126	122	126	125
S_B	64	64	64	64	64
S_{BL}	18	18	16	16	16
S_{PA}	179	192	282	302	333
S_{WLD}	246	246	288	235	251
S_{NET}	49	49	56	57	56
$Total_{Measured}$	2440	4132	6095	7547	9598
$Total_{Calculated}$	2618	4131	6093	7545	9595

between the size requirements and speed up can be analyzed more effectively. The past two chapters have described each of these processes individually, and the next chapter will explain how to combine the two parallelization techniques together.

CHAPTER 6

Multiple Multi-Input Hardware Neurons

In the previous chapters, adding multiple inputs to a hardware neuron or multiple hardware neurons to a hardware layer were used to decrease the number of cycles required to simulate a software network. These concepts can be combined together to create a hardware layer that contains multiple multi-input hardware neurons (MMIHN). This chapter will cover the changes that must be made to the hardware network in order to accommodate the additional inputs and layers. First, the general problem is described, then the hardware changes that must be made are discussed. Next, the equations for the required cycles and slices for the general network are introduced. A test case is considered to ensure the calculation is correct, and then the equations are verified. Finally, a plot of the maximum expected speedup is plotted and compared against the slices required.

6.1 Overview of Hardware Implementation of Multiple Multi-Input Hardware Neurons

By combining the multi-input neuron (MIHN) of Chapter 5 with the concept of multiple hardware neurons from Chapter 4, parallelism can be maximally exploited, with multiple MIHNs. As shown in Figure 6.1, the neuron contains multiple multipliers and then an adder tree. This MIHN is then copied. The weight matrix is divided along the rows for the multiple neurons and divided along the columns for the multiple inputs. Because the weight matrix is divided in both dimensions, maximal parallelism is exploited when the number of inputs is equal to the number of rows, and the num-

ber of neurons is equal to the number of columns. Adding additional hardware past this point does not provide any more speedup. This means that the software network determines what kind of hardware should be generated. In this example, the Weight ROMs in the first neuron contain the odd rows, while the Weight ROMs in the second neuron contain the even rows. The first Weight ROMs in both neurons contain the even columns, while the second Weight ROMs in each neuron contain the odd columns of the weight matrix. In this structure, the values that do not exist in the weight matrix will need to be set to zero. Logic will need to be added to the Input/Output RAM to accommodate loading multiple inputs and storing multiple outputs.

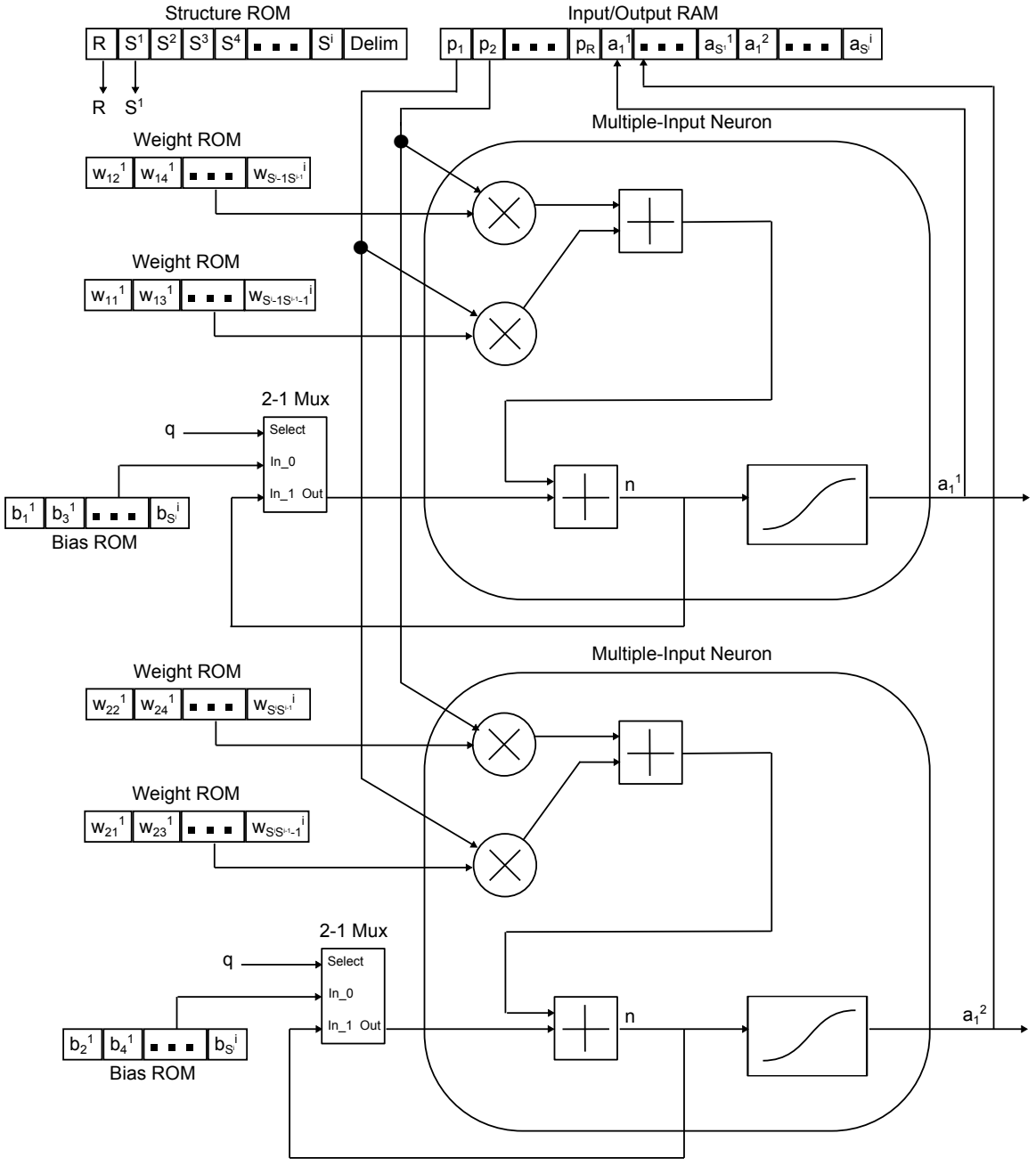

Figure 6.1: The General Multiple Multi-Input Hardware Neurons Diagram

6.2 Implementation in VHDL

Because of the modular design from the previous chapters, both designs can be combined together by taking certain modules from each chapter. From Chapter 4, the modules used are the b_loader module and the modules which copy the neuron and Bias ROMs. The modules used from Chapter 5 are the MIHN and the module which copies the Weight ROM. The w_loader and a_loader_store modules cannot be copied, because they require additional logic to handle both inputs and outputs. The network_control module is the same as in the other chapters, because it needs to control when values are being loaded into the memory modules within the neuron.

6.2.1 Changes in the Neuron

The neuron is the MIHN from Chapter 5, and there are not any changes to the schematic or logic controlling the neuron. As seen in Figure 6.2, in order to create a dynamically scaling number of inputs, the multiplier block is assigned signals w_mlt_16 and p_mlt_16. A more detailed schematic can be found in Appendix E, Figure E.5. The variable MLT, short for multipliers, is used as the number of inputs to the neuron, because the number of inputs to the neuron corresponds to the number of multipliers within the neuron. By using the notation w_mlt_16, the signal stands for the weight value, copied MLT times, and is 16-bits. The output of the neuron is a single a_16 value, because only one output is computed at this level of abstraction. Table 6.1 provides a description of each of the signals in the schematic.

States of the neuron_ctrl module do not change for any number of inputs. The signals are all based on when b_reg is complete or when the transfer function is done. As seen in Figure 6.3, a value for R is loaded for this software layer. If the inputs are ready to be calculated, then the multiplier is enabled. The results are cascaded

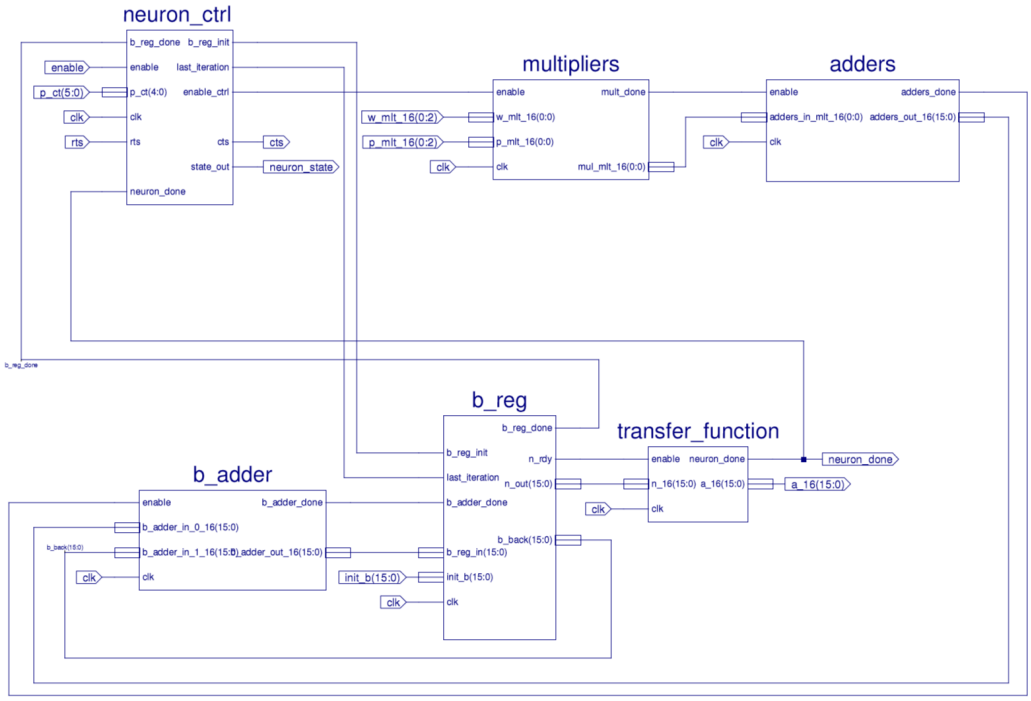

Figure 6.2: The Final Hardware Neuron Schematic

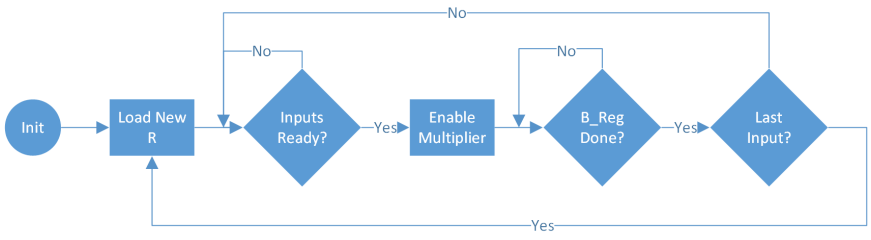

Figure 6.3: The Final Hardware Neuron Flow Diagram

through the multipliers, adder tree, and b_adder until the accumulated n value is produced, setting b_reg_done high. If it is the last set of inputs, then after b_reg is done, the transfer function will be enabled, causing the neuron_done flag to be set high, and the output a_16 to be set. This completes the calculation of one neuron.

Table 6.1: Neuron Schematic Signals

Signal	Description
enable	Enables only the neuron_ctrl block.
clk	Clocks signals for timing.
p_ct	Number of software inputs to process.
rts	External module sets high to request to change inputs.
cts	Neuron control sets high to allow change of inputs.
neuron_state	Allows for easy test bench access to states.
w_mlt_16	Array of MLT 16-bit weight values.
p_mlt_16	Array of MLT 16-bit input values.
a_16	The 16-bit output value.
init_b	The bias value.
mul_mlt_16	$\lceil \frac{MLT}{2} \rceil$ 16-bit multiply results.
mult_done	Enables the adder tree when multipliers values are computed.
adders_done	Enables the b_adder when the adder tree output value is computed.
neuron_done	Set high when transfer function has been computed.
b_reg_done	Set high when b_reg is finished saving b_adder output.
b_reg_init	Set high when input bias needs loaded into b_reg.
b_adder_done	Enables b_reg when the b_adder is complete.
n_rdy	Set high when n is computed.
n_out	The value of n.
b_back	Value used to accumulate values, initialized as bias.

