Implementing the Macgregor point neuron model in a Virtex-II FPGA architecture
by Anthony James Lukes III

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering
Montana State University

Abstract:
Understanding how the brain processes sensory information is one of the end goals for neuroscience research. An important tool that will enable such understanding is the ability to create models of neurons that mimic neural functions. Typically simulations of neurons and networks of neurons are performed using PCs, but it appears that these simulations can be accelerated through the use of Field Programmable Gate Arrays (FPGAs). Specifically, models that can be represented in a fixed-point format without a significant degradation of model integrity are ideal candidates. The Macgregor point model neuron is one such model. It adequately mimics the high level biological function of a single neuron using four state variables: potassium conductance, membrane voltage, threshold, and spike initiation. Mathematically this model is well suited for digital implementation. A hardware MacGregor point neuron state solver was written in Very high speed integrated circuit Hardware Descriptive Language (VHDL) and implemented in a Xilinx Virtex-II FPGA architecture. Performance gains of several orders of magnitude were demonstrated compared to the high performance PCs typically used.

IMPLEMENTING THE MACGREGOR POINT NEURON MODEL IN A VIRTEX-II FPGA ARCHITECTURE

by

Anthony James Lukes III

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

of

Master of Science

in

Electrical Engineering

MONTANA STATE UNIVERSITY
Bozeman, Montana

April 2003

APPROVAL

of a thesis submitted by

Anthony James Lukes III

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

Ross K. Snider Ross Snider 4-21-03
(Signature) Date

Approved for the Department of Electrical and Computer Engineering

James Peterson James N. Peterson 4/21/03
(Signature) Date

Approved for the College of Graduate Studies

Bruce McLeod Bruce R. McLeod 4-22-03
(Signature) Date

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature Anthony James Lukes III

Date 4/21/2003

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Abstract

Understanding how the brain processes sensory information is one of the end goals for neuroscience research. An important tool that will enable such understanding is the ability to create models of neurons that mimic neural functions. Typically simulations of neurons and networks of neurons are performed using PCs, but it appears that these simulations can be accelerated through the use of Field Programmable Gate Arrays (FPGAs). Specifically, models that can be represented in a fixed-point format without a significant degradation of model integrity are ideal candidates. The Macgregor point model neuron is one such model. It adequately mimics the high level biological function of a single neuron using four state variables: potassium conductance, membrane voltage, threshold, and spike initiation. Mathematically this model is well suited for digital implementation. A hardware MacGregor point neuron state solver was written in Very high speed integrated circuit Hardware Descriptive Language (VHDL) and implemented in a Xilinx Virtex-II FPGA architecture. Performance gains of several orders of magnitude were demonstrated compared to the high performance PCs typically used.

INTRODUCTION

Modeling individual neurons as well as networks of neurons is an important part of neuroscience. Depending on the complexity of the neuron model or the complexity of a network this can be a very computationally intensive task (Nelson et al., 1989). In some cases it is possible to accelerate such computations by several orders of magnitude through the use of Field Programmable Gate Arrays (FPGAs). Such devices are programmable, but unlike programming for a fixed hardware, the hardware itself is programmable. Over the last few years FPGA performance has reached a level where it appears to be more feasible to conduct simulations of neuron networks inside the FPGAs as compared with traditional desktop computers. Today's high end FPGA architectures contain vast amounts of logic resources as well as embedded multipliers, blocks of memory, and other complex structures. There are also tools available for taking high level descriptions of systems written in familiar programming languages such as C or Java and then compiling these descriptions into lower level hardware languages such as VHDL or Verilog. This thesis demonstrates a method of implementing the MacGregor's Point Neuron Model (MacGregor, 1987) in a Virtex II using VHDL. MacGregor developed many different neuron models of varying degrees of complexity, but in this thesis the MacGregor Point Neuron Model refers specifically to the Point Neuron Model 10.

Custom hardware has been developed at Montana State University for the purpose of real-time neural modeling, data acquisition, spike sorting, and general purpose signal

processing. This hardware utilizes the pairing of a high performance FPGA with a Digital Signal Processor (DSP). The purpose of the FPGA is to accelerate computationally intensive algorithms and the DSP is used for floating point arithmetic as well as system control. Such a system can acquire and process data directly from analog-to-digital (A/D) converters, avoiding the significant latencies involved with transferring data on and off typical desktop computers. With minimal latency for data acquisition, data can be collected from a live system and processed in the time frame of a propagating action potential. Thus, in response to a given stimulus, the output of both a live system as well as a simulated system could be compared in real time (Figure 1).

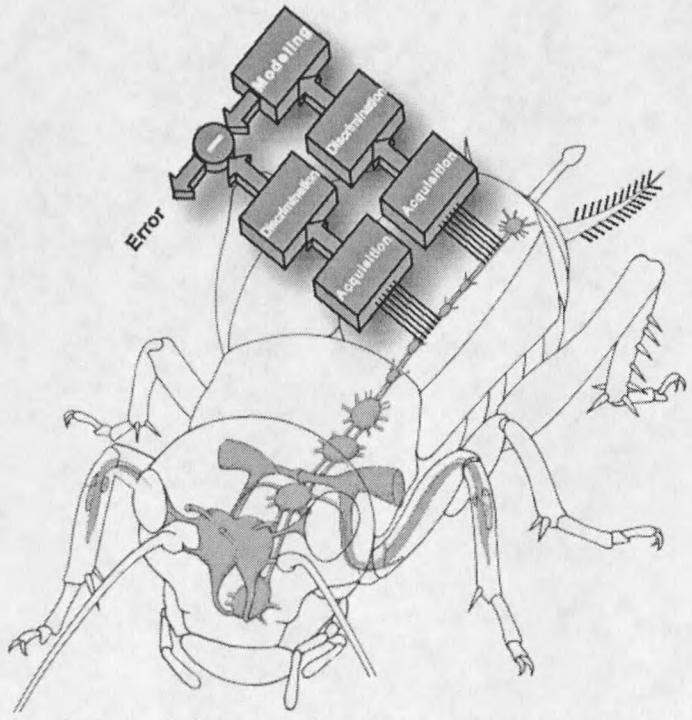

Figure 1 – Online modeling of a cricket ganglia where acquired spike information is fed into a modeled ganglia where the output is compared with the actual output of the cricket ganglia.

To show that FPGAs can provide good platforms for neural modeling, a model of medium complexity was chosen instead of a simple integrate and fire model or a complex compartmental model. The Macgregor Point Neuron Model demonstrates that FPGAs can serve as a platform for performing computations of neuron models and networks. Although relatively simple, the model is able to adequately mimic the high level biological function of a single neuron. Thus, semi-realistic networks of neurons can be assembled to mimic higher level neural systems. Also, the neuron model is not so overwhelmingly complex that using it in place of simple integrate-fire neurons in larger networks is still reasonable. Such a model allows more resources to be allocated for tasks such as implementing more complex networks.

BIOLOGICAL BACKGROUND

What is a Neuron?

The neuron is a basic building block of nervous systems. Its primary function is to receive, process, and send information both electrically and chemically. These interactions are responsible for a nervous system's functionality. Electrical signals called action potentials have the ability to travel significant distances and can travel between neurons.

A typical neuron consists of axons, dendrites, and a cell body often called the soma (Figure 2). The dendrites are tree like structures that behave as inputs to the neuron. Axons act as conduits for signals traveling to other neurons, which tend to be much longer than dendrites. The contact point between an axon of one neuron and a dendrite of another neuron is called a synapse. A synapse contains special mechanisms for allowing chemical signals to propagate between neurons.

Between the axon and the cell body resides a region called the Axon Hillock. In response to the summation of signals entering the dendrites, this region can initiate an action potential that will travel down the axon.

Within each neuron there are many different molecules, proteins, and ions that are used by the neuron to communicate messages to other neurons. In particular, movement of these ions across the cell membrane is responsible for creating the electrical signals

called action potentials that are important for neuron communication. Action potentials are able to travel the long distances that axons span.

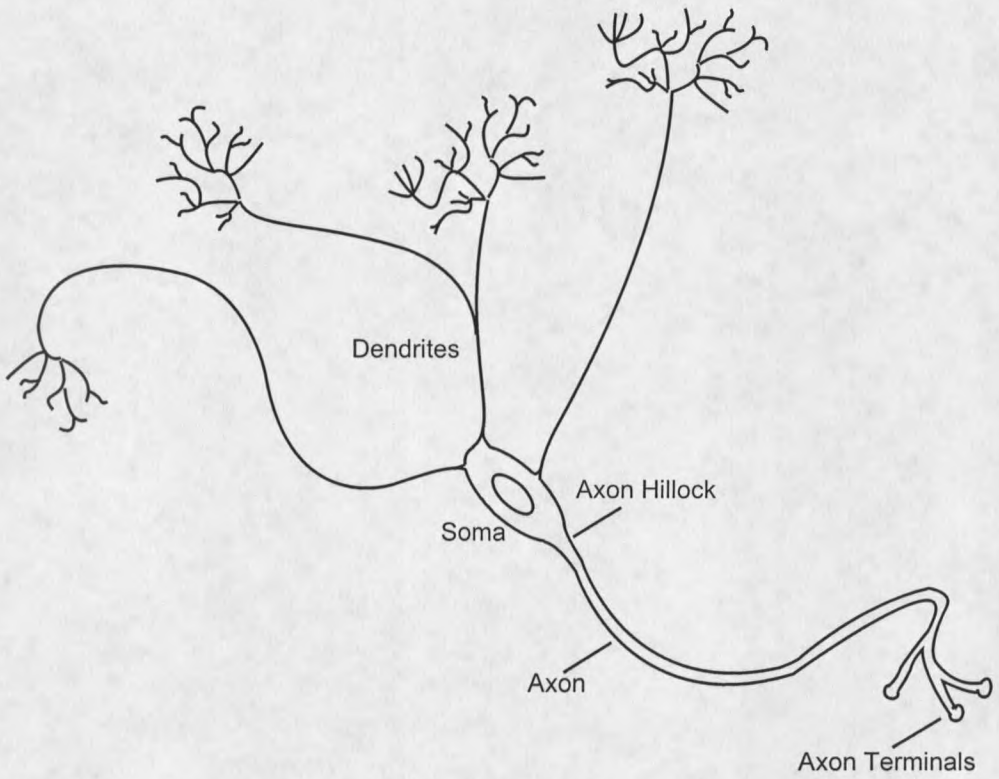

Figure 2 – Diagram of a typical neuron showing the important features such as the axons, dendrites, axon hillock, soma and axon terminals.

Cell Membrane

Before a description of the process involved in the creation and propagation of an action potential can be given, it is first necessary to understand some basic properties of the cell membrane. Within each cell the mobile electric charges tend to repel each other and build up on the inside of cell membrane. The three most important of these mobile

charges are Na+, Ca_2+, and K+. This buildup of charge on the inside of the cellular membrane causes a similar buildup of charge on the outside of cell membrane due to electric forces.

For the most part the cell membrane acts as an insulator, but channels exist that selectively allow passage of certain ions and also actively transport some ions across the membrane by expending energy. Under typical conditions these channels maintain differing concentrations of ions inside and outside the cell resulting in a net negative charge within the cell. This creates a voltage potential across the cell membrane. When describing this built in membrane voltage it is with respect to the extracellular medium.

There are two physical laws responsible for this ionic movement across this cell membrane, diffusion and electrical forces. Diffusion is a thermodynamic process that causes particles to move from higher to lower concentrations. The difference in concentrations of ions inside and outside the cell creates a diffusive flow of ions across the cell membrane. Electrical forces are exerted on charged particles in the presence of an electric field. The electric field across the cell membrane will tend to move positively charged ions into the cell. In order for the membrane to maintain its polarization, current caused by electrical forces must counteract this diffusion current. The Nernst-Plank Equation (Johnston and WU, 1995) is a mathematical description for the ionic current flow across a cell membrane due to this electrical field and concentration gradient (Equation 1). Although there are situations not accounted for by this equation, this model does a reasonable job of accounting for the most important ions such as K, Na, and Ca. In current density form I is expressed as A/cm^2; [C] is concentration of ions

(molecules/cm^3); Na is Avogadro's number; R is the gas constant; F is Faraday's constant; and u is μ/Na or molar mobility.

Equation 1

$$I = -\left(uz^2 F[C] \frac{dV}{dx} + uzRT \frac{d[C]}{dx} \right)$$

The resting or equilibrium potential of the membrane with respect to a particular ion type can be found setting the NPE equation equal to zero (diffusion and electrical currents equal). This results in the Nernst Equation (Equation 2). Thus, using intercellular and extracellular ion concentrations the resting potential for a particular ion across the cell membrane can be found. This is important because almost all neuron models incorporate the resting potential of at least one or more ion species.

Equation 2

$$E = \frac{RT}{Fz} \ln\left(\frac{[outside]}{[inside]} \right)$$

At this stage the electrical properties of a neuron can be modeled by an equivalent circuit (Figure 3). Since the cell membrane acts as an insulator it can be modeled as a capacitor with capacitance Cm. This capacitance is directly related to the surface area (Cm=10nF/mm^2) of the cell and is approximately the same for all neurons. The channels within the membrane act as resistances that are typically dependent upon membrane

voltage and time. These channels also exhibit resting potentials for particular ions which can be modeled as voltage sources. R represents the resistance of the membrane to a particular ion and E represents the resting potential of the membrane for a particular ion.

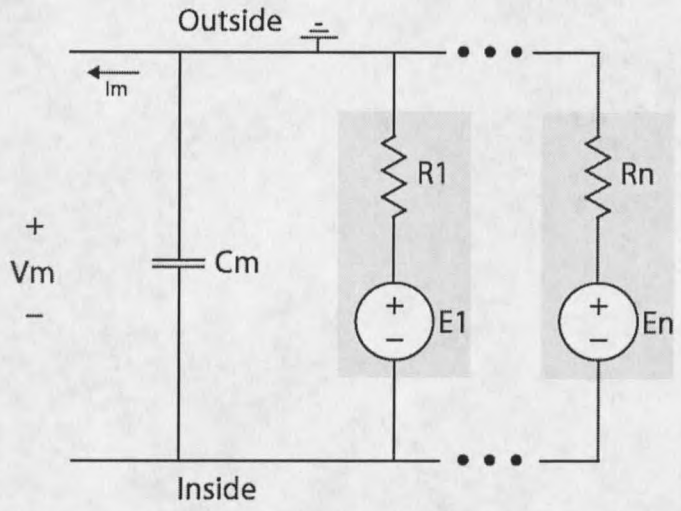

Figure 3 – Equivalent circuit description of cell membrane where V_M represents membrane voltage, C_M represents membrane capacitance, Rn represents the resistance of the cell membrane to ion n, and En represents the resting potential of the cell membrane for ion type n.

Using circuit analysis techniques an equation for the total membrane current can be quickly written assuming the current direction Im and also remembering that membrane current is defined as positive when positive ions leave the cell (Equation 3).

Equation 3

$$I = -\left(C_m \frac{\partial V_m}{\partial t} + \frac{V_m - E_1}{R_1} + \cdots + \frac{V_m - E_n}{R_n} \right)$$

Action Potential

An action potential is a electrical signal that originates at the axon Hillock and propagates down an axon (Figure 4). Under resting conditions the cell membrane has a net negative charge inside the cell membrane. In this resting state, a higher concentration of Na+ ions is maintained outside the cell. Also, a higher concentration of K+ ions is maintained inside the cell. At the Axon Hillock the membrane voltage is determined by the stimulus present at the dendrites. When this voltage crosses a certain threshold, gates are opened in the cell membrane allowing Na+ to flood into the cell. As Na+ enters the cell, the net negative charge is diminished, resulting in more Na+ gates opening and allowing more Na+ to enter. This is known as the depolarization phase and is the rising portion of the action potential.

Eventually the inside of the cell membrane reaches a positive potential and the Na+ gates close. In addition, K+ gates open up allowing the excess of ions inside the cell to flow out of the cell resulting in a decreasing potential within the cell. This is called the repolarization phase. Eventually, these gates are also closed.

As an action potential occurs at a point on the cell membrane, the surrounding cell membrane is also forced into this action potential process. Also, once the membrane has experienced an action potential it enters a resting period where the K+ and Na+ gates are held closed regardless of the membrane voltage. This is known as the refractory period and sets an upper limit on the firing rate of a neuron.

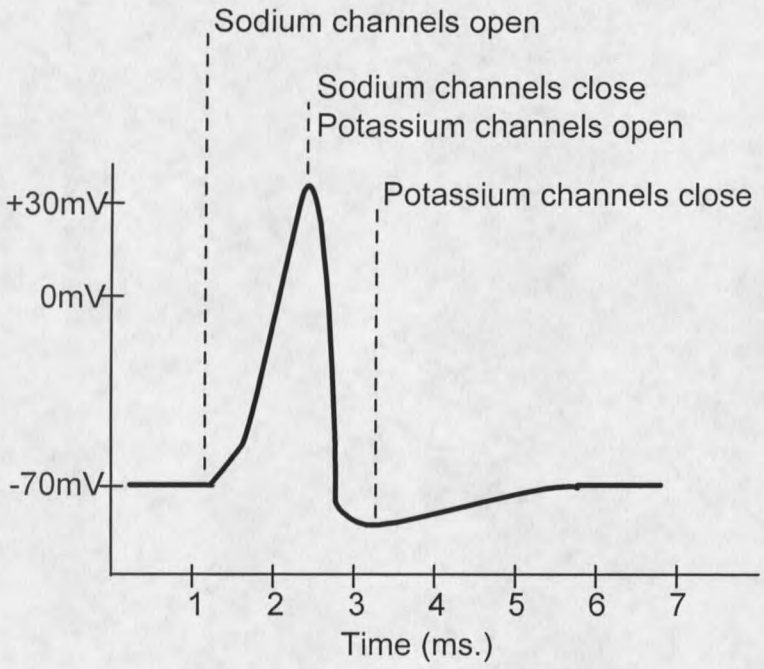

Figure 4 – Voltage vs. time plot of a typical action potential showing the opening and closing times of the potassium and sodium ion channels.

Classic Neuron Models

Accurately modeling neurons is a difficult task. To begin with there are many aspects of cellular function that are still unknown. Within a network of neurons, knowing exactly how information is represented is complicated and not fully understood. The importance of action potentials for propagating information is known, but there are many other possible sources of information transfer such as subthreshold potentials and chemical signals that would typically only be modeled in extremely detailed models (Andersson and Rosenfeld, 1988). Even determining what information an action potential conveys is under debate. It has been shown that in some cases the spike times are the important features involved in some neural computations (Hopfield, 1998). In

other cases the mean firing rate of a particular neuron appears to contain information also (Zohary et al., 1994). To make things even more complicated a particular action potential may have been inadvertently caused and would be considered noise. Even if all relevant information could be extracted, determining the neural code is another difficult task. Is information temporal in nature? Different types of neurons behave in many different ways. Until some of these questions are answered there will be many different models each with their own strengths and weaknesses.

Amidst all of these unknowns some models work quite well at a particular level of abstraction. The classical model of the action potential in axons proposed by Hodgkin and Huxley (1952) works quite well at modeling the electrical characteristics of the membrane even though it is not based on biophysical cellular processes. Compartmental models work well in modeling dendritic trees and typically use cable theory to describe signal propagation (Perkel et al., 1981). Although very simple, the integrate and fire model can be used to describe some high level neural function.

The appropriate level of abstraction defined by the goals of simulation determines what model should be used. For example, it is not computationally possible to model a network of millions of neurons using complex compartmental models in a reasonable amount of time using standard computer platforms. An even more limiting factor is the construction of the network itself. Using sophisticated neuron models usually involves a large amount of work under a microscopic mapping and measuring intricate networks of dendrite trees and axon structures. Performing such a task for a large scale simulation is not possible. And of course, time and computational power may be wasted if

computations are being done to reproduce cellular function when it is unnecessary. An example of this would be calculating the exact trajectory of an action potential when a simple digital "on" or "off" approach could be used with no information loss.

Integrate and Fire Neuron

The integrate and fire neuron model is the result of describing a neuron by its high level functionality instead of modeling the actual processes responsible for neural function. In fact it was first used before the details of neural function were known. The model proposed by Lapicque (1907) was a simple parallel RC circuit. When the voltage across the capacitor reached a certain threshold an action potential occurs and the capacitor discharges.

Although these simple models have evolved to more adequately describe neural function, the action potential is still treated as a binary state that occurs when certain input conditions are met. These input conditions typically consist of electrical stimulus from other neurons that are summed. When this summed voltage crosses a threshold, an action potential occurs and the cell must then wait for a length of time before firing again. By treating the action potential as a binary state the computations required to produce the voltage trajectory curve can be eliminated. Because of the relative ease of computation, these models are useful in modeling vast networks of neurons.

Hodgkin and Huxley Model Neuron

Hodgkin and Huxley proposed that the electrical behavior of a neuron membrane can be described using a simple electrical circuit (Figure 5). In this model the total ionic current is modeled as the sum of potassium and sodium currents as well as a leakage current that attempts to group all other ionic currents. The sodium and potassium currents are controlled by sodium and potassium conductances which are functions of both time and membrane voltage. All resting potentials as well as GL and Cm are considered constant and this was experimentally confirmed.

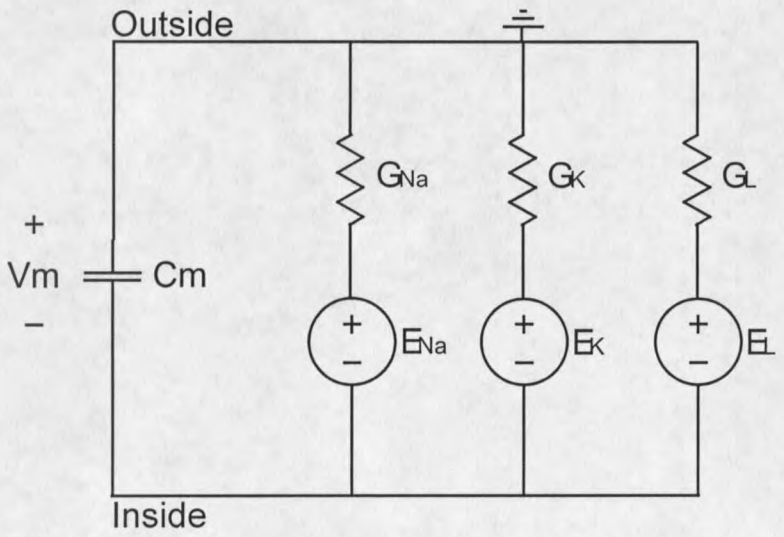

Figure 5 – Equivalent circuit model used by Hodgkin and Huxley which models both potassium and sodium ions and lumps all other ion types together as L.

A series of voltage clamp experiments were conducted and the resulting data was used to discover the mathematical relationships governing the changes in conductance of the cell membrane for the K and Na ions. By adjusting the voltage across the cell

membrane and observing the current responses they were able to show that currents due to both K+ and Na+ could be measured independently. Also, they showed that when the voltage across the cell membrane changed instantaneously the current and voltage relationship for a particular ion type obeyed Ohm's law. A instantaneous voltage change was necessary because it didn't allow time for the conductance to change. Once it was shown that Ohms law was applicable, the conductance could be calculated for a specific voltage input by dividing the resulting current by the driving voltage ($V-E_{ion}$). Plots of conductance as a function of membrane voltage were then created and equations for G_K and G_{NA} were developed by fitting curves to the resulting empirical data.

They were then able to propose an equation for total membrane current where G_{NO} and G_{KO} are the maximum conductance for potassium and sodium across the cell membrane (Equation 4). It was found that both ions could be modeled by using power functions of hypothetical first order rate equations *m*, *h*, and *n* (Equations 5-7). These rate functions are controlled by rate governing variables α and β, which in turn are all exponential functions of membrane potential. The α and β parameters for each hypothetical rate equation were determined through curve fitting techniques. Using these empirically determined parameters the model was able to predict the electrical characteristics of an action potential in the squid giant axon quite well.

Equation 4

$$I = C\frac{dE}{dt} + G_{NO}m^3h(E_{Na} - E) + G_{KO}n^4(E_K - E) + G_L(E_L - E)$$

Equation 5

$$\frac{dn}{dt} = \alpha_n(1-n) - \beta_n n$$

Equation 6

$$\frac{dm}{dt} = \alpha_m(1-m) - \beta_m m$$

Equation 7

$$\frac{dh}{dt} = \alpha_h(1-h) - \beta_h h$$

Multi-Compartmental Models

Any model that treats membrane voltage as constant across the entire cell surface is considered a compartmental model. This is a poor model for most neurons because long thin segments have intercellular resistance associated with them. This creates a scenario where a single compartment can no longer describe a neuron, but instead it only describes a region within a neuron where conditions are similar. Such a neuron can then only be modeled using a series of compartments (Perkel et al, 1981). Intercellular resistance can be modeled quite well using cable theory (Rall, 1964). This was especially useful in modeling the vast thin branches of dendrite trees. In addition to utilizing cable theory to describe these intercellular resistances it was also beneficial to treat dendrites as a arrangement of smaller compartments. Using such an approach allows properties such

as diameter, which is the most important parameter in describing the signal propagation through a dendrite segment, to be treated as a constant.

Compartmental models of neurons tend to be very complex and can model individual neurons and small networks quite well (Hines, 1994). Thousands of compartments can be used to describe the function of a single neuron. The model for each compartment can also be quite complex. Typically, modeling large networks of neurons is not done this way for computational reasons.

Macgregor Model

The model chosen to be implemented within the architecture of a FPGA is the MacGregor point model neuron. The model alone is probably not very useful and by today's standards would be considered quite primitive. However, it was not chosen for its applicability to model real neurons or networks. Instead, it was chosen to demonstrate the proof of concept that FPGAs can provide good platforms for performing neural simulations. It is a very computationally efficient model that is easy to understand. It also mimics spike generation mechanisms in more detail than a simple integrate and fire model. The model is well documented and has been used to model a variety of different neuron types. In its most simple form the model consists of four differential equations (Equations 8-11). Each equation models one of the four state variables: membrane potential (V_M), threshold (Th), potassium conductance (G_K), and spike (S). The threshold variable is driven away from its resting potential depending on the membrane voltage.

This models the accommodation (Hill, 1936) and adaptation (Kernell, 1968) properties of a neuron which essentially means that the threshold level for a particular neuron tends to rise as it is stimulated at the subthreshold level. The potassium conductance variable is responsible for repolarizing the cell membrane as was seen in the Hodgkin and Huxley model. The constant parameters will be described in detail later.

These equations realistically model the ongoing input/output dynamics of various neuron types through the manipulation of model parameters. The input to the system is an injection of current.

Equation 8

$$\frac{dV_m}{dt} = \frac{-V_m + \{I_{in} + G_K * (E_K - V_m)\}}{T_{mem}}$$

Equation 9

$$\frac{dTh}{dt} = \frac{-(Th - Th_0) + C * V_m}{T_{Th}}$$

Equation 10

$$\frac{dG_K}{dt} = \frac{-G_K + B * Spike}{T_{GK}}$$

Equation 11

$$Spike = \begin{cases} 0 & if \quad E < Th \\ 1 & if \quad E \geq Th \end{cases}$$

All equations can be placed in exponential form (Equation 12) which can easily be numerically solved (Equation 13). A(t) and B(t) are assumed to be constant over a given integration time step. By using this integration method, all of the point neuron state equations can be re-written in a form where the next neuron state is a function of previous states as well as constants (Equations 14-17).

Equation 12

$$\frac{dE(t)}{dt} = -A(t)E(t) + B(t)$$

Equation 13

$$E_{i+\Delta} = E_i e^{-A\Delta} + \frac{B}{A}(1 - e^{-A\Delta})$$

A Matlab script was used to iteratively update the state variables for a typical neuron (Appendix A). These 4 states were then plotted as a function of time (Figure 6). Also shown are some of the parameters used to adjust the dynamics of the neuron model. The parameters adjust the time constants and amplitude of the various exponential functions.

Equation 14

$$V_m(i+1) = V_m(i)e^{\frac{-(1+G_K(i))\Delta}{T_{mem}}} + \left(\frac{G_K(i)E_K + I_{in}}{1+G_K(i)}\right)\left(1 - e^{\frac{-(1+G_K(i))\Delta}{T_{mem}}}\right)$$

Equation 15

$$Th(i+1) = Th(i)e^{\frac{-\Delta}{T_{Th}}} + \left(C * V_m(i) + Th_0\right)\left(1 - e^{\frac{-\Delta}{T_{Th}}}\right)$$

Equation 16

$$G_K(i+1) = G_K(i)e^{\frac{-\Delta}{T_{GK}}} + B * Spike(i) * \left(1 - e^{\frac{-\Delta}{T_{GK}}}\right)$$

Equation 17

$$Spike(i+1) = \begin{cases} 0 & if \quad E_{i+\Delta} < Th_{i+\Delta} \\ 1 & if \quad E_{i+\Delta} \geq Th_{i+\Delta} \end{cases}$$

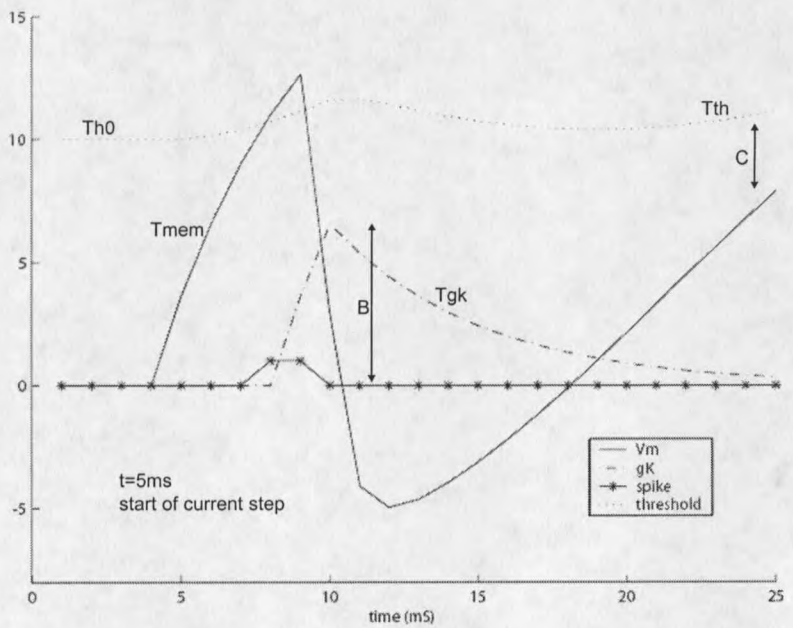

Figure 6 – Plots of the four state variables of the MacGregor Point Neuron as a function of time in response to an injected current.