6.2.2 Changes When Adding the Weight ROMs

In order to maximize parallelism in the structure, the w_mlt_mem module, shown in Figure 6.4, copies memory modules MLT times to create the w_mlt_16 array of signals. This module contains the RTS and CTS signal lines, which are connected to the loader modules at the highest level of abstraction. The benefit is that the size and type of the copied memory modules can be changed to optimize area later in development, without changing the timing analysis.

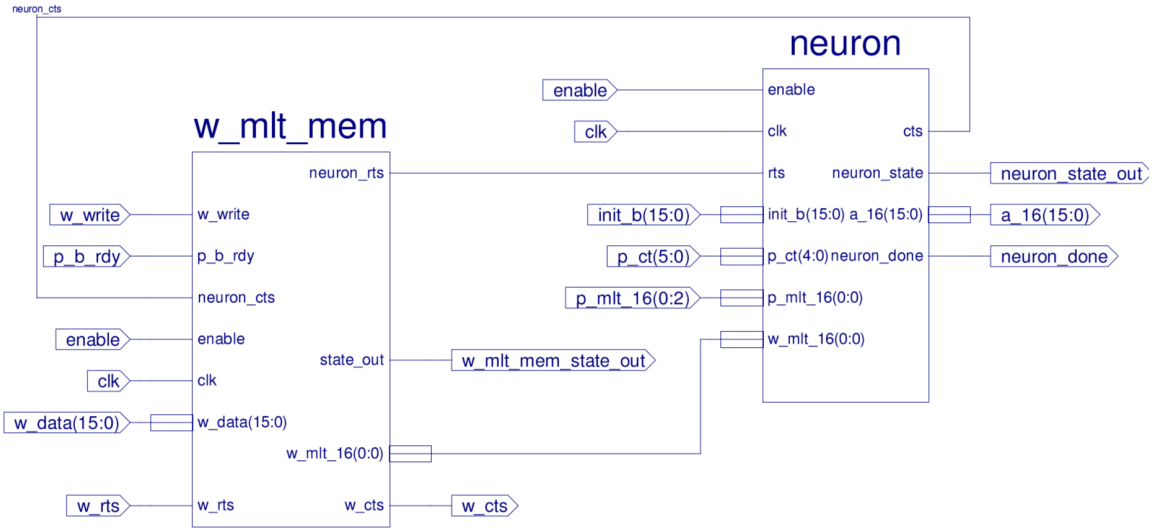

Figure 6.4: The Final Weight ROMs Schematic

The final states of w_mlt_mem are shown in Figure 6.5. They are the same as in Chapter 5, because this level of abstraction contains no changes to the design when adding multiple hardware neurons. The w_mlt_mem module checks the w_write signal to see if values are ready, then stores mlt values into each of the copied Weight ROMs. After MLT values have been saved, the counter is reset in order to start storing from the first Weight ROM again. If values have already been saved, the values are immediately clocked out in parallel by changing the address after each

Table 6.2: Weight ROM New or Important Schematic Signals

Signal	Description
enable	Enables both the w_mem and neuron.
clk	The clock for the modules.
p_b_rdy	Set high when input and bias are ready.
neuron_rts	Set high when inputs, bias, and weights are ready.
neuron_cts	Set high when neuron allows values to change.
w_mlt_16	Array of MLT 16-bit weight values.
w_mem_state_out	The state of the w_mem for the test bench.
w_write	Set high when the Weight ROMs are being written to.
w_data	The data being stored into the Weight ROMs.
w_rts	Set high when w_data is requested to change.
w_cts	Set high when w_data is safe to change.

time the values are loaded.

6.2.3 Changes When Adding the Bias ROMs

A VHDL generate statement can be used to duplicate the hardware neurons in the design. The number of hardware neurons is assigned the variable HWN, which stands for hardware neurons. The VHDL statement is encapsulated in the multi_hardware_neuron block shown in Figure 6.6. Notice that the a outputs are an array of values, denoted a_hwn_16, along with the weight ROM's RTS and CTS signals. This allows for individual communication to each set of Weight ROMs to the loader modules at a higher level of abstraction. Notice that the input signals for w_data are not copied, and neither are the p_mlt_16 input signals. This is because all inputs into the hardware layer will be the same at each input iteration. A table of all new or important values is shown in Table 6.3 for convenience. The b_mhwn_mem module does not change from

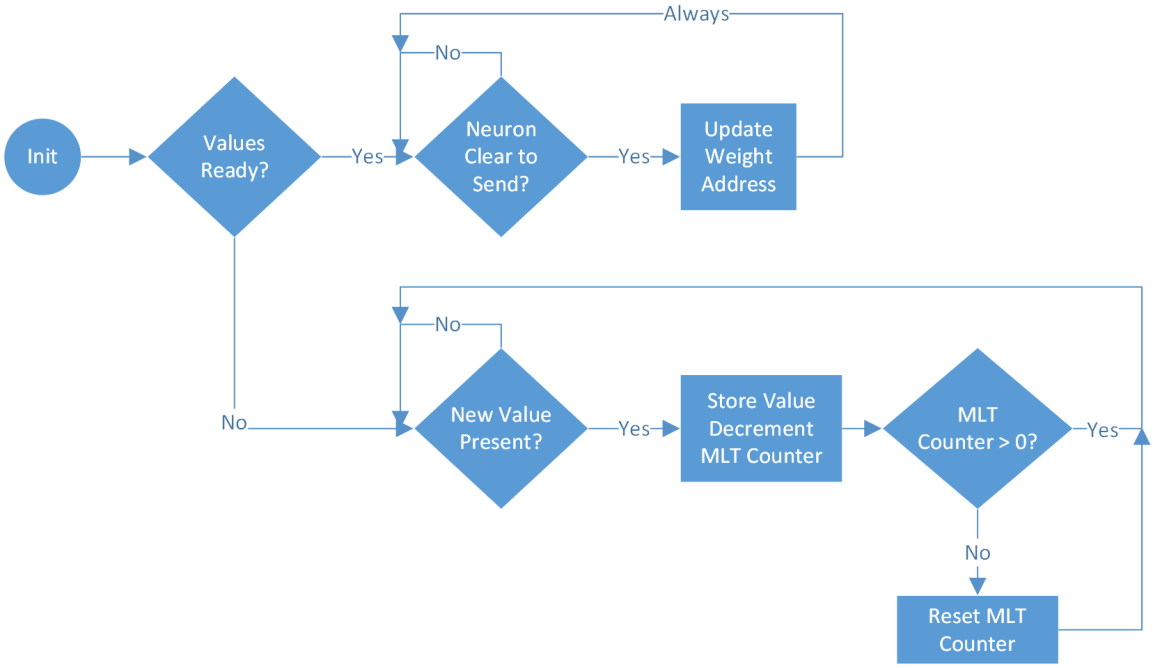

Figure 6.5: The Final w_mlt_mem Flow Diagram

Chapter 4. It contains HWN copies of the nested Bias ROM in order to parallelize the inputs.

The states of the b_mhwn_mem module are the same as in Chapter 4 as well. If the values have been loaded, signaled by b_write being set low, then the values are clocked out in parallel, updating the address for each set of values loaded. However, if b_write is set high, then the values will need to be stored individually into each ROM. This is done by only updating the address signal after HWN values have been loaded. In Figure 6.7, the states for when to store or load values, and when to increment the address, are shown. When loading values, if a new value is present, decrement a counter, if the counter has reached zero, then reset the counter. This is used to enable and disable individual memory modules.

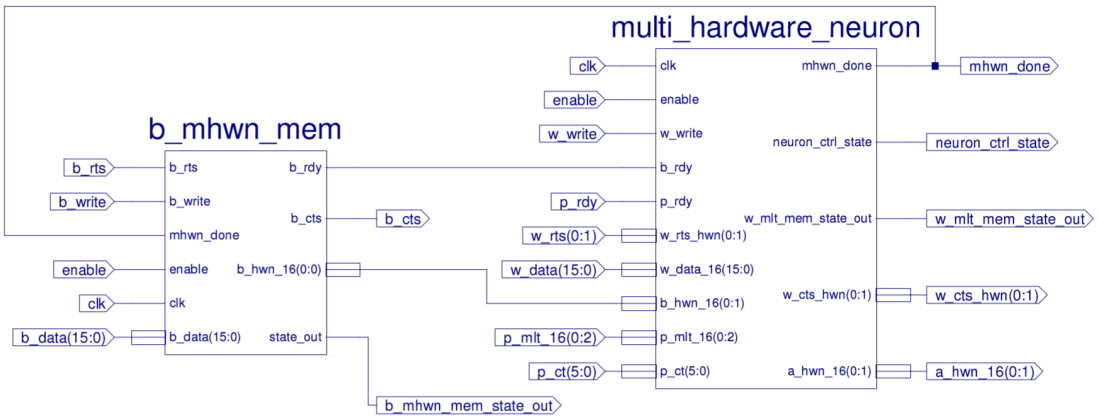

Figure 6.6: The Final Bias ROMs Schematic

6.2.4 Changes to the Network Schematic

The final network schematic, shown in Figure 6.8, shows the parallelized mhwn_b_mem module next to the loader modules and the network control. A more detailed schematic is located in Appendix E, Figure E.4. The mhwn_b_mem module contains the MMIHN layer and the b_mhwn_mem module. This block contains the parallelized hardware layer. The loader modules talk to the memory modules within the hardware layer to set values at initialization. The only major change to the schematic from the one in Chapter 4 is the p_mlt_16 signal going from the a_loader_store module into the mhwn_b_mem module. Table 6.4 contains a description of the new or important signals. The network_control module contains the Structure ROM and the a_loader_store module contains the Input/Output RAM. The b_loader and w_loader contain only one Bias ROM and one Weight ROM that match the description from Chapter 3.

The a_loader_store module needs to accommodate both multiple inputs and multiple outputs. The logic to handle these cases is shown in Figure 6.9. For each input iteration, MLT inputs are serially loaded into an array of wires. When all inputs are ready, the RTS flag is set high. This signals to the neuron that the input values are ready to be input into the neuron. The neuron will handshake back to the loader

Table 6.3: Bias ROM New or Important Schematic Signals

Signal	Description
enable	Enables b_mem and w_hardware_neuron.
clk	Clock for the modules.
p_rdy	Set high when the input value is ready.
b_rdy	Set high when the bias value is ready.
b_hwn_16	Array of HWN 16-bit bias values.
a_hwn_16	Array of HWN 16-bit output values.
mhwn_done	Set high when the output of the multi_hardware_neuron is ready.
b_rts	Set high when b_data is requested to change.
b_cts	Set high when b_data is safe to change.
b_write	Set high when values are being written to Bias ROMs.
b_data	The value being stored into the Bias ROMs.
w_rts_hwn	Array of HWN RTS values for w_hwn_mem.
w_cts_hwn	Array of HWN CTS values for w_hwn_mem.

when it is ready for the next set of values. When all inputs to the neuron have been loaded, the final set of output values need to be stored. First, the remaining S values are checked to prevent bad output values from being saved. Then HWN hardware neuron values are checked. If there are remaining software neurons, but all HWN neurons have been stored, then the layer is not complete, so new values of R and S are not loaded, and the process continues by loading MLT input values again. If there are no longer any software neurons to be saved, then the layer is complete, and new values for R and S can be loaded, which indicates a new layer is to be calculated.

The b_loader will use the number of hardware neurons and the layer software neurons to decide whether to clock in zero values to the nested memory. Figure 6.10 shows this process. If the HWN counter has remaining hardware neurons, then check

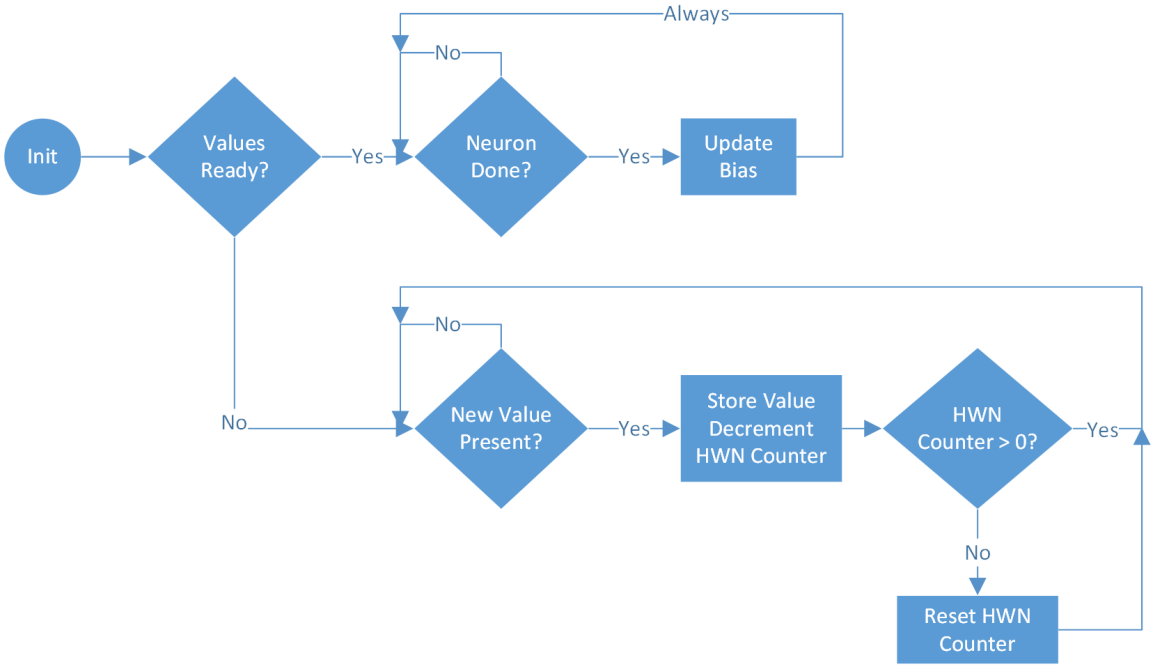

Figure 6.7: The Final Bias ROMs Schematic

if the a_ct counter, the remaining software neurons, requires values to be stored. If there are software neurons, then the value is stored, and the counters each have a value subtracted. If the a_counter runs out of values, then a zero is stored and the HWN counter has a value subtracted. If the HWN counter is zero, but there are values remaining, then the HWN counter is reset. Otherwise the process is complete and a new value for a_ct is loaded.

The w_loader must check for the number of hardware neurons, the number of software neurons, the number of inputs to a hardware neuron, and the number of software inputs to a layer. All values are initialized with a counter. If there is a remaining value, then a value is loaded. When software inputs have been loaded, then zeros begin loading until the MLT counter runs out. When the MLT counter runs out, if software inputs remain, the MLT counter is reset. Otherwise, the MLT counter is reset and the HWN and a_ct counters are decremented. Whenever the a_ct counter runs out of software neurons, MLT zeros will need to be stored. The p_ct

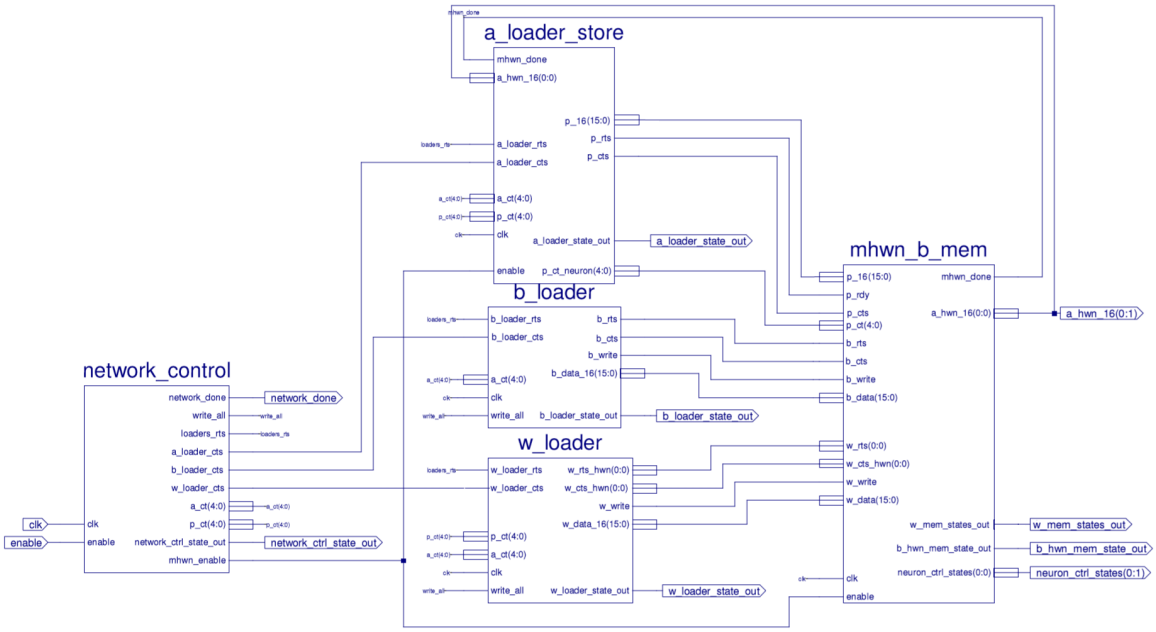

Figure 6.8: The Final Scalable Hardware Neural Network

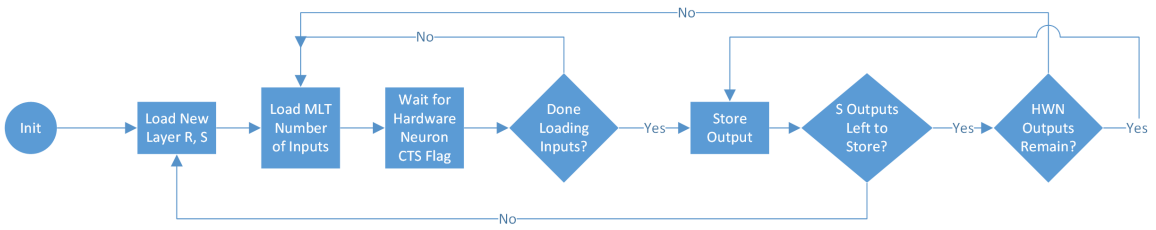

Figure 6.9: The Final a_loader_store Flow Diagram

signal should still be decremented in these states in order to store the correct number of zeros into the memory modules. The HWN counter is decremented after the correct number of zeros have been stored. After HWN is zero, if there are remaining software neurons, the HWN counter is reset, but if there are no more software neurons remaining, new values of a_ct and p_ct are loaded to start a new layer.

Table 6.4: Network Schematic New or Important Signals

Signal	Description
enable	Enables only the network_control.
clk	Clock for the modules.
network_done	Set high when software network has been calculated.
write_all	Set high when writing values to nested memory.
loaders_rts	Set high when a_ct and p_ct requested to change.
a_loader_cts	Set high when a_ct and p_ct safe to change.
a_ct	Software neurons for layer.
p_ct	Software inputs for layer.
network_ctrl_state_out	State of network_control for test bench.
mhwn_enable	Enables the calculation of the hardware layer.
p_mlt_16	Array of MLT 16-bit input values.
a_hwn_16	Array of HWN 16-bit output values.
p_rts	Set high when new input value requested to change.
p_cts	Set high when new input value safe to change.
a_loader_state_out	The state of a_loader_store for test bench.
p_ct_neuron	The number of software inputs for the layer.

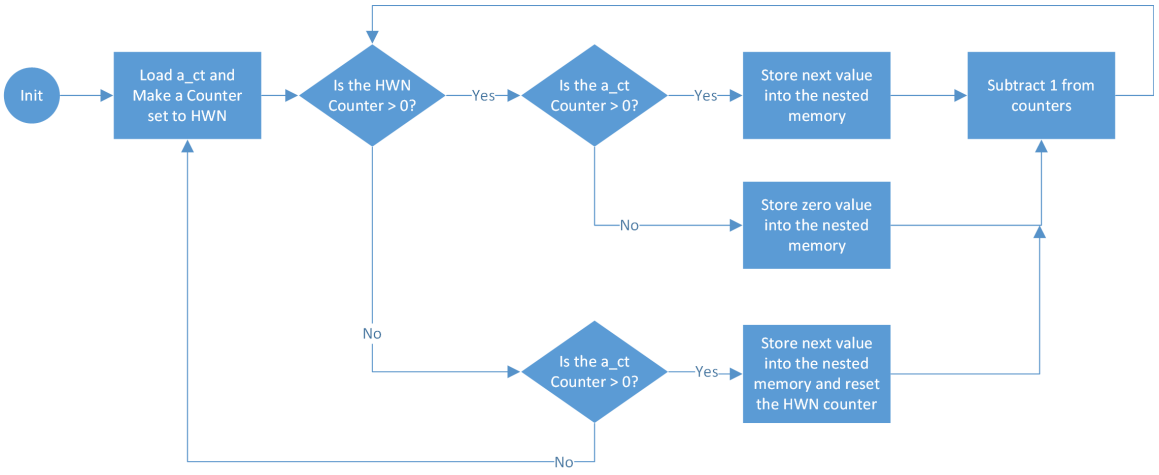

Figure 6.10: The Final b_loader Flow Diagram

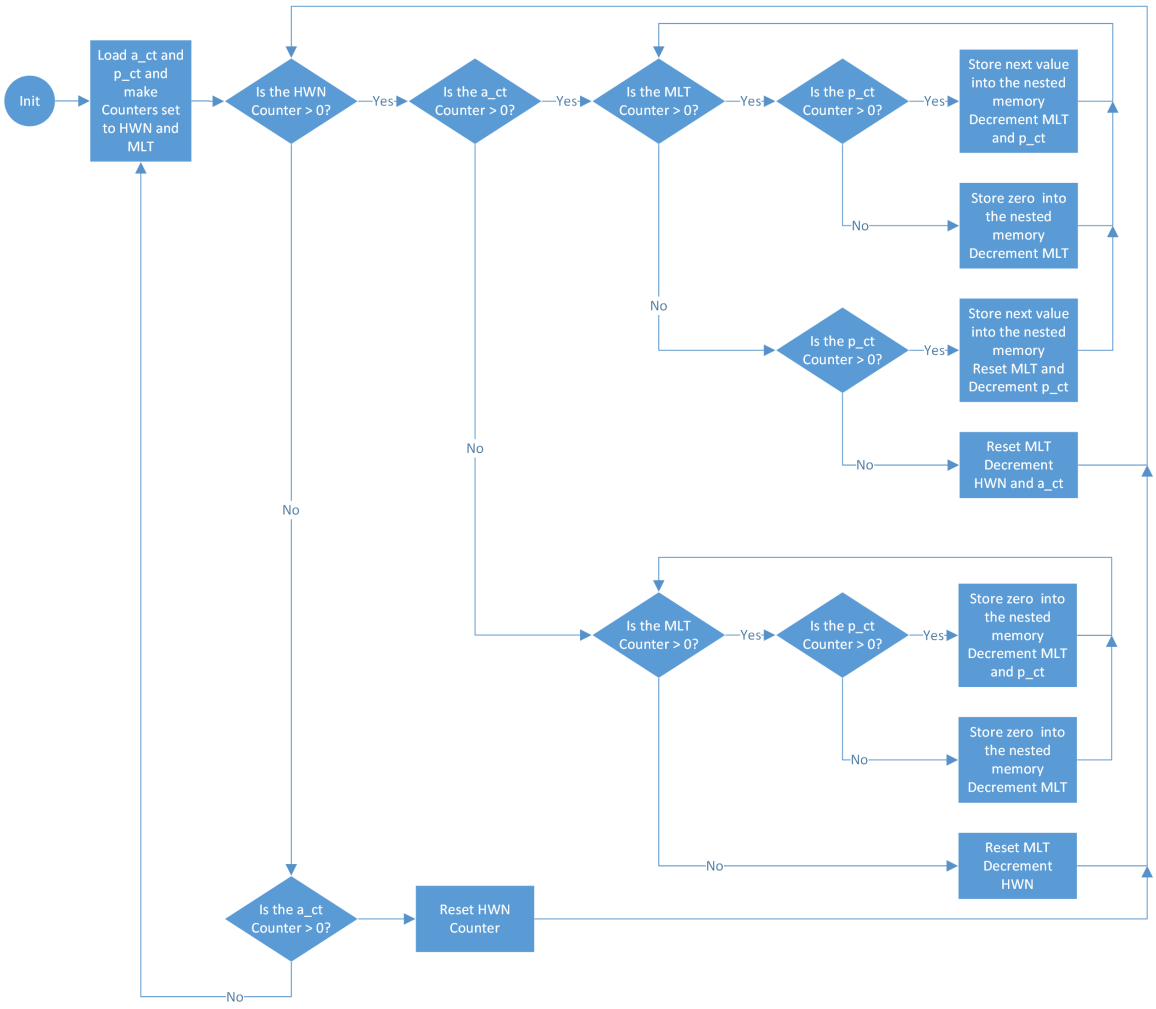

Figure 6.11: The Final w_loader Flow Diagram

6.3 Timing Analysis

Because the outputs are divided by the number of hardware neurons, and the inputs are divided by the number of inputs to the hardware neuron, concepts of input iterations and hardware layer iterations can be used. These calculate the total number of cycles that are required for a given layer. Equations 6.1 and 6.2 show how to calculate the iterations. Input iterations represent the number of times that inputs will need to be loaded into the neuron in order to calculate n. The hardware layer iterations are based on the number of times that all inputs must be reloaded to calculate the set of neuron outputs.

$$HardwareLayerIterations = \left\lceil \frac{S}{HWN} \right\rceil \tag{6.1}$$

$$InputIterations = \left\lceil \frac{R}{MLT} \right\rceil \tag{6.2}$$

All variables, found in Table 6.5, have been described in previous chapters. The variables account for the modules which require different numbers of cycles to transition through their states. By modularizing the design, the cycles can be more easily measured by looking at the state transitions for each module.