The T_{mem}, T_{Th}, and T_{GK} parameters adjust the time constants for the membrane voltage, threshold and potassium conductance curves. C and B adjust the amplitude of the threshold and potassium conductance curves. The Th_0 parameter determines the resting threshold state for the cell membrane. By adjusting these parameters many different neuron types can be modeled.

The C parameter determines how sensitive the threshold is to the membrane voltage, or in other words how much rise in Th is associated with a rise in V_M. Typically C ranges from 0 to 1. When C=0 the threshold is not adjusted to a change in membrane voltage and thus no accommodation takes place and the neuron fires monotonically (Figure 7). Here, the threshold is simply a straight line and the firing rate of the neuron is constant. With C=1 the neuron will eventually stop firing with a constant current input (Figure 8). Here the neuron fires 3 times in response to a constant current pulse and stops firing even though current is still being injected.

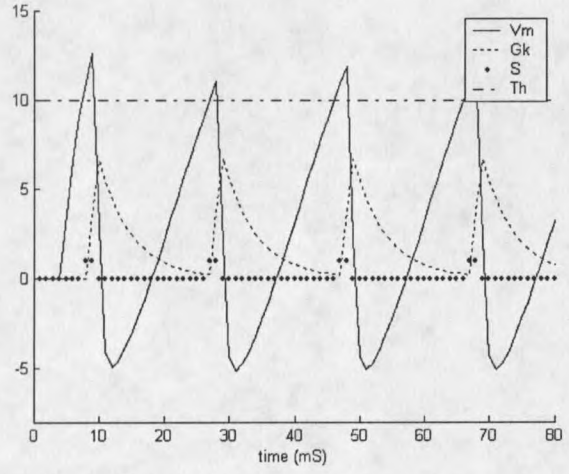

Figure 7 – Effect of setting the model parameter C = 0 for a typical neuron.

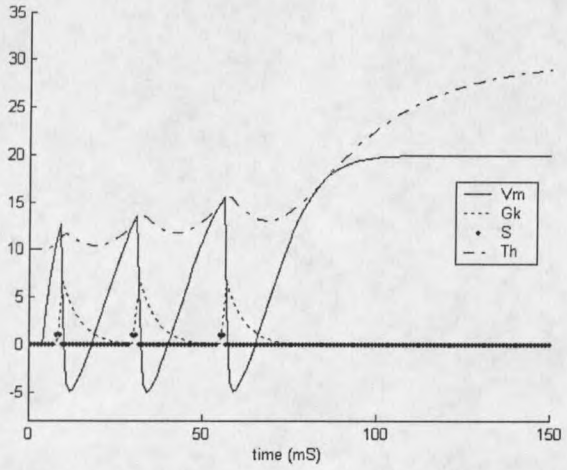

Figure 8 – Effect of setting the model parameter C = 1 for a typical neuron.

T_{Th} is the time constant for accommodation and is typically 20-25ms. Adjusting T_{Th} affects how fast the threshold can change in response to a changing membrane voltage. Again a current pulse is used as the stimulus. A large time constant prevents the threshold level from adapting to the membrane voltage (Figure 9). In contrast a small time constant allows the threshold to adjust quickly to membrane voltage (Figure 10).

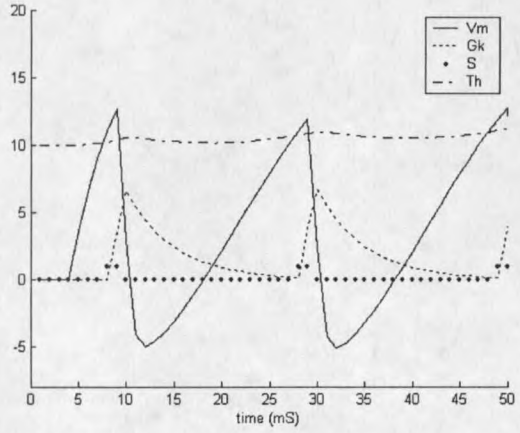

Figure 9 – Effect of setting the model parameter T_{Th} = 40ms for a typical neuron.

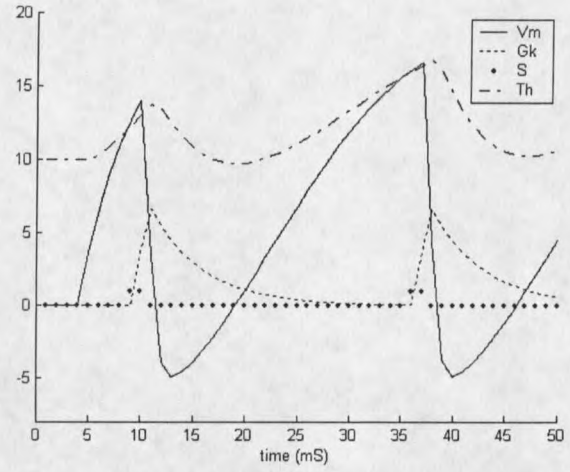

Figure 10 – Effect of setting the model parameter T_{Th} = 5ms for a typical neuron.

B is the sensitivity to potassium conductance and is nominally 20. Adjusting this value adjusts the height of the post-firing potassium conductance decay which is a restoring mechanism for the membrane voltage. Using a large value for B causes the membrane voltage to rapidly return to its resting state (Figure 11). With a small value for B, the membrane voltage is not restored very quickly (Figure 12).

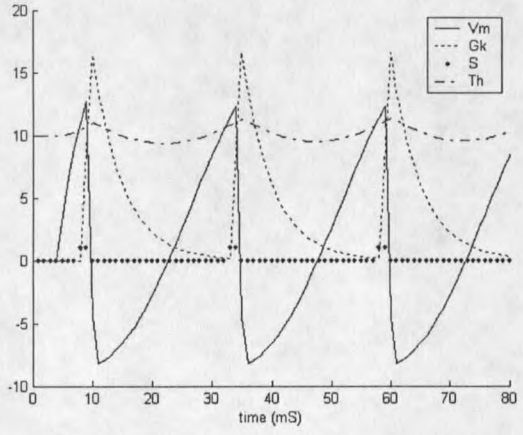

Figure 11 – Effect of setting the model parameter B = 50 for a typical neuron.

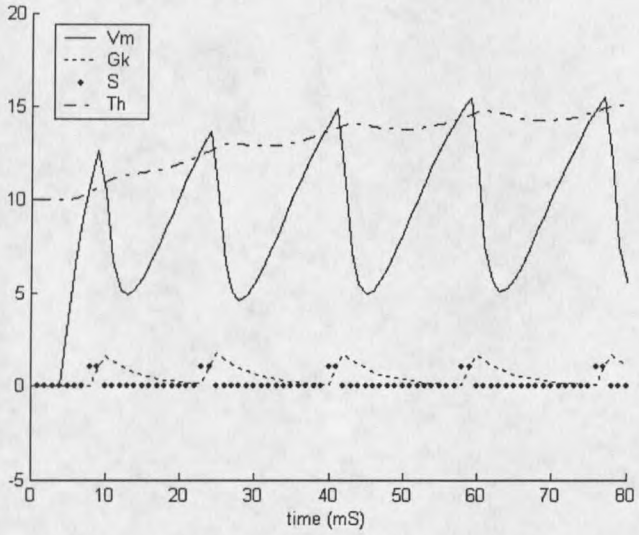

Figure 12 – Effect of setting the model parameter B = 5 for a typical neuron.

T_{GK} is the refractory time constant for Potassium conductance and is typically 3-10ms. Adjusting this value adjusts the decay rate of the post-firing Potassium conductance. This is an important parameter in changing the firing rate or refractory period of a neuron. Using a large time constant for the potassium conductance decay creates a large recovery period before the stimulated neuron can fire again (Figure 13). In contrast, a short time constant allows the stimulated neuron to quickly recover from a spike and fire again (Figure 14).

T_{mem} represents the time constant of the membrane which is typically 5-11ms. This parameter adjusts the responsiveness of the membrane to input current stimulus. Using a small time constant prevents the membrane voltage from changing rapidly as a constant current is injected (Figure 15). Using a large membrane voltage time constant

slows the ability of the membrane voltage to change (Figure 16). In conjunction with the potassium conductance time constant this parameter can adjust the firing rate of a neuron.

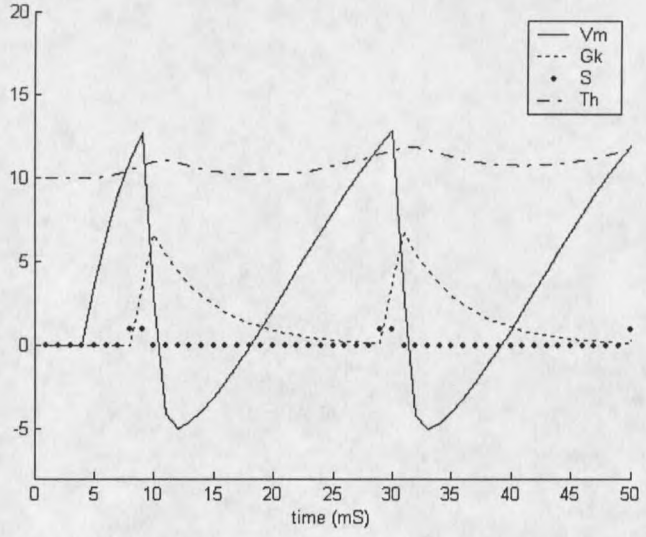

Figure 13 – Effect of setting the model parameter T_{GK} = 5ms for a typical neuron.

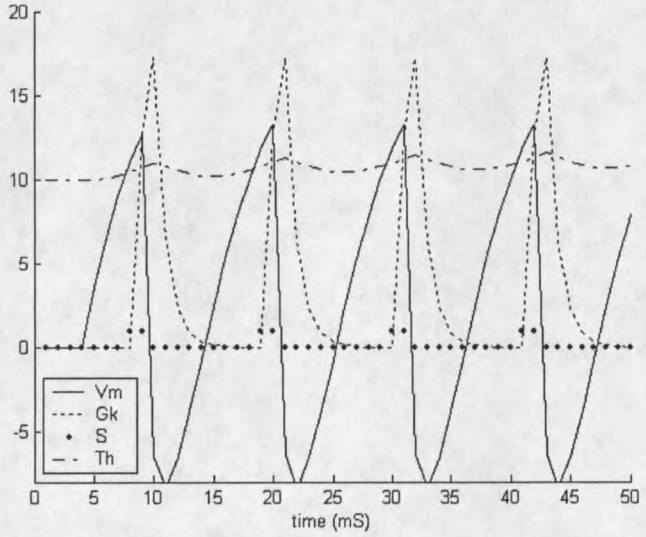

Figure 14 – Effect of setting the model parameter T_{GK} = 1ms for a typical neuron.

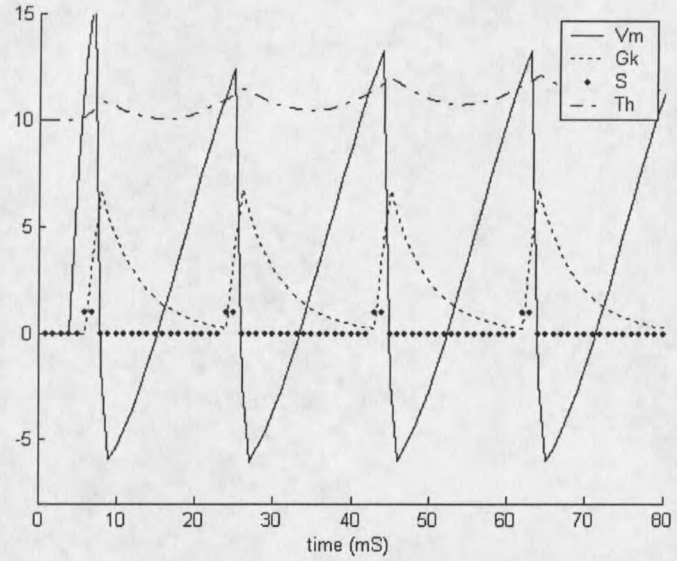

Figure 15 – Effect of setting the model parameter T_{mem} = 2ms for a typical neuron.

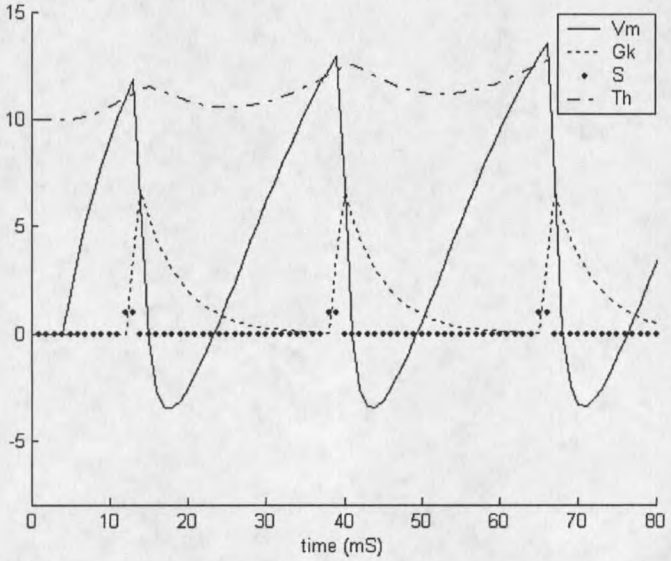

Figure 16 – Effect of setting the model parameter T_{mem} = 10ms for a typical neuron.

One important note is that the membrane voltage variable does not model the exact form of an action potential. This is because the membrane voltage state variable is

used to account for the membrane voltage but not represent it exactly. Thus, to extract a simulated action potential it is necessary to use a conditioning equation that is a function of membrane voltage and spike (Equation 18). This creates an output that mimics what would be seen when recording from the soma (Figure 17).

Equation 18

$$Soma\ Potential = Vm + Spike \times (50 - Vm)$$

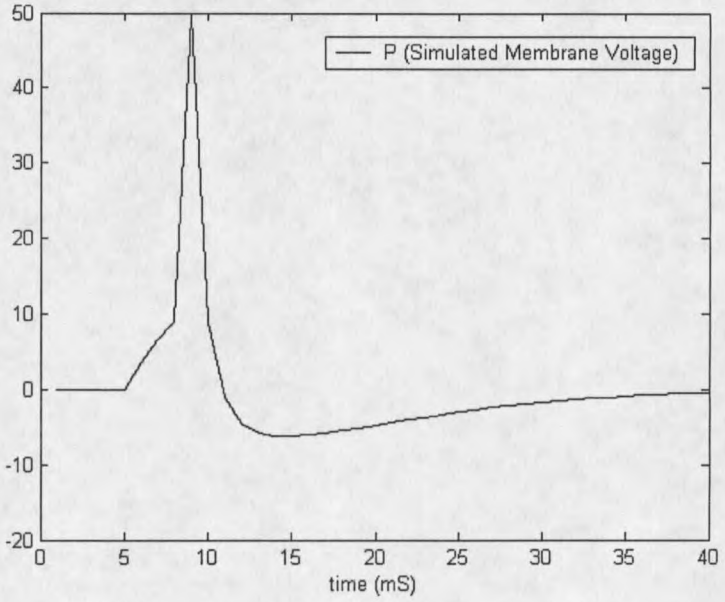

Figure 17 – Plot showing the simulated action potential output of the MacGregor Point Neuron using Equation 18.

HARDWARE

The Reconfigurable Online Modeling Platform (ROMP) is a signal processing platform that utilizes the pairing of Digital Signal Processors (DSPs) and Field Programmable Gate Arrays (FPGAs). The goal of this combination is the development of a platform that will allow real-time analysis of neural systems as well as general purpose signal processing. Specifically, this will allow the decoding of neural information streams through spike sorting and optimal filtering methods (Gozani et al., 1994) to be done in real time. Thus, off-line post experiment analysis of recorded data will be replaced by on-line analysis and interaction with data while experiments are in progress. Additionally, models of high level neural processing areas can be developed utilizing networks of neurons that mimic actual biological systems.

Two boards have been developed in the Snider Lab at Montana State University, one utilizing a DSP and another utilizing an FPGA (Figure 18). The two boards act as a single processing node where communication takes place via a connector that shares the address and data bus of the DSP. Communication can also take place via high speed data links that are present on both the FPGA board and the DSP board. This allows for networks of nodes to be assembled for a fully scalable processing system.

For the purpose of neuron modeling only the FPGA subsection will be considered. The FPGA board consists of the following key components: a FPGA where reconfigurable computing takes place, a Complex Logic Device (CPLD) that is responsible for controlling the board at boot up, programmable ROM which holds FPGA

configuration data, a Double Data Rate (DDR) SODIMM connector that can provide up to 512 Mbytes of memory for each node, and high speed transceiver devices which provide inter-nodal communication.

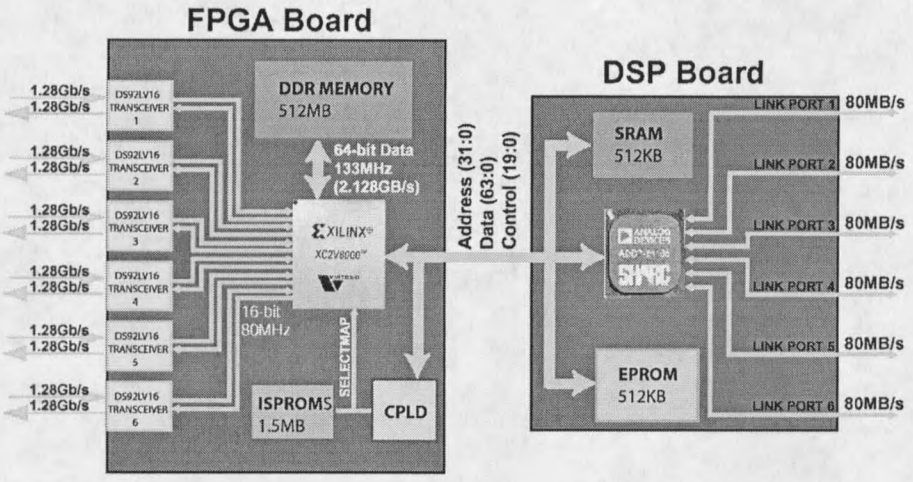

Figure 18 – System diagram of the Reconfigurable Online Modeling Platform.

FPGA

The heart of the FPGA board consists of a Virtex-II FPGA. Specifically the board is designed for a Virtex-II 3000 part but since the BF957 package is pin compatible with larger devices, the board can easily be upgraded.

FPGA architectures such as the Virtex-II are ideal for meeting the re-configurable computing requirements of this system. In addition to vast logic resources there are also dedicated RAM and multiplier structures which are specially designed for doing Multiply and Accumulate (MAC) operations, which is a common occurrence in many signal

processing algorithms such as FIR filtering. Next to each embedded multiplier structure exists 18Kbit blocks of dual port RAM. Thus, both 18-bit inputs to a multiplier can be supplied in one cycle.

In addition to a co-processing resource, the FPGA also acts as a network switch for all of the communications taking place via the high speed transceiver devices. Each transceiver device operates on 16-bit send and receive busses operating at 80MHz. Nodes that are not the source or destination of data streams efficiently pass communications through in a manner such that short paths through the system are taken. The Virtex-II 3000 has over 600 available IO pins and easily accomplishes this task.

LVDS Transceivers

The National Instruments DS92LV16 device is a LVDS (Low Voltage Differential Signal) serializer/deserializer which is used for inter-board communication. A 16-bit LVCMOS bus operating at 80MHz is translated into a LVDS serial data stream operating at 1.28 Gbit/s with embedded clock information. Each device has independent receive and transmit busses allowing for 2.56 Gbit/s of full duplex throughput. In addition, PLLs (Phase Locked Loops) recover the embedded clock signal from the serial data stream. The serial data streams can take place over any twisted pair cable, but in order to accommodate frequencies above 1GHz higher performance cables such as twin-axial are necessary.

Six DS92LV16 devices are present on each board which enables the board to have bidirectional communication with each nearest neighbor when the boards are placed in a 3-dimensional configuration (Figure 19).

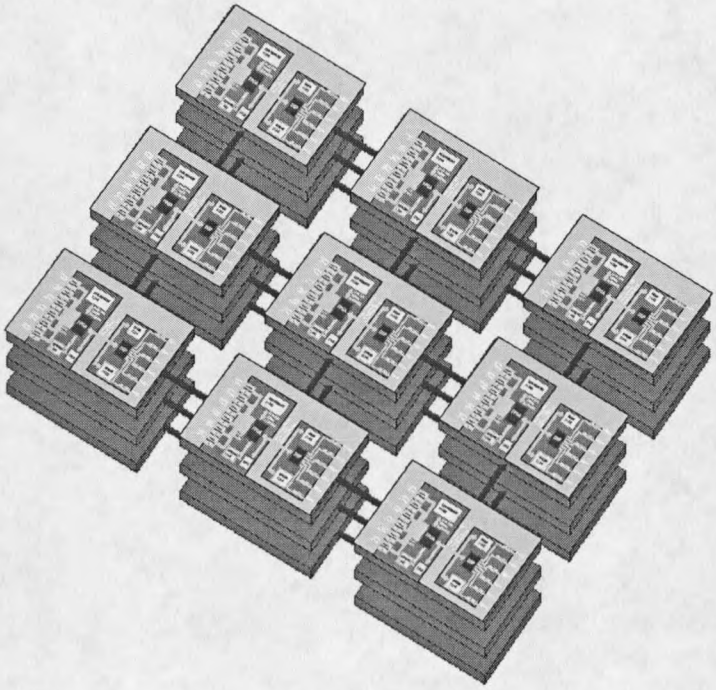

Figure 19 – Diagram showing the use of high speed serial connections to form a 3x3x3 array of processing nodes.

DDR SODIMM

In order to provide a flexible memory solution for the FPGA board a SODIMM connector was used. The connector will accept any size Double Data Rate (DDR) SODIMM memory stick, but was designed specifically around a 512 Mbyte Micron DDR

SODIMM. DDR SDRAM provides over 2 Gbyte/s of data throughput providing a high speed storage solution for the FPGA.

CPLD and IsProms

The Xilinx XC95144XL CPLD acts as the boot sequencer and board controller. The primary purpose of the CPLD is to control the board initialization after a reset or power cycle has occurred. Like the FPGA, the CPLD is also a reconfigurable logic device, although the architecture only supports limited functionality. Unlike the FPGA, the CPLD is a non-volatile device allowing configuration information to be retained while the device is not powered. Typically the role of a CPLD is to act as glue logic between components. On the FPGA board, the CPLD controls the IsPROMs and FPGA during boot-up and also acts as an interface for the DSP to reconfigure the FPGA.

The Xilinx XC18V04 in system PROMs (Programmable ROMs) are necessary to hold configuration bitstreams for the FPGA. Each device holds up to 524 Kbyte of data, and to accommodate the large (1.3Mbyte) bitstream of the Virtex-II 3000 device, a chained configuration of 3 IsPROMs is necessary (see bootup section). The devices interface to the FPGA using the SelectMAP configuration scheme and are easily programmed using a JTAG interface cable. On the FPGA board this JTAG interface is part of a chain that includes all IsPROMs and the CPLD.

MACGREGOR MODEL IN AN FPGA

With only four state variables the MacGregor Point Neuron Model can be solved numerically using a minimal amount of program code. The integration scheme can be accomplished using 16-bit fixed-point arithmetic which is ideal for implementation within a FPGA. It has also been observed that the neurons themselves do not exhibit 16-bit precision. Using fixed point is important because floating point arithmetic is generally not a good choice for FPGAs because such operations tend to require excessive area within the device and are generally slow.

To make things easier, the Fortran code proposed by MacGregor has been translated into Matlab code (Appendix A). The core of this code involves the calculation of the following state variables: potassium conductance (GK), membrane voltage (Vm), threshold, and a binary action potential spike (Spike). This needs to be done for each simulation time step of 1 ms. To make things simple all constants in the numerical solution to the MacGregor model were replaced by K's (Equations 19-22).

Equation 19

$$G_K(i+1) = G_K(i) \times K_1 + Spike(i) \times K_2$$

Equation 20

$$V_m(i+1) = V_m(i) \times e^{-(1+G_K(i)) \times K_3} + \frac{(I_{in} + G_K(i) \times K_4)(1 - e^{-(1+G_K(i)) \times K_3})}{(1 + G_K(i))}$$

Equation 21

$$Th(i+1) = K_5 + (Th(i) - K_5) \times K_6 + V_m(i) \times K_7$$

Equation 22

$$Spike(i+1) = \begin{cases} 1 & if \quad V_m(i) > Th(i) \\ 0 & otherwise \end{cases}$$

These equations can be solved by any machine capable of performing addition, subtraction, multiplication, division, and the exponential function. The first three operations can be performed easily within FPGA architectures such as the Virtex-II which are specially designed for doing a Multiply and Accumulate (MAC) operations. The reason for this is the MAC operation is a common occurrence in many signal processing algorithms such as FIR filtering. Next to each embedded multiplier structure exists a 18Kbit block of dual port RAM. Thus, both 18-bit inputs to a multiplier can be supplied in one cycle.

Performing exponential functions and division within a FPGA poses a problem. Typically if these operations have to be done, "tricks" are used such as using look-up tables that store pre-calculated results or shift-add algorithms that improve the accuracy of the result with each shift. The approach taken was to simply use the Taylor series approximation of the exponential function. The Taylor series can then be implemented as a series of multiplications by constants and addition steps. The division operation $1/(1+G_K)$ used in the membrane voltage calculation is of the form $1/(1+x)$ which also has a Taylor Series associated with it. However, in this case the Taylor series converges for

values of x less than 1. Thus, it was necessary to use a look up table of pre-calculated values. This decreases the set of mathematical operations necessary to solve the set of state equations down to the three that work well in an FPGA: addition, subtraction, and multiplication. The largest Virtex-II PRO device contains 612 embedded multipliers and 18K-bit RAM pairs and 28,832 Configurable Logic Blocks (CLBs). Adders and subtractors can be built utilizing the programmable logic cells within each FPGA. A single 16-bit adder/subtractor can be implemented in 4 CLBs. Thus, over 7,000 16-bit adders could be implemented in a single device. With these resources there are many options for performing these operations.

The next consideration involves how to take advantage of the architecture within the FPGA while maximizing the number of neurons that can be simulated within a single FPGA. There are typically two approaches to designing logic in FPGAs. One approach is to make calculations as fast as possible without worrying about the logic footprint within the device. The reason for this is because fast designs are usually highly parallel and highly pipelined structures. The other approach is to try to optimize the amount of logic used to perform a task without particular attention to the speed of the design. This method relies heavily on resource re-use and floor-planning.

Methods of Implementation

The question then arises as to how to make the best use of this FPGA architecture. The point neuron equations can be thought of as a set of constants and variables that need

to be operated on in some manner. The operations are defined by the set of equations necessary to update the state variables. Two very different approaches were examined as possible solutions.

Approach #1 – A Neuron CPU

The first approach would involve building a custom CPU which is specifically designed to implement the MacGregor Point Neuron Model. A general purpose CPU is able to solve almost any problem by manipulating data in some manner defined through a set of instructions. Additionally, each instruction contains specific information defining the operation of all the sub-components of processor. Creating a processor that is both high performance and general purpose is a difficult task especially within a FPGA fabric. Instead, a small processor designed specifically for FPGA architectures is a much better approach. Typically, these processors have smaller custom instruction sets (< 40 instructions) and slower clock speeds (< 100MHz). The complexity of a processor can be seen by looking at its instruction set. Not much is required in terms of hardware for a CPU that only does a small number of tasks and the resulting instruction set is very small. Individually these processors can't compete with desktop CPU performance, but what they lack in performance can quickly be overcome by the sheer number of processors that can be implemented on a single FPGA. Hundreds of soft processors such as Xilinx's PicoBlaze processor can be implemented within a single FPGA. (Chapman, 2002)

It is entirely feasible to design a processor with a custom instruction set (10-15 instructions) that is highly optimized for solving sets of equations such as those necessary to simulate the MacGregor Point Neuron Model. Specifically, arithmetic instructions such as multiply, add, and subtract would be necessary as well as load and store operations to move data in and out of memory. Branching instructions as well as a minimal set of logic instructions would also be necessary to provide a minimal amount of program flow. This processor could be designed around a single embedded block ram and multiplier structure (Figure 20). Structures such as a program counter, register banks, a arithmetic logic unit and instruction decoder would also be necessary.