Using the variables described, the total cycles required for a layer can be calculated using Equations 6.3 and 6.4. The number of input iterations multiplied by the number of output iterations gives the total number of iterations of the hardware layer. The cycles which are multiplied by this number are the number of cycles that are required for computation at each iteration. Cycles which are multiplied only by the output iterations only occur when a neuron's or set of neurons' outputs are ready to be

Table 6.5: Cycle Variables

Variable	Description
R^n	Number of inputs to the layer n
S^n	Number of neurons of the layer n
C_A	Cycles required for one adder to complete
C_M	Cycles required for one multiplier to complete
C_{LNC}	Cycles of the neuron control logic per input
C_{TF}	Cycles required for the transfer function
C_{LA}	Cycles required for loading a single value
C_{SA}	Cycles required for storing a single output
C_{LS}	Cycles required to load a new structure
C_{SAL}	Cycles required for the logic to begin storing values

stored. The final addition of cycles occur only once per layer.

$$Cycles^1 = \left\lceil \frac{R^1}{MLT} \right\rceil \left\lceil \frac{S^1}{HWN} \right\rceil (C_M + C_A(\lceil Log_2(MLT) \rceil + 1) + C_{LNC} + MLT(C_{LA})) + \left\lceil \frac{S^1}{HWN} \right\rceil (C_{TF} + C_{SAL}) + S^1 C_{SA} + C_{LS} \quad (6.3)$$

$$Cycles^n = \left\lceil \frac{S^{n-1}}{MLT} \right\rceil \left\lceil \frac{S^n}{HWN} \right\rceil (C_M + C_A(\lceil Log_2(MLT) \rceil + 1) + C_{LNC} + MLT(C_{LA})) + \left\lceil \frac{S^n}{HWN} \right\rceil (C_{TF} + C_{SAL}) + S^n C_{SA} + C_{LS} \quad (6.4)$$

6.4 Slice Requirement Analysis

Slices variables, found in Table 6.6, represent each module that exists within the design. These variables change when the size of the network changes. This means that quick calculations cannot be extrapolated exactly. However, they will stay relatively constant and can be used to extrapolate within a margin of error.

Table 6.6: Slice Variables

Variable	Description
S_M	Slices per multiplier
S_A	Slices per adder
S_{TF}	Slices per transfer function
S_W	Slices per weight ROM
S_{WL}	Slices per weight logic
S_{NC}	Slices per neuron control block
S_B	Slices per bias ROM
S_{BL}	Slices per bias logic
S_{PA}	Slices per a_loader_store block
S_{BLD}	Slices per b_loader block
S_{WLD}	Slices per w_loader block
S_{NET}	Slices per network_control block

Because each module is created a certain number of times, the slices required by each module can be multiplied by that component to calculate the total slices. Because the adder tree is completely removed when MLT is 1, Equation 6.6 should be used during that case. Equation 6.5 represents the general case for the size of the structure. This combines multiplying the number of hardware neurons by the size of the MIHN.

$$Slices_{MLT>1} = HWN(MLT(S_M + S_W) + (\frac{\lceil \frac{MLT}{2} \rceil (\lceil \frac{MLT}{2} \rceil + 1)}{2} + 1)S_A + S_{TF} + S_{WL} + S_{NC} + S_B) + S_{BL} + S_{PA} + S_{BLD} + S_{WLD} + S_{NET} \quad (6.5)$$

$$Slices_{MLT=1} = HWN(S_M + S_W + S_A + S_{TF} + S_{WL} + S_{NC} + S_B) + S_{BL} + S_{PA} + S_{BLD} + S_{WLD} + S_{NET} \quad (6.6)$$

6.5 Simulation Testing and Verification in Vivado

Using a test bench, a behavioral simulation can verify that the design works as expected. The same test case as the three previous chapters can be used in order to ensure that, over different hardware sizes, the same calculation can be performed using more area and less time. This test case uses 4 inputs, 10 hidden neurons, and a single last layer neuron. This case will test that the neurons are being split up as expected and that multiple layers can be transitioned correctly. As seen in Table 6.7, all final values for the output layer are correct for all numbers of hardware neurons and all numbers of inputs into the hardware neurons. This scalable network was tested extensively over multiple software networks of different sizes and was found to calculate the correct values in all cases.

Table 6.7: Final Layer Output Across Changing HWN and MLT

a_2^1	$MLT = 1$	$MLT = 2$	$MLT = 3$	$MLT = 4$	$MLT = 5$
$HWN = 1$	$3A7B$	$3A7B$	$3A7B$	$3A7B$	$3A7B$
$HWN = 2$	$3A7B$	$3A7B$	$3A7B$	$3A7B$	$3A7B$
$HWN = 3$	$3A7B$	$3A7B$	$3A7B$	$3A7B$	$3A7B$
$HWN = 4$	$3A7B$	$3A7B$	$3A7B$	$3A7B$	$3A7B$
$HWN = 5$	$3A7B$	$3A7B$	$3A7B$	$3A7B$	$3A7B$

Cycles can be measured from the state transitions in the test bench. The variables needed to compute the number of cycles in Equation 6.3 and 6.4 can be found in Table 6.8. Notice that because the adder tree is not generated when MLT is equal to 1,

the number of cycles for the neuron_ctrl module is one cycle less. This is because the neuron_ctrl module waits for the adder tree to reset before starting the next set of inputs which takes one cycle. All other values are not changed from the previous chapters.

Table 6.8: Values of Cycle Variables

Variable	Value	Description
R^1	4	Number of inputs to the layer 1
S^1	10	Number of neurons of the layer 1
S^2	1	Number of neurons of the layer 2
C_A	8	Cycles required for one adder to complete
C_M	6	Cycles required for one multiplier to complete
$C_{LNC}^{MLT=1}$	8	Cycles of the neuron control logic per input iteration
$C_{LNC}^{MLT>1}$	9	Cycles of the neuron control logic per input iteration
C_{TF}	3	Cycles required for the transfer function
C_{LA}	1	Cycles required for loading a single output
C_{SA}	1	Cycles required to store a single value
C_{SAL}	4	Cycles required for the logic to begin storing values
C_{LS}	1	Cycles required to load a new structure

Plugging the measured cycles into Equations 6.3, the number of cycles can be calculated as shown in Table 6.9. These are the values measured and calculated for the software network. The equations match the measured values exactly, and so can be used to extrapolate over larger hardware networks. Notice that the number of cycles required decreases more quickly when adding hardware neurons than when increasing the number of inputs.

Extrapolating these equations over a software network of 23 inputs and 23 software-neurons, Figure 6.12 plots the speedup as the number of inputs to a hardware-neuron

Table 6.9: Measured and Calculated Cycle Requirements Across HWN and MLT

Cycles	$MLT = 1$	$MLT = 2$	$MLT = 3$	$MLT = 4$	$MLT = 5$
$HWN = 1$	1240	915	1098	649	714
$HWN = 2$	745	550	643	399	419
$HWN = 3$	646	477	552	360	360
$HWN = 4$	547	404	461	301	301
$HWN = 5$	349	331	370	242	229

and the number of hardware neurons increase. Notice that the speedup as HWN increases is increasing consistently faster than when MLT increases. MLT also causes the network to slow down after another layer of the adder tree is added into the hardware. This should hint that, depending on the software network, the number of hardware neurons should generally be created before the multipliers are added. However, to ensure this is the case, the amount of hardware that is consumed by these cases should be considered too.

As shown in Tables 6.10 and 6.11, the calculated number of slices does not perfectly match the measured number of slices. This is because the variables representing the number of slices for each module change as MLT and HWN change. They are also not constant within the same MLT and HWN. The copied modules consume different amounts of space, even when using the same MLT and HWN. They generally consume about the same amount of space across the cases, but this means that exact values cannot be calculated. Because the calculations for speedup and area cutoff can be very close, it may be critical to ensure that these equations are more accurate. However, because of the difficulty of this problem, using an iterative process, the largest and fastest network could be created despite the inaccuracy of the slice equations.

Because the equations are accurate enough to justify extrapolation, the equations can be used to look at the LUT slices required by a software network of 23 inputs

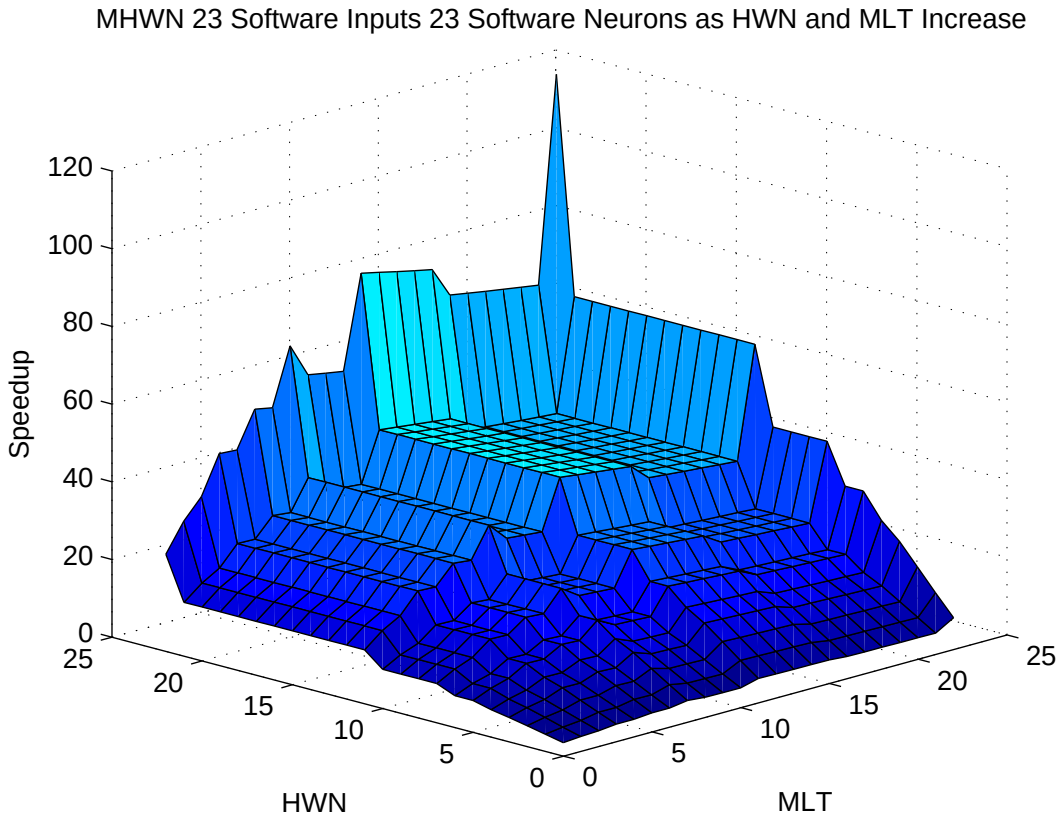

Figure 6.12: Speedup for 23 Input 23 Software Neuron Network

and 23 software neurons as the HWN and MLT values change. Figure 6.13 shows the area as the HWN and MLT vary. Notice that the area required is smooth and has an almost constant gradient. This means that increasing HWN increases the area by roughly the same amount as increasing MLT. The maximum slices could be compared with the speedup table to see which configuration would be most optimal for this case.

Assuming that the nested memory modules in the hardware are sized correctly, the memory space will be approximately constant, instead of being multiplied by HWN or MLT. The calculated slices can be used to ignore the points on the speedup graph that are unobtainable. Using the LUT slices with a maximum value of 16000, Figure 6.14 shows the number of slices that can be achieved for this structure. The

Table 6.10: Measured LUT Slice Requirements Across HWN and MLT

Slices	$MLT = 1$	$MLT = 2$	$MLT = 3$	$MLT = 4$	$MLT = 5$
$HWN = 1$	2521	4132	6095	7633	9684
$HWN = 2$	4514	7941	11658	14553	18674
$HWN = 3$	6519	11579	17133	21477	27620
$HWN = 4$	8563	15174	22598	28537	36579
$HWN = 5$	10402	18844	28077	35743	45513

Table 6.11: Calculated LUT Slice Requirements Across HWN and MLT

Slices	$MLT = 1$	$MLT = 2$	$MLT = 3$	$MLT = 4$	$MLT = 5$
$HWN = 1$	2521	4172	5912	7385	9303
$HWN = 2$	4469	7771	11251	14197	18033
$HWN = 3$	6417	11370	16590	21009	26763
$HWN = 4$	8365	14969	21929	27821	35493
$HWN = 5$	10313	18568	27268	34633	44223

maximum speedup, 22.8, is found when setting HWN to 12 and MLT to 4. This speedup is almost the same as setting HWN to 23, a speedup of 22.56. Because adding inputs uses slightly less space than adding an entire neuron, there are points along the speedup chart that can be obtained which provide an optimal speedup for the area required. The structure of the neuron could be changed so that some neurons would not include a transfer function, which would change the finding. The spike in the center of the graph represents the largest speedup. Other values are set to zero to more clearly see where the maximum speedup is. The second highest speedup is the spike at the edge of the graph. More designs can be considered by modifying the equations.

If fixed-point, instead of floating-point calculations were used, the adders could be

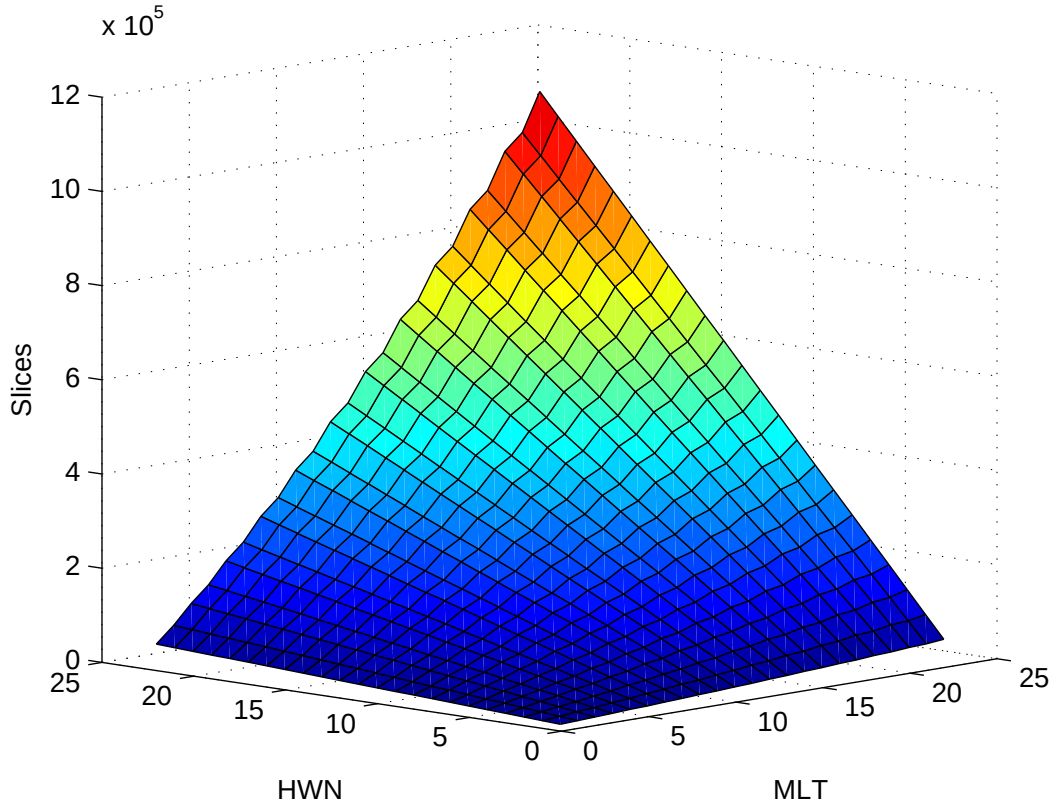

Figure 6.13: Speedup as MLT and HWN Increase for 23 Input 23 Software Neuron Network

completed in a single cycle, so the speedup for adding inputs could rival the speedup for adding hardware neurons. Figure 6.15 shows the maximum speedup for this case. The number of LUT slices decreases dramatically for the fixed-point case, because the DSP slices can be used. Notice that the available area for this case is significantly greater than in the floating-point case. This allows for many more multipliers to be instantiated. However, this design would require 276 DSP slices at the maximum peak. Because the board only has only 240 DSP slices, the multipliers would need to be loaded off to the LUT slices. Because fixed point multipliers use about 400 LUT slices each, and 36 would need to be instantiated, there is not enough space on the board for the maximum speedup in this graph. However, the next smallest peak is at 23 hardware neurons and 8 inputs, which would be achievable. So, for fixed-point,

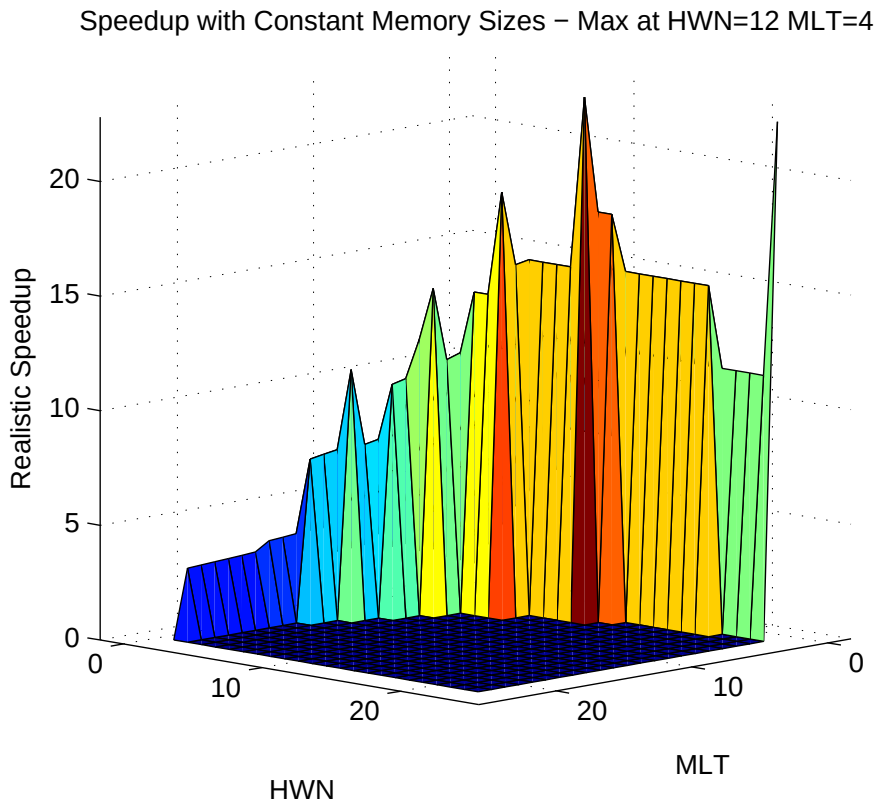

Figure 6.14: Achievable Speedup for 23 Input 23 Software Neuron Network

the speedup could be more than double the speedup of the floating-point case. It is interesting that, for this software network, it is more efficient to use fewer multipliers because of the adder tree and the way the inputs are loaded into the network. An optimization could be made by loading inputs into the network in parallel while the neuron output is being calculated.

6.6 Summary for the Multiple Multi-Input Hardware Neurons

This chapter outlined the final steps in creating a fully configurable scalable hardware neural network and provided equations to analyze the size and cycle requirements of the architecture. Using two variables, MLT and HWN, the number of inputs to the neuron and the number of hardware neurons can be easily changed. This, combined

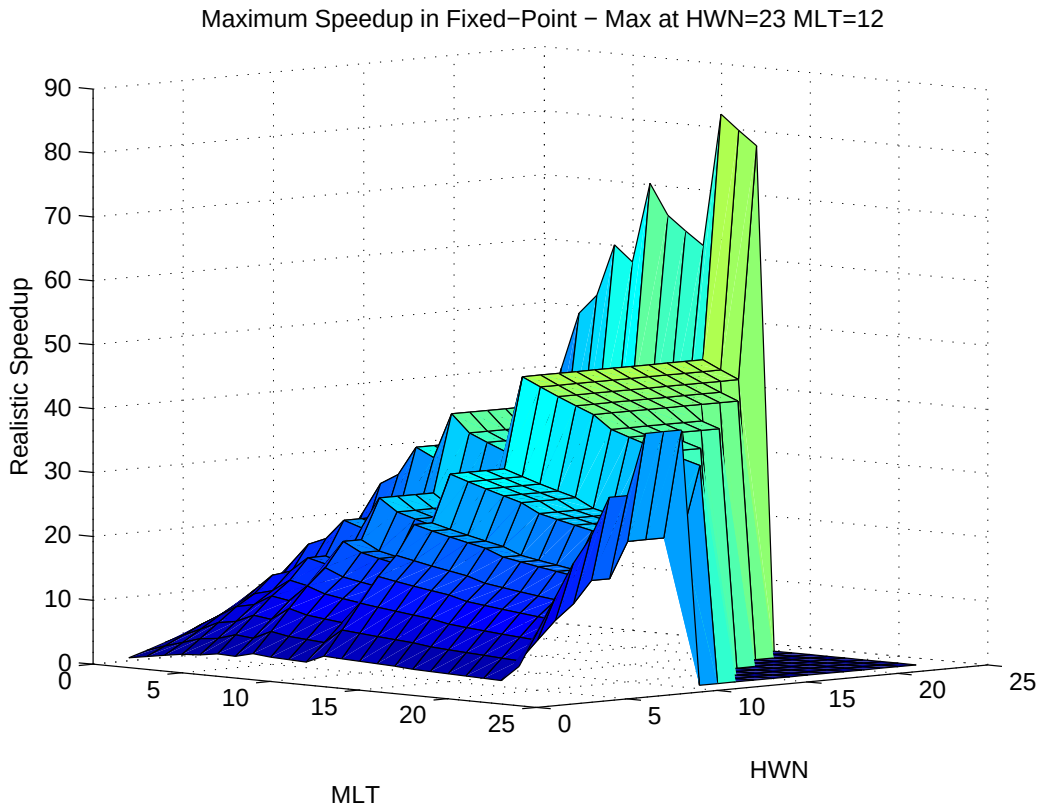

Figure 6.15: Speedup for 23 Input 23 Software Neuron Network - Fixed Point

with the verification that the calculations produce valid results, shows that the design produces valid output and is scalable. By using a modular approach, each module can be encapsulated to represent a particular piece of the project that can be verified. Test benches were created for each module and verified during the design process in order to make debugging simpler. Working up from the neuron and slowly adding each higher abstraction module, simple schematics can be made to show small parts of the project. Using the equations for expected cycles and slice requirements, along with measured variables, results can be extrapolated to see how the network would behave in different conditions.