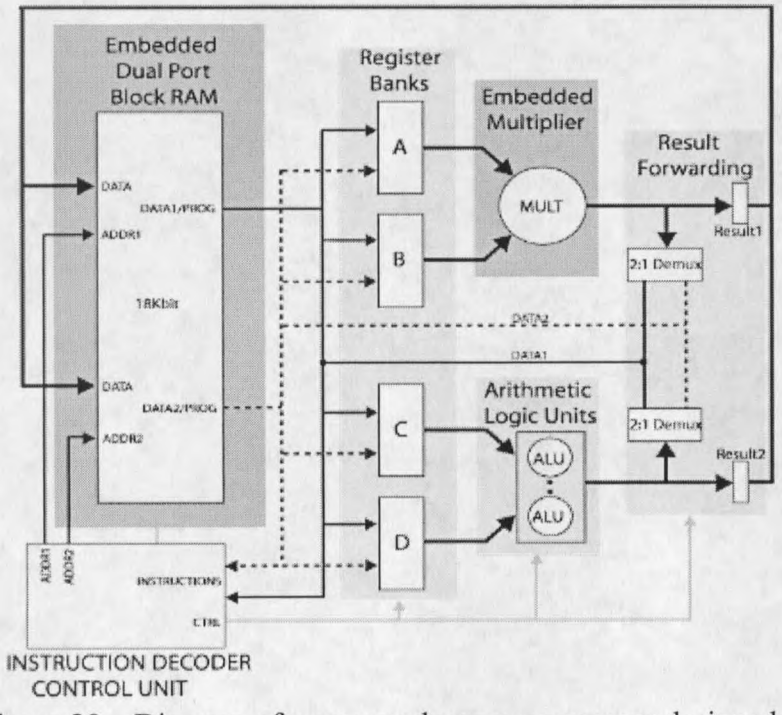

Figure 20 – Diagram of a proposed neuron processor designed around a block RAM and multiplier optimized for performing the arithmetic operations used in the MacGregor Point Neuron Model.

Program code as well as data is stored inside each block RAM. For each neuron being simulated, the set of operations necessary to update the state variables (code) is the same, but the data values different. Thus, reusing the same code on different data would be the ideal solution. This CPU design methodology is typically called Single Instruction Multiple Data (SIMD). In this scenario the number of neurons able to be simulated by a single neuron processor depends mostly on the amount data that can fit into a single block RAM (Figure 21). In a homogenous network of neurons, the constants that describe neuron behavior only need to be stored in memory once. Under this scenario depending on the length of the program upwards of 200 neurons could be simulated within a single 1K word block RAM. (800 words/4 states). Assuming, that each neuron has different properties probably only 80 neurons could be simulated within a single block RAM (800words/(4 states + 7constants)).

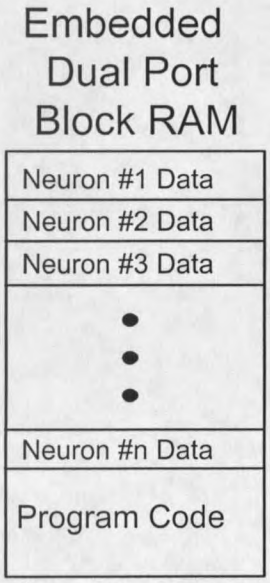

Figure 21 – Memory map of proposed neuron processor.

This approach has some benefits. First, the model itself could be changed relatively easily by changing the program. Also, simulating a large number of neurons is possible. For example, if a neuron processor was build around each of the 168 block RAM and multiplier pairs that exist in a large Virtex-II, then simulating over 30,000 neurons would be possible (200 neurons x 168 processors). Of course, simulating 30,000 non-networked identical neurons is not useful.

At first glance this approach looks appealing, but there are some significant drawbacks. The biggest drawback is the fact that implementing some sort of network for the neurons would be both challenging and potentially resource consuming. For example, neuron A being simulated in processor X should be able to interact with any neuron being simulated in any processor. With the proposed architecture this is a difficult task. The networking fabric is far too complex to be implemented within the program code of the neuron processors as it would vastly diminish the number of neurons that each processor could simulate. The network itself would have to be external to the neuron processors, yet be somewhat integrated into the structure of each processor. Under the assumption that any neuron can stimulate any other neuron in the entire system routing problems are generated. For example, at any particular clock cycle a neuron in every neuron processor can generate a spike. Thus a mechanism for sending information from one neuron to any or all other neurons needs to be developed. This is a very difficult task as the sheer number of possible sources and destinations in the system as well as the simultaneous nature of requests creates a serious routing problem. Also, the idea that neurons in separate processors can affect each other implies that the memory

space inside each processor can be externally modified. This adds a level of complexity in the processor design and possibly involves interrupting processing to allow external interaction.

Approach #2 – A Parallel Processing Structure

Looking again at the numerical solution to the MacGregor Point Neuron Model (Equations 19-22) a different approach was taken. Instead of using memory to hold the processing information necessary to update states this information was hard wired into the FPGA fabric itself. The solutions to the 4 state equations consisted of large parallel structures that multiply, add and subtract data. This involved using a embedded multiplier for each multiply that exists in the equation set. Addition and subtraction were performed using adders or subtractors created using the configurable logic blocks within the device. This resulted in a parallel and pipelined structure that can update all four state variables for a single neuron in one clock cycle.

The state equations for potassium conductance, membrane voltage, and threshold can now be represented as a multiply and add tree structure (Figures 22-24). The spike state equation is trivial and not shown in tree form. The constants and states are stored in Block RAM. The z^{-x} blocks represent delays.

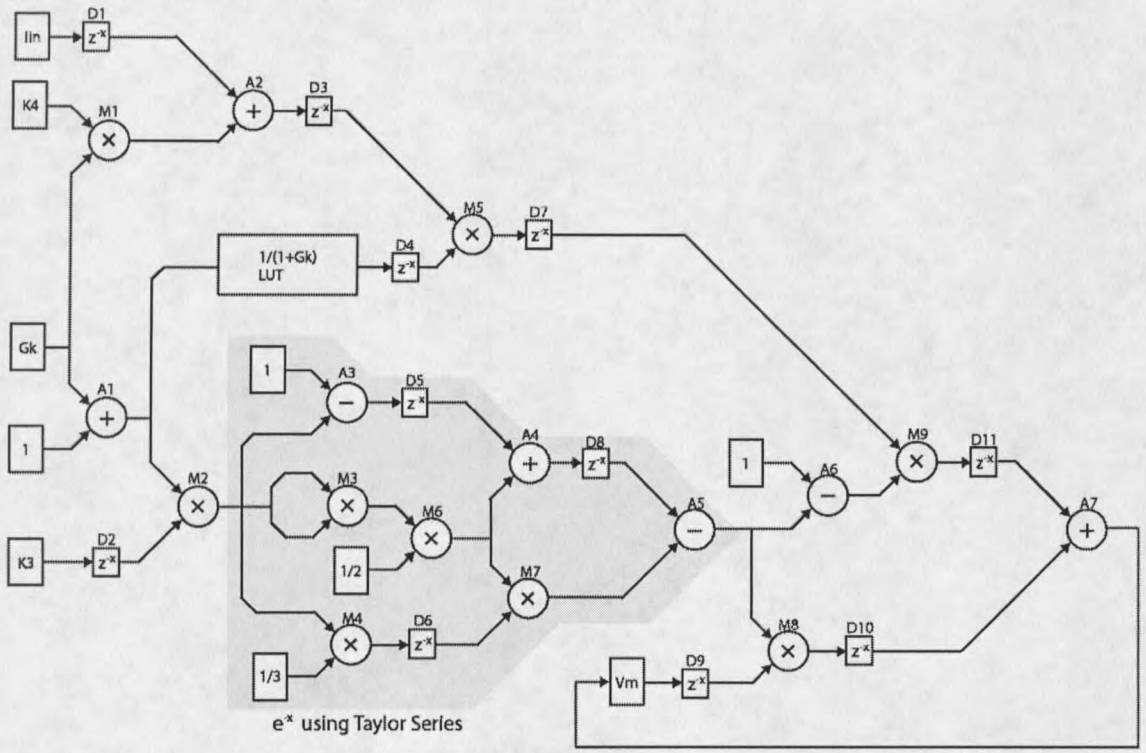

Figure 22 – Equation tree used to solve for updated membrane voltage states using multipliers, adders, and delay blocks. Difficult functions such as e^x and division are performed using a Taylor series (shown in grey) and look-up tables.

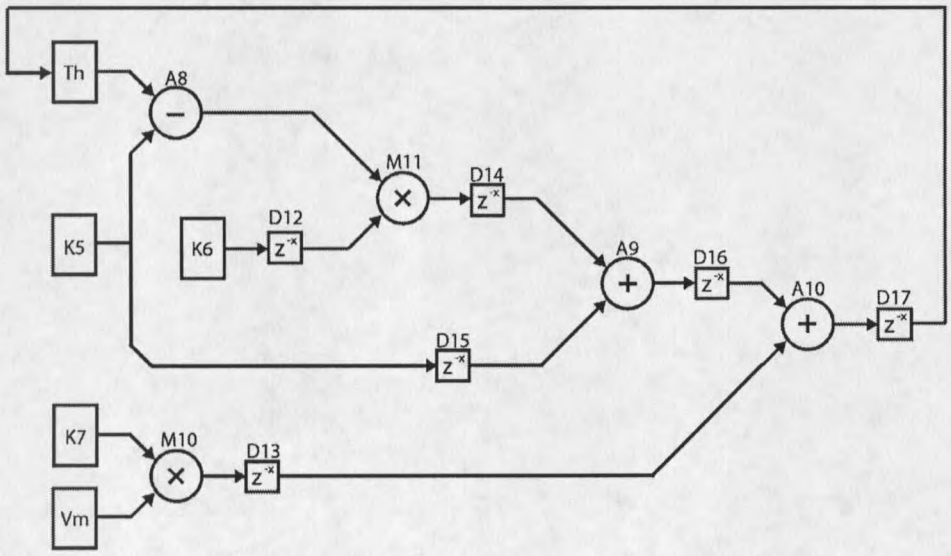

Figure 23 – Equation tree used to solve for updated threshold states using multipliers, adders, and delay blocks.

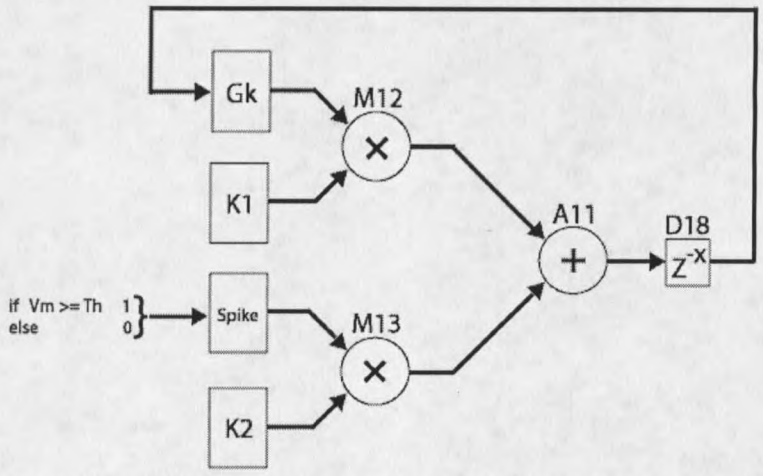

Figure 24 – Equation tree used to solve for updated potassium conductance states using multipliers, adders, and delay blocks

Immediately apparent is that the processing tree solution for membrane voltage has a very large latency. In order to get the maximum performance out of the embedded multipliers a 3 stage pipeline was used for each multiplier. The total latency through the membrane voltage tree was found by assuming a delay of 1 for each adder and 3 for each multiplier. This latency was 19 clock cycles. Without implementing a pipeline through this tree structure a latency of 19 made this approach a very poor choice. Instead a pipeline with depth equal to the latency through the tree was used. Once the pipeline was full this method allows for updating all four state variables for a neuron in one clock cycle. Although approximately equal in length to a Pentium 4 processor pipeline, this particular pipeline always remains full.

Having a full pipeline created a data alignment problem for the tree structure. During any given clock cycle both inputs to each multiplier and adder in the entire structure needed to represent data for the same neuron. This was a tricky task

considering that when the pipeline is full it contains almost 20 different neuron states. Also, the total latency through each equation tree structure was intentionally created to be the same as the latency through the membrane voltage tree. Matching the delays of all equation trees simplified the addressing of neurons. During each clock cycle state information for a neuron is loaded into all of the trees simultaneously and updated state information for a different neuron is present on the outputs of the equation trees. Thus, for all data that is loaded into the equation trees a common neuron address is all that is required.

Once the equation trees were designed, a block RAM controller circuit was designed that loads state information into the trees as well as stores the updated states present on the output of the trees (Figure 25). Each block RAM is capable of containing state information for 1024 neurons. On startup each block RAM is configured with initial state values from a global simulation controller. Data for a specific neuron is stored at the same address location across all block RAMs used in the simulation. Until all of the block memory has been initialized the global simulation controller keeps all block RAMs in a state where data is not unintentionally overwritten. Also, at this time the external data input to the multiplexer is selected instead of the updated state input. All block RAMs share a common global data bus.

Once all of the block memory has been initialized the block RAM can be switched into a partial simulation mode. A partial simulation mode is necessary because until the pipeline is full the data outputs of the equation trees are not valid and should not be written back into memory. During this short delay, port A is disabled for writing data

into the block RAM. After this delay the system is switched into full simulation mode. In full simulation mode the state data input to the data multiplexer is selected and port A is enabled for writing data back into memory. Also, the address multiplexer for port A selects a delayed version of the address for port B. This address is delayed to match the address of the updated neuron state information to be loaded back into memory.

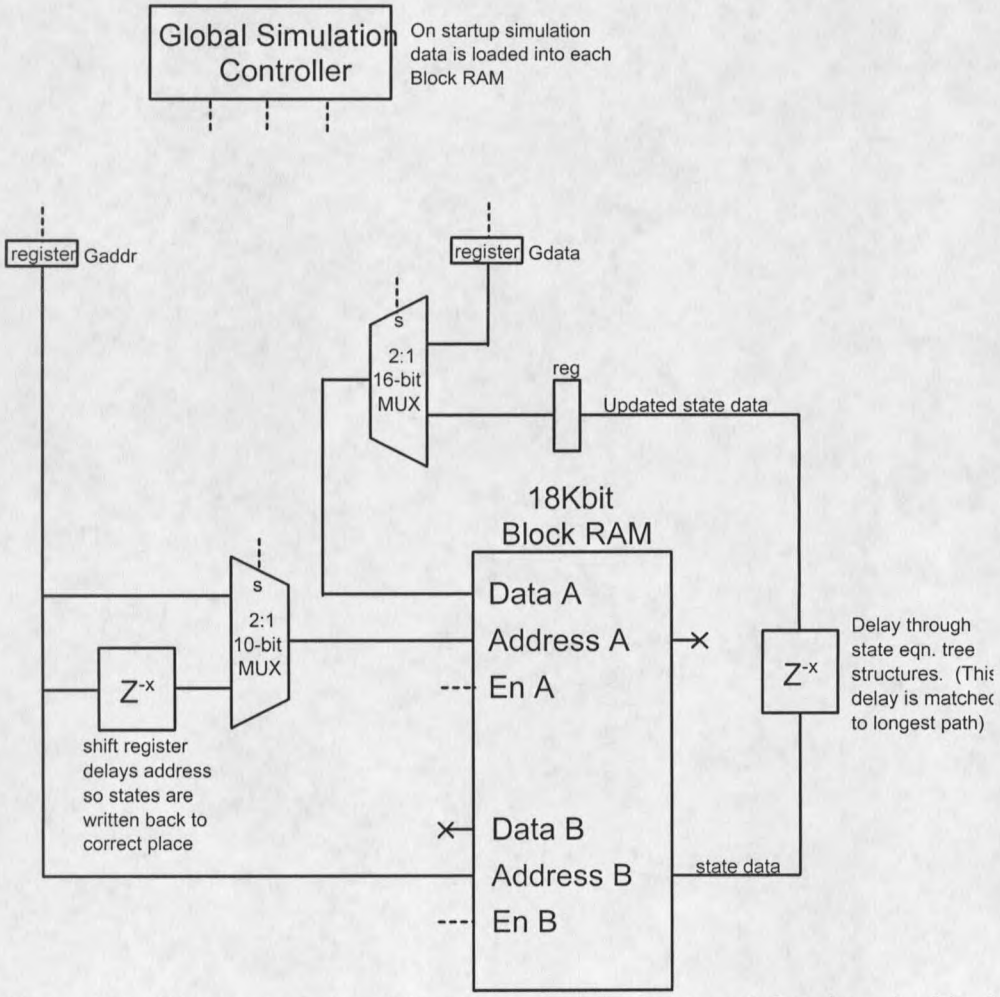

Figure 25 – The control circuit that is used to load external data into the block RAMs as well as the loading and storing of neuron states into and out of the equation trees.

When systems of dissimilar neurons are simulated, the constants that describe neuron behavior are stored in block RAM. A controller for a block RAM that holds neuron parameters doesn't need to write back data during the simulation thus multiplexing circuitry is not necessary.

This approach of gratuitously adding delays into the system may seem impractical at first. This is indeed true if such delays are created by using flip-flops. Each CLB in a Virtex-II FPGA contains 8 flip-flops. Using flip-flops to implement a delay queue of 16 clock cycles on 16-bit data uses 32 CLBs. This would quickly use up the FPGA resources. Instead, these delays can be created through the use of shift registers which are the result of configuring LUTs in a different manner than they are typically used. Each LUT is configured as 16x1 bit shift register. This process is similar to using LUTs as distributed RAM. Each CLB in a Virtex-II device contains 8 of these LUTs. Thus, a 16-bit shift register with a delay up to 16 cycles can be implemented using only 2 CLBs. This approach allowed the creation of the necessary delays and used only minimal FPGA resources.

Fixed Point Arithmetic and the MacGregor Model

Fixed Point Data Types in Matlab

Before a hardware solution was implemented inside a FPGA, testing was necessary to ensure that a fixed point data type would work when used with the

Macgregor Point Neuron model. Although Matlab supports integer types, no mathematical operations are defined for them because the operations are ambiguous on the boundary of the set. Thus, a custom 16-bit signed fixed point data type was created for Matlab with all the appropriate mathematical operators defined (Appendix E). At the time of creation each fp16 type must be told the location of its binary point. For instance a binary position of 10 creates a fixed point type with 6 signed integer bits as well as 10 fractional bits All ambiguities on the boundary of the set are removed by raising an error flag and halting execution when any result is outside of the set. Raising an error was important in the simulations because with two's complement arithmetic any overflow results in a sign change.

Position of the Binary Point

In order to determine the best choice of binary point position, the maximum range of possible values of all state variables needed to be determined for all realistic neuron types. This was accomplished by creating a Matlab script that randomly chose constant values for T_{mem}, C, Th_0, Tth, T_{GK}, and B from their typical ranges and simulated the resulting neuron. During the simulation of this randomly chosen neuron type the maximum and minimum values of all states were recorded. This process was repeated over and over again until a good estimate for the maximum range of values was determined (Table 1). The simulation showed that using 8-bits (+127 to -128) before the

binary point would be sufficient to represent all of the states for any realistic neuron type for which the MacGregor Point Neuron Model is applicable.

Table 1 – Empirically determined ranges for state variables by randomly adjusting all of the constants over their typical ranges.

State Variable	Max. Value	Min. Value
V_M	20	-9
Th	40	9
G_K	12	0

In addition to giving insight into the location of the binary point this simulation was also helpful in choosing methods for performing the division and exponential functions required to solve the state equations. For example, the range of potassium conductance values was used in determining the method for solving the division term 1/(1+GK) of the state equation for membrane voltage. A Taylor series expansion was ruled out because it only holds for values of GK that are less than one. Instead a look up table was used to store pre-calculated results.

Fixed Point 16-bit Numbers and the MacGregor Point Neuron Model

A simulation of the MacGregor point neuron was performed using 16-bit fixed point data types with 8 fractional bits and compared to the same simulation using double precision floating point data types (Figure 26). For this simulation typical values for T_{mem}, C, Th_0, T_{Th}, T_{GK}, and B were chosen. Also, the number of fractional bits was

reduced to 5 to show the effect on the model (Figure 27). Instead of plotting all of the states, only Vm and GK are plotted to make the graphs more clear. The degradation of the model as the precision was decreased was apparent. However, it is likely that any simulation in hardware would have at least 8 fractional bits.

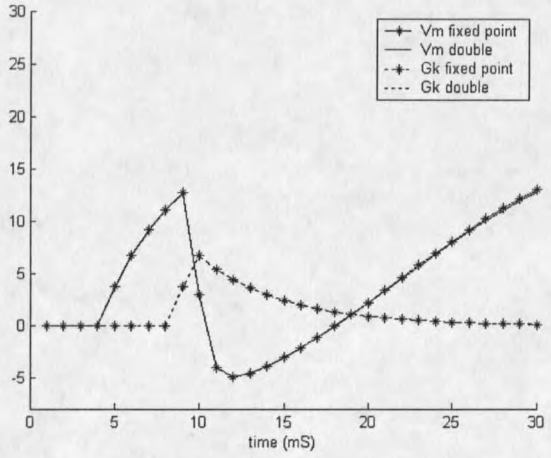

Figure 26 – Comparison of a neuron simulation using double precision floating point vs. a simulation using 16-bit fixed point with 8 fractional bits.

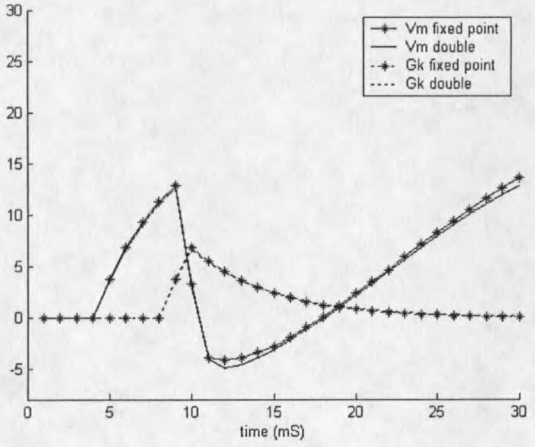

Figure 27 – Comparison of a neuron simulation using double precision floating point vs. a simulation using 16-bit fixed point with 5 fractional bits.

Simulation to Prevent Overflows

The next simulation step involves simulating the model to assure that no overflows occur during the intermediate computational steps. This process involves rewriting the Matlab code such that all of the intermediate multiplication and addition steps are accurately represented. For example, the equation tree for potassium conductance involves three arithmetic operations (Equation 23).

Equation 23

$$M_{12} = G_K(i-1) \times K_1$$
$$M_{13} = S(i-1) \times K_2$$
$$A_{11} = M_{12} + M_{13}$$
$$G_K(i) = A_{11}$$

M12 and M13 represent the multiplier outputs and A11 represents the adder outputs. It was important to rewrite the Matlab code to exactly represent the arithmetic steps that exists in hardware. Had this not been done, the standard order of operations might not have predicted an overflow condition that existed in the hardware approach. Once a Matlab script was created it was then aggressively tested over a wide range of typical cell parameters to ensure that an overflow condition was not possible.

FPGA IMPLEMENTATION

Programming hardware can be done at varying levels of abstraction as is the case with programming a typical desktop computer. High level languages such as C and Java can be used to program hardware just as they are used for CPUs, however the specific nature of this design did not allow the use of such languages. Also, these high level languages tend to produce designs that are slower and less efficient than those created using good HDL design. Instead, a Hardware Descriptive Language (HDL) such as VHDL or Verilog is a better choice because it allows for better control over how the design is implemented. Although semantically different, VHDL and Verilog are similar in terms of the resulting logic from similar designs. Verilog code is similar to C while VHDL is a more strongly typed language similar to ADA. VHDL was chosen as the programming language. Other possibilities for programming a FPGA include a schematics based entry and a core generator that automatically generates typical logic structures such as ALUs, microcontrollers, etc. It is even possible to "hand wire" the primitive logic resources together using the FPGA editor, but this is typically only done to achieve the maximum performance possible for a small design.

VHDL

Once the choice to use VHDL was made, there were very different approaches that could have been taken to solve the problem at hand. To understand these different

approaches an understanding of VHDL was necessary. VHDL is a hierarchical language for describing component interfaces and behavior. Each component consists of an entity and architecture. The entity describes the component interface. The architecture defines the functionality of the component. Typically, within the architecture section of one component, other components are used to perform sub-tasks. When pre-existing components are defined and used within an architecture it is called component *instantiation*. These instantiated components can be Xilinx primitives such as a shift registers or multipliers or they can be a user created components such as adders. In contrast, the behavior of a component can also be described through high level code in the architecture section. Although the code does not directly define what the underlying logic will be it is *inferred* through the synthesis of the code.

Both inference and instantiation are typically mixed to some degree throughout the description of a design. Both have their benefits and drawbacks. Inference is typically used when it is not important to know the underlying logic used to realize a design. Instead all that is important is that it works. For example, control logic such as an address decoder is typically inferred through the use of a case statement. This is a much better approach than creating the address decoder at the gate level. Although VHDL is considered a high level language it still can be used to instantiate and map the internal FPGA logic at the gate level. However, this approach is not feasible for large designs. Of course there are also many situations where inferring the underlying logic is not the best approach. This is especially true when creating a high performance design that utilizes Xilinx primitives such as multipliers. Multipliers can be instantiated or inferred.

However, instantiating a Xilinx primitive usually allows a much higher degree of control over the primitive. Attributes can be used to tell the implementation tools how to initialize outputs at startup, memory contents after configuration, placement in the device, and many other important features. Creating a high speed design usually involves many of these features.

Inferring a design may produce functionally correct code, but the performance may be poor because of how the underlying logic was generated. This creates a scenario where subtle differences in coding style can produce vastly different results in the underlying hardware. This process of rewriting code into a functionally equivalent form to get different results from the synthesis tools is often called "pushing on a rope". The typical example of this is using a case statement instead of a nested if structure. Both code structures are functionally equivalent, but the underlying logic created for a case statement is typically more parallel. Logic resulting from a nested if tends to have a long combinatorial path. In most cases these differences are much more subtle. However, with each new generation of synthesis tools this becomes less of an issue.

Maximizing Performance

To show the value of using a FPGA in place of a desktop computer for certain neuroscience applications a major goal was to maximize the performance. Placing a performance constraint on the design helps to define how some of the principal components should be implemented. Perhaps the best example of this involves the

embedded multiplier. Inferring multipliers is a bad idea for many reasons, but to achieve maximum performance they must be instantiated. The non-pipelined embedded multiplier can operate at 80 MHz. The pipelined version with one clock cycle latency can run at 133MHz. Since the multiplier clock frequency defines the maximum frequency of the equation trees it is important to run the multipliers as fast as possible. As it turns out the multipliers can be operated at over 200MHz quite easily through some simple performance tuning (Adhiwiyogo, 2002). The critical path that causes the poor multiplier performance can be reduced greatly by placing registers on the inputs and outputs of the multiplier, and then placing constraints on the interconnect that connects the registers to the multiplier ports. By registering the inputs and outputs to the multiplier, the delay through the interconnect from the design logic to the multiplier inputs and outputs can be minimized. This minimizes a critical path that exists on the multiplier inputs and allows for a large performance increase. By carefully designing the equation trees, they were able to run at 230 MHz. Similar care was taken when designing around the embedded block RAMs.

Another important consideration when maximizing performance is making sure to minimize any combinatorial path. Using a Register Transfer Level (RTL) design methodology the entire design can be thought of as a set of registers with some sort of logic function between them. After each clock cycle data are transferred from one register to another. Thus, the period of the clock and the setup and hold times of the registers defines the allowable delay that can occur through the logic and routing that

exists between registers. The Virtex architecture is rich with registers so this approach is well suited.

For the neuron simulation, the performance goal was to operate the system at the maximum speed of the multipliers. Thus, anywhere a combinatorial path decreased this maximum clock frequency it was split up into two paths by adding a register level. Minimizing the routing delays within the FPGA is another reason for adding registers throughout the design. A routed signal connection that crosses the entire FPGA can have considerable delay associated with it. If a signal has to cross the entire device and then go through some logic before reaching a register it can easily become the critical path. Given this emphasis on performance, the resulting design was highly parallel as is shown in the equation trees (Figures 22-24).

Coding Structure

Many levels of VHDL hierarchy were used to create a system that would simulate MacGregor point neurons in a FPGA (Figure 28). The source code for each VHDL file shown in this system is located in Appendix C.

The top level entity is the global simulation controller which initializes all of the state variables by loading their initial values into block RAM. For the most part this controller merely helped start the simulation running. A state machine is employed to initialize the neuron states in block RAM, start the simulation without state feedback, and finally run the simulation in full operation. It controls the addressing of neurons as well

as the enable lines for both ports of each block RAM. During the initial startup sequence the write port of the block RAM is disabled until information is valid on the output of the equation trees. Once the simulation is started the only job of this global simulation controller is to continually address neuron information which is then loaded into the state solver. At this time both ports of the block RAM are enabled.