CHAPTER 7

Summary

The objective of this research has been to design the most efficient reconfigurable implementation of a neural network on an FPGA. When power consumption on an FPGA is not a concern, utilizing the maximum amount of resources for parallelization can result in speedup of the computation of a neural network. Making the network reconfigurable implies that any multilayer network architecture can be implemented using any available FPGA resources. In addition, the VHDL code should be able to be modified to accommodate different numbers of bits for accuracy and be able to use different bit-representations, like fixed-point or floating-point. Measuring the number of cycles needed for each section allows for the equations to be relevant within the same architecture, no matter the size or representation. Being able to modify the hardware for speedup and space allows for the most efficient hardware implementation to be described for any given software neural network. This document described the multiple stages involved in developing the fully scalable and reconfigurable hardware neural network.

7.1 Accomplishments

The single-input hardware neuron (SIHN) is the most simple hardware description that can compute a general multi-layer perceptron network. Creating this structure forms the base for expanding the number of inputs and the number of neurons for the hardware layer. The SIHN was implemented in VHDL using Xilinx Vivado on an Artix-7 100-t FPGA. It was tested for a variety of neural network architectures,

and the accuracy was verified. Equations were developed and verified to predict the number of clock cycles required to perform the neural network computation and the amount of FPGA resources required to implement the SIHN. These equations formed the basis for the multi-input and multiple neuron cases.

The multiple-single-input hardware neurons (MSIHN) case was an extension of the SIHN case. The design uses a VHDL generate statement to dynamically copy the SIHN so that multiple neurons can be calculated at once. This allows for the hardware to take advantage of parallelism when the software network requires multiple software neurons in a layer. Parallelism is achieved by dividing the rows of the weight matrices among the hardware neurons. To maximize parallelism of the structure, all components, including the hardware neuron, are copied. This allows for future work to implement optimizations in special cases for additional speedup. Because the matrix is being split up as the hardware neurons are increased, the weight and bias ROM memory sizes should decrease. Memory sizes were not decreased during this research, because regeneration of the memory cores would take too much time. Using the idea of hardware layer iterations, the equations for the number of cycles and slices from the SIHN case were modified to create new equations that change based on the number of hardware neuron outputs. It was found that the speedup obtained using MSIHN was almost linear, until the number of hardware neurons exceeded the number of software neurons. It was also found that the number of required slices changes for the modules with state machines when the number of neurons is increased, which means that extrapolating hardware sizes will not be exact.

The multi-input (or multiple multiplier) hardware neuron (MIHN) was designed in order to take advantage of parallelism by dividing the inputs. This corresponds to dividing the columns of the weight matrices among the multipliers. An additional adder tree was used in order to sum the results of the multiple multipliers. The adder tree causes delay and sometimes consumes more area compared to increasing

the number of neurons. The SIHN equations were modified to accommodate the additional cycles and slices required for the new MIHN design. In part because of the requirement for the adder tree, the speedup achieved by the MIHN was well below the ideal linear speedup.

The multiple multi-input hardware neurons (MMIHN) design was created using a combination of the previous two cases by dividing the inputs and the software neurons (columns and rows of the weight matrix). Each input requires one multiplier, then each neuron requires one full set of input multipliers. Equations for the number of slices were developed using the concept of input iterations and hardware layer iterations. The input iterations are decreased by the number multipliers per hardware neuron, while the number of hardware layer iterations are decreased by the number of hardware neurons. As described earlier, increasing the number of hardware neurons produces better speedup than increasing the number of multipliers, although this will depend to some extent on the specific software network being computed.

7.2 Conclusions

Comparing the values calculated for speedup to the number of slices required, the maximum space on the board can be used to find the largest cases that can be created. Using the speedup values and the size equations, it was found that the conditions for maximum speedup change dramatically based on the software network being constructed. There is a trade-off between the number of hardware neurons (HWN) and the number of multipliers (MLT), because increasing MLT increases speedup more slowly, but uses less area. This means that some cases use area more efficiently and can maximize the speedup by utilizing parallelization in ways that could not be achieved by only copying the number of neurons. Also, because a given software network may not always have equal numbers of inputs and outputs, it is useful to be able

to fully tailor the hardware design to the application. However, because the speedup is more consistent when adding hardware neurons, if there exists enough hardware to add the maximum amount of hardware neurons, this will generally produce the fastest speedup. So, inputs to the neuron are generally a lower priority, which makes sense because of the delay and area cost of the adder tree.

7.3 Future Work

The accuracy of the algorithm could be improved by either increasing the number of bits in the floating-point representation, or using fixed-point instead. The floating-point values can prevent overflow in the calculation of n, but will lose accuracy when the exponent value is shifted. In order to prevent overflow while calculating n, the number of bits for the adder tree can be increased by 1 during each stage, and then all adders can be increased by the maximum number of input iterations for the given software network. Alternatively, the adder tree could be replaced with a block-carry-look-ahead (BCLG) adder, which would add each component, then have a final carry propagate adder at the end. Adding the extra internal bits for accuracy will consume significant area, but will still consume less space than the equivalent floating-point network. The fixed-point multipliers use significantly less space as well, and so the multipliers could be moved to any of the 240 DSP slices to make more room on the board for more control logic or memory space. More information on the trade-offs between floating and fixed point are given in Appendix A.

Design changes could be made to benefit the overall network design. The sizes of the memory do not automatically scale to the size that should be required for a given number of hardware or software neurons. The sizes of other memory modules for the Weight and Bias ROMs are also based on the software network. The logic which loads values could also be optimized to minimize the area for each of the ROMs. This could be done with a scripting tool like python, or ideally a custom VHDL module could

be made. An optimization here seems very feasible, because the memory is currently single cycle access. If the ROMs were moved to the block RAM to save even more space, then the core modules may have to be automatically generated. Also, if the hardware layer is equal to the largest number of software layer inputs and software neurons, then the storing stage could be skipped, speeding up computation.

Another type of neural network can be implemented with some modifications to the current design. A recurrent network is similar to a multi-layer perceptron network, but uses feedback and tapped-delay-lines (TDL) in order to model dynamic systems. All TDLs of the network can be created in hardware using a single shift register, assuming that all time steps are equal in duration. The connections between the layers of a recurrent network could be precalculated and stored into a set of memory modules. The modules could be incremented by one address at a time, loading the address of each input and the address to store each output. This would be expensive, but would provide the maximum amount of flexibility. In order to account for multiple connections between layers, a single output value may need to be stored more than one time. The scalability of the network may not map very well to the physical world, because the time step would need to be known beforehand. However, it should be possible to do the computation with enough precalculation and memory. A Matlab simulation was created to test this process and was able to calculate a NARX network (a type of recurrent network), so the calculation in hardware is feasible.

BIBLIOGRAPHY

[1] J. Zhu, G. J. Milne, and B. Gunther, "Towards an fpga based reconfigurable computing environment for neural network implementations," 1999.

[2] C. Latino, M. A. Moreno-Armendariz, and M. Hagan, "Realizing general mlp networks with minimal fpga resources," in *Neural Networks, 2009. IJCNN 2009. International Joint Conference on*, pp. 1722–1729, IEEE, 2009.

[3] A. Gomperts, A. Ukil, and F. Zurfluh, "Development and implementation of parameterized fpga-based general purpose neural networks for online applications," *Industrial Informatics, IEEE Transactions on*, vol. 7, no. 1, pp. 78–89, 2011.

[4] A. Gomperts, A. Ukil, and F. Zurfluh, "Implementation of neural network on parameterized fpga.," in *AAAI Spring Symposium: Embedded Reasoning*, 2010.

[5] E. M. Ortigosa, A. Cañas, E. Ros, P. M. Ortigosa, S. Mota, and J. Díaz, "Hardware description of multi-layer perceptrons with different abstraction levels," *Microprocessors and Microsystems*, vol. 30, no. 7, pp. 435–444, 2006.

[6] S. Himavathi, D. Anitha, and A. Muthuramalingam, "Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization," *Neural Networks, IEEE Transactions on*, vol. 18, no. 3, pp. 880–888, 2007.

[7] A. W. Savich, M. Moussa, and S. Areibi, "The impact of arithmetic representation on implementing mlp-bp on fpgas: A study," *Neural Networks, IEEE Transactions on*, vol. 18, no. 1, pp. 240–252, 2007.

[8] E. Z. Mohammed and H. K. Ali, "Hardware implementation of artificial neural network using field programmable gate array," *International Journal of Computer Theory and Engineering*, vol. 5, no. 5, 2013.

[9] M. A. Çavuşlu, C. Karakuzu, and S. Şahin, "Neural network hardware implementation using fpga," in *ISEECE 2006 3rd international symposium on electrical, electronic and computer engineering symposium proceedings. TRNC, Nicosia*, pp. 287–290, 2006.

[10] S. Jung and S. S. Kim, "Hardware implementation of a real-time neural network controller with a dsp and an fpga for nonlinear systems," *Industrial Electronics, IEEE Transactions on*, vol. 54, no. 1, pp. 265–271, 2007.

[11] X. Li, M. Moussa, and S. Areibi, "Arithmetic formats for implementing artificial neural networks on fpgas," *Electrical and Computer Engineering, Canadian Journal of*, vol. 31, no. 1, pp. 31–40, 2006.

[12] J. Zhu and P. Sutton, "Fpga implementations of neural networks–a survey of a decade of progress," in *Field Programmable Logic and Application*, pp. 1062–1066, Springer, 2003.

[13] A. R. Lopes and G. A. Constantinides, "A fused hybrid floating-point and fixed-point dot-product for fpgas," in *Reconfigurable Computing: Architectures, Tools and Applications*, pp. 157–168, Springer, 2010.

[14] L. Zhuo and V. K. Prasanna, "Scalable and modular algorithms for floating-point matrix multiplication on fpgas," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, p. 92, IEEE, 2004.

[15] J.-W. Jang, S. B. Choi, and V. K. Prasanna, "Energy-and time-efficient matrix multiplication on fpgas," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 13, no. 11, pp. 1305–1319, 2005.

[16] H. B. Demuth, M. H. Beale, O. De Jess, and M. T. Hagan, *Neural network design*. Martin Hagan, 2014.

[17] A. R. Omondi and J. C. Rajapakse, *FPGA implementations of neural networks*, vol. 365. Springer, 2006.

[18] D. J. Smith, "Vhdl and verilog compared and contrasted-plus modeled example written in vhdl, verilog and c," in *Design Automation Conference Proceedings 1996, 33rd*, pp. 771–776, IEEE, 1996.

[19] Xilinx, "Artix®-7 product table." `http://www.xilinx.com/publications/prod_mktg/Artix7-Product-Table.pdf`, 2015. [Online; accessed 8-March-2015].

[20] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

[21] A. Savran and S. Ünsal, "Hardware implementation of a feed forward neural network using fpgas," in *The third International Conference on Electrical and Electronics Engineering (ELECO 2003)*, pp. 3–7, 2003.

[22] S. Sahin, Y. Becerikli, and S. Yazici, "Neural network implementation in hardware using fpgas," in *Neural Information Processing*, pp. 1105–1112, Springer, 2006.

APPENDIX A

16-Bit Floating and Fixed Slice Comparison

In general, a floating-point multiplier or adder takes more gates than a fixed-point multiplier or adder. This also means that the delay time is longer, and the design will consume more power and be more complex. As shown in Table A.1, from the generated cores, the fixed-point cores consume less space.

Table A.1: 16-bit Fixed-Point and Floating-Point Core Sizes

Core Generated	Fixed Adder	Float Adder		Fixed Mult	Float Mult
LUTs	16	178		416	186
Slice Registers	16	251		31	219

The fixed adder is obviously a significant improvement over the floating adder - more than a factor of ten. However, the multipliers are not as obvious. LUTs are used for quick access, while the slice registers are used for combinational or timing logic. The increased number of Slice registers used for the floating multiplier hints that the logic is more complicated than that of the fixed multiplier. Using the DSP slices can reduce the size even further. However, the floating-point adder cannot use a DSP slice, because we are using a custom number of bits. The sizes of the adders and multipliers can be found in Table A.2.