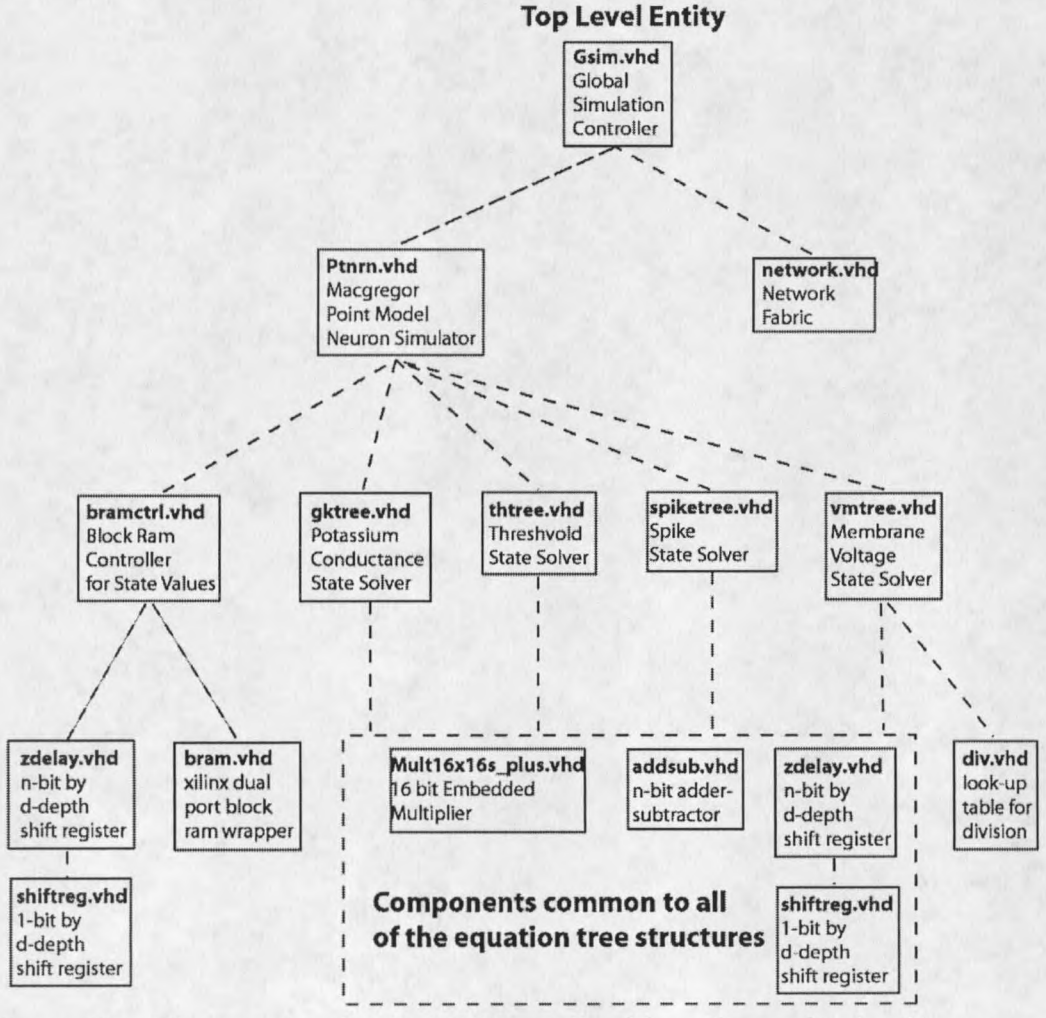

Figure 28 – Diagram showing the VHDL coding hierarchy used to implement a neuron simulation in a FPGA.

Below the global simulation controller is the point neuron simulator. This is the unit that contains all of the structures necessary to simulate the Macgregor Point Neuron Model. Each instance of the point neuron simulator can simulate up to 1024 neurons. Basically this entity instantiates and connects a particular equation tree with its corresponding block RAM controller. This combination is repeated for each of the four state variables.

The equation tree entities are built almost entirely out of multiplier, adder, and shift register components with the exception of the simple spike tree and the division look-up table used in the membrane voltage calculation. For the spike tree a comparator was created to compare the membrane voltage threshold values to determine if the neuron should be spiking or not.

Simulation:

Once the VHDL description of the process was complete it was necessary to begin the arduous task of simulating the design to verify its correctness. This was accomplished using programs such as ModelSim which compile and then simulate the VHDL design. ModelSim has built in support for all of the Xilinx primitives such as multipliers. Although this process can be performed entirely with a waveform editor, a VHDL testbench was employed to provide the input vectors for the system under test. A VHDL testbench is simply a simulation module that instantiates the design component and then applies whatever test vectors are necessary to ensure that the design works as

intended. Usually all of the internal signals in the design can be probed and examined. Also, a testbench can utilize the full VHDL language capabilities not just the small subset of the language that can be synthesized. For example, file input and output functions can be used to take test vectors from files and apply them to the unit under test. There are varying degrees to which a design can be simulated. A behavioral simulation will only test the VHDL code itself. With a more detailed simulation such as one that utilizes post place and route information, actual parameters of the logic and routing delays are used. A design that works with this level of simulation can usually be assured to work in the actual device.

Using the file input and output procedures available in VHDL it was possible to write output values to a file. This is particularly useful because the results of the neuron simulation were imported into Matlab and then plotted (Figure 29).

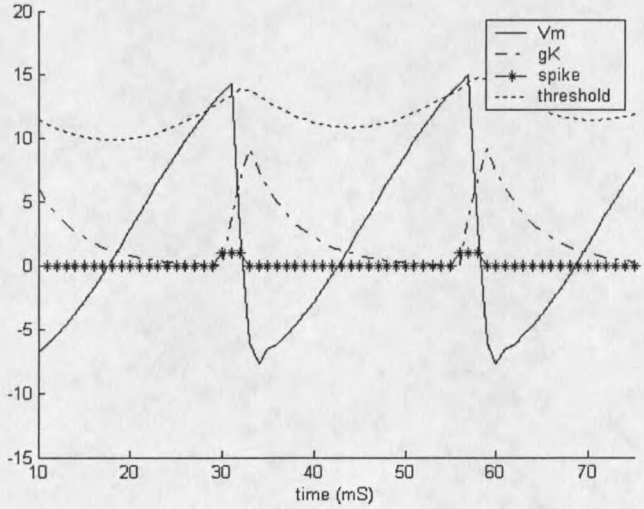

Figure 29 – Results of the MacGregor Point Neuron Model implemented in VHDL and simulated using Modelsim.

Size

After the code passed the simulation and verification stages it was ready to be implemented in a Virtex II FPGA. Once the design was simulated and verified that it was correct, a bitstream was generated and loaded into the target FPGA. After the synthesis stage the Xilinx software generates important information about the design, specifically the logic footprint and maximum operating frequency (Tables 2-3).

Table 2 – Post synthesis timing report.

Minimum period:	4.296 ns
Minimum input arrival time before clock:	2.930 ns
Maximum output required time after clock:	6.503 ns
Maximum Frequency:	232.775 MHz

Table 3 – Resource utilization for a single neuron simulator implemented within a Virtex-II 3000 FPGA

	Used	Available	%
Slices	824	14336	5
Slice Flip Flops	1291	28672	4
4 input LUTs	520	28672	1
BRAMs	5	96	5
MULT18X18s	13	96	13
GCLKs	1	16	6

At this stage it is also interesting to take a look at what the floor planned design looks like (Figure 30). The FPGA resources used for the design have been highlighted in grey.

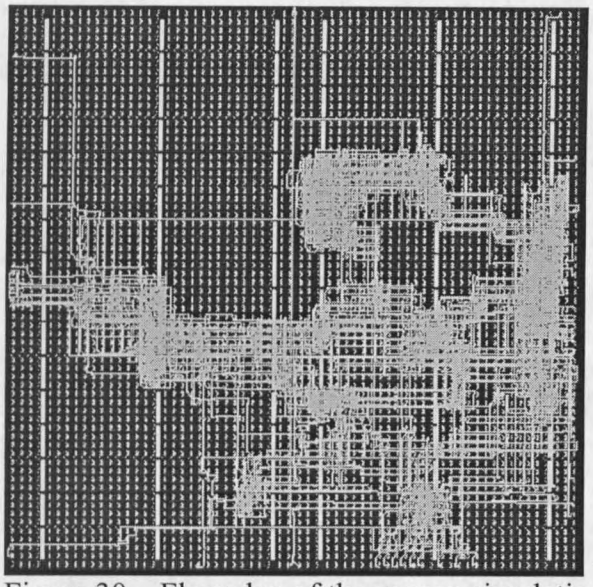

Figure 30 – Floorplan of the neuron simulation in a FPGA.

Performance

The performance of the neuron simulator is best described in neuron state updates per second. A single neuron state update is defined as the complete calculation of the four updated state variables. Each neuron state update defines the updated properties of a neuron after a simulated 1ms time step. Each neuron simulator can produce 1 neuron state update every clock cycle in an FPGA. Also each neuron simulator is capable of simulating up to 1024 different neurons. Typical desktop CPUs were used as the platforms for comparison. Simulation performance of a desktop CPU was slightly more difficult to quantify. Achieving the highest performance possible with a CPU such as the Pentium 4 is a difficult and complicated task.

As described earlier in order to achieve high performance within the FPGA architecture floating point arithmetic was not an option and a fixed point approach was taken. Since today's desktop processors are highly optimized for floating point arithmetic this is the choice for the data type used in the computer simulation. In fact it would be difficult to create a fixed point data type using C++ that would run faster than a floating point approach.

Desktop Benchmarks

A program was written in the C language to perform the neuron updates. Since each instance of the neuron simulator in the FPGA system can update 1 neuron state (4 state variable updates) per clock cycle at over 200MHz a similar simulation was performed using the C program (Appendix D). The four equations necessary to perform the same task were placed inside a loop that cycles for 200 million times. Also included in this program were system calls to obtain the system time before and after this for loop. Finally, the elapsed time was calculated by taking the difference of these two values and comparing this to the fixed time obtained from the FPGA system.

When considering code performance on a desktop CPU the first consideration was the choice of a compiler. Code execution times can vary greatly on the same processor when different compilers are used (Table 4). For this comparison the two most common compilers were used, Microsoft Visual C++ 6 and Gcc. Pentium 4 performance was terrible using these compilers so the Intel compiler was used (unsuccessfully) as a

attempt to help the poor P4 results. In all cases the compilers were instructed to optimize the resulting programs for speed. Although it is likely that much higher performance could be achieved by aggressively tailoring the algorithm to a specific processor for the purpose of these comparisons typical code utilizing standard compilers was used.

Table 4 – Performance comparison of FPGA neuron simulators versus desktop PCs for solving updated neuron states.

Platform and (Compiler Used)	Time to Update 200 Million State Updates
AMD XP 2200 + (Gcc)	69 sec.
AMD XP 2200 + (Visual C++ 6)	74 sec.
Intel P4 2.4 GHz (Intel)	452 sec.
Virtex-II 3000 with 1 Simulator	1 sec.
Virtex-II 3000 with 5 Simulators	.2 sec
Virtex-II Pro 125 with 45 Simulators	.02 sec

To make sure the current generation of CPUs is compared with the highest performing generation of FPGAs the synthesis step was re-run targeting the highest performance FPGA currently available, the Xilinx Virtex-II Pro 125 (Table 5).

Immediately apparent is the huge performance advantage of the FPGA platform. The FPGA approach is almost 70 times faster than the Athlon CPU when just 1 simulator implementation is running in a FPGA. By placing 5 simulators in the FPGA the performance increases to over 350 times faster than the Athlon CPU. It would theoretically be possible to fit close to 10 neuron simulators within a single Virtex II 3000 part, but 5 simulators is a more realistic number due to routing and thermal issues.

The results of using the Virtex-II Pro 125 FPGA are very impressive with a speedup of over 3000 times faster than a 2GHz desktop machine.

Table 5 – Resource utilization for a single neuron simulator implemented in a Virtex-II Pro 125 FPGA

	Used	Available	%
Slices	824	57664	1
Slice Flip Flops	1291	115328	1
4 input LUTs	520	115328	0
BRAMs	5	612	0
MULT18X18s	13	612	2
GCLKs	1	16	6

Network

The final stages of design involved the creation of a simple network of neurons merely as a way of demonstrating that the model works. Although very sophisticated networking should be possible without any significant performance degradation for the purposes of this thesis, only a very simple ring network will be considered. Because the MacGregor Point Neuron Model only models the triggering section of a neuron, a network was not particularly meaningful. A more realistic model would have to describe the interactions through the dendritic tree and the funneling into the trigger section of the

neuron. Nevertheless, a rudimentary network can be created by using the spiking of one neuron as the stimulus mechanism for another neuron.

Ring Network

For this simple ring network a neuron will be the stimulus source for another neuron in a circular fashion. Under these conditions a large ring of neurons can be assembled in a manner such that a action potential will propagate in a circular fashion around the ring. Again, Matlab was used to first model this scenario using the fixed point neuron model. A network of three neurons was modeled to show how an action potential propagates around the network (Figure 31). Here there is a delay of about 6 ms (simulated time) between the firing of two connected neurons. The delay represents the time it takes for a neuron to fire after the onset of a current step function.

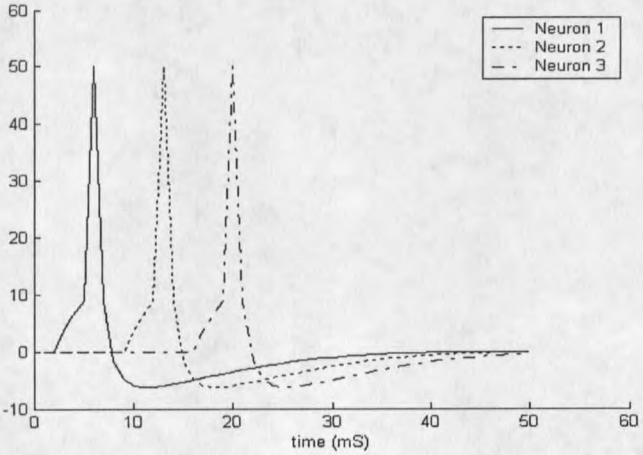

Figure 31 – Simulated recordings from three different neurons connected in a simple ring network topology.

The last step before implementing the network in a FPGA was to simulate the network in ModelSim (Figure 32). It was possible to show the firing of each neuron by observing the spike variable output of the neuron simulator. During the firing phase of each neuron, the spike state was shown to remain high for two clock cycles.

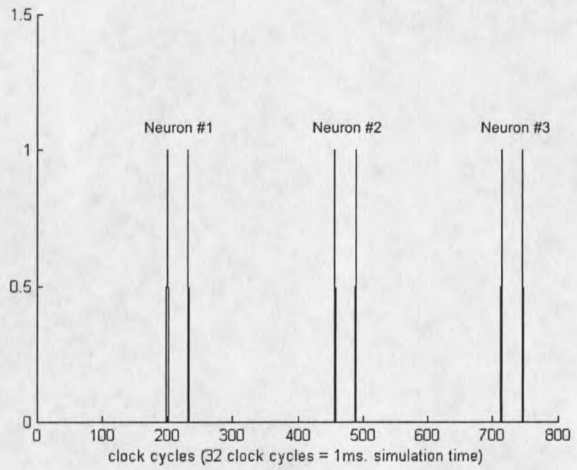

Figure 32 – Spike output taken from a VHDL simulation of a ring network of MacGregor Point Neurons.

Finally, the network was implemented on the Virtex-II 3000 FPGA (Figure 33). The reason for such a simple network was to demonstrate a functional network of interacting neurons using the limited interface that was available via the FPGA board at that time. Saving the 16-bit values of all four state variables was not possible, but the binary spike state was easily monitored with a oscilloscope. The spike state was output to a single pin and was monitored and the resulting oscilloscope waveform data points

were saved and plotted in Matlab. The actual output was the same as predicted by the simulation.

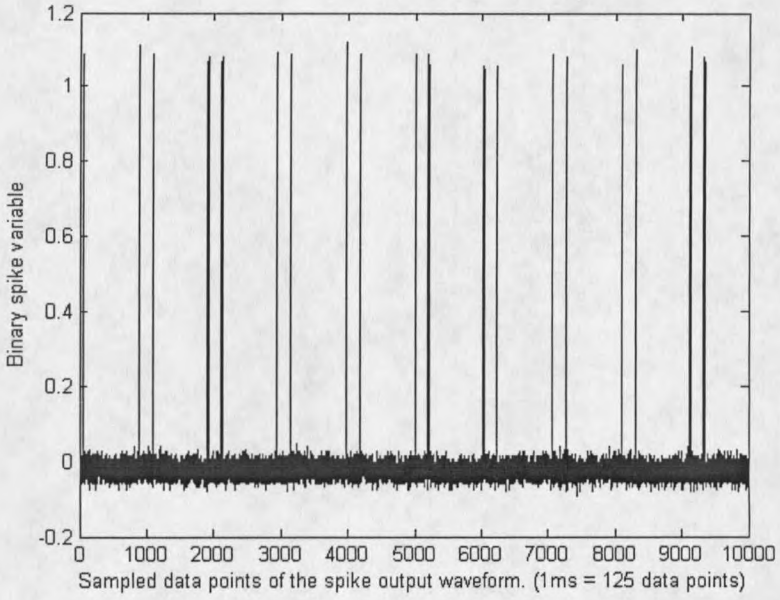

Figure 33 – Sampled spike waveform taken from a ring network of 30 neurons running in a FPGA at 5MHz.

CONCLUSION

Although the MacGregor Point Neuron is not widely used, its implementation within the architecture of a FPGA was an important step in showing the usefulness of these devices for neuroscience research. Such modeling is typically done using computers because they are relatively easy to program. Although FPGAs are generally more difficult to program they can provide huge performance advantages over computers.

Methods were demonstrated for adapting a model that used floating point arithmetic to a model that used fixed point arithmetic while maintaining the integrity of the model. When the model was implemented in a FPGA a computational speedup of several orders of magnitude over a typical high performance desktop CPU was demonstrated. By using such an approach, a years worth of computations on a desktop CPU could be reduced down to a single day of computational time on the FPGA system.

A very simple network was used to show the possibility of creating a functional network out of simulated neurons. A more realistic network would need to model neuron interaction more realistically as well as provide a means for any neuron to effect any other neuron in the network.

Further study is necessary to find computational bottlenecks and determine the possibility of their speedup using FPGA systems. As this thesis demonstrates, such systems appear to hold significant promise for neuroscience research.

BIBLIOGRAPHY

Andersson JA, Rosenfeld E. Neurocomputing: Foundations of Research. Cambridge, MA: MIT Press. 1988.

Adhiwiyogo, Markus. "Optimal Pipelining of the I/O Ports of Virtex-II Multipliers." <http://www.xilinx.com/xapp/xapp636.pdf>

Chapman, Ken. "Creating Embedded Microcontrollers (Programmable State Machines). Parts 1-5" Xilinx TechXclusives 1 Jan. 2002. Aug. 30 2002 <http://www.xilinx.com/support/techxclusives/techX-home.htm>

Hill, A. V. "Excitation and accommodation in nerve." Proc. R. Soc. London, 119 (1936): 305-355.

Hines, M. The NEURON simulation program. In: Neural Network Simulation Environments, edited by J. Skrzypek. Norwell, MA: Kluwer, 1994, p. 147-163.

Hodgkin, A.L. and Huxley, A.F. "A quantitative description of membrane current and its application to conduction and excitation in nerve." J. Physiol. 117(1952): 500-544.

Hopfield, John J. "Pattern recognition computation using action potential timing for stimulus representation." Nature 376 (1995): 33-36.

Johnston, Daniel, & Samuel WU. Foundations of Cellular Neurophysiology. Cambridge, MA: MIT Press. 1995.

Kernell, D. "The repetitive impulse discharge of a simple neuron model compared to that of spinal motoneurons." Brain Res. 11(1968): 685-687.

Lapicque, L. "Recherches quantitatives sur l'excitation electrique des nerfs traitee comme une polarization." J. Physiol. Pathol. Gen. 9(1907): 620-635.

MacGregor, Ronald J. Neural and Brain Modeling. San Diego: Academic Press, Inc. 1987.

Nelson, ME, W Furmanski and JM Bower. Simulating Neurons and Networks on Parallel Computers. In: Methods in Neural Modeling: From Synapses to Networks edited by C. Koch and I. Segev. Cambridge, MA: MIT Press, 1989, p. 397-437.

Perkel, DH , B Mulloney and RW Budelli. "Quantitative methods for predicting neuronal behavior." Neuroscience 6 (1981): 823- 837.

Rall, W. Cable theory for dendritic neurons. In: Methods in Neural Modeling: From Synapses to Networks, edited by C. Koch and I. Segev. Cambridge, MA: MIT Press, 1989, p. 8-62.

Zohary, E, MN Shadlen and WT Newsome. "Correlated neuronal discharge rate and its implications for psychophysical performance." Nature 370 (1994): 140-143.

APPENDICES

Appendix A

Matlab Code for the MacGregor Model Neuron

```
% Chip Lukes
% 3/8/2003
%
% Translation of MacGregor's fortran implementation of the
% point neuron 10
% Neural and Brain Modeling - page 458
%
clear all

%simulation parameters
step = 1; % simulation time step (ms.)
tstart = 1;  % start time
tend = 51; % end time of simulation

%input current parameters
Cstart = 5;  %start time for current pulse into dendtrites
Cend = 8;  %end time of current pulse into dendrites
Camp = 20; %amplitude of current pulse

%constants
C= 1;  %Threshold sensitivity (0-1)
Tth= 25;  %Time constant for accomodation (20-25 ms)
B= 20;  %Sensitivity to Potassium conductance typically 20
Tgk= 5;  % Refractory time constant for Potassium (3-10 ms)
Th0= 10;  % Initial threshold (10-20 mV)
Tmem= 5;  % Membrane time constant (5-11 ms)
Ek= -10;  % Resting Potassium potential (-10 mV)
Dcth= exp(-step/Tth);  % Decay constant for threshold
Dgk= exp(-step/Tgk);  %Decay constant for Potassium action

%state variables
Vm=zeros(tstart,tend);  %membrane voltage
Th=zeros(tstart,tend);  %Threshold
S=zeros(tstart,tend);  %Spike
Gk=zeros(tstart,tend);  %Potassium conductance

%initialize state variables
Vm(1)= 0; %initial membrane voltage
Th(1)= Th0; %initial Threshold
S(1)= 0; %initial Spike
Gk(1)= 0; %initial Potassium conductance

%initialize output vector
```

```
P=zeros(tstart,tend);  %familiar cell potential output

%start of simulation
for i= tstart+1:step:tend
   %current pulse
   if (i >= Cstart && i <= Cend)
      Iin = Camp;
   else
      Iin= 0;
   end

   Gtot=1+Gk(i-1);  %intermediate values used in state calcs.
   Dce=exp(-Gtot*step/Tmem);  % " "

   Gk(i)=Gk(i-1)*Dgk+B*S(i-1)*(1-Dgk);  %update potassium conductance
   Vm(i)=Vm(i-1)*Dce+(Iin+Gk(i-1)*Ek)*(1-Dce)/Gtot;  %update membrane voltage
   Th(i)=Th0+(Th(i-1)-Th0)*Dcth+C*Vm(i-1)*(1-Dcth);  %update threshold

   %update spike (works better if function of present values)
   if Vm(i) >= Th(i)
      S(i)=1;
   else
      S(i)=0;
   end

   P(i)=Vm(i-1)+S(i-1)*(50-Vm(i-1));

end

%Plotting results
time=tstart:step:tend;

hold on
plot(time,Vm,'k-');
plot(time,Gk,'k-.');
plot(time,S,'k*-');
plot(time,Th,'k:');
axis([0 25 -8 15]);
xlabel('time (mS)')
legend('Vm', 'gK', 'spike', 'threshold')
figure
plot(time,P,'k');
xlabel('time (mS)')
```

```
legend('P (Simulated Membrane Voltage)')
axis([0 40 -20 50]);
```

Appendix B

FPGA Board User's Manual

System Overview

The FPGA board is one part of a Reconfigurable Online Modeling Platform (ROMP) that utilizes the combination of Digital Signal Processors (DSPs) and Field Programmable Gate Arrays (FPGAs) for real time analysis of neural systems as well as general signal processing. Two boards have been developed for this task, one utilizing a DSP and another utilizing an FPGA. The two boards act as a single processing node where communication takes place via a connector that shares the address and data bus of the DSP. Communication can also take place via high speed data links that are present on the FPGA board and DSP board. This allows for networks of nodes to be assembled for a fully scalable processing system.

FPGA Board Overview

The FPGA board consists of the following key components: a FPGA where reconfigurable computing takes place, a Complex Logic Device (CPLD) that is responsible for controlling the board at boot up, programmable ROM which holds FPGA configuration data, a Double Data Rate (DDR) SODIMM connector that can provide up to 512 MB of memory for each node, and high speed transceiver devices which provide inter-nodal communication. A system view of this board is shown in Figure 1.

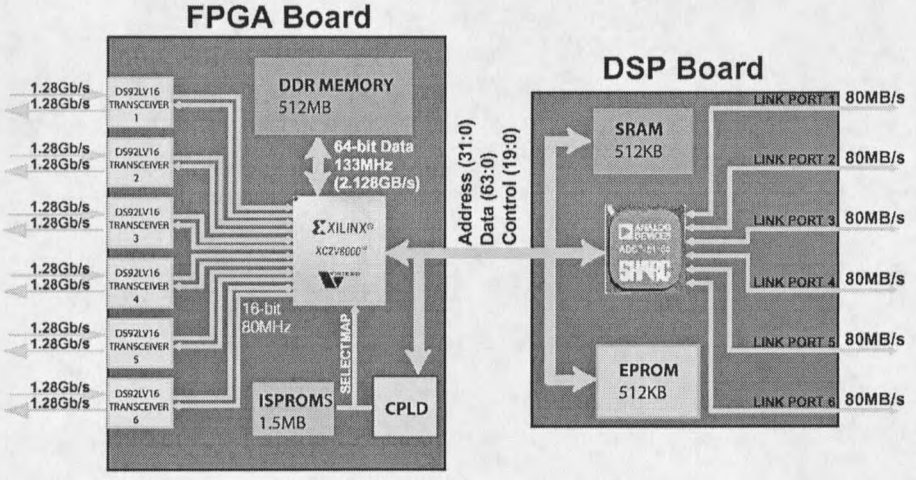

Figure 1 - Functional diagram of a processing node consisting of a DSP board and a FPGA board that share a common address and data bus.

FPGA. The heart of the FPGA board consists of a Virtex-II FPGA. Specifically the board is designed for a Virtex-II 3000 part but since the BF957 package is pin compatible with larger devices the board can easily be upgraded.

FPGA architectures such as the Virtex-II are ideal for meeting the re-configurable computing requirements of this system. In addition to vast logic resources there are also dedicated RAM and multiplier structures which are specially designed for doing a Multiply and Accumulate (MAC) operations. The reason for this is the MAC operation is a common occurrence in many signal processing algorithms such as FIR filtering. Next to each embedded multiplier structure exists 18Kbit blocks of dual port RAM. Thus, both 18-bit inputs to a multiplier can be supplied in one cycle.

In addition to a co-processing resource, the FPGA also acts as a network switch for all of the communications taking place via the high speed transceiver devices. Each transceiver device operates on 16-bit send and receive busses operating at 80MHz. Nodes that are not the source or destination of data streams need to efficiently pass communications through in a manner such that short paths are taken. The Virtex-II 3000 has over 600 available IO pins and easily accomplishes this task.

<see Xilinx Virtex II datasheet>

CPLD. The Xilinx XC95144XL CPLD acts as the boot sequencer and board controller. The primary purpose of the CPLD is to control the board initialization after a reset or power cycle has occurred. Like the FPGA the CPLD is also a reconfigurable logic device, although the architecture only supports limited functionality. Unlike the FPGA, the CPLD is a non-volatile device allowing configuration information to be retained while the device is not powered. Typically the role of a cpld is to act as glue logic between components. On the FPGA board CPLD controls the IsPROMs and FPGA during bootup (see Bootup section) and also acts as an interface for the DSP to reconfigure the FPGA.

<see Xilinx XC95144XL datasheet>

IsPROMS. The Xilinx XC18V04 in system PROMs (Programmable ROMs) are necessary to hold configuration bitstreams for the FPGA. Each device holds up to 524 KB of data, and to accommodate the large (1.3MB) bitstream of the Virtex-II 3000 device a chained configuration of 3 IsPROMs is necessary (see bootup section). The devices interface to the FPGA using the SelectMAP configuration scheme and are easily programmed using a JTAG interface cable. On the FPGA board this JTAG interface is part of a chain that includes all IsPROMs and the CPLD.

<see Xilinx XC18V04 datasheet>

LVDS Transceivers. The DS92LV16 bus LVDS (Low Voltage Differential Signal) serializer/deserializer devices are used for inter-board communication. A 16-bit LVCMOS bus operating at 80Mhz is translated into a LVDS serial data stream operating at 1.28 Gb/s with embedded clock information. Each device has independent receive and transmit busses allowing for 2.56Gb/s of full duplex throughput. In addition, PLLs (Phase Locked Loops) recover the embedded clock signal from the serial data stream. The serial data streams can take place over any twisted pair cable, but in order to accommodate frequencies above 1GHz higher performance cables such as twin-axial are necessary. Special connectors on the FPGA board are designed to accommodate this particular cable.

LVDS is a differential signaling transmission method that allows data communication to take place using a very low voltage swing, typically about 350 mV. Because the signal is differential it is very immune to common mode noise and produces low EMI. The low voltage signal results in very low power consumption.

Six DS92LV16 devices are present on each board which enables the board to have bidirectional communication with each nearest neighbor when the boards are placed in a 3-dimensional configuration as shown in Figure 2.

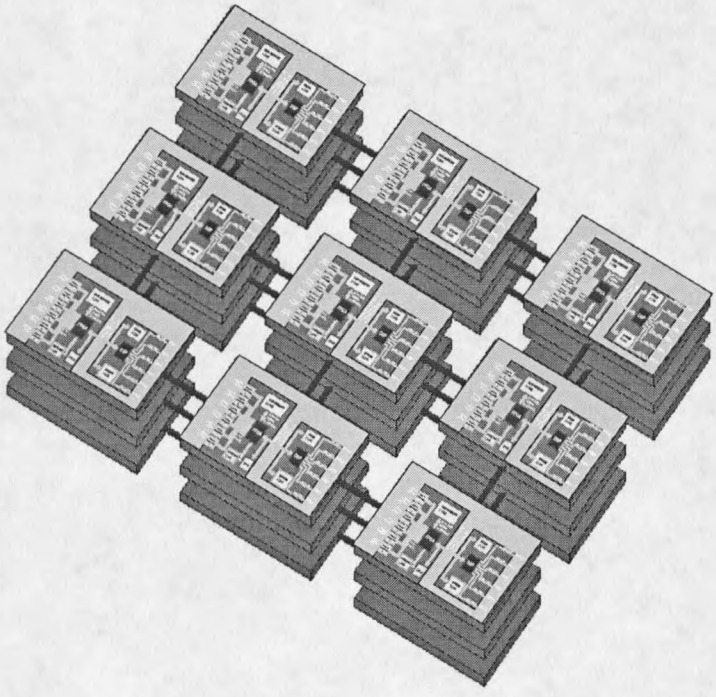

Figure 2 Diagram Showing a 3x3x3 Array of Processing Nodes

DDR. In order to provide a flexible memory solution for the FPGA board a SODIMM connector was used. The connector will accept any size DDR SODIMM memory stick, but was designed specifically around a 512 MB Micron DDR SODIMM. DDR SDRAM provides over 2 GB/s of data throughput providing a high speed storage solution for the FPGA.