Using DSP slices reduces both of the fixed-point cores drastically. However, the floating-point adder does not benefit at all, while the floating-point multiplier achieves only a modest improvement. Prior investigation [17, p. 56] has shown that fixed-point implementations can be 12x greater in speed, and over 13x smaller in area. However,

Table A.2: 16-bit Fixed and Floating Core Sizes with DSP Slices

Core Generated	Fixed Adder	Float Adder		Fixed Mult	Float Mult
DSP Slices	1	0		1	1
LUTs	1	178		0	74
Slice Registers	1	251		0	110

saturation error can occur with a fixed-point dot-product calculation [17, p. 56]. A similar problem arises with floating-point calculations, but instead of information being completely destroyed, accuracy is lost.

Because of the values through the transfer function can be trimmed to $\pm 3.\overline{999}$, the fixed-point notation would be more efficient and more accurate. Although, when calculating a dot-product over multiple iterations, if a partial sum saturates the n value, then an addition of a negative number will result in an value with error because n will re-enter the transition region of the tansig function. This can be solved by making the adder, or adders within an adder tree increase in bit range to accommodate the overflow. This may be a viable option considering the size of a floating-point adder logic. In order to fit the largest number of neurons on a board, the fixed-point notation is the obvious choice, but until the saturation problem is solved, the floating-point representation will continue to be used. This is because the goal of the design is to create a scalable neural network, so the size of the hardware can be optimized at a later date.

APPENDIX B

Master and Slave handshaking schemes.

Typically, handshaking pins are used in a master and slave environment where one module will behave more quickly. It is defined as the master, and then when a submodule behaves more slowly, it will be defined as the slave. Masters have the request to send because they are sending data down into the slave. The slaves have clear to send signals so the master knows when it is okay to update the values. The schematics follow a convention that the masters are typically on the righthand side of the schematic, while the slaves are on the lefthand side. This follows a convention that inputs are typically on the lefthand side of schematics, while outputs are typically on the righthand side. Figure B.1 shows a flow chart of a master communicating with a slave.

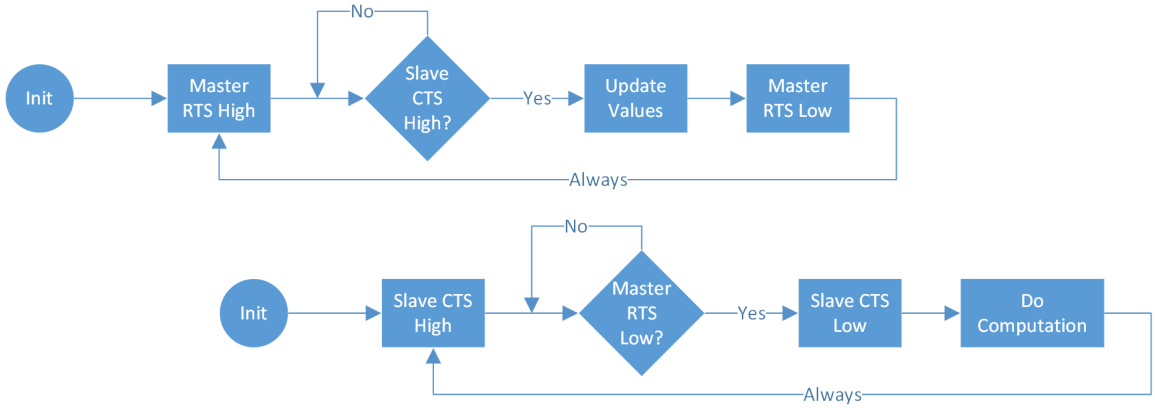

Figure B.1: Master and Slave RTS and CTS Signaling

Notice that because the slave is slower than the master, the master will typically be waiting on the slave's CTS to go high. However, the slave does not want to clock

in bad data, so the master's RTS pin must go low for one cycle in order for the slave to verify that new data has been clocked in. Also, note that this convention uses an active high approach to RTS and CTS handshaking. The neuron is assumed to be the slave in this construct, because it is accepting data from the uppder modules. The modules that send the data to the neuron are considered masters and therefore have the RTS signals and are instantiated above the neuron module abstraction.

APPENDIX C

Tables of Required Cycles

Tables are on the next set of pages.

Table C.1: Calculated Cycles and Speedup for 1 input and 23 Software-Neurons Layer

Hardware Neurons	Linear Speedup	Cycles Required	Calculated Speedup
1	1	714	1
2	2	384	1.859375
3	3	264	2.704545455
4	4	204	3.5
5	5	174	4.103448276
6	6	144	4.958333333
7	7	144	4.958333333
8	8	114	6.263157895
9	9	114	6.263157895
10	10	114	6.263157895
11	11	114	6.263157895
12	12	84	8.5
13	13	84	8.5
14	14	84	8.5
15	15	84	8.5
16	16	84	8.5
17	17	84	8.5
18	18	84	8.5
19	19	84	8.5
20	20	84	8.5
21	21	84	8.5
22	22	84	8.5
23	23	54	13.22222222

Table C.2: Calculated Cycles and Speedup for 23 inputs and 1 Software-Neuron Layer

Hardware Neuron Inputs	Linear Speedup	Calculated Cycles	Calculated Speedup
1	1	5371	1
2	2	3921	1.369803622
3	3	3281	1.637000914
4	4	2481	2.164852882
5	5	2481	2.164852882
6	6	2001	2.684157921
7	7	2001	2.684157921
8	8	1521	3.531229454
9	9	1761	3.049971607
10	10	1761	3.049971607
11	11	1761	3.049971607
12	12	1201	4.472106578
13	13	1201	4.472106578
14	14	1201	4.472106578
15	15	1201	4.472106578
16	16	1201	4.472106578
17	17	1361	3.946362968
18	18	1361	3.946362968
19	19	1361	3.946362968
20	20	1361	3.946362968
21	21	1361	3.946362968
22	22	1361	3.946362968
23	23	721	7.449375867

APPENDIX D

Tables of Measured and Calculated Slice Requirements

Table D.1: Slice Register Requirements by Hardware Neuron Count

Variable	HWN=1	HWN=2	HWN=3	HWN=4	HWN=5
S_M	110	110	110	110	110
S_A	251	251	251	251	251
S_{TF}	37	37	37	37	37
S_W	16	16	16	16	16
S_{WL}	22	22	22	22	22
S_{NC}	54	54	54	54	54
S_B	16	16	16	16	16
S_{BL}	16	33	37	35	36
S_{PA}	148	148	148	147	148
S_{BLD}	69	68	68	68	69
S_{WLD}	95	98	100	102	103
S_{NET}	31	31	31	31	31
$Total_{Measured}$	865	1387	1893	2398	2905
$Total_{Calculated}$	865	1390	1902	2407	2917

$$Total_{Calculated} = HWN(S_M + S_A + S_{TF} + S_W + S_{WL} + S_{NC} + S_B) + S_{BL} + S_{PA} + S_{BLD} + S_{WLD} + S_{NET} \quad \text{(D.1)}$$

APPENDIX E

Detailed Schematics

Schematics are on the next pages.