The DDR SDRAM operates on the SSTL-2 signaling method. Utilizing LVTTL at frequencies greater than 133MHz becomes problematic thus Stub-Series Tranceiver Logic (SSTL) is used with all high speed DRAMs. As implied by the name, both stub and series termination resistors are used as well as a lower operating voltage level (2.5V with SSTL-2). The stub termination resistors can be seen surrounding the SODIMM connector on the printed circuit board and terminate to 1.25V. The series termination resistors are present within the Virtex-II when the I/O pins are configured to implement the SSTL-2 standard.

Careful layout of the printed circuit board was necessary to ensure proper operation of the high speed DDR circuitry. In particular, as the design guidelines recommend, lengths of traces within the same data group (ie. Data bits 0-7 + 2 control lines) are within .1 in. Also, across all data groups, the largest trace length difference does not exceed .5 in. It was also necessary to maintain at least 15 Mils between adjacent pcb traces in order to limit crosstalk.

A 2.5V phase lock loop clock driver was necessary to take the differential clock coming from the FPGA and distribute to the 3 different clock inputs of the SODIMM connector. The PLL in the clock driver aligns all clock outputs to the incoming clock from the FPGA. From the clock driver to the SODIMM connector the clock lines are all the same length to eliminate skew.

A ML6554 3 Amp bus termination regulator was required to source and sink the large currents associated with all of the 1.25V SSTL-2 termination resistors. In addition to 1.25V outputs, a Vref output is also provided. Vref is an output that is exactly half of the supply voltage for the SSTL-2 I/O which is nominally 2.5V, but can fluctuate depending on the load. A very precise Vref signal is required by the FPGA as well as the SODIMM.

<see Micron datasheet>

Power Distribution. A 5V input voltage for the FPGA board is supplied by connecting a 5V source to the pin 3 of the power block (see board map). Pins 1&2 are tied to ground. This 5V voltage is then regulated down to the 3 voltage levels used on the board: 3.3V, 2.5V and 1.5V. All components on the board operate at 3.3V with the exception of the DDR SDRAM and FPGA.

The FPGA has requires separate voltages for both its internal logic (Vccint) and output drivers (Vcco). The internal logic level of the Virtex-II is 1.5V and an entire plane of the printed circuit board is devoted to this level. An entire plane of the printed circuit was also used for 3.3V since it was used throughout the board. The FPGA can be divided into 8 separate I/O banks, where each bank operates on a different I/O standard. Different I/O standards utilize different voltage levels and termination schemes. The Virtex-II FPGA supports 25 different I/O standards, but for the FPGA board only SSTL-2 and LVTTL were used.

The DDR SDRAM operates using SSTL-2 I/O standard thus the two I/O banks of the FPGA that interface to the SDRAM operated with Vcco at 2.5V. The 2.5V level was not given an entire printed circuit board plane, instead a pour under the SODIMM connector and part of the FPGA was enough to suffice.

According to JEDEC DDR specifications a specific power up sequence is necessary to ensure proper SDRAM initialization. Power must first be applied to the 2.5V source and then to the Vtt and Vref voltages (1.25V). This power up sequence is realized using the circuit shown in Figure 3. The power good (PG) output of one regulator is used as the enable pin for the next regulator in the sequence as shown in Figure 3.

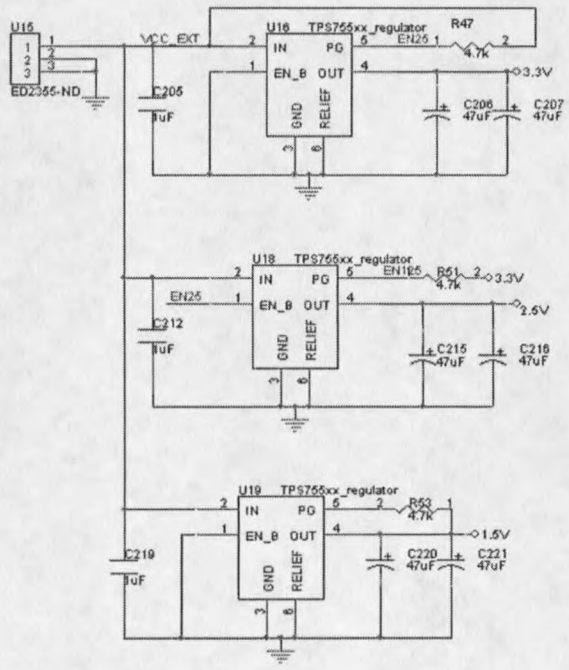

Figure 3 – Power-up sequencing and regulation circuitry.

Initialization and Bootup

FPGAs are volatile devices requiring configuration each time a power cycle occurs. For the Xilinx Virtex-II there are multiple configuration options that are selected via the mode bits as shown in Table 5. For the highest degree of configurability both slave SelectMAP and Boundary Scan (JTAG) were chosen for the FPGA board. The mode bits for selecting the configuration bit are made available through jumpers J14, J15, and J17 on the FPGA board.

Table 1. Virtex-II Configuration Mode Pin Settings

Configuration Mode	M2	M1	M0	CCLK Direction	Data Width	Serial Dout[2]
Master Serial	0	0	0	Out	1	Yes
Slave Serial	1	1	1	In	1	Yes
Master SelectMAP	0	1	1	Out	8	No
Slave SelectMAP	1	1	0	In	8	No
Boundary Scan	1	0	1	N/A	1	No

Under typical operation the DSP is responsible for loading the bitstreams necessary for configuring the FPGA board devices. This is accomplished using the SelectMAP configuration mode. This is the fastest way for programming the FPGA device and is typically the solution when microprocessor or DSP is available. This also allows the FPGA can be configured at speeds up to 66MHz. A CPLD is then used to act as an address decoder for the DSP which then programs the FPGA through a set of configuration registers located in the DSP's address space. A diagram of a typical SelectMap configuration is shown in Figure 4 with pin descriptions shown in Table 1..

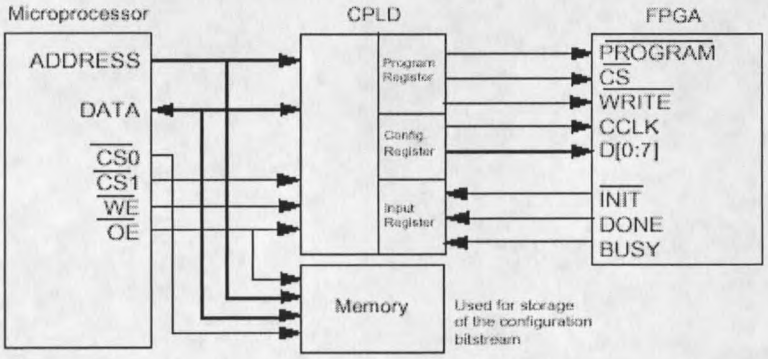

Figure 4 – Diagram showing a typical SelectMap configuration.

Table 2 – SelectMap Pin Descriptions.

Signal Name	Direction	Description
CCLK	Input	Configuration clock
PROG	Input/Output	Asynchronous reset to configuration logic. Indicates when device has cleared configuration memory.
INIT	Input/Output	Input to delay configuration. Indicates when device is ready to receive configuration data; also indicates configuration errors.
DONE	Input/Output	Input to delay device startup. Indicates when configuration is in the startup sequence.
M[2:0]	Input	Configuration Mode selection
D[7:0]	Input	Byte-wide configuration data input
$\overline{CS}$	Input	Active Low Chip Select input
$\overline{WRITE}$	Input	Active Low Write Select/Read Select
BUSY	Output	Handshaking signal to indicate successful data transfe (same pin as DOUT in Serial mode).

In order to accommodate device configuration independent of the DSP, a default boot up through In System Programmable ROMS (ISPROMs) were added to the board. To accommodate the large configuration file size of the xc2v3000 FPGA 3 ISPROMs are necessary. Depending upon how the cpld is configured either the DSP or the ISPROMS can be the source of the programming data stream. Figure 5 shows a schematic of the SelectMap configuration utilizing the Xilinx XC18V04 ISPROMs

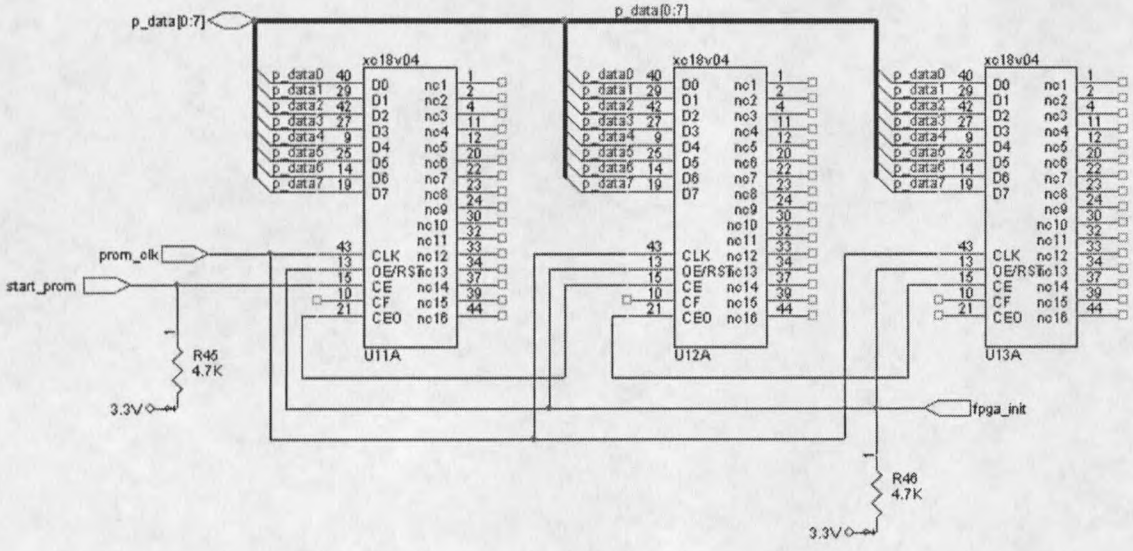

Figure 5 - IsPROM circuit required for storing and loading the Virtex-II 3000 bitstream.

The CPLD is connected to the programming pins on the chain of ISPROMs as well as the FPGA. Furthermore, the CPLD has access to the address and data lines from the DSP board. The CPLD is programmed to act as a controller for the bootup of the FPGA and can be reprogrammed to realize whatever configuration scheme is required. For instance, the CPLD could be programmed to use the ISPROMs as the configuration source at bootup and later allow the DSP to re-configure the FPGA as necessary. This will be the typical operation of the CPLD. Figure 6 shows how the CPLD acts as a hub for configuring the FPGA.

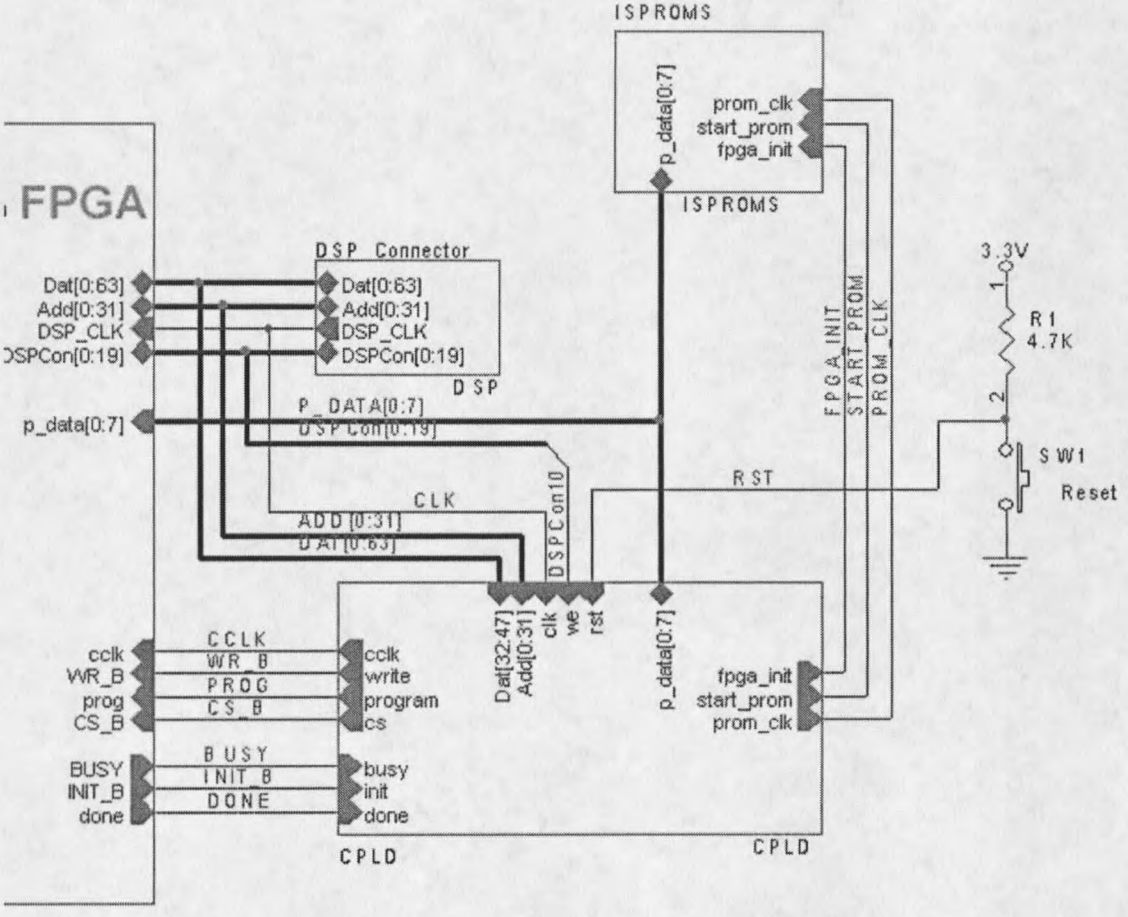

Figure 6 – Diagram of CPLD bootup controller circuit.

In addition to the multiple SelectMAP programming sources the option to program the FPGA via JTAG was added. JTAG is used for programming the ISPROMs and CPLD which are both non-volatile. Also, JTAG allows for device chaining so the ISPROMS and CPLD are all part of the same programming chain as shown in Figure 7.

The FPGA was kept on a separate chain because it would not typically be programmed at the same time as the other devices and would typically only be used for debugging. JTAG is a much slower configuration method than selectMAP, but it is necessary for the initial configuration of the CPLD and ISPROMs.

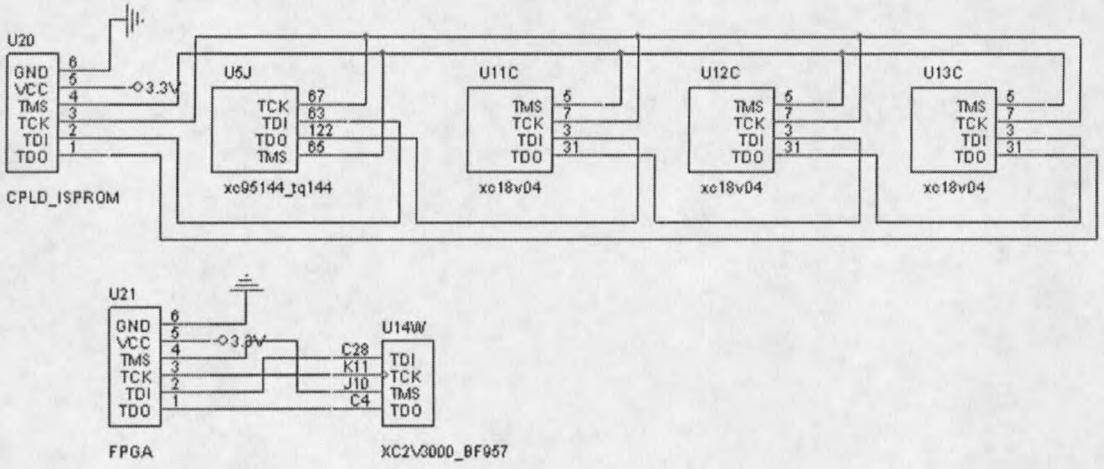

Figure 7 – Schematic of JTAG chain.

Board Features Map

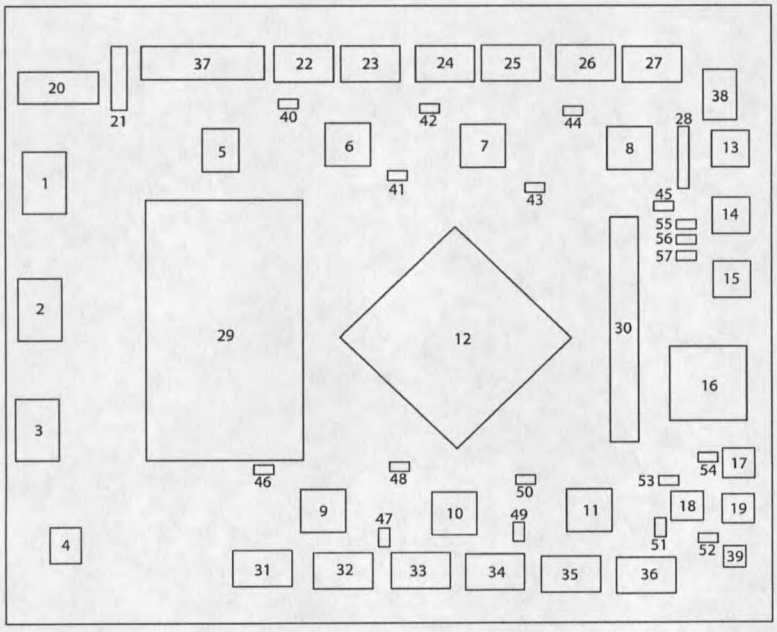

Figure 8 - Component, jumper, and connector map of FPGA board.

Table 3 - ICs

#	Device
1	TPS75533 3.3V Regulator
2	TPS75525 2.5V Regulator
3	TPS75515 1.5V Regulator
4	ML6554 3A DDR Bus Termination Regulator
5	CDCV857 2.5V Phase Lock Loop Clock Driver
6	Transceiver #1 DS92LV16 16-Bit Bus LVDS Serializer/Deserializer
7	Transceiver #2 DS92LV16 16-Bit Bus LVDS Serializer/Deserializer
8	Transceiver #3 DS92LV16 16-Bit Bus LVDS Serializer/Deserializer
9	Transceiver #4 DS92LV16 16-Bit Bus LVDS Serializer/Deserializer
10	Transceiver #5 DS92LV16 16-Bit Bus LVDS Serializer/Deserializer
11	Transceiver #6 DS92LV16 16-Bit Bus LVDS Serializer/Deserializer
12	Virtex-II 3000 FPGA
13	XC18V04 In-System Programmable Configuration PROMS
14	XC18V04 In-System Programmable Configuration PROMS
15	XC18V04 In-System Programmable Configuration PROMS
16	XC95144XL CPLD
17	CSX750A Oscillator
18	CSX750A Oscillator
19	CSX750A Oscillator

Table 4 - Connectors

#	Device
20	Power Connector
21	FPGA JTAG Connector
22	Transceiver #1-Snd LVDS Connector
23	Transceiver #1-Rcv LVDS Connector
24	Transceiver #2-Snd LVDS Connector
25	Transceiver #2-Rcv LVDS Connector
26	Transceiver #3-Snd LVDS Connector
27	Transceiver #3-Rcv LVDS Connector
28	CPLD & IsPROM JTAG Connector
29	DDR SODIMM Connector
30	DSP Board Interface Connector
31	Transceiver #4-Rcv LVDS Connector
32	Transceiver #4-Snd LVDS Connector
33	Transceiver #5-Rcv LVDS Connector
34	Transceiver #5-Snd LVDS Connector
35	Transceiver #6-Rcv LVDS Connector
36	Transceiver #6-Snd LVDS Connector

Table 5 - Jumpers

#	Device
40	J3 - Tranceiver #1 Receiver Power Down (Jumper in place)
41	J4 - Tranceiver #1 Transmitter Power Down (Jumper in place)
42	J1 - Tranceiver #2 Receiver Power Down (Jumper in place)
43	J2 - Tranceiver #2 Transmitter Power Down (Jumper in place)
44	J5 - Tranceiver #3 Receiver Power Down (Jumper in place)
45	J6 - Tranceiver #3 Transmitter Power Down (Jumper in place)
46	J8 - Tranceiver #4 Transmitter Power Down (Jumper in place)
47	J7 - Tranceiver #4 Receiver Power Down (Jumper in place)
48	J10 - Tranceiver #5 Transmitter Power Down (Jumper in place)
49	J9 - Tranceiver #5 Receiver Power Down (Jumper in place)
50	J12 - Tranceiver #6 Transmitter Power Down (Jumper in place)
51	J11 - Tranceiver #6 Receiver Power Down (Jumper in place)
52	J20 - Oscillator Output Enable (Jumper in place)
53	J18 - Oscillator Output Enable (Jumper in place)
54	J16 - Oscillator Output Enable (Jumper in place)
55	J14 - FPGA M0 configuration bit (Jumper pulls Low)
56	J15 - FPGA M1 configuration bit (Jumper pulls Low)
57	J16 - FPGA M2 configuration bit (Jumper pulls Low)

Table 6 - LEDs and switches

#	Device
37	FPGA LED Bank
38	CPLD LED Bank
39	Board Reset Switch

FPGA Pinouts

Table 6 - DDR Interface

Net	Pin	Net	Pin	Net	Pin
A0	P2	DQ15	A7	DQ49	G14
A1	N1	DQ16	B7	DQ5	B13
A10	P1	DQ17	B6	DQ50	F12
A11	K2	DQ18	A4	DQ51	F11
A12	K3	DQ19	D3	DQ52	F15
A2	N3	DQ2	B12	DQ53	F14
A3	M1	DQ20	A6	DQ54	G11
A4	M2	DQ21	C6	DQ55	G10
A5	M3	DQ22	B3	DQ56	G9
A6	L1	DQ23	D2	DQ57	G8
A7	L2	DQ24	D1	DQ58	H7
A8	L3	DQ25	E2	DQ59	J7
A9	K1	DQ26	G2	DQ6	C12
BA0	R3	DQ27	H3	DQ60	F9
BA1	R1	DQ28	E3	DQ61	F8
CAS#	E12	DQ29	F3	DQ62	H6
CKE0	J1	DQ3	A11	DQ63	J6
CKE1	J3	DQ30	G1	DQ7	B11
ddr_clk	E15	DQ31	H1	DQ8	C11
ddr_clk_n	D15	DQ32	D12	DQ9	B10
ddr_clk	H16	DQ33	D11	DQS0	C13
ddr_clk_n	H15	DQ34	D8	DQS1	A9
DM0	A12	DQ35	D7	DQS2	B5
DM1	B9	DQ36	E11	DQS3	F2
DM2	C5	DQ37	E10	DQS4	E9
DM3	F1	DQ38	E7	DQS5	H5
DM4	E8	DQ39	D6	DQS6	F13
DM5	H4	DQ4	B15	DQS7	F7
DM6	G12	DQ40	F5	RAS#	E13
DM7	G6	DQ41	G5	S0#	D13
DQ0	A15	DQ42	J5	SA0	P7
DQ1	A13	DQ43	K5	SA1	R7
DQ10	C9	DQ44	F4	SA2	R6
DQ11	C8	DQ45	G4	SCL	P6
DQ12	A10	DQ46	J4	SDA	N6
DQ13	C10	DQ47	L5	WE#	D14
DQ14	A8	DQ48	G15		

Table 7 - LVDS transceiver interface 1-3

Net	Pin	Net	Pin	Net	Pin
TX1_DEN	C17	TX2_DEN	A24	TX3_DEN	H29
TX1_DIN0	C19	TX2_DIN0	C26	TX3_DIN0	J30
TX1_DIN1	B19	TX2_DIN1	B26	TX3_DIN1	J31
TX1_DIN10	B22	TX2_DIN10	D31	TX3_DIN10	M31
TX1_DIN11	A22	TX2_DIN11	E29	TX3_DIN11	N29
TX1_DIN12	C23	TX2_DIN12	E30	TX3_DIN12	N30
TX1_DIN13	B23	TX2_DIN13	E31	TX3_DIN13	N31
TX1_DIN14	A23	TX2_DIN14	F29	TX3_DIN14	P30
TX1_DIN15	C24	TX2_DIN15	F30	TX3_DIN15	P31
TX1_DIN2	A19	TX2_DIN2	A26	TX3_DIN2	K29
TX1_DIN3	C20	TX2_DIN3	C27	TX3_DIN3	K30
TX1_DIN4	B20	TX2_DIN4	B27	TX3_DIN4	K31
TX1_DIN5	A20	TX2_DIN5	A27	TX3_DIN5	L29
TX1_DIN6	C21	TX2_DIN6	A28	TX3_DIN6	L30
TX1_DIN7	B21	TX2_DIN7	B29	TX3_DIN7	L31
TX1_DIN8	A21	TX2_DIN8	D29	TX3_DIN8	M29
TX1_DIN9	C22	TX2_DIN9	D30	TX3_DIN9	M30
TX1_LINE_LE	E18	TX2_LINE_LE	G20	TX3_LINE_LE	G28
TX1_LOCAL_LE	D17	TX2_LOCAL_LE	F19	TX3_LOCAL_LE	G27
TX1_RCLK	A18	TX2_RCLK	A17	TX3_RCLK	E16
TX1_REN	B18	TX2_REN	A25	TX3_REN	J29
TX1_ROUT0	E28	TX2_ROUT0	L26	TX3_ROUT0	T27
TX1_ROUT1	D26	TX2_ROUT1	L25	TX3_ROUT1	R28
TX1_ROUT10	E21	TX2_ROUT10	F23	TX3_ROUT10	L27
TX1_ROUT11	D20	TX2_ROUT11	G23	TX3_ROUT11	K27
TX1_ROUT12	E20	TX2_ROUT12	G22	TX3_ROUT12	J28
TX1_ROUT13	D19	TX2_ROUT13	F21	TX3_ROUT13	J27
TX1_ROUT14	E19	TX2_ROUT14	G21	TX3_ROUT14	H28
TX1_ROUT15	D18	TX2_ROUT15	F20	TX3_ROUT15	H27
TX1_ROUT2	D25	TX2_ROUT2	K25	TX3_ROUT2	R27
TX1_ROUT3	E25	TX2_ROUT3	J26	TX3_ROUT3	P28
TX1_ROUT4	D24	TX2_ROUT4	J25	TX3_ROUT4	P27
TX1_ROUT5	E24	TX2_ROUT5	H26	TX3_ROUT5	N28
TX1_ROUT6	D23	TX2_ROUT6	H25	TX3_ROUT6	N27
TX1_ROUT7	E23	TX2_ROUT7	G26	TX3_ROUT7	M28
TX1_ROUT8	E22	TX2_ROUT8	F24	TX3_ROUT8	M27
TX1_ROUT9	D21	TX2_ROUT9	G24	TX3_ROUT9	L28
TX1_TCLK	B17	TX2_TCLK	B25	TX3_TCLK	H31

Table 8 - LVDS transceiver interface 4-6

Net	Pin	Net	Pin	Net	Pin
TX4_DEN	AC3	TX5_DEN	AL6	TX6_DEN	AK15
TX4_DIN0	AB3	TX5_DIN0	AL5	TX6_DIN0	AK13
TX4_DIN1	AA1	TX5_DIN1	AK5	TX6_DIN1	AJ13
TX4_DIN10	V1	TX5_DIN10	AF1	TX6_DIN10	AJ10
TX4_DIN11	V2	TX5_DIN11	AF2	TX6_DIN11	AL9
TX4_DIN12	U1	TX5_DIN12	AF3	TX6_DIN12	AK9
TX4_DIN13	U2	TX5_DIN13	AE1	TX6_DIN13	AJ9
TX4_DIN14	U3	TX5_DIN14	AE2	TX6_DIN14	AL8
TX4_DIN15	T3	TX5_DIN15	AD1	TX6_DIN15	AJ8
TX4_DIN2	AA2	TX5_DIN2	AL4	TX6_DIN2	AL12
TX4_DIN3	AA3	TX5_DIN3	AK4	TX6_DIN3	AK12
TX4_DIN4	Y1	TX5_DIN4	AH1	TX6_DIN4	AJ12
TX4_DIN5	Y2	TX5_DIN5	AH2	TX6_DIN5	AL11
TX4_DIN6	Y3	TX5_DIN6	AH3	TX6_DIN6	AK11
TX4_DIN7	W1	TX5_DIN7	AG1	TX6_DIN7	AJ11
TX4_DIN8	W2	TX5_DIN8	AG2	TX6_DIN8	AL10
TX4_DIN9	W3	TX5_DIN9	AG3	TX6_DIN9	AK10
TX4_LINE_LE	AE4	TX5_LINE_LE	AF11	TX6_LINE_LE	AG15
TX4_LOCAL_LE	AF5	TX5_LOCAL_LE	AE12	TX6_LOCAL_LE	AG16
TX4_RCLK	AH15	TX5_RCLK	AJ15	TX6_RCLK	AL15
TX4_REN	AB2	TX5_REN	AJ6	TX6_REN	AL13
TX4_ROUT0	U5	TX5_ROUT0	Y7	TX6_ROUT0	AG4
TX4_ROUT1	U4	TX5_ROUT1	Y6	TX6_ROUT1	AH6
TX4_ROUT10	AB5	TX5_ROUT10	AE8	TX6_ROUT10	AG12
TX4_ROUT11	AC5	TX5_ROUT11	AF8	TX6_ROUT11	AH12
TX4_ROUT12	AC4	TX5_ROUT12	AE9	TX6_ROUT12	AG13
TX4_ROUT13	AD5	TX5_ROUT13	AF9	TX6_ROUT13	AH13
TX4_ROUT14	AD4	TX5_ROUT14	AE10	TX6_ROUT14	AG14
TX4_ROUT15	AE5	TX5_ROUT15	AE11	TX6_ROUT15	AH14
TX4_ROUT2	V5	TX5_ROUT2	AA7	TX6_ROUT2	AH7
TX4_ROUT3	V4	TX5_ROUT3	AA6	TX6_ROUT3	AG8
TX4_ROUT4	W5	TX5_ROUT4	AB7	TX6_ROUT4	AH8
TX4_ROUT5	W4	TX5_ROUT5	AC7	TX6_ROUT5	AG9
TX4_ROUT6	Y5	TX5_ROUT6	AC6	TX6_ROUT6	AH9
TX4_ROUT7	Y4	TX5_ROUT7	AD7	TX6_ROUT7	AG10
TX4_ROUT8	AA5	TX5_ROUT8	AD6	TX6_ROUT8	AG11
TX4_ROUT9	AA4	TX5_ROUT9	AE6	TX6_ROUT9	AH11
TX4_TCLK	AB1	TX5_TCLK	AK6	TX6_TCLK	AK14