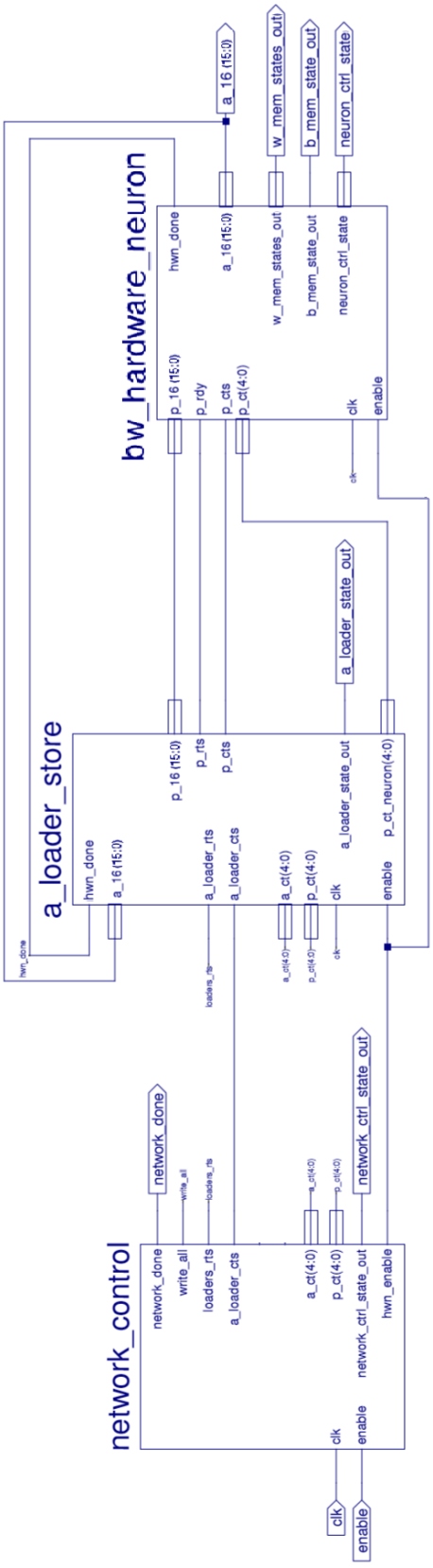

Figure E.1: Schematic of Single Neuron Network

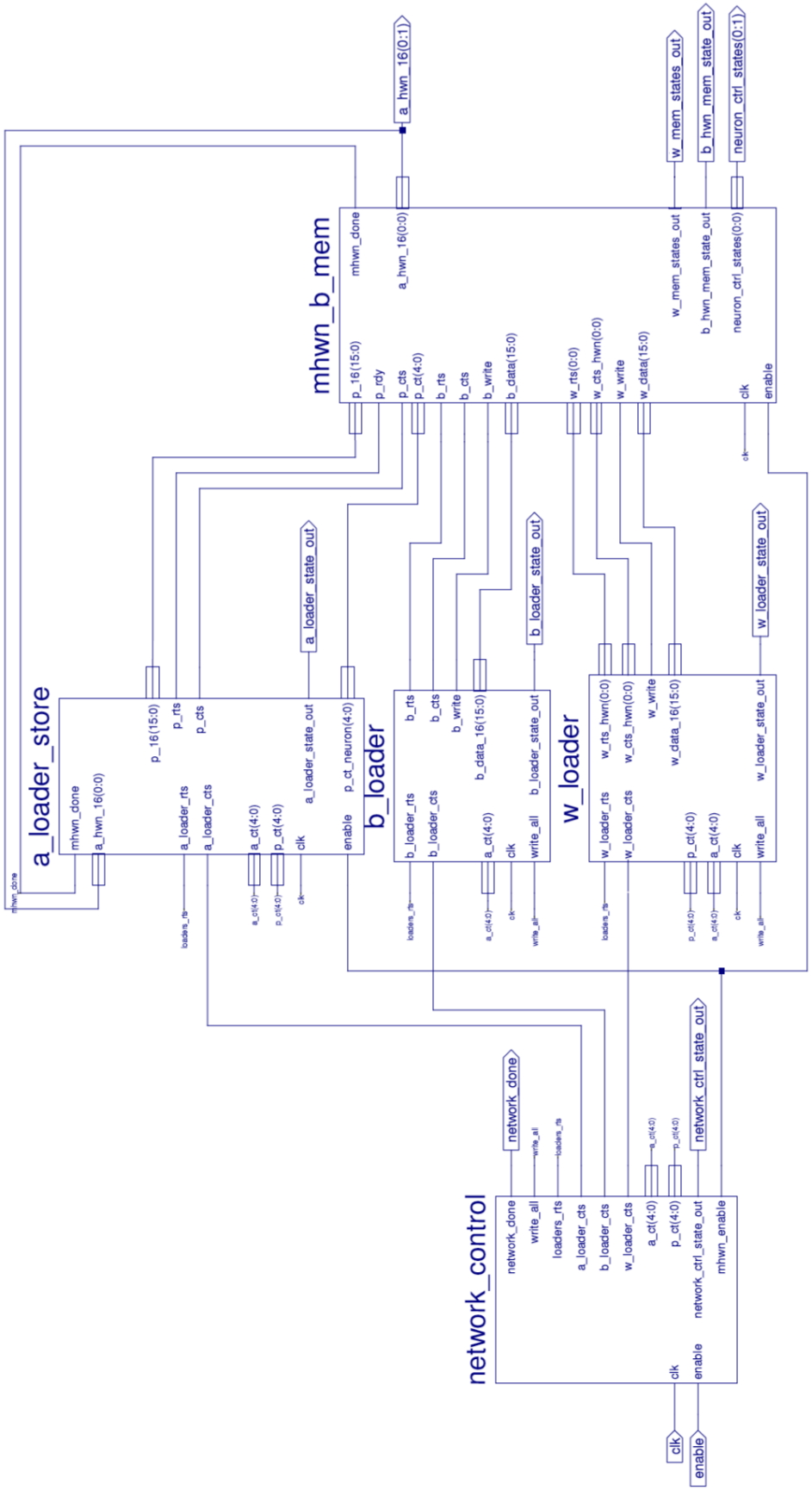

Figure E.2: Detailed Schematic of Single Neuron Network

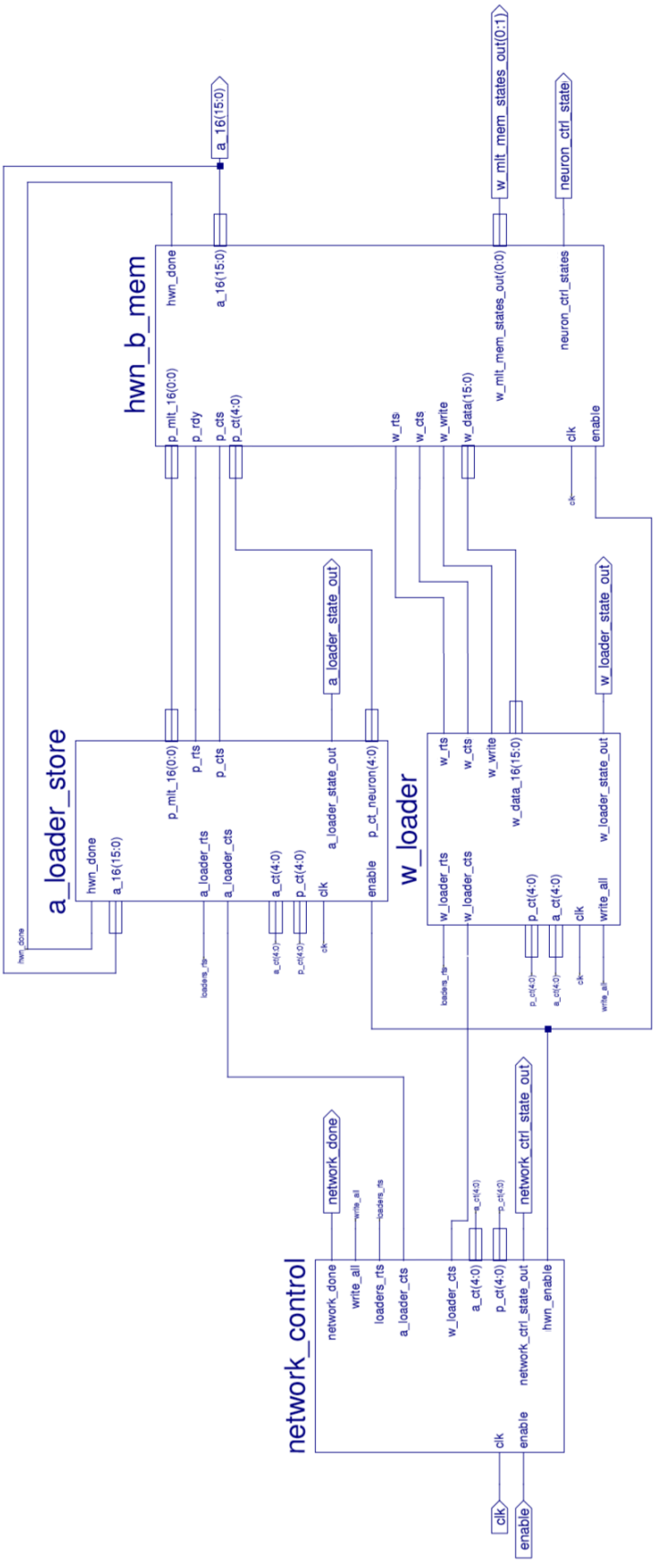

Figure E.3: Schematic of Neural Network Multi-Input Single-Hardware Neuron

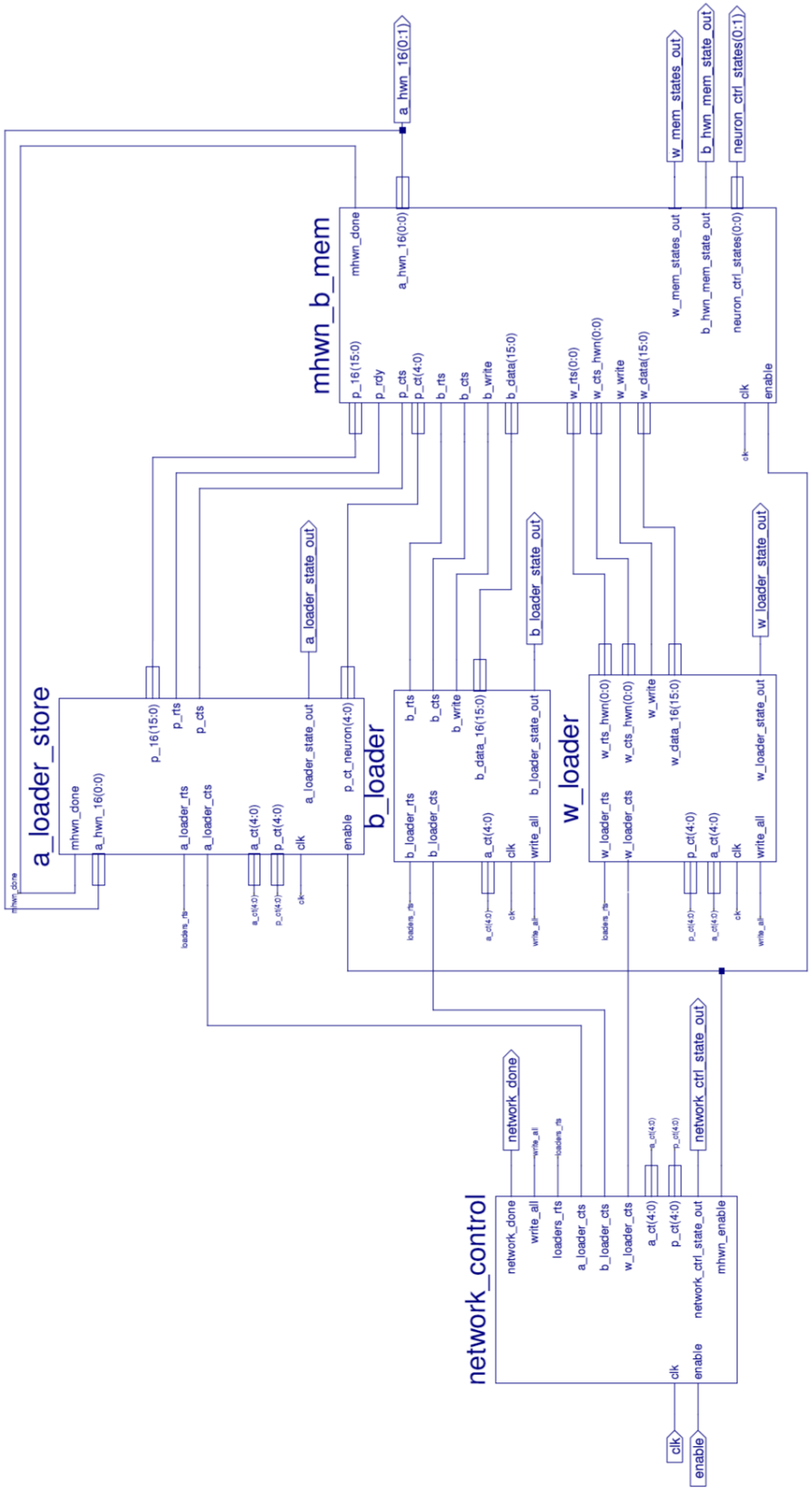

Figure E.4: Schematic of Final Scalable Hardware Neural Network

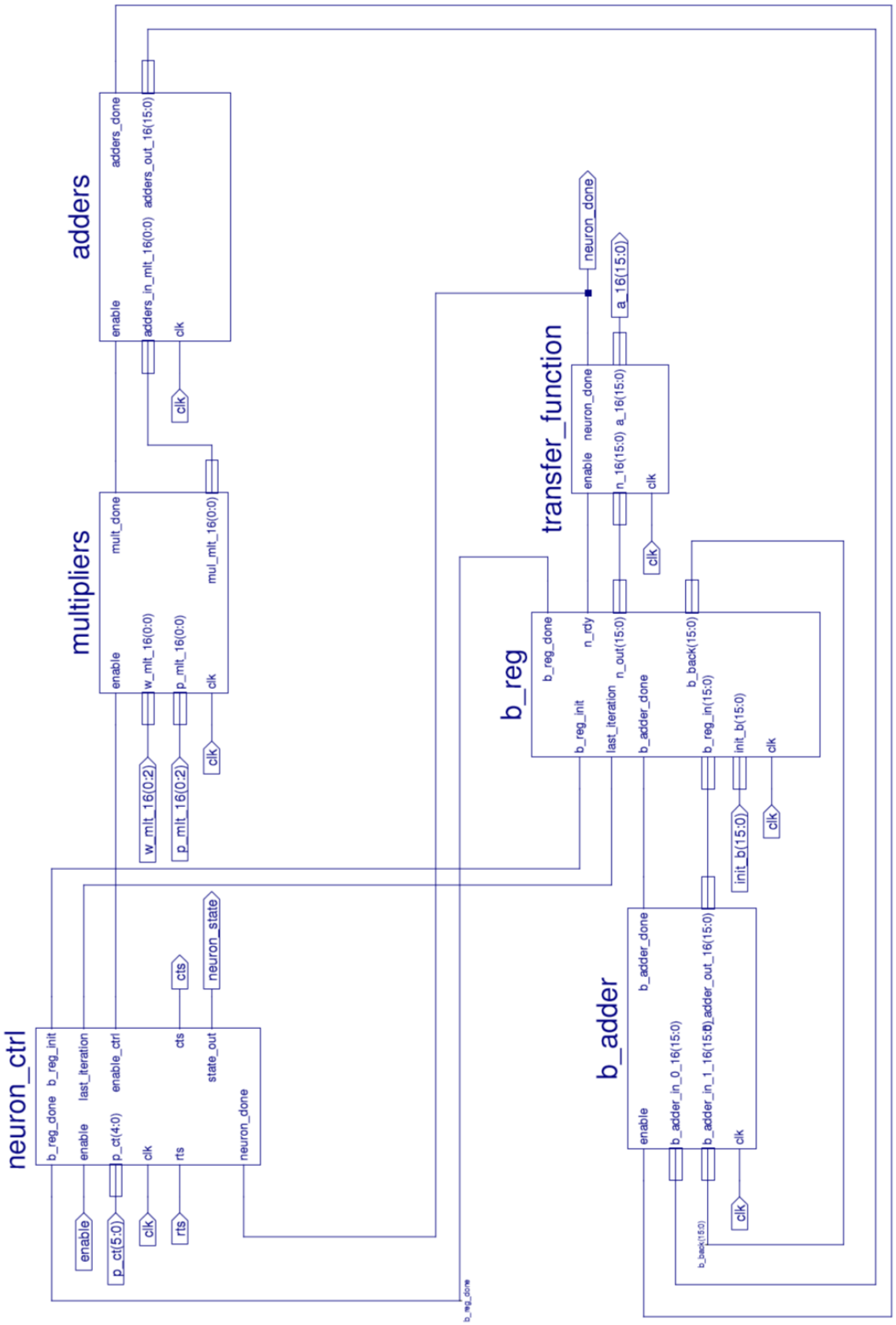

Figure E.5: Schematic of Final Hardware Neuron Schematic

VITA

Evan Williams

Candidate for the Degree of

Master of Science

Dissertation: CREATING SCALABLE NEURAL NETWORKS WITH MAXIMAL FPGA RESOURCES

Major Field: Electrical Engineering

Biographical:

Education:

Completed the requirements for the degree of Master of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in May, 2015.

Completed the requirements for degree of Bachelor of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in December, 2013.

Experience:

Garmin Marine Intern — Software Engineering — Summer 2013

Subsite Division of Charles Machine Works Intern — Electrical and Computer Engineering — Summer 2011, Summer 2012