Table 9 - DSP interface

Net	Pin	Net	Pin	Net	Pin
ADD0	AF30	DAT15	AB31	DAT50	AG22
ADD1	AF31	DAT16	AC29	DAT51	AG21
ADD10	AJ27	DAT17	AC30	DAT52	AH21
ADD11	AK27	DAT18	AC31	DAT53	AG20
ADD12	AL26	DAT19	AD29	DAT54	AH20
ADD13	AJ26	DAT2	U31	DAT55	AG19
ADD14	AK26	DAT20	AD31	DAT56	AH19
ADD15	AL25	DAT21	AE30	DAT57	AG18
ADD16	AK25	DAT22	AE31	DAT58	AH18
ADD17	AJ24	DAT23	AF29	DAT59	AG17
ADD18	AL24	DAT24	U27	DAT6	W31
ADD19	AJ23	DAT25	U28	DAT60	T26
ADD2	AG29	DAT26	V27	DAT61	U25
ADD20	AK23	DAT27	V28	DAT62	U26
ADD21	AL23	DAT28	W27	DAT63	V25
ADD22	AJ22	DAT29	W28	DAT7	Y29
ADD23	AK22	DAT3	V30	DAT8	Y30
ADD24	AL22	DAT30	Y27	DAT9	Y31
ADD25	AJ21	DAT31	Y28	DSPCON0	V26
ADD26	AK21	DAT32	AA27	DSPCON1	W26
ADD27	AL21	DAT33	AA28	DSPCON10	AE23
ADD28	AJ20	DAT34	AB27	DSPCON11	AE22
ADD29	AK20	DAT35	AC27	DSPCON12	AE21
ADD3	AG30	DAT36	AC28	DSPCON13	AL17
ADD30	AL20	DAT37	AD27	DSPCON14	AK17
ADD31	AJ19	DAT38	AD28	DSPCON15	AJ17
ADD4	AG31	DAT39	AE27	DSPCON16	AL18
ADD5	AH29	DAT4	W29	DSPCON17	AK18
ADD6	AH30	DAT40	AE28	DSPCON18	AL19
ADD7	AH31	DAT41	AF27	DSPCON19	AK19
ADD8	AL28	DAT42	AF28	DSPCON2	Y25
ADD9	AL27	DAT43	AG28	DSPCON3	Y26
DAT0	U29	DAT44	AH26	DSPCON4	AA25
DAT1	U30	DAT45	AG25	DSPCON5	AA26
DAT10	AA29	DAT46	AH25	DSPCON6	AB25
DAT11	AA30	DAT47	AH24	DSPCON7	AC25
DAT12	AA31	DAT48	AG23	DSPCON8	AC26
DAT13	AB29	DAT49	AH23	DSPCON9	AD25
DAT14	AB30	DAT5	W30	DSP_CLK	AJ16

Table 10 - Configuration pins

Net	Pin
BUSY	AD9
cclk	AJ4
CS_B	AK29
done	AG6
INIT_B	AD10
M0	AH27
M1	AJ28
M2	AG26
osc1	AD17
osc2	AD16
osc3	AH17
P_DATA0	AF7
P_DATA1	AG7
P_DATA2	AK3
P_DATA3	AJ5
P_DATA4	AF23
P_DATA5	AF24
P_DATA6	AG24
P_DATA7	AF25
prog	D27
WR_B	AK28

Table 11 - JTAG pins

Net	Pin
TCK	K11
TDI	C28
TDO	C4
TMS	J10

Table 12 - LEDs

Net	Pin
Led0	F27
Led1	M25
Led2	M26
Led3	N26
Led4	P26
Led5	R26

CPLD Pinouts

Table 13 - DSP interface

Name	Pin	Name	Pin
ADD0	4	ADD31	46
ADD1	7	ADD4	11
ADD10	17	ADD5	12
ADD11	19	ADD6	13
ADD12	20	ADD7	14
ADD13	21	ADD8	15
ADD14	22	ADD9	16
ADD15	23	DAT32	52
ADD16	24	DAT33	53
ADD17	25	DAT34	54
ADD18	26	DAT35	56
ADD19	27	DAT36	57
ADD2	9	DAT37	59
ADD20	28	DAT38	60
ADD21	31	DAT39	61
ADD22	33	DAT40	71
ADD23	34	DAT41	74
ADD24	35	DAT42	75
ADD25	39	DAT43	76
ADD26	40	DAT44	78
ADD27	41	DAT45	79
ADD28	43	DAT46	81
ADD29	44	DAT47	83
ADD3	10	We	30
ADD30	45	Clk	38

Table 14 - Programming interface

Name	Pin
done	125
FPGA_init	98
init	124
P_DATA0	118
P_DATA1	128
P_DATA2	130
P_DATA3	132
P_DATA4	134
P_DATA5	135
P_DATA6	137
P_DATA7	139
program	113
prom_clk	105
rst	143
start_prom	102
busy	120
write	110
cclk	106
cs	115

Table 15 - JTAG pins

Name	Pin
TCK	67
TDI	63
TDO	122
TMS	65

Table 16 - LEDs

Name	Pin
Led0	94
Led1	93
Led2	97
Led3	91

Appendix C

VHDL Code for the MacGregor Model Neuron

Gsim.vhd

```
-- Chip Lukes - 3/24/2003 - gsim.vhd
--
-- The purpose of this module is to run a simulation that utilizes
-- the Macgregor point neuron 10 solver (ptnrn.vhd) as well as a
-- simple ring network (ringnet.vhd). This demonstrates a very simple
-- and unrealistic simulation of a network of neurons (the number is
-- adjustable up to 1024). This was created as a quick way of
-- demonstrating that the equation solver ptnrn.vhd works with a
-- thesis deadline quickly approaching. However, the ringnet.vhd could
-- easily be replaced by a more realistic network given more time.
--
-- Notes on this file:
--      All neurons are assumed to be identical. If each neuron had different
--   parameters blockram would have to be added that loaded the K parameters
--   into the ptnrn.vhd equation solver. This is not a difficult task, but
--   was not done for simulation simplicity and thesis deadline reasons.


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity gsim is
  Port ( clk : in std_logic;
                rst : in std_logic;
                gkout : out std_logic_vector(15 downto 0);
                 thout : out std_logic_vector(15 downto 0);
                vmout : out std_logic_vector(15 downto 0);
                spikeout : out std_logic_vector(15 downto 0);
                testout : out std_logic_vector(9 downto 0));
end gsim;

architecture Behavioral of gsim is

        COMPONENT ptnrn
        PORT(
                data : IN std_logic_vector(15 downto 0);
                addr : IN std_logic_vector(9 downto 0);
                iin : IN std_logic_vector(15 downto 0);
                clk : IN std_logic;
                ena_gk : IN std_logic;
                ena_th : IN std_logic;
```

```
            ena_vm : IN std_logic;
            ena_s : IN std_logic;
            enb_gk : IN std_logic;
            enb_th : IN std_logic;
            enb_vm : IN std_logic;
            enb_s : IN std_logic;
            gkout : OUT std_logic_vector(15 downto 0);
            thout : OUT std_logic_vector(15 downto 0);
            vmout : OUT std_logic_vector(15 downto 0);
            spikeout : OUT std_logic_vector(15 downto 0)
            );
    END COMPONENT;

    COMPONENT ringnet
    PORT(
            addr : IN std_logic_vector(9 downto 0);
            en_s : IN std_logic;
            spike : IN std_logic_vector(15 downto 0);
            clk : IN std_logic;
            cout : OUT std_logic_vector(15 downto 0);
            testout : OUT std_logic_vector(9 downto 0)
            );
    END COMPONENT;

-- state machine stuff
type state is (     load_vm, load_gk, load_th, load_spike,
                                                load_network, sim_delay, sim_full  );
signal sim_state: state;
-- constants
-- for simulating different numbers of neurons these need to be changed
constant ENDVM: unsigned(13 downto 0) := "00000000100000";   -- 32
constant ENDGK: unsigned(13 downto 0) := "00000001000000";  -- 64
constant ENDTH: unsigned(13 downto 0) := "00000001100000";  -- 96
constant ENDSPIKE: unsigned(13 downto 0) := "00000010000000";  -- 128
constant ENDNET: unsigned(13 downto 0) := "00000010100000";  -- 160
constant ENDDELAY: unsigned(13 downto 0) := "00000010110111";  -- 160 + tree delay
constant CONTINUE: unsigned(13 downto 0) := "00000011000000";  -- 192
constant ZERO: std_logic_vector(15 downto 0) := "0000000000000000";

--signals and registered signal values
signal    data, data_reg : std_logic_vector(15 downto 0);
signal    addr, addr_reg : std_logic_vector(9 downto 0);
signal    ena_gk, ena_gk_reg : std_logic;
signal    ena_th, ena_th_reg : std_logic;
signal    ena_vm, ena_vm_reg : std_logic;
signal    ena_s, ena_s_reg : std_logic;
signal    enb_gk, enb_gk_reg : std_logic;
signal    enb_th, enb_th_reg : std_logic;
signal    enb_vm, enb_vm_reg : std_logic;
signal    enb_s, enb_s_reg : std_logic;
signal    en_s, en_s_reg : std_logic;
```

```
-- eventually this will come from network component
signal cout : std_logic_vector(15 downto 0);
signal spike_tmp : std_logic_vector(15 downto 0);
signal ignore_ctr : std_logic;

-- other
signal count, count_reg : unsigned (13 downto 0);

begin
	Inst_ptnrn: ptnrn PORT MAP(
		data => data_reg,
		addr => addr_reg,
		iin => cout,
		clk => clk,
		ena_gk => ena_gk_reg,
		ena_th => ena_th_reg,
		ena_vm => ena_vm_reg,
		ena_s => ena_s_reg,
		enb_gk => enb_gk_reg,
		enb_th => enb_th_reg,
		enb_vm => enb_vm_reg,
		enb_s => enb_s_reg,
		gkout => gkout,
		thout => thout,
		vmout => vmout,
		spikeout => spike_tmp
	);

	spikeout <= spike_tmp;

	Inst_ringnet: ringnet PORT MAP(
		addr => addr_reg,
		en_s => en_s_reg,
		spike => spike_tmp,
		clk => clk,
		cout => cout,
		testout => testout
	);
	state_machine: process(clk)
	begin
		if rising_edge(clk) then
			if rst = '1' then
				sim_state <= load_vm;
				ignore_ctr <= '0';
			else
				if ignore_ctr = '0' then
					case count is
						when ENDVM =>
							sim_state <= load_gk;
						when ENDGK =>
							sim_state <= load_th;
						when ENDTH =>
```

```
                                                    sim_state <= load_spike;
                                                when ENDSPIKE =>
                                                    sim_state <= load_network;
                                                when ENDNET =>
                                                    sim_state <= sim_delay;
                                                when ENDDELAY =>
                                                    sim_state <= sim_full;
                                                    ignore_ctr <= '1';
                                                when others =>
                                                    sim_state <= sim_state;
                                        end case;
                                end if;
        end if;
                end if;
        end process;

        -- counter for state machine
        cntr: process(clk)
        begin
          if rising_edge(clk) then
                if rst = '1' then
                        count <= "00000000000000";
                else
                        count <= count + 1;
                end if;
          end if;
        end process;

  addr <= "00000"&std_logic_vector(count_reg(4 downto 0));
--  cout <= "0001010000000000";

  ctrl_sel: process(sim_state)
  begin
        case sim_state is
                -- loading initial membrane voltage values into memory
                when load_vm =>
                        data <= ZERO;  -- initial membrane voltage is zero
                        ena_gk <= '0';
                        ena_th <= '0';
                        ena_vm <= '1';
                        ena_s <= '0';
                        enb_gk <= '0';
                        enb_th <= '0';
                        enb_vm <= '0';
                        enb_s <= '0';
                        en_s <= '0';
                -- loading initial potassium conductance values into memory
                when load_gk =>
                        data <= ZERO;  -- initial Gk = zero
                        ena_gk <= '1';
                        ena_th <= '0';
                        ena_vm <= '0';
```

```
            ena_s <= '0';
            enb_gk <= '0';
            enb_th <= '0';
            enb_vm <= '0';
            enb_s <= '0';
            en_s <= '0';
-- loading initial threshold values into memory
when load_th =>
            data <= "0000101000000000";  -- Threshold = 10
            ena_gk <= '0';
            ena_th <= '1';
            ena_vm <= '0';
            ena_s <= '0';
            enb_gk <= '0';
            enb_th <= '0';
            enb_vm <= '0';
            enb_s <= '0';
            en_s <= '0';
-- loading initial spike values into memory
when load_spike =>
            data <= ZERO;  -- no neurons spiking yet
            ena_gk <= '0';
            ena_th <= '0';
            ena_vm <= '0';
            ena_s <= '1';
            enb_gk <= '0';
            enb_th <= '0';
            enb_vm <= '0';
            enb_s <= '0';
            en_s <= '0';
-- if more complicated network existed it could be loaded here
when load_network =>
            data <= "0001010000000000";  -- dont care
            ena_gk <= '0';
            ena_th <= '0';
            ena_vm <= '0';
            ena_s <= '0';
            enb_gk <= '0';
            enb_th <= '0';
            enb_vm <= '0';
            enb_s <= '0';
            en_s <= '0';          -- whenever this is low it resets active neuron to 0
-- Data takes a while to fill up huge pipeline (no writeback)
when sim_delay =>  -- delay should be tree depth
            data <= ZERO;
            ena_gk <= '0';
            ena_th <= '0';
            ena_vm <= '0';
            ena_s <= '0';
            enb_gk <= '1';
            enb_th <= '1';
            enb_vm <= '1';
```

```
                        enb_s <= '1';
                        en_s <= '1';
                -- simulation running with state writeback
                when sim_full =>
                        data <= ZERO;
                        ena_gk <= '1';
                        ena_th <= '1';
                        ena_vm <= '1';
                        ena_s <= '1';
                        enb_gk <= '1';
                        enb_th <= '1';
                        enb_vm <= '1';
                        enb_s <= '1';
                        en_s <= '1';
                when others =>
                        null;
        end case;
 end process;

        -- state registers
        reg: process (clk)
        begin
                if rising_edge(clk) then
                        data_reg <= data;
                        addr_reg <= addr;
                        ena_gk_reg <= ena_gk;
                        ena_th_reg <= ena_th;
                        ena_vm_reg <= ena_vm;
                        ena_s_reg <= ena_s;
                        enb_gk_reg <= enb_gk;
                        enb_th_reg <= enb_th;
                        enb_vm_reg <= enb_vm;
                        enb_s_reg <= enb_s;
                        count_reg <= count;
                        en_s_reg <= en_s;
                end if;
        end process;
end Behavioral;
```

Ptnrn.vhd

```
-- Chip Lukes - 3/24/2003 - ptnrn.vhd
--
-- The purpose of this module is to solve for and update the
-- state variables for a Macgregor Point 10 Model Neuron every
-- clock cycle.  The state variables Potassium Conductance (Gk),
-- Threshold (Th), Membrane Voltage (Vm) and Spike are read from
-- memory, updated through equation trees, and then written back
-- into memory.  Each time the state variable update corresponds
```

```
-- to a simulation time step of approximately 1 ms.

-- Notes:
--      Initially, effort was made to manually instantiate registers
--   on the multiplier and block ram inputs and outputs, but this
--   caused some problems with simulation, and they were replaced
--   by inferred registers.  However this did not make a noticable
--   performance change so I just left the inferred registers.


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity ptnrn is
   generic (       K1: in std_logic_vector(15 downto 0) := "0000000011010001";
                   K2: in std_logic_vector(15 downto 0) := "0000001110101100";
                   K3: in std_logic_vector(15 downto 0) := "0000000000110011";
                   K4: in std_logic_vector(15 downto 0) := "1111011000000000";
                   K5: in std_logic_vector(15 downto 0) := "0000101000000000";
                   K6: in std_logic_vector(15 downto 0) := "0000000011110101";
                   K7: in std_logic_vector(15 downto 0) := "0000000000001011");
   Port ( data : in std_logic_vector(15 downto 0);
                   addr : in std_logic_vector(9 downto 0);
                   Iin : in std_logic_vector (15 downto 0);
                   clk : in std_logic;
                   ena_gk : in std_logic;
                   ena_th : in  std_logic;
                   ena_vm : in std_logic;
                   ena_s : in std_logic;
                   enb_gk : in std_logic;
                   enb_th : in  std_logic;
                   enb_vm : in std_logic;
                   enb_s : in std_logic;
                   gkout : out std_logic_vector(15 downto 0);
                   thout : out std_logic_vector(15 downto 0);
                   vmout : out std_logic_vector(15 downto 0);
                   spikeout : out std_logic_vector(15 downto 0));
end ptnrn;

architecture Behavioral of ptnrn is

        -- GK
        COMPONENT gktree
        PORT(
                gk_in : IN std_logic_vector(15 downto 0);
```

```
        k1 : IN std_logic_vector(15 downto 0);
        spike : IN std_logic_vector(15 downto 0);
        k2 : IN std_logic_vector(15 downto 0);
        clk : IN std_logic;
        gk_out : OUT std_logic_vector(15 downto 0)
        );
END COMPONENT;

-- Th
COMPONENT thtree
PORT(
        k5 : IN std_logic_vector(15 downto 0);
        k6 : IN std_logic_vector(15 downto 0);
        k7 : IN std_logic_vector(15 downto 0);
        vm : IN std_logic_vector(15 downto 0);
        th_in : IN std_logic_vector(15 downto 0);
        clk : IN std_logic;
        th_out : OUT std_logic_vector(15 downto 0)
        );
END COMPONENT;

--Vm
COMPONENT vmtree
PORT(
        iin : IN std_logic_vector(15 downto 0);
        k4 : IN std_logic_vector(15 downto 0);
        gk : IN std_logic_vector(15 downto 0);
        k3 : IN std_logic_vector(15 downto 0);
        vm_in : IN std_logic_vector(15 downto 0);
        clk : IN std_logic;
        vm_out : OUT std_logic_vector(15 downto 0)
        );
END COMPONENT;

-- Spike
COMPONENT spike
PORT(
        vm : IN std_logic_vector(15 downto 0);
        th : IN std_logic_vector(15 downto 0);
        clk : IN std_logic;
        spike : OUT std_logic_vector(15 downto 0)
        );
END COMPONENT;


-- BlockRAM controllers
component bramctrl is
        generic( TREE_DELAY : in natural := 23);  -- delay through tree
Port (  gdata : in std_logic_vector(15 downto 0);
                gaddr : in std_logic_vector(9 downto 0);
                ena : in std_logic;
                enb : in std_logic;
```

```
                                        clk : in std_logic;
                        sdata_in : in std_logic_vector(15 downto 0);
                        sdata_out : out std_logic_vector(15 downto 0)          );
    end component;

    -- Previous and next state values
    signal vm_old :  std_logic_vector(15 downto 0);
    signal th_old :  std_logic_vector(15 downto 0);
    signal gk_old : std_logic_vector(15 downto 0);
    signal spike_old :  std_logic_vector(15 downto 0);

    signal vm_new :  std_logic_vector(15 downto 0);
    signal th_new :  std_logic_vector(15 downto 0);
    signal gk_new : std_logic_vector(15 downto 0);
    signal spike_new :  std_logic_vector(15 downto 0);

    --attribute rloc: string;
    attribute hu_set: string;

    attribute hu_set of gkram: label is "gkram" ;
    attribute hu_set of thram: label is "thram" ;
    attribute hu_set of spikeram: label is "spikeram" ;
    attribute hu_set of vmram: label is "vmram" ;

begin

    --Gk
    gktree1: gktree PORT MAP(
            gk_in => gk_old,
            k1 => k1,
            spike => spike_old,
            k2 => k2,
            gk_out => gk_new,
            clk => clk
    );

    -- Th
    thtree1: thtree PORT MAP(
            k5 => k5,
            k6 => k6,
            k7 => k7,
            vm => vm_old,
            th_in => th_old,
            clk => clk,
            th_out => th_new
    );

    --Vm
    Inst_vmtree: vmtree PORT MAP(
            iin => Iin,
            k4 => k4,
            gk => gk_old,
```

```
            k3 => k3,
            vm_in => vm_old,
            clk => clk,
            vm_out => vm_new
);

--Spike
spiketree1: spike PORT MAP(
            vm => vm_old,
            th => th_old,
            clk => clk,
            spike => spike_new
);

-- RAM for Gk
gkram: bramctrl PORT MAP(
            gdata => data,
            gaddr => addr,
            ena => ena_gk,
            enb => enb_gk,
            clk => clk,
            sdata_in => gk_new,
            sdata_out => gk_old
);

            -- RAM for Th
thram: bramctrl PORT MAP(
            gdata => data,
            gaddr => addr,
            ena => ena_th,
            enb => enb_th,
            clk => clk,
            sdata_in => th_new,
            sdata_out => th_old
);

            -- RAM for Vm
vmram: bramctrl PORT MAP(
            gdata => data,
            gaddr => addr,
            ena => ena_vm,
            enb => enb_vm,
            clk => clk,
            sdata_in => vm_new,
            sdata_out => vm_old
);

            -- RAM for Spike
spikeram: bramctrl PORT MAP(
            gdata => data,
            gaddr => addr,
            ena => ena_s,
```

```
                enb => enb_s,
                clk => clk,
                sdata_in => spike_new,
                sdata_out => spike_old
        );

        spikeout <= spike_new;
        gkout <= gk_new;
        thout <= th_new;
        vmout <= vm_new;

end;
```

Network.vhd

```
-- chip lukes - 3/24/2003 - network.vhd
--
-- The purpose of this module is to simulate a simple ring
-- network topology involving macgregor point neurons
-- Current stimulus is applied to a neuron at a point until
-- it fires.  Once a neuron fires it then applies a current
-- stimulus to the next neuron at a point.  Using this crude
-- method the delay between neurons firing is merely the amount
-- of time it takes given neuron to fire when a step current
-- is injected.  This is typically about 8 simulation time
-- steps which represent approximately 8 ms.
-- The propagation delay is zero in this case.  In order to
-- model action potential propagation times as well as dendritic
-- delays a different model should be used.
-- Macgregor point 20, and point 30 models these effects
--
-- Important information when using this ringnet:
--  cout will be delayed by one thus, to remedy this
--          simply reduce Iin delay in vmtree by 1
--
--  minimal effort was taken to make this efficient within
--  the fpga

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity ringnet is
```

```
   Port ( addr : in std_logic_vector(9 downto 0);
          en_s : in std_logic;
          spike : in std_logic_vector(15 downto 0);
          clk : in std_logic;
          cout : out std_logic_vector(15 downto 0);
                                 testout : out std_logic_vector(9 downto 0));
end ringnet;

architecture Behavioral of ringnet is

          COMPONENT zdelay
          generic (
                    QWIDTH : natural := 16;
                    DELAY : natural := 16);  -- default delay of 16
          PORT(
                    qin : IN std_logic_vector(QWIDTH-1 downto 0);
                    clk : IN std_logic;
                    qout : OUT std_logic_vector(QWIDTH-1 downto 0)
                    );
          END COMPONENT;

          -- signals for registering the inputs
          signal addr_delay: std_logic_vector(9 downto 0);
          signal addr_reg: std_logic_vector(9 downto 0);
          signal spike_reg: std_logic_vector(15 downto 0);
          signal en_s_reg: std_logic;
          -- reg holds which neuron being stimulated
          signal spiking_neuron: std_logic_vector(9 downto 0);
          signal cout_reg : std_logic_vector(15 downto 0);  -- registered current   output
          signal iout : std_logic_vector(15 downto 0);  -- current output signal
          signal load_neuron, load_neuron_reg : std_logic; -- flag signaling a new neuron
                                                                  -- firing.
          -- constants
          constant ZERO : std_logic_vector(15 downto 0) := "0000000000000000";
          constant ONE : std_logic_vector(15 downto 0) := "0000000100000000";
          constant TWENTY : std_logic_vector(15 downto 0) := "0001010000000000";

begin

          -- this should delay the current address
          -- such that when a spike output happens
          -- the address should contain the address of
          -- the next neuron (to test this an output port was
          -- added)
          -- A delay of 23 should produce a spike from neuron(addr)
          -- want to then set neuron(addr+1) as the spiking neuron

          -- delayed address tells which neuron caused spike
          addr_shifter: zdelay
          GENERIC MAP (          QWIDTH => 10,
                                          DELAY => 23)
          PORT MAP(
```

```
                qout => addr_delay,
                qin => addr_reg,
                clk => clk
        );

        -- when a neuron spikes it inputs Iin to the next
        -- neuron until it spikes and the process continues
        iout <= TWENTY when addr_reg=spiking_neuron  else
                        ZERO;
        load_neuron <= '1' when addr_delay=spiking_neuron and spike_reg=ONE  else
        '0';

        --load new address when spike occurs
        --register outputs
        process (clk)
        begin
                if rising_edge(clk) then
                        addr_reg <= addr;
                        en_s_reg <= en_s;
                        spike_reg <= spike;
                        load_neuron_reg <= load_neuron;
                        if en_s_reg = '0' then
                                spiking_neuron <= "0000000000";
                                cout_reg <= ZERO;
                        else
                                if load_neuron_reg='1' then
                                        spiking_neuron <= addr_delay;
                                end if;
                                cout_reg <= iout;
                        end if;
                end if;
  end process;
        testout <= spiking_neuron;
        cout <= cout_reg;
end Behavioral;
```

Bramctrl.vhd

```
-- Chip Lukes
-- 3/12/2003
--
-- Local controller for a block ram that contains state
-- values for neurons.
-- This controller interfaces to a global simulation controller
-- that loads the initial memory contents at startup.
-- the instantiated bram has registered inputs and additionally
-- registers are infered on all bram inputs for a total delay of 2
-- to get data into the bram.  (see illustrator diagram)
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.all;

--library unisim;
--use unisim.vcomponents.all;

entity bramctrl is
        generic( TREE_DELAY : in natural := 4);  -- 4 = delay through ram
   Port ( gdata : in std_logic_vector(15 downto 0);
                gaddr : in std_logic_vector(9 downto 0);
                ena : in std_logic;
                enb : in std_logic;
                        clk : in std_logic;
                sdata_in : in std_logic_vector(15 downto 0);
                sdata_out : out std_logic_vector(15 downto 0)          );
end bramctrl;

architecture Behavioral of bramctrl is

        component bram
                port (    datain :  in std_logic_vector(15 downto 0);
                          adra :    in std_logic_vector(9 downto 0);
                          adrb :    in std_logic_vector(9 downto 0);
                          en_a  :   in std_logic;
                          en_b  :   in std_logic;
                          dataout : out std_logic_vector(15 downto 0);
                          clk   :   in std_logic        );
        end component;

        COMPONENT zdelay
        generic (
                QWIDTH : natural := 16;
                DELAY : natural := 16);  -- default delay of 16
        PORT(
                qin : IN std_logic_vector(QWIDTH-1 downto 0);
                clk : IN std_logic;
                qout : OUT std_logic_vector(QWIDTH-1 downto 0)
                );
        END COMPONENT;

signal muxdata: std_logic_vector(15 downto 0);
signal muxaddr: std_logic_vector(9 downto 0);
signal shiftaddr: std_logic_vector(9 downto 0);

-- registered input signals (inferred)
signal gdata_reg, sdata_reg: std_logic_vector(15 downto 0);
signal gaddr_reg: std_logic_vector(9 downto 0);
signal ena_reg,enb_reg: std_logic;

begin
```

```
        -- instantiate some dual port mem
        state : bram
                port map(       datain => muxdata,
                                adra => muxaddr,
                                adrb => gaddr_reg,
                                en_a => ena_reg,
                                en_b => enb_reg,
                                dataout => sdata_out,
                                clk => clk      );

        -- instantiate shift register to make
        -- sure writeback address corresponds
        -- to neuron state values present on
        -- outputs of eqn. trees.
        addshifter: zdelay
                generic map (   DELAY => TREE_DELAY,
                                QWIDTH => 10)
                PORT MAP(
                        qin => gaddr_reg,
                        qout => shiftaddr,
                        clk => clk
        );

-- infers registers all inputs
dreg: process (clk)
begin
        if rising_edge(clk) then
                gdata_reg <= gdata;
                gaddr_reg <= gaddr;
                ena_reg <= ena;
                enb_reg <= enb;
                sdata_reg <= sdata_in;
        end if;
end process;

-- multiplex data lines
muxdata <=      gdata_reg when ena_reg ='1' and enb_reg='0' else
                sdata_reg;

-- multiplex address lines
muxaddr <=      shiftaddr when enb_reg ='1' else
                gaddr_reg;

end Behavioral;
```

Gktree.vhd

```
-- Chip Lukes - 3/24/2003 - gktree.vhd
--
-- This module creates a tree structure that utilizes
-- adders, subtractors, and multipliers to solve the
-- Gk state for the Macgregor Point Neuron 10.  Shift registers
-- are used wherever necessary to realalign data throughout
-- the tree structure.  Also, the output of this state must be
-- delayed such that states match those that flow through the
-- longest equation tree (Vm).

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity gktree is
   Port ( gk_in : in std_logic_vector(15 downto 0);
          k1 : in std_logic_vector(15 downto 0);
          spike : in std_logic_vector(15 downto 0);
          k2 : in std_logic_vector(15 downto 0);
          gk_out : out std_logic_vector(15 downto 0);
          clk : in std_logic);
end gktree;

architecture Behavioral of gktree is

component MULT16X16S_PLUS
 port(    A_INPUT, B_INPUT:in std_logic_vector(15 downto 0);
          CLK, RST, CE: in std_logic;
          P_OUTPUT: out std_logic_vector(15 downto 0));
end component;

component addsub
          generic (
   WIDTH  : in  natural := 16 );
   Port (          A : in std_logic_vector(WIDTH-1 downto 0);
          B : in std_logic_vector(WIDTH-1 downto 0);
          ADD : in std_logic;   -- 1=add 0=sub
          Q : out std_logic_vector(WIDTH-1 downto 0);
          CO : out std_logic;
          CLK : in std_logic);
end component;

          COMPONENT zdelay
          generic (
                    QWIDTH : natural := 16;
                    DELAY : natural := 16);  -- default delay of 16
```

```
        PORT(
                qin : IN std_logic_vector(QWIDTH-1 downto 0);
                clk : IN std_logic;
                qout : OUT std_logic_vector(QWIDTH-1 downto 0)
                );
        END COMPONENT;

-- define all signals

signal M12out : std_logic_vector (15 downto 0);
signal M13out : std_logic_vector (15 downto 0);
signal A11out : std_logic_vector (15 downto 0);
signal D18out : std_logic_vector (15 downto 0);
--attribute rloc: string;
attribute hu_set: string;
attribute hu_set of M12 : label is "m12" ;
attribute hu_set of M13 : label is "m13" ;

begin

M12 : MULT16X16S_PLUS
                port map(       A_INPUT => gk_in,
                                B_INPUT => k1,
                                CLK => clk,
                                RST => '0',
                                CE => '1',
                                P_OUTPUT => M12out);

M13 : MULT16X16S_PLUS
                port map(       A_INPUT => spike,
                                B_INPUT => k2,
                                CLK => clk,
                                RST => '0',
                                CE => '1',
                                P_OUTPUT => M13out);

A11 : addsub
                port map(       A => M12out,
                        B => M13out,
                        ADD => '1',
                        Q => A11out,
                        CO => open,
                        CLK => clk);

D18 :  zdelay
        generic map (DELAY => 15)
        port map (      Qout => D18out,
                        Qin => A11out,
                        clk => clk);        -- 15 delay

gk_out <= D18out;
end Behavioral;
```

Thtree.vhd

```
-- Chip Lukes - 3/24/2003 - thtree.vhd
--
-- This module creates a tree structure that utilizes
-- adders, subtractors, and multipliers to solve the
-- Th state for the Macgregor Point Neuron 10.  Shift registers
-- are used wherever necessary to realalign data throughout
-- the tree structure.  Also, the output of this state must be
-- delayed such that states match those that flow through the
-- longest equation tree (Vm).
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity thtree is
   Port ( K5 : in std_logic_vector(15 downto 0);
                  K6 : in std_logic_vector(15 downto 0);
                  K7 : in std_logic_vector(15 downto 0);
                  Vm : in std_logic_vector(15 downto 0);
                  Th_in : in std_logic_vector(15 downto 0);
                  clk : in std_logic;
                  Th_out: out std_logic_vector(15 downto 0));
end thtree;

architecture Behavioral of thtree is

component MULT16X16S_PLUS
 port(    A_INPUT, B_INPUT:in std_logic_vector(15 downto 0);
          CLK, RST, CE: in std_logic;
          P_OUTPUT: out std_logic_vector(15 downto 0));
end component;

component addsub
          generic (
   WIDTH  : in  natural := 16 );
   Port (          A : in std_logic_vector(WIDTH-1 downto 0);
          B : in std_logic_vector(WIDTH-1 downto 0);
          ADD : in std_logic;   -- 1=add 0=sub
          Q : out std_logic_vector(WIDTH-1 downto 0);
```

```
        CO : out std_logic;
        CLK : in std_logic);
end component;

        COMPONENT zdelay
        generic (
                QWIDTH : natural := 16;
                DELAY : natural := 16);  -- default delay of 16
        PORT(
                qin : IN std_logic_vector(QWIDTH-1 downto 0);
                clk : IN std_logic;
                qout : OUT std_logic_vector(QWIDTH-1 downto 0)
                );
        END COMPONENT;

-- define all signals
signal D12out   : std_logic_vector (15 downto 0);
signal D13out   : std_logic_vector (15 downto 0);
--signal D14out   : std_logic_vector (15 downto 0);
signal D15out   : std_logic_vector (15 downto 0);
--signal D16out   : std_logic_vector (15 downto 0);
signal D17out   : std_logic_vector (15 downto 0);
signal M10out     : std_logic_vector (15 downto 0);
signal M11out     : std_logic_vector (15 downto 0);
signal A8out      : std_logic_vector (15 downto 0);
signal A9out      : std_logic_vector (15 downto 0);
signal A10out     : std_logic_vector (15 downto 0);

--attribute rloc: string;
attribute hu_set: string;
attribute hu_set of M10: label is "m10" ;
attribute hu_set of M11: label is "m11" ;

begin

M10 : MULT16X16S_PLUS
                port map(       A_INPUT => K7,
                                B_INPUT => Vm,
                                CLK => clk,
                                RST => '0',
                                CE => '1',
                                P_OUTPUT => M10out);

M11 : MULT16X16S_PLUS
                port map(       A_INPUT => A8out,
                                B_INPUT => D12out,
                                CLK => clk,
                                RST => '0',
                                CE => '1',
                                P_OUTPUT => M11out);

A8 : addsub
```

```
                port map(       A => Th_in,
                        B => K5,
                        ADD => '0',         --subtract
                        Q => A8out,
                        CO => open,
                        CLK => clk);

A9 : addsub
                port map(       A => M11out,
                        B => D15out,
                        ADD => '1',
                        Q => A9out,
                        CO => open,
                        CLK => clk);

A10 : addsub
                port map(       A => A9out,
                        B => D13out,
                        ADD => '1',
                        Q => A10out,
                        CO => open,
                        CLK => clk);

D12 :  zdelay
        generic map (DELAY => 1)
        port map (      Qout => D12out,
                        Qin => K6,
                        clk => clk);        -- delay 1

D13 :  zdelay
        generic map (DELAY => 2)
        port map (      Qout => D13out,
                        Qin => M10out,
                        clk => clk);  -- delay 2

--D14 :  zdelay
--      port map (      Qout => D14out,
--              Qin => M11out,
--              clk => clk,
--              delay => "0010");

D15 :  zdelay
        generic map (DELAY => 2)
        port map (      Qout => D15out,
                        Qin => K5,
                        clk => clk);        -- delay 4

--D16 :  zdelay
--      port map (      Qout => D16out,
--              Qin => A9out,
--              clk => clk,
--              delay => "0010");
```

```
D17 :  zdelay
        generic map (DELAY => 13)
        port map (      Qout => D17out,
                        Qin => A10out,
                        clk => clk);  -- delay 13

Th_out <= D17out;

end Behavioral;
```

Spiketree.vhd

```
-- Chip Lukes     Spike Equation Tree <spiketree.vhd>
-- 3/14/2003
--
-- spike calculation consists of determining whether or not
-- Vm >= Th and if so the spike output is one
-- else spike output zero
-- Instead of just using std_logic_signed I decided to manually
-- perform the compare function as described below:
--              determination based on Vm(15) Th(15) and the msb of (Vm-Th)
--
--      Vm(15)   Th(15)   Result(15) = (Vm-Th)
--      ------   ------   ----------
--       0        0        0                Vm > Th
--       0        0        1                Vm < Th
--       0        1        0                     Vm > Th
--       0        1        1                     Vm > Th
--       1        0        0                Vm < Th
--       1        0        1                Vm < Th
--       1        1        0                     Vm > Th
--       1        1        1                     Vm < Th
--
--    a delay of 19 clock cycles is introducted to match delay through
--    other eqn trees.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity spike is
  Port ( Vm : in std_logic_vector(15 downto 0);
```

```
                Th : in std_logic_vector(15 downto 0);
                clk : in std_logic;
                spike : out std_logic_vector(15 downto 0));
end spike;

architecture Behavioral of spike is

        COMPONENT zdelay
        generic (
                QWIDTH : natural := 16;
                DELAY : natural := 16);  -- default delay of 16
        PORT(
                qin : IN std_logic_vector(QWIDTH-1 downto 0);
                clk : IN std_logic;
                qout : OUT std_logic_vector(QWIDTH-1 downto 0)
                );
        END COMPONENT;


        component addsub
        generic (
                WIDTH  : in  natural := 16 );
        Port (     A : in std_logic_vector(WIDTH-1 downto 0);
        B : in std_logic_vector(WIDTH-1 downto 0);
        ADD : in std_logic;   -- 1=add 0=sub
        Q : out std_logic_vector(WIDTH-1 downto 0);
        CO : out std_logic;
        CLK : in std_logic);
        End component;

        constant ONE : std_logic_vector(15 downto 0) := "0000000100000000";
        constant ZERO : std_logic_vector(15 downto 0) := "0000000000000000";

        signal Vm_reg, Th_reg, mux_out, sub_out : std_logic_vector(15 downto 0);
        signal signs_reg : std_logic_vector(1 downto 0);
        signal bool_sel : std_logic_vector(2 downto 0);
        signal ge : std_logic;
begin
        regin: process (clk)
        begin
                if rising_edge(clk) then
                        Vm_reg <= Vm;
                        Th_reg <= Th;
                        signs_reg <= Vm_reg(15)&Th_reg(15);
                end if;
        end process;

        A2 : addsub
                port map(           A => Vm_reg,
                B => Th_reg,
                ADD => '0',  -- subtract
                Q => sub_out,
```

```
                CO => open,
                CLK => clk);

        bool_sel <= signs_reg&sub_out(15);

        muxctrl: process (clk)
        begin
                if rising_edge(clk) then
                        case bool_sel is
                        when "000"  =>
                                mux_out <= ONE;
                        when "001"  =>
                                mux_out <= ZERO;
                        when "010"  =>
                                mux_out <= ONE;
                        when "011"  =>
                                mux_out <= ONE;
                when "100"  =>
                                mux_out <= ZERO;
                        when "101"  =>
                                mux_out <= ZERO;
                        when "110"  =>
                                mux_out <= ZERO;
                        when "111"  =>
                                mux_out <= ONE;
                        when others  =>
                                mux_out <= ZERO;
                end case;
        end if;
    end process;

    outdly: zdelay
            generic map (DELAY => 16)
    port map (      Qout => Spike,
                    Qin => mux_out,
                    clk => clk);  -- delay 16

end Behavioral;
```

Vmtree.vhd

```
-- Chip Lukes - 3/24/2003 - vmtree.vhd
--
-- This module creates a tree structure that utilizes
-- adders, subtractors, and multipliers to solve the
-- Gk state for the Macgregor Point Neuron 10.  Shift registers
-- are used wherever necessary to realalign data throughout
-- the tree structure.  Also, the output of this state is the
-- longest path through any of the trees, thus any changes
```

```
-- to this equation tree will require the other trees to be updated
-- accordingly.
--
-- Notes:
--          The Vm calculation involved a divide and e^x calculation
--    which is difficult to achieve in hardware.  For the e^x
--    a Taylor series was used which will work as long as the Gk
--    value does not get > 9.  For the divide a look up table was
--    used.  This is valid for Gk values less than 6.
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity vmtree is
   Port ( Iin : in std_logic_vector(15 downto 0);
                  K4 : in std_logic_vector(15 downto 0);
                  Gk : in std_logic_vector(15 downto 0);
                  K3 : in std_logic_vector(15 downto 0);
                  Vm_in : in std_logic_vector(15 downto 0);
                  clk : in std_logic;
                  Vm_out: out std_logic_vector(15 downto 0));
end vmtree;


architecture Behavioral of vmtree is

-- fixed pt constants                -- 0.33203 (00000000.01010101)
constant ONE_THIRD : std_logic_vector(15 downto 0) := "0000000001010101";
constant ONE_HALF : std_logic_vector(15 downto 0) := "0000000010000000";
constant ONE : std_logic_vector(15 downto 0) := "0000000100000000";

component MULT16X16S_PLUS
 port(    A_INPUT, B_INPUT:in std_logic_vector(15 downto 0);
          CLK, RST, CE: in std_logic;
          P_OUTPUT: out std_logic_vector(15 downto 0));
end component;

component addsub
          generic (
   WIDTH  : in  natural := 16 );
   Port (          A : in std_logic_vector(WIDTH-1 downto 0);
          B : in std_logic_vector(WIDTH-1 downto 0);
          ADD : in std_logic;   -- 1=add 0=sub
          Q : out std_logic_vector(WIDTH-1 downto 0);
          CO : out std_logic;
```

```
        CLK : in std_logic);
end component;

        COMPONENT zdelay
        generic (
                QWIDTH : natural := 16;
                DELAY : natural := 16);  -- default delay of 16
        PORT(
                qin : IN std_logic_vector(QWIDTH-1 downto 0);
                clk : IN std_logic;
                qout : OUT std_logic_vector(QWIDTH-1 downto 0)
                );
        END COMPONENT;

component div
  Port (
                divin : in std_logic_vector(15 downto 0);
                clk : in std_logic;
                divout : out std_logic_vector(15 downto 0));
end component;

-- define all signals

signal D1out   : std_logic_vector (15 downto 0);
signal D2out   : std_logic_vector (15 downto 0);
signal D3out   : std_logic_vector (15 downto 0);
signal D4out   : std_logic_vector (15 downto 0);
signal D5out   : std_logic_vector (15 downto 0);
signal D6out   : std_logic_vector (15 downto 0);
signal D7out   : std_logic_vector (15 downto 0);
signal D8out   : std_logic_vector (15 downto 0);
signal D9out   : std_logic_vector (15 downto 0);
signal D10out  : std_logic_vector (15 downto 0);
--signal D11out  : std_logic_vector (15 downto 0);

signal M1out      : std_logic_vector (15 downto 0);
signal M2out      : std_logic_vector (15 downto 0);
signal M3out      : std_logic_vector (15 downto 0);
signal M4out      : std_logic_vector (15 downto 0);
signal M5out      : std_logic_vector (15 downto 0);
signal M6out      : std_logic_vector (15 downto 0);
signal M7out      : std_logic_vector (15 downto 0);
signal M8out      : std_logic_vector (15 downto 0);
signal M9out      : std_logic_vector (15 downto 0);
signal divout  : std_logic_vector (15 downto 0);
signal A1out      : std_logic_vector (15 downto 0);
signal A2out      : std_logic_vector (15 downto 0);
signal A3out      : std_logic_vector (15 downto 0);
signal A4out      : std_logic_vector (15 downto 0);
signal A5out      : std_logic_vector (15 downto 0);
signal A6out      : std_logic_vector (15 downto 0);
signal A7out      : std_logic_vector (15 downto 0);
```

```
--attribute rloc: string;
attribute hu_set: string;
attribute hu_set of M1: label is "m1" ;
attribute hu_set of M2: label is "m2" ;
attribute hu_set of M3: label is "m3" ;
attribute hu_set of M4: label is "m4" ;
attribute hu_set of M5: label is "m5" ;
attribute hu_set of M6: label is "m6" ;
attribute hu_set of M7: label is "m7" ;
attribute hu_set of M8: label is "m8" ;
attribute hu_set of M9: label is "m9" ;

begin

M1 : MULT16X16S_PLUS
                port map(       A_INPUT => K4,
                                B_INPUT => Gk,
                                CLK => clk,
                                RST => '0',
                                CE => '1',
                                P_OUTPUT => M1out);

M2 : MULT16X16S_PLUS
                port map(       A_INPUT => A1out,
                                B_INPUT => D2out,
                                CLK => clk,
                                RST => '0',
                                CE => '1',
                                P_OUTPUT => M2out);

M3 : MULT16X16S_PLUS
                port map(       A_INPUT => M2out,
                                B_INPUT => M2out,
                                CLK => clk,
                                RST => '0',
                                CE => '1',
                                P_OUTPUT => M3out);

M4 : MULT16X16S_PLUS
                port map(       A_INPUT => M2out,
                                B_INPUT => ONE_THIRD,
                                CLK => clk,
                                RST => '0',
                                CE => '1',
                                P_OUTPUT => M4out);

M5 : MULT16X16S_PLUS
                port map(       A_INPUT => D3out,
                                B_INPUT => D4out,
                                CLK => clk,
                                RST => '0',
```

```
                    CE => '1',
                    P_OUTPUT => M5out);

M6 : MULT16X16S_PLUS
        port map(   A_INPUT => M3out,
                    B_INPUT => ONE_HALF,
                    CLK => clk,
                    RST => '0',
                    CE => '1',
                    P_OUTPUT => M6out);

M7 : MULT16X16S_PLUS
        port map(   A_INPUT => M6out,
                    B_INPUT => D6out,
                    CLK => clk,
                    RST => '0',
                    CE => '1',
                    P_OUTPUT => M7out);

M8 : MULT16X16S_PLUS
        port map(   A_INPUT => A5out,
                    B_INPUT => D9out,
                    CLK => clk,
                    RST => '0',
                    CE => '1',
                    P_OUTPUT => M8out);

M9 : MULT16X16S_PLUS
        port map(   A_INPUT => D7out,
                    B_INPUT => A6out,
                    CLK => clk,
                    RST => '0',
                    CE => '1',
                    P_OUTPUT => M9out);

A1 : addsub
        port map(   A => Gk,
                B => ONE,
                ADD => '1',
                Q => A1out,
                CO => open,
                CLK => clk);

A2 : addsub
        port map(   A => D1out,
                B => M1out,
                ADD => '1',
                Q => A2out,
                CO => open,
                CLK => clk);

A3 : addsub
```

```
                port map(       A => ONE,
                        B => M2out,
                        ADD => '0',  --subtract
                        Q => A3out,
                        CO => open,
                        CLK => clk);

A4 : addsub
                port map(       A => D5out,
                        B => M6out,
                        ADD => '1',
                        Q => A4out,
                        CO => open,
                        CLK => clk);

A5 : addsub
                port map(       A => D8out,
                        B => M7out,
                        ADD => '0',         -- subtract
                        Q => A5out,
                        CO => open,
                        CLK => clk);

A6 : addsub
                port map(       A => ONE,
                        B => A5out,
                        ADD => '0',         --subtract
                        Q => A6out,
                        CO => open,
                        CLK => clk);

A7 : addsub
                port map(       A => M9out,
                        B => D10out,
                        ADD => '1',
                        Q => A7out,
                        CO => open,
                        CLK => clk);

D1 :  zdelay
        generic map (   DELAY => 5) -- this was changed to facilitate the network
        port map (      Qout => D1out,
                        Qin => Iin,
                        clk => clk);        --delay 3

D2 :  zdelay
        generic map (   DELAY => 1)
        port map (      Qout => D2out,
                        Qin => K3,
                        clk => clk);        --delay 1

D3 :  zdelay
```

```
        generic map (DELAY => 1)
        port map (      Qout => D3out,
                        Qin => A2out,
                        clk => clk);  -- delay 1

D4 :  zdelay
        generic map (   DELAY => 4)
        port map (      Qout => D4out,
                        Qin => Divout,
                        clk => clk);  --delay 3  now 4 with Gk into
                                -- div instead of 1+gk

D5 :  zdelay
        generic map (   DELAY => 5)
        port map (      Qout => D5out,
                        Qin => A3out,
                        clk => clk);  --delay 5

D6 :  zdelay
        generic map (   DELAY => 3)
        port map (      Qout => D6out,
                        Qin => M4out,
                        clk => clk);  --delay 3

D7 :  zdelay
        generic map (   DELAY => 7)
        port map (      Qout => D7out,
                        Qin => M5out,
                        clk => clk);  --delay 7

D8 :  zdelay
        generic map (   DELAY => 2)
        port map (      Qout => D8out,
                        Qin => A4out,
                        clk => clk);  --delay 2

D9 :  zdelay
        generic map (   DELAY => 14)
        port map (      Qout => D9out,
                        Qin => Vm_in,
                        clk => clk);  -- delay 14

D10 :  zdelay
        generic map (   DELAY => 1)
        port map (      Qout => D10out,
                        Qin => M8out,
                        clk => clk);  --delay 1

--D11 :  zdelay
--      port map (      Qout => D11out,
--              Qin => M9out,
--              clk => clk,
```

```
--                delay => "0010");

DV: div
                  port map (      divin => Gk,      -- changed this to be a function of
                                  clk => clk,                          -- Gk instead of 1+Gk
                                  divout => divout);

Vm_out <= A7out;
end Behavioral;
```

Zdelay.vhd

```
-- Chip Lukes - 3/24/2003 - zdelay.vhd
--
-- The purpose of this module is to inferr a shift register
-- for busses of arbitrary size and delay depth.  ISE 5.1 does
-- a good job of picking out the shift registers.  This uses a
-- generate statement and instantiates the appropriate number
-- of n-bit shifters.  Thus, this file needs to accompany
-- shiftreg.vhd


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
library UNISIM;
use UNISIM.VComponents.all;

entity zdelay is
        generic (
                QWIDTH : natural := 16;  -- width of data to be delayed
                DELAY : natural := 16);  -- default delay of 16
   Port ( Qout : out std_logic_vector(QWIDTH-1 downto 0);
       Qin : in std_logic_vector(QWIDTH-1 downto 0);
       clk : in std_logic);
end zdelay;

architecture Behavioral of zdelay is

        COMPONENT shiftreg
        generic (
                DELAY : natural := 16);  -- default delay of 16
        PORT(
```

```
                q_in : IN std_logic;
                clk : IN std_logic;
                q_out : OUT std_logic
                );
        END COMPONENT;

begin

zgen: for i in 0 to QWIDTH-1 generate

        shifter: shiftreg
        generic map (DELAY => DELAY)
        PORT MAP(
                q_in => Qin(i),
                q_out => Qout(i),
                clk => clk
        );

end generate;

end Behavioral;
```

Shiftreg.vhd

```
-- Chip Lukes - 3/24/2003 - shiftreg.vhd
--
-- The purpose of this module is to inferr a n-bit shift register
-- of arbitrary depth.  ISE 5.1 does a good job of picking out
-- the shift registers.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity shiftreg is
        generic (
                DELAY : natural := 16);  -- default delay of 16
        Port (  q_in : in std_logic; -- bit in
                q_out : out std_logic; -- bit out delayed by DELAY
                clk : in std_logic);
end shiftreg;
```

```
architecture Behavioral of shiftreg is

        signal Q_INT: std_logic_vector(31 downto 0);
        constant DELAY_LEN : natural := (natural(DELAY/17) + 1)*16;

begin

process(clk)
begin
        if rising_edge(clk) then
                Q_INT(DELAY_LEN-1 downto 0) <=
                        (Q_INT(DELAY_LEN-2 downto 0) & q_in);
        end if;
end process;

q_out <= Q_INT(DELAY-1);

end Behavioral;
```

Bram.vhd

```
-- Chip Lukes - 3/23/2003 - bram.vhd
--
-- The purpose of this module is to create a wrapper
-- around an instantiated 18bit x 1K dual port block ram
-- port A is a dedicated write port and port B is a
-- dedicated read port.
--
--
-- Notes:
--      Initially, effort was made to manually instantiate registers
--   on the block ram inputs and outputs, but this
--   caused some problems with simulation, and they were replaced
--   by inferred registers.  However this did not make a noticable
--   performance change so I just left the inferred registers.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library UNISIM;
use UNISIM.VComponents.all;

entity bram is
        port (  datain :  in std_logic_vector(15 downto 0);  -- data to be written
                adra :    in std_logic_vector(9 downto 0); -- address for write
                adrb :    in std_logic_vector(9 downto 0); -- address for read
                en_a  :   in std_logic;  -- enable port A
```

```
                en_b   :  in std_logic; -- enable port B
                dataout : out std_logic_vector(15 downto 0);  -- read data
                clk   :    in std_logic         );
end bram;

architecture behavioral of bram is

-- Component Declaration for these design elements
-- should be placed after architecture statement but before begin keyword
-- For the following component declaration, enter RAMB16_S9_{S9 | S18 | S36},
-- RAMB16_S18_{S18 | S36}, or RAMB16_S36_S36

component RAMB16_S18_S18
-- synthesis translate_off
generic (
INIT_00 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_01 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_02 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_03 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_04 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_05 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_06 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_07 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_08 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_09 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_0A : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_0B : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_0C : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_0D : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_0E : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_0F : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_10 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_11 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_12 : bit_vector :=
```

```
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_13 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_14 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_15 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_16 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_17 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_18 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_19 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_1A : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_1B : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_1C : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_1D : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_1E : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_1F : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_20 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_21 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_22 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_23 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_24 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_25 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_26 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_27 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_28 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_29 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_2A : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_2B : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_2C : bit_vector :=
```

```
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_2D : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_2E : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_2F : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_30 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_31 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_32 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_33 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_34 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_35 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_36 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_37 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_38 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_39 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_3A : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_3B : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_3C : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_3D : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_3E : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_3F : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INIT_A : bit_vector := "0000000000000000";
INIT_B : bit_vector := "0000000000000000";
INITP_00 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INITP_01 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INITP_02 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INITP_03 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INITP_04 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INITP_05 : bit_vector :=
```

```
X"0000000000000000000000000000000000000000000000000000000000000000";
INITP_06 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INITP_07 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
SRVAL_A : bit_vector := "00000000000000000";
SRVAL_B : bit_vector := "00000000000000000";
WRITE_MODE_A : string := "WRITE_FIRST";
WRITE_MODE_B : string := "WRITE_FIRST"
);


-- synthesis translate_on
port (    DOA : out STD_LOGIC_VECTOR (15 downto 0);
          DOB : out STD_LOGIC_VECTOR (15 downto 0);
          DOPA : out STD_LOGIC_VECTOR (1 downto 0);
          DOPB : out STD_LOGIC_VECTOR (1 downto 0);
          ADDRA : in STD_LOGIC_VECTOR (9 downto 0);
          ADDRB : in STD_LOGIC_VECTOR (9 downto 0);
          CLKA : in STD_ULOGIC;
          CLKB : in STD_ULOGIC;
          DIA : in STD_LOGIC_VECTOR (15 downto 0);
          DIB : in STD_LOGIC_VECTOR (15 downto 0);
          DIPA : in STD_LOGIC_VECTOR (1 downto 0);
          DIPB : in STD_LOGIC_VECTOR (1 downto 0);
          ENA : in STD_ULOGIC;
          ENB : in STD_ULOGIC;
          SSRA : in STD_ULOGIC;
          SSRB : in STD_ULOGIC;
          WEA : in STD_ULOGIC;
          WEB : in STD_ULOGIC  );
end component;


-- Port Registers  not 100% sure if rlocs are best choice
-- need to look around the brams to see where regs are

signal DOB, DIA : std_logic_vector(15 downto 0);
signal ADDRA, ADDRB : std_logic_vector(9 downto 0);
signal WEA, ENA, ENB: std_logic;  -- convert between ulogic and logic


attribute maxdelay: string;

-- keep delays so that place and route keeps the registers
-- close to the ram inputs
attribute maxdelay of DOB: signal is "500ps";
attribute maxdelay of DIA: signal is "500ps";
attribute maxdelay of ADDRA: signal is "500ps";
attribute maxdelay of ADDRB: signal is "500ps";
attribute maxdelay of WEA: signal is "500ps";
attribute maxdelay of ENA: signal is "500ps";
```

```
attribute maxdelay of ENB: signal is "500ps";

begin

-- Component Attribute Specification for design element
-- should be placed after architecture declaration
-- but before the begin keyword
-- Put attributes, if necessary
-- Component Instantiation for design element
-- Should be placed in architecture after the begin keyword

wrapped_ram : RAMB16_S18_S18

        -- synthesis translate_off
        generic map (
                WRITE_MODE_A => "WRITE_FIRST",
                WRITE_MODE_B => "WRITE_FIRST")

        -- synopsys translate_on
        -- I am not sure if it is ok to leave some of these
        -- ports open.
        port map (      DOA => open,
                        DOB => DOB,
                        DOPA => open,
                        DOPB => open,
                        ADDRA => ADDRA,
                        ADDRB => ADDRB,
                        CLKA => clk,
                        CLKB => clk,
                        DIA => DIA,
                        DIB => "0000000000000000",  -- don't care about this now
                        DIPA => "00",  -- don't care about this now
                        DIPB => "00",  -- don't care about this now
                        ENA => ENA,
                        ENB => ENB,
                        SSRA => '0',
                        SSRB => '0',
                        WEA => '1',
                        WEB => '0');
-- infer registers
process (clk)
begin
        if rising_edge(clk) then
                DIA <= datain;
                ADDRA <= adra;
                ADDRB <= adrb;
                ENA <= en_a;
                ENB <= en_b;
                dataout <= DOB;
        end if;
end process;
```

```
end Behavioral;
```

Mult16x16s_plus.vhd

```
-- Chip Lukes - 3/24/2003 - MULT16x16S_PLUS.VHD
--
-- For the most part this is a wrapper for the embedded
-- multipliers present in Virtex II FPGAs.  This is
-- similar to the wrapper given by Xilinx, but I was
-- having problems with the floorplanned registers
-- so I just inferred registers instead which worked just
-- fine as far as I can tell.
--
-- Notes:
--        Multiplier component for equation trees
--        I did some serious commenting out here
--        assumed binary point position = 8 on mult inputs

library ieee;
use ieee.std_logic_1164.all;

library UNISIM;
use UNISIM.VComponents.all;

entity MULT16X16S_PLUS is
 port(    A_INPUT, B_INPUT      : in std_logic_vector(15 downto 0);
          CLK, RST, CE          : in std_logic;
          P_OUTPUT                      : out std_logic_vector(15 downto 0));
end MULT16X16S_PLUS;

architecture MULT16X16S_PLUS_BEHAVIOR of MULT16X16S_PLUS is

component MULT18X18S
   port ( A       : in STD_LOGIC_VECTOR (17 downto 0);
                  B       : in STD_LOGIC_VECTOR (17 downto 0);
                  C       : in STD_ULOGIC ;
                  CE      : in STD_ULOGIC ;
                  P       : out STD_LOGIC_VECTOR (35 downto 0);
                  R       : in STD_ULOGIC );
end component;

signal areg, breg: std_logic_vector(17 downto 0);
signal preg: std_logic_vector(35 downto 0);

attribute maxdelay: string;

attribute maxdelay of areg: signal is "500ps";
attribute maxdelay of breg: signal is "500ps";
```

```
attribute maxdelay of preg: signal is "500ps";

begin

-- register inputs and outputs of multiplier
process (clk)
begin
        if rising_edge(clk) then
                areg <= A_input(15)&A_input(15)&A_input;
                breg <= B_input(15)&B_input(15)&B_input;
                P_output <= preg(23 downto 8);
        end if;
end process;

test_mult16x16s : MULT18X18S
        port map(       P => preg,
                        A => areg,
                        B => breg,
                        C => CLK,
                        CE => CE,
                        R => RST);

end MULT16X16S_PLUS_BEHAVIOR;
```

Addsub.vhd

```
-- Chip Lukes - 2/24/2003 - addsub.vhd
--
-- Straightforward adder/ subtractor with delay =1
-- Subtract operation produces A - B

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.all;
library unisim;
use unisim.vcomponents.all;

entity addsub is
 generic (
                WIDTH  : in  natural := 16 );

  Port (        A : in std_logic_vector(WIDTH-1 downto 0);
        B : in std_logic_vector(WIDTH-1 downto 0);
        ADD : in std_logic;   -- 1=add 0=sub
        Q : out std_logic_vector(WIDTH-1 downto 0);
        CO : out std_logic;
        CLK : in std_logic);
end addsub;
```

```
architecture behavior of addsub is

signal ci_bit     : natural range 0 to 1;
signal in1, in2, out1 : unsigned (WIDTH downto 0);

begin

in1 <= '0'& unsigned(A);
in2 <= '0' & unsigned(B) when add='1' else
                    '1' & unsigned(not B);
ci_bit <= 0 when add='1' else
              1;
Q <= std_logic_vector(out1(WIDTH-1 downto 0));
CO <= std_logic(out1(WIDTH));

process (clk)
begin
        if rising_edge(clk) then
                out1 <= in1 + in2 + ci_bit;
        end if;
end process;

end behavior;
```

Div.vhd

```
-- Chip Lukes - 2/24/2003 - div.vhd
--
-- The purpose of this module is to use a single port
-- block ram as a look up table to produce the results
-- to 1/(1+x).  The ram is initialized to the correct
-- memory contents and the file used to create the initial
-- memory vectors is a matlab file divlut.m
-- The ram is 8 bits wide since over this range the funtion
-- is always less than 1.

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
library UNISIM;
use UNISIM.VComponents.all;

entity div is
```

```
   Port ( divin : in std_logic_vector(15 downto 0);
          clk : in std_logic;
          divout : out std_logic_vector(15 downto 0));
end div;

architecture Behavioral of div is

-- using dual port ram because xilinx is retarded

-- Component Declaration for these design elements
-- should be placed after architecture statement but before begin keyword
-- For the following component declaration, enter RAMB16_S9_{S9 | S18 | S36},
-- RAMB16_S18_{S18 | S36}, or RAMB16_S36_S36
component RAMB16_S9

generic (
INIT : bit_vector := "000000000";
INIT_00 : bit_vector :=X"d0d1d3d4d5d7d8d9dbdcdddfe0e2e3e5e6e8e9ebeceef0f1f3f5f6f8fafcfeff";
INIT_01 : bit_vector :=X"afb0b0b1b2b3b4b5b6b7b8b9babbbcbdbebfc0c1c2c3c4c5c7c8c9cacbcccecf";
INIT_02 : bit_vector :=X"96979898999a9a9b9c9d9d9e9f9fa0a1a2a2a3a4a5a6a6a7a8a9aaaaabacadae";
INIT_03 : bit_vector :=X"84858586868787888889898a8a8b8c8c8d8d8e8e8f909091919293939495959 6";
INIT_04 : bit_vector :=X"767677777778787979797a7a7b7b7c7c7d7d7d7e7e7f7f808081818282838384";
INIT_05 : bit_vector :=X"6a6b6b6b6c6c6c6d6d6d6e6e6e6f6f6f707070717172727273737474757575";
INIT_06 : bit_vector :=X"6161616262626263636363646464656565666666666767676868686969696a6a";
INIT_07 : bit_vector :=X"595959595a5a5a5a5b5b5b5b5c5c5c5c5d5d5d5d5e5e5e5e5f5f5f5f60606060";
INIT_08 : bit_vector:=X"525252535353535354545454545555555555565656565757575757585858 59";
INIT_09 : bit_vector :=X"4c4c4d4d4d4d4d4d4d4e4e4e4e4e4e4f4f4f4f4f505050505050515151515252";
INIT_0A : bit_vector :=X"474747484848484848494949494949494a4a4a4a4a4a4a4b4b4b4b4b4b4c4c4c4c";
INIT_0B : bit_vector:=X"434343434343444444444444444545454545454545454646464646464647474747";
INIT_0C : bit_vector :=X"3f3f3f3f3f3f404040404040404041414141414141414242424242424243 43";
INIT_0D : bit_vector :=X"3b3c3c3c3c3c3c3c3c3c3c3d3d3d3d3d3d3d3d3d3e3e3e3e3e3e3e3e3e3f3f3f";
INIT_0E : bit_vector :=X"383838393939393939393939393a3a3a3a3a3a3a3a3a3a3a3a3b3b3b3b3b3b3b3b";
INIT_0F : bit_vector:=X"353636363636363636363636373737373737373737373737383838383838 3838";
INIT_10 : bit_vector:=X"333333333333333334343434343434343434343435353535353535353535 3535";
INIT_11 : bit_vector:=X"313131313131313131313131323232323232323232323232323333333333";
INIT_12 : bit_vector :=X"2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f303030303030303030303030303031";
INIT_13 : bit_vector :=X"2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2e2e2e2e2e2e2e2e2e2e2e2e2e2e2e2e2e2e";
INIT_14 : bit_vector :=X"2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2c2c2c2c2c2c2c2c2c2c2c2c2c2c2c2c2c2d";
INIT_15 : bit_vector :=X"29292929292929292a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2b2b2b2b2b";
INIT_16 : bit_vector:=X"2828282828282828282828282828282828282829292929292929292929292929";
INIT_17 : bit_vector:=X"2626262626262626272727272727272727272727272727272727272727272828";
INIT_18 : bit_vector:=X"2525252525252525252525252525252626262626262626262626262626262626";
INIT_19 : bit_vector:=X"2424242424242424242424242424242424242424242424252525252525252525";
INIT_1A :bit_vector:=X"2223232323232323232323232323232323232323232323232323232323242424";
INIT_1B : bit_vector:=X"2121212122222222222222222222222222222222222222222222222222222222";
INIT_1C : bit_vector:=X"2020202020212121212121212121212121212121212121212121212121212121";
INIT_1D : bit_vector :=X"1f1f1f1f2020202020202020202020202020202020202020202020202020202020";
INIT_1E : bit_vector :=X"1e1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f";
INIT_1F : bit_vector :=X"1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e";
INIT_20 : bit_vector :=X"1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1e1e1e1e1e";
INIT_21 : bit_vector :=X"1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1d1d1d1d1d1d1d1d1d1d1d1d1d";
INIT_22 : bit_vector :=X"1b1b1b1b1b1b1b1b1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c";
```

```
INIT_23 : bit_vector:=X"1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b";
INIT_24 : bit_vector :=X"1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a";
INIT_25 : bit_vector :=X"191919191919191a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a";
INIT_26 : bit_vector:=X"1919191919191919191919191919191919191919191919191919191919191919";
INIT_27 : bit_vector:=X"1818181818181818181818181818181818181919191919191919191919191919";
INIT_28 : bit_vector:=X"1818181818181818181818181818181818181818181818181818181818181818";
INIT_29 : bit_vector:=X"1717171717171717171717171717171717171717171717171818181818181818";
INIT_2A :bit_vector:=X"1717171717171717171717171717171717171717171717171717171717171717";
INIT_2B : bit_vector:=X"1616161616161616161616161616161616161616161616161717171717171717";
INIT_2C : bit_vector:=X"1616161616161616161616161616161616161616161616161616161616161616";
INIT_2D :bit_vector:=X"1515151515151515151515151515151515161616161616161616161616161616";
INIT_2E : bit_vector:=X"1515151515151515151515151515151515151515151515151515151515151515";
INIT_2F : bit_vector:=X"1414141414141415151515151515151515151515151515151515151515151515";
INIT_30 : bit_vector:=X"1414141414141414141414141414141414141414141414141414141414141414";
INIT_31 : bit_vector:=X"1414141414141414141414141414141414141414141414141414141414141414";
INIT_32 : bit_vector:=X"1313131313131313131313131313131313131313141414141414141414141414";
INIT_33 : bit_vector:=X"1313131313131313131313131313131313131313131313131313131313131313";
INIT_34 : bit_vector:=X"1313131313131313131313131313131313131313131313131313131313131313";
INIT_35 : bit_vector:=X"1212121212121212121212121212121212121212121212121213131313131313";
INIT_36 : bit_vector:=X"1212121212121212121212121212121212121212121212121212121212121212";
INIT_37 : bit_vector:=X"1212121212121212121212121212121212121212121212121212121212121212";
INIT_38 : bit_vector:=X"1111111111111111111111111111111111111212121212121212121212121212";
INIT_39 : bit_vector:=X"1111111111111111111111111111111111111111111111111111111111111111";
INIT_3A :bit_vector:=X"1111111111111111111111111111111111111111111111111111111111111111";
INIT_3B : bit_vector:=X"1111111111111111111111111111111111111111111111111111111111111111";
INIT_3C : bit_vector:=X"1010101010101010101010101010101010101010101010101010101010101010";
INIT_3D :bit_vector:=X"1010101010101010101010101010101010101010101010101010101010101010";
INIT_3E : bit_vector:=X"1010101010101010101010101010101010101010101010101010101010101010";
INIT_3F : bit_vector:=X"1010101010101010101010101010101010101010101010101010101010101010";
INITP_00 bit_vector:=X"0000000000000000000000000000000000000000000000000000000000000000";
INITP_01 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INITP_02 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INITP_03 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INITP_04 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INITP_05 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INITP_06 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
INITP_07 : bit_vector :=
X"0000000000000000000000000000000000000000000000000000000000000000";
SRVAL : bit_vector := "000000000";
WRITE_MODE : string := "WRITE_FIRST"
);
 port (
     DO    : out STD_LOGIC_VECTOR (7 downto 0);
     DOP   : out STD_LOGIC_VECTOR (0 downto 0);
     ADDR  : in STD_LOGIC_VECTOR (10 downto 0);
     CLK   : in STD_ULOGIC;
```

```
    DI    : in STD_LOGIC_VECTOR (7 downto 0);
    DIP   : in STD_LOGIC_VECTOR (0 downto 0);
    EN    : in STD_ULOGIC;
    SSR   : in STD_ULOGIC;
    WE    : in STD_ULOGIC
    );

end component;

signal divtmp : std_logic_vector(7 downto 0);

begin

-- Component Instantiation for design element
-- Should be placed in architecture after the begin keyword
divram : RAMB16_S9

generic map (
INIT =>   "000000000",
INIT_00 => X"d0d1d3d4d5d7d8d9dbdcdddfe0e2e3e5e6e8e9ebeceef0f1f3f5f6f8fafcfeff",
INIT_01 => X"afb0b0b1b2b3b4b5b6b7b8b9babbbcbdbebfc0c1c2c3c4c5c7c8c9cacbcccecf",
INIT_02 => X"96979898999a9a9b9c9d9d9e9f9fa0a1a2a2a3a4a5a6a6a7a8a9aaaaabacadae",
INIT_03 => X"84858586868787888889898a8a8b8c8c8d8d8e8e8f909091919293939495959 6",
INIT_04 => X"767677777778787979797a7a7b7b7c7c7d7d7d7e7e7f7f808081818282838384",
INIT_05 => X"6a6b6b6b6c6c6c6d6d6d6e6e6e6f6f6f70707071717272727373737474757575",
INIT_06 => X"6161616262626263636363646464656565666666666767676868686969696a6a",
INIT_07 => X"595959595a5a5a5a5b5b5b5b5c5c5c5c5d5d5d5d5e5e5e5e5f5f5f5f60606060",
INIT_08 => X"525252535353535354545454545555555555565656565757575757585858 59",
INIT_09 => X"4c4c4d4d4d4d4d4d4e4e4e4e4e4e4f4f4f4f4f5050505050505151515151515252",
INIT_0A => X"474747484848484848494949494949494a4a4a4a4a4a4b4b4b4b4b4b4c4c4c4c",
INIT_0B => X"434343434343444444444444444545454545454545464646464646464647474747",
INIT_0C => X"3f3f3f3f3f3f404040404040404041414141414141424242424242424343",
INIT_0D => X"3b3c3c3c3c3c3c3c3c3c3c3d3d3d3d3d3d3d3d3d3e3e3e3e3e3e3e3e3e3f3f3f",
INIT_0E => X"383838393939393939393939393a3a3a3a3a3a3a3a3a3a3a3a3b3b3b3b3b3b3b3b",
INIT_0F => X"353636363636363636363636373737373737373737373737383838383838383838",
INIT_10 => X"333333333333333334343434343434343434343434353535353535353535353535",
INIT_11 => X"313131313131313131313131323232323232323232323232323333333333",
INIT_12 => X"2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f2f303030303030303030303030303031",
INIT_13 => X"2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2d2e2e2e2e2e2e2e2e2e2e2e2e2e2e",
INIT_14 => X"2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2b2c2c2c2c2c2c2c2c2c2c2c2c2c2c2c2d",
INIT_15 => X"29292929292929292a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2b2b2b2b2b",
INIT_16 => X"2828282828282828282828282828282828282829292929292929292929292929",
INIT_17 => X"2626262626262626272727272727272727272727272727272727272727272828",
INIT_18 => X"2525252525252525252525252525252626262626262626262626262626262626",
INIT_19 => X"2424242424242424242424242424242424242424242525252525252525252525",
INIT_1A => X"2223232323232323232323232323232323232323232323232323232323242424",
INIT_1B => X"2121212122222222222222222222222222222222222222222222222222222222",
INIT_1C => X"2020202020212121212121212121212121212121212121212121212121212121",
INIT_1D => X"1f1f1f1f2020202020202020202020202020202020202020202020202020202020",
INIT_1E => X"1e1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f1f",
INIT_1F => X"1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e1e",
INIT_20 => X"1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1d1e1e1e1e",
```

```
INIT_21 => X"1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1d1d1d1d1d1d1d1d1d1d1d1d1d1d",
INIT_22 => X"1b1b1b1b1b1b1b1b1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c1c",
INIT_23 => X"1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b1b",
INIT_24 => X"1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1b1b1b1b1b1b1b",
INIT_25 => X"1919191919191919a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1a1",
INIT_26 => X"1919191919191919191919191919191919191919191919191919191919191919",
INIT_27 => X"1818181818181818181818181818181819191919191919191919191919191919",
INIT_28 => X"1818181818181818181818181818181818181818181818181818181818181818",
INIT_29 => X"1717171717171717171717171717171717171717171718181818181818181818",
INIT_2A => X"1717171717171717171717171717171717171717171717171717171717171717",
INIT_2B => X"1616161616161616161616161616161616161616161616161717171717171717",
INIT_2C => X"1616161616161616161616161616161616161616161616161616161616161616",
INIT_2D => X"1515151515151515151515151515151516161616161616161616161616161616",
INIT_2E => X"1515151515151515151515151515151515151515151515151515151515151515",
INIT_2F => X"1414141414141415151515151515151515151515151515151515151515151515",
INIT_30 => X"1414141414141414141414141414141414141414141414141414141414141414",
INIT_31 => X"1414141414141414141414141414141414141414141414141414141414141414",
INIT_32 => X"1313131313131313131313131313131313131314141414141414141414141414",
INIT_33 => X"1313131313131313131313131313131313131313131313131313131313131313",
INIT_34 => X"1313131313131313131313131313131313131313131313131313131313131313",
INIT_35 => X"1212121212121212121212121212121212121212121212121213131313131313",
INIT_36 => X"1212121212121212121212121212121212121212121212121212121212121212",
INIT_37 => X"1212121212121212121212121212121212121212121212121212121212121212",
INIT_38 => X"1111111111111111111111111111111111112121212121212121212121212121",
INIT_39 => X"1111111111111111111111111111111111111111111111111111111111111111",
INIT_3A => X"1111111111111111111111111111111111111111111111111111111111111111",
INIT_3B => X"1111111111111111111111111111111111111111111111111111111111111111",
INIT_3C => X"1010101010101010101010101010101010101010101010101010101010101010",
INIT_3D => X"1010101010101010101010101010101010101010101010101010101010101010",
INIT_3E => X"1010101010101010101010101010101010101010101010101010101010101010",
INIT_3F => X"1010101010101010101010101010101010101010101010101010101010101010",
INITP_00 => X"0000000000000000000000000000000000000000000000000000000000000000",
INITP_01 => X"0000000000000000000000000000000000000000000000000000000000000000",
INITP_02 => X"0000000000000000000000000000000000000000000000000000000000000000",
INITP_03 => X"0000000000000000000000000000000000000000000000000000000000000000",
INITP_04 => X"0000000000000000000000000000000000000000000000000000000000000000",
INITP_05 => X"0000000000000000000000000000000000000000000000000000000000000000",
INITP_06 => X"0000000000000000000000000000000000000000000000000000000000000000",
INITP_07 => X"0000000000000000000000000000000000000000000000000000000000000000",
SRVAL=> "000000000",
WRITE_MODE => "WRITE_FIRST")

port map (          DO => divtmp,
      DOP => open,
      ADDR => divin(11 downto 1),
      CLK => clk,
      DI => "00000000",
      DIP => "0",
      EN => '1',
      SSR => '0',
      WE => '0'
      );
```

```
divout <= "00000000"&divtmp;

end Behavioral;
```

Appendix D

C Code for the MacGregor Model Neuron

```
// my translation of MacGregor's fortran implementation of the
// point neuron 10
// Neural and Brain Modeling - page 458
#include <math.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

void main (void)
{
        time_t startTime, endTime;
        long i,j;

        //simulation parameters
        //long step = 1; // simulation time step (ms.)
        long tstart = 1;  // start time
        long tend = 200000000; // end time of simulation
        //long neurons = 20000000;

        //input current parameters
        long Cstart = 5;  //start time for current pulse into dendtrites
        long Cend = 10000000;  //end time of current pulse into dendrites
        float Camp = 20; //amplitude of current pulse

        //constants
        float fstep = 1; // simulation time step (ms.)
        float C = 1;  //Threshold sensitivity (0-1)
        float Tth= 25;  //Time constant for accomodation (20-25 ms)
        float B= 20;  //Sensitivity to Potassium conductance (4)
        float Tgk= 5;  // Refractory time constant for Potassium (3-10 ms)
        float Th0= 10;  // Initial threshold (10-20 mV)
        float Tmem= 5;  // Membrane time constant (5-11 ms)
        float Ek= -10;  // Resting Potassium potential (-10 mV)
        float Dcth= exp(-fstep/Tth);  // Decay constant for threshold
        float Dgk= exp(-fstep/Tgk);  //Decay constant for Potassium action
        float Gtot,Dce;

        //initialize state variables
        float Vm; //membrane voltage
        float Th; //Threshold
        float S; // Spike
        float Gk; //Potassium conductance
        float Iin = 0; //Potassium conductance
```

```
//initialize state variables
float Vm_prev; //membrane voltage
float Th_prev=Th0;
float S_prev; // Spike
float Gk_prev; //Potassium conductance

//initialize state variables

//start of simulation
(void)time(&startTime);//set timer startTime

        for (i=tstart;i<tend;i++) {


                //current pulse
                if (i >= Cstart && i <= Cend)
                   { Iin = Camp;}
                else
                   {Iin= 0;}

                Gtot=1+Gk_prev;  //intermediate values used in state calcs.
                Dce=exp(-Gtot*fstep/Tmem);  // " "

                //update potassium conductance
                Gk=Gk_prev*Dgk+B*S_prev*(1-Dgk);

                //update membrane voltage
                Vm=Vm_prev*Dce+(Iin+Gk_prev*Ek)*(1-Dce)/Gtot;

                //update threshold
                Th=Th0+(Th_prev-Th0)*Dcth+C*Vm_prev*(1-Dcth);

                //update spike (works better if function of present values)
                if (Vm_prev >= Th_prev)
                   {S=1;}
                else
                   {S=0;}

                Gk_prev=Gk;
                Vm_prev=Vm;
                Th_prev=Th;
                S_prev=S;
```

```
        }
        (void)time(&endTime);//set timer endTime
        printf("Elapsed Time %d seconds.\n", endTime-startTime);

}
```

Appendix E

16-Bit Fixed Point Class for Matlab

fp16.m

```
function fp = fp16(a,pt)
%  fp16 16-bit fixed point class constructor.
%  f = double(a,b) creates a fixed point object from the double or vector value a,
%  with fixed point location pt (MSB leftmost bit)
%  will also result in a loss of information

if nargin == 0  %when no arguments return 0
  fp.b = [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
  fp.pt = 0;
  fp = class(fp,'fp16');
elseif nargin == 1  %when 1 arguments check if already of type fp16
   if isa(a,'fp16')  %if a already of type return type
      fp=a;
   else
      error('Single inputs to fp16() must be of type fp16.')
   end
elseif pt > 15 | pt < 0  % check b <=15 and >= 0
   error('Invalid fixed point location (pt must be <= 15 and >= 0)')
elseif isa(a,'double')
   % if a is double convert to best precision int possible
   % need to find unsigned binary representation
   % already handled case when magnitude too large thus sign bit will
   % be correct using the following code.  If check wasn't already made
   % positive integers > 2^(# pre-binary point bits)-1 would be represented
   % as negative numbers

   [row,col]=size(a);  %get dimension of a matrix

   %do all error checking here
   if floor(a(1:1:row,1:1:col)) > (2^(16-pt-1)-1)
      error('A number magnitude too large for fixed point representation.  Positive
Overflow.')
   end

   if a(1:1:row,1:1:col) < -(2^(16-pt-1))
        error('A number magnitude too large for fixed point representation. ...
        NegativeOverflow.')
   end
```

```
    %create matrix of unsigned 16 bit arrays
    for k=1:1:row
       for j=1:1:col
          tmp2=abs(a(k,j));
          for i=1:1:16
             tmp1=2^(16-pt-i);
             if tmp1 <= tmp2
                tmp2=tmp2-tmp1;
                fp(k,j).b(i)=1;
             else
                fp(k,j).b(i)=0;
             end
             fp(k,j).pt=pt;
          end
       end
    end


    % finish creating fp16 type
    fp = class(fp,'fp16');

   % now fix sign bit
   % probably best just to use a loop for this
    for i=1:1:row
       for j=1:1:col
          if sign(a(i,j))<0
             fp(i,j)=-fp(i,j);  %put in twos complement form
          end
       end
    end

elseif isa(a,'char')
    if length(a)==16
       %convert string to fp16 type
       for i=1:16
          digit=str2num(a(1,i));
          if digit==0 | digit == 1
             fp.b(i)=digit;
          else
             error('Char Array typecast had a non binary value')
          end
       end
       %finish creating type
```

```
      fp.pt=pt;
      fp = class(fp,'fp16');
    else
      error('Typecast failed (Character Arrays must be size 16)')
    end
else
  error('Typecast failed (Make sure inputs are doubles)')
end
```

double.m

```
function d = double(fp)
% converts a fp16 type to a double
% if fp is a fp16 matrix then result is a double matrix
% only works with 2 dimensional matrices now

[row,col]=size(fp);  %get dimensions of fp16 matrix
change_sign=0;

for i=1:1:row
   for j=1:1:col
      %Check whether positive or negative
      if fp(i,j).b(1) == 1
         change_sign=1;
         fp(i,j)=-fp(i,j);  % use 2's complement operator to find unsigned value
      end

      %find double equivalent of unsigned binary
      if change_sign==1
         d(i,j)=0-[fp(i,j).b]*[2.^((15-fp(i,j).pt):-1:(-fp(i,j).pt))'];
      else
         d(i,j)=[fp(i,j).b]*[2.^((15-fp(i,j).pt):-1:(-fp(i,j).pt))'];
      end
      change_sign=0;
   end
end
```

display.m

```
function display(fp)
% Displays fixed point types in the command window
% For now just handles printing of 2-D matrices

if ndims(fp) <= 2
   [m,n]=size(fp);
   disp(' ');
   disp([inputname(1) ' = ']);
   disp(' ');

   for i=1:1:m  %rows
      str=' ';
      for j=1:1:n  %columns
         str=[str sprintf('%s (%s)   ',...
            num2str(double(fp(i,j))),fp2str(fp(i,j)))];
      end
      disp(str)
      disp(' ');
   end
else
   disp(' ');
   disp([inputname(1) ' = ']);
   disp('Not going to mult > 2d matrices (write your own function)')
   disp(' ');
end
```

eq.m

```
function bool = eq(fp1,fp2)
% tests equality of two fp16 datatypes
% returns 1 if equal else 0
% for now only allow same binary pt else raise error

bool=1;  %initialize to true
for i=1:1:16
   if fp1.b(i) ~= fp2.b(i)
      bool=0;
```

```
   end
end
if fp1.pt ~= fp2.pt
   error('Equality test failed. (Different binary point locations)');
end
```

ge.m

```
function bool = ge(fp1,fp2)
% tests if fp1 >= fp2
% returns 1 if true,  0 if false

bool = double(fp1)>=double(fp2);
```

gt.m

```
function bool = gt(fp1,fp2)
% tests fp1 > fp2 of two fp16 datatypes
% returns 1 if true,  0 if false

bool = double(fp1)>double(fp2);
```

le.m

```
function bool = le(fp1,fp2)
% tests fp1 <= fp2 of two fp16 datatypes
% returns 1 if true,  0 if false

bool = double(fp1)<=double(fp2);
```

lt.m

```
function bool = lt(fp1,fp2)
% tests fp1 < fp2 of two fp16 datatypes
```

```
% returns 1 if true,  0 if false

bool = double(fp1)<double(fp2);
```

minus.m

```
function fpmin = minus(fp1,fp2)
% subtracts two fp16 data types fp1 - fp2
% for now just allow binary points that are equal
% raises error if overflow condition met

fpmin=fp16(double(fp1)-double(fp2),fp1(1,1).pt);
```

mtimes.m

```
function prod = mtimes(fp1,fp2)
% matrix multiply
% returns product of fp1 and fp2
% error occurs when overflow or underflow condition
% only allow 2d matrix mult now
% assumes same binary point position for all elements
% this would have to be checked in a loop

prod=fp16(double(fp1)*double(fp2),fp1(1,1).pt);
```

ne.m

```
function bool = ne(fp1,fp2)
% tests fp1 < fp2 of two fp16 datatypes
% for now only allow same binary pt else not equal
% returns 1 if true,  0 if false

bool = not(fp1==fp2);
```

plus.m

```
function fpsum = plus(fp1,fp2)
% adds two fp16 data types
% for now just allow binary points that are equal

fpsum=fp16(double(fp1)+double(fp2),fp1(1,1).pt);
```

times.m

```
function prod = times(fp1,fp2)
% element by element matrix multiply
% returns element-wise product of fp1 and fp2
% error occurs when overflow or underflow condition
% only allow 2d matrix mult now
% assumes same binary point position for all elements
% this would have to be checked in a loop


prod=fp16(double(fp1).*double(fp2),fp1(1,1).pt);
```

uminus.m

```
function fpcom = uminus(fp)
%takes the 2's complement of the fp16 type
% now need to invert, add one
% which can be done with the following process:
% does not raise error if fp=-128

one_found=0;
for i=16:-1:1 % looking LSB to MSB
  if (one_found == 0)
    if (fp.b(i)==1) % find first 1
      one_found=1; % once 1 found then all subsequent bits toggle
    end
    fpcom.b(i)=fp.b(i);
  else
    if fp.b(i)==0
```

```
        fpcom.b(i)=1;
      else
        fpcom.b(i)=0;
      end
    end
end
% finish creating fp16 type
fpcom.pt=fp.pt;
fpcom = class(fpcom,'fp16');
